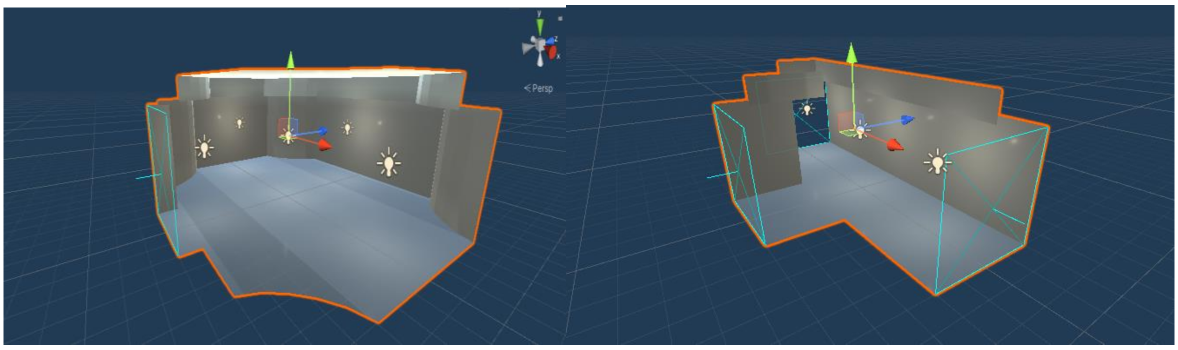
Implementation

Connectors

The prefabs include one or more connectors and any other elements the level designer desires to spawn with it. These prefabs are attached to each other by connectors during the generation process.

Connectors are the components that determine where two level pieces will join. Each connector has a pin which determines what other connectors it can connect with.

Connectors can only form pairs with each other if they have an equal number of pins When two connectors are matched, the prefab being evaluated will move so its connector and the connector of the selected placed prefab are facing each other.



Code to achieve the connectors:

[ConnectorsCode]

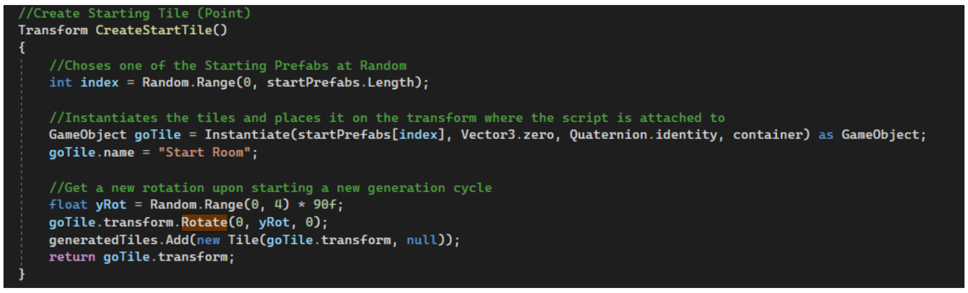
Generation Manager and Parameters

Parameter	Description
CreateStartTile()	Transform. Creates a random starting point for the Dungeon.
CreateTile()	Transform. Creates tiles (corridors, halls, etc) after the Starting Room is spawned.
GetRandomConnector()	Transform. Gets a random connector from the generated tile.
ConnectTiles()	Gets the random connector from the tile and attaches it to another.
isConnected	Bool. Checks if connectors pin is already attached to another.
Start()	Start dungeon build
Update()	Create a new dungeon
DungeonBuild	IEnumerator. Builds the dungeon, using and putting together all the information generated by the other functions.

DungeonState{}	Enum of all the DungeonBuild states
SpawnDoors()	Gets all connectos

Starting Point

The algorithm begins by selecting and placing the starting prefab/tile of the level in the game world.

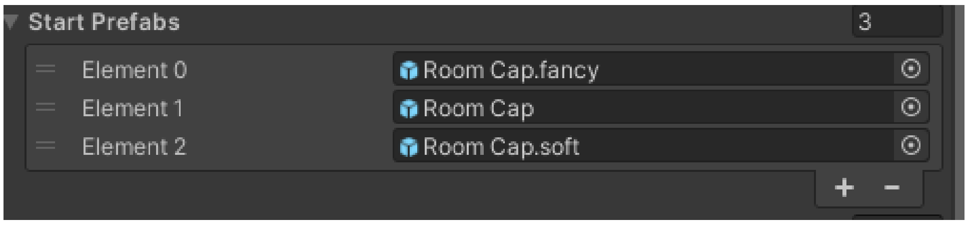


```
//Create Starting Tile (Point)
Transform CreateStartTile()
{
    //Choses one of the Starting Prefabs at Random
    int index = Random.Range(0, startPrefabs.Length);

    //Instantiates the tiles and places it on the transform where the script is attached to
    GameObject goTile = Instantiate(startPrefabs[index], Vector3.zero, Quaternion.identity, container) as GameObject;
    goTile.name = "Start Room";

    //Get a new rotation upon starting a new generation cycle
    float yRot = Random.Range(0, 4) * 90f;
    goTile.transform.Rotate(0, yRot, 0);
    generatedTiles.Add(new Tile(goTile.transform, null));
    return goTile.transform;
}
```

On the code snippet above a function is called to create the starting point, an index is made to capture the number of prefabs who are part of the starting point category.

A prefab is then picked at random, instantiated and placed on the world.

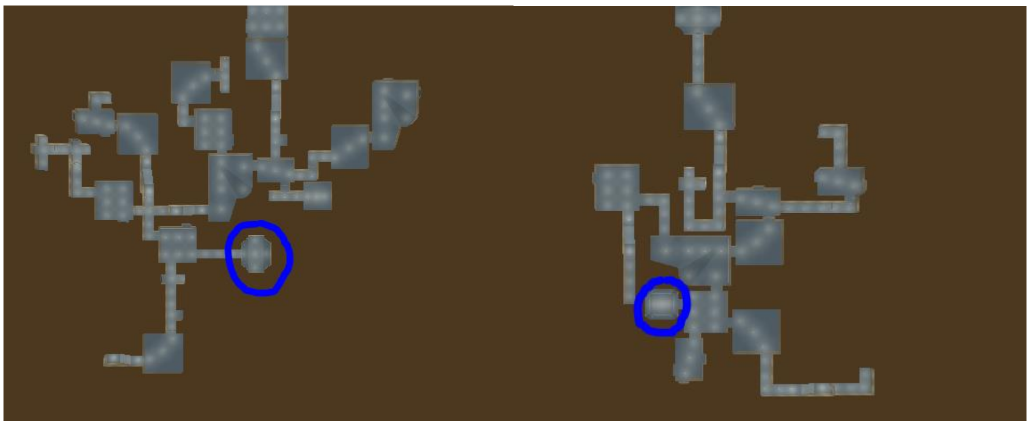
Start Prefab List



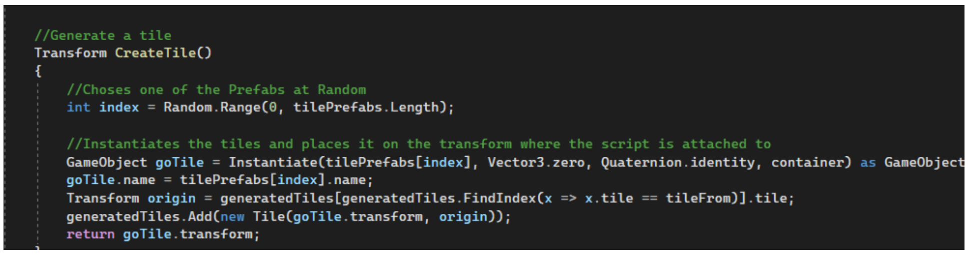
Start Prefabs 3

Element 0	Room Cap.fancy
Element 1	Room Cap
Element 2	Room Cap.soft

Starting point:



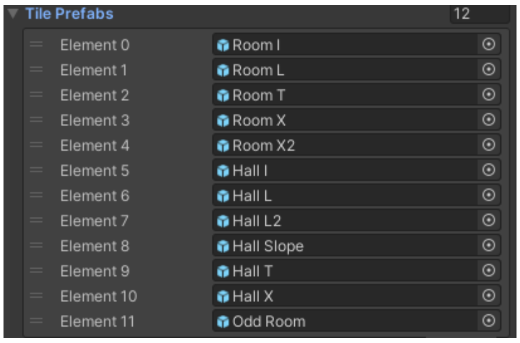
Create and connect tiles



```
//Generate a tile
Transform CreateTile()
{
    //Choses one of the Prefabs at Random
    int index = Random.Range(0, tilePrefabs.Length);

    //Instantiates the tiles and places it on the transform where the script is attached to
    GameObject goTile = Instantiate(tilePrefabs[index], Vector3.zero, Quaternion.identity, container) as GameObject
    goTile.name = tilePrefabs[index].name;
    Transform origin = generatedTiles[generatedTiles.FindIndex(x => x.tile == tileFrom)].tile;
    generatedTiles.Add(new Tile(goTile.transform, origin));
    return goTile.transform;
```

The snippet above is used to create a new tile, once the starting tile is created.

One of the tiles from the Tile Prefabs list is selected at random and spawned onto the game scene once the generator is running.

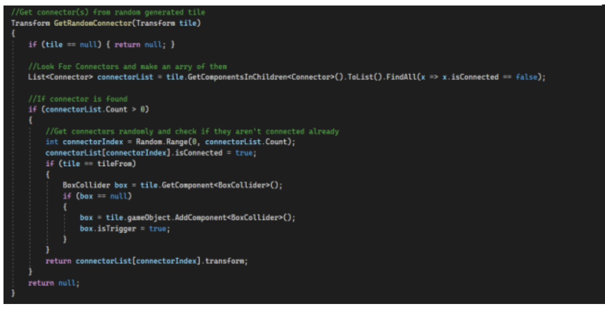


Tile Prefabs	12
Element 0	Room I
Element 1	Room L
Element 2	Room T
Element 3	Room X
Element 4	Room X2
Element 5	Hall I
Element 6	Hall L
Element 7	Hall L2
Element 8	Hall Slope
Element 9	Hall T
Element 10	Hall X
Element 11	Odd Room

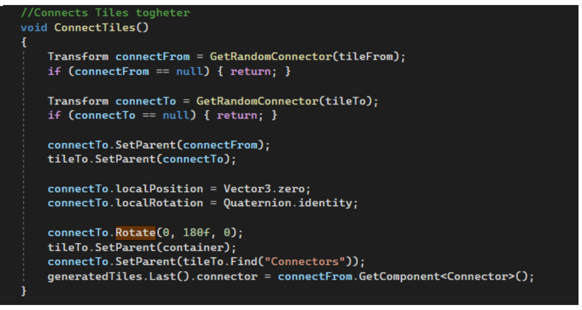
But just creating the next tile is not enough, they have to connect with the starting tile, otherwise both will pile up on top of each other, therefore, a connector is needed.

GetRandomConnector(), looks for connector from the generated tiles, and checks if any of them are already connected to other connectors.

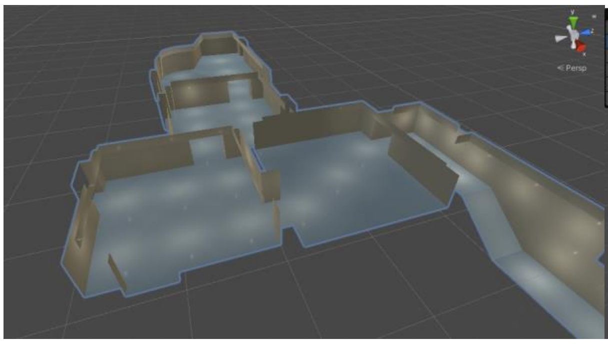
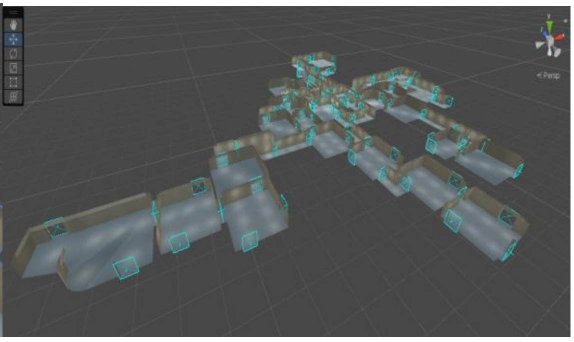
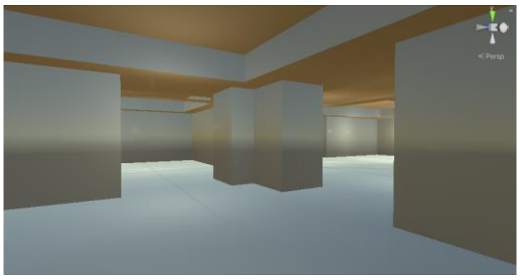
If none of them are, the boolean variable *isConnected* is set to true, if not then false, and regardless of the result of the condition, the list of connectors along with their positions will be returned, where *ConnectTiles()* can process the information and connect the free tile connector together.



```
//Get connector(s) from random generated tile
Transform GetRandomConnector(Transform tile)
{
    if (tile == null) { return null; }

    //Look For Connectors and make an arry of them
    List<Connector> connectorList = tile.GetComponentsInChildren<Connector>().ToList().FindAll(x => x.isConnected == false);

    //If connector is found
    if (connectorList.Count > 0)
    {
        //Get connectors randomly and check if they aren't connected already
        int connectorIndex = Random.Range(0, connectorList.Count);
        connectorList[connectorIndex].isConnected = true;
        if (tile == tileFrom)
        {
            BoxCollider box = tile.GetComponent<BoxCollider>();
            if (box == null)
            {
                box = tile.gameObject.AddComponent<BoxCollider>();
                box.isTrigger = true;
            }
        }
        return connectorList[connectorIndex].transform;
    }
    return null;
}
```



```
//Connects Tiles togheter
void ConnectTiles()
{
    Transform connectFrom = GetRandomConnector(tileFrom);
    if (connectFrom == null) { return; }

    Transform connectTo = GetRandomConnector(tileTo);
    if (connectTo == null) { return; }

    connectTo.SetParent(connectFrom);
    tileTo.SetParent(connectTo);

    connectTo.localPosition = Vector3.zero;
    connectTo.localRotation = Quaternion.identity;

    connectTo.Rotate(0, 180f, 0);
    tileTo.SetParent(container);
    connectTo.SetParent(tileTo.Find("Connectors"));
    generatedTiles.Last().connector = connectFrom.GetComponent<Connector>();
}
```

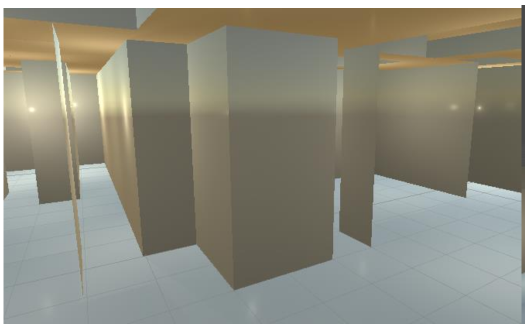
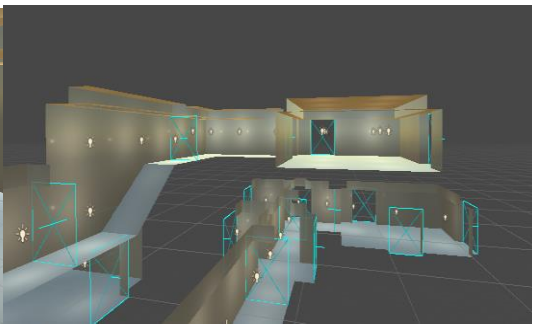




Technical Challenges

The dungeon generator works, but there are still some problems that needed to be addressed.

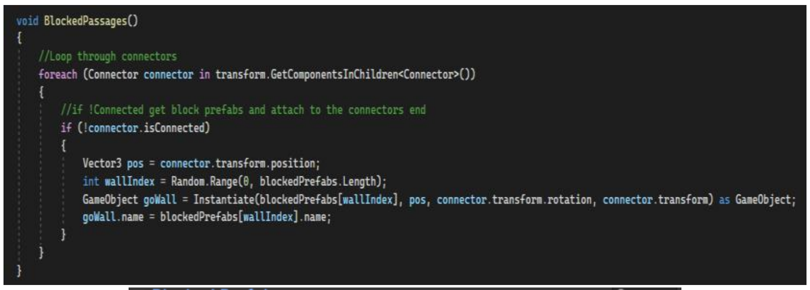
Such problems were:

- Close the gaps / Holes on the walls
- Create proper door frames
- Fix tile clipping and collisions
- Minimize the Randomness

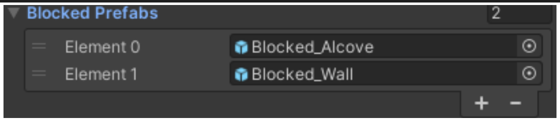



Close the gaps

BlockedPassages() function, will loop through each connector form generated tiles, and if said connectors are free, a random block tile will be randomly selected from the Blocked Prefab List and be placed on to the free connector, covering the hole.

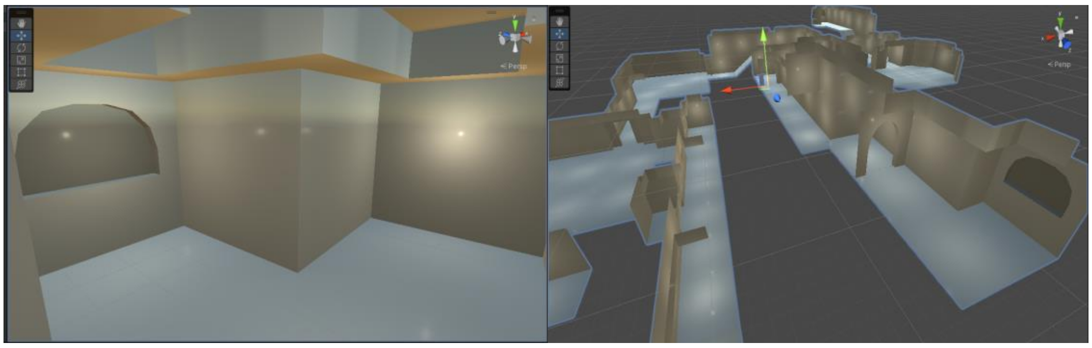


```
void BlockedPassages()
{
    //Loop through connectors
    foreach (Connector connector in transform.GetComponentsInChildren<Connector>())
    {
        //if !Connected get block prefabs and attach to the connectors end
        if (!connector.isConnected)
        {
            Vector3 pos = connector.transform.position;
            int wallIndex = Random.Range(0, blockedPrefabs.Length);
            GameObject goWall = Instantiate(blockedPrefabs[wallIndex], pos, connector.transform.rotation, connector.transform) as GameObject;
            goWall.name = blockedPrefabs[wallIndex].name;
        }
    }
}
```



Blocked Prefabs 2

Element 0	Blocked_Alcove
Element 1	Blocked_Wall

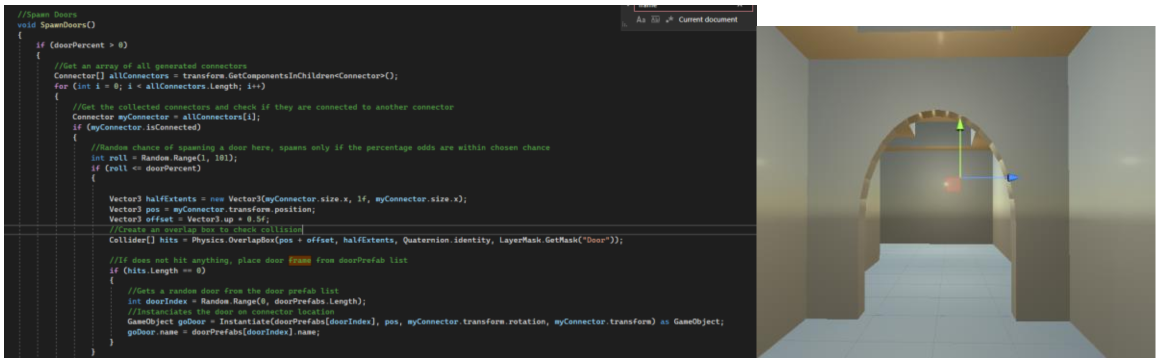


Create proper door frames

SpawnDoor() function Creates an array of all connectors generated on the level, iterates them and checks if they are connected to another connector.

The *roll<=doorPercent* condition will only spawn a door if the odds are within chosen chance, and if so, a collision box is created in place of the collected connectors.

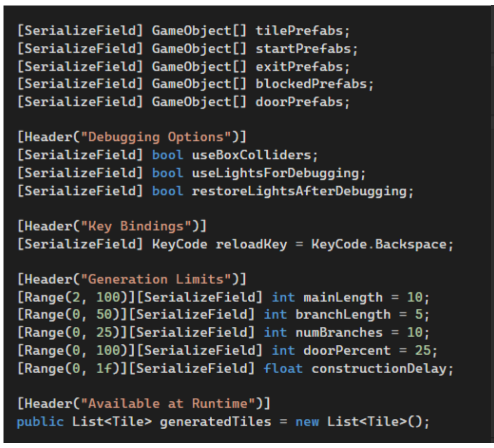
If the box overlaps with anything along the connector area, a door frame is placed from the list of Door prefabs.



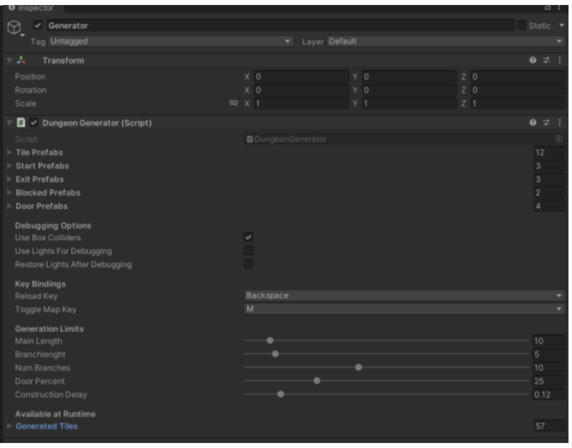
```
//Spawn Doors
void SpawnDoors()
{
    if (doorPercent > 0)
    {
        //Get an array of all generated connectors
        Connector[] allConnectors = transform.GetComponentsInChildren<Connector>();
        for (int i = 0; i < allConnectors.Length; i++)
        {
            //Get the collected connectors and check if they are connected to another connector
            Connector myConnector = allConnectors[i];
            if (myConnector.isConnected)
            {
                //Random chance of spawning a door here, spawns only if the percentage odds are within chosen chance
                int roll = Random.Range(1, 101);
                if (roll <= doorPercent)
                {

                    Vector3 halfExtents = new Vector3(myConnector.size.x, 1f, myConnector.size.x);
                    Vector3 pos = myConnector.transform.position;
                    Vector3 offset = Vector3.up * 0.5f;
                    //Create an overlap box to check collision
                    Collider[] hits = Physics.OverlapBox(pos + offset, halfExtents, Quaternion.identity, LayerMask.GetMask("Door"));

                    //If does not hit anything, place door frame from doorPrefab list
                    if (hits.Length == 0)
                    {
                        //Gets a random door from the door prefab list
                        int doorIndex = Random.Range(0, doorPrefabs.Length);
                        //Instanciates the door on connector location
                        GameObject goDoor = Instantiate(doorPrefabs[doorIndex], pos, myConnector.transform.rotation, myConnector.transform) as GameObject;
                        goDoor.name = doorPrefabs[doorIndex].name;
                    }
                }
```

Minimize Randomness

Random was minimized by exposing, core variables of the main functions to the user, transforming the project from a random generator to a procedural one.



```csharp
[SerializeField] GameObject[] tilePrefabs;
[SerializeField] GameObject[] startPrefabs;
[SerializeField] GameObject[] exitPrefabs;
[SerializeField] GameObject[] blockedPrefabs;
[SerializeField] GameObject[] doorPrefabs;

[Header("Debugging Options")]
[SerializeField] bool useBoxColliders;
[SerializeField] bool useLightsForDebugging;
[SerializeField] bool restoreLightsAfterDebugging;

[Header("Key Bindings")]
[SerializeField] KeyCode reloadKey = KeyCode.Backspace;

[Header("Generation Limits")]
[Range(2, 100)][SerializeField] int mainLength = 10;
[Range(0, 50)][SerializeField] int branchLength = 5;
[Range(0, 25)][SerializeField] int numBranches = 10;
[Range(0, 100)][SerializeField] int doorPercent = 25;
[Range(0, 1f)][SerializeField] float constructionDelay;

[Header("Available at Runtime")]
public List<Tile> generatedTiles = new List<Tile>();
```



```
[✓] Generator
Tag Untagged                Layer Default

Position    X 0   Y 0   Z 0
Rotation    X 0   Y 0   Z 0
Scale       X 1   Y 1   Z 1

Script                  DungeonGenerator
Tile Prefabs            12
Start Prefabs           3
Exit Prefabs            3
Blocked Prefabs         2
Door Prefabs            4

Debugging Options
Use Box Colliders               ✓
Use Lights For Debugging        ☐
Restore Lights After Debugging  ☐

Key Bindings
Reload Key        Backspace
Toggle Map Key    M

Generation Limits
Main Length         10
Branchlenght        5
Num Branches        10
Door Percent        25
Construction Delay  0.12

Available at Runtime
Generated Tiles     57
```