

Dungeon Generator – How it works

Adam Speers

Support email: adam.speers@hotmail.co.uk

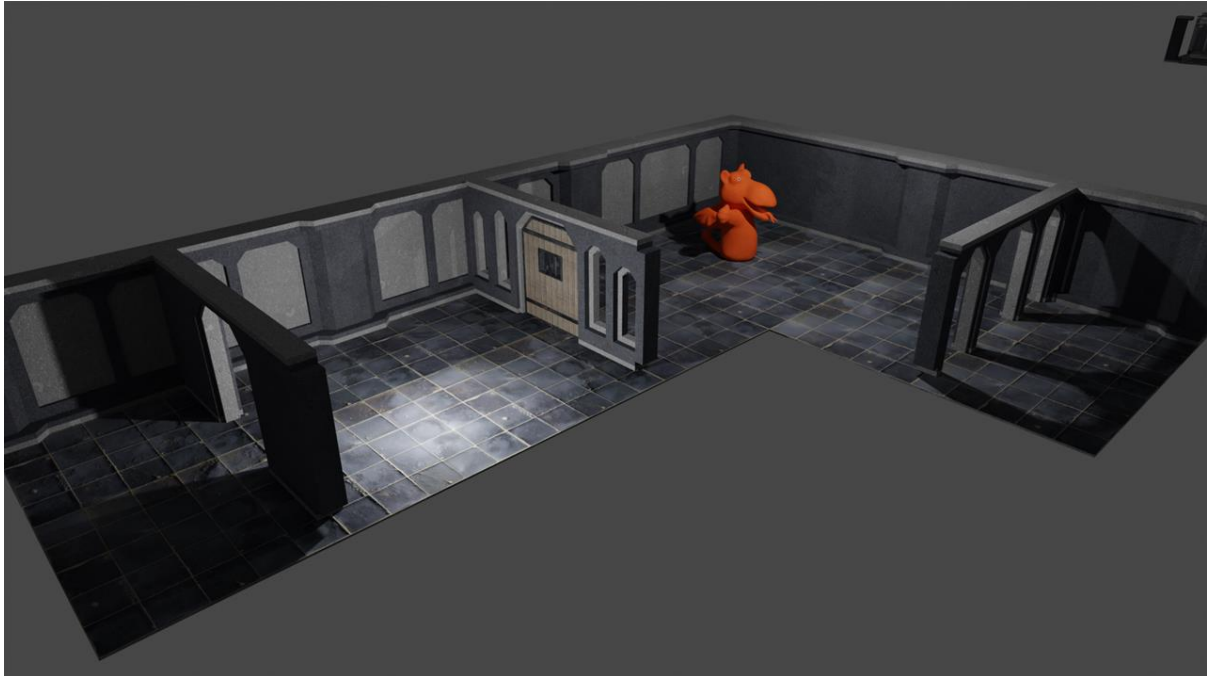


Figure 1 - Sample Dungeon

Introduction

Tile Dungeon Generator uses a procedural maze generator combined with a tile-based approach to generate a dungeon. The maze algorithm uses a combination walk forward two, and recursive back track algorithm. The routine first creates a series of arrays and data structures that represent the logical maze layout. These arrays are then converted from a one-dimensional structure into a 3D representation. The code then assesses each path tile against its surrounding neighbours, to find the correct tile type. Once the correct tile type is known, a scriptable object holding references to the prefabs for each type, is read and the object spawned into the correct grid position and rotated.

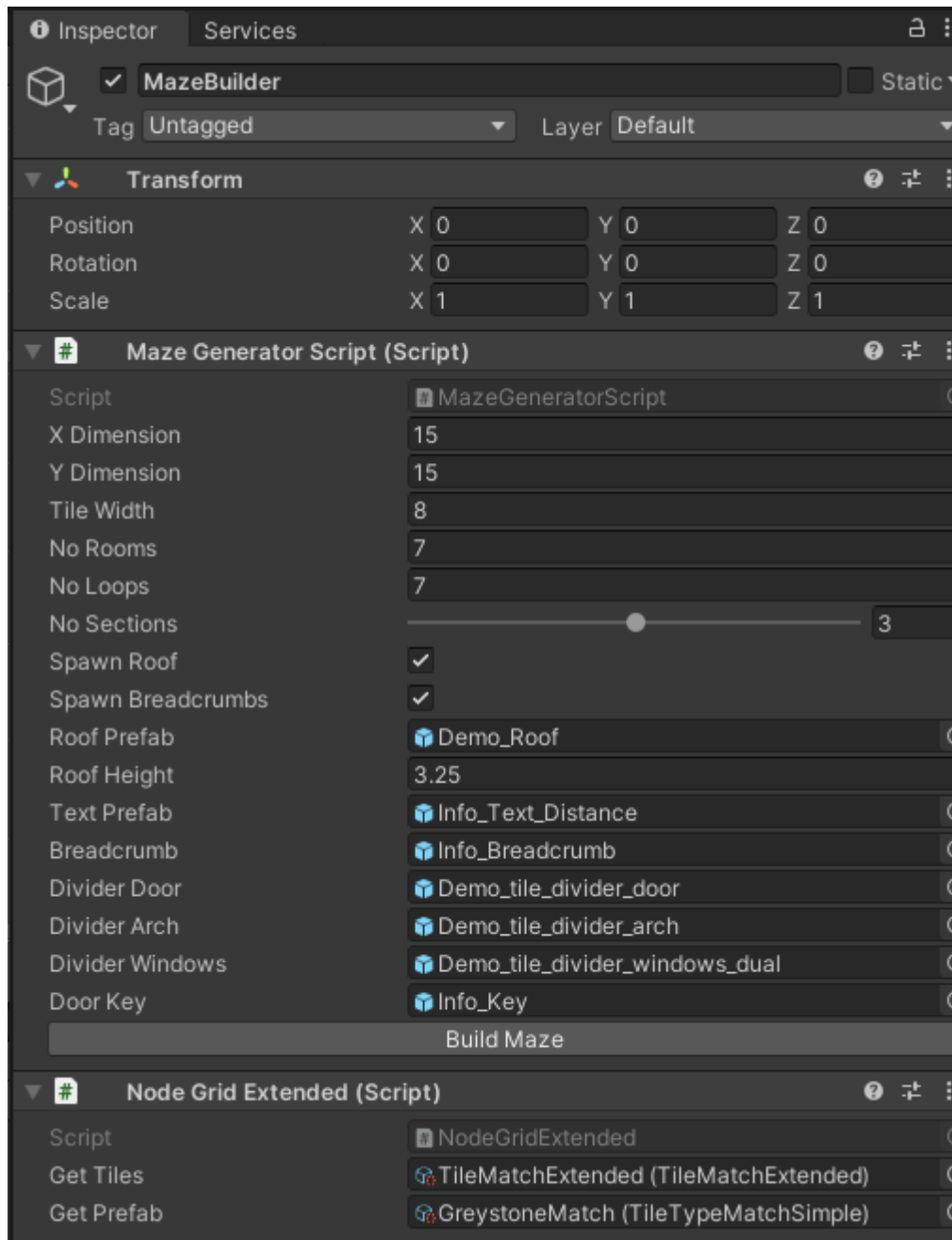
This document is split into four parts.

- Part 1: Quick start – Get started straight away by exploring the DemoScene and generating a dungeon with the provided assets.
- Part 2: Describes the logical design of the code so you can get a good understanding of how it works and why.
- Part 3: Describes how to build your own dungeon tiles, how they should be created, correct default rotations so they function correctly with the tool.
- Part 4: Describes how the tile matching scriptable objects work, how they are used to place and rotate your tiles into the scene.

Part 1: Quick Start

Demo Scene

After importing the package into your Unity project navigate to the Demo scene. Click on the MazeBuilder object in the scene hierarchy. You should see the controls for the Dungeon Generator.



Click the Build Button! This will generate a maze with the default settings. Once you have done this delete the generated objects that appear as children of MazeBuilder in the hierarchy. Change some settings and build another one. Please note that the generator works best when selecting odd numbers for both X and Y dimensions.

Part 2: Dungeon Generator Design

Process Overview

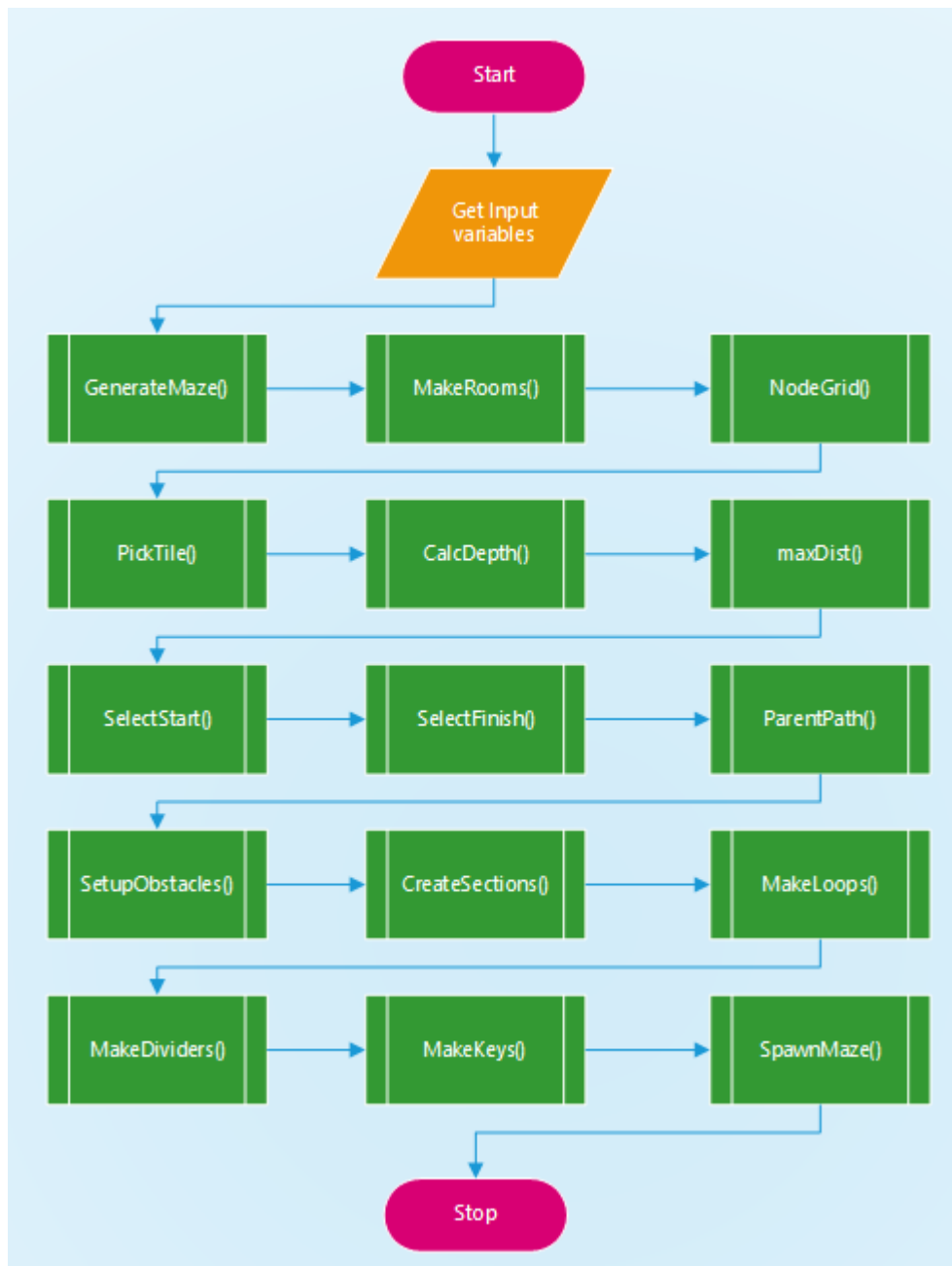


Figure 2 - Overview

The dungeon generator utilises 5 key scripts in its implementation which can be found in the code repository (available [here](#)):

- MazeGeneratorScript.cs
- Node.cs
- NodeGridExtended.cs
- TileMatchExtended.cs

- TileTypeMatchSimple.cs

Together these scripts perform a number of process steps in order to generate a dungeon. (Figure 2)

Process 1: Generate Maze Logic

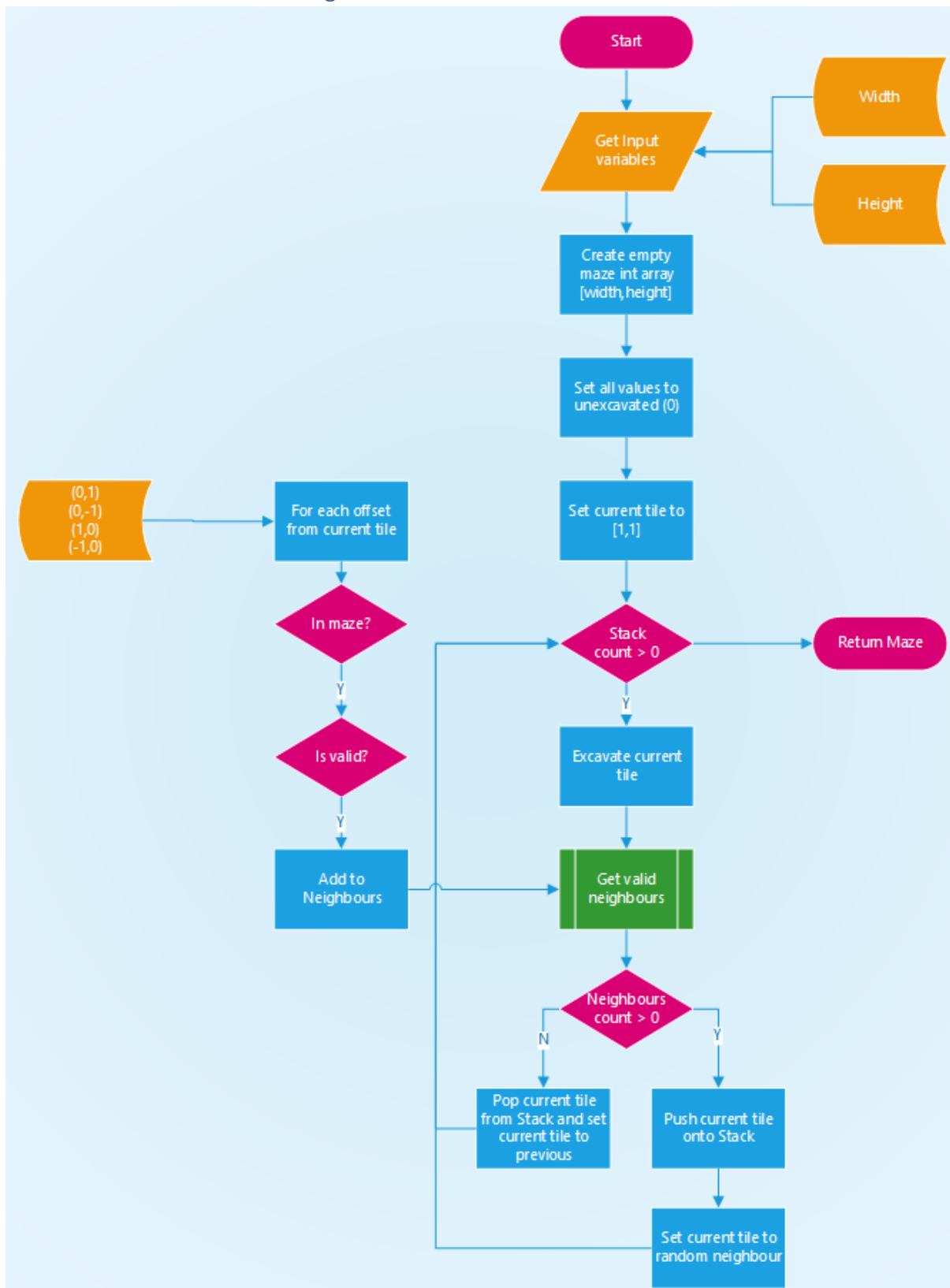


Figure 3 - Generate maze

The first stage in the process (Figure 3) is creating the base structure for the dungeon, this is formed of a maze generated using a recursive backtrack algorithm.

Summary:

1. From the chosen starting point (default is 1,0,1) private variable but can be changed in the script.
2. Randomly choose a neighbour and carve a passage through to that cell, but only if the neighbour has not been visited yet. This becomes the new current cell.
3. If all neighbour cells have been visited, back up to the last cell that has unvisited neighbours and repeat.
4. The algorithm ends when the process has backed up all the way up to the starting point.

Code:

This process calls the following methods: contained within MazeGeneratorScript.cs

- GenerateMaze() - Lines 63-79
- CreateMaze() - Lines 81-113
- GetValidNeighbours(CurrentTile) - Lines 118-144
- HasThreeWallsIntact(Vector2 Vector2ToCheck) - Lines 528-548
- IsInside(neighborToCheck) - Lines 550-573

Process 2: Make Rooms

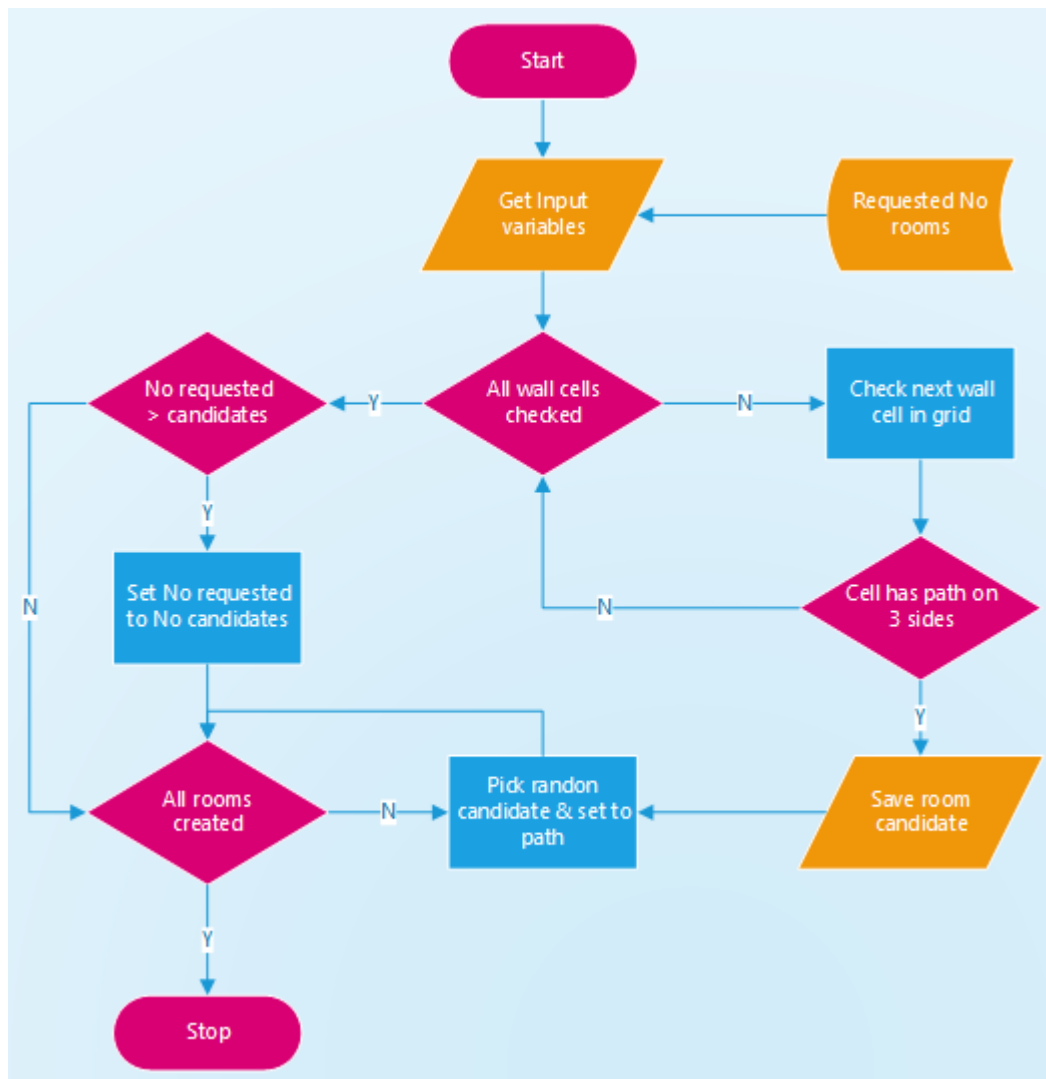


Figure 4 - Make Rooms

Summary:

The designer may have requested (Via Variable) that rooms be created. Rooms are created by identifying candidates in the maze data grid. To qualify a cell must be surrounded on three sides (checking the four cardinal directions) by paths. Qualifying cells are added to a candidate's pool. A check is made to ensure the requested number of rooms does not exceed the number of candidates, if the number requested is greater, it is capped at the number possible. The requested number of candidates (or maximum possible) are then randomly selected from the pool and changed into paths; these cells now form larger open areas.

Code:

This process calls the following methods contained within MazeGeneratorScript.cs

- SaveEndCaps() - Lines 555-5573
- HasThreePaths(Vector2 Vector2ToCheck) - Lines 593-613

- MakeRooms() - Lines 575-591

Process 3: Node Grid

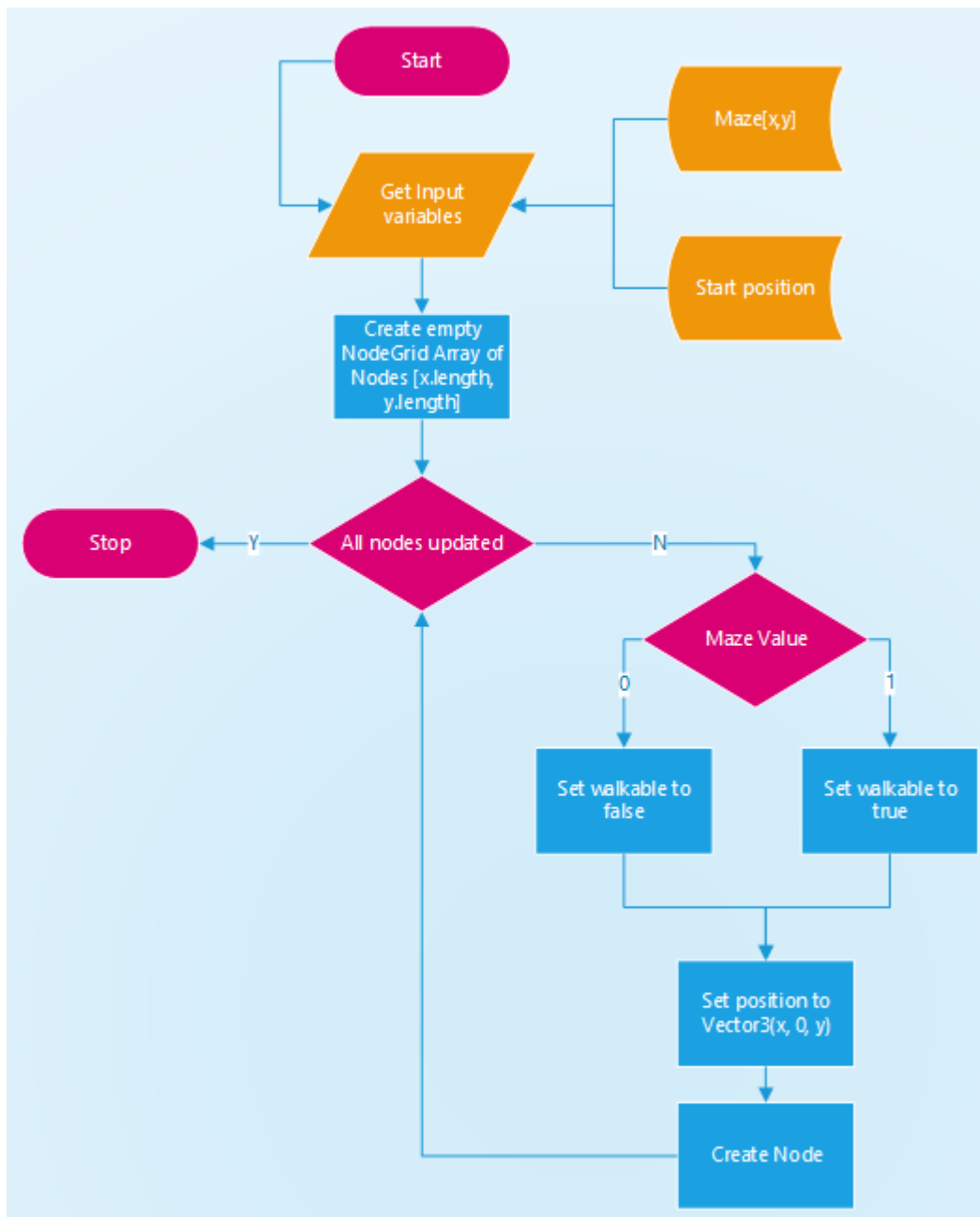


Figure 5 - Node Grid

Summary:

This process uses the scripts Node.cs and NodeGridExtended.cs to expand on the maze grid previously created. It begins by building a Nodegrid, an array of Nodes holding rich information about each cell in the dungeon, initially it records:

- public bool walkable;
- public Vector3 position;
- public int gridX;

- `public int gridY;`

Code:

This process calls the following methods / scripts:

- `GetDepthMap()` - Lines 237-241 (`MazeGeneratorExtended.cs`)
- `CreateNodeGrid()` - Lines 35-49 (`NodeGridExtended.cs`)
- `Node()` - Lines 26-32 (`Node.cs`)

Process 4: Pick Tile

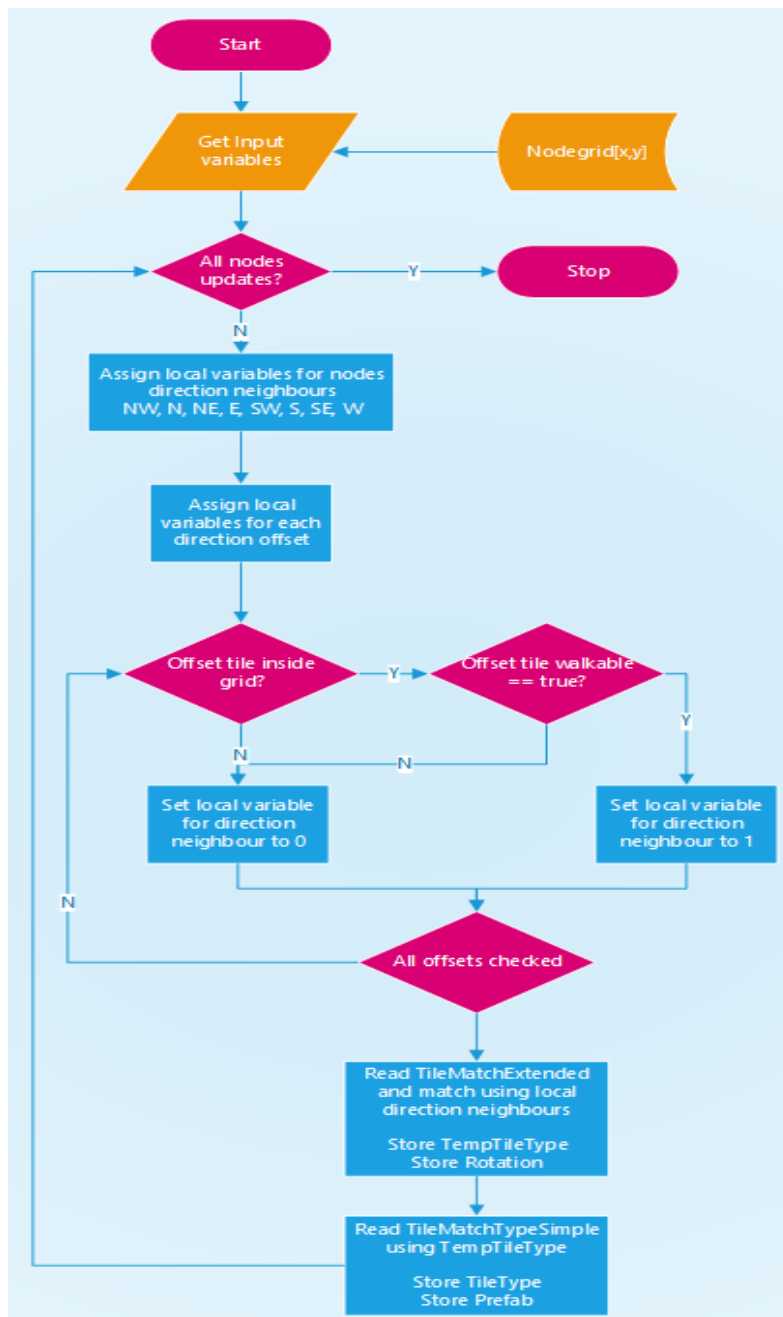


Figure 6 - Pick Tile Type

Summary:

This process uses NodeGridExtended.cs and the scriptable Object references held for TileMatchExtended.cs and TileTypeMatchSimple.cs.

1. For each node in nodegrid.
2. Create a local variable for each direction neighbour.
3. For each neighbour offset, read tile in nodegrid and assign local variable based on value of walkable, if requested offset location is outside grid walkable is false.

4. Read the scriptable object for tile matching using local variables direction neighbour walkable value. Store rotation and tiletype.

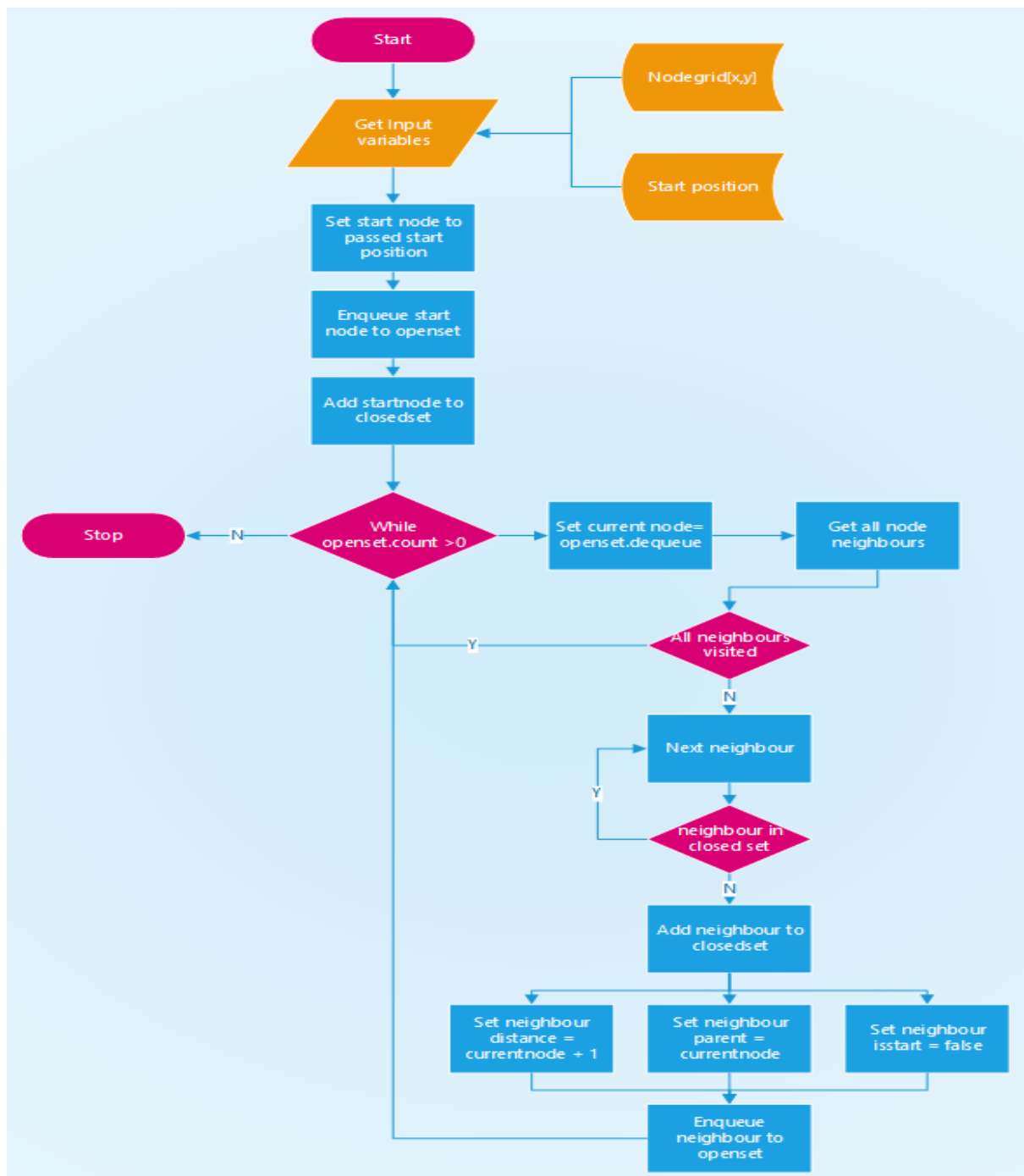
5. Read the scriptable object for tile type and store the Prefab.

Code:

This process calls the following methods:

- PickTile() Line 111-251 (NodeGridExtended.cs)

Process 5: Calculate Depth Map



Summary:

This process calculates a depthmap for the entire nodegrid.

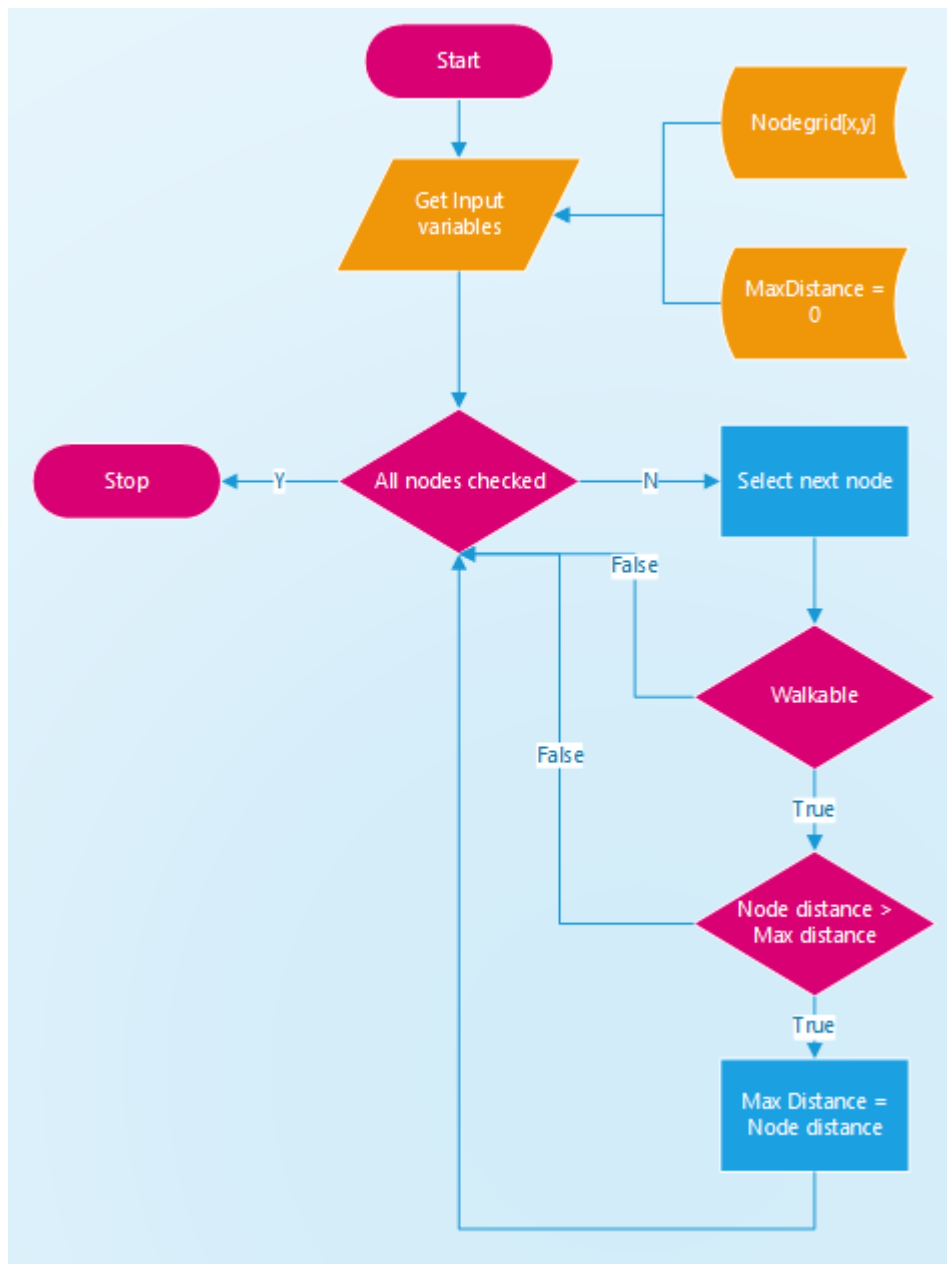
1. From the starting position set depth to 0. Set to current node.
2. Find each walkable neighbour.
3. For each unexplored neighbour (Add to closedset). set distance to current distance + 1, Store a reference to the neighbours parent.
4. Add explored neighbour to queue for exploring its neighbours.
5. Continue until all walkable nodes have been explored.

Code:

This process calls the following methods:

- `NodeGridCalcDepth(Vector3 startPos)` Line 71-109 (`NodeGridExtended.cs`)

Process 6: Max Distance



Summary:

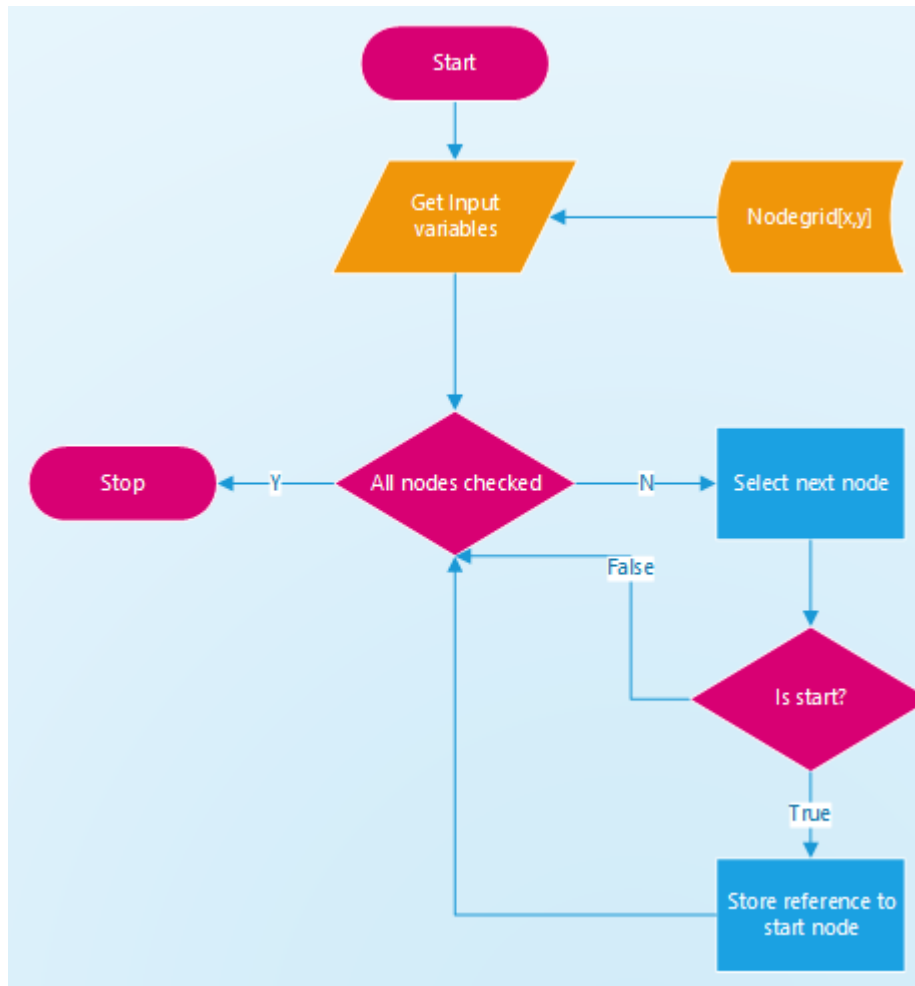
This process follows on from the depth map calculation and parses the node grid to identify the maximum distance from the start position. This max distance value can be used to identify a logic exit for the dungeon as discussed in (Dungeon Generator: Part 3).

1. Set maximum distance to 0.
2. For each node in node grid.
3. Test to see if the node is walkable
4. If walkable, is the node distance > current max distance. If greater set the maximum distance = node distance.
5. Continue until all walkable nodes have been explored.

Code:

This process calls the following methods:

- maxDist() Line 71-109 (MazeGeneratorExtended.cs)

Process 7: Set Start Node**Summary:**

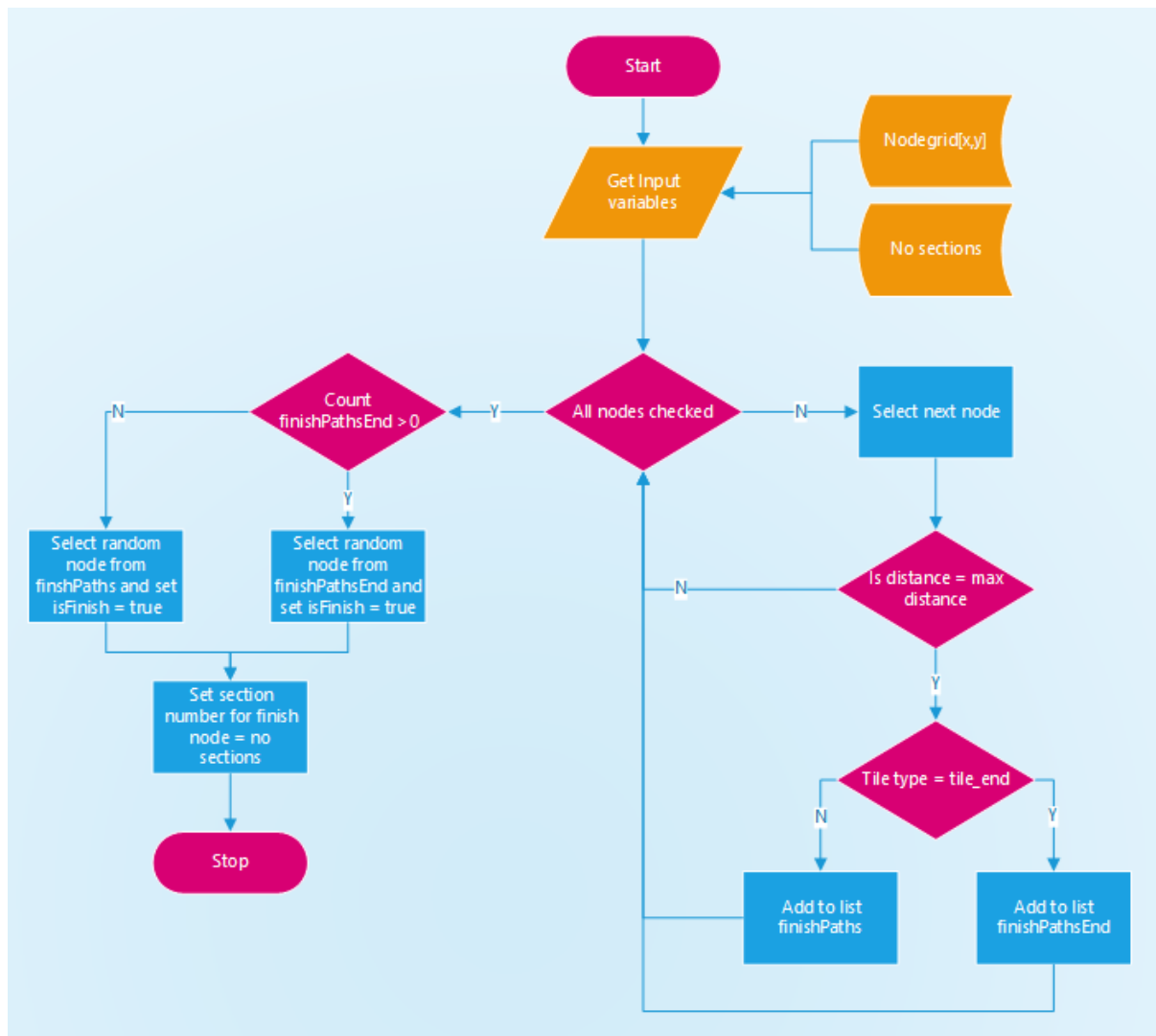
This simple process parses the node grid and identifies the start node, it stores this as a separate reference for later use in Process 9: Parent Path.

Code:

This process calls the following method:

- selectStart() Line 158-168 (MazeGeneratorExtended.cs)

Process 8: Select Finish



Summary:

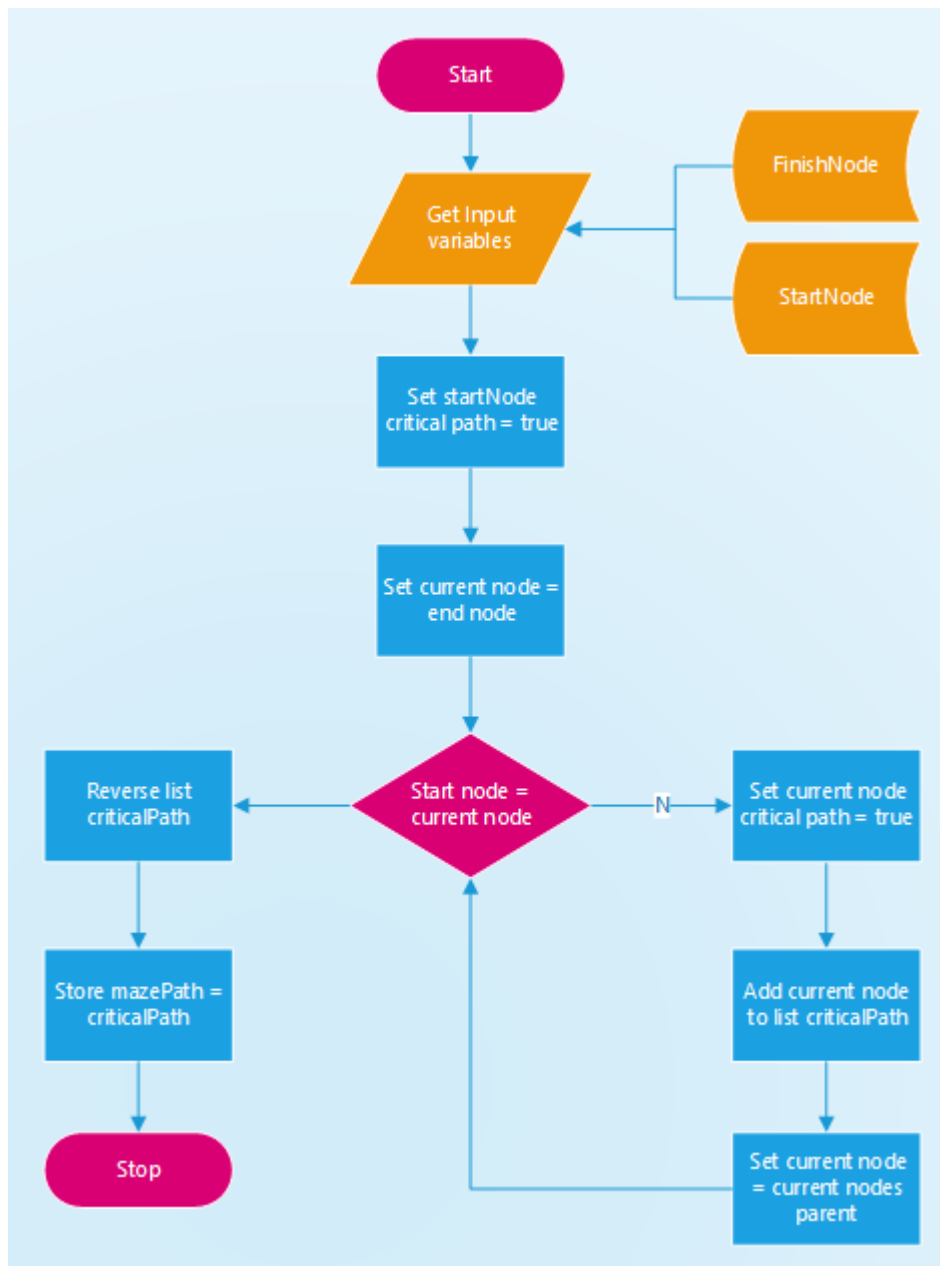
This process parses the node grid and identifies any node whose distance from the start = maximum distance calculated in Process 6. These nodes are stored into 2 lists. Those that have the tile type "tile_end" into list finishPathsEnd and everything else into finishPaths. If there are candidates in list finishPathsEnd select a node at random and set it to be the finish. Otherwise select a node at random from finishPaths and set it to be the finish.

Code:

This process calls the following method:

- selectFinish() Line 170-207 (MazeGeneratorExtended.cs)

Process 9: Parent Path



Summary:

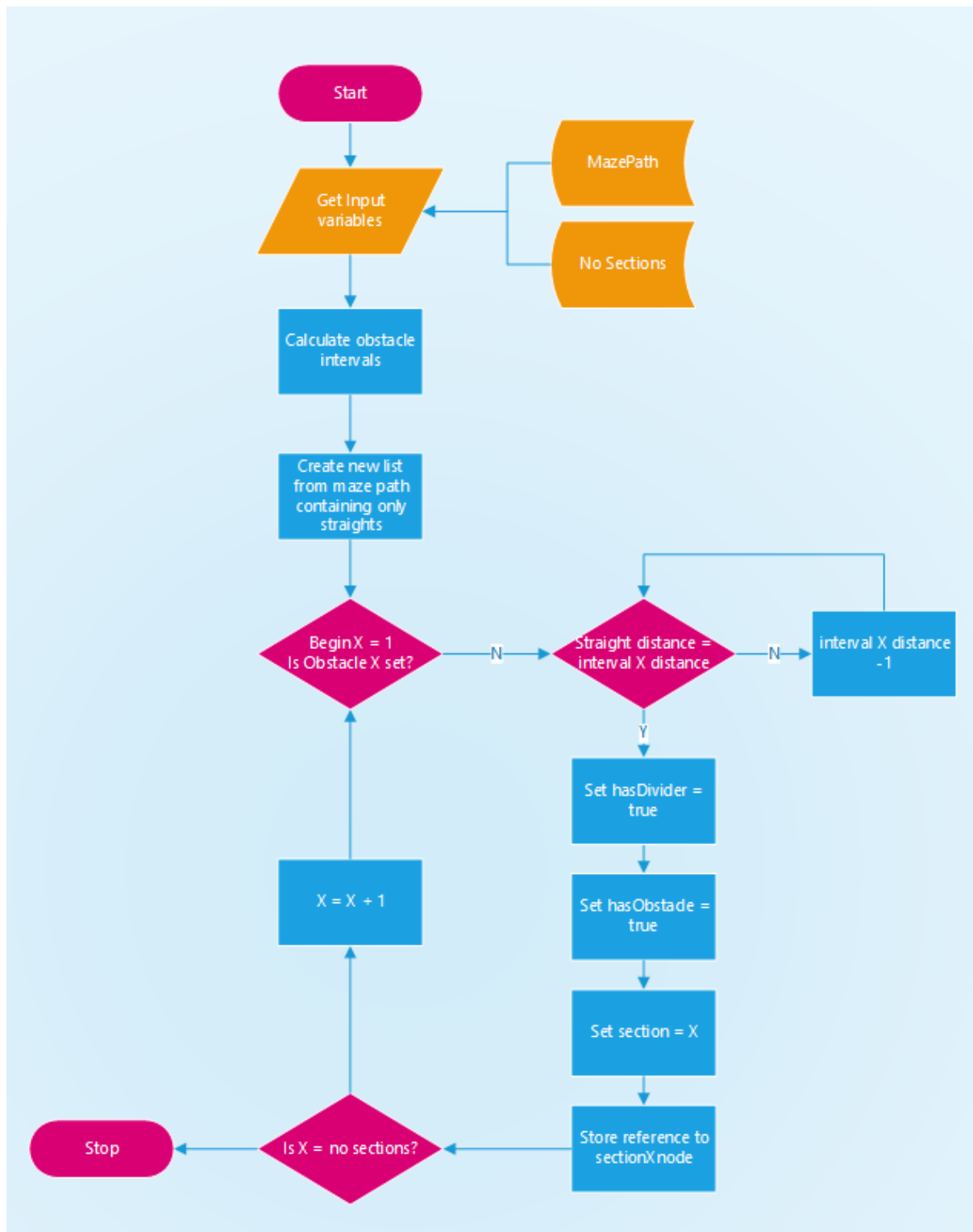
This process performs a walk through the node grid starting with the finish node identified in Process 8. The walk will continue to move through the grid by selecting each node's parent in turn, until it has reached the start node. Each node it passes through is flagged as forming part of the critical path. This critical path can then be used to implement bottlenecks and challenges as discussed in (Dungeon Generator: Part 1).

Code:

This process calls the following method:

- ParentPath(Node startNode, Node endNode) Line 680-696 (MazeGeneratorExtended.cs)

Process 10: Setup Obstacles



Summary:

This process uses the critical path created previously in Process 9. And uses this information to create bottlenecks. These are required to implement the desired three-act structure as discussed in (Dungeon Generator: Part 1).

1. Read the number of sections required - default is 3 max 4

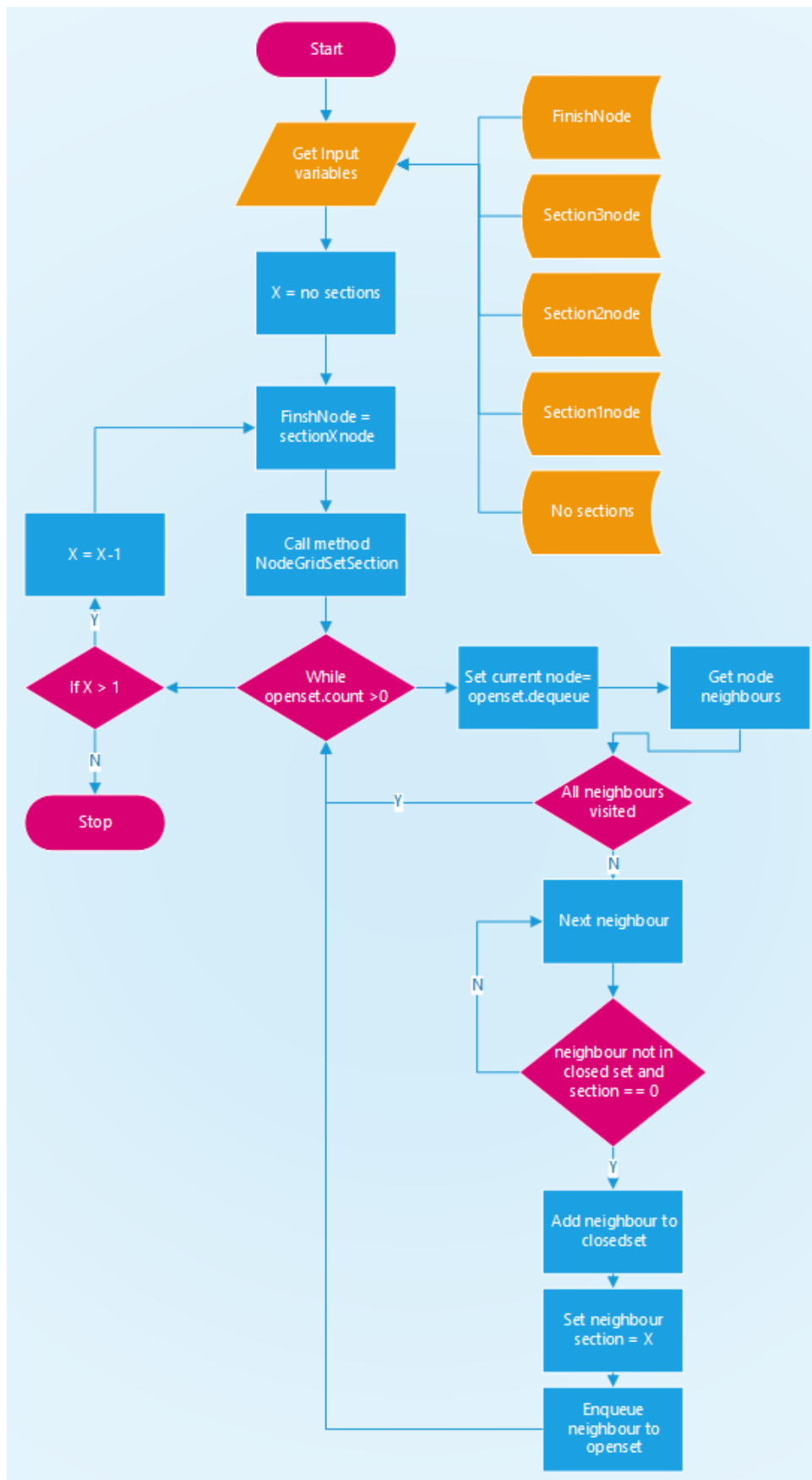
2. If the number required is > 1.
3. Calculate desired obstacle position ($\text{obstacleInterval} = \text{endNode.distance} / \text{noSections}$).
 $\text{int obstacle1} = \text{obstacleInterval}$; $\text{int obstacle2} = \text{obstacleInterval} * 2$; $\text{int obstacle3} = \text{obstacleInterval} * 3$;
3. Create a new list straights and add nodes from the Mazepath with tile type "tile_straight".
4. Setup obstacle 1: Foreach node in list straights, if the nodes distance = obstacle1. Set node variables: hasDivider = true; hasObstacle = true; section = 1; obstacle1set = true; section1Node = straights.node;
5. If the nodes distance <> obstacle1 set obstacle1 = obstacle1 - 1 and repeat step 4.
6. Repeat steps 4 & 5 as required for the remaining sections.

Code:

This process calls the following method:

- SetupObstacles() Line 240-323 (MazeGeneratorExtended.cs)

Process 11: Set Section Identifiers



Summary:

This process uses a similar method to the parent path, its purpose is set the section number identity for each node. The method always begins with the endnode of the maze and walks back towards the start. The walk will continue until it has found all the members of a section. Sections end at the next bottleneck encountered or the startnode.

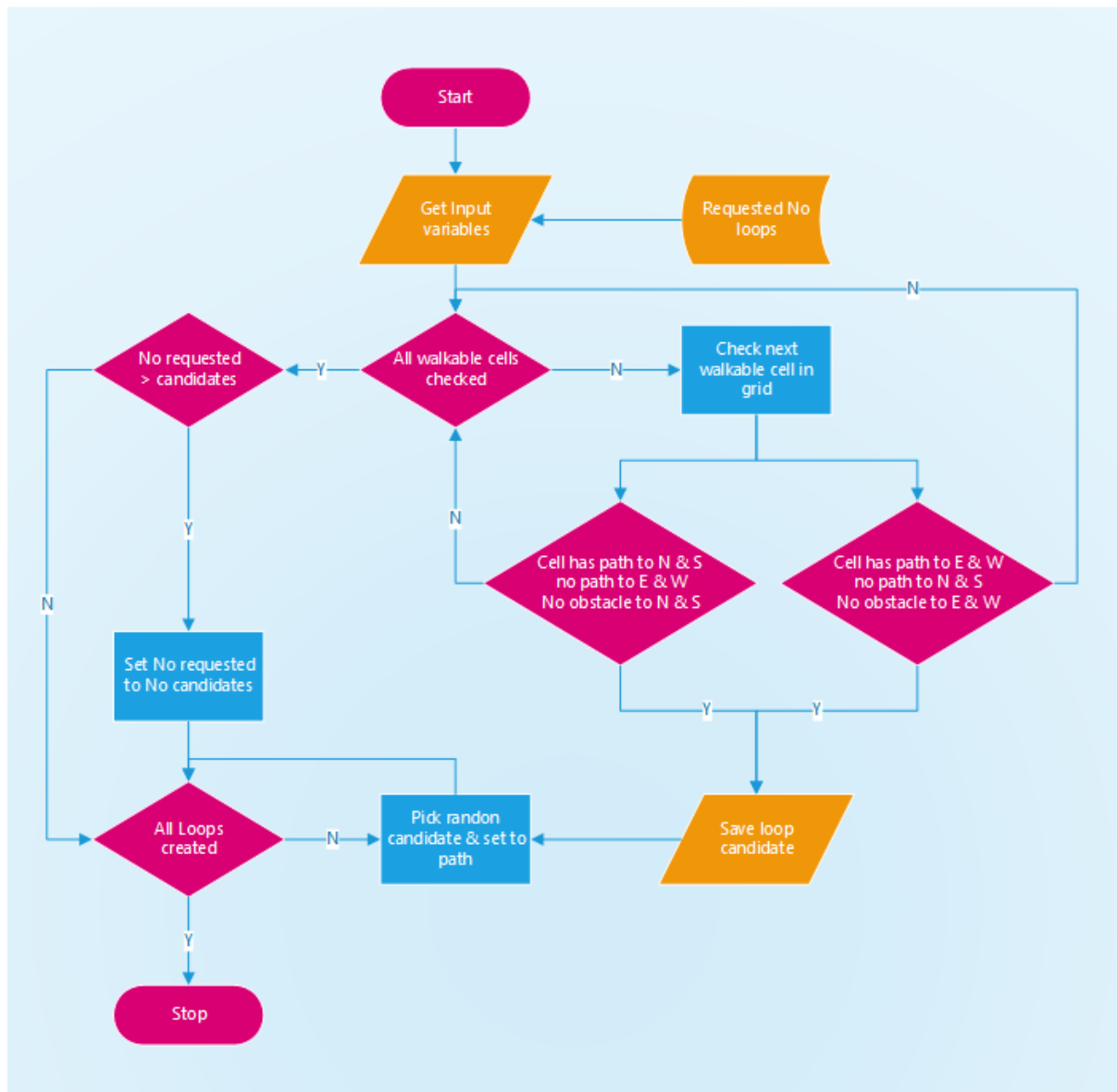
The identification of each sector, allows for the implementation of solutions to obstacles. This forms another part of the desired three-act structure as discussed in (Dungeon Generator: Part 1).

Code:

This process calls the following methods:

- CreateSections() Lines 325-348 (MazeGeneratorExtended.cs)
- NodeGridSetSection(Vector3 startPos, int section) Lines 622-657 (MazeGeneratorExtended.cs)

Process 12: Create loop paths



Summary:

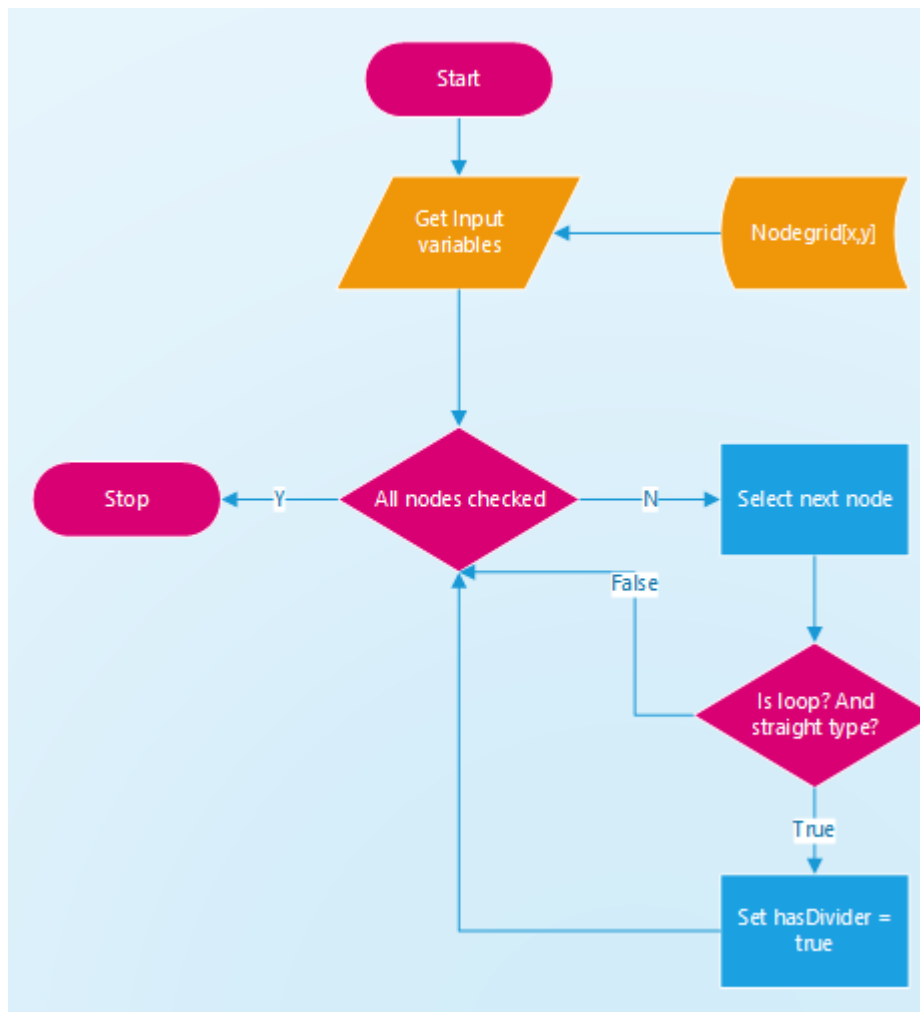
This process creates additional loop paths (braids) within the dungeon. These are desired in order to encourage additional exploration and aid the illusion of player choice. The assignment of section identities in the process 11, allows constraint when creating loop paths. The code will only form new corridors between members of the same section, this preserves the bottlenecks previously created. Further information about identifying braid candidates can be found in (Dungeon Generator: Part 2).

Code:

This process calls the following methods:

- `SaveNodeLoopCandidates()` Lines 553-591 (MazeGeneratorExtended.cs)
- `MakeNodeLoops()` Lines 592-610 (MazeGeneratorExtended.cs)

Process 13: Make dividers



Summary:

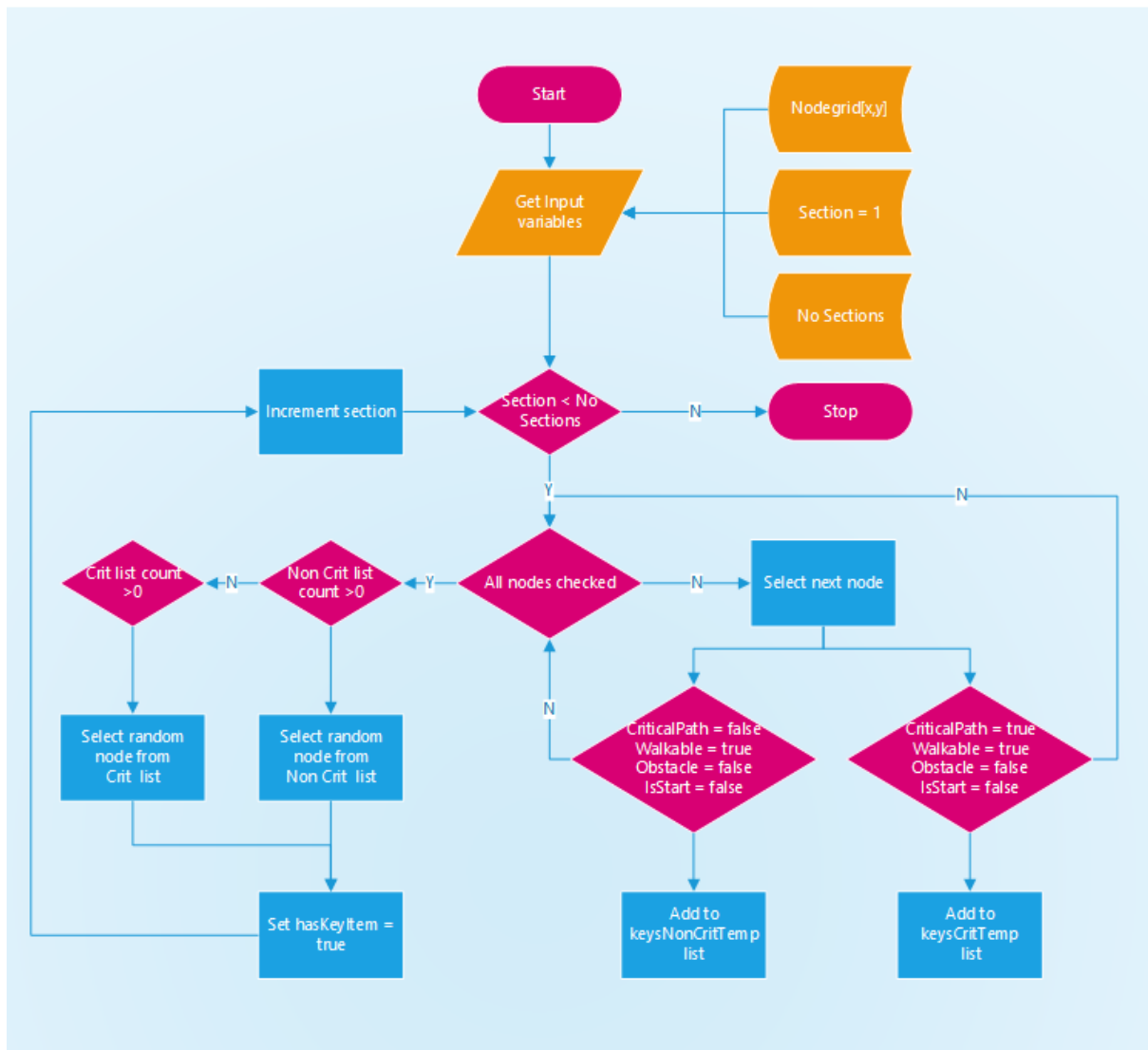
This process parses through the newly created loop paths and if a newly created loop has tile type "tile_straight" sets a variable in the node information map for having a divider. These divider indicators can subsequently be used to spawn in addition geometry, providing additional variation to the dungeon, such as a wall, archway or destructible object.

Code:

This process calls the following method:

- MakeLoopsDividers() Lines 612-621 (MazeGeneratorExtended.cs)

Process 14: Make Solutions (Keys)



Summary:

This process identifies a cell within a section as a candidate for spawning a solution to an obstacle. The player will be required to find / interact with this object before they can proceed past the bottleneck. In the prototype dungeon demo scene this is demonstrated using a system of locked doors and keys.

1. For each section prior to last section
2. Identify 2 lists of candidate nodes, keysNonCritTemp and keysCritTemp. The difference being cells that are also on the critical path.
3. If keysNonCritTemp count > 0 select a node at random and set hasKeyItem = true.
4. Else If keysCritTemp count > 0 select a node at random and set hasKeyItem = true.

Code:

This process calls the following method:

- MakeKeyHolders() Lines 700-750 (MazeGeneratorExtended.cs)

Process 15: Spawn Dungeon

Summary:

This process is the culmination of all the previous steps. The system parses through the node grid and for each tile spawns the appropriate Prefab(s). If the debug switches are selected within the editor (Figure 7.1) the system will also spawn addition information into the scene:

Debug Information

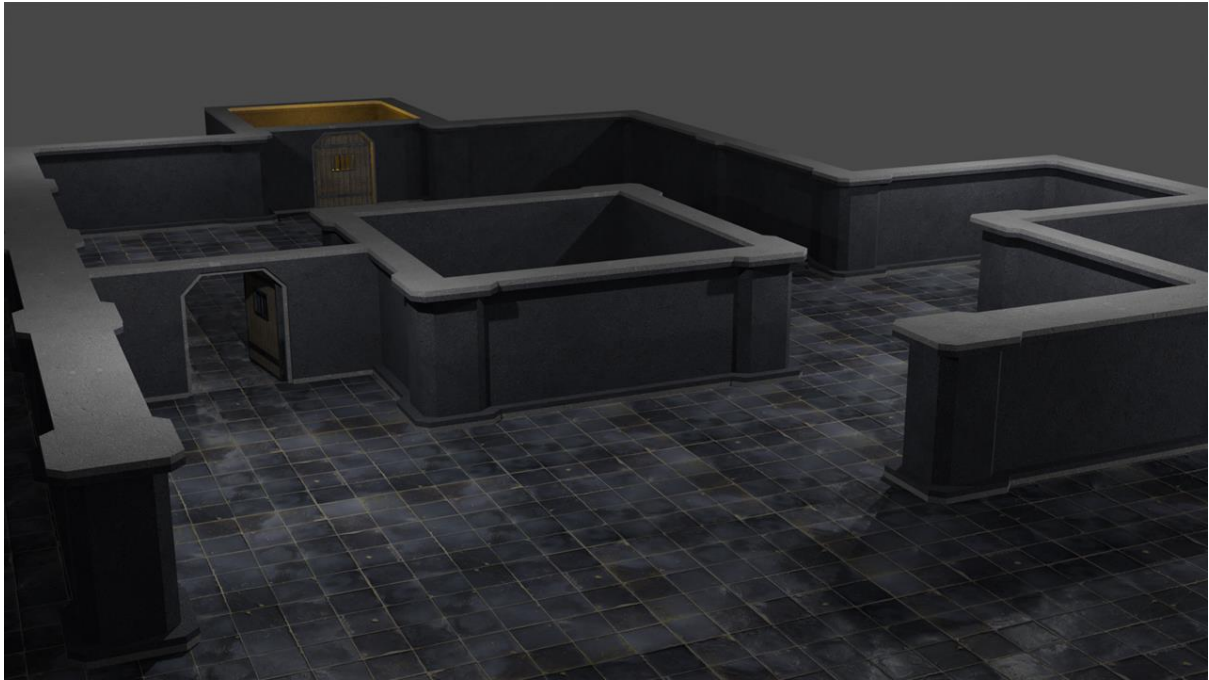
- Distance text
- Section text
- Tile type indicator (Coloured)
- Breadcrumb (Critical path)

Prefabs

- Tiles
- Divider
- Wall + Door
- Key

Part 3: Designing Tiles to work with the tool

Constructing the basics

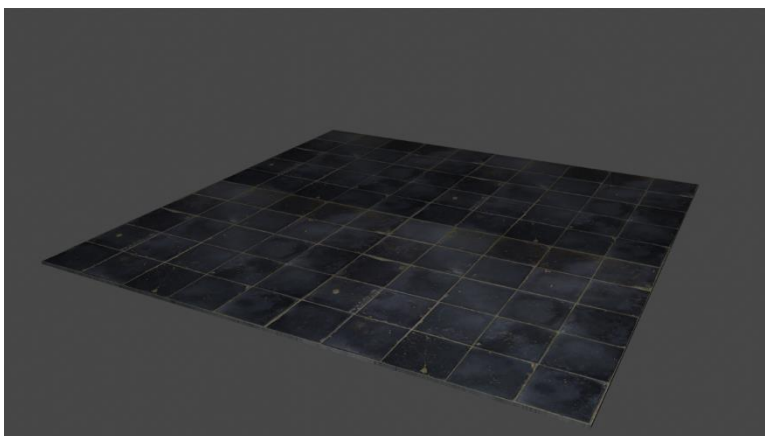


In order to fully utilise the dungeon generator tool, a designer must understand the construction rules used to create the example assets. Dungeon tiles are organised within a scene based on a grid system and formed into the tile shapes using a combination of three simple building blocks. When created, following these rules, the finished tile set will create a seamless environment.

The sample assets are based on a grid square measuring 8m x 8m (Or 8 unity units), however you can adjust the script to suit if you prefer a different measurement.

Base components:

Floor



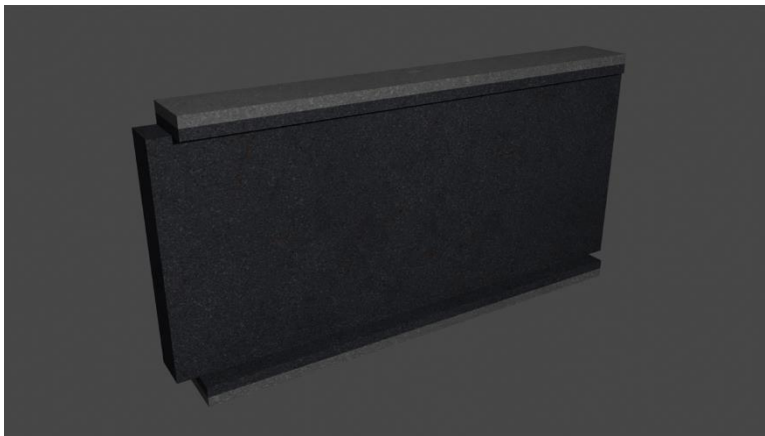
- Outer Dimensions: 8m x 8m x 0.05m
- Pivot Location: Centre of upper face

Pillar



- Outer Dimensions: 1m (0.75m) x 1m (0.75m) x 3.25m
- Pivot Location: Bottom rear corner vertex

Wall

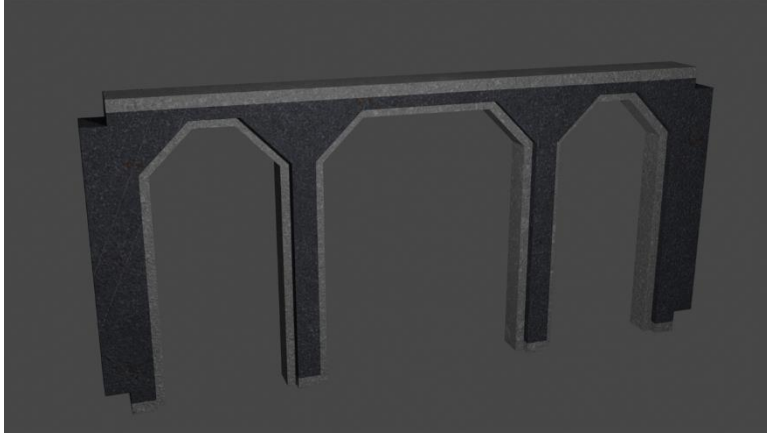


- Outer Dimensions: 0.75m x 6.5m x 3.25m
- Pivot Location: Bottom rear centre

Additional Components:

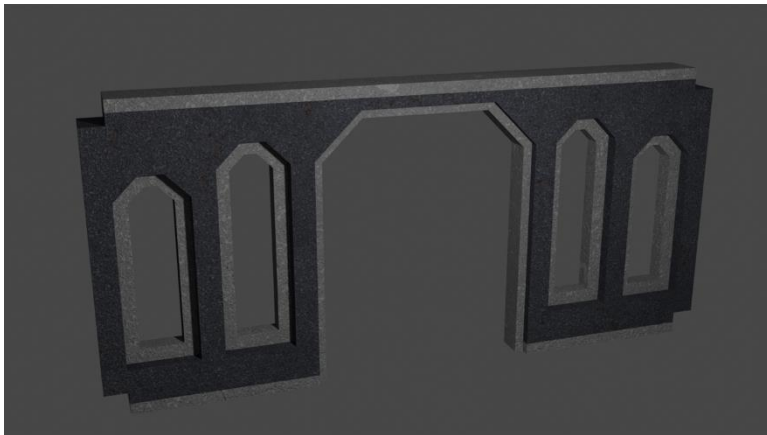
These components are added to the dungeon to support the semi-linear structure and do not form part of the tile matching process.

Divider Arch



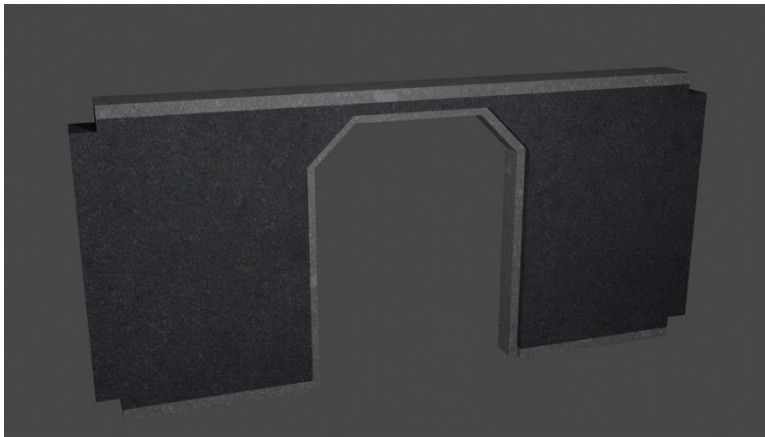
- Outer Dimensions: 0.5m x 7.0m x 3.25m
- Pivot Location: Bottom centre

Divider Windows



- Outer Dimensions: 0.5m x 7.0m x 3.25m
- Pivot Location: Bottom centre

Divider Door Frame



- Outer Dimensions: 0.5m x 7.0m x 3.25m
- Pivot Location: Bottom centre

Door



- Outer Dimensions: 0.5m x 7.0m x 3.25m
- Pivot Location: Bottom rear corner

Tile Construction Diagrams

Using the base components, all necessary tile types can be constructed, facilitating population of a generated dungeon with geometry.


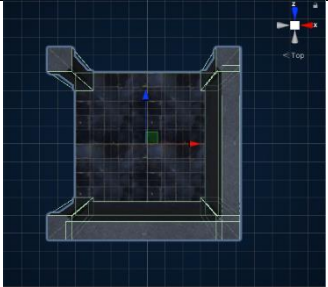

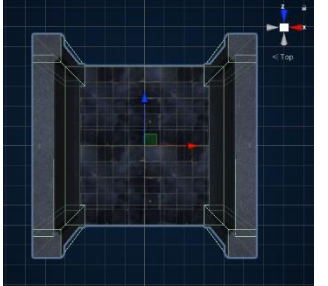

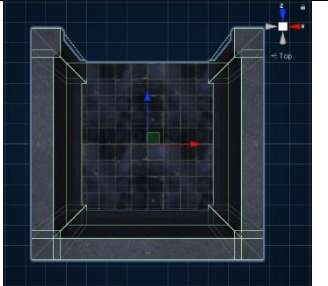

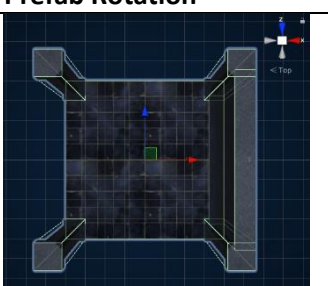
The **simple** dungeon generator uses 5 tile types.


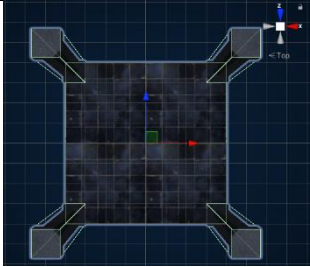
The **extended** version of the generator includes 9 additional tile types.


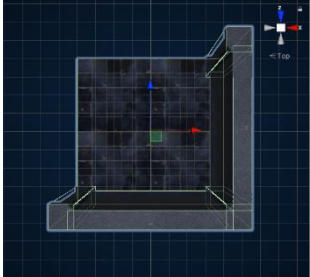
The following set of construction diagrams illustrate how each example asset has been formed into a prefab within Unity (Unity Technologies, 2020). They also serve as templates for each tile type. A designer may also use alternate dimensions for tiles, provided that each tile in the new tile-set has identical square footprints. The tool (MazeGeneratorExtended.cs) has an input variable for the base dimensions in the editor which will adjust the spawning rules automatically to accommodate the chosen dimensions. This functionality allows the designer freedom to implement tile designs whose



aesthetics evoke desired emotional responses. For example: tight narrow areas may evoke feeling of claustrophobia; cluttered spaces to make players feel nervous and towering architecture to make the player feel small and insignificant.

Tile Prefabs



Tile_Corner_1	Render	Prefab Rotation
1 Floor 2 Walls 4 Pillars		
Tile_Straight	Render	Prefab Rotation
1 Floor 2 Walls 4 Pillars		
Tile_End	Render	Prefab Rotation
1 Floor 3 Walls 4 Pillars		
Tile_Tjunc_2	Render	Prefab Rotation
1 Floor 1 Wall 4 Pillars		

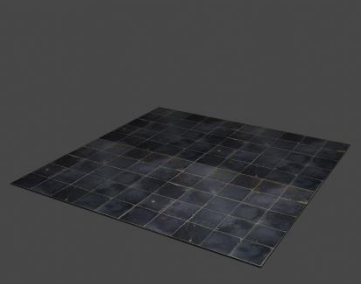

Tile_Crossroads_4	Render	Prefab Rotation
1 Floor 4 Pillars		


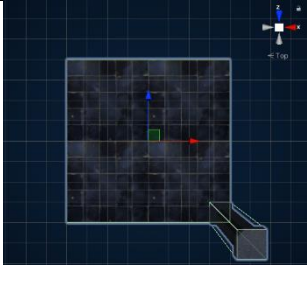
Tile_Corner_0	Render	Prefab Rotation
1 Floor 2 Walls 3 Pillars		

Tile_Tjunc_0	Render	Prefab Rotation
1 Floor 1 Wall 2 Pillars		


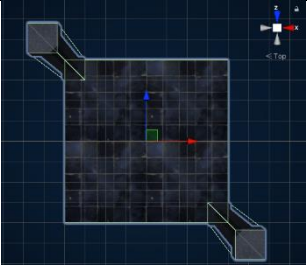
Tile_Tjunc_1a	Render	Prefab Rotation
1 Floor 1 Wall 3 Pillars		


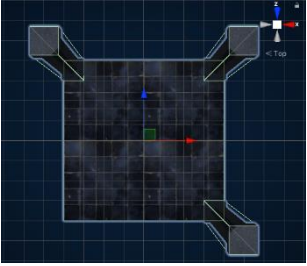
Tile_Tjunc_1b	Render	Prefab Rotation
1 Floor 1 Wall 3 Pillars		

Tile_Crossroads_0	Render	Prefab Rotation
1 Floor		

Tile_Crossroads_1	Render	Prefab Rotation
1 Floor 1 Pillar		

Tile_Crossroads_2a	Render	Prefab Rotation
1 Floor 2 Pillars		

Tile_Crossroads_2b	Render	Prefab Rotation
1 Floor 2 Pillars		

Tile_Crossroads_3	Render	Prefab Rotation
1 Floor 3 Pillars		

Part 4: Picking correct tile type

How it picks the correct tile?

Picking the correct tile type

Examining the sample tile-set, it is possible to visualise how the different tile elements fit together. The dungeon generator uses a method to determine the correct tile and rotation for each position. This has been achieved by looking at the node information grid, navigating to each neighbour for a cell and recording if the neighbour is walkable or not (Figure 6.42). A cells tile type is determined by the pattern formed around the cell by its neighbours:

Look for neighbours:

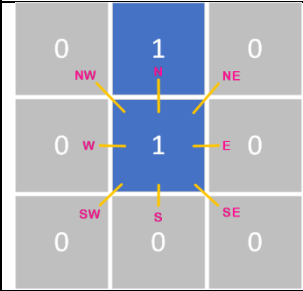
Simple: North, East, South, West

Extended: North West, North, North East, East, South West, South, South East, West

The process for finding the correct tile type can be summarised as:

1. Read the walkable value for a cell's neighbours in each direction
2. Lookup corresponding values in the tile match table scriptable object.
3. Store the matching tile type and rotation.

Each tile Prefab is stored as a single object and has a default rotation. Patterns for matching tiles are repeated for each direction. The lookup provides both the matching tile type and rotation. The lookup rotation will be applied to the Prefab default and thus align the Prefab with the pattern found.

Tile Matching	Lookup	Spawn
	<div>Element 76</div> <div>Element 77</div> <div>Element 78</div> <div>Element 79</div> <div>Element 80</div> <div>North West Tile</div> <div>0</div> <div>North Tile</div> <div>1</div> <div>North East Tile</div> <div>0</div> <div>West Tile</div> <div>0</div> <div>East Tile</div> <div>0</div> <div>South West Tile</div> <div>0</div> <div>South Tile</div> <div>0</div> <div>South East Tile</div> <div>0</div> <div>Rotation</div> <div>0</div> <div>Tile Type</div> <div>tile_end</div>	