Edgar

Offline documentation for version 2.0.0

Table of contents:

- Offline documentation
- Quickstart
- Room templates
- Level graphs
- Generator setup
- Level structure and rooms data
- Performance tips
- Dungeon generator
- Post-processing
- (PRO) Platformer generator
- (PRO) Custom input
- FAQ
- Keep prefabs in editor

Offline documentation

Welcome to the offline documentation of Edgar.

For the best experience, please use the online documentation available at https://ondrejnepozitek.github.io/Edgar-Unity/docs/introduction/. Moreover, to make this document shorter, all the example setups and guides are present only in the online version.

If you have any questions, you can contact me at ondra@nepozitek.cz.

Quickstart

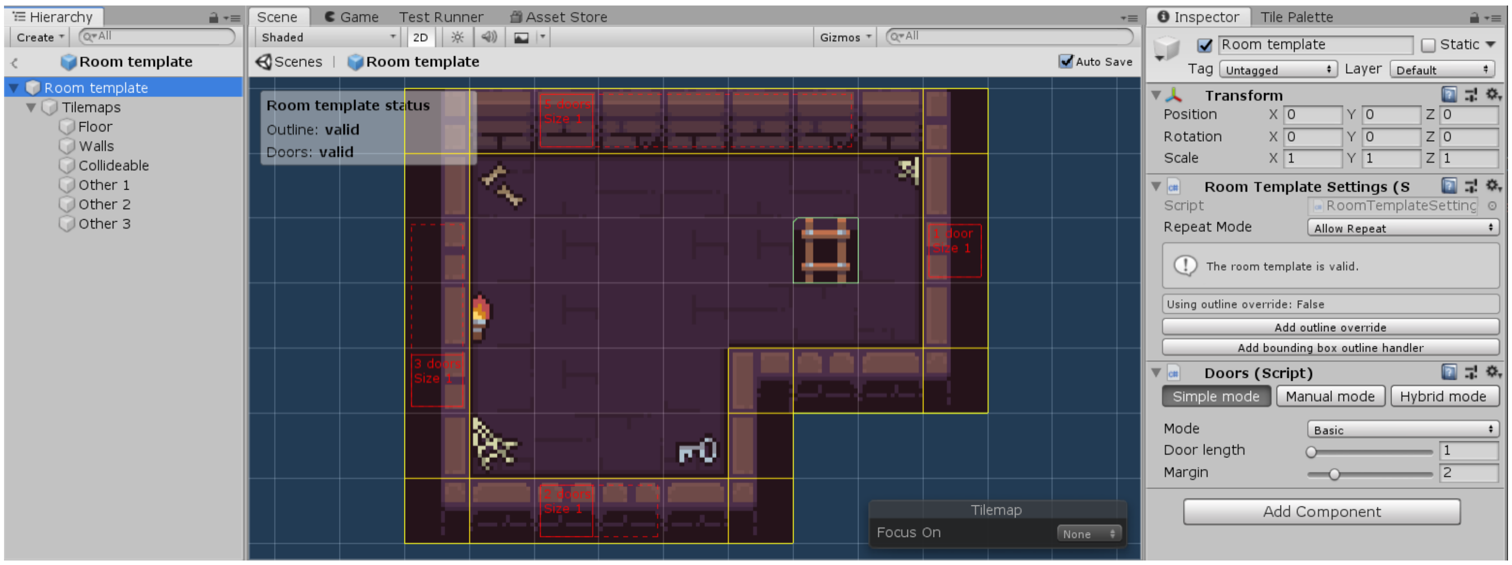
On this page, you should learn the basics needed to generate your first level. This guide is not meant to be comprehensive. Each topic covered in this guide has its own dedicated page containing all the details that were not mentioned here.

> **Note:** If you prefer a video tutorial, you can watch one on youtube.

Room templates

The main documentation page about room templates can be found here.

Room templates are one of the core concepts of the generator. They describe how individual rooms in the dungeon look and how they can be connected to one another.



Room template status
Outline: valid
Doors: valid

Room template
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Room Template Settings (S
Script RoomTemplateSetting
Repeat Mode Allow Repeat
The room template is valid.
Using outline override: False
Add outline override
Add bounding box outline handler
Doors (Script)
Simple mode | Manual mode | Hybrid mode
Mode Basic
Door length 1
Margin 2

Example of a complete room template. Outline of the room template is highlighted with yellow and possible door positions are red.

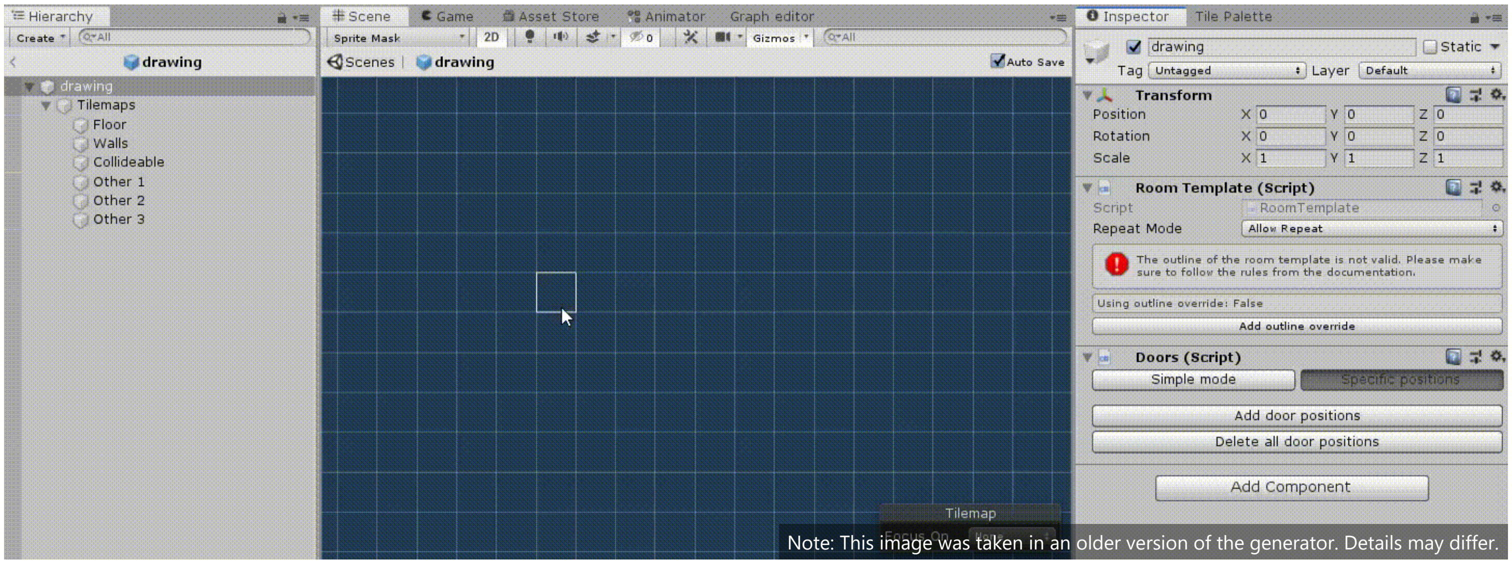
Creating room templates

To create a new room template, you have to:

- navigate to the folder where the room template prefab should be saved
- right click in the *Project window* and choose `Create → Edgar → Dungeon room template`

- (optional) rename the prefab file to anything you want

Designing room templates



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: RoomTemplate

Repeat Mode: Allow Repeat

The outline of the room template is not valid. Please make sure to follow the rules from the documentation.

Using outline override: False

Note: This image was taken in an older version of the generator. Details may differ.

You can use all the available tools (brushes, rule tiles, etc.) to draw room templates

After you open a room template prefab, you will see that there is a *Tilemap* game object that contains multiple tilemap layers like *Walls*, *Floor*, etc. You can use all the available tilemap tools like brushes or rule tiles to design the room template.

VERY IMPORTANT: If you want to change the structure of room templates (add another tilemap layer, add a collider), read the Room template customization guide. But you should probably use the default structure if this is the first time working with Edgar.

Note: Under the hood, the generator has to compute an outline of each room template. Therefore, there are some limitation as to how a room template can look like. For example, a room template cannot consist of two groups of tiles that are not connected by any other tile.

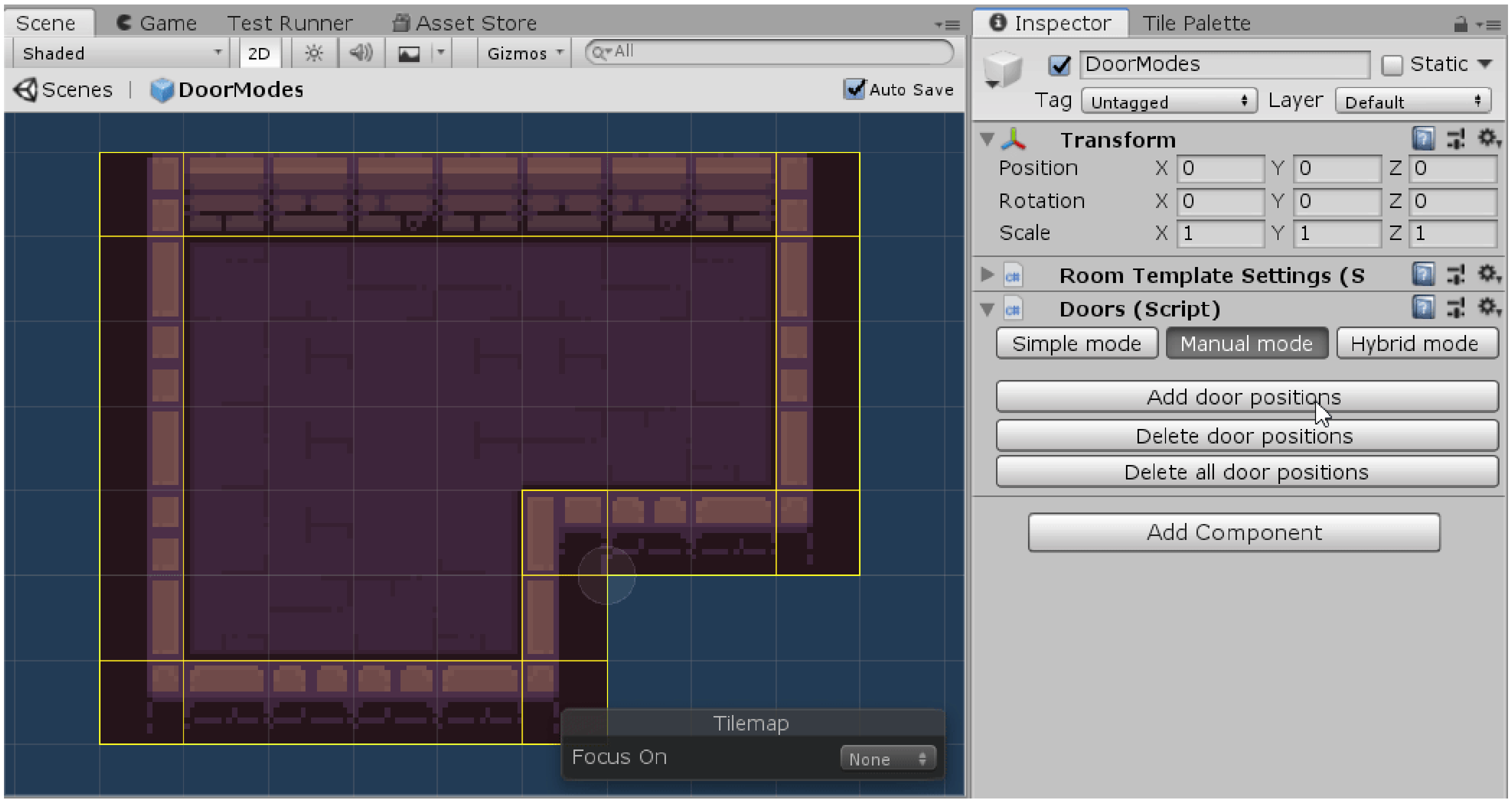
Doors

When you are happy with the visuals of a room template, you can add doors. By specifying door positions, you tell the algorithm how individual room templates can be connected. The most important rule is that **two doors must have the same length in order to be compatible**.

Door positions are added through the **Doors (Grid2D)** component that is attached to the root game object of the room template. There are multiple ways of adding room templates:

- **Simple door mode:** With the simple door mode, you basically say that you do not care too much where the doors are: they can be anywhere on the outline. You just specify the length of the doors (and their margin) and the algorithm automatically adds all the doors.
- **Manual door mode:** With the manual door mode you will have to mark all the door positions manually.

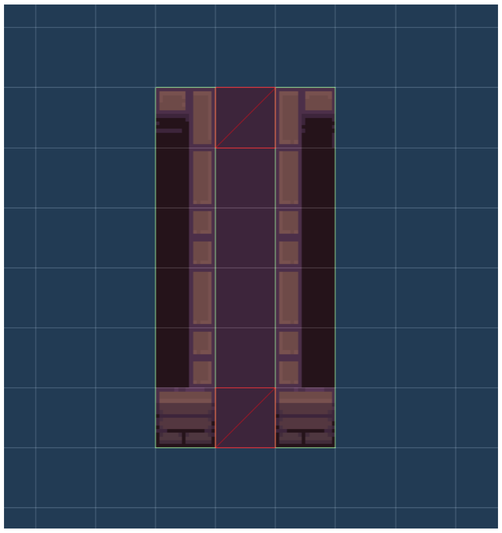
Note: If you are just starting with Edgar, it is recommended using the **Simple door mode** whenever possible. The reason is that the generator might not be able to generate a level if you specify too few door positions. With the simple door mode, there are usually enough door positions for the generator to be fast enough.

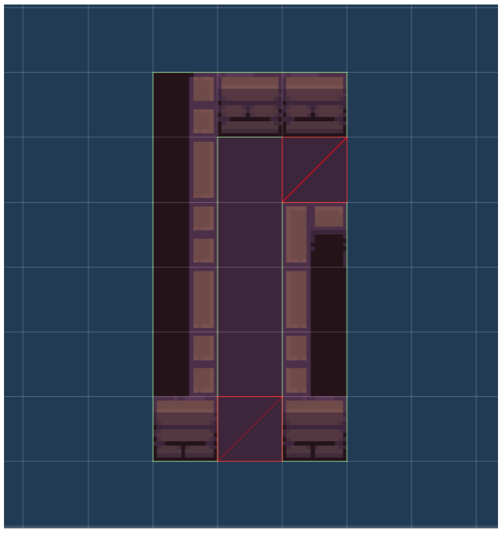


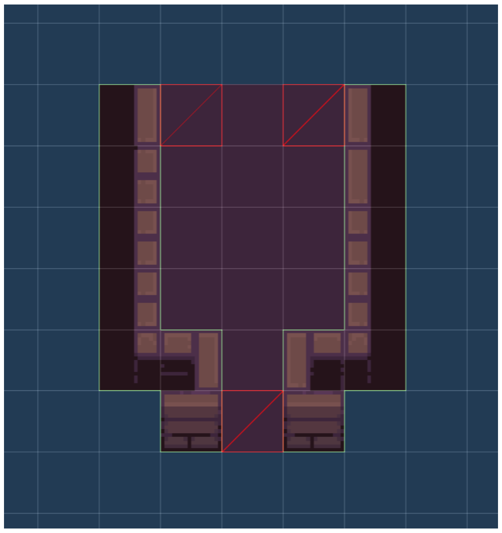
How to add doors with the manual door mode. Click on the first tile of the door and then drag the cursor to the last tile.

Corridors

In order to connect two ordinary room templates in a level, the generator uses so-called corridor room templates. They are created the same way as normal room templates. The only difference is that **they usually have exactly two door positions**.


Recommended - narrow straight corridor


Bad - doors not on opposite sides


Bad - more than 2 door positions

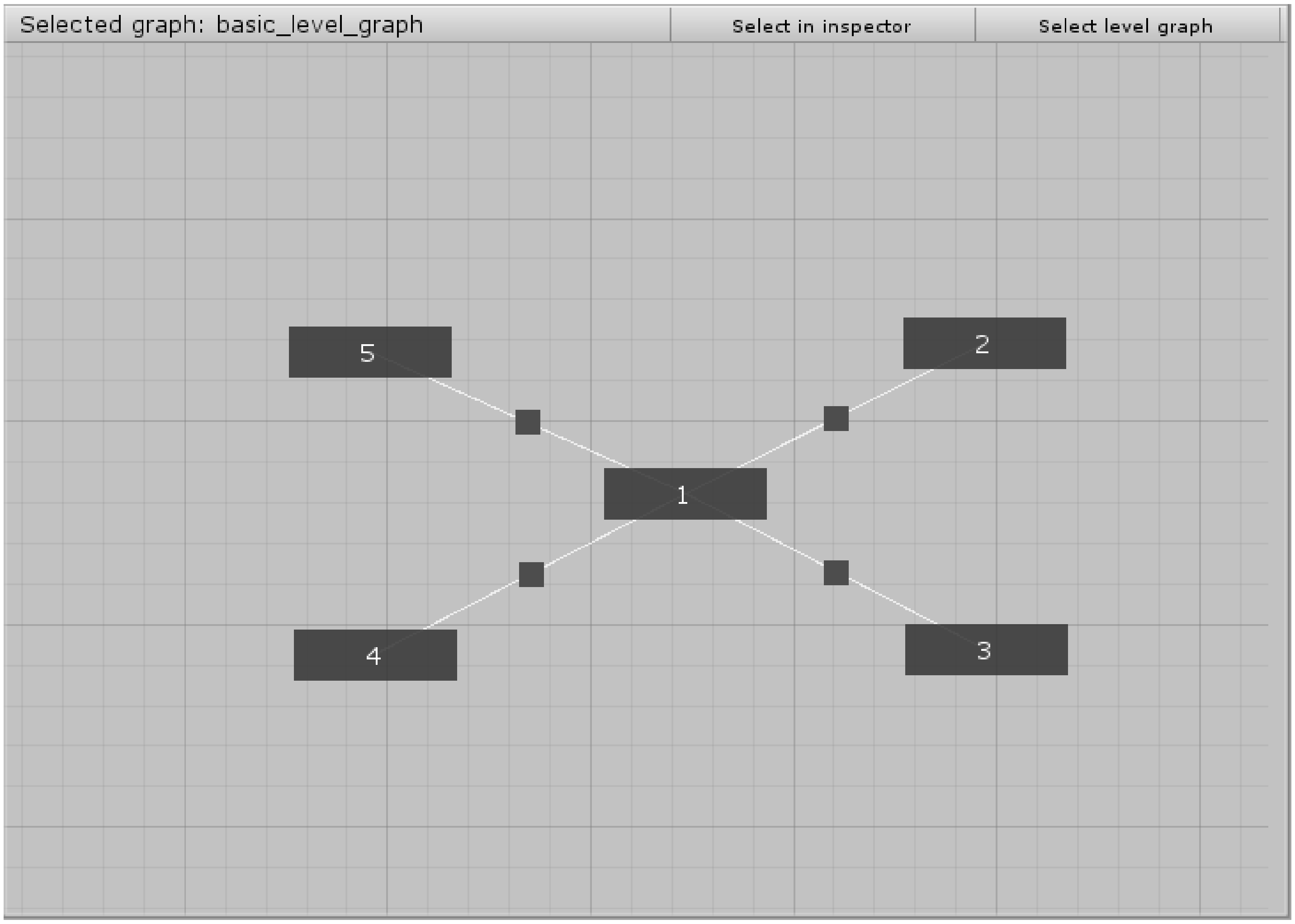
> **Note:** The second and third type of corridor room templates are actually supported by the generator. It is just not recommended due to performance implications.

Level graphs

The main documentation page about level graphs can be found here.

The second most important concept of the generator are so-called level graphs. They describe:

- how many rooms there should be in each generated level
- which rooms should be connected
- which room templates can be used for different rooms in the level



Simple level graph with 5 rooms and 4 room connections. With this level graph, all generated levels will have exactly 5 rooms.

Working with level graphs

First, create a level graph using `Create → Edgar → Level graph`. Then, double-click the created scriptable object to open the editor.

Basic controls:

- *Create room*: double-click on an empty space in the grid
- *Configure room*: double-click on an existing room
- *Delete room*: press *Ctrl* + *Del*, or right-click on a room and select *Delete room*
- *Move room*: left click and then drag around
- *Add connection*: hold *Ctrl* while left-clicking a room and then move the cursor to a different room
- *Delete connection*: right-click on a connection handle and select *Delete connection*

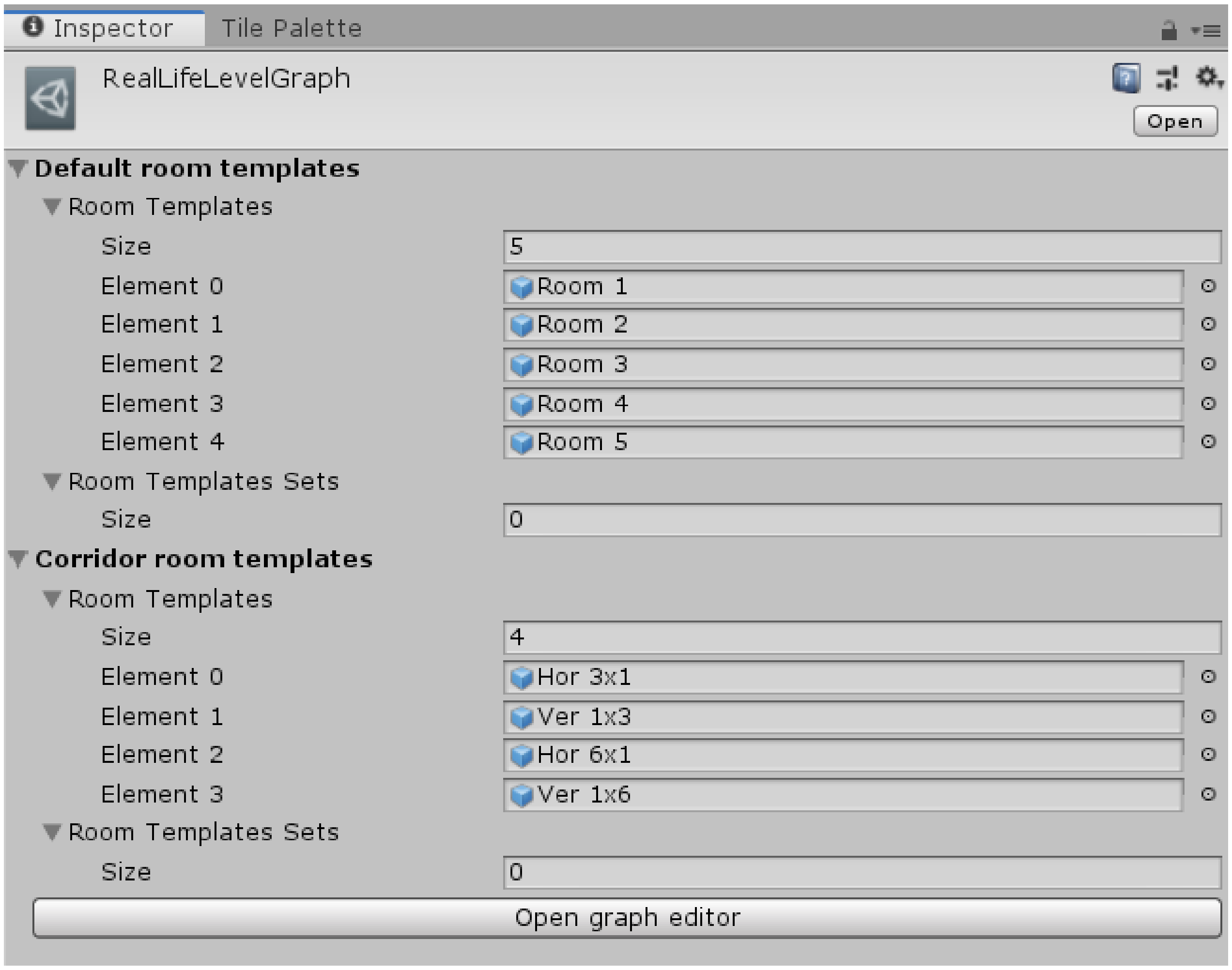
Note: When creating your first level graph, **you should start small**:

- Add only a few rooms (i.e. less than 6).
- Do not add too many connections (i.e. make the graph connected but use at most a single cycle)

It is best to start small and make sure that you understand all the main concepts of the generator. If you start with a really complex level, there is a high chance that you have missed some important detail of the whole setup, and it will be much harder to figure out where the problem is. **This also applies to designing room templates**.

Assign room templates

When you have the level structure ready, it is time to choose which room templates are available for the generator. In the *Level graph* inspector window, you can see 2 sections - *Default room templates* and *Corridor room templates*. Drag and drop your basic room templates to the `Default room templates → Room Templates` field and your corridor room templates to the `Corridor room templates → Room Templates` field.



RealLifeLevelGraph

Default room templates
Room Templates
Size 5
Element 0 Room 1
Element 1 Room 2
Element 2 Room 3
Element 3 Room 4
Element 4 Room 5
Room Templates Sets
Size 0
Corridor room templates
Room Templates
Size 4
Element 0 Hor 3x1
Element 1 Ver 1x3
Element 2 Hor 6x1
Element 3 Ver 1x6
Room Templates Sets
Size 0
Open graph editor

Example of assigned room templates in the level graph inspector.

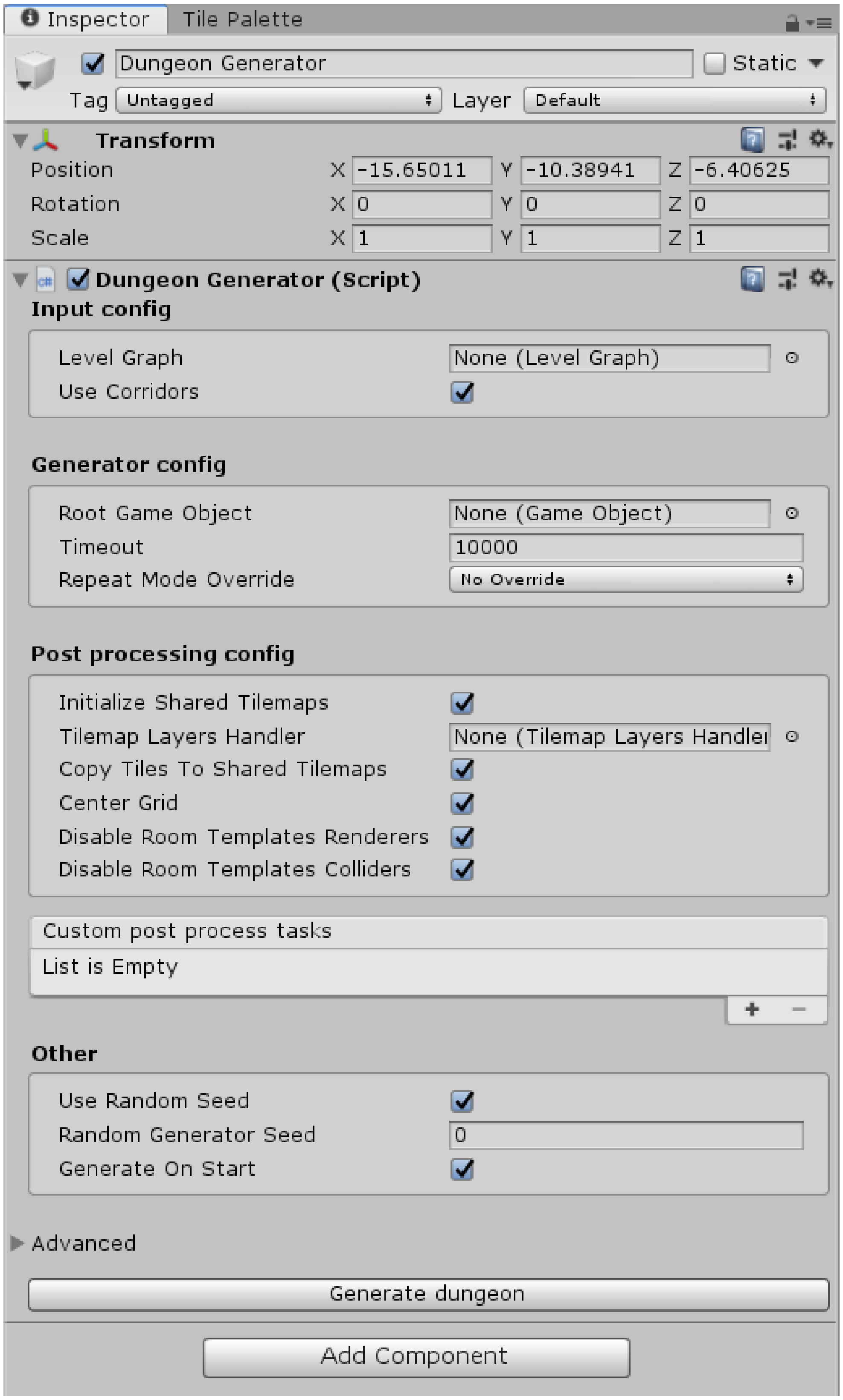
Note: You can also specify that some rooms should have different room templates. To do so, double-click a room in the level graph and add room templates to the *Individual Room Templates* field. If there is at least a single room template assigned to a room, it will be used instead of the default room templates.

Generator setup

The main documentation page about the dungeon generator setup can be found here.

The last step is create an instance of the dungeon generator.

- Add the **Dungeon Generator (Grid2D)** component to any Game Object in the Scene.
- Assign your level graph to the **Level Graph** field.
- Hit the **Generate dungeon** button or enable **Generate on start** and enter play mode



Dungeon Generator ✓ | Static
Tag Untagged | Layer Default

Transform
Position X -15.65011 Y -10.38941 Z -6.40625
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Dungeon Generator (Script) ✓

Input config
Level Graph: None (Level Graph)
Use Corridors: ✓

Generator config
Root Game Object: None (Game Object)
Timeout: 10000
Repeat Mode Override: No Override

Post processing config
Initialize Shared Tilemaps: ✓
Tilemap Layers Handler: None (Tilemap Layers Handler
Copy Tiles To Shared Tilemaps: ✓
Center Grid: ✓
Disable Room Templates Renderers: ✓
Disable Room Templates Colliders: ✓

Custom post process tasks
List is Empty

Other
Use Random Seed: ✓
Random Generator Seed: 0
Generate On Start: ✓

Dungeon generator runner

Troubleshooting

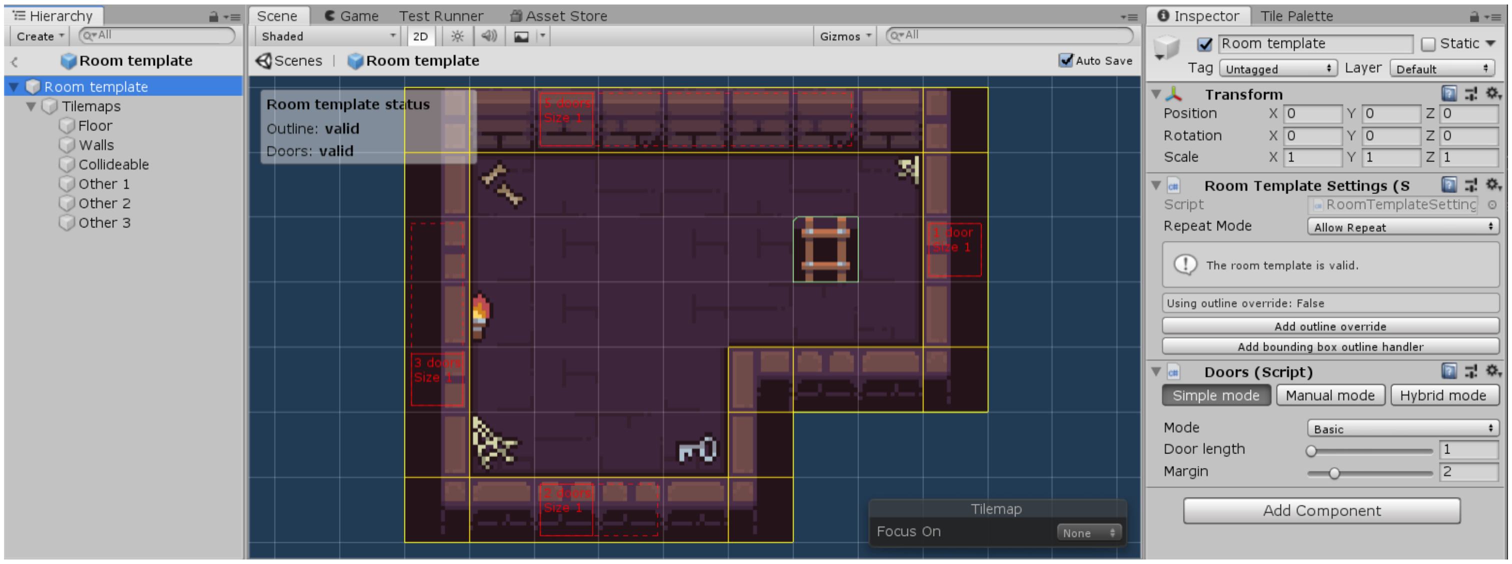
It sometimes happens that the generator will freeze and then output a timeout error to the console. It usually means that you either configured the generator badly or that your setup is just too difficult for the generator to handle.

When that happens, take a look at the console output as there will usually be some diagnostic information that should help you fix the problem. For example, the diagnostics algorithm might tell you that you have too many cycles in your level, and you have to change that.

If you are not able to fix the problem yourself, come join our Discord server where I can usually help you quite quickly.

Room templates

Room templates are one of the main concepts of the generator. They describe how individual rooms in the dungeon look and how they can be connected to one another.



Room template status
Outline: valid
Doors: valid

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

Script RoomTemplateSetting
Repeat Mode Allow Repeat
The room template is valid.
Using outline override: False

Mode Basic
Door length 1
Margin 2

Example of a complete room template. Outline of the room template is highlighted with yellow and possible door positions are red.

Creating room templates

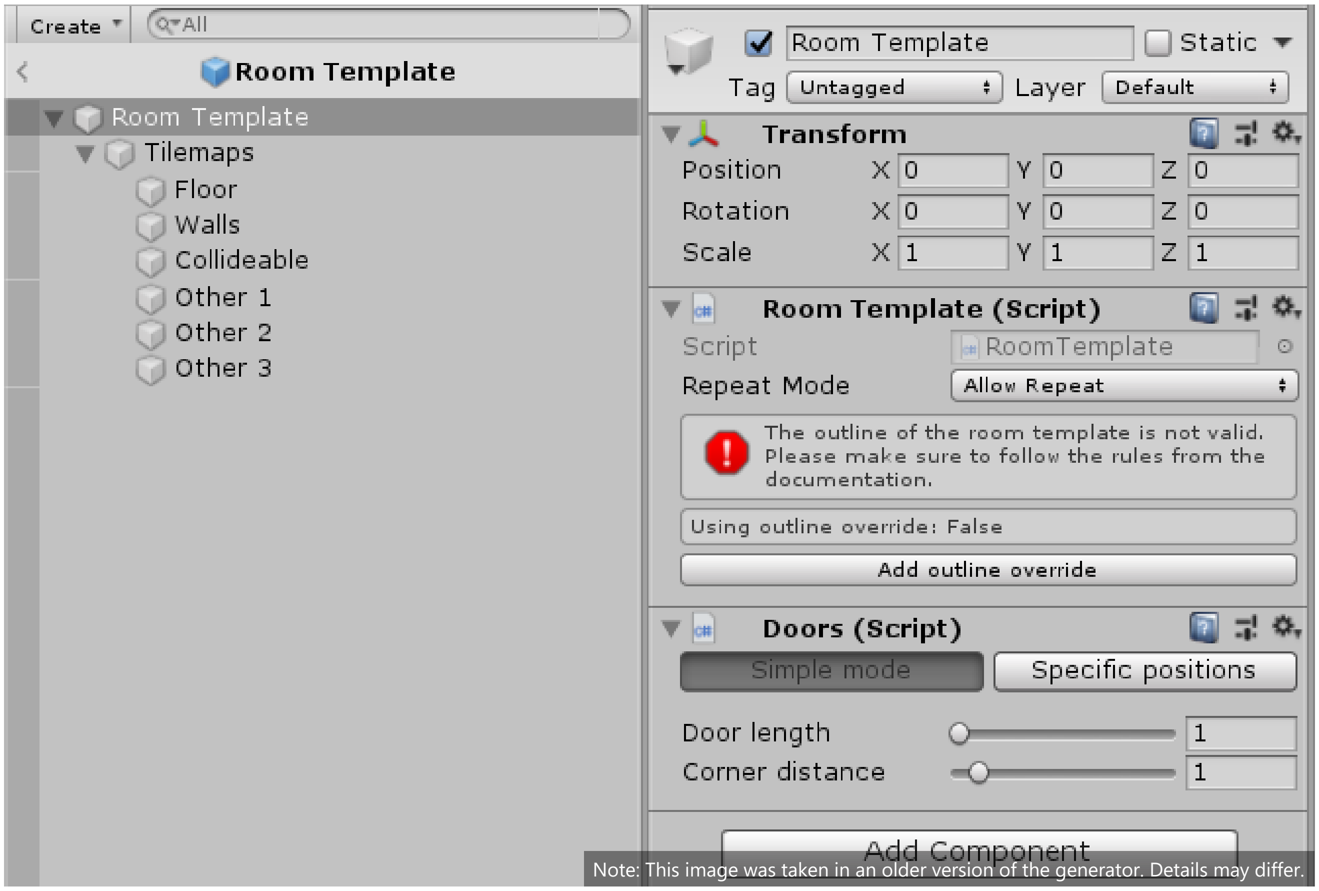
To create a new room template, you have to:

- navigate to the folder where the room template prefab should be saved
- right click in the *Project window* and choose `Create → Edgar → Dungeon room template`
- (optional) rename the prefab file to anything you want

Room template structure

Below you can see the room template structure after the room template is created:

- **Tilemaps** game object that contains several tilemaps attached as children
- **Room Template** script attached to the root game object
- **Doors** script attached to the root game object



The outline of the room template is not valid. Please make sure to follow the rules from the documentation.

Using outline override: False

Add outline override

Note: This image was taken in an older version of the generator. Details may differ.

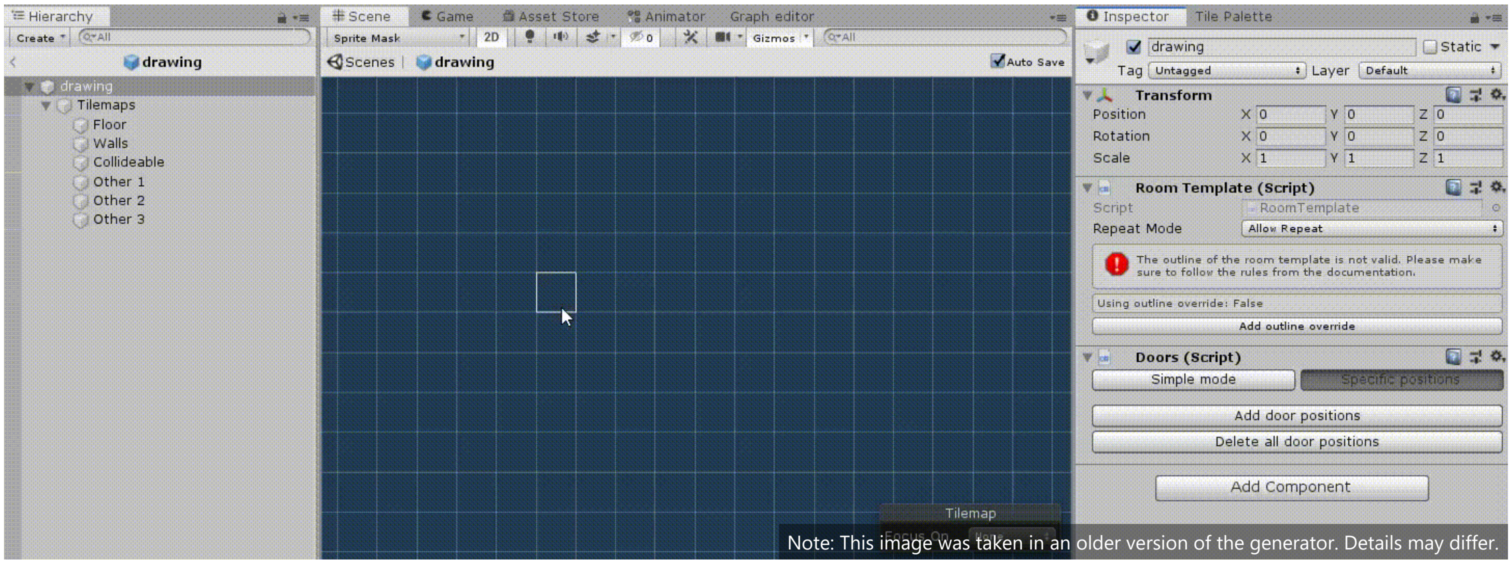
Room template structure

Designing room templates

We will use Unity Tilemaps to design our room templates, so you should be familiar with that. By default, room templates come with several tilemap layers that are children of the *Tilemap* game object:

- **Floor** - order 0
- **Walls** - order 1, with collider
- **Collideable** - order 2, with collider
- **Other 1** - order 3
- **Other 2** - order 4
- **Other 3** - order 5

It is **VERY IMPORTANT** that all the room templates have exactly the same structure of tilemaps because the generator will copy all the tiles from individual room templates to shared tilemaps. If you need a different structure of tilemaps, you can override the default behaviour. See Room template customization.



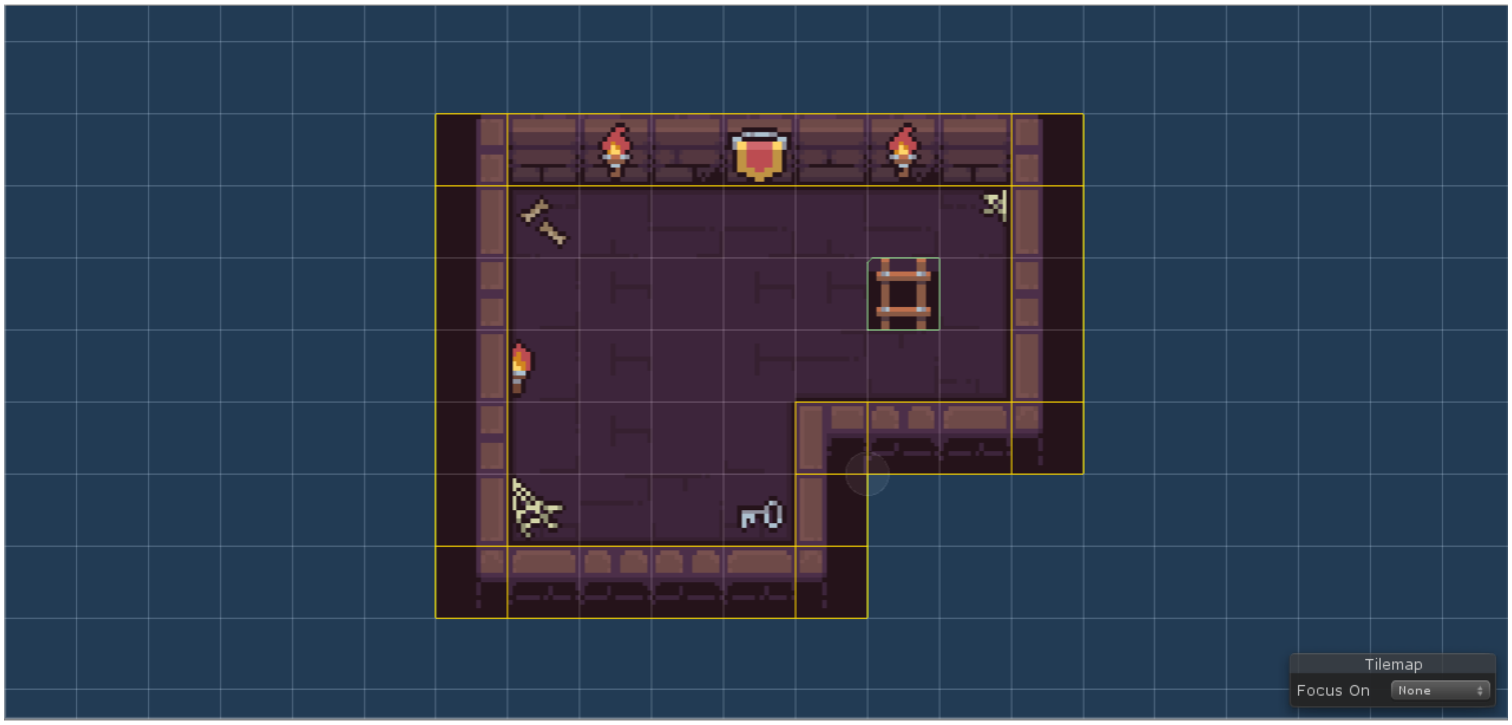
drawing Static
Tag Untagged Layer Default
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Script RoomTemplate
Repeat Mode Allow Repeat
The outline of the room template is not valid. Please make sure to follow the rules from the documentation.
Using outline override: False

Note: This image was taken in an older version of the generator. Details may differ.

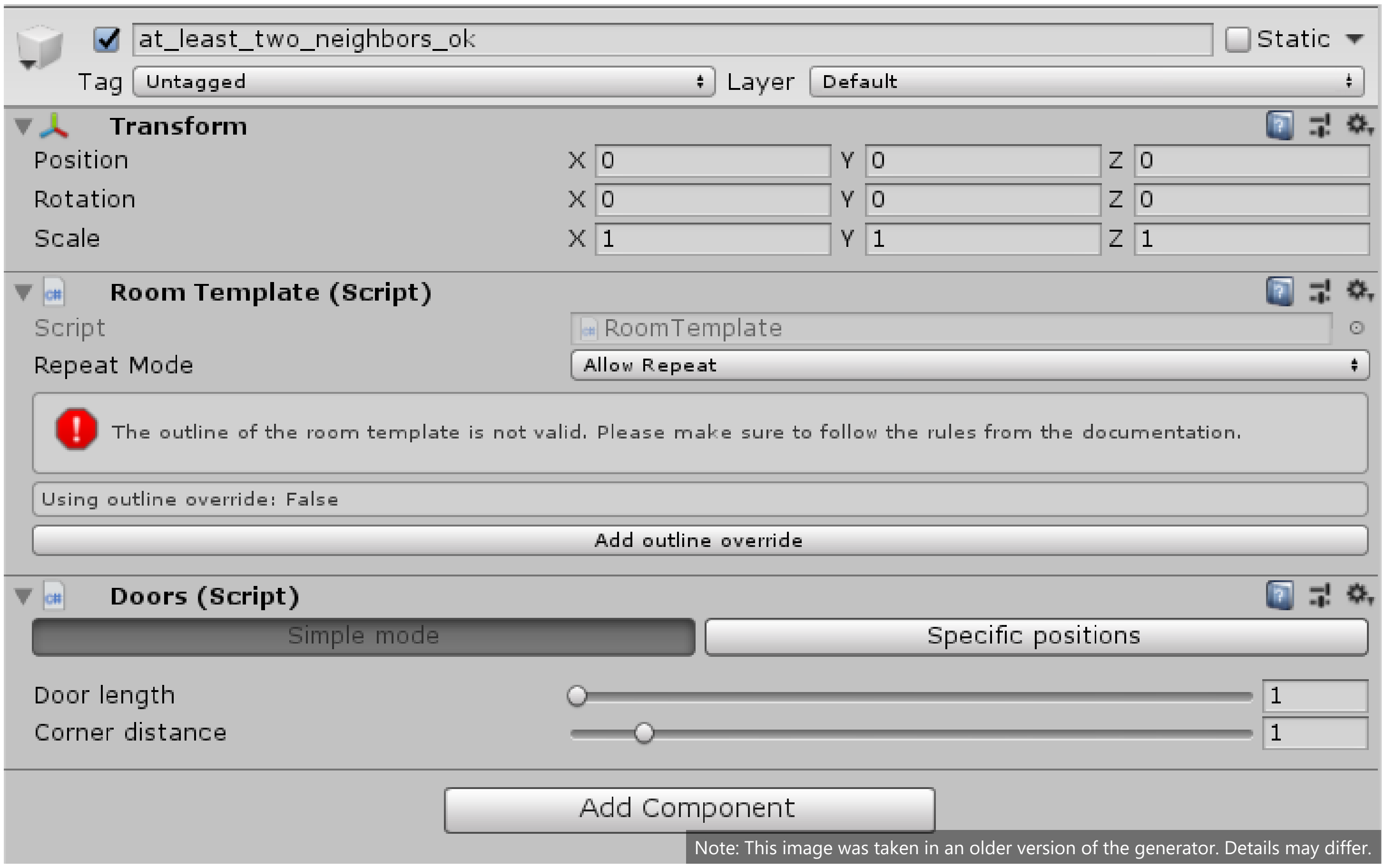
You can use all the available tools (brushes, rule tiles, etc.) to draw room templates

Limitations

The underlying algorithm works with polygons, not tilemaps, tiles and sprites. We are interested in the outlines of individual room templates. However, there are some limitations as to how a room template may look like in order for the algorithm to be able to compute its outline. The goal of this section is to describe which rules we should follow when designing room templates.



The yellow color shows the outline of the room template as it is seen by the generator


at_least_two_neighbors_ok
Static
Tag: Untagged
Layer: Default

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

Room Template (Script)
Script: RoomTemplate
Repeat Mode: Allow Repeat

The outline of the room template is not valid. Please make sure to follow the rules from the documentation.

Using outline override: False

Add outline override

Doors (Script)
Simple mode | Specific positions

Door length 1
Corner distance 1

Add Component

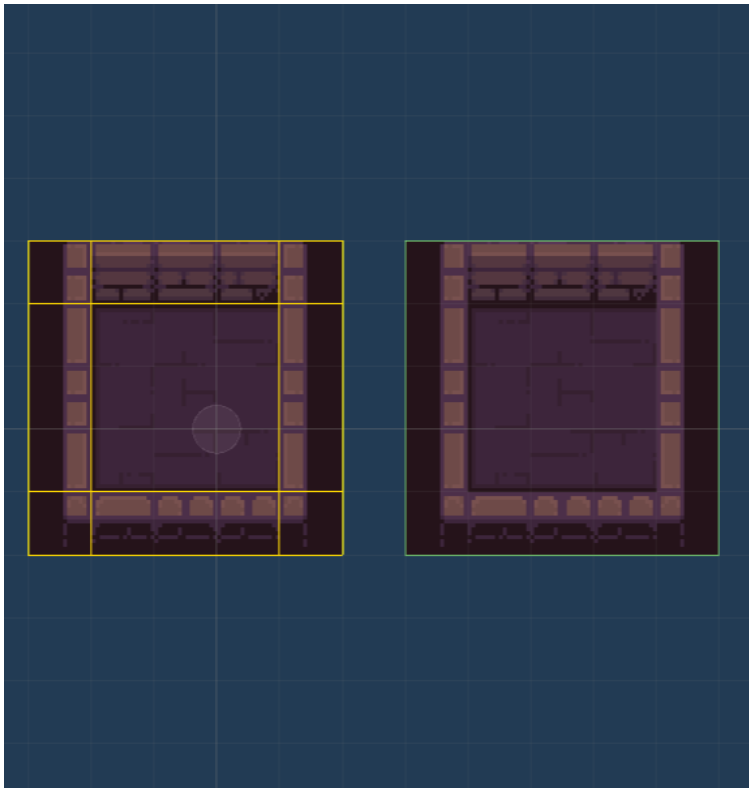
Note: This image was taken in an older version of the generator. Details may differ.

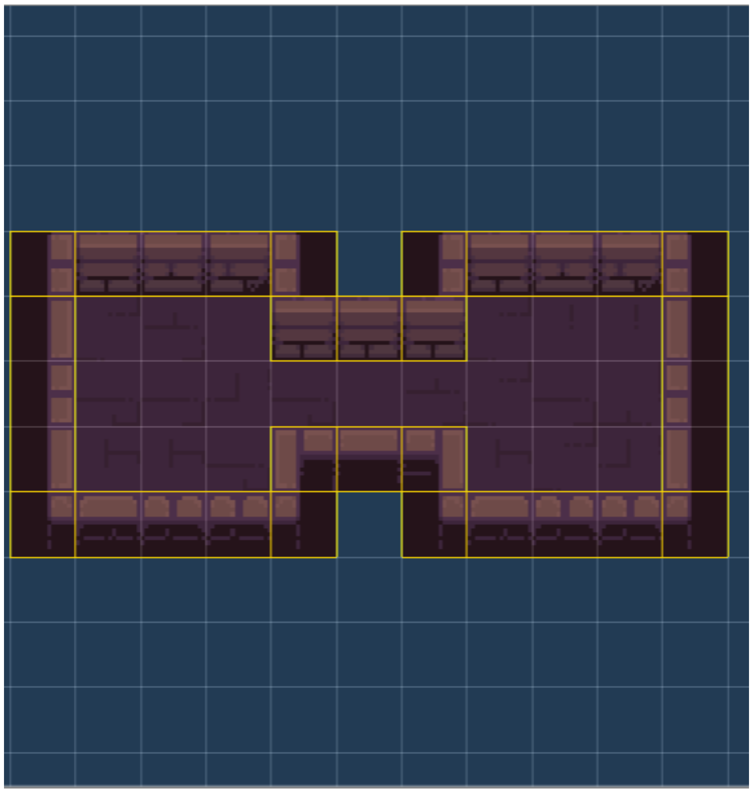
*If the outline is invalid, we can see a warning in the *Room Template* script*

Note: The underlying algorithm does not care about individual tilemaps layers. Instead, it merges all the layers together and then finds all the non-null tiles. Therefore, the outline of the room template will be the same no matter which tilemap layers we use.

One connected component

I will not go into formal definitions. The image below should be self-explanatory.

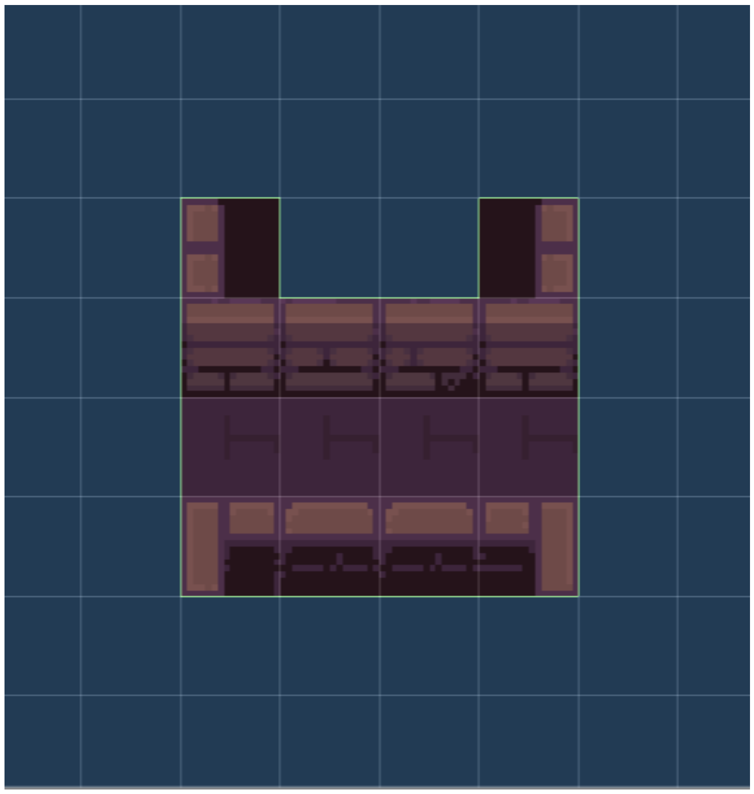

Wrong


Correct

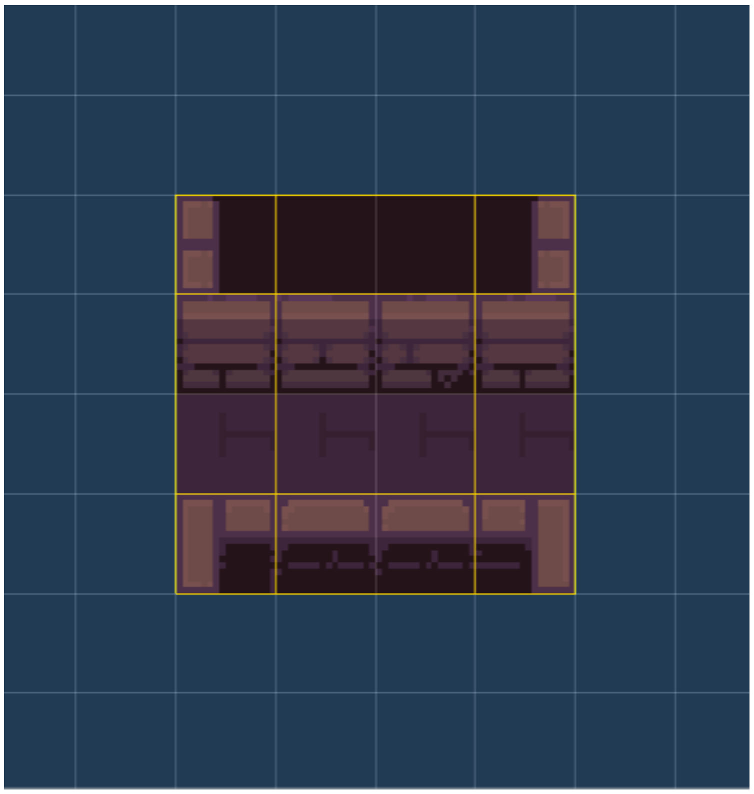
Note: You can see that the algorithm computed some outline (yellow) in the wrong room template. The current implementation stops after any outline is found and does not check whether all tiles are contained in that outline. This will be improved in the future.

Each tile at least two neighbours

Each tile must be connected to at least two neighbouring tiles. In the image below, both tiles in the upper row are connected to only a single neighbour, so the room shape is not valid. If we need these two tiles, we can use **Outline override** that is described in the next section.



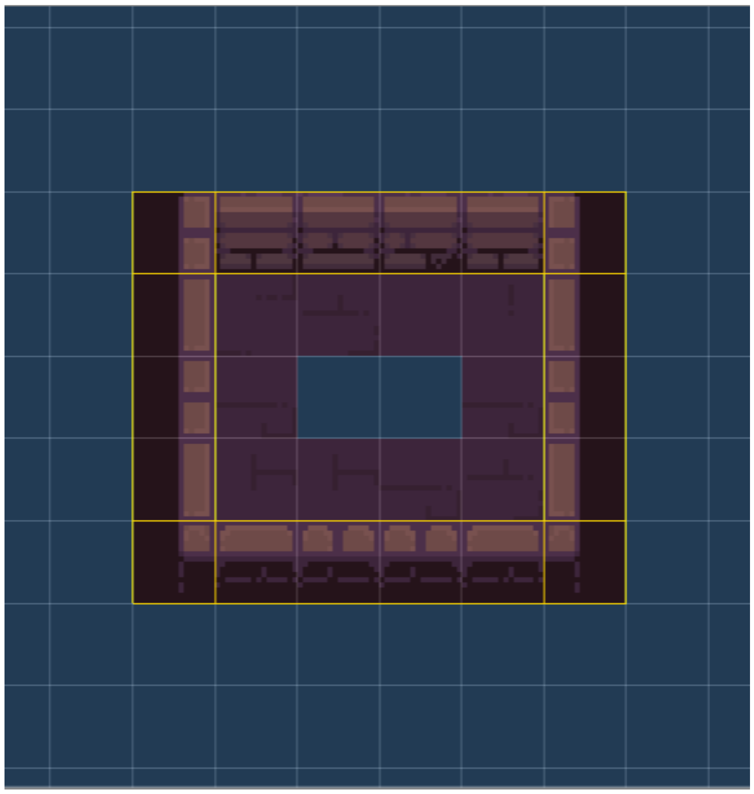
Wrong

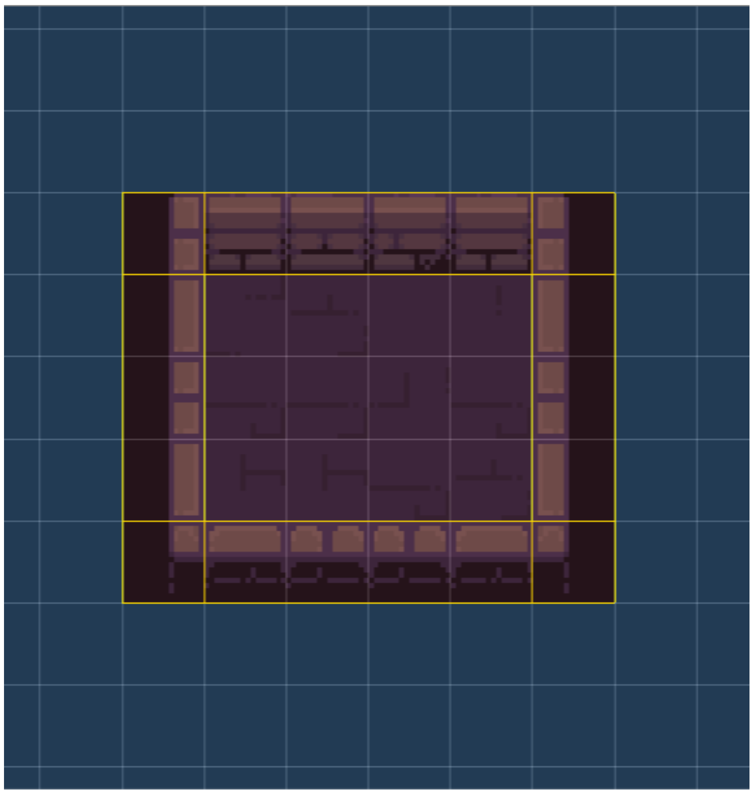


Correct

May contain holes

There may be holes inside the room template.


Correct

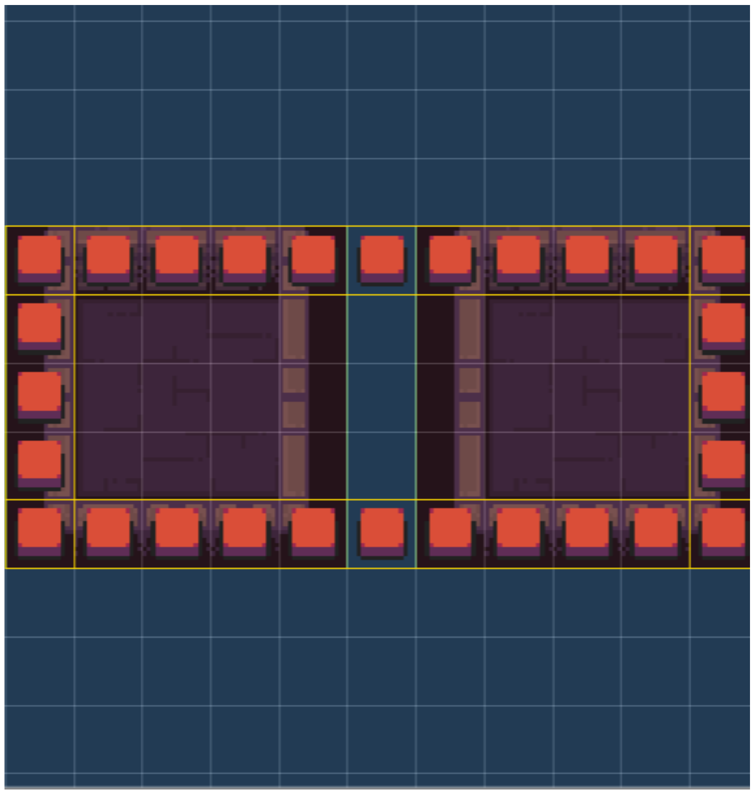

Correct

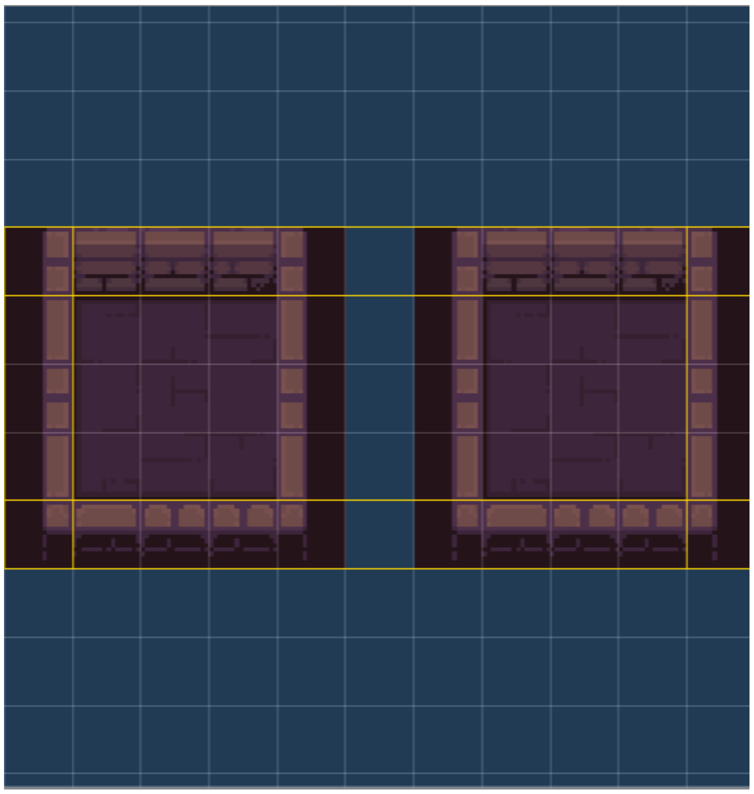
NOTE: This was not possible in the 1.x.x version.

Outline override

If we really need to have a room template whose outline is not valid, we can use the so-called *Outline override*. It can be enabled by clicking the *Add outline override* button in the *Room template* script. As a result, a new tilemap layer called *Outline override* is created. With this functionality enabled, the algorithm ignores all the other layers when computing the outline. Moreover, nothing that is drawn on this layer will be used in the resulting level, so we can use any tiles to draw on this layer.

Note: When we are done with drawing the outline, we can make the layer (game object) inactive so that we can see how the room template actually looks like. However, **we must not destroy the game object**.

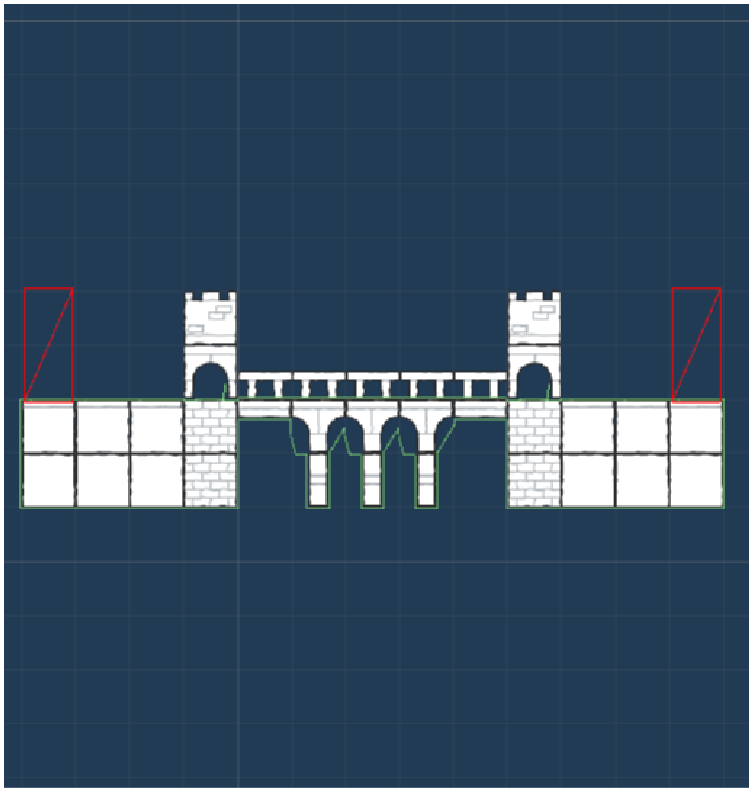

We can use any tiles to draw the outline


If we disable the Outline override game object, we should still see that the outline is overridden

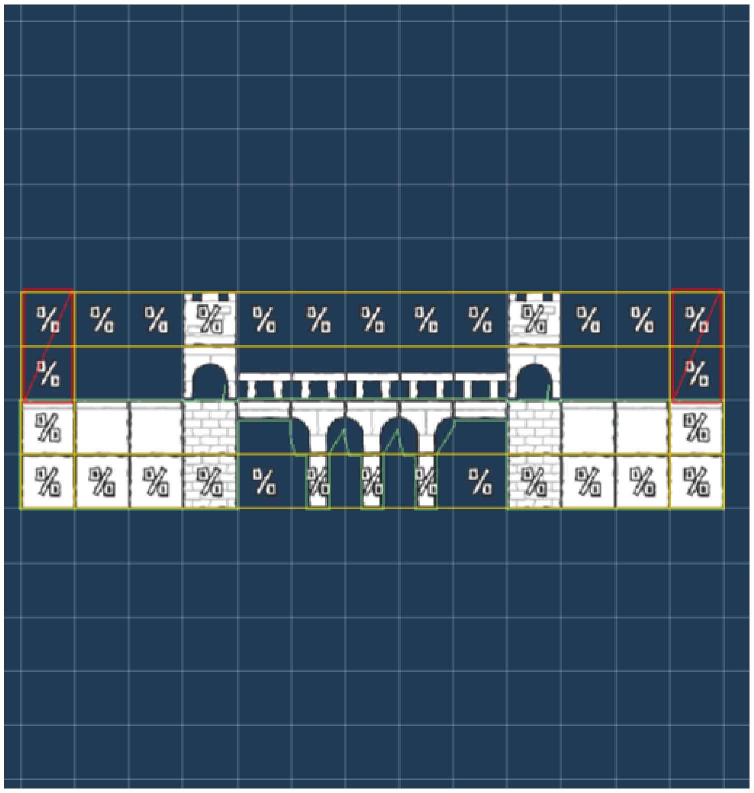
Bounding box outline handler

In some situations, it would be useful to have an outline which looks like the bounding box of all the tiles in the room template. For example, it can be used to handle an outline of some platformer levels (see the images below). It is also possible to add padding to the top of the outline, which is convenient if we need to add doors that are higher than the outline.

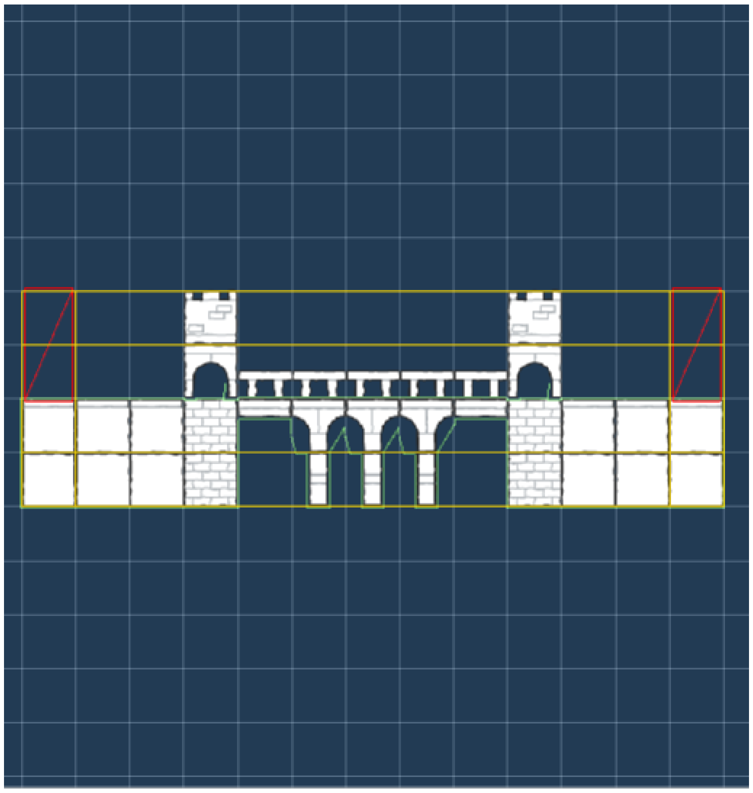
To add the *Bounding box outline handler* click the **Add bounding box outline handler** button in the *Room template* inspector.



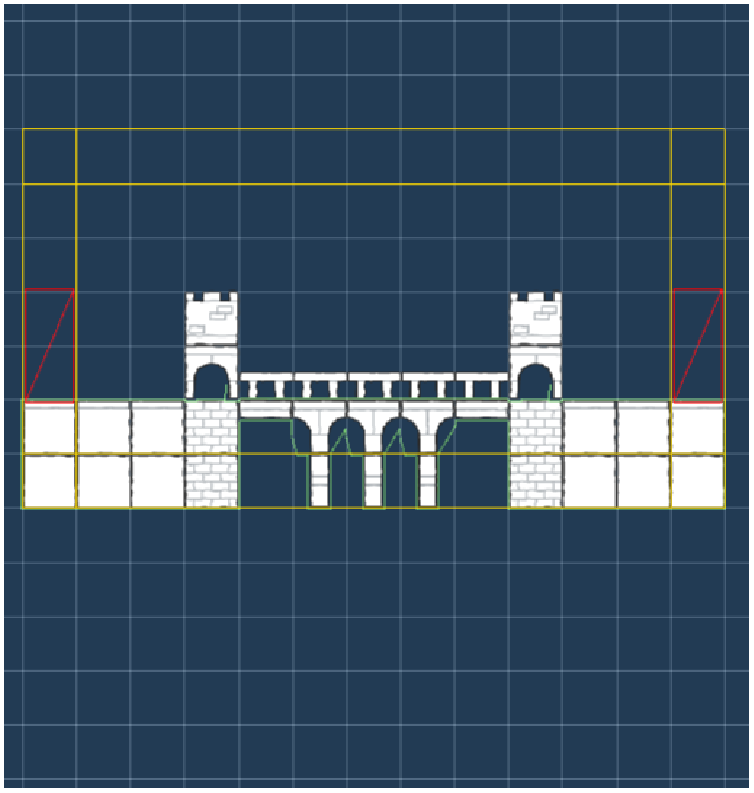
Invalid outline



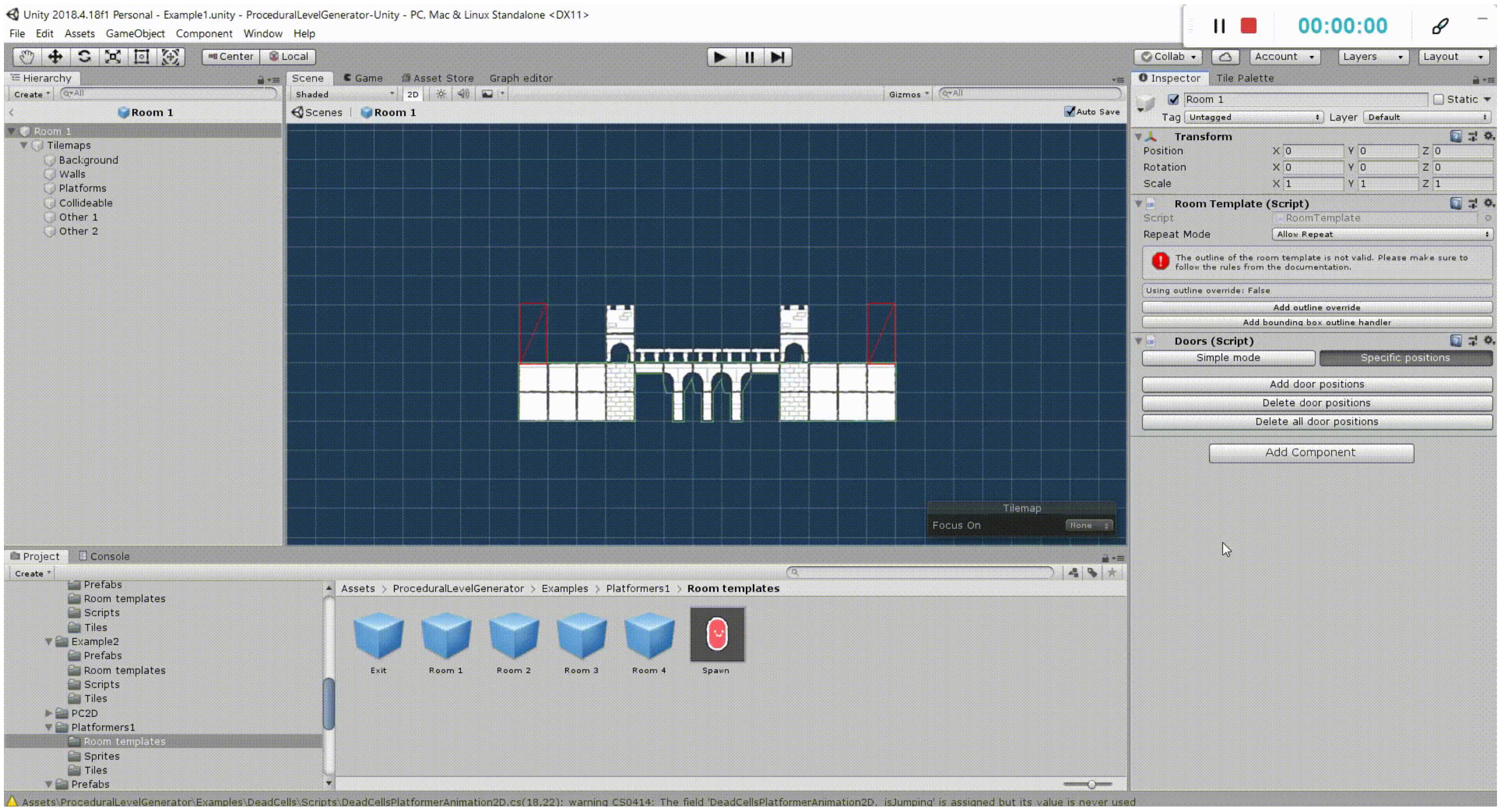
Corrected manually with Outline override



Corrected automatically with Bounding box outline handler



Example of Padding top 3



Unity 2018.4.18f1 Personal - Example1.unity - ProceduralLevelGenerator-Unity - PC, Mac & Linux Standalone <DX11>

Room 1 — Tag: Untagged, Layer: Default

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

Room Template (Script)
Script: RoomTemplate
Repeat Mode: Allow Repeat

The outline of the room template is not valid. Please make sure to follow the rules from the documentation.

Using outline override: False

Doors (Script): Specific positions

Assets > ProceduralLevelGenerator > Examples > Platformers1 > Room templates

```
Assets\ProceduralLevelGenerator\Examples\DeadCells\Scripts\DeadCellsPlatformerAnimation2D.cs(18,22): warning CS0414: The field 'DeadCellsPlatformerAnimation2D._isJumping' is assigned but its value is never used
```

Example bounding box outline usage

Adding doors

When we are happy with how the room template looks, we can add doors. By specifying door positions, we tell the algorithm how can individual room templates be connected.

The algorithm may connect two room templates if:

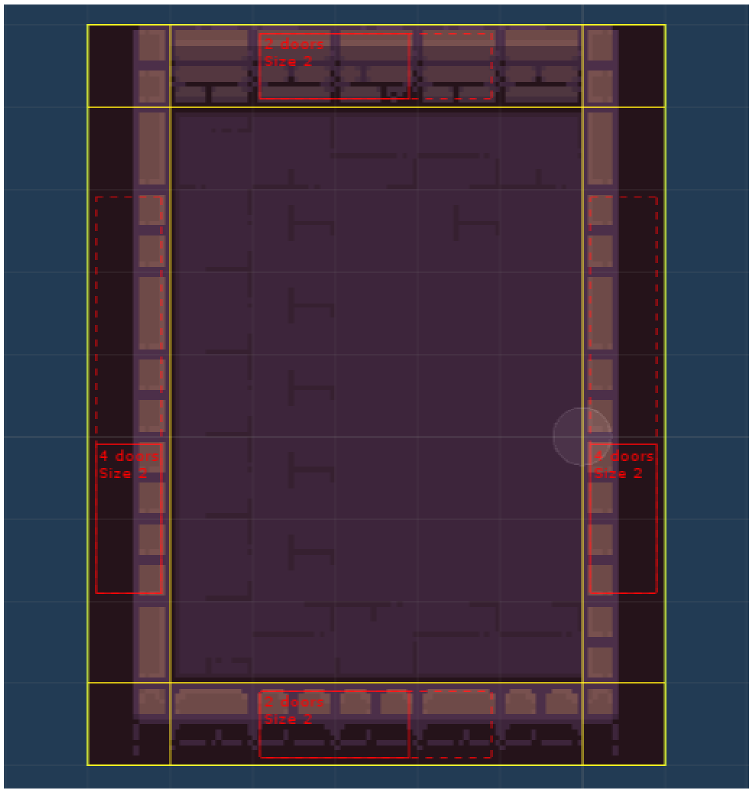
- there exist door positions with the same length
- the two room templates do not overlap after we connect them
  - but they may share tiles on the outlines

Note: In some level generators, if you define *N* doors, it means that the room must be connected to *N* neighbours. That is not the case here! By adding door positions, **you specify where a door can be**. But if there is a room template with 20 possible door positions, the generator might still use this room template for a room that has only a single neighbour. Moreover, a high number of available door position usually means better performance.

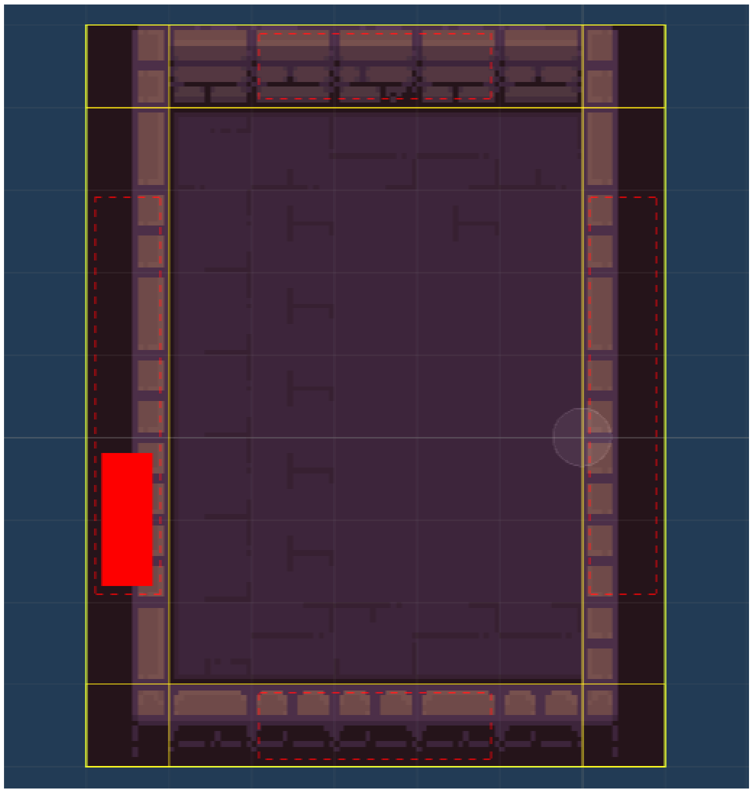
How to interpret door gizmos

Before we start adding doors to our room templates, I think it is important to understand how to read the editor gizmos that represent doors. In the image below (left), we can see an example room template with red rectangles showing the available door positions. The **dashed red rectangles** represent individual **door lines** where a door line is set of all the possible doors inside rectangle. The **solid red rectangles** show the **length of the doors**. In the room template below, all doors are 2 tiles wide. The solid rectangle also contains information about how many door positions there are in the door line.

The GIF on the right shows an animation of all the possible doors positions from the room template on the left. An important thing to understand is that the door positions can overlap, and it is even good for the performance of the generator. The reason for that is that there are more possible door positions to choose from, so the generator finds a valid layout faster.



Example room template


Animation of all the possible door positions

Door modes

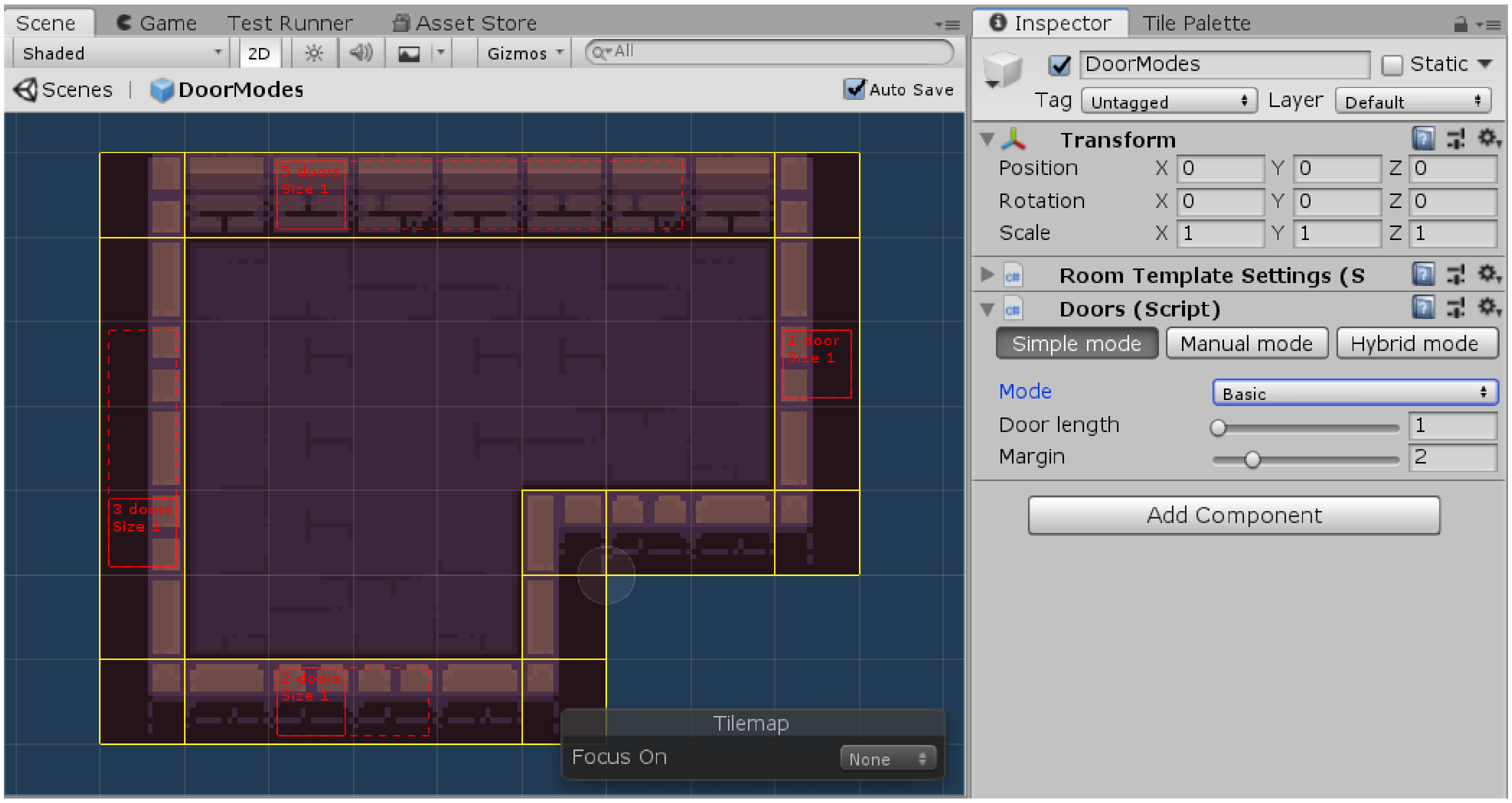
To manipulate with the doors, there must be a `Doors` component attached to the root of the room template prefab.

There are currently three different ways of defining door positions. A universal rule of all the different modes is that all door positions must be on the outline of the corresponding room template.

Simple mode

In the *simple mode*, you specify how wide should all the doors be and the margin, i.e. how far from corners of the room template must the doors be. This door mode is great for when you do not really care where exactly the doors can be. This door mode also usually has the best performance because there are many door positions to choose from.

Below you can see how this door mode looks in the editor.



DoorModes
Tag Untagged Layer Default

Transform

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

Room Template Settings (S

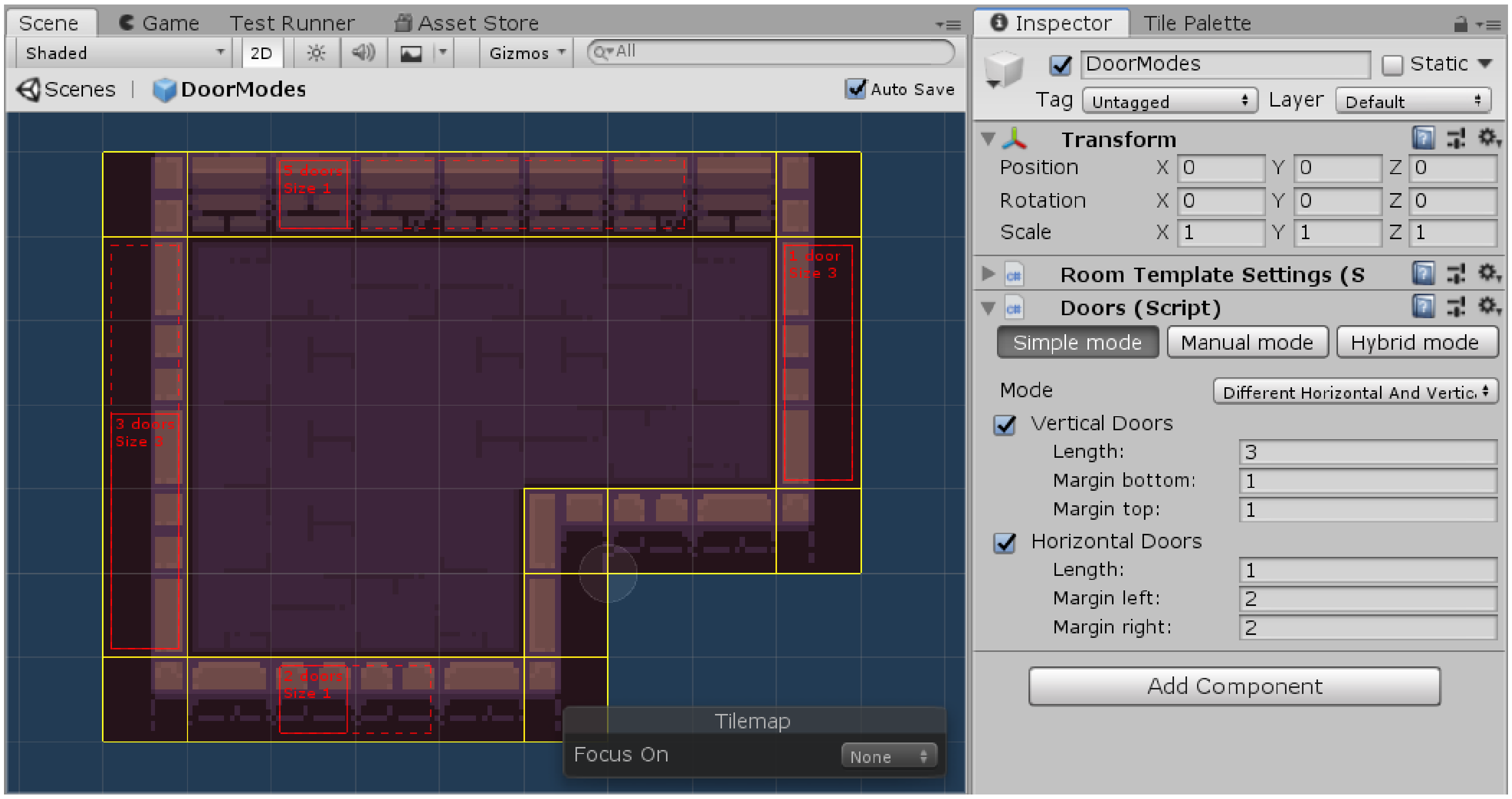
Doors (Script)

Simple mode | Manual mode | Hybrid mode

Mode: Basic
Door length: 1
Margin: 2

Simple door mode - length 1, margin 2

In side-scroller games, there are often different requirements for horizontal and vertical doors. For example, the player might be 3 tiles high but only 1 tile wide, so we would need wider vertical doors. To achieve this, we can change the *Mode* dropdown to `Different Horizontal And Vertical`. With this setting enabled, we can now choose different properties for vertical and horizontal doors. Or we might also disable one type of doors.



DoorModes ☐ Static
Tag Untagged Layer Default

Transform

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

Room Template Settings (S

Doors (Script)

Simple mode | Manual mode | Hybrid mode

Mode: Different Horizontal And Vertic

✓ Vertical Doors
- Length: 3
- Margin bottom: 1
- Margin top: 1

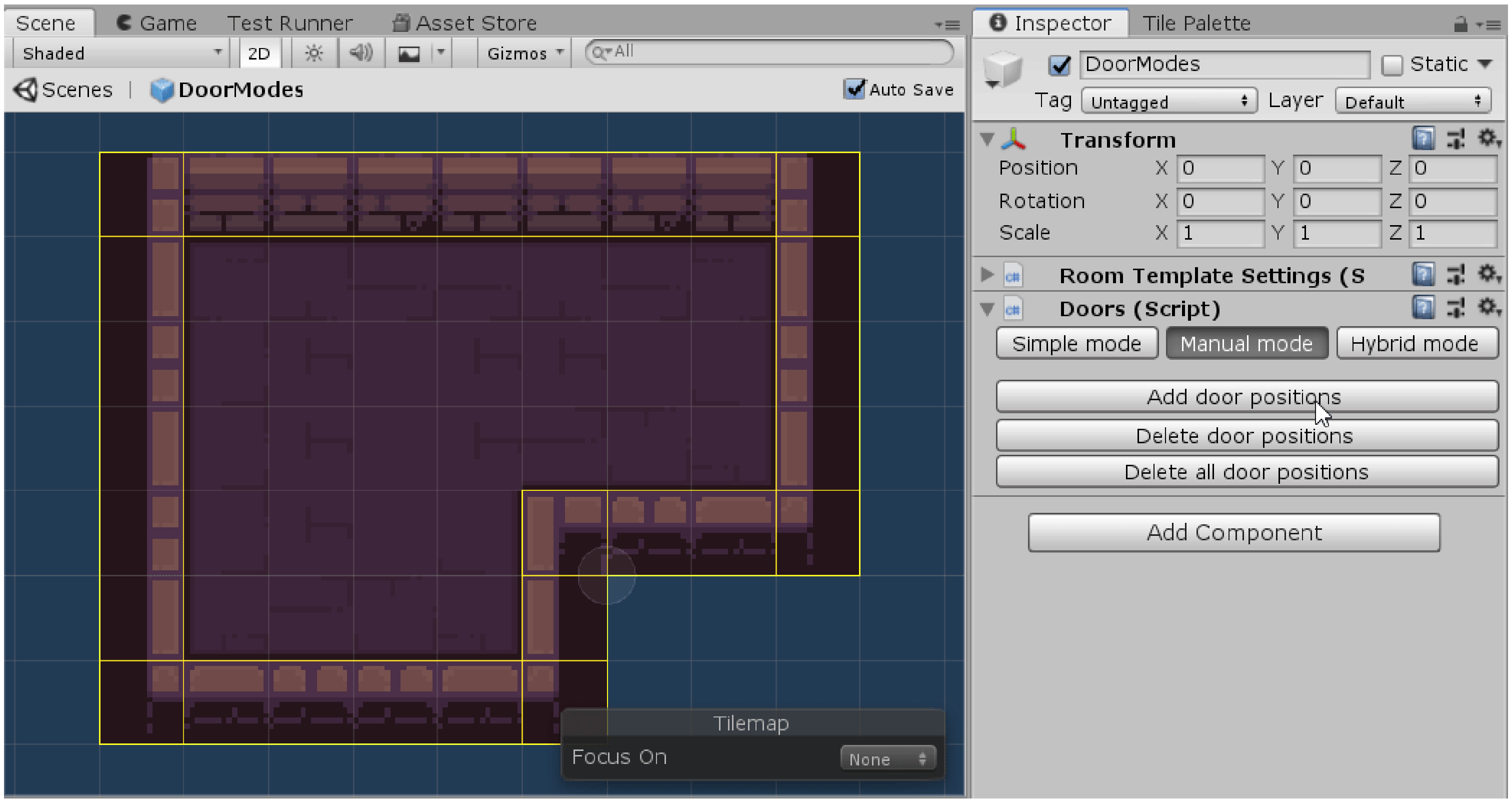
✓ Horizontal Doors
- Length: 1
- Margin left: 2
- Margin right: 2

Simple door mode - different vertical and horizontal doors

Manual mode

In the *manual mode*, you have to manually specify all the door positions of the room template. This door mode is great for when you have only a couple of doors at very specific positions.

To start adding doors, click the *Manual mode* button in the *Doors* script and then click the *Add door positions* button to toggle edit mode. Then you can simply draw door positions by clicking on the first tile of the door and dragging the mouse to the last tile of the door.



Manual mode setup

In the example above, we can see that we can have doors with different lengths - vertical doors are 3 tiles high and horizontal doors are 1 tile wide.

Note: If you accidentally add a door position that you did not want to add, there are two ways of removing doors:

1. Click the *Delete all door positions* button to delete all the door positions.
2. Click the *Delete door positions* button and then click on door positions that should be deleted.

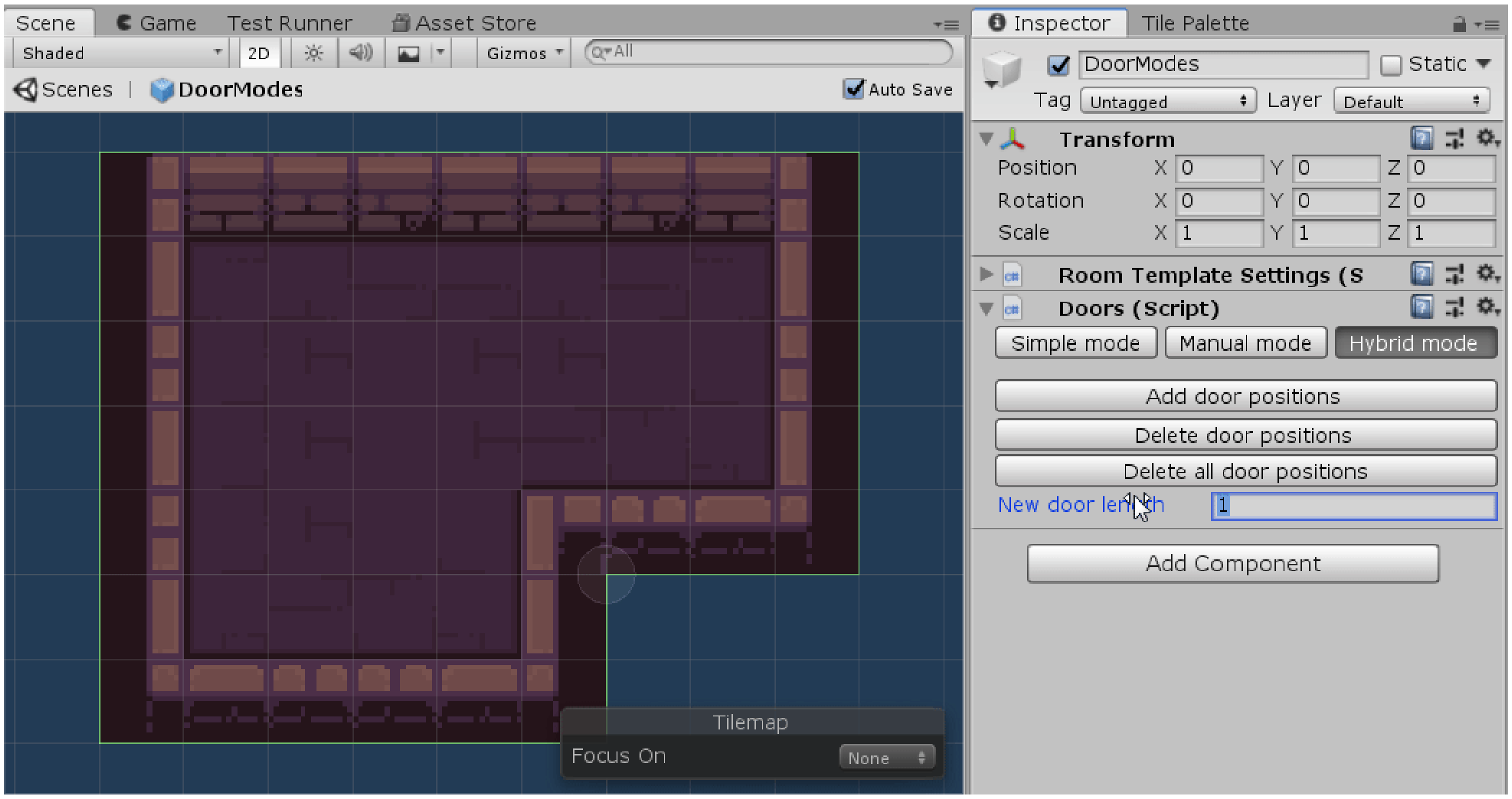
Note: With multiple doors overlapping, the GUI gets quite messy. You should usually use the *hybrid mode* when you have overlapping doors.

Note: The inspector script currently lets you add door positions that are not on the outline of the room shape. It will, however, result in an error when trying to generate a dungeon. It should be improved in the future.

Hybrid mode

The *hybrid mode* is somewhere between the *simple* and *manual* modes. Instead of drawing individual door positions (like in the manual mode), we can draw whole door lines (multiple doors at once).

To start adding doors, click the *Manual mode* button in the *Doors* script and then click the *Add door positions* button to toggle edit mode. **Then you have to configure the length of the doors in the field below.** This is the main difference when compared to the *manual mode*. In the manual mode, the length of doors is determined by the movement of the mouse. But in the hybrid mode, the length of doors is configured in the editor, and the movement of the mouse specifies how many doors there are next to each other.



Hybrid mode setup

Note: The *hybrid mode* is great for when you cannot use the simple mode and the manual mode would require too much time to set up. Also, the hybrid mode also nicely handles **overlapping doors** because the definition of door lines implicitly contains them. Moreover, the hybrid mode also leads to **better performance** (when compared to the manual mode) because it promotes having many doors and the doors are in a format that the generator can easily work with.

(PRO) Door sockets

By default, when the generator computes how can two room templates be connected, it looks for doors with the same length. If you want to have more control over this process, you can use Door sockets.

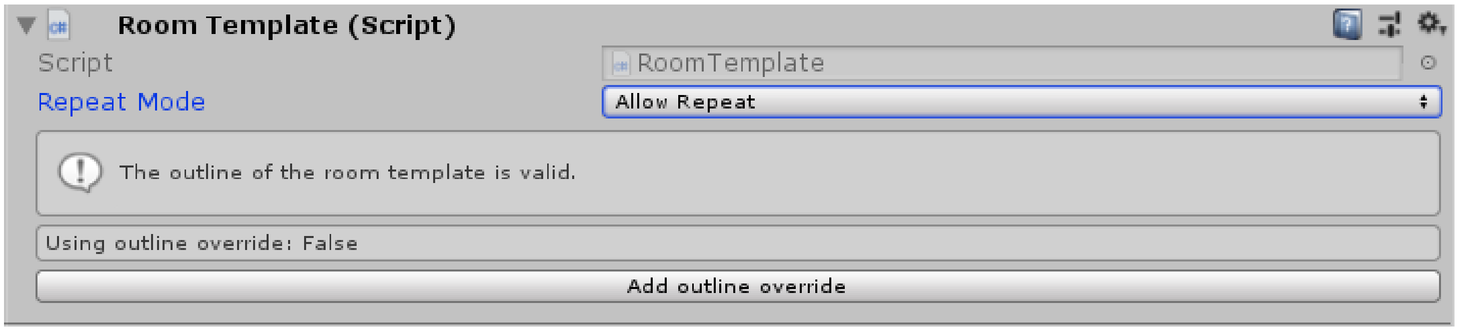
(PRO) Door directions

By default, all doors are undirected, meaning that they can be used both as an entrance or as an exit. With manual door mode, it is possible to configure doors as entrance-only or exit-only. When combined with directed level graphs, it gives you more control over generated levels. See the Directed level graphs guide for more information and examples.

Repeat mode

Each *Room template* script has a field called *Repeat Mode* that is initially set to *Allow Repeat*. Using this field, we can tell the algorithm whether the room template can be used more than once in generated levels. There are the following possibilities:

- **Allow repeat** - The room template may repeat in generated levels.
- **No immediate** - Neighbors of the room template must be different.
- **No repeat** - The room template can be used at most once.


Room Template (Script)
Script RoomTemplate
Repeat Mode Allow Repeat
The outline of the room template is valid.
Using outline override: False

Specific positions mode

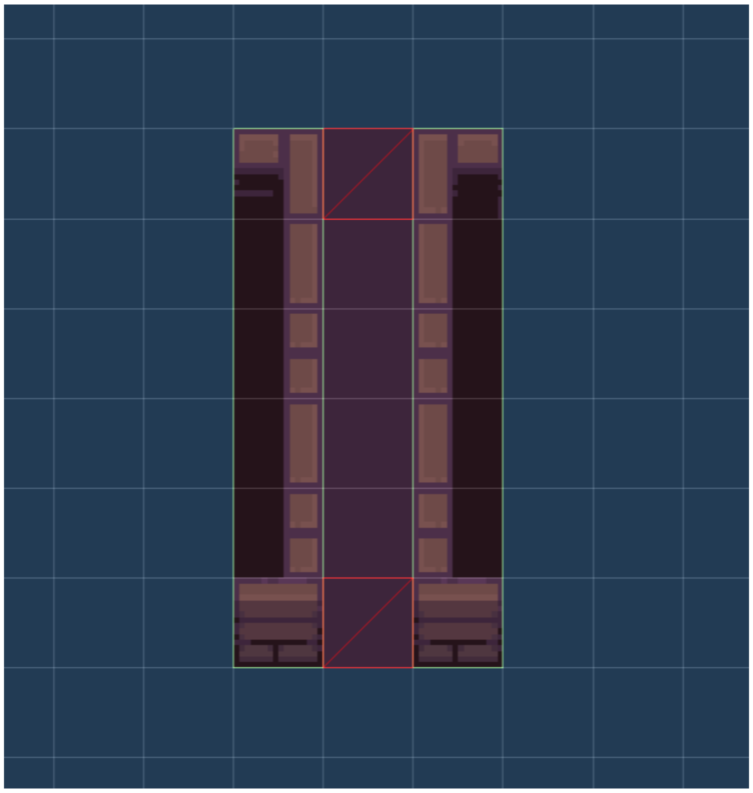
Note: Instead of setting the *Repeat mode* on a per room template basis, you can use global override which is configured directly in the dungeon generator.

Note: If you provide too few room templates, they may repeat in generated levels even if you choose the **No immediate** or **No repeat** options. To make sure that the repeat mode is satisfied, please provide enough room templates to choose from.

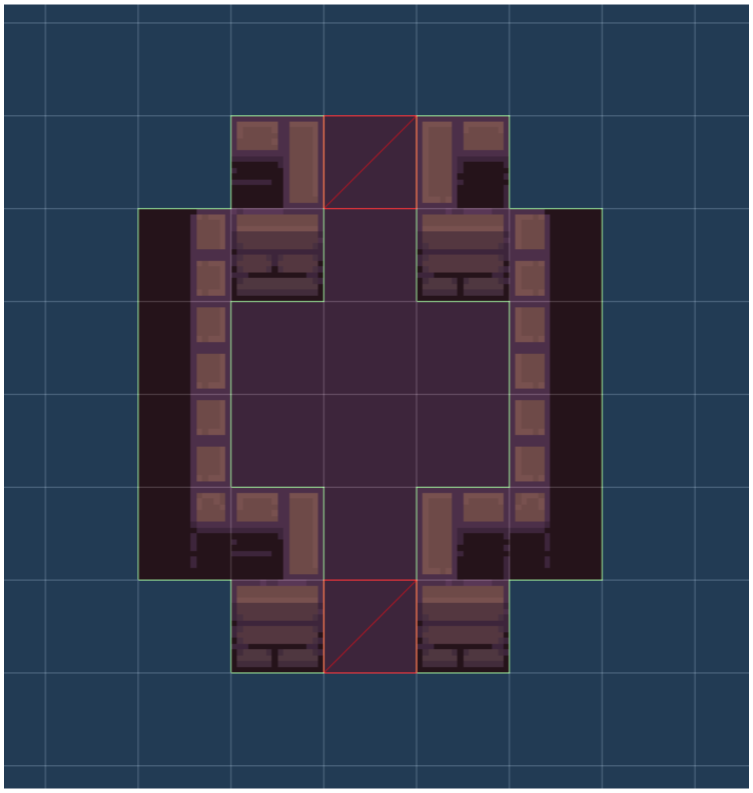
Corridors

The algorithm distinguishes two types of room templates - basic room templates and room templates for corridor rooms. In theory, we can use any room template with at least two doors to act as a corridor room template. **However, to make the algorithm fast, we should follow these recommendations**:

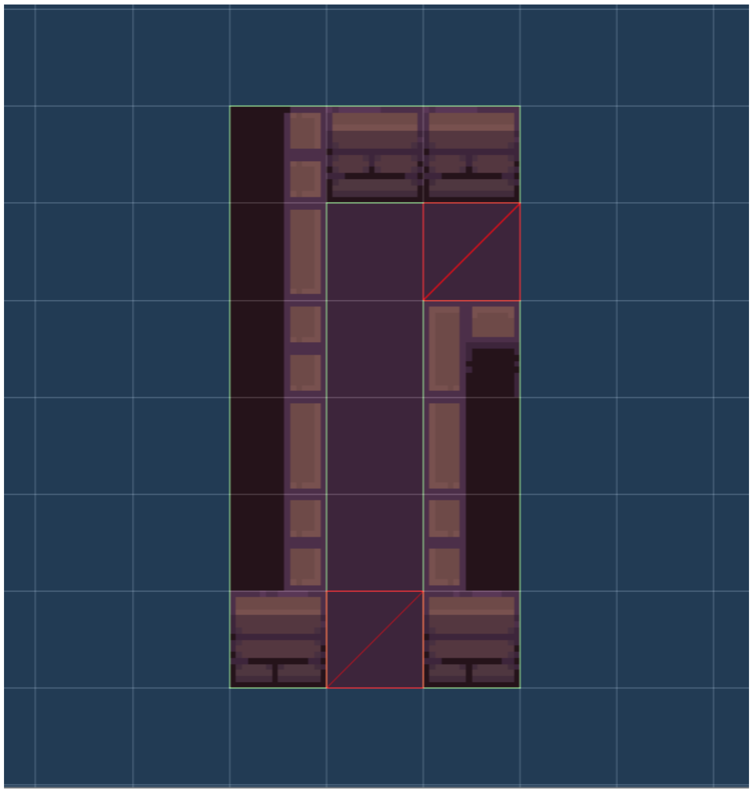
1. There should be exactly two door positions.
2. The two doors should be on the opposite sides of the room template.
3. The corridor should not be too long or too wide.



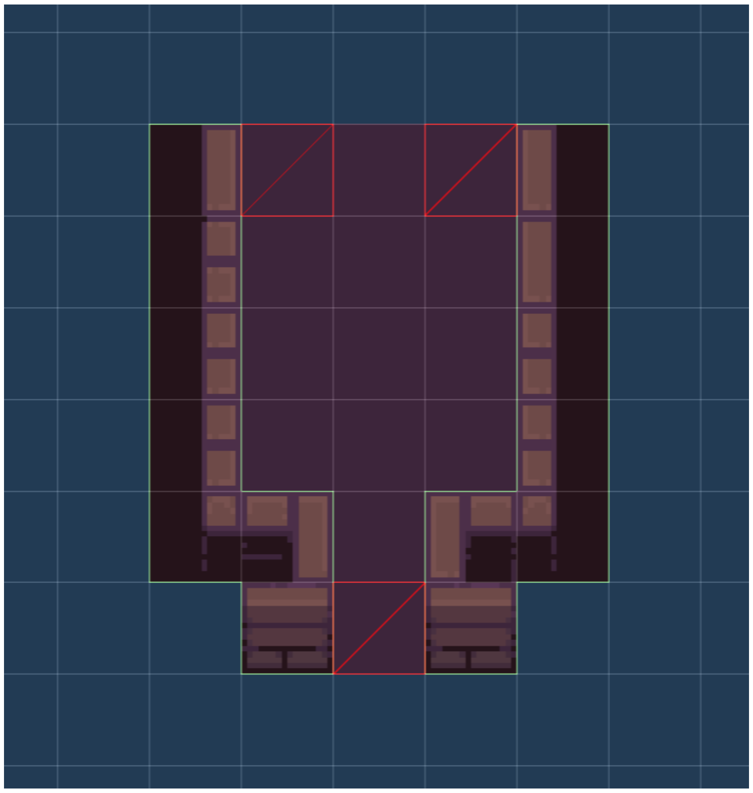
Recommended - narrow straight corridor



OK - little too wide but should be ok



Not recommended - doors not on opposite sides



Not recommended - more than 2 door positions

Final steps

After creating a room template game object, you can simply save it as a prefab, and it is ready to be used in a level graph.

Level graphs

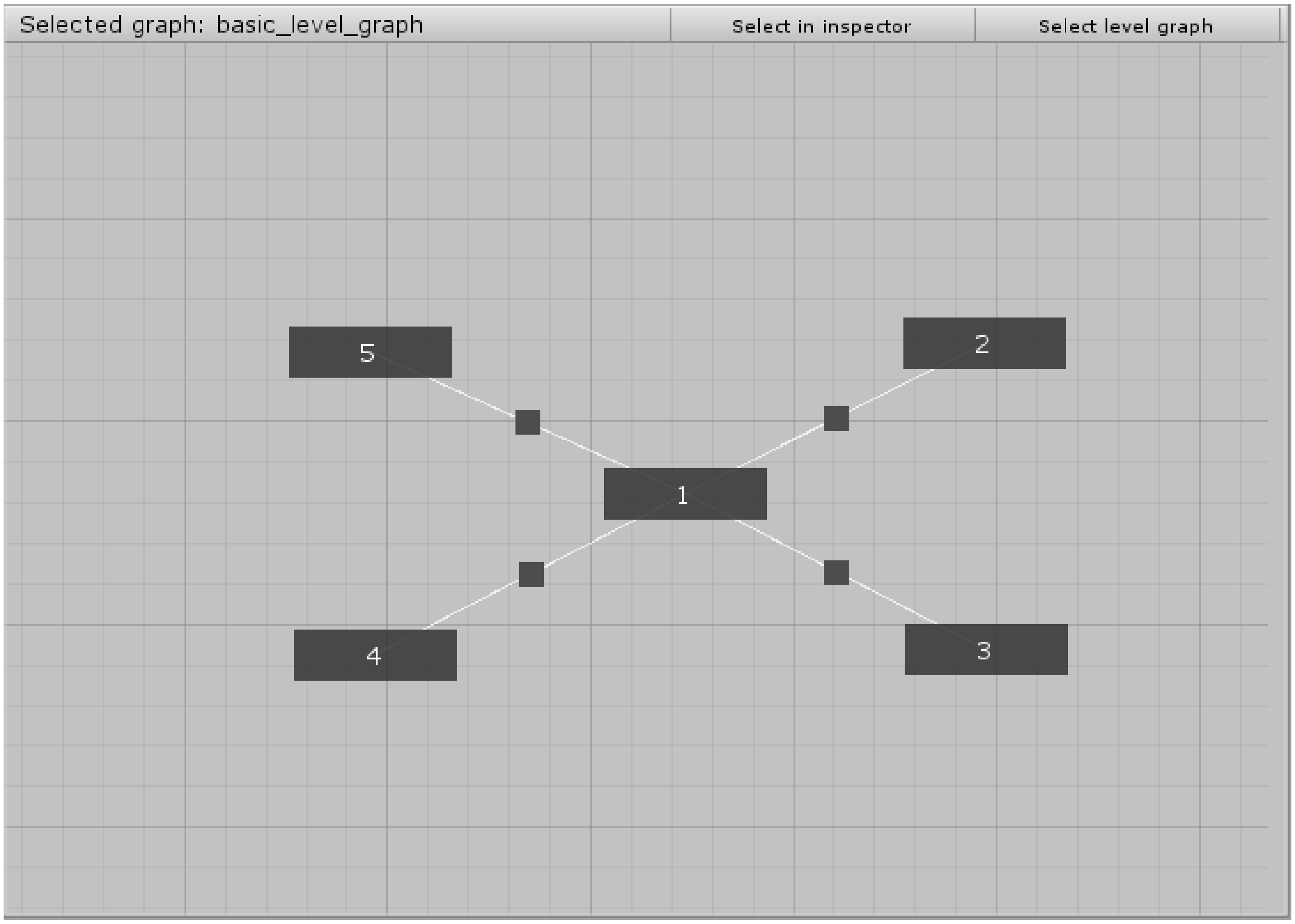
Level graph is an abstraction that lets us control the structure of generated levels.

Note: In the context of this plugin, the term *graph* is used to refer to a mathematical structure consisting of nodes and edges, not a way to visualize functions.

Basics

Level graph consists of rooms and room connections. Each room corresponds to a room in a generated level and each connection tells the algorithm that the two rooms must be connected either directly to each other or via a corridor.

Below you can see a simple level graph with 5 rooms and 4 connections. If we use this level graph as an input for the algorithm, each generated dungeon will have exactly 5 rooms and *room 1* will be connected to every other room in the dungeon.



Simple level graph with 5 rooms and 4 room connections

> **Note:** It is not important how we draw the graph. It is only important how many rooms there are and which rooms are connected to each other.

Limitations

Planar graphs

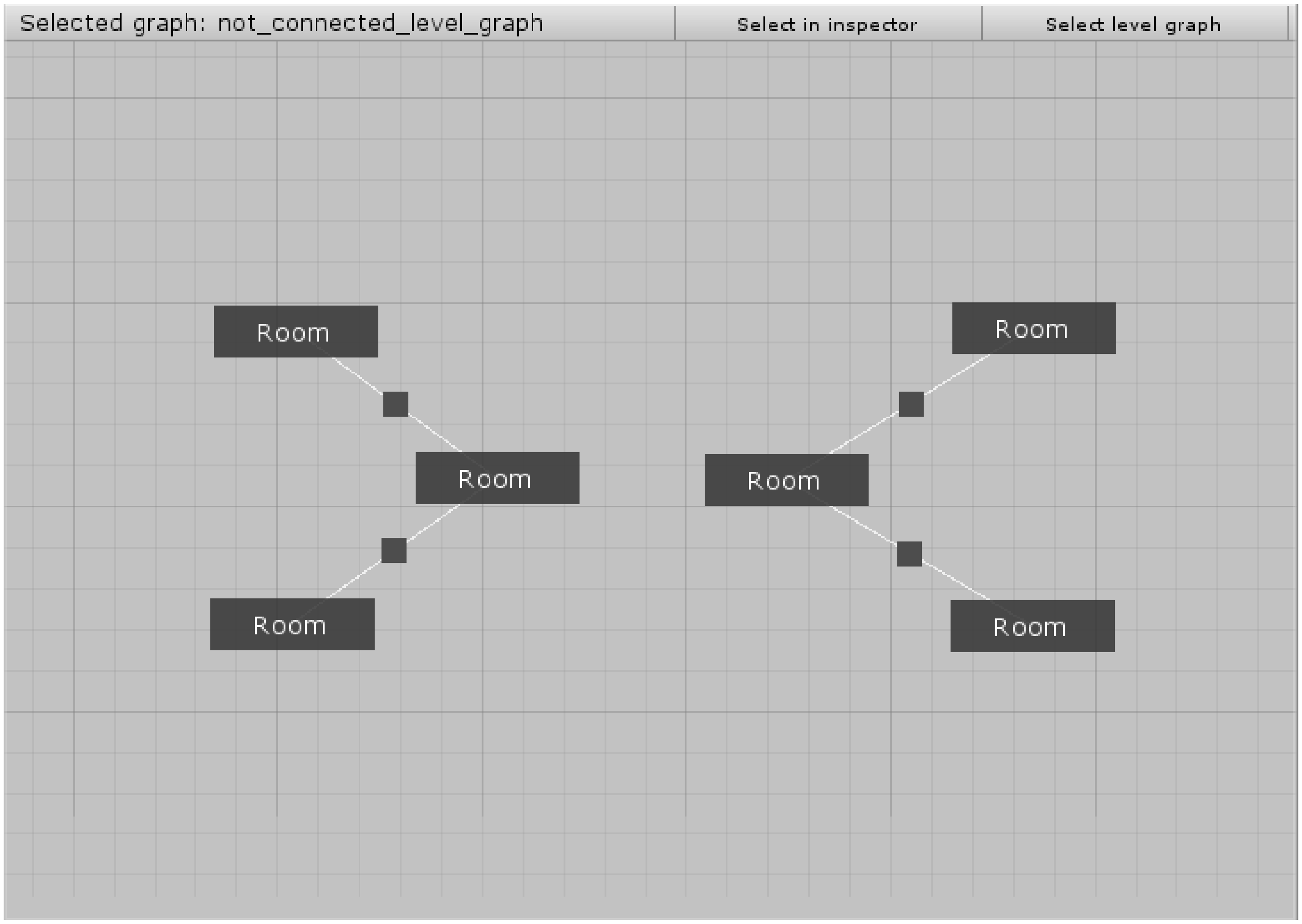
Level graphs must be **planar**. We say that a graph is planar if it can be drawn on the plane in such a way that no two edges intersect. In our case that means that no two connection lines may intersect. If the input graph was not planar, we would not be able to find a dungeon without rooms or corridors overlapping one another.

> **Note:** A level graph may be planar even if we draw it in a way that some edges intersect. It is because even if one drawing of the graph is "incorrect", that does not mean that there are intersecting edges

in all the drawings.

Connected graphs

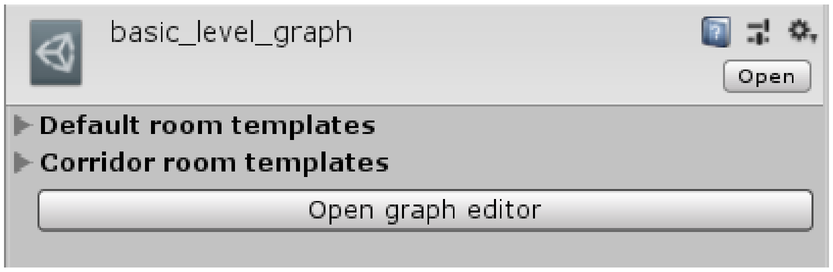
Level graphs must be **connected**. We say that a graph is connected if there is a path between every pair of vertices. Below you can see a level graph that is not connected because there is no path between vertices on the left side and vertices on the right side.



Example of a level graph that is not connected

Creating level graphs

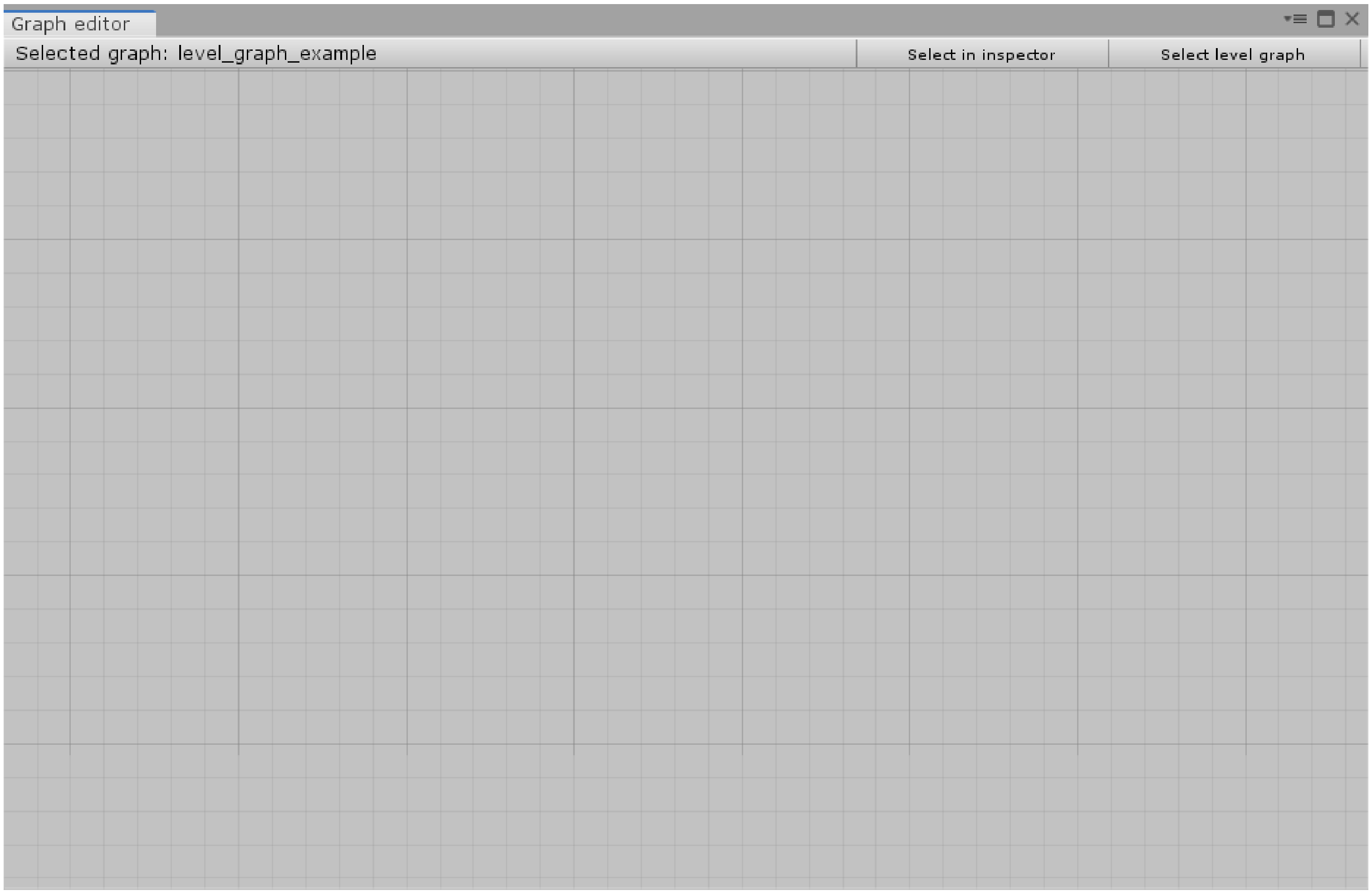
LevelGraph is a *ScriptableObject* that can be created by navigating to `Create → Edgar → Level graph`. Below you can see how are level graphs displayed in the Inspector window.



Level graph in the Inspector window

Graph editor

The Graph editor window can be opened double-clicking the level graph *ScriptableObject*.



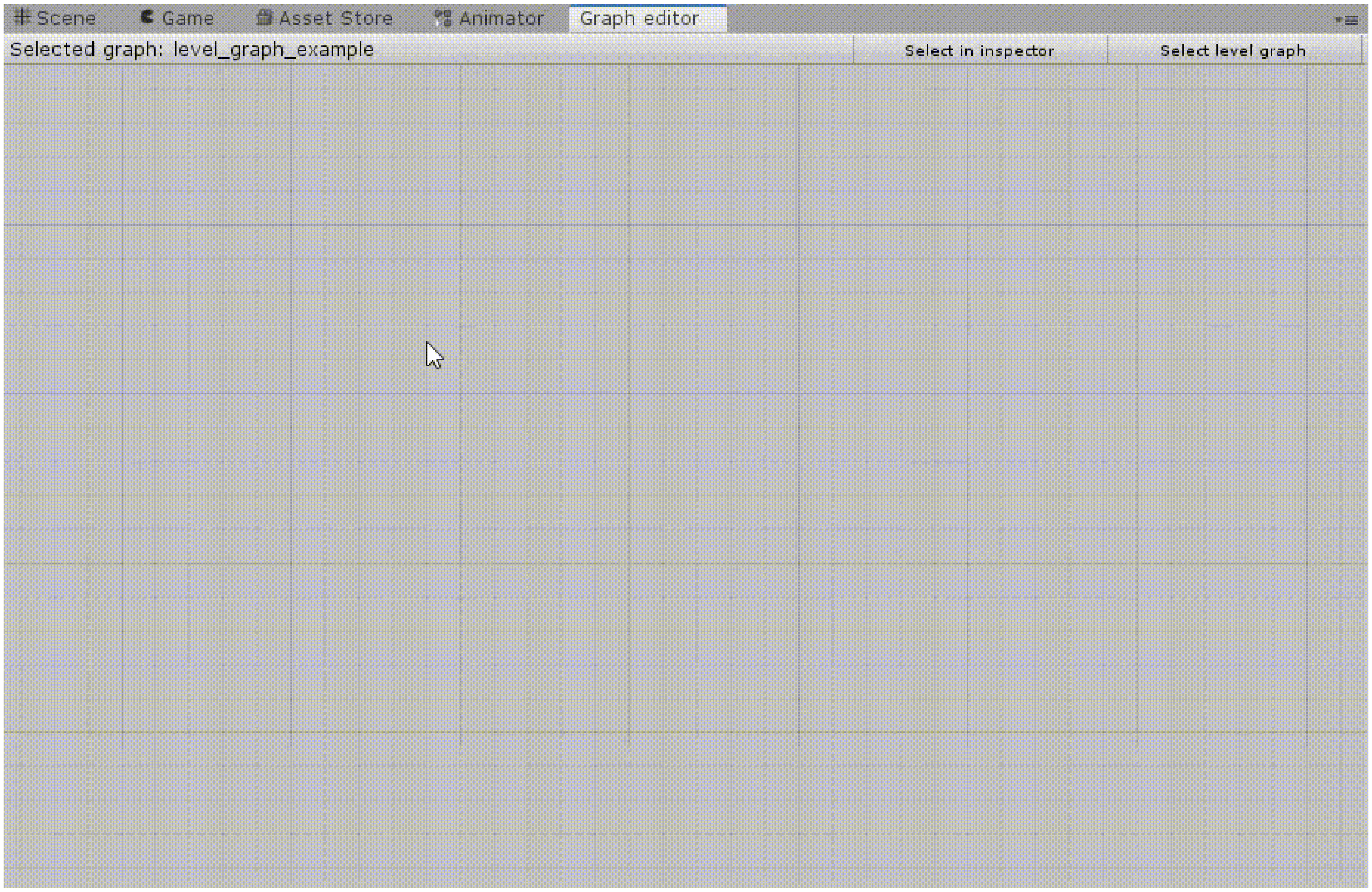
Graph editor window

Window controls:

- *Selected graph*: the name of the currently selected level graph
- *Select in inspector*: selects the current graph in the inspector window
- *Select level graph*: selects a different level graph

Working with level graphs:

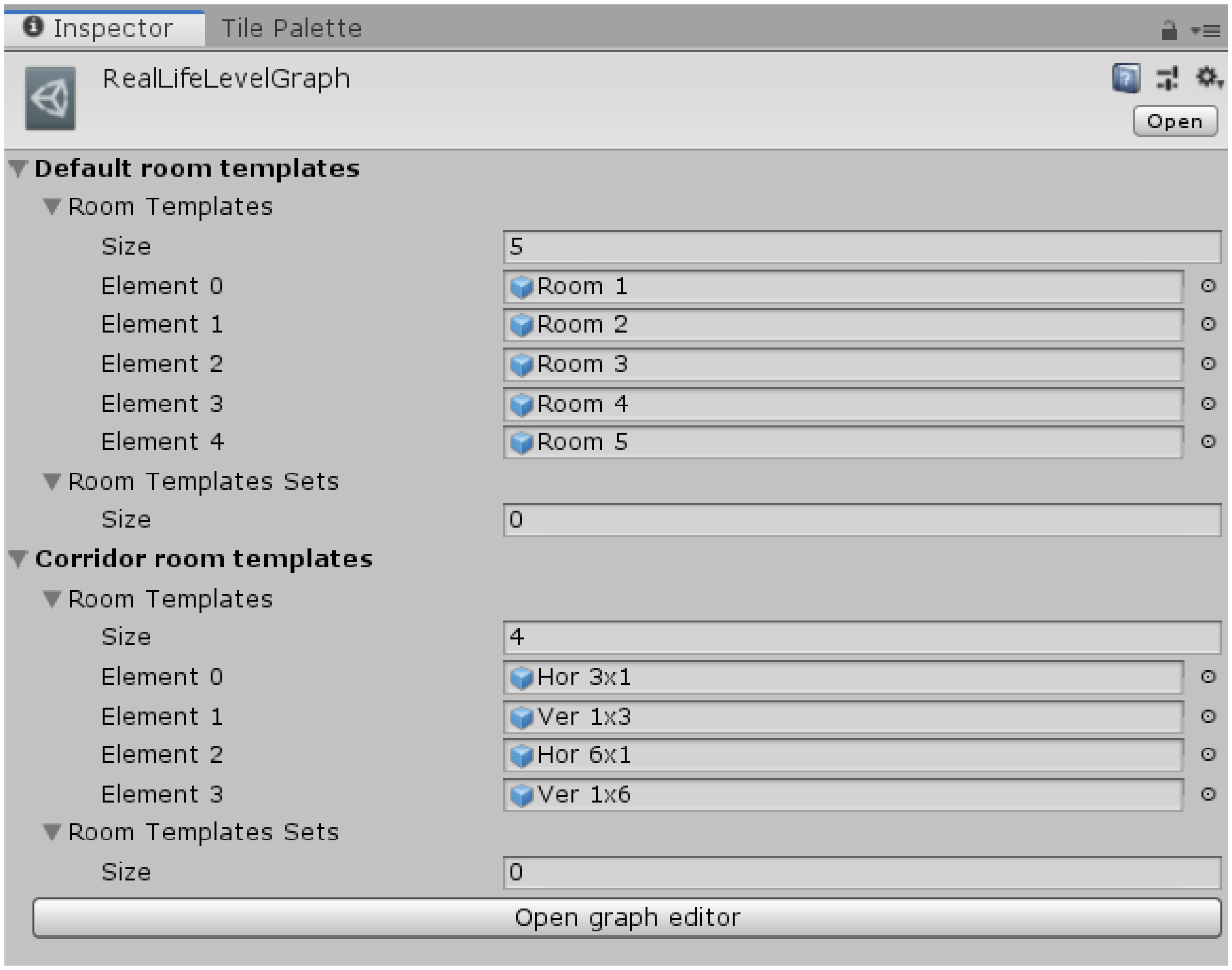
- *Create room*: double-click on an empty space in the grid
- *Configure room*: double-click on an existing room
- *Delete room*: press *Ctrl* + *Del*, or right-click on a room and select *Delete room*
- *Move room*: left click and then drag around
- *Add connection*: hold *Ctrl* while left-clicking a room and then move the cursor to a different room
- *Delete connection*: right-click on a connection handle and select *Delete connection*



Level graph controls

Room templates

When we have our rooms and connections, it is time to set up room templates. In the *Level graph* inspector window above, we can see 2 sections - *Default room templates* and *Corridor room templates*. These sections are used to specify which room templates are available for which room. Below you can see the setup from Example 1.



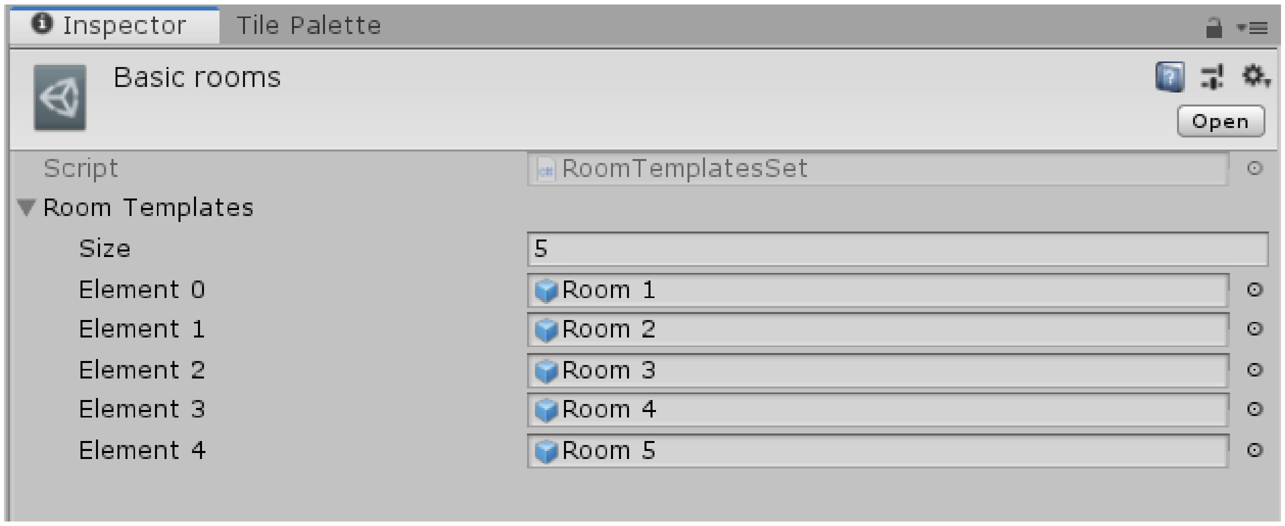
RealLifeLevelGraph

Default room templates
Room Templates
Size 5
Element 0 Room 1
Element 1 Room 2
Element 2 Room 3
Element 3 Room 4
Element 4 Room 5
Room Templates Sets
Size 0
Corridor room templates
Room Templates
Size 4
Element 0 Hor 3x1
Element 1 Ver 1x3
Element 2 Hor 6x1
Element 3 Ver 1x6
Room Templates Sets
Size 0

Example of assigned room templates

Room templates sets

It may sometimes be useful to group our room templates into groups like *Shop rooms*, *Boss rooms*, etc. We can create a so-called **Room templates set** by navigating to `Create → Edgar → Room templates set`. It is a simple ScriptableObject that holds an array of room templates, and we can use it instead of assigning individual room templates one by one. The main advantage is that if we later decide to add a new shop room template, we do not have to change all the shop rooms to include this new template - we simply add it to the room templates set.



Script	RoomTemplatesSet
Room Templates	
Size	5
Element 0	Room 1
Element 1	Room 2
Element 2	Room 3
Element 3	Room 4
Element 4	Room 5

Example of a room templates set that holds all our basic rooms. If we add another room template later, the change gets propagated to all the rooms in the level graph that are using this room templates set.

Default room templates

Room templates

Array of room templates that will be used for rooms that have no room shapes assigned. We can use this for our basic rooms and then configure our special rooms (spawn room, boss room, etc.) to use a different set of room templates.

Room templates sets

Array of room templates sets that will be used for rooms that have no room shapes assigned. Room templates from these sets are used together with individual room templates.

Corridor room templates

Room templates

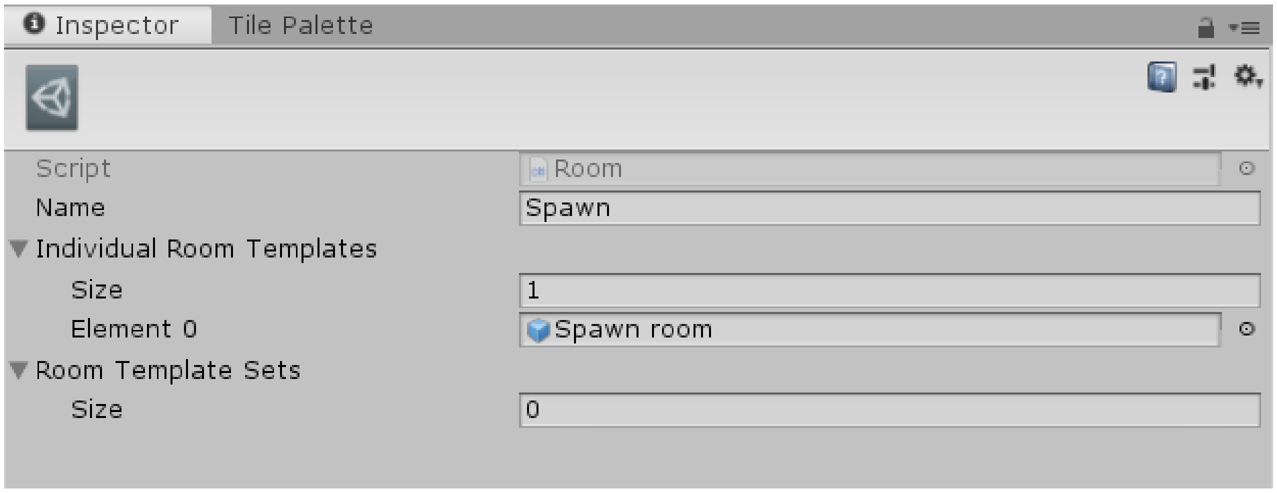
Array of room templates that will be used for corridor rooms. These room templates will be used if we set up the algorithm to use corridors instead of connecting rooms directly by doors. Can be left empty if we do not want to use corridors.

Room templates sets

Array of room templates sets that will be used for corridor rooms. Room templates from these sets are used together with individual room templates.

Configuring individual rooms

If we double-click on a room in the Graph editor, it gets selected, and we can configure it in the inspector. We can set the name of the room which will be displayed in the Graph editor. We can also assign room templates and room templates sets that will be used only for this room. By assigning any room template or room template set, we override the default room templates that are set in the level graph itself.



Script	Room
Name	Spawn
Individual Room Templates	
Size	1
Element 0	Spawn room
Room Template Sets	
Size	0

Configuration of a spawn room

(PRO) Custom rooms and connections

It may be often useful to add additional information to individual rooms (or connections). For example, we may want to add a type to each of the rooms and then do something based on the type. To achieve that, we can provide our own implementation of the RoomBase and ConnectionBase classes. There are at least two possible approaches.

Where is this feature used?

Below is the list of examples/tutorials where we use this feature. Feel free to check them out if you're not sure how something works or if you just want to see this feature used in action.

Dead Cells	The definition of rooms is enhanced with a custom type

Inherit from Room

The first approach is that we create a class that inherits from the Room class which is the default implementation that is used in level graphs. This approach is good if we want to just add something and do not want to change how the room works. We can also override the `GetDisplayName()` method to change how is the room displayed in the level graph editor.

This is the recommended approach for the majority of users.

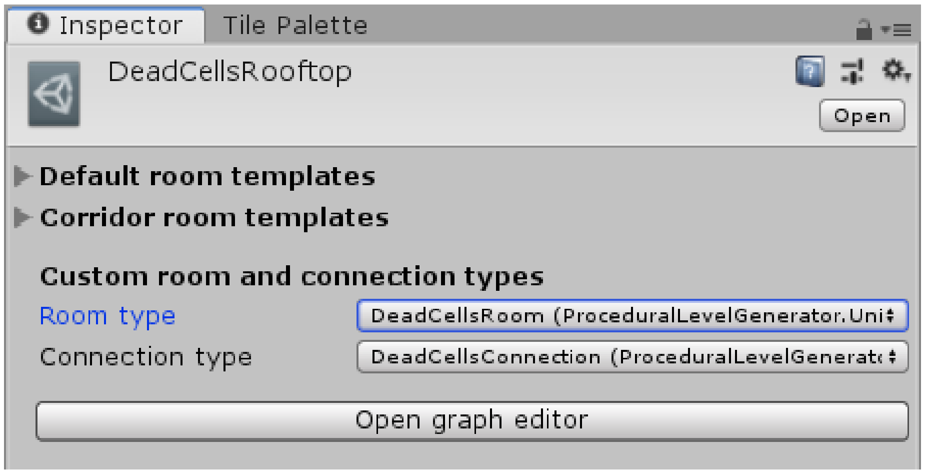
Inherit from RoomBase

The second approach is that we inherit directly from the RoomBase class. If we do that, we have to implement all the abstract methods (currently `GetDisplayName()` and `GetRoomTemplates()`). An advantage of this approach is that in some situations, we may not need any logic related to room templates, so we just return null from the method, and we will not see anything related to room templates in the inspector of the room. This may be useful in a situation where we resolve room templates manually based on the type of the room.

> **Note:** The same logic applies to inheriting from Connection or ConnectionBase.

Configure level graph

When we have our custom room or connection type ready, we have to configure the level graph to use them. If we open the level graph in the inspector, we should be able to choose the custom types from the dropdown.



Custom room and connection types

Room type: DeadCellsRoom (ProceduralLevelGenerator.Uni
Connection type: DeadCellsConnection (ProceduralLevelGenerat

Custom room and connection types (PRO version)

> **Note:** It is not possible to easily convert a level graph from using one room/connection type to another. Therefore, it is important to decide if you want to use a custom room/connection before you

create your level graphs. Otherwise, you will have to recreate them later with the correct types.

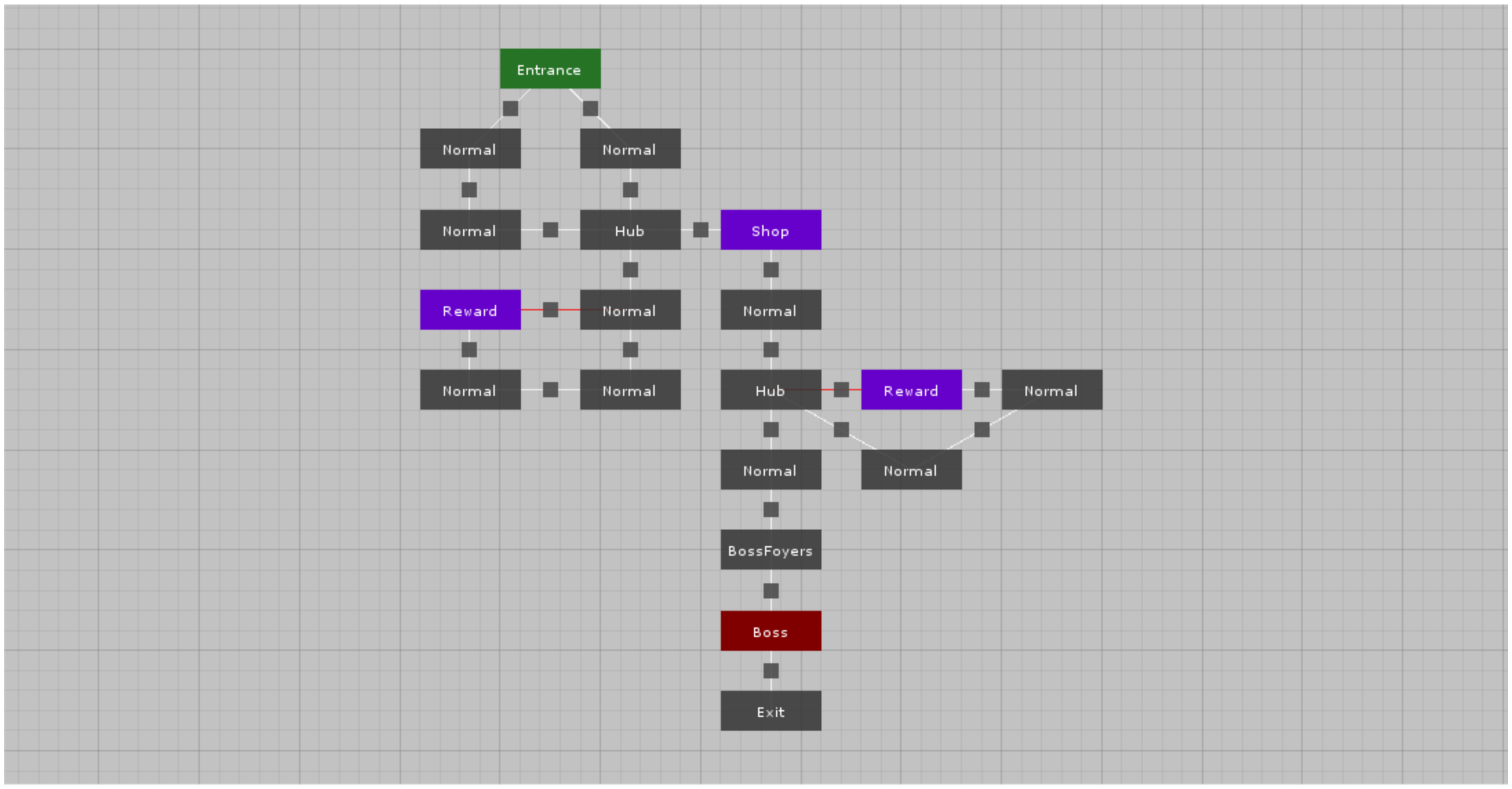
Accessing room information

If we add some additional information to a room or connection, we probably expect to somehow use this information later. The first step is to get access to the RoomInstance class which is described here. When we have an instance of this class, we can use the `RoomInstance.Room` property. This property is of the `RoomBase` type, so we have to cast it to our custom room type.

Custom colours in the level graph editor

It is also possible to change how custom rooms and connections look in the level graph editor. We just have to override the `GetEditorStyle()` method and return an instance of RoomEditorStyle or ConnectionEditorStyle.

```
public class GungeonRoom : RoomBase
{
    public GungeonRoomType Type;

    /* ... */

    public override RoomEditorStyle GetEditorStyle(bool isFocused)
    {
        var style = base.GetEditorStyle(isFocused);

        var backgroundColor = style.BackgroundColor;

        // Use different colors for different types of rooms
        switch (Type)
        {
            case GungeonRoomType.Entrance:
                backgroundColor = new Color(38/256f, 115/256f, 38/256f);
                break;

            /* ... */
        }

        style.BackgroundColor = backgroundColor;

        // Darken the color when focused
        if (isFocused)
        {
            style.BackgroundColor = Color.Lerp(style.BackgroundColor, Color.black, 0.7f);
        }

        return style;
    }
}
```



Different colours for special types of rooms

(PRO) Directed level graphs

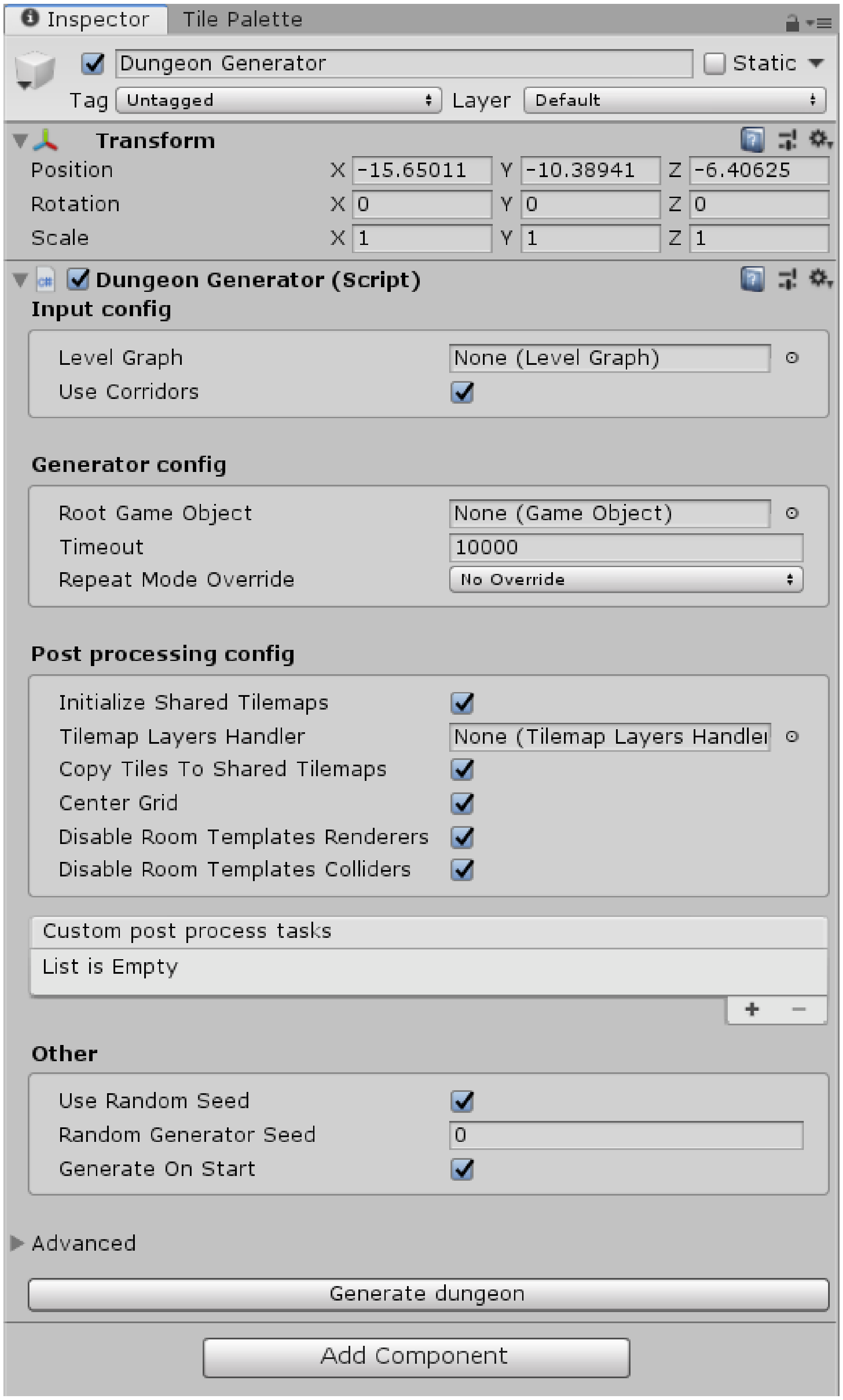
By default, all level graphs are undirected, meaning that it does not matter whether you create a connection from *Room 1* to *Room 2* or the other way around. If you want to have more control over generated levels, you can make level graphs directed and combine that with entrance-only and exit-only doors. See the Directed level graphs guide for more information and examples.

Generator setup

With our level graph prepared, we are now ready to generate our very first dungeon. In order to do that, we have to set up our procedural dungeon generator.

The setup is very easy:

1. Create an empty game object in the scene
2. Add the **Dungeon Generator** component to that game object
3. Assign your level graph to the **Level Graph** field
4. Hit the **Generate dungeon** button or enable **Generate on start** and enter play mode


Inspector | Tile Palette

✓ Dungeon Generator | Static

Tag: Untagged | Layer: Default

Transform

	X	Y	Z
Position	-15.65011	-10.38941	-6.40625
Rotation	0	0	0
Scale	1	1	1

✓ Dungeon Generator (Script)

Input config

Level Graph	None (Level Graph)
Use Corridors	✓

Generator config

Root Game Object	None (Game Object)
Timeout	10000
Repeat Mode Override	No Override

Post processing config

Initialize Shared Tilemaps	✓
Tilemap Layers Handler	None (Tilemap Layers Handlei
Copy Tiles To Shared Tilemaps	✓
Center Grid	✓
Disable Room Templates Renderers	✓
Disable Room Templates Colliders	✓

Custom post process tasks

List is Empty

Other

Use Random Seed	✓
Random Generator Seed	0
Generate On Start	✓

Advanced

Dungeon generator runner

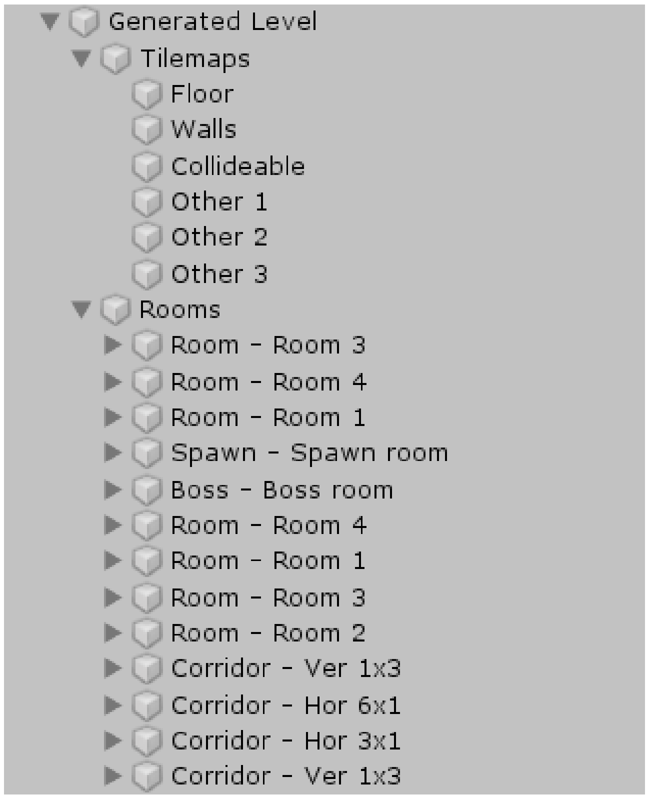
Note: The goal of this guide is not to describe individual options of the generator but rather to describe how to get an instance of the generator with a reasonable default configuration. If you are interested in individual options, head to the Dungeon generator page.

Level structure and rooms data

Level structure

In the image below, we can see the structure of a level. The game object that holds the level has two children - **Tilemaps** game object and **Rooms** game object:

- **Tilemaps** game object holds all the tilemap layers.
- **Rooms** game object holds instances of all room templates that are used in the level. Name of each of the children is formed as **"{roomName} - {roomTemplate}"** to make it easier to find a specific room when debugging.



```
Generated Level
  Tilemaps
    Floor
    Walls
    Collideable
    Other 1
    Other 2
    Other 3
  Rooms
    Room - Room 3
    Room - Room 4
    Room - Room 1
    Spawn - Spawn room
    Boss - Boss room
    Room - Room 4
    Room - Room 1
    Room - Room 3
    Room - Room 2
    Corridor - Ver 1x3
    Corridor - Hor 6x1
    Corridor - Hor 3x1
    Corridor - Ver 1x3
```

Structure of the level

> **Note:** If you want to retrieve any of the game objects from a script, the best-practice is to use the GeneratorConstants static fields instead of using hardcoded names.

Rooms information

The generator also produces information about individual rooms in the level - their positions, which room template is used, what are neighbours of the room, etc. All this information is exposed through the RoomInstance class.

There are at least two ways of getting an instance of this class:

- **From the game object of the room.** All the room template instances that were described in the previous section have a RoomInfo component attached. This component has a reference to the corresponding *RoomInstance*.
- **From a post-processing task**. Each custom post-processing task receives an instance of the GeneratedLevel class which has a method `GetRoomInstances()` that can be used to retrieve all the room instances from the level.

Performance tips

When used correctly, Edgar can generate very complex levels. Unfortunately, it is also relatively simple to prepare an input that is just too difficult for the generator, and you will end up with a `TimeoutException`. The goal of this page is to provide some performance tips that you can follow to improve the performance of the generator.

The general idea is that if you make it harder for the generator in one way (e.g. by having many rooms), you should compensate for that in some other way (e.g. by not having cycles in your level graph). Also, I recommend starting simple and only making things more complex when you get the hang of how the generator behaves.

Room templates

Try to provide as many door positions as possible. I cannot stress enough how important this is. You should aim to use the *Simple* or *Hybrid* door modes as much as possible, and only use the *Manual* door mode when it is absolutely necessary. The only exception is when you are trying to generate levels without cycles, then you can get away with having a relatively small number of door positions.

Make sure that default room templates make sense. The easiest way to configure room templates is to add them as *Default room templates*, making them available to all rooms in the level graph. However, it is not recommended adding room templates that can be used only in very specific scenarios. For example, if you have a secret room that has exactly one door position, you should not add it to the default list. The reason is that the generator might try to use this room template for a room that has multiple neighbours, wasting precious time. Instead, assign these unique room templates only to the rooms where it makes sense.

Level graphs

Limit the number of rooms. The number of rooms in a level graph greatly affects the performance. As a rule of thumb, you should aim to have **less than 20 rooms**. However, if you follow the other performance tips, you can generate levels with up to 40 rooms.

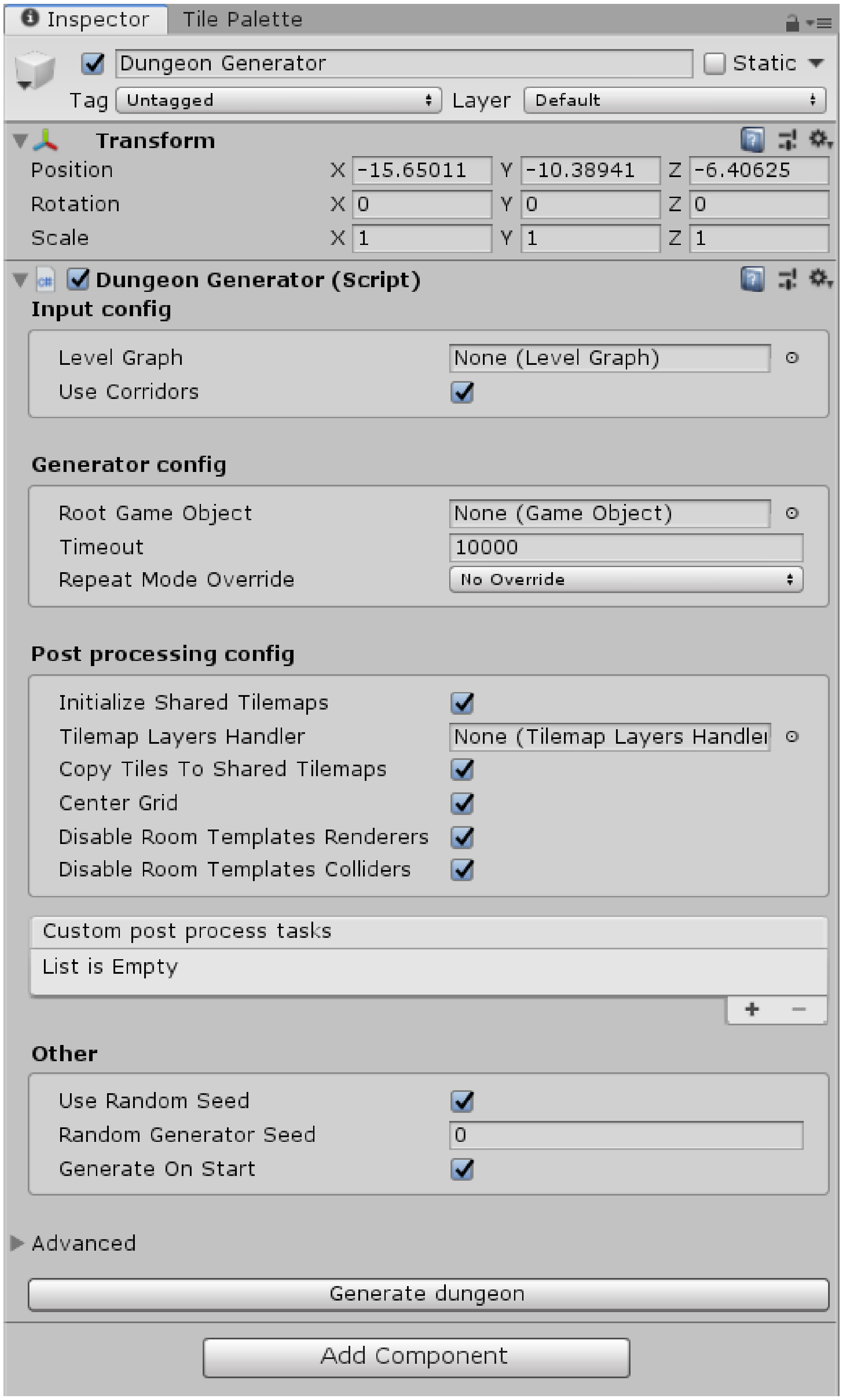
Limit the number of cycles. It is very hard to generate levels with cycles. Therefore, the number of cycles greatly affects the performance. Usually, you should start with 0-1 cycles and only increase the number when you are already familiar with the core concepts of Edgar. In the Enter the Gungeon example, you can see levels graphs with up to 3 cycles and the generator is still relatively fast.

Avoid interconnected cycles. Cycles are hard, but interconnected cycles are even harder. If you want to have multiple cycles in a level graph, make sure that the cycles do not have any rooms in common. Usually, it should not be too hard to design your level graphs without interconnected cycles. For example, all the level graphs in *Enter the Gungeon* have this property, and it does not make the game any worse.

Dungeon generator

Minimal setup

- Add the **Dungeon Generator Component** to any Game Object in the Scene.
- Assign your level graph to the **Level Graph** field.
- Hit the **Generate dungeon** button or enable **Generate on start** and enter play mode



✓ Dungeon Generator ☐ Static

Tag Untagged Layer Default

Transform

Position	X	-15.65011	Y	-10.38941	Z	-6.40625
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

✓ Dungeon Generator (Script)

Input config

Level Graph	None (Level Graph)
Use Corridors	✓

Generator config

Root Game Object	None (Game Object)
Timeout	10000
Repeat Mode Override	No Override

Post processing config

Initialize Shared Tilemaps	✓
Tilemap Layers Handler	None (Tilemap Layers Handler
Copy Tiles To Shared Tilemaps	✓
Center Grid	✓
Disable Room Templates Renderers	✓
Disable Room Templates Colliders	✓

Custom post process tasks

List is Empty

Other

Use Random Seed	✓
Random Generator Seed	0
Generate On Start	✓

Advanced

Dungeon generator runner

Configuration

Input config (`FixedLevelGraphConfigGrid2D`)

- **Level Graph** - Level graph that should be used. Must not be null.
- **Use corridors** - Whether corridors should be used between neighbouring rooms. If enabled, corridor room templates must be provided in the level graph.

Generator config (`DungeonGeneratorConfigGrid2D`)

- **Root Game Object** - Game Object to which the generated level will be attached. New Game Object will be created if null.
- **Timeout** - How long (in milliseconds) should we wait for the algorithm to generate a level. We may sometimes create an input that is too hard for the algorithm, so it is good to stop after some amount of time with an error.
- **Repeat Mode Override** - Whether to override the repeat mode of individual room templates.
  - **No override** - Nothing is overridden, keep repeat modes from room templates.
  - **Allow repeat** - All room templates may repeat in generated levels.
  - **No immediate** - Neighbouring room must have different room templates.
  - **No repeat** - All rooms must have different room templates.

> **Note:** If you provide too few room templates, they may repeat in generated levels even if you choose the **No immediate** or **No repeat** options. To make sure that the repeat mode is satisfied, please provide enough room templates to choose from.

- **Minimum Room Distance** - The minimum distance between non-neighbouring rooms.
  - If equal to **0** - walls from one room can occupy the same tiles as walls from a different room.
  - If equal to **1** (default) - walls from different rooms can be next to each other but not on top of each other.
  - If equal to **2** - there must be at least one empty tile between walls of different rooms. (This is good for when using rule tiles and weird things are happening.)

> **Note:** Higher values of *Minimum Room Distance* may negatively affect the performance of the generator. Moreover, with very short corridor, it might even be impossible to generate a level with a high value of this parameter.

Post-processing config (`PostProcessingConfigGrid2D`)

Please see the Post-processing page to find detailed information about this configuration.

- **Initialize Shared Tilemaps** - Whether to initialize tilemaps that will hold the generated level.

- **Tilemap Layers Handler** - Which tilemap layers handler should be used to initialize shared tilemaps. Uses the `DungeonTilemapLayersHandler` if not set.
- **Tilemap Material** - Material that will be used in Tilemap Renderers of shared tilemaps. This is useful, for example, for lights. If left null, the default material will be used.
- **Copy Tiles To Shared Tilemaps** - Whether to copy tiles from individual room template to the shared tilemaps.
- **Center Grid** - Whether to move the level so that its centre is approximately at (0,0). Useful for debugging in Scene view in the editor.
- **Disable Room Template Renderers** - Whether to disable tilemap renderers of individual rooms, useful only when *Copy Tiles To Shared Tilemaps* is enabled.
- **Disable Room Template Colliders** - Whether to disable tilemap colliders of individual rooms, useful only when *Copy Tiles To Shared Tilemaps* is enabled.

Other config (available directly on the generator class)

- **Use Random Seed** - Whether to use a random seed for each new generated level.
- **Random Generator Seed** - Random generator seed that will be used when **Use Random Seed** is disabled. Useful for debugging.
- **Generate On Start** - Whether to generate a new level when play mode is entered.

Change the configuration from a script

You can also easily change the configuration of the generator directly from a script:

```
// Get the generator component
var generator = GameObject.Find("Dungeon Generator").GetComponent<DungeonGeneratorGrid2D>();

// Access the input config
generator.FixedLevelGraphConfig.UseCorridors = false;

// Access the generator config
generator.GeneratorConfig.Timeout = 5000;

// Access the post-processing config
generator.PostProcessConfig.CenterGrid = false;

// Access other properties
generator.UseRandomSeed = false;
generator.RandomGeneratorSeed = 1000;
generator.GenerateOnStart = false;
```

Call the generator from a script

It is very simple to call the generator from a script:

1. Get the `DungeonGenerator` component from the game object that holds the generator
2. Call the `Generate()` method

Example:

```
var generator = GameObject.Find("Dungeon Generator").GetComponent<DungeonGeneratorGrid2D>();
generator.Generate();
```

> **Note:** The `Generate()` method blocks the main Unity thread, so the game may freeze while the dungeon is generated. The PRO version comes with an implementation that uses coroutines to make sure that the games does not freeze.

(PRO) With coroutines

If we do not want to block the main Unity thread when the level is generating, we can use a coroutine. There are two ways of doing that.

The simple approach uses only built-in Unity coroutines and works like this:

```
var generator = GameObject.Find("Dungeon Generator").GetComponent<DungeonGeneratorGrid2D>();
StartCoroutine(generator.GenerateCoroutine());
```

There is one problem with the simple approach - coroutines cannot really handle exceptions. So if there is some problem with the generator or with custom post-processing logic, the coroutine just dies, and we are not able to do any clean-up. Therefore, I implemented a smarter coroutine that lets us handle any errors. Example usage:

```
public class CoroutineWithDataExampleAdvanced : MonoBehaviour
{
    public void Start()
    {
        var generator = GameObject.Find("Dungeon Generator").GetComponent<DungeonGeneratorGrid2D>();
        StartCoroutine(GeneratorCoroutine(generator));
    }

    private IEnumerator GeneratorCoroutine(DungeonGeneratorGrid2D generator)
    {
        // Start the smart coroutine
        // StartCoroutineWithData is a custom extension method of MonoBehaviour, be sure to use the Edgar.Unity
namespace
        var generatorCoroutine = this.StartSmartCoroutine(generator.GenerateCoroutine());

        // Wait until the smart coroutine is completed
        // Make sure to yield return the Coroutine property and not the generator coroutine itself!!
        yield return generatorCoroutine.Coroutine;

        // Check if the coroutine was successful
        if (generatorCoroutine.IsSuccessful)
        {
            Debug.Log("Level generated!");
        }
        // If there were any errors, we can access the Exception object
        // Or we can also rethrow the exception by calling generatorCoroutine.ThrowIfNotSuccessful();
        else
        {
            Debug.LogError("There was an error when generating the level!");
            Debug.LogError(generatorCoroutine.Exception.Message);
        }
    }
}
```

Post-processing

After a level is generated, we may often want to run some additional logic like spawning enemies, etc. This can be achieved by providing your own post-processing logic that will be called after the level is generated and provided with information about the level.

To better understand how the generator works, we will first describe which post-processing is done by the generator itself and then provide ways to extend this behaviour and provide your own logic. You can skip right to Custom post-processing if that is what you are looking for.

Built-in post-processing steps

0. Instantiate room template with correct positions

This is not actually a post-processing as it happens in the generator stage of the whole generator and cannot be disabled. At this point, the generator knows the final position and room template for each room in the level. The generator goes through the rooms and instantiates the corresponding room template game object and moves it to its correct position. If we disabled all the other post-processing steps, we would get a bunch of correctly positioned rooms, but there would often be weird overlap between the rooms.

1. Initialize shared tilemaps

In this step, the generator initializes the structure of shared tilemaps to which we will copy individual rooms in the next step. These tilemaps will contain all the tiles in the level. If you provided your own *Tilemap Layers Handler*, this is the time it gets called.

2. Copy rooms to shared tilemaps

In this step, the generator copies individual rooms to shared tilemaps. If we use corridors, it is important to first copy other rooms and only then corridors. By doing so, corridors will make holes into the walls of other rooms, so we can go from one room to another.

3. Center grid

In this step, the whole level is moved in a way that its centre ends up at (0,0). This step is only used to make it easier to go through multiple generated levels without having to move the camera around.

4. Disable room template renderers

At this point, we display both tiles from shared tilemaps and tiles from individual room template game objects that we instantiated in the step 0. Therefore, we have to disable all tilemap renderers from individual room templates.

You may think why do not we just disable the whole room template. The reason for that is that there may be some additional game objects like lights, enemies, etc. and we do not want to lose that.

5. Disable room template colliders

The last step is very similar to the previous step. At this point, there are colliders from individual room templates that would prevent us from going from one room to another. We keep only the colliders that are set to trigger because these may be useful for example for Current room detection.

Custom post-processing

There are currently 2 ways of implementing custom processing-logic: you either implement a custom component or use a *ScriptableObject*. I would recommend starting with a custom component as it is a bit easier, any only use *ScriptableObjects* if you need some of their benefits.

> **Note:** Previously (before *v2.0.0-beta.0*), it was only possible to create custom post-processing logic using *ScriptableObjects*. But that process is relatively tedious: you have to add the `CreateAssetMenu` attribute (which I never remember) and then create an instance of that *ScriptableObject*. Therefore, I also made it possible to use a `MonoBehaviour` component and just attach it to the generator game object.

Using a `MonoBehaviour` component

The first approach is to create a class that inherits from the `DungeonGeneratorPostProcessingComponentGrid2D` class (which in turn inherits from `MonoBehaviour`). This class expects you to override the `void Run(DungeonGeneratorLevelGrid2D level)` method where you should place your post-processing logic.

```
public class MyCustomPostProcessingComponent : DungeonGeneratorPostProcessingGrid2D
{
    public override void Run(DungeonGeneratorLevelGrid2D level)
    {
        // Implement the logic here
    }
}
```

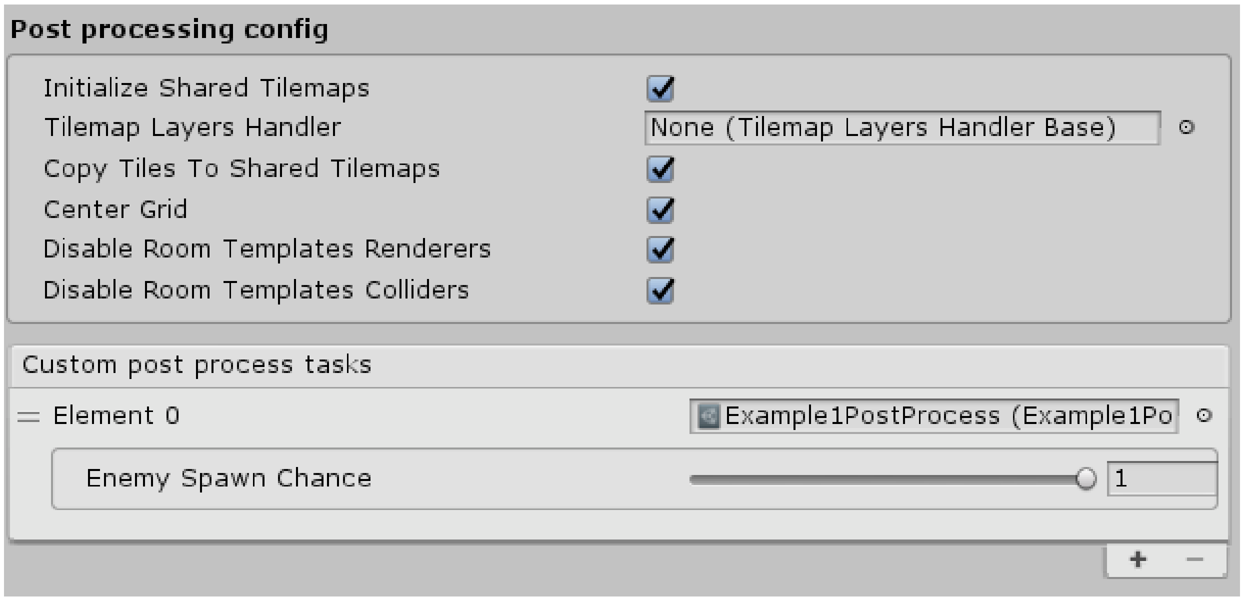
When you have your implementation ready, go to the scene where you have the generator and attach this component to the game object with the generator. If you now run the generator, your post-processing code should be called.

Using a `ScriptableObject`

The second approach is to create a class that inherits from `DungeonGeneratorPostProcessingGrid2D` (which in turn inherits from `ScriptableObject`). And because the base class is a *ScriptableObject*, you need to add the `CreateAssetMenu` attribute, so you are able to create an instance of that *ScriptableObject*. This class expects you to override the `void Run(DungeonGeneratorLevelGrid2D level)` method where you should place your post-processing logic.

```
[CreateAssetMenu(menuName = "Edgar/Examples/Docs/My custom post-processing", fileName =
"MyCustomPostProcessing")]
public class MyCustomPostProcessing : DungeonGeneratorPostProcessingGrid2D
{
    public override void Run(DungeonGeneratorLevelGrid2D level)
    {
        // Implement the logic here
    }
}
```

When you have your implementation ready, you now have to create an instance of that ScriptableObject by right-clicking in the project view and then *Create → Edgar → Examples → Docs → My custom post-processing*. And the last step is to drag and drop this ScriptableObject in the *Custom post process tasks* section of the dungeon generator.


Post processing config

Initialize Shared Tilemaps	✓
Tilemap Layers Handler	None (Tilemap Layers Handler Base)
Copy Tiles To Shared Tilemaps	✓
Center Grid	✓
Disable Room Templates Renderers	✓
Disable Room Templates Colliders	✓

Custom post process tasks

Element 0	Example1PostProcess (Example1Po
Enemy Spawn Chance	1

Add the ScriptableObject to the Custom post process tasks array

Where is this feature used?

Below is the list of examples/tutorials where we use this feature. Feel free to check them out if you're not sure how something works or if you just want to see this feature used in action.

Example 1, Dead Cells	A custom post-processing task is used to spawn enemies after a level is generated
Dead Cells	Move the player to the spawn position of the level

(PRO) Priority callbacks

(PRO) Platformer generator

Minimal setup

- Add the **Platformer Generator Component** to any Game Object in the Scene.
- Assign your level graph to the **Level Graph** field.
- Hit the **Generate platformer** button or enable **Generate on start** and enter play mode

Configuration and usage

The configuration is currently the same as for the Dungeon Generator. See this page. The only difference is that if we want to use the generator from code, we use the `PlatformerGenerator` class.

Default tilemaps structure

Platformer room templates can be created via *Create → Edgar → Platformer room template* and their default tilemaps structure is the following:

- **Background**
- **Walls** - collider
- **Platforms** - collider and platform effector
- **Collideable** - collider
- **Other 1**
- **Other 2**

Limitations

There are some limitation regarding the platformer generator.

Acyclic level graphs

We should use only **acyclic** graphs, i.e. graphs without cycles. The room templates for platformers are often too restrictive to allow cycles. The generator currently allows graphs with cycles, but it often happens that it is not able to generate any level.

Solvability of generated levels

The generator cannot guarantee that all the levels are solvable, i.e. it is possible to successfully traverse generated levels without being stuck at dead-ends caused by impossible jumps, etc. Probably the easiest way to handle that is to design room templates and level graphs in such a way that the generator cannot connect two rooms in a way that it is not possible go one from the first one to the other one.

(PRO) Custom input

In the free version of the asset, the input for the generator is fixed. That means that we create a level graph in the GUI and give it directly to the generator. However, there are situations where we might want to alter the level graph. For example, we may want to add a secret room that is connected to a random room in the level.

In this tutorial, we will learn how to implement custom inputs in order to have more control over the input for the generator.

`LevelGraph` and `LevelDescription`

The first thing that we need to understand is the difference between `LevelGraph` and `LevelDescription` classes. If you are reading this tutorial, you probably know what is a level graph. It is a collection of rooms and connections, and it describes the high-level structure of generated levels. With each level graph is associated an instance of the `LevelGraph` scriptable object.

However, level graphs are not directly given to the generator as an input. First, we need to convert the `LevelGraph` to an instance of the `LevelDescription` class. The reason for that is that level graphs are made primarily for the GUI editor, and we need to convert them to a real graph data structure.

Both `LevelGraph` and `LevelDescription` revolve around rooms and connections. The following code should demonstrate the basic API of both classes and how to convert one to the other one:

```
// [CreateAssetMenu(menuName = "Dungeon generator/Examples/Docs/My custom input task", fileName =
"MyCustomInputSetup")]
public class CustomInputExample : DungeonGeneratorInputBaseGrid2D
{
    public LevelGraph LevelGraph;
    public bool UseCorridors;

    protected override LevelDescriptionGrid2D GetLevelDescription()
    {
        var levelDescription = new LevelDescriptionGrid2D();

        // Go through rooms in the level graph and add them to the level description
        foreach (var room in LevelGraph.Rooms)
        {
            levelDescription.AddRoom(room, GetRoomTemplates(room));
        }

        // Go through connections in the level graph
        foreach (var connection in LevelGraph.Connections)
        {
            // If corridors are enabled, add corridor connection
            if (UseCorridors)
            {
                // Create a room for the corridor
                var corridorRoom = ScriptableObject.CreateInstance<Room>();
                corridorRoom.Name = "Corridor";


                levelDescription.AddCorridorConnection(connection, corridorRoom, GetCorridorRoomTemplates());
            }
            // Else connect the rooms directly
            else
            {
                levelDescription.AddConnection(connection);
            }
        }

        return levelDescription;
    }

    /// <summary>
    /// Gets room templates for a given room.
    /// </summary>
    private List<GameObject> GetRoomTemplates(RoomBase room)
    {
        // Get room templates from a given room
        var roomTemplates = room.GetRoomTemplates();

        // If the list is empty, we use the defaults room templates from the level graph
        if (roomTemplates == null || roomTemplates.Count == 0)
        {
            var defaultRoomTemplates = LevelGraph.DefaultIndividualRoomTemplates;
            var defaultRoomTemplatesFromSets = LevelGraph.DefaultRoomTemplateSets.SelectMany(x =>
x.RoomTemplates);

            // Combine individual room templates with room templates from room template sets
            return defaultRoomTemplates.Union(defaultRoomTemplatesFromSets).ToList();
        }

        return roomTemplates;
    }

    /// <summary>
    /// Gets corridor room templates.
    /// </summary>
    private List<GameObject> GetCorridorRoomTemplates()
    {
        var defaultRoomTemplates = LevelGraph.CorridorIndividualRoomTemplates;
        var defaultRoomTemplatesFromSets = LevelGraph.CorridorRoomTemplateSets.SelectMany(x =>
x.RoomTemplates);

        return defaultRoomTemplates.Union(defaultRoomTemplatesFromSets).ToList();
    }
```

Custom input implementation

Custom inputs are quite similar to Custom post-processing logic. We have to create a class that inherits from `DungeonGeneratorInputBase`. And because the base class is a ScriptableObject, we need to add the `CreateAssetMenu` attribute, so we are able to create an instance of that ScriptableObject. The `DungeonGeneratorInputBase` class has one abstract method that we need to implement - `LevelDescription GetLevelDescription()`:

```
// [CreateAssetMenu(menuName = "Dungeon generator/Examples/Docs/My custom input task", fileName =
"MyCustomInputSetup")]
public class CustomInputExample2 : DungeonGeneratorInputBaseGrid2D
{
    /* Create the level description here */
    protected override LevelDescriptionGrid2D GetLevelDescription()
    {
        /* ... */
    }
}
```

After we implement the logic, we have to create an instance of that ScriptableObject by right-clicking in the project view and *Create → Edgar → Examples → Docs → My custom input*. And the last step is to switch the *Input Type* in the generator inspector to *Custom Input* and drag and drop the ScriptableObject instance to the *Custom Input Task* field.

Typical use cases

Add rooms to the level graph

One typical use case is adding additional rooms (for example a random secret room) to an existing level graph. The workflow is usually as follows:

1. Create the static part of the level graph in the GUI
2. Create a custom input task with a public level graph field that we will assign our level graph to
3. Convert the `LevelGraph` to `LevelDescription` (as discussed above)
4. Create additional rooms and connect them to existing rooms in the level description

To make it easier to work with the graph of rooms and connections, `LevelDescription` has a `IGraph<RoomBase> GetGraph()` method to get the current graph of rooms. The graphs contain all the expected methods like getting all rooms or checking if two rooms are neighbours.

For an example of how can this be implemented, see the Enter the Gungeon example where we connect a secret room to a random room in the graph.

> **Note:** The graph which is returned by the `GetGraph()` method currently does not get updated when you modify the level description. You need to call the method again to get a new graph.

Assign room templates automatically

Another typical use case is implementing custom logic for assigning room templates to individual rooms. For example, if we use custom rooms, we may want to assign room templates based on the type of the room instead of manually assigning room templates to individual rooms. This can be seen both in Enter the Gungeon and Dead Cells examples.

Procedural graphs

It is also possible to have a completely procedural structure of levels by creating the whole level description on the fly without any static parts.

FAQ

This document contains solutions to common questions that are often asked on Discord and other channels.

Table of Contents

How to spawn player in a specific room

The easiest solution is to design a special *spawn* room template and place the player prefabs inside that room template. Then, make this room template the only room template for the *Spawn* room in your level graph. This approach is described in Example 1.

Another option is to move your player in a post-processing logic. Instead of putting the player prefabs inside the *Spawn* room template, you can just mark the spawn spot with an empty game object. Then, after a level is generated, you can run a post-processing script that will move the player to the marked position. This approach is described in the Dead Cells example.

What to do with a `TimeoutException`

Sometimes, when you want to generate a level, you get a `TimeoutException` in the console instead. The error means that the generator was not able to produce a level in a given time limit which is 10 seconds by default. The error can have two different meanings:

- the level graph is too hard for the generator (there are too many rooms, too many cycles, restrictive room templates, etc.)
- or there is a problem somewhere in the configuration (maybe the doors of two neighbouring room templates are not compatible)

Usually, it is the second case. To help you fix the error, the generator dumps some diagnostic information *above* the error in the console. The type of information that you can find in the console is for example that the lengths of doors are suspicious or that there are maybe too many rooms in the level graph.

If you are not able to fix the problem yourself, come to our Discord and I will try to help. Also, you can read the Performance tips page.

How to deal with wider walls

Some tilesets have walls that are wider than a single tile. If you try to approach this scenario the same way as 1-wide walls, you will find out that the corridor will not go through the whole multi-tile-wide wall but only through the first tile. The solution for this problem is using the *Outline Override* feature to modify the outline of corridors. This setup is described in the Example 2 tutorial where there is a tileset with an additional layer of wall tiles on top of horizontal walls. (The most important part is the Vertical corridos section.)

Changes to a room template are lost after a level is generated

It often happens that you want to change the default structure of a room template. Maybe you want to add a collider, add another tilemap layer, or change the properties of the grid like the cell size. But when you hit the *Generate* button, the changes are not there and the level looks exactly like before.

The reason is that after a level is generated, all the room templates are merged to a single set of shared tilemaps. So, if you want to change something, you also have to instruct the generator that the changes should also be applied to the shared tilemaps. You can find a dedicated guide here.

Rooms are generated too close to one another

The generator will often produce a level where the walls of one room are right next to the walls of another room. Usually, this is perfectly okay, but sometimes, it may cause problems. For example, if rule tiles are used for walls, a room that is too close may interfere with tiles in a different room.

To solve this problem, look for the `Minimum Room Distance` field in the dungeon generator inspector and increase the value to *2*. You can increase the value even more for even larger distance between rooms. Just keep it mind that every increase of this field makes it harder for the generator to produce a level (and can, therefore, lead to timeouts). Read more about the `Minimum Room Distance` config here.

Send the same level to multiple players (in a multiplayer game)

If you want to use the generator in a multiplayer scenario, you might wonder how to distribute the same level to all the players. The easiest solution is to send the **seed** of the generator to every player and then run the generator with the seed on all clients. You can read more about the seed here and how to see the seed programmatically here.

Keep prefab references when generating levels inside editor

There is a dedicated guide here.

Keep prefabs in editor

After a level is generated, each room template prefab that was used in the level goes through the `Object.Instantiate()` method. This method removes the connection to the original prefab, which basically means that the whole prefab is unpacked (described here in Unity docs.) However, it might be sometimes useful to keep the references to the prefabs. For example, if the level generator is used in the Editor and then manual changes are made to that level.

Solution

There is currently no simple solution that would require just ticking some checkbox in the Editor. However, it is possible to change the default behaviour by changing a line or two directly in the source code of the asset.

Locate the `GeneratorUtilsGrid2D` class and inside the class there is the `InstantiateRoomTemplate` (as shown in the source code below). In that method, there are two constants which you can change to keep prefab references when generating levels directly in the Editor.

```
private static GameObject InstantiateRoomTemplate(GameObject roomTemplatePrefab)
{
    // Set to true if you want to keep prefabs when generating levels in the editor
    const bool keepPrefabsInEditor = false;

    // Set to true if you want to unpack the root game object of the prefab
    // (keepPrefabsInEditor must be true for this constant to change anything)
    const bool unpackRootObject = false;

    // ReSharper disable once ConditionIsAlwaysTrueOrFalse
    if (!Application.isPlaying && keepPrefabsInEditor)
    {
        #if UNITY_EDITOR
        var roomTemplateInstance = (GameObject) PrefabUtility.InstantiatePrefab(roomTemplatePrefab);

        if (unpackRootObject)
        {
            #pragma warning disable CS0162 // Unreachable code detected
            PrefabUtility.UnpackPrefabInstance(
                roomTemplateInstance,
                PrefabUnpackMode.OutermostRoot,
                InteractionMode.AutomatedAction
            );
            #pragma warning restore CS0162 // Unreachable code detected
        }

        return roomTemplateInstance;
        #endif
    }
    return Object.Instantiate(roomTemplatePrefab);
}
```