Throughout this project, I honestly learned a lot. During my game development journey, I didn't really work on implementing algorithms a lot, since most of our coursework focused on specific assignments we had to complete. So, I was genuinely excited to experience procedural maze generation for the first time. It was very educational, and I enjoyed the process. I hope you like it happy reading!

DTT UNITY ASSESSMENT

Maze Procedural Generation

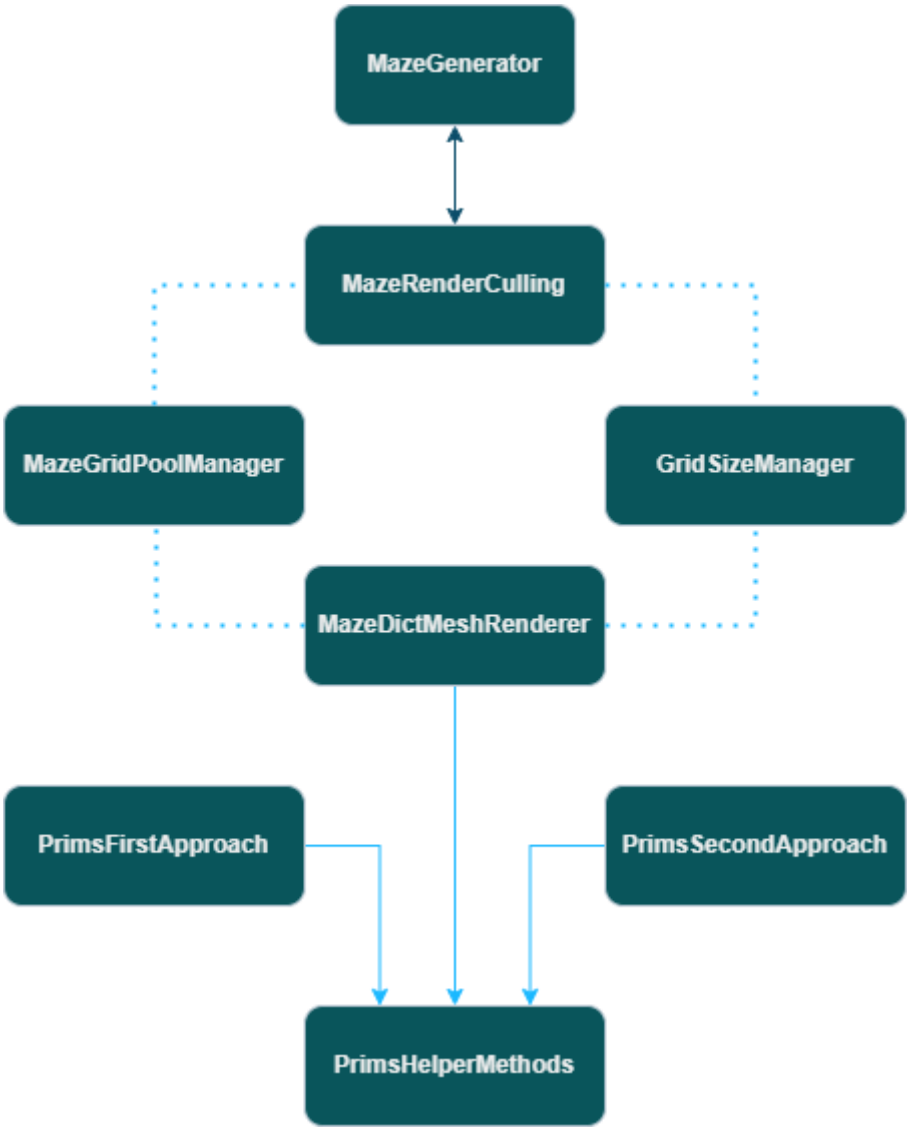
Abdulrahman Bani Almarjeh

Inhoudsopgave

Introduction

This document presents the high level architecture of the Maze Generation system implemented in Unity. It covers the core components (classes), their responsibilities, and interactions. The central algorithm is Prim’s Maze Generation, supplemented by performance optimizations using Unity's threading and batching features.

Quick peek of the UML design before coding



Architecture Overview

The core maze system consists of eight primary classes:

1. **MazeGenerator**: Orchestrates grid creation via three modes (one-off, batched, pre-instantiated).
2. **MazeGridPoolManager**: Manages enabling/disabling of maze cell GameObjects using the dictionary from MazeGenerator.
3. **MazeRenderCulling**: Performs distance-based culling of renderers and colliders to improve runtime performance.
4. **MazeDictMeshRenderer**: Gradually toggles renderer components in batches, firing an event upon completion.
5. **GridSizeManager**: Uses Unity's Job System and Burst compiler to resize and reposition cells in parallel.
6. **PrimsHelperMethods**: Provides utility functions for Prim's algorithm: neighbor detection, carving, and animations.
7. **PrimsFirstApproach**: Visual, coroutine-driven implementation of Prim's algorithm with step-by-step animation.
8. **PrimsSecondApproach**: Automated, frame-by-frame Prim's carving without explicit visualization delays.

These classes collaborate through shared data structures—primarily a Dictionary<Vector2Int, GameObject> mapping grid coordinates to cell objects, and a byte[,] map for the internal maze representation.

Component Responsibilities

1. **MazeGenerator**: Entry point for grid creation. Ensures odd dimensions, clears old data, and branches to the chosen generation mode.
2. **MazeGridPoolManager**: Acts as a lightweight pool manager that simply toggles GameObject activation; avoids costly instantiations/destructions at runtime.
3. **MazeRenderCulling**: Performs regular culling based on the player's proximity to reduce rendering and physics overhead.
4. **MazeDictMeshRenderer**: Provides batched visual toggling (e.g., fade-in/fade-out) and signals when rendering passes complete.
5. **GridSizeManager**: Offloads heavy resizing/repositioning work to Unity's multithreaded job system, then applies final adjustments on the main thread.
6. **PrimsHelperMethods**: Encapsulates algorithmic primitives neighborhood logic, carving rules, and visual helper routines (coloring, animations).
7. **PrimsFirstApproach**: Coroutine-driven, step-by-step visualization of Prim's algorithm.
8. **PrimsSecondApproach**: Automated, fast execution of Prim's algorithm in a deterministic loop, suitable for real-time gameplay.

Data Flow

1. **Initialization**: MazeGenerator.Generate() populates cellsByLocation and map via one of three strategies.
2. **Pooling**: MazeGridPoolManager receives dictionary reference and toggles cell GameObjects on/off.
3. **Rendering**: MazeDictMeshRenderer takes the pooled cells and reveals them in batches; MazeRenderCulling continues runtime culling as the player moves.
4. **Sizing**: GridSizeManager resizes and repositions cells based on cubeSize and spacing parameters.
5. **Maze Carving**: PrimsHelperMethods drives the carving logic PrimsFirstApproach or PrimsSecondApproach orchestrates usage and visualization.
6. **Finalization**: PrimsHelperMethods.PullDownPath animates the dug path, giving a satisfying drop effect.

Conclusion

This architecture cleanly separates concerns grid generation, pooling, rendering, culling, sizing, and algorithmic carving into focused components. The use of Unity specific optimizations (Jobs, Burst, batching, coroutines) ensures both performance and clarity, while helper utilities maintain code reuse and testability.