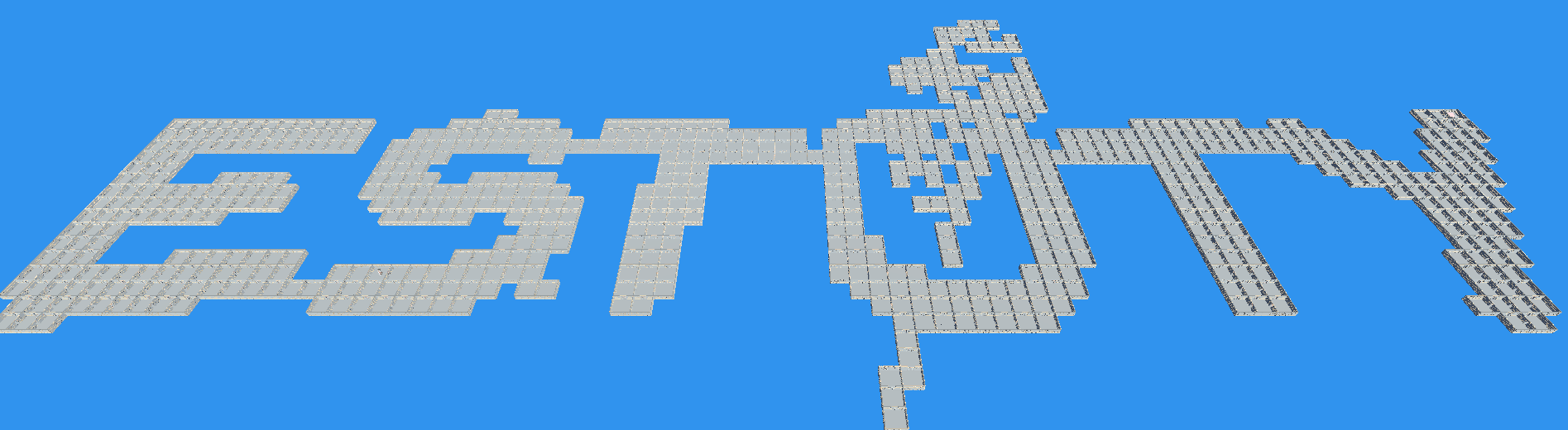
Maze Escape



[**Overview 2**](#_qlinrp33f7eg)

[**Things used in project 2**](#_gybvqcv0abh1)

[**Maze generation 3**](#_6olpemwwqt3m)

[**AI 10**](#_xxnzjvxlq52m)

# Overview

The game features a dynamically generated maze with both random and procedural elements, ensuring each playthrough presents a unique challenge. Every maze includes a distinct starting point and a designated finish. Importantly, all generated mazes are designed to be navigable.

Upon commencement, the player is positioned at the starting point. Victory is achieved by successfully navigating the maze and reaching the designated finish.

Within the maze, formidable enemy seekers pose a threat to the player. These enemies possess designated damage areas, and any player within such an area sustains damage. The player's health points (HP) are crucial, and the game concludes if the player's HP falls below zero.

Enemy seekers follow patrol paths until they detect the player. If an enemy spots the player, it will pursue the player, deviating from its patrol path. In the event that the player successfully evades the enemy and is no longer in sight, the enemy will backtrack to the player's last known location. Subsequently, if the player remains undetected, the enemy will revert to its original patrol mode.

For a comprehensive demonstration of the game's functionalities, three distinct scenes have been meticulously crafted.

These scenes can be accessed through the following path: **Assets -> MazeEscape -> Scenes**.

* **SquareMaze** - This scene showcases square maze generation.
* **RandomMaze** - Here, the game exhibits maze generation based on random directions.
* **ImageMaze** - This scene highlights the creation of a maze through image input.

# Project Components

Various elements were employed in the development of this project to enhance organization, efficiency, and overall functionality. Here is an overview of the key components utilized:

* Namespaces
  + Purpose: Used for organizing and categorizing code.
  + Benefit: Enhances code readability and maintainability by structuring code into logical groupings.
* Assemblies
  + Purpose: Employed as an effective method for separating different systems and mitigating cyclic dependencies.
  + Benefit: Facilitates a modular and scalable project structure, contributing to better code management.
* Disabled Domain Reloading
  + Purpose: Implemented for expedited play mode loading during development.
  + Benefit: Accelerates the testing and iteration process by reducing the time required for reloading the domain in play mode.
* Scriptable Objects
  + Purpose: Utilized as a versatile solution for storing properties.
  + Benefit: Offers a user-friendly means of adjusting settings, allowing the creation of multiple options, such as distinct generation processes for various levels.
* Zenject (Dependency Injection)
  + Purpose: Implemented for efficient dependency management.
  + Benefit: Enhances code maintainability and scalability by providing a clean and organized approach to handling dependencies.
* LeanTween (UI Animations)
  + Purpose: Employed for creating smooth and visually appealing UI animations.
  + Benefit: Enhances the overall user experience by incorporating polished and fluid animations into the game's interface.
* "Flood Fill" Algorithm
  + Purpose
    - Applied for maze validation to make sure that all cells can be accessed via entrances
    - For finish cell generation to find furthest cell
  + Benefit: Ensures the integrity and navigability of generated mazes by validating their structure. For detailed information on the algorithm, refer to [Flood Fill Algorithm](https://en.wikipedia.org/wiki/Flood_fill).
* AI Agent Navigation
  + Purpose: Integrated to enable intelligent navigation for AI agents within the game.
  + Benefit: Enhances the realism and challenge of the game by providing AI entities with dynamic and adaptive movement capabilities.

These components collectively contribute to the robust and efficient development of the project, ensuring a well-organized codebase.

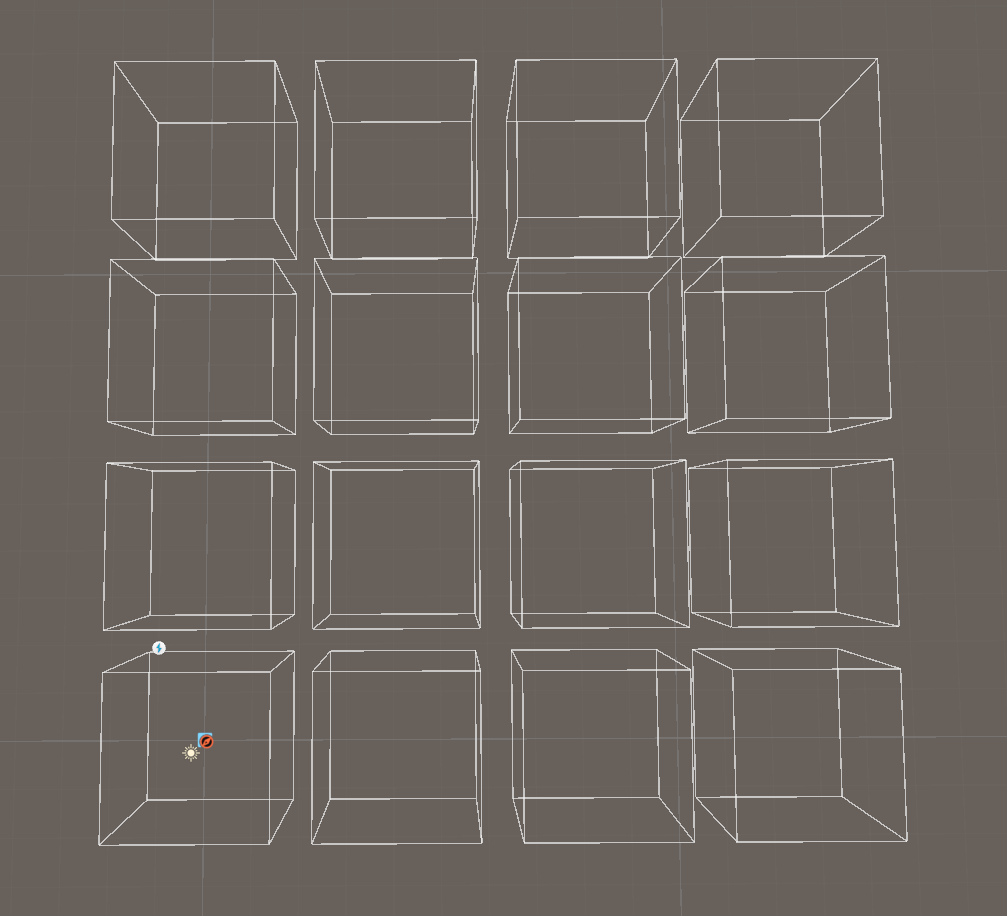
# Maze generation

In this work, maze generation is organized using a specific generation scheme. Each of these elements turns in sequence, responsible for generating specific sections of the maze and transitioning seamlessly into the next section.

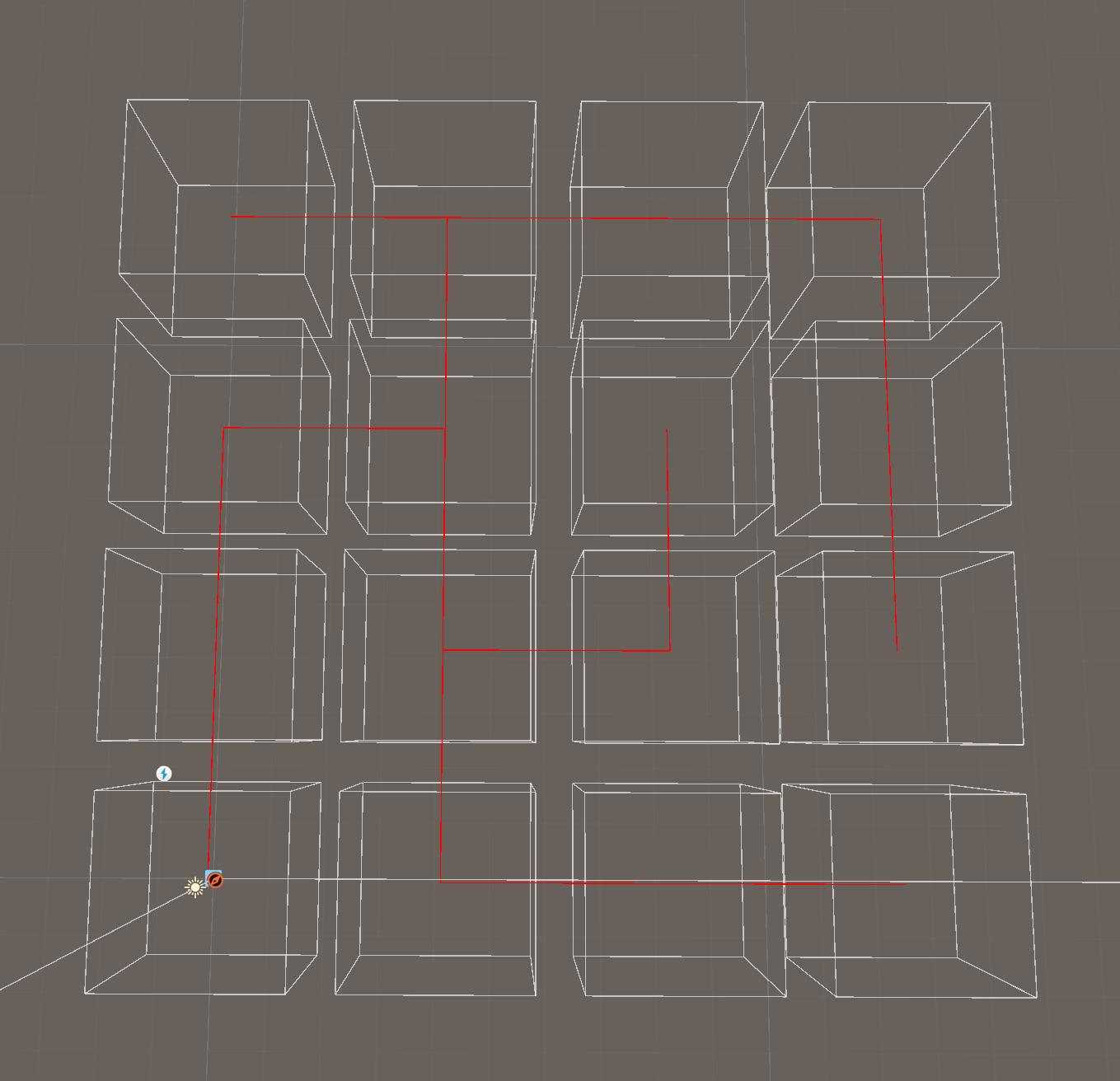
This structured approach ensures that the maze is strategically designed, and that each event contributes to the overall design. By dividing generation into stages, the system is modular and adaptive, allowing for easy modification and expansion in the future. This simplified workflow contributes to efficiency and clarity in the maze creation process.

Full generation flow:

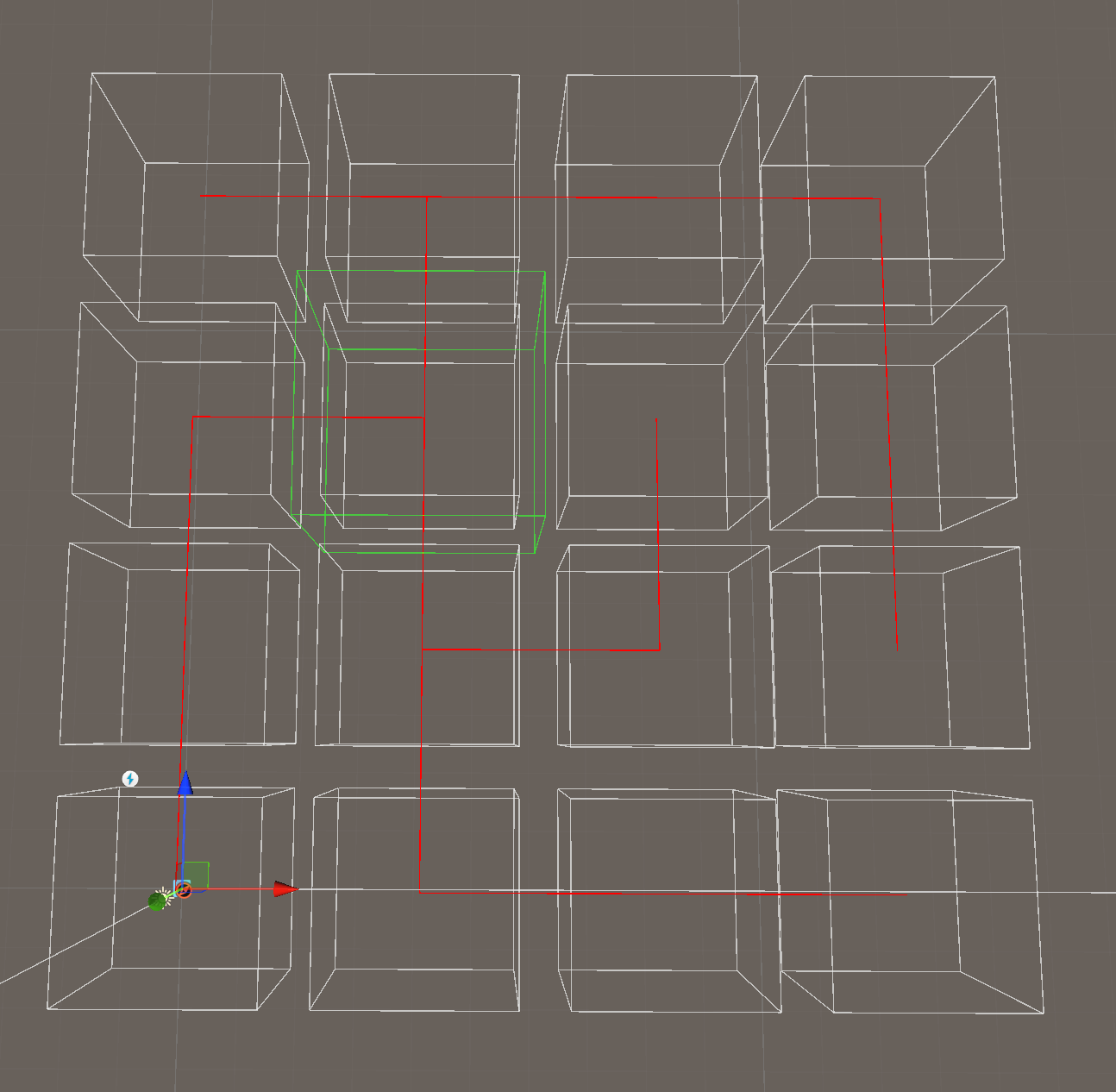
1. **Maze cell generation**. Set number of maze cells are spawned and are assigned their coordinates. (Example 4x4 square maze, cells are white squares).



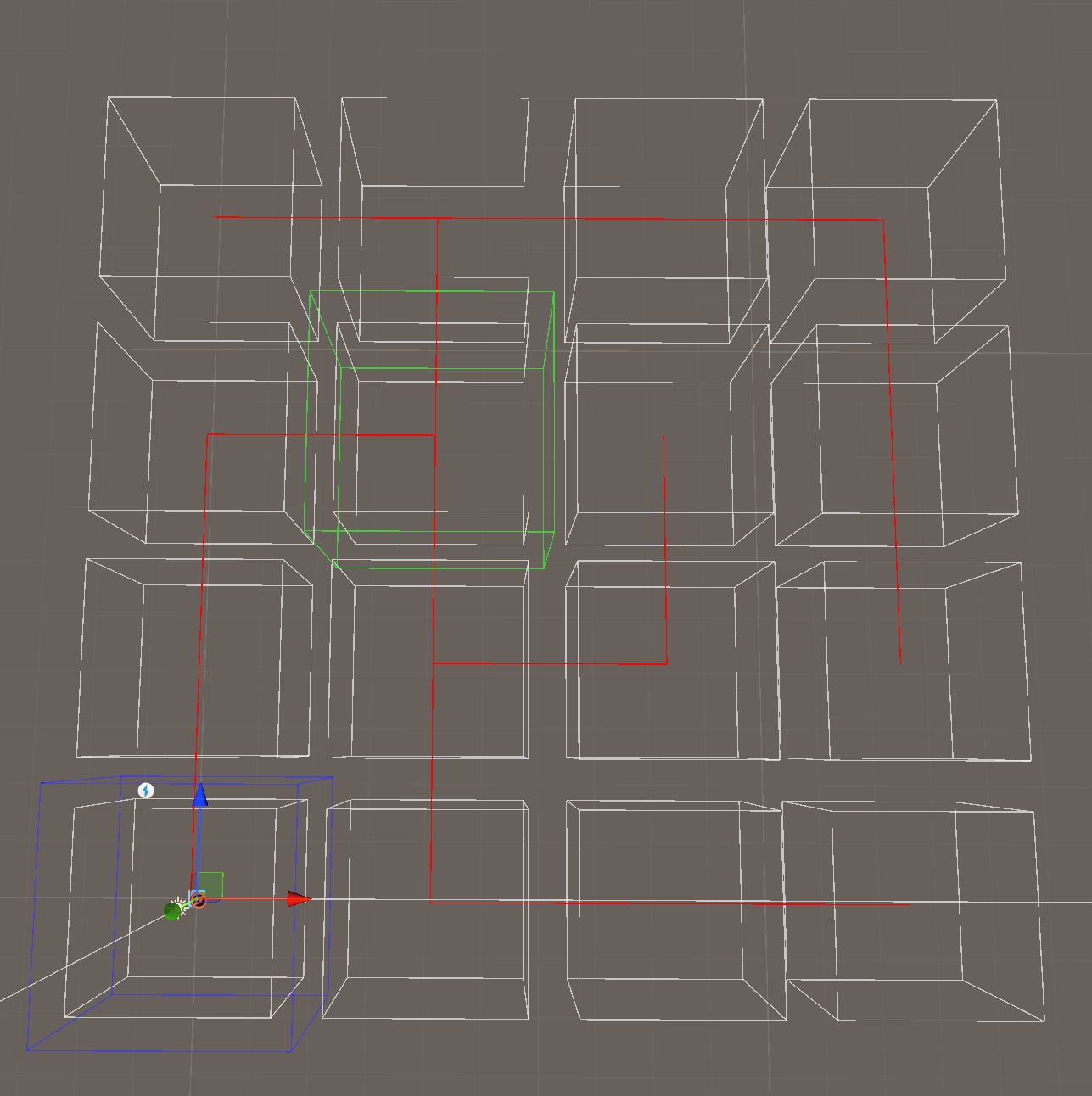
1. **Passage generation**. Passages are added for all cells to all adjustent cells. After that passages are randomly deleted. Randomization is partly decided by weights set by the user. After each passage deletion maze is validated using the "Flood Fill" algorithm the maze is still fully passable. If the maze becomes impassable, last passage deletion is reset. For example if a cell has 4 passages and randomization decides that it should have only 1 passage, random passages will be deleted one by one until 1 is left, except if by deleting some passage maze becomes impassable. (passages are red lines).



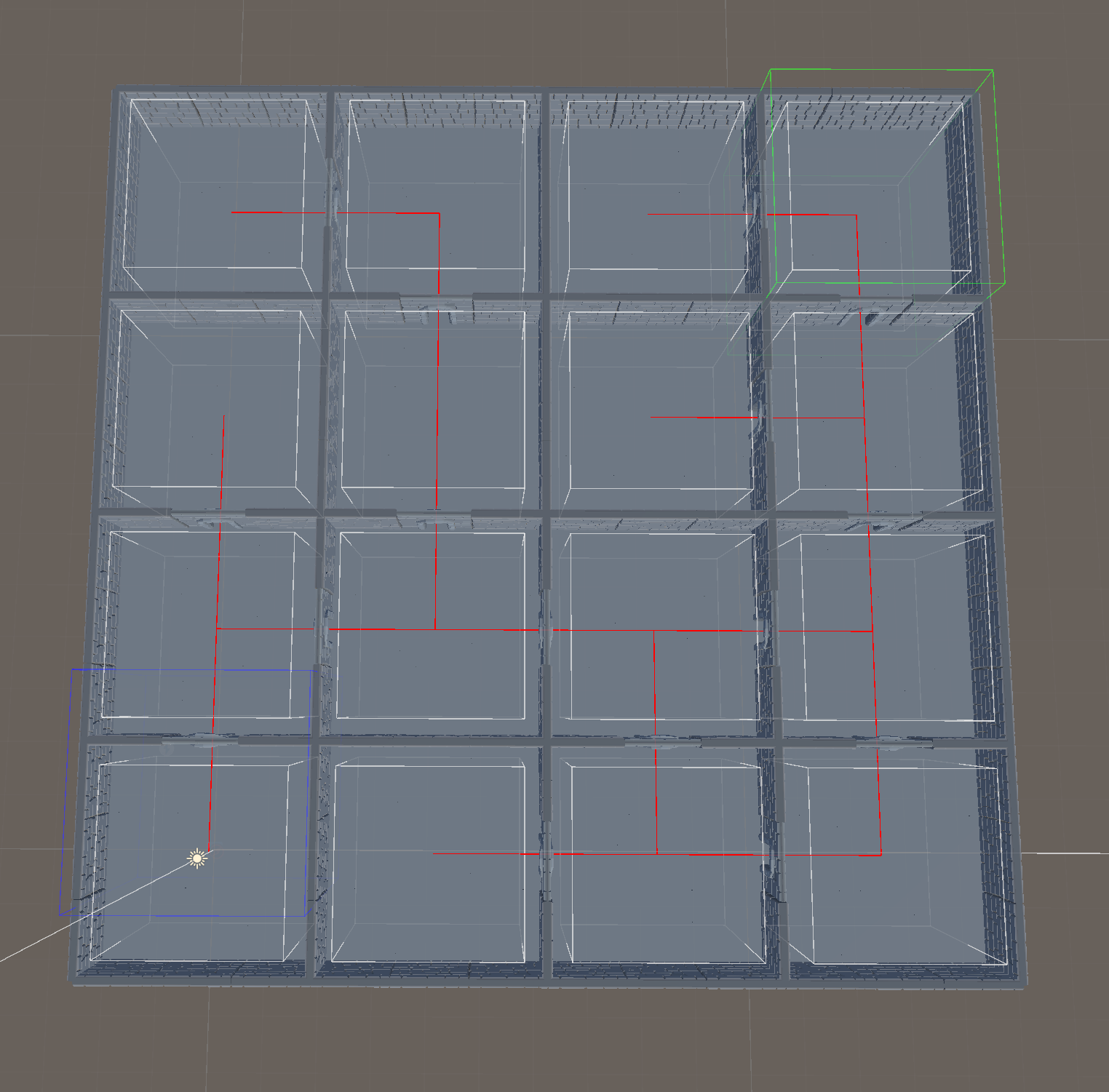
1. **Start generation.** One of all cells is randomly picked to be the starting cell (green square).



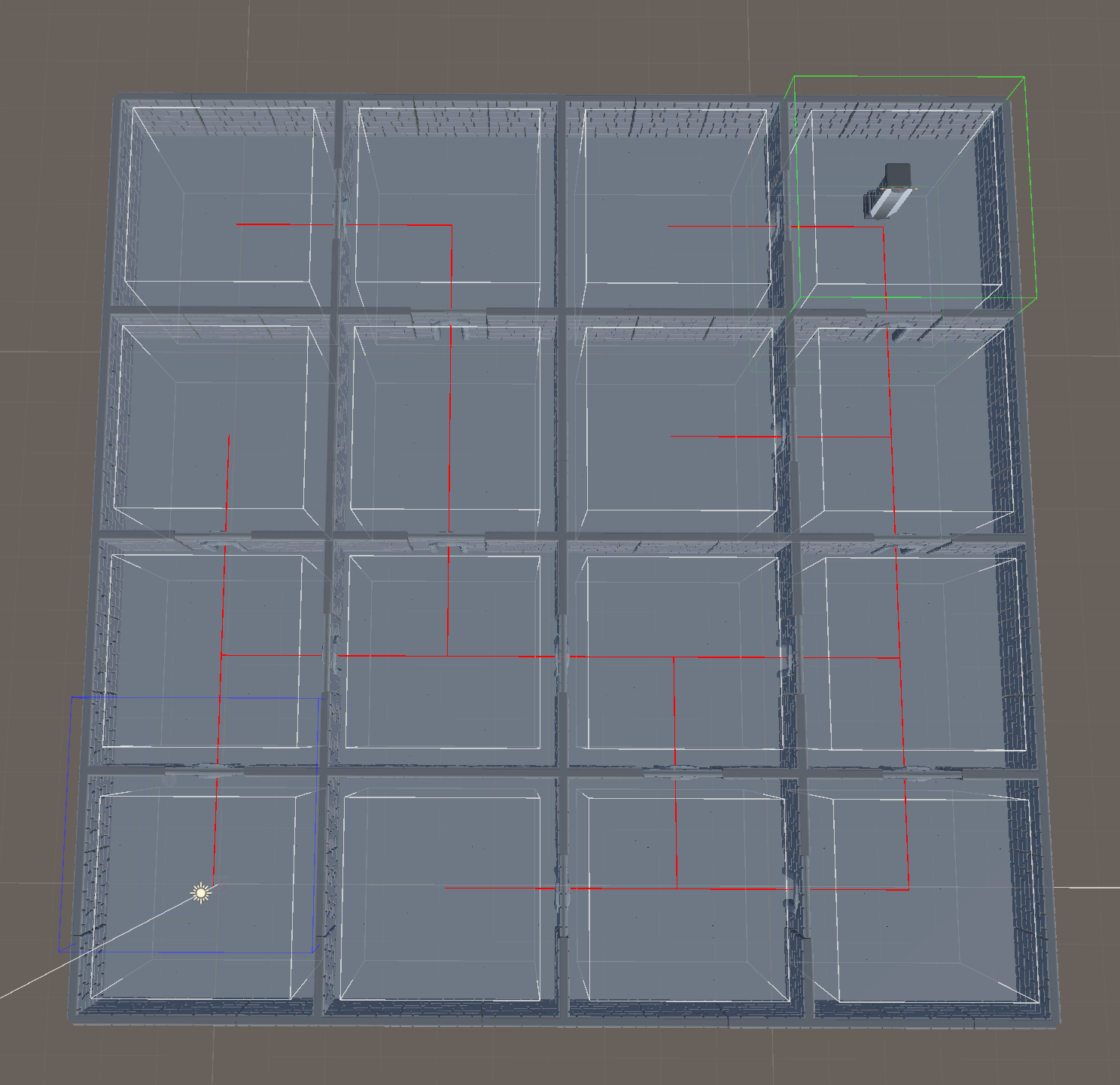
1. **Finish generation**. Finish cell is determined using the "Flood Fill" algorithm starting from the start cell to find the furthest cell (blue square).



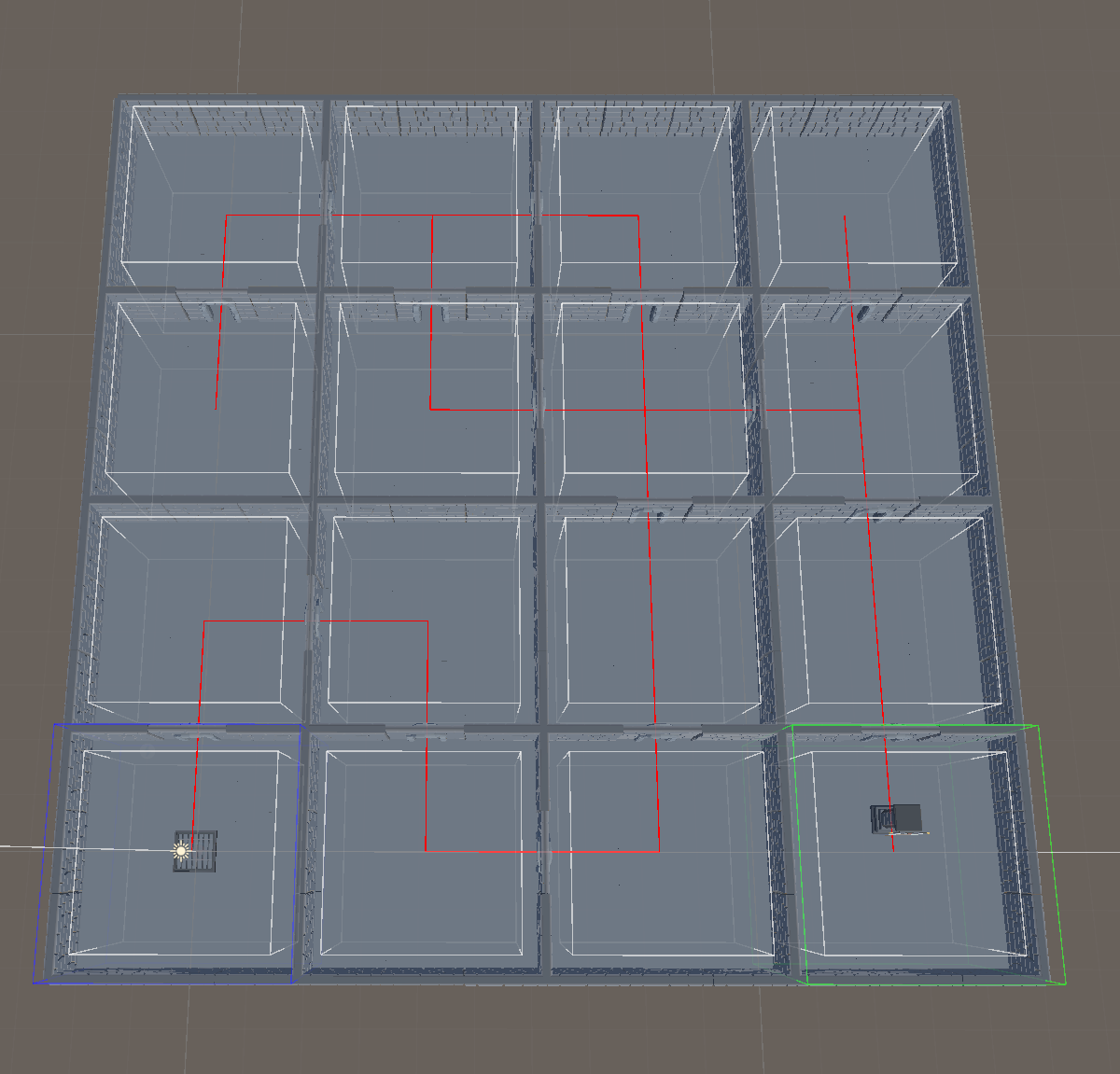
1. **Cell visuals generation**. Visuals are spawned from available prefabs to match entrances.



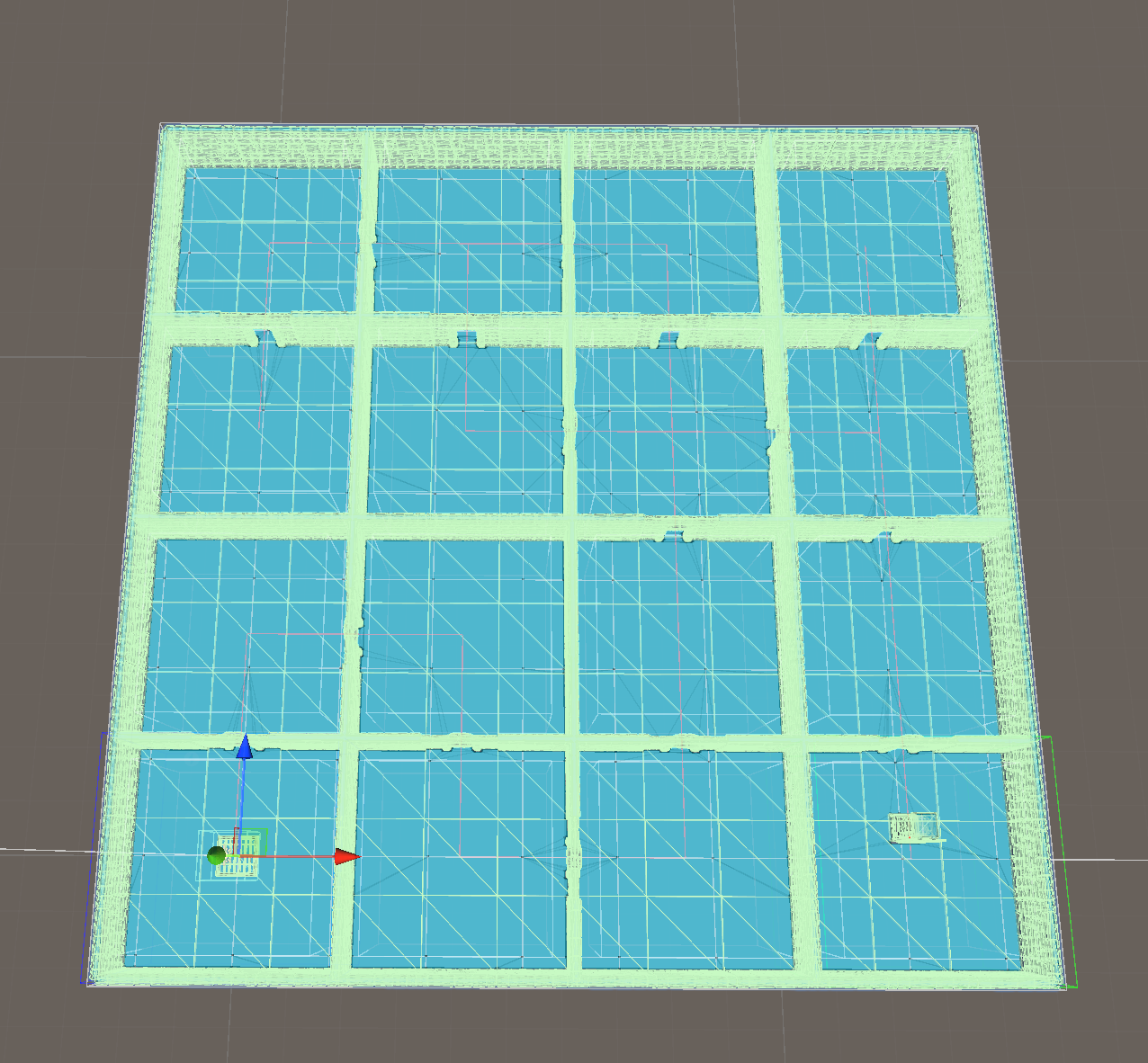
1. **Start visual generation**. Start visual prefab are spawned in the start cell.



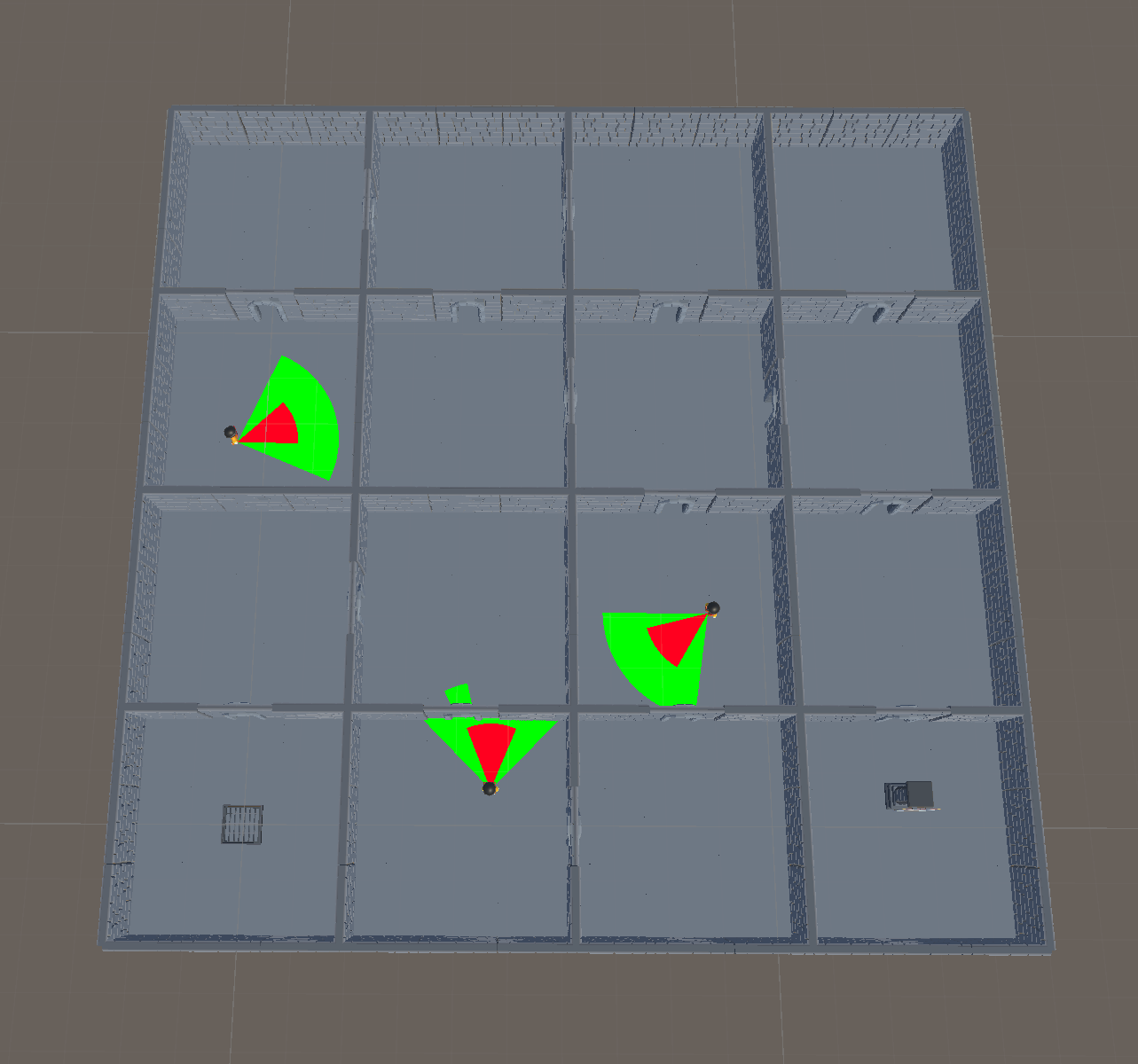
1. **Finish visual generation**. Finish visual prefab are spawned in the finish cell.



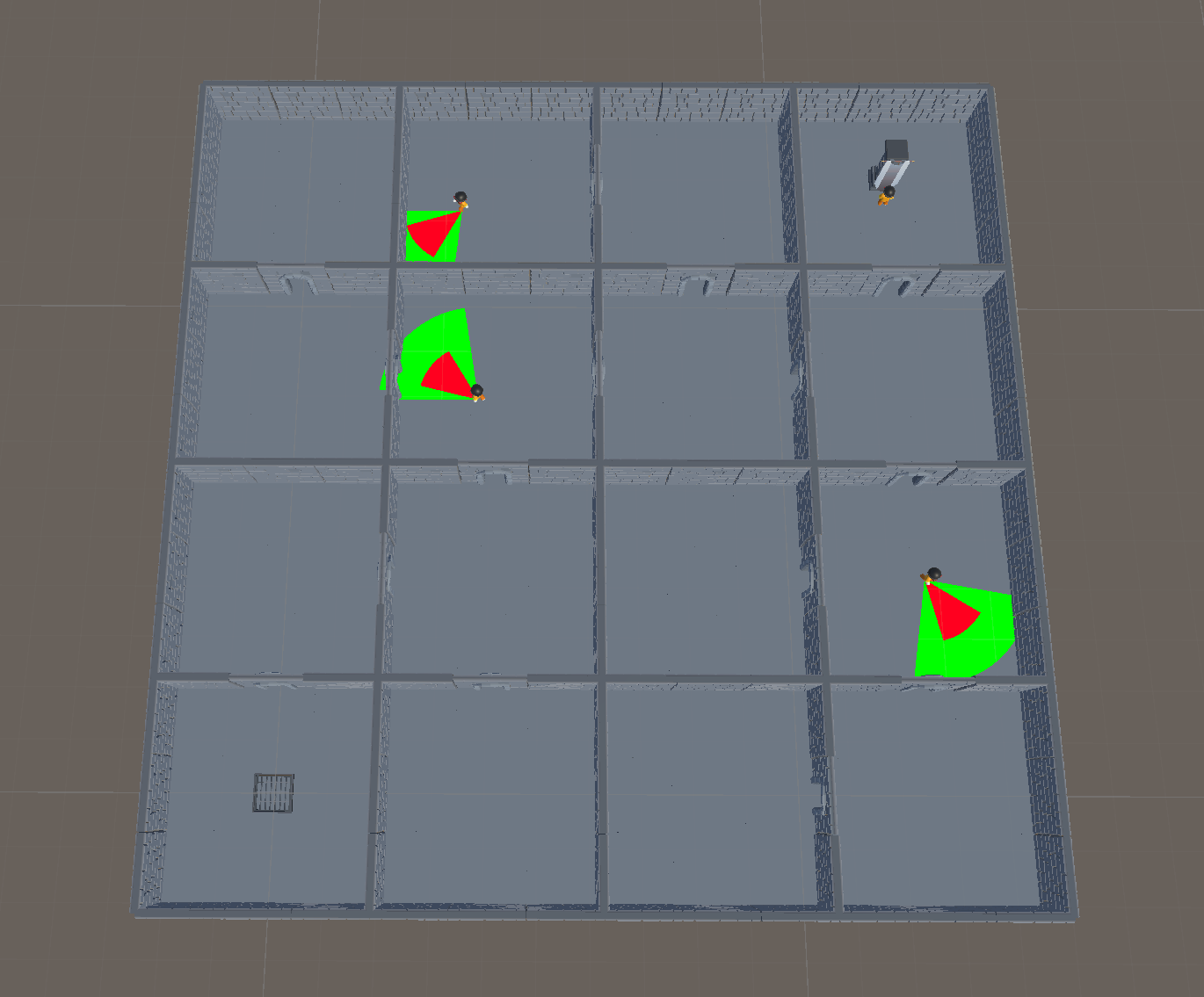
1. **Nav mesh generation**. Nav mesh generation for AI (blue ground).



1. **Enemy generation**. Enemies are spawned in random cells except the start cell.



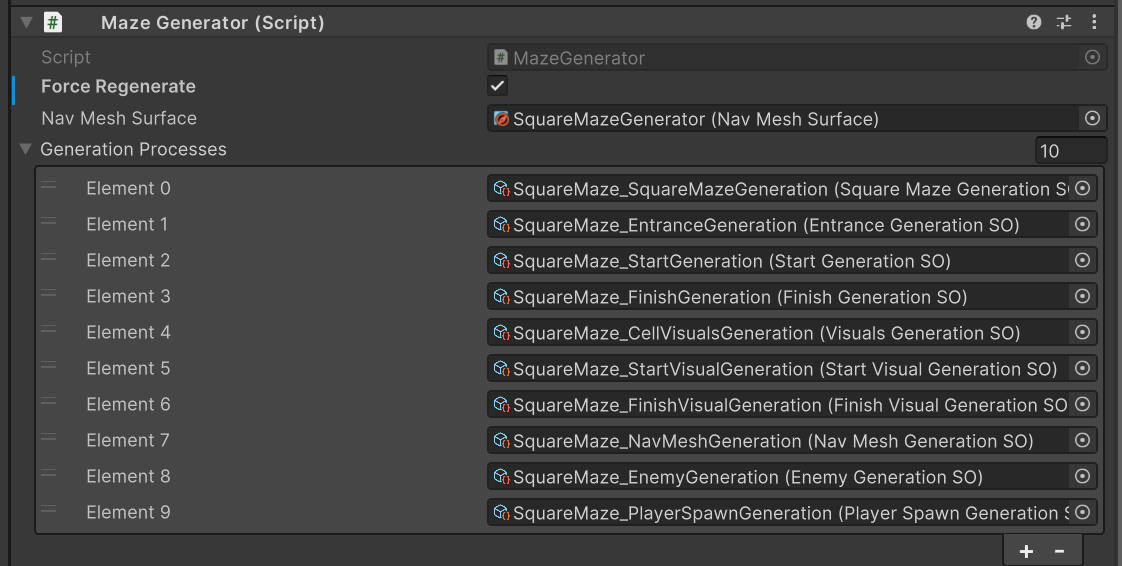
1. **Player spawn generation**. Player prefab is spawned and the game is started.



One of the key design principles underlying this project is the creation of a highly adaptable and modular maze generation system. All the aforementioned processes can be effortlessly interchanged with different ones, removed, or augmented by adding more processes. This flexibility empowers developers to tailor the maze generation to suit specific design requirements.

Notably, all generation variables are conveniently exposed in the inspector, allowing for seamless adjustments within the scriptable objects associated with each process. This intentional design choice simplifies the modification of generation parameters, providing developers with a user-friendly interface to customize the maze generation process.

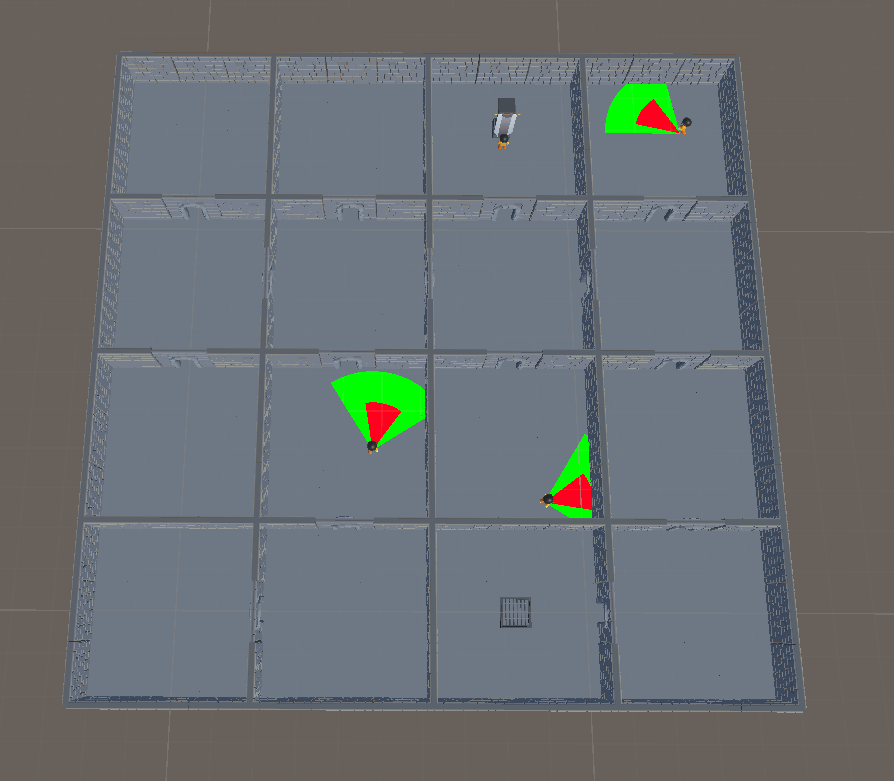
In essence, the project is built upon the concept of a modular system, fostering easy modification and expansion. This approach not only enhances the development workflow but also encourages experimentation and innovation in crafting diverse and engaging maze layouts.



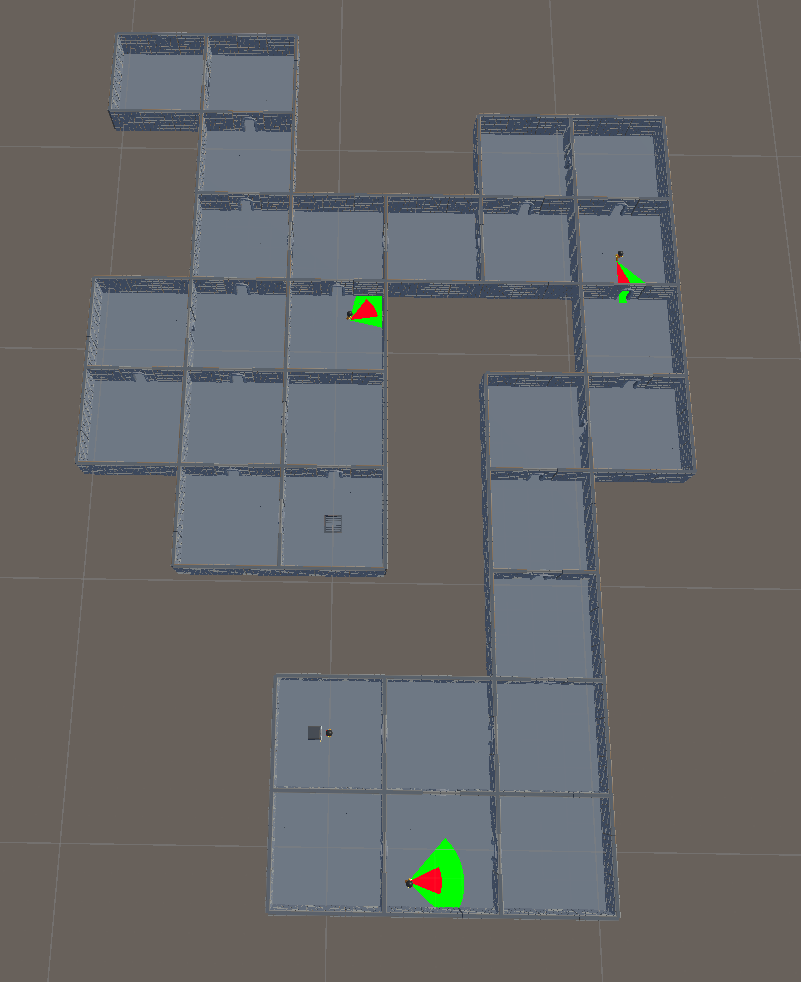
In the presented example, the cell generation process is exemplified by the creation of a square maze. However, it's important to note that this process is designed to be easily interchangeable with alternative methods. The project includes several pre-built cell generation processes, offering a range of options to suit specific requirements.

Included Cell Generation Processes:

* SquareMazeGeneration
  + Example: Generates a traditional square maze.



* RandomCellGeneration
  + Functionality: Generates cells in random directions, introducing an element of unpredictability to the maze layout.



* ImageGeneration
  + Functionality: Utilizes an provided image to generate a maze, offering a unique and visually influenced approach to maze creation.



To manage inter-process dependencies within the project, Zenject dependency injection is employed. This strategic implementation ensures a robust and organized system, preventing racing conditions and laying the groundwork for potential future enhancements such as asynchronous process initiation for improved efficiency.

Currently, the project incorporates four interfaces, each serving a specific purpose:

* ICellGeneration
  + Provides access to generated cells. Entrance, start cell and finish cell generations rely on this interface.
* IEntranceGeneration
  + Provides access to passages between cells. Finish cell generation relies on this interface.
* IStartCell
  + Provides access to the start cell. Finish cell generation relies on this interface.
* IFinishCell
  + Provides access to the finish cell.

By using Zenject to inject objects with these required interfaces, the project ensures a seamless and controlled flow of dependencies, enhancing the stability and reliability of the maze generation processes. This architecture not only mitigates potential issues arising from dependencies but also establishes a foundation for future optimizations, such as asynchronous process initiation, to further optimize the overall flow of processes.

# AI

The game introduces enemies strategically placed within mazes to patrol and engage players. Similar to other processes, the spawning of enemies is configurable, allowing for dynamic adjustments. Enemies in the game exhibit three distinct states, each influencing their behavior:

* Patrolling State (default state)
  + Trigger - upon spawning, enemies enter the patrolling state, navigating around their assigned cell. This state can be altered if the enemy transitions to another cell after pursuing a player.
* Alert State
  + Trigger - when an enemy spots the player, it enters the alert state.
  + Behavior - In this state, the enemy ceases patrolling and initiates pursuit, transitioning to the chasing state. The enemy's view cone changes from green to yellow when the player is detected.
* Chasing State
  + Trigger - when the player is seen by the enemy.
  + Behavior - The enemy actively follows the player during this state.
* Confused State
  + Trigger - when the player successfully evades the enemy and the last seen location is reached.
  + Behavior - the enemy scans the surroundings for the player. When the player is not found, the enemy reverts to the patrolling state.

Enemy Action Areas:

Enemies have two distinctive action areas that play a crucial role in their interaction with players:

* View Cone
  + Default Appearance: Represented as a green or yellow cone.
  + When the player is within the view cone, it transitions to yellow, signaling that the player has been detected.
* Damage Cone
  + Appearance: Depicted as a red cone.
  + Effect: Inflicts damage to the player if they enter this area.

