Smart Rooms

Procedural Level Generator & Room Builder

USER GUIDE

Release 1.0.0

16. 7. 2023

Getting Started

Overview

This User Guide is crafted to equip ***Smart Rooms*** users with a basic understanding of the tool's features.

Setting Up

Upon downloading Smart Rooms from Unity's Asset Store, navigate to: "Assets->Import Package->Custom Package...". Locate and select the SmartRoomsProceduralLevelGeneratorRoomBuilder.unitypackage file in the Import Asset window. Once the "Importing package" window pops up in Unity, ensure all items to import are chosen and then click the Import button positioned at the lower right corner of the window.

Getting Started

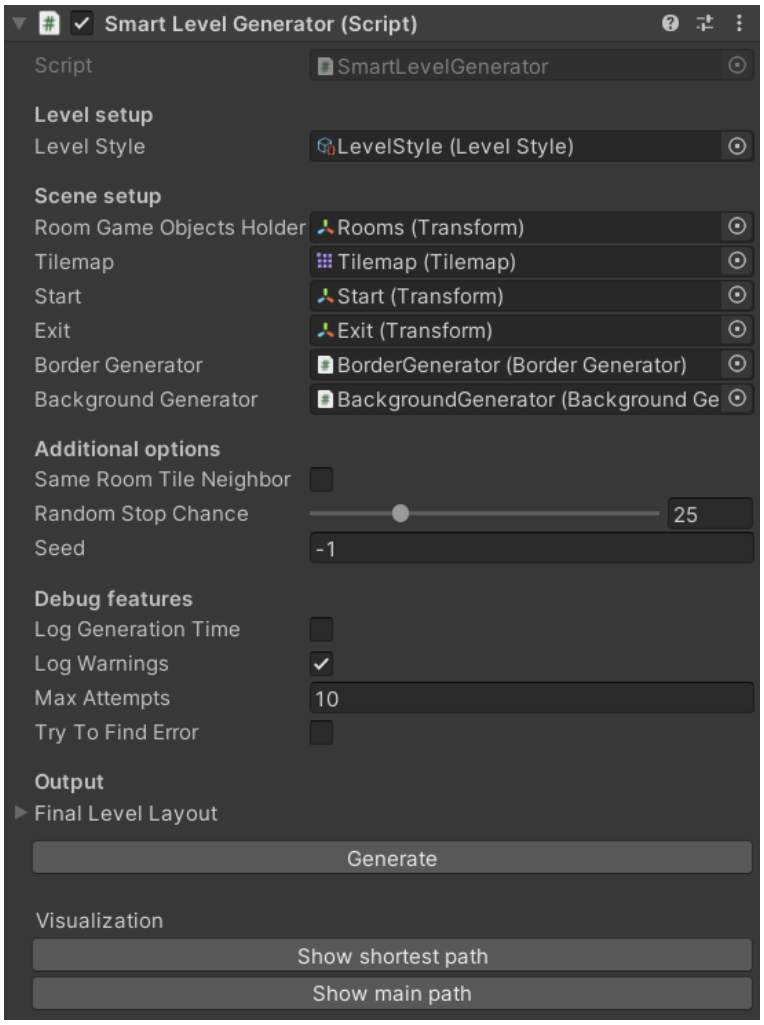
A scene is prepared which contains a demo of Level Generation. You can find the scene at Plugins/SmartRooms/Scenes/LevelGeneration.unity.
When you open the scene hit play in the Editor and play with the game a bit to explore it. Getting a good understanding of how the generator works before diving deeper is crucial for building deep knowledge.

Help & API Documentation

For any inquiries or support needs, please visit our Discord channel at https://discord.gg/BqjyfzeYYX where you can access additional details, ask questions, get Video Tutorials, and FAQs. If you can't find the information you're looking for, feel free to ask for help in our channel.

Generators

Smart Level Generator



The SmartLevelGenerator class is responsible for generating a level based on a **LevelStyle**.

Settings

maxBuildIterations: An integer constant that represents the maximum number of build iterations allowed.

_levelStyle: A LevelStyle object that represents the style of the level to be generated.

_roomGameObjectsHolder: A Transform object that represents the holder for the room game objects.

_tilemap: A Tilemap object that represents the tilemap for the level.

_start: A Transform object that represents the Start Game Object.

_exit: A Transform object that represents the Exit Game Object.

_borderGenerator: A BorderGenerator which generates the border for the level.

_backgroundGenerator: A BackgroundGenerator object which generates the background based on level style.

_sameRoomTileNeighbor: A boolean value that determines whether identical Room Tiles can be adjacent.

_randomStopChance: The chance of randomly stopping the generation process when the final layer is reached.

_seed: An integer value that represents the seed for the random number generator.

_logGenerationTime: A boolean value that determines whether or not to log the generation time.

_logWarnings: A boolean value that determines whether or not to log warnings.

_maxAttempts: An integer value that represents the maximum number of attempts to generate the level.

_tryToFindError: A debug boolean used to find an error during the generation loop.

API

StartGeneration(LevelStyle levelStyle): A method that generates the level based on the provided LevelStyle.

StartGeneration(): A method that generates the level based on the current LevelStyle.

Tilemap TargetTilemap: Returns the current target Tilemap

LevelStyle CurrentLevelStyle: Returns current level style used

LevelLayout FinalLevelLayout: Returns the final generated level layout

Events

LevelGenerated: This event is raised when level generation is complete.

Unity Editor Buttons

Generate: Generates a new level. Can be used both in Edit and Play mode.

Show shortest path: Shows the shortest path to the exit generated using A* algorithm.

Show generation order: Shows the main path to the exit of the current level.

Generation Logic

The level generator generates the level out of rooms provided in the Level Style. The level generator converts rooms into Room Tiles which are then placed into the level layout to fulfill certain conditions like direction of player travel and guaranteed movement possibility between tiles. The generation steps are as follow:

1) Convert all rooms into Room Tiles
2) Rotate all Room Tiles based on settings inside the Rooms. This creates more Room Tiles by duplicating and rotating the provided Room Tiles.
3) Calculate for each Room Tile which other Room Tiles it can connect to on each of its sides.
4) Build Level:
    a) Build main path of the level:
        i) Place one of the starting Room Tiles in the starting row/column.
        ii) Pick a direction in which to go based on available directions of the previous Room Tile and randomly select a new tile from the available Room Tiles which can be connected. This creates a guaranteed path from starting Room Tile to this Room Tile. Repeat this step until you reach the final layer. Only ever go in the direction of the final layer or sideways, but never backward.

iii) At the final layer you either have to stop due to space constraint or due to random chance. When the generation stops it replaces the last placed Room Tile with an exit Room Tile.

iv) This process guarantees a traversable path from start to finish

v) If one of the steps fails to continue to build a traversable path then this step fails which triggers a rebuild of the level.

b) Spawn special rooms:

i) Go through the main path Room Tiles in the level layout and try to spawn special Room Tiles which are connected to the main path. Not mandatory special rooms might not spawn due to random chance.

ii) Some special rooms can be bigger, composed of two Room Tiles. They can only spawn if the whole structure can fit in the level.

iii) If a special mandatory room ID has not been placed into the level then this step fails which triggers rebuild of the level.

c) Fill empty spots in level with random Room Tiles:

i) Go through all the empty spots in the level and randomly pick a Room Tile to put there.

ii) These Room Tiles don't have to be connected to the main path and have no constraint, so level generation cannot fail here.

d) Write spawned Room Tiles to level layout:

i) All spawned Room Tiles until now were only referenced inside the level layout, but now the algorithm takes all their TileBase Tiles and writes them into the output buffer.

ii) At the same time, all substructures' TileBase Tiles inside the Room Tiles are written to the output buffer as well.

5) Calculate Start and Exit locations:

a) The algorithm goes through all the Room Tiles and calculates the Start and Exit positions based on the position of the Start/Exit in the Level layout.

6) Try to complete level using A*:

a) An A* algorithm now goes through the whole level and tries to complete it. Any tile which is not strictly air is treated as a wall to make sure there is always a path to exit.

b) If no path from Start to Exit is found then this step fails and triggers a rebuild of the whole level.

7) Place Start and Exit Game Objects:

a) All previous steps happened on the background thread to prevent lagging the UI/UX. Now we switch to the main thread and place Start/Exit Game Objects in their correct locations in the level.

8) Update Tilemap And Border:

a) Now in the main thread the algorithm places Unity Tiles inside the Tilemap.

b) The Border Generator now places tiles around the level which act as a border and moves border sprites into position which simulate an infinite border around the level.

9) Generate Game Objects:

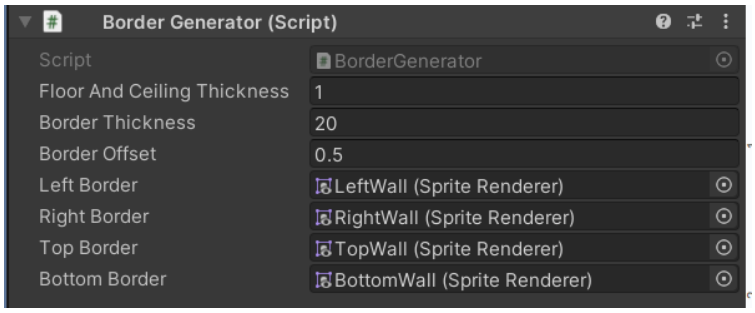
a) Now on the main thread all of the Game Objects inside Rooms and Substructures are spawn in their correct locations.

10) Level Generated event is raised

a) The LevelGenerated event is raised, so any subscriber can now act with full level generated.

Border Generator

Border generator creates a border around the level using Tiles and big sprites which act as an infinite repeating wall. The border generator accepts a Level Style which changes the border Tiles, sprite, and material.



Settings

Floor and Ceiling Thickness: The thickness in tiles around the level. Recommended to keep at 1 since it impacts performance and the tiles should be indestructible anyway.

Border Thickness: The thickness of the sprites which fake an infinite wall. Has to be big enough to prevent the camera from looking outside.

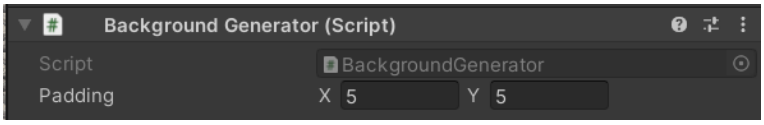
Border Offset: The overlap for the border sprites and border Tiles. At 0.5 it covers half the tiles which allows for having border logic and other logic on the tile game objects, but seamlessly continue into the infinite wall.

API

UpdateSettings(Tilemap tilemap, LevelStyle levelStyle): Updates target Tilemap and Level Style

GenerateBorder(int minX, int maxX, int minY, int maxY): Generates background to surround the dimensions given.

Background Generator



Creates a background using tileable sprite which stretches across the level's dimensions. The sprite and color are determined using Level Style.

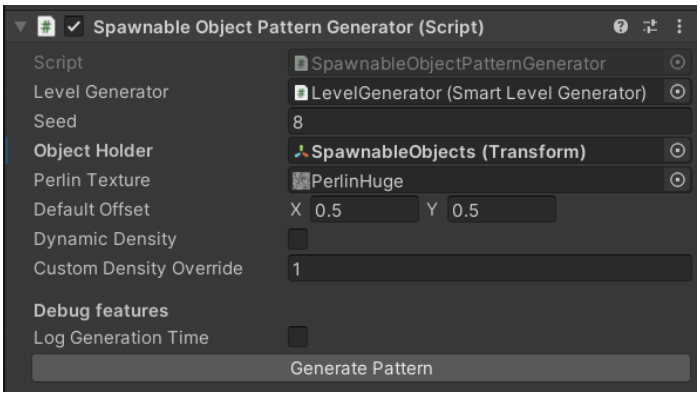
Settings

Padding: This is how much the background will overextend the generated level size.

API

GenerateBackground(LevelStyle levelStyle, Vector2Int tilemapSize): Generates the background based on the size of the Tilemap and Level Style provided.

Spawnable Object Pattern Generator



Spawns objects based on pattern matching. In the example there are corner shadow objects which are spawned using a rule which matches a corner between two tiles. The rule can also be flipped horizontally and vertically which effectively covers all possible corners in the level.

Settings

Level Generator: This object subscribes to the Level Generated event in Level Generator to start the spawning cycle.

Seed: Integer seed used for generating the level. Each seed has the same layout on the same level. -1 means random seed.

Object Holder: Transform which will be the parent of all spawned objects.

Perlin Texture: Perlin texture used for random spawning.

Default Offset: Offset for spawning objects. 0.5 is the default offset when using Unity Tilemap because Unity Tile position is defined from (0,0), but the center of that tile is 0.5.

Dynamic density: Toggles if dynamic density is used

Custom Density Override: Multiplier for how many objects can spawn.

Log Generation Time: Logs how many milliseconds each step took.

Procedure

1) Any previously spawned objects are disabled using the pooling system.
2) The level Tilemap is cached for faster access later in a smaller data structure.
3) Dynamic density is calculated.
4) The level is parsed through all Spawnable Object patterns in the Level Style.
    a) Each pattern is Initialized.
    b) Based on settings the pattern is iterated through the map several times with different flipping in X and Y direction. This can be only once if no flipping is enabled.
        i) For each position in the level the pattern is matched against the tiles in the level. If the tiles have Game Objects then it is considered a Block, it is also a Block if outside the bounds. Tiles which are null or don't have a Game Object are Air.
        ii) The pattern is validated per position using its own rule against the Air/Block tiles.
        iii) If pattern is matched well then the position is saved as a valid location in a list.
    c) All spawnable objects are attempted to spawn in the valid locations.
        i) Each spawnable object initialized a random perlin noise texture offset which will be used for generation.
        ii) The valid locations are shuffled and spawning begins through all of them in random order.
        iii) The perlin texture is read with the random offset and then a validation happens. The validation happens when the perlin exceeds the spawn chance of the object. Tuning this value creates a natural looking environment.
        iv) If the spawning succeeds the position might be removed from the pool of valid locations for other objects based on the pattern settings.
            (1) If spawning on edges is disabled then the object won't spawn if touching map edges. Also the spawning fails if the spawning is outside of level area.
            (2) The spawning can fail randomly based on random chance.
            (3) Spawning the object means getting the object from the pool. This can fail if the pool does not have enough items. Dynamic density can increase the pool of objects which can spawn.
            (4) The object is then placed in the correct location with both fixed and random location/rotation offsets. Then it is enabled.

API

GenerateWithLevelGeneratorNextFrame(): Will generate all objects next frame using patterns from level style, Tilemap, and seed taken from referenced Level Generator.

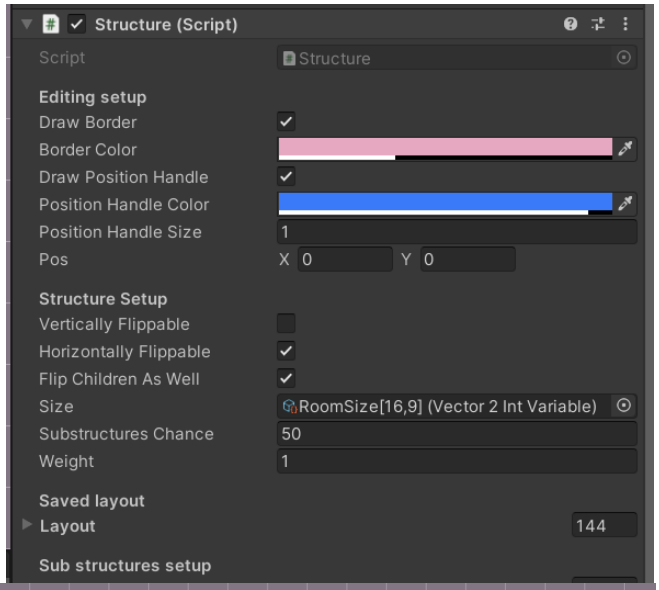
GenerateForMap(Tilemap map, LevelStyle levelStyle, int seed = -1): Generates objects immediately using defined Tilemap, LevelStyle, and seed.

Unity Editor Buttons

Generate Pattern: Generates objects in the current level using Level Style. This does not work very well in Editor mode and spawn very less objects than expected. In play mode it works perfectly.

Room components overview

Structure



A structure is a collection of Unity Tiles which fit inside the given dimensions. The structure can also house substructures which must be placed as Transform children and can also store Game Objects. The structure is used both as a spawnable chunk inside a Room and as a data structure which holds Room data.

Settings

Draw Border: Draw a border around the structure to indicate that the user cannot draw Tiles outside. As shown in the image below.

Draw Position Handle: When a structure is inside a room (another structure) then it is considered a substructure. To position this substructure inside the parent structure a handle is presented which allows for drag and dropping of tiles.

Pos: The position inside its parent substructure.

Vertically Flippable: If the structure can be spawned vertically flipped during level generation. Mostly used for Rooms.

Horizontally Flippable: If the structure can be spawned horizontally flipped during level generation. Mostly used for Rooms.

Flip Children As Well: Should child Game Objects and substructures be flipped (their X/Y scale component) when this structure is flipped during Level Generation?

Size: The size of the substructure. For easy setup and less errors this is a Vector2 Variable Scriptable Object, so all Rooms and substructures of the same kind have the same size.

Substructures Chance: The chance a substructure will spawn when this is spawned. Currently only used for Rooms.

Weight: When this is a substructure with what weight should it spawn since only one substructure will spawn.

API

Vector2Int Size: Dimensions of this structure

bool SkipDrawingBorderForFrame: This is set by a selected child substructure to prevent overdrawing of borders.

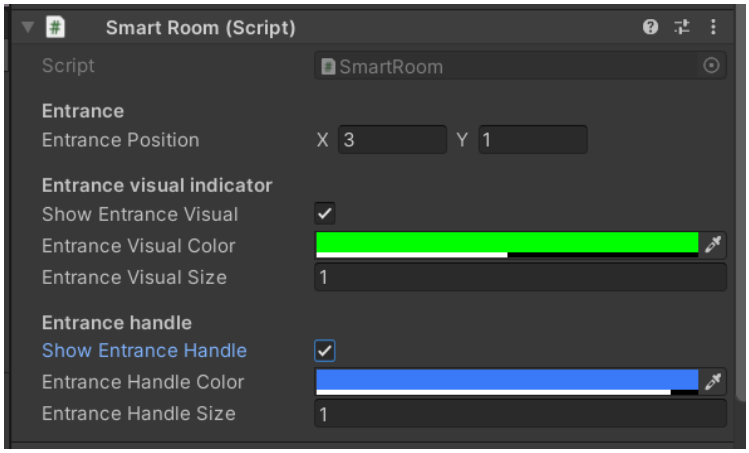
float Weight: The weight for spawning this structure as a substructure.

bool IsRoot: Returns if it is not a substructure.

Vector2Int ParentStructureSize: Gets a size of the parent structure. Return zero if IsRoot.

StructureData GetStructureData(): Gets structure data. Used in level generation.

Smart Room



This component is used in a standard room. It defines a room which has an entrance. All Room Tiles used in level generation for the main path have an entrance set up as a safety measure. The entrance is usually only used on Start and Exit Room Tiles.

Settings

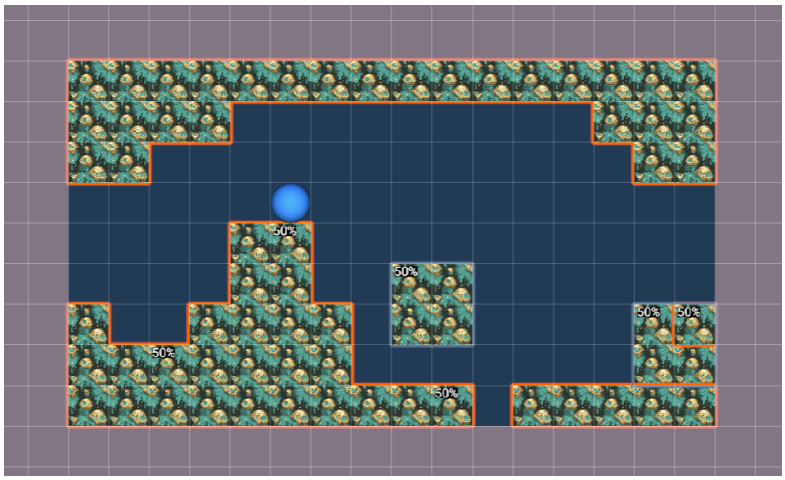
Entrance Position: The position of the entrance in the Room which can be used as Start or Exit. It is measured from the bottom left corner in Tile coordinates.

Show Entrance Visual: This boolean toggles whether to show the entrance in the Room using a Gizmo.

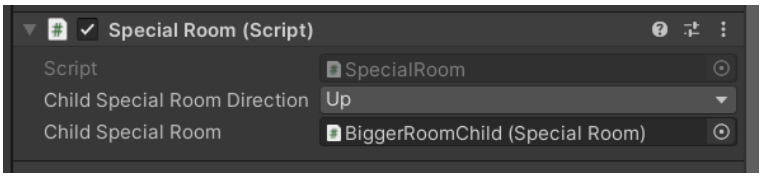
Show Entrance Handle: This boolean toggles whether to show the entrance movable Gizmo which can be dragged to adjust position of the entrance.

API

RoomData GetRoomData(): Get the Room Data of this Room for level generation.



Special Room



Special rooms are Rooms which can contain a second room alongside itself creating a bigger room. Otherwise they are mostly used for mandatory rooms which are not on the main path. If they are set up to contain a second room then it must be set in the inspector. Otherwise it would be treated as a substructure which could result in errors. Only the main Room is considered to be attached to the rest of the level. The child room might not be.

Settings

Child Special Room Direction: Direction in which to place the second child room.

Child Special Room: The reference to the child special room.

API

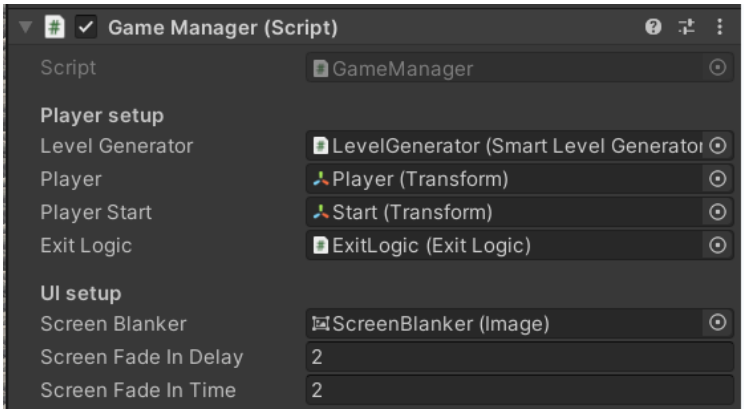
SpecialRoomData GetSpecialRoomData(bool mandatory = false, string id = null, float spawnChance = 100f): This gets special room data. Some parameters should be provided from the LevelStyle.

Managers overview

Game Manager

The Game Manager allows for basic procedural gameplay.

1) It blocks the screen with black color when entering or exiting level, and generates a new level during blanking.
2) It listens to the Exit Logic's Exit event which is fired when the player touches the Exit.
3) It spawns the player at the start of the newly generated level when level generation finishes.



Movement controller

Player

The player controller is a complex component which handles player input using a state machine. Each state is its own component which is attached to the player prefab.

1) States: Each state has to be assigned a component which handles that particular state.
2) Level setup: These settings change how the player interacts with the level and the camera follows them.
3) Movement: These settings determine how the player moves. Can be tweaked to suit any gameplay style you want.

For more information on how the player behaves I recommend checking out the Player prefab which is located in Plugins/SmartRooms/MovementController/Prefabs/Player.prefab. It contains many components which are required for correct visual, audio, and movement aspects of the controller.