DUNGEN 2.11.2

DunGen is a Unity editor extension that lets you generate procedural dungeon-style layouts at runtime quickly and easily.

The extension uses a room-based approach that allows you to design and create segments of a level as you usually would inside Unity. Simply create a prefab that represents a "room" in your dungeon, specify doorway positions, and let DunGen generate your random dungeons.

With an easy-to-use graph-based interface, you can control the flow of your dungeon, placing specific rooms, or varying the type of dungeon that is placed along the main path.

CONTENTS

INTRODUCTION

Terminology

Here is a list of the terminology used throughout the software and this documentation that you'll need to know to fully understand how to use DunGen:

 **Tile**	The building blocks of your dungeon that you design for DunGen to later piece together. Think of them as rooms
 **Tile Set**	A collection of *Tiles*. You can organise these however you like.
 **Archetype**	A section of a dungeon. Describes which *Tile Sets* can be used, how the dungeon branches, etc. This is used to split your dungeon layout, for example, into multiple distinct themes.
Dungeon Flow	Describes the overall structure of your dungeon layout using a flow graph; in which nodes represent a *Tile Set* and the connections between them represent *Archetypes*.
Dungeon	The actual final layout that DunGen has created and spawned in the scene.
Main Path	The primary route or "hot path" through your dungeon that goes from the start tile to the goal tile.
Branch Path	One of any number of optional branches coming off the main path. These are not meant to be critical to gameplay.

Overview

DunGen pieces together hand-designed **Tiles** per certain rules set by the user. Here is a breakdown of the steps DunGen takes when creating the dungeon:

1. The Main Path

When told to generate a dungeon layout, DunGen first creates the **Main Path** by running through the **Dungeon Flow** from start to finish. Each step adds a new tile adjacent to the one created in the previous step, constructing the dungeon one tile at a time. Since the **Main Path** is considered critical to gameplay, if DunGen fails to generate it following the constraints set, it will backtrack or restart until it succeeds (**NOTE:** In the editor, DunGen will give up if it fails to generate the dungeon a certain number of times to avoid running into an infinite loop).

2. Branch Paths

Once the **Main Path** has been generated, DunGen loops back through the dungeon adding **Branch Paths** per the settings provided by the **Archetype**. If DunGen fails at any point when generating a branch, it will consider the last **Tile** placed to be the end of that branch, and move on.

3. Post-Processing

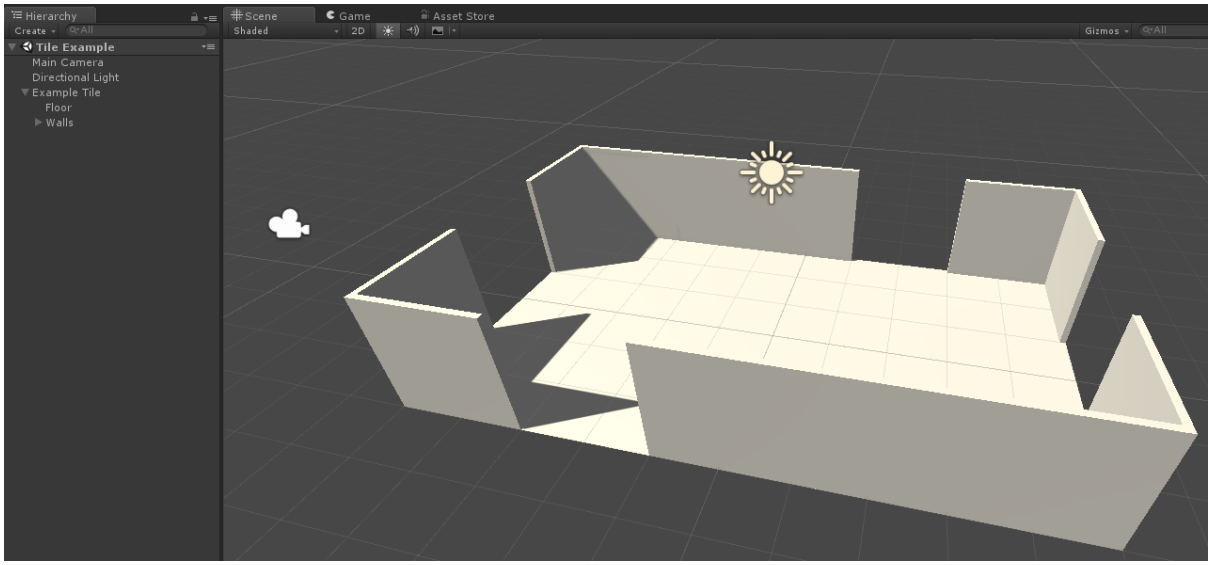
During this phase DunGen places all the props and runs the Lock & Key system to place locked doors and corresponding keys. This is also the time where any custom post-process steps are run.

QUICK START

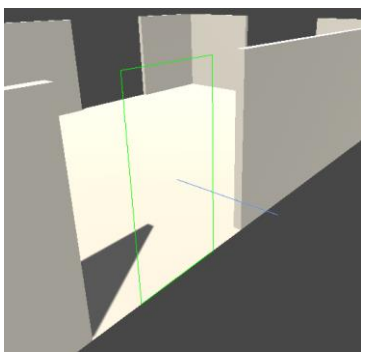
Tiles

First, we'll create a very simple tile. Start by creating an empty GameObject and give it an appropriate name – our room will be contained within this GameObject and we'll later turn it into a prefab. Setup up the basic geometry of the room, but leave gaps where you want doorways to potentially spawn in the future. Not all of them will be used as doorways, so feel free to add more than you need.

IMPORTANT: Doorways must appear on the edge of the Tile's AABB (Axis-Aligned Bounding Box). For more information, see the Limitations section.

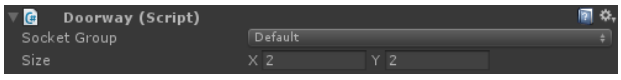


Now we need a way to tell DunGen where we want our doorways to be. To do this, we'll create another empty GameObject and call it "Doorway"; be sure to parent it to your tile GameObject. Now we'll add a **Doorway** component to our new GameObject (*Add Component > DunGen > Doorway*).



The doorway should be placed on the very edge of the tile with its local Z-axis (shown by the blue line in the editor) pointing outwards. Also, if you're working in 2D, make sure the door is correctly oriented to match your up-direction.

The green rectangle represents the extents of the doorway and can be adjusted using the Size property in the inspector.



Socket Types

Each doorway has a *Socket Group* property which is used to determine how doorways are matched together. By default, doorways can only be matched together if they both belong to the same Socket Group; this allows you to, for example, have doorways of different sizes and ensure that they're no incorrectly linked together. If all your doorways are the same size, you can safely use the "Default" group for all your doorways.

```
public enum DoorwaySocketType
{
    Default,
    Large,
    Vertical,
    MyNewSocketGroup,
}
```

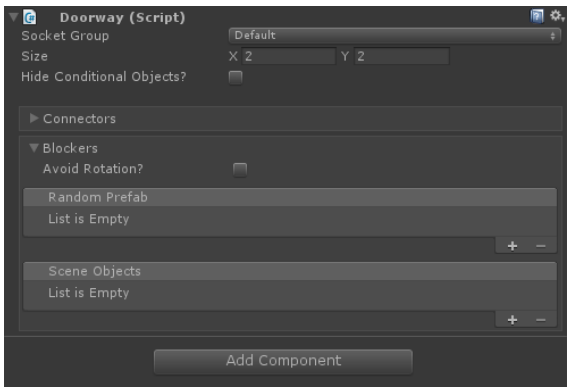
If additional Socket Groups are required, they can be manually added to the *DoorwaySocketType* enumerator in "DunGen/Code/DoorwaySocket.cs" as shown in the example to the left.

The default matching behaviour can also be overridden in code by replacing the included ***IsMatchingSocket()*** method found in the same file - but for most cases, the default behaviour should suffice.

Adding & Removing Doorway-Related Objects

At this point, DunGen can use this doorway as a socket to attach two tiles together – we've set up a very minimal doorway and this is where we could stop if we want; However, we'd like DunGen to block-off unused doorways so we don't have gaps in the walls. This can be a wall, a closed-door mesh, a pile of debris, a bookcase, or anything else that can be used to block the player.

From the doorway inspector, we can add **Connectors** or **Blockers**. Connectors are objects that are meant to be present when the doorway is in use (when it is connected to another tile), Blockers are objects that are present when the doorway is **not** in use.



Let's add a wall to be placed when the doorway is not in use. Expand the **Blockers** section in the inspector; you'll notice there are two ways to add blockers (the same is true for connectors): as a prefab to be spawned at the doorways position, and from a list of objects that already exist in the scene.

Random Prefab

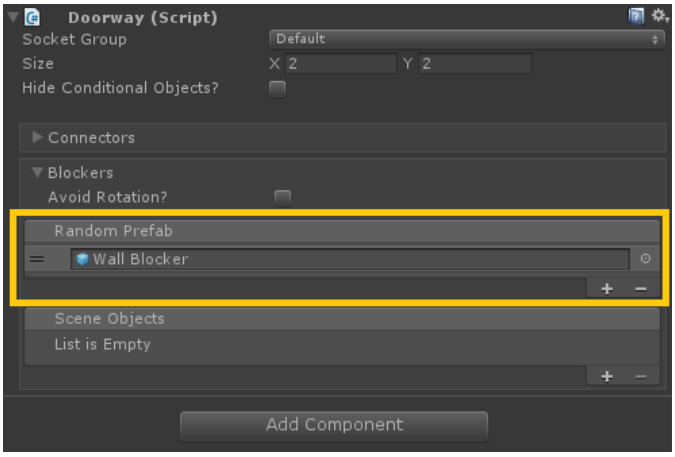
This is a list of prefabs from which one will be selected at random to be used for this doorway when it is not connected (the opposite is true for prefabs under the **Connectors** category). Only one prefab will be selected per doorway-pairing. The prefab will be instantiated at the doorway's position, and at the doorway's rotation unless "Avoid Rotation?" is checked.

Scene Objects

This is a list of GameObjects in your Tile itself, all of which will be kept if the doorway is not connected. They will be destroyed if the doorway is connected to another tile (the opposite is true for scene objects under the **Connectors** category).

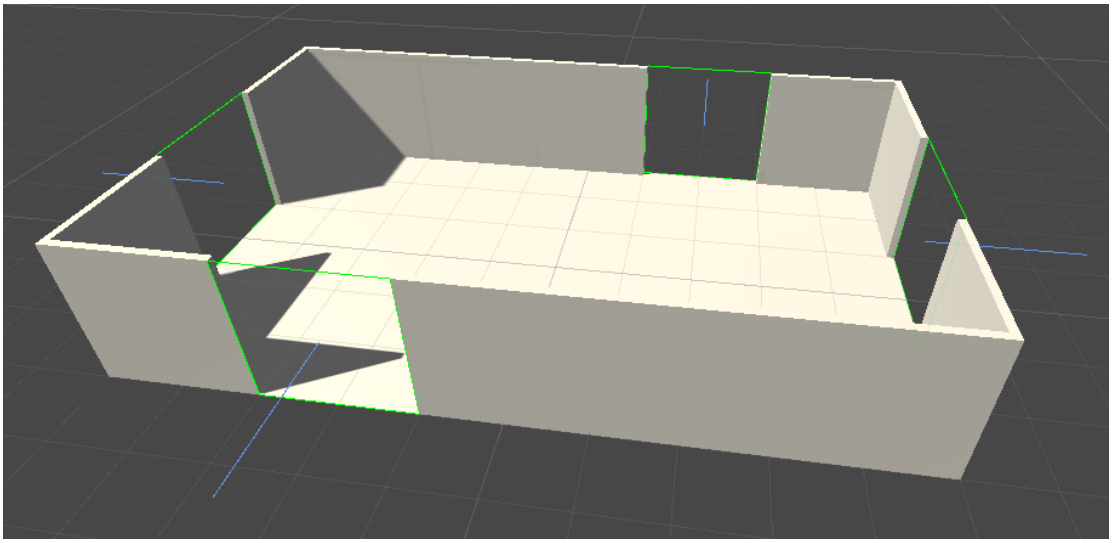
For this example, we'll use the prefab approach. Let's create a wall that fits the gap created by the open doorway, and save it as a prefab named "Wall Blocker".

Back in the doorway inspector, add a new entry to the "Random Prefab" list and drag your newly created wall prefab into the slot.



We could repeat the same process for the **Connectors** category to add an object that should be spawned when the doorway is in use, such as a door frame or even a functioning door, but we'll skip that for now.

Now we'll duplicate our door GameObject and fill all the remaining gaps in our tile. Once that's done, we've completed our first functioning tile for our dungeon. Now we can save it as a prefab and move on.

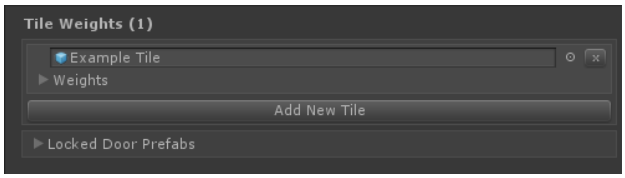


Dungeon Flow

> **NOTE:** We'll be glossing over a lot of stuff in this section in favour of getting DunGen to create something for us more quickly. The settings we skip will be covered in the next section: Customisation.

Now that we have at least one tile, we'll want to create a Dungeon Flow asset to tell DunGen how to build our dungeon layout. Before we do that though, we'll need two new assets, a **Tile Set** and an **Archetype**.

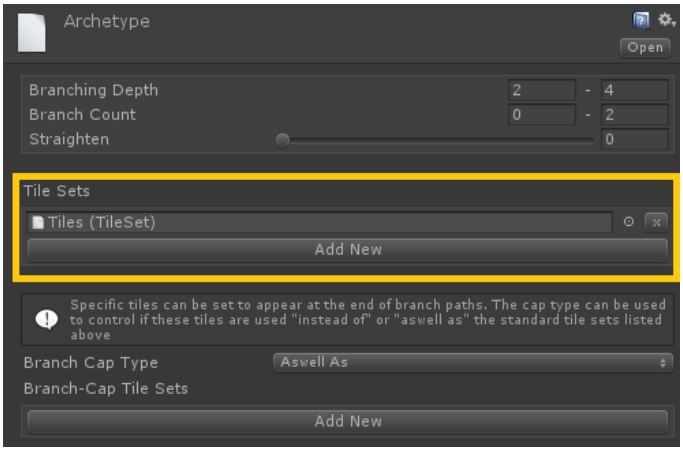
Right-click in the Project view and select "Create > DunGen > Tile Set", we'll just call it "Tiles" for now. Select the Tile Set and in the inspector, click the "Add New Tile" button and drop your tile prefab into the slot. We don't need to worry about any of the other settings for now.



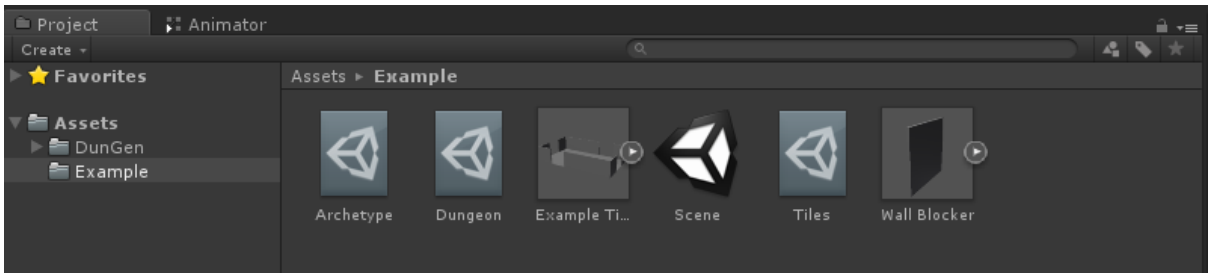
Right-click in the Project view again and select "Create > DunGen > Dungeon Archetype". We'll call it just "Archetypes" for now. Select the archetype and in the inspector, under "Tile Sets", click the "Add New" button and drag in the tile set we made in the previous step.

There are many more options here, but as before we'll skip over them for now. We have one final asset to create: The Dungeon Flow that ties them all together.

Right-click in the Project view once again and select "Create > DunGen > Dungeon Flow" and just call it "Dungeon" for now.



At this point, you should have several different assets ready for DunGen: your tile prefab, wall prefab, a Tile Set, an Archetype, and a Dungeon Flow. Now all we need to do is tie them all together.



Select your **Dungeon** asset and in the inspector, click “Open Flow Editor” and you’ll be presented with a window with a simple line which looks something like this:

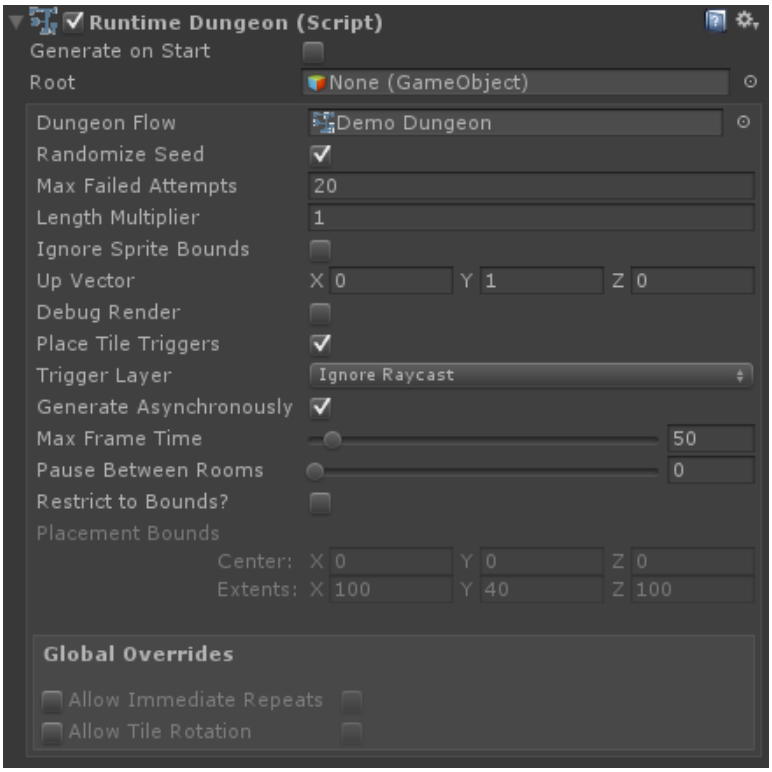
The line down the centre represents a string of tiles to be placed by DunGen. The nodes at either end represent a single tile that DunGen will place. We’ll go into more detail on this in later sections.

Clicking on any node will allow you to assign any number of tile sets to them in the inspector. For now, we’ll assign our only tile set to both the start and goal nodes. Similarly, clicking on a line segment will allow you to assign any number of archetypes to them in the inspector. Let’s assign our one and only archetype to the line segment.

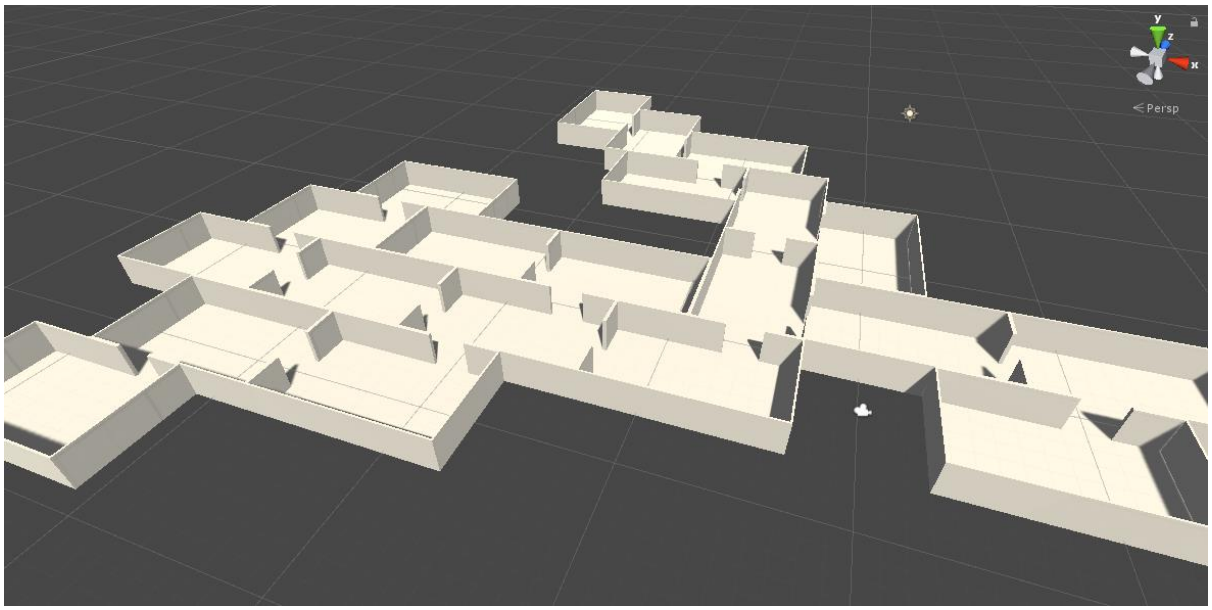
You can right-click anywhere on the line to split the line segment or add a new node, which allows for a lot more complex customisation of the dungeon layout, for now though, this basic setup will work.

Scene Setup

There's one final step to complete before we can have DunGen generate our new dungeon layout and that is to set up the scene with a dungeon generator. Add a new empty GameObject and name it "Generator", then add a new "Runtime Dungeon" component to this GameObject (*Add Component > DunGen > Runtime Dungeon*).

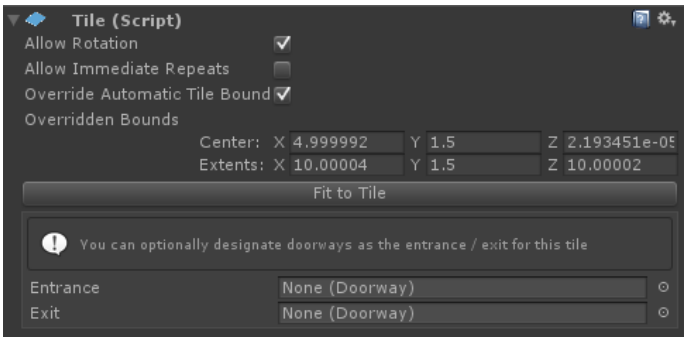


There are a lot of settings here, but for the moment we're only concerned with two of them. Drag your dungeon asset into the "Dungeon Flow" slot and ensure "Generate on Start" is checked. Now when we hit play, our simple dungeon layout will have been generated.



CUSTOMISATION

Tile Component

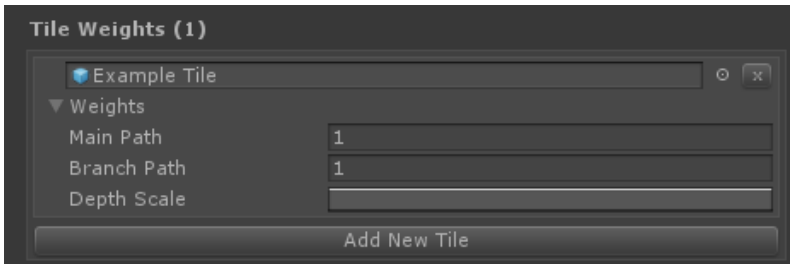


When creating a tile prefab, it's possible to manually add a **Tile Component** to the root GameObject. DunGen does this automatically if there isn't one already present, but doing so manually allows for some tweaking of settings that otherwise wouldn't be possible.

Allow Rotation	If checked, DunGen can rotate the tile to match the adjacent tile's doorway. Uncheck this if your tile doesn't work when rotated. (e.g. it's isometric or has a specific back-wall) NOTE: The dungeon generator has a global override for this setting so you don't need to set it up on a per-tile basis if you need it for all tiles.
Allow Immediate Repeats	If checked, DunGen is allowed to place duplicates of this tile back-to-back. With this unchecked, you won't see the exact same room connected to itself.
Override Automatic Tile Bounds	By default, DunGen automatically calculates the bounds of each tile by its contents. If for whatever reason DunGen's automatic bounds aren't working for you, you can check this box to manually set the bounds yourself.
Overridden Bounds	*[Only visible when "Override Automatic Tile Bounds" is checked]* The manual bounding volume to use to check for collisions between tiles. You can hit the "Fit to Tile" button to see what DunGen thinks the tile bounds should be.
Entrance	A doorway optionally designated as the tile's entrance. If set, this doorway will always be used as the entry point for the tile. If unset, any doorway can be used as the entrance.
Exit	A doorway optionally designated as the tile's exit. If set, this doorway will always be used as the first exit point for the tile. If unset or once the exit is already in use, any doorway can be used as an exit.

Weighting

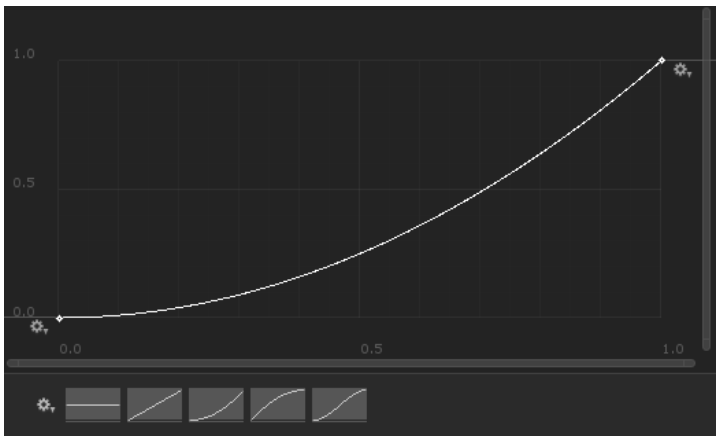
The concept of weights makes an appearance in multiple places throughout the software. Weights determine how likely some action is to occur relative to all other actions in the list.



For example, a **Tile Set** contains a list of **Tiles**. The likelihood of each tile being chosen to be placed is based on its weight relative to all the other tiles in the set.

There are two main components to a weight in DunGen: the base weight value - which is configurable for both the main and branch paths (to allow for different chances based on whether we're on the main path or one of the branches) – and the depth scale, which modifies the two base values based on how deep we currently are into a path.

If two tiles both have a weight of 1, they are equally likely to be chosen; this is true if both tiles have a weight of 0.3, or any other number since weights are relative to one another. The depth scale provides a curve which is multiplied by the base weight value based on how deep we are into a given path.

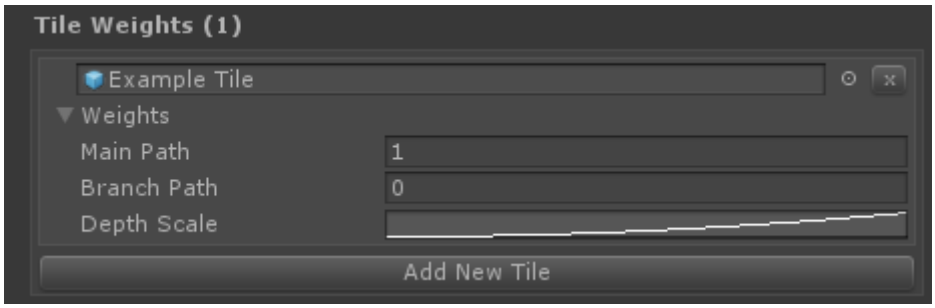


The X-axis of the graph represents the normalised (0-1) depth of the path we're currently on. If we're on the main path this is simply how far we are through the dungeon (the start tile has a depth of 0, the goal tile has a depth of 1). If we're on a branch path however, the normalised depth **through the branch** is used instead (where 1 is the end of the branch).

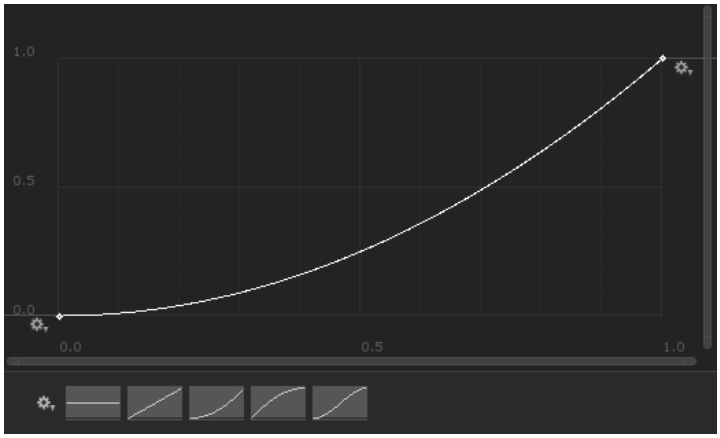
Tile Sets

Tile Sets are an arbitrary collection of tiles that you can organize however you like to fit the specific needs of your project. For example, you might have a few tiles which you want to be chosen at random to be used as the start tile, the tile the player starts in. You can create a tile set named “Start Tiles” and assign it to the start node in your **Dungeon Flow** asset, now only tiles from that set can be used as the opening of your dungeon.

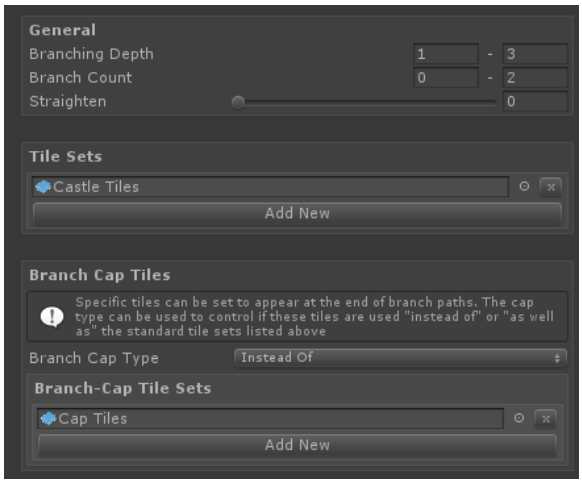
Each tile in a set can also have an assigned weight (see previous section) to allow for alterations to the likelihood of a tile being placed based on its location in the dungeon. For example, you could have a tile spawn only on the main path, and more frequently towards the end of the dungeon by using the following settings:



And for the depth scale:



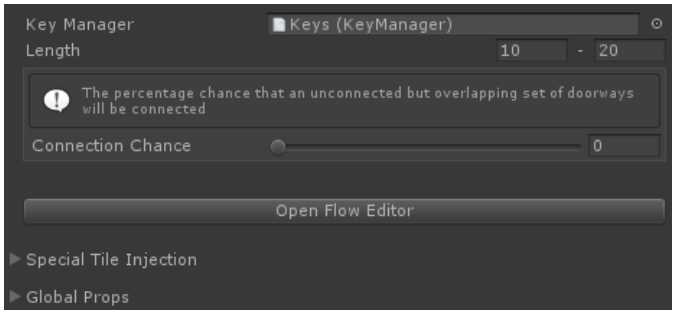
Archetypes



Archetypes are a collection of tile sets which represent a section or theme of your dungeon. Archetypes allow you to split your dungeon layout along the main path into multiple sections, each with its own settings.

Branching Depth	A range representing the minimum and maximum branching depth of this archetype. This is how long any particular branch can be. Note: The minimum value is not a rigid constraint and DunGen can ignore it if its incapable of adding a branch path that's long enough.
Branch Count	A range representing the number of branch paths that can originate from a tile on the main path. Setting these both to zero will result in a dungeon segment with no branching paths. Note: The minimum value is not a rigid constraint and DunGen can ignore it if its incapable of adding enough branch paths.
Straighten	The higher this value is, the more DunGen will attempt to lay new tiles out in a straight line.
Tile Sets	A list of all the tile sets to be considered when placing new tiles in a section of dungeon covered by this archetype.
Branch Cap Type	The method used to combine tile sets in conjunction with the "Branch-Cap Tile Sets" list below. **Instead Of** – Only tile sets from the branch-cap list below will be considered **As Well As** – The branch-cap list will be combined with the normal tile sets list for branch cap tiles
Branch-Cap Tile Sets	Specify a set of tile sets that can be used specifically as the final tile in a branch path. Note: DunGen doesn't guarantee that branches will be of the desired length so there are times where a branch tile will not a tile set from this list.

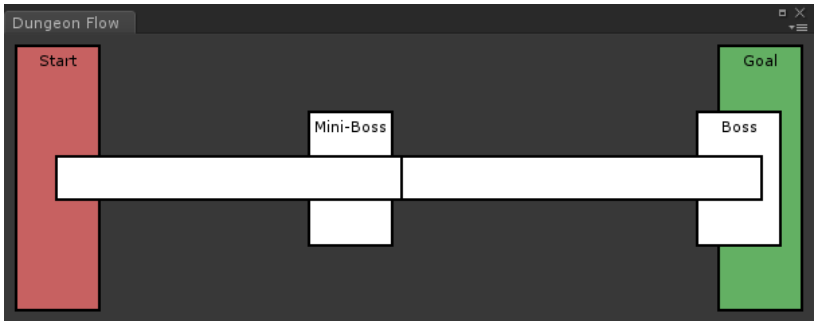
Dungeon Flow



The dungeon flow ties all the pieces together and describes how DunGen should lay out the dungeon. Most of the settings here are covered in other sections, see the **Lock & Key**, **Injecting Special Tiles**, and **Using Props for Variety** sections for more information.

Length	A range of values representing the minimum and maximum length of the main path. This will determine the length of dungeon generated.
Connection Chance	While DunGen can't deliberately create looping paths, if two otherwise unconnected doorways happen to overlap by chance, this value will determine the likelihood of them being connected. Note: This can potentially result in unintended shortcuts through the main path of the dungeon.

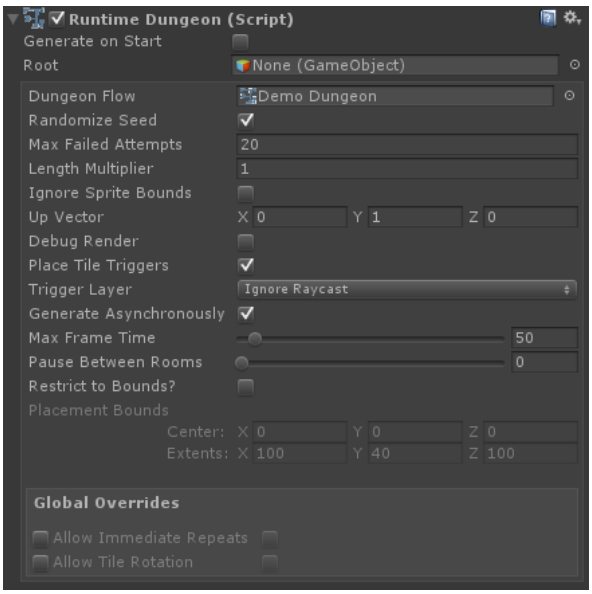
Flow Editor



Using the dungeon flow graph editor, it's possible to create more complex dungeon layouts such as the one to the left (taken from the included demo scene).

In this dungeon, we first have a "Crypt" section of tiles which lasts for half of the main path, at which point a mini-boss room is encountered before continuing into the "Castle" section for the remainder of the dungeon until we reach a boss room, immediately followed by the goal.

Dungeon Generator



The dungeon generator is responsible for taking the rules outlined by all the other asset types and actually creating the dungeon layout. In previous sections, we saw the **Runtime Dungeon** component, which is the most common method of utilising the dungeon generator. It's also possible to manually create an instance of the *DungeonGenerator* class through code, but that really shouldn't ever be necessary.

There's one final way to utilise the dungeon generator and that's in-editor by going to "Window > DunGen > Generate Dungeon" in the menu bar. This will bring up a window that will allow you to generate a dungeon layout directly into the scene in Unity.

Generate on Start	If checked, the dungeon layout will be generated as soon as the game starts.
Root	The root GameObject that the dungeon layout will be parented to. If not set, the generator will create its own GameObject named "Dungeon".
Dungeon Flow	The dungeon flow asset used to generate the dungeon.
Randomize Seed	If checked a completely random dungeon will be generated.
Seed	If "Randomize Seed" is not checked, this seed will be used to generate the dungeon layout. Generating the dungeon again in the future with the same seed will produce exactly the same layout. Note: Changing any settings such as tiles, tile sets, archetypes, etc. will result in a different dungeon.

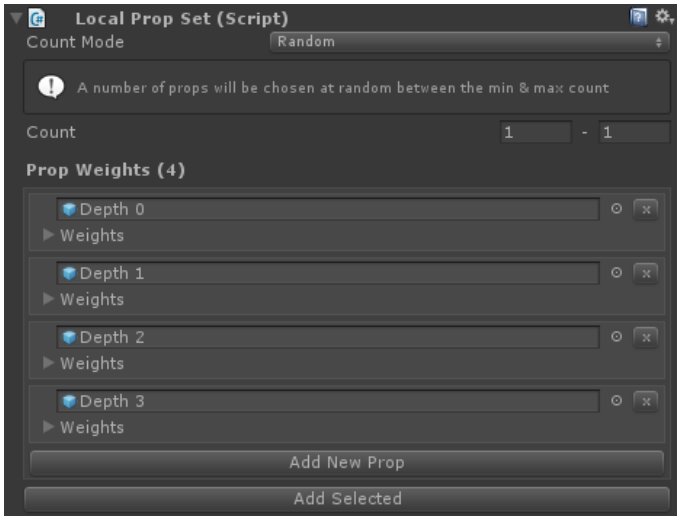
Max Failed Attempts	The maximum number of times DunGen is allowed to fail at generating a dungeon layout before it's forced to give up. This is for the editor only, in a standalone build, DunGen will keep trying indefinitely.
Length Multiplier	Used to alter the length of the dungeon without modifying the Dungeon Flow asset. A value of 1 will use the normal length, a value of 2 will double the main path length, etc.
Ignore Sprite Bounds	When calculating bounding boxes for tiles, if this is checked, sprites will be ignored.
Up Vector	The up direction of the dungeon. This won't actually rotate your dungeon, rather it's used in conjunction with vertical doors to ensure that they are placed correctly. This needs to match the expected up-vector of your dungeon layout – usually (0, 1, 0) for 3D and (0, 0, -1) for 2D.
Debug Render	If checked, coloured boxes will be rendered around tiles in the scene view window representing that tiles position in the dungeon. **Main path** tiles will be coloured **red** to **green** based on normalised depth along the main path. **Branch path** tiles will be coloured **blue** to **purple** based on normalised depth along the branch they are on.
Place Tile Triggers	If checked, DunGen will automatically place box-colliders as triggers around every one of the tiles in the dungeon. Can be used in conjunction with the **DunGen Character** component to be notified when the player passes between tiles.
Trigger Layer	The layer to place the tile trigger on if "Place Tile Triggers" is checked.
Generate Asynchronously	If checked, DunGen will generate the dungeon layout over multiple frames so as to not block Unity's main thread. Allowing for animated loading screens to play while generating. Note: This will increase the overall generation time slightly.
Max Frame Time	The maximum number of milliseconds-per-frame DunGen can spend generating the dungeon layout. Lower values result in more responsiveness from the game at the cost of longer generation times.
Pause Between Rooms	An optional delay (in seconds) between placing rooms. Useful for visualising and debugging the generation process.
Placement Bounds	If enabled, DunGen will not place any tiles outside of these bounds. Enabling this option can increase generation times.

BEYOND THE BASICS

Using Props for Variety

Now that the dungeon layout is complete, it would be nice to add some variety. We could do this by just creating more tiles; another option however, it to add variety to our existing tiles by randomly changing some of their contents; we can do this using props. There are three types of prop in DunGen: Local Prop Set, Random Prefab, and Global Prop.

Local Prop Set



A Local Prop Set is a collection of objects inside a tile of which a certain number are chosen at random to be kept, the rest are discarded.

We can make use of a Local Prop Set by overfilling our tiles with props/furniture, and have this component discard some of it at random.

We can add a Local Prop Set component to any GameObject in our tile (*Add Component > DunGen > Random Props > Local Prop Set*)

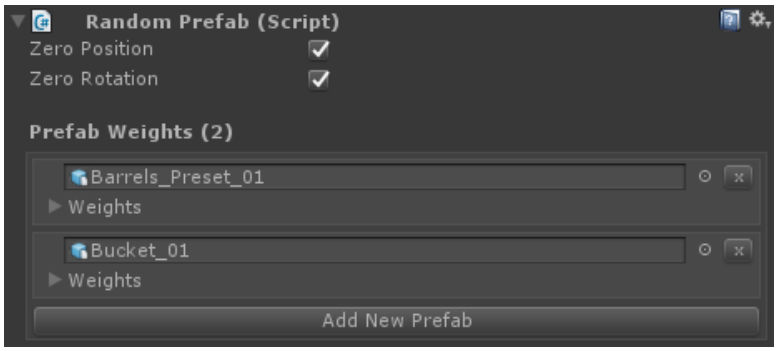
GameObjects in the tile are added to the "Prop Weights" list; note that each prop has its own weight as described in an earlier section.

Above this is a "Count" range which determines the minimum and maximum number of the listed props that are kept when the tile is placed. There's also a "Count Mode" which controls the logic behind how many props are selected to be preserved. These are:

Random	A random number between min & max will be selected at random
Depth Based	Provides a curve whose X-axis represents the normalised depth through the path this tile lays on. The Y-axis is used to determine how many props are preserved (where 0 is the min count and 1 is the max count)
Depth Multiply	The number of props to be preserved is selected at random is with the "Random" option, but is then multiplied by a value read from a curve whose X-axis represents the normalised depth through the path this tile lays on.

Random Prefab

The Random Prefab prop type (*Add Component > DunGen > Random Props > Random Prefab*) can be attached to an empty GameObject and will allow us to spawn a single random prefab from the list at the GameObject's current position & rotation.

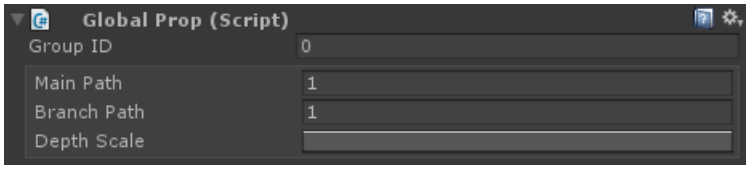


Once again, each entry has an assigned weight as described in an earlier section to allow us to alter how likely a particular prefab is to spawn based on its position in the dungeon layout.

Above this, there are two additional options, "Zero Position", and "Zero Rotation"; if these are unchecked, the prefab will spawn at the GameObject's position & rotation, but offset by the position & rotation of the prefab itself (if any).

Global Prop

Global props are used to tell DunGen that only a certain number of a particular object should be preserved, any additional instances will be discarded. This is useful, for example, if we want to place a shrine object in each of our tiles, but we'd like for only one of them to appear in the generated DunGen.



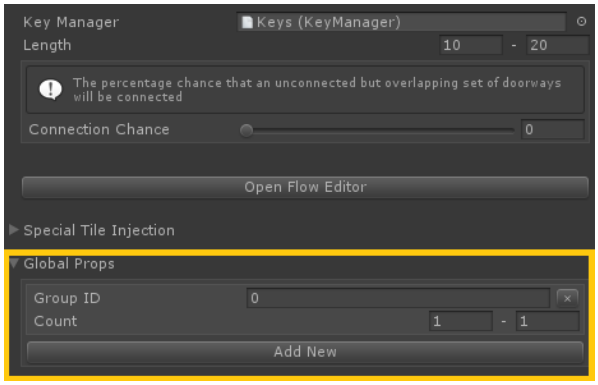
The Global Prop component should be added to a GameObject inside one of our tiles (*Add Component > DunGen > Random Props > Global Prop*)

As with many other components in DunGen, there is a weight to control how the Global Prop reacts to different positioning in the dungeon layout.

The "Group ID" field is the important part here, this is how DunGen identifies an object as being part of a particular prop set (for example, all shrines would be assigned a Group ID of 0; all treasure chests and ID of 1, etc.)

But how do we determine how many of this object to preserve in the dungeon? That's defined in the Dungeon Flow asset under the "Global Props" heading.

Clicking "Add New" will add a new entry to the list and provide us with a field to enter the "Group ID" which must match the one we supplied earlier.

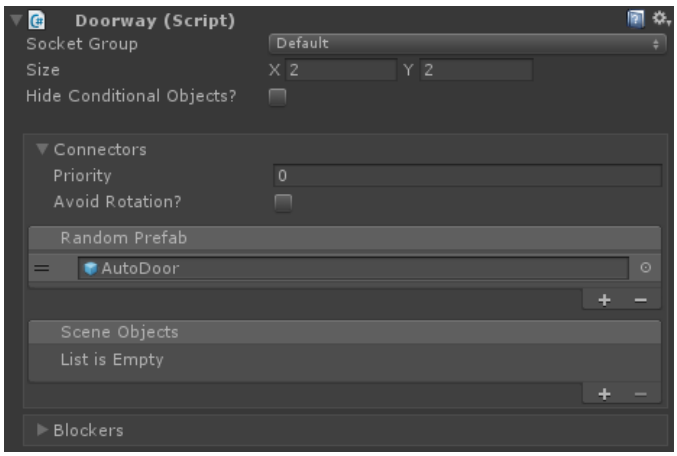


We can also specify the minimum and maximum number of Global Props with the given Group ID that should be present in the dungeon layout. Any additional props beyond this amount will be discarded.

Doors

In the Quick Start section, we added a blocker to our **Doorway** component to plug the gap created when that doorway was not connected to another tile. We can use a similar approach to add a door to our doorway if it **is** connected.

Expanding the "Connectors" category in the doorway component's inspector will reveal mostly the same settings we previously used to add a blocker.



The only addition is the "Priority" field. As only one door prefab is placed per doorway-pairing, there is a chance that both doorways want to assign a different door prefab. This priority is used to determine which prefab to use (the doorway with the highest priority wins. If the priority values are equal, a prefab is picked at random from both doorways)

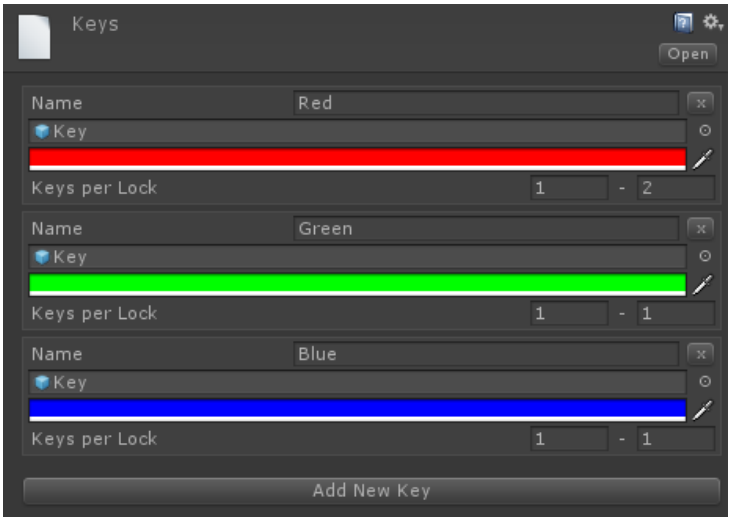
In this case, we've added the "AutoDoor" prefab included in the demo scene to our "Random Prefab" list. Now when this doorway is in use, a door will be placed that automatically opens when the player approaches it.

When DunGen spawns a door prefab (either for a doorway, or using the Lock & Key system as described in a future section), it will assign a **Door** component to the GameObject which - in addition to providing some information about the connection between doorways – contains a ***IsOpen*** property which can be set through code as is done by the aforementioned "AutoDoor" prefab. This property is used by culling & pathfinding integration to make adjustments based on the state of the door so it's important if you're making a custom opening door to correctly assign a value to the door component's *IsOpen* property.

Lock & Key System

The lock & key system requires some programming to integrate into your own game. This section will look at the simple implementation that is used in the included demo scene. DunGen exposes multiple interfaces to help create your own logic for how locked doors and their corresponding keys are handled.

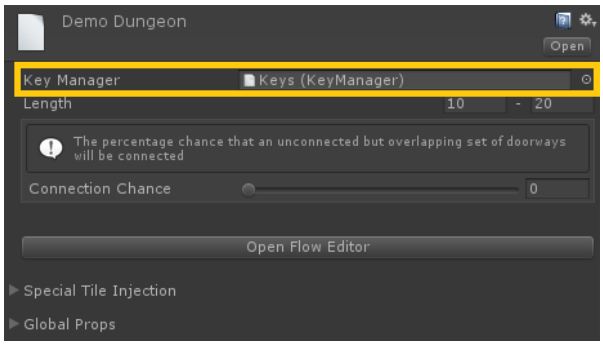
Creating the Key Manager



First, we need a **Key Manager** asset (*Assets > Create > DunGen > Key Manager*) to handle all of the keys we'll be making for our DunGen. With the newly created key manager selected, click the "Ass New Key" button in the inspector. This will give you a new slot that represents a key type. A key type has a few optional properties which will be explained here:

Name	A human-readable name for identifying your key type. This can be accessed at runtime and displayed on screen to the player when picking up keys or approaching locked doors.
Prefab	A prefab representing the key in the scene. This would be used if we want the key itself to be placed as an object in the scene.
Colour	Used for colour-coding our keys if we want the association between lock & keys to be more apparent visually.

All of these settings are optional and whether you want to use them or not will be based on your specific implementation for locks & keys. We make use of all of them in the demo scene as keys are placed as objects to be picked up in the scene and each key-lock pairing is assigned a colour.



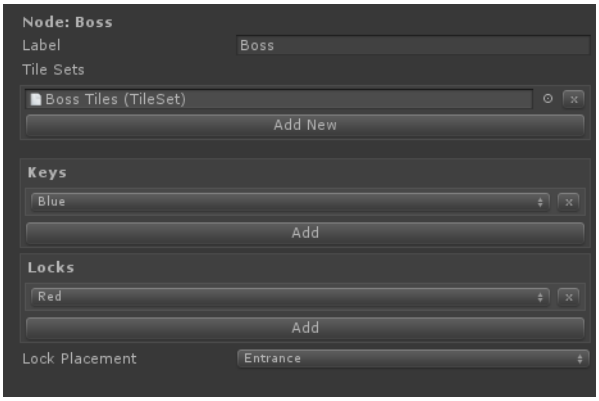
Once we've added all our keys to the Key Manager, we can move on to setting them up in the dungeon flow.

Select your dungeon flow asset and assign the key manager asset we just created. This tells the dungeon flow which keys it has available for use.

Placing Keys into the Dungeon Flow

Click the "Open Flow Editor" button to open the dungeon flow graph where we can start assigning our keys to tiles in the dungeon. As before, selecting any of the nodes or lines in the graph will display their options in the inspector.

Each node and line segment has options for a list of keys and locks to be placed on them.



Nodes will have any keys/locks placed on the single tile that they create in the dungeon layout. Line segments on the other hand will have keys/locks placed at random anywhere along the set of tiles that the line segment represents.

Nodes have an extra option related to keys that line segments don't have: A Lock Placement. This is used to determine where on the tile the locked door can be placed. As in the example above, this allows us to assign the Red lock to the Boss room and ensure that the locked door is only placed on the entrance of the tile, so that the player will need a key found previously in the dungeon to access the boss. Of course, we also need to have assigned the Red key to be placed in a section of the dungeon prior to this node or we'd have a dungeon that isn't completable.

Locks placed on a line segment also have a "Count" field which is used to specify how many locked doors should be placed throughout this section. DunGen will ensure that enough keys are placed prior to the doors to avoid running into a situation in which the dungeon is not completable.

Tying it into our Game

At this point, we've told DunGen where we want the keys and locked doors to be placed, but it doesn't have any idea how we want keys to be handled in our particular game. We might want them to spawn as collectible objects in the dungeon as they do in the demo, we might instead want keys to be able to drop from slain enemies. It's up to us to program the exact integration of the lock & key system in our game; luckily DunGen makes this as easy as possible, all we need to do is implement a few interfaces.

IKeySpawnable

```
public interface IKeySpawnable
{
    void SpawnKey(Key key, KeyManager manager);
}
```

This is how DunGen knows where and how to spawn keys. The interface has a single function: SpawnKey which is called when DunGen finds a place to spawn the key.

If we have a key type named "Red", when DunGen places a "Red" lock on a doorway in the dungeon, it will look through any preceding tile that is marked as being able to contain a red key. From these tiles, a component that implements the `IKeySpawnable` interface will be selected at random, and the *SpawnKey* function will be called. Some possible implementations include:

Key Spawn Point – We could implement a script that acts as the key spawn point (as is done in the demo scene) that simply spawns an instance of the key prefab at the current position when *SpawnKey* is called.

Inventory Spawn – We could have an inventory component for our game on which we implement `IKeySpawnable` to add the key to a monster's inventory when the *SpawnKey* method is called.

> **NOTE:** Locks & Keys are placed in the dungeon after all of the props are processed. This means that we can have chests as random props and DunGen can properly select them as potential key spawn points if they implement the `IKeySpawnable` interface.

IKeyLock

```
public interface IKeyLock
{
	void OnKeyAssigned(Key key, KeyManager manager);
}
```

This interface also has only one method: OnKeyAssigned, which is called when a key type is assigned to a lock or a spawned key.

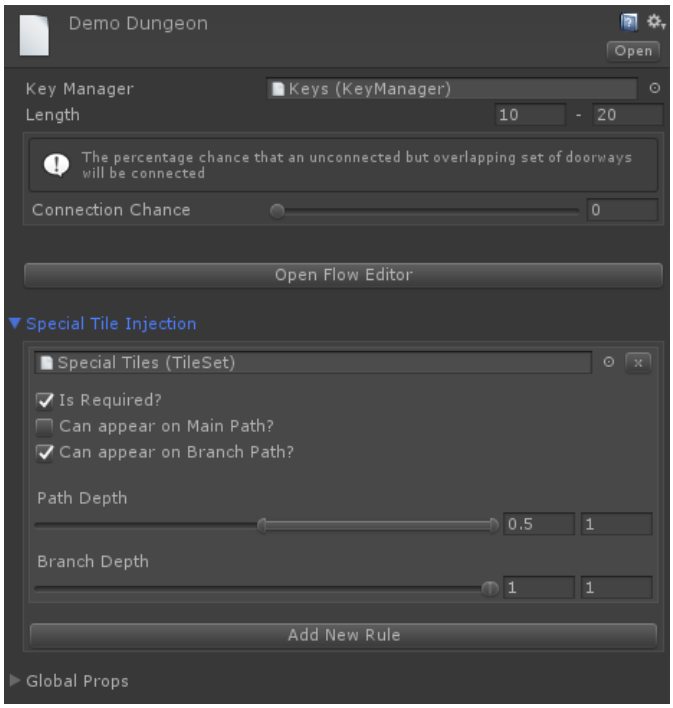
When a lock is placed on a doorway, the OnKeyAssigned method is called on any component that implements `IKeyLock` anywhere in the locked door prefab and any of its children. Similarly, when a key prefab is spawned, the OnKeyAssigned method is called on any component that implements `IKeyLock` anywhere in the key prefab and any of its children.

For the demo scene, a **KeyColour** component which implements this interface was added to both the key prefab and the locked door prefab to visually change the colour of the corresponding meshes to match the assigned key type.

It's also completely up to us how we want to handle the player picking up keys, checking if the player has a key, and unlocking doors if they do. The demo scene provides one example in which keys are spawned as collectible objects in the dungeon itself. When walked into, these keys are placed into a very simple inventory system. Locked doors will open if approached while the player has the corresponding key in their inventory.

Injecting Special Tiles

In some cases, we might want specific tiles to spawn only once throughout the entire dungeon (or other more specialised logic). DunGen handles this by allowing us to "inject" tiles into the generation process. For simple cases, this can be done through the inspector in our Dungeon Flow asset.



Under the "Special Tile Injection" category, click the "Add New Rule" button to add a new tile injection rule.

The first field is used to tell DunGen which tile set to select a tile from. When this rule is consumed, a random tile from this set will be chosen and placed at the specified position in the dungeon.

Is Required?	If checked, this tile must be placed. If DunGen fails to generate a dungeon with this tile successfully injected, it will try again until it succeeds. Note: Using this option can increase generation time, especially if the rule is very restrictive of the tile's possible position.
Can appear on Main Path?	If checked, the tile is allowed to appear at some point on the main path
Can appear on Branch Path?	If checked, the tile is allowed to appear at some point on any of the branch paths
Path Depth	The range of normalised depths along the main path at which this tile can possibly appear (where 0 is the start tile and 1 is the goal tile)
Branch Depth	The range of normalised depths along a branch path at which this tile can possibly appear (where 0 is the first tile in the branch and 1 is the last). This is only used if "Can appear on Branch Path?" is checked

Tile Injection Through Code

For more advanced cases, it may be more desirable to handle rules through code. To do this, simply make a new method matching the `TileInjectionDelegate` and add this to the `DungeonGenerator`'s *TileInjectionMethods* list. The method itself should add new instances of `InjectedTile` to the list that's passed as an argument, for example:

```
private void Start()
{
	var runtimeDungeon = FindObjectOfType<RuntimeDungeon>();
	var generator = runtimeDungeon.Generator;

	generator.TileInjectionMethods += InjectTiles;
}

private void InjectTiles(Random randomStream, ref List<InjectedTile> tilesToInject)
{
	bool isOnMainPath = false;
	float pathDepth = 0.5f;
	float branchDepth = 1.0f;

	var tile = new InjectedTile(MyTileSet, isOnMainPath, pathDepth, branchDepth);
	tilesToInject.Add(tile);
}
```

The above method will add a tile from *MyTileSet* onto the end of the first branch encountered after the half-way point of the dungeon's main path. This example is far too simple to warrant doing through code, but helps to show how to inject tiles into DunGen manually where necessary.

Culling

DunGen includes a basic built-in culling solution which works best for interior dungeons in a first-person perspective, especially if each doorway has an auto-closing door. This is a very use-case specific solution which won't work for all types of game.

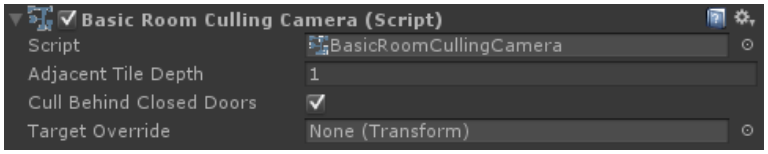
The solution works by first collecting a set of visible tiles; a tile is considered visible if:

- The player is in it

OR

- It is adjacent to a tile the player is in (up to a customisable max adjacent tile depth)

Visible tiles can be optionally removed from this list if they are behind a closed door. Any tile not considered to be visible will be culled. This culling can be enabled by adding a Basic Room Culling Camera component (*Add Component > DunGen > Culling > Basic Room Culling Camera*) to any camera we want the culling to be applied to.



Adjacent Tile Depth	How deep into the adjacency tree should we delve when determining if a tile should be visible. A value of 0 means only the tile the player is in will be visible. A value of 1 means that tile plus all adjacent tiles will be visible, etc.
Cull Behind Closed Doors	If checked, tiles that would otherwise be considered visible will instead be culled if they are behind a closed door.
Target Override	An optional transform component to be used when determining where the player object is for games where the player and camera are physically separate. If not set, the camera's position will be used.

It's also possible to implement 3rd party culling solutions into DunGen. Check the Integration section for any included integrations; if one doesn't already exist, it's possible to integrate it as an adapter by deriving from the `CullingAdapter` base class and implementing the abstract *PrepareForCulling* method.

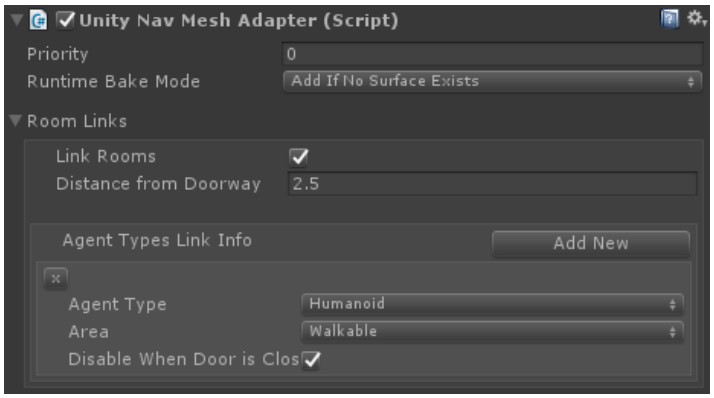
Pathfinding

DunGen comes with built-in integration for Unity's new (as of 5.6) NavMesh component which at the time of writing this, must be downloaded separately from GitHub here:

https://github.com/Unity-Technologies/NavMeshComponents

This integration allows us to add a NavMeshSurface component to each of our tiles and bake them in the editor, or bake new ones at runtime if we need to.

To use this integration, first we must navigate to "Assets/DunGen/Integration" and double-click "Unity NavMesh.unitypackage" to extract it to our project. This will give us access to the component we need.



Now we need to add the **Unity NavMesh Adapter** component to the same GameObject as our **Runtime Dungeon Generator** (*Add Component > DunGen > Pathfinding > Unity NavMesh Adapter*).

Priority	The priority of the adapter, used to determine the order in which multiple adapters are run during DunGen's post-processing step. This can be left alone most of the time.
Runtime Bake Mode	Which mode to use for runtime baking. Pre-Baked Only – Don't add any new navigation mesh surfaces at runtime, only use those already provided and pre-baked in the tiles. This is the most performant solution as no runtime NavMesh building needs to occur. Add If No Surface Exists – If no NavMeshSurface component exists in a tile, a new one will be created and baked at runtime. Always Re-bake - If no NavMeshSurface component exists in a tile, a new one will be created and baked at runtime. Existing, pre-baked surfaces will also be re-baked at runtime.
Link Rooms	If checked, DunGen will place NavMeshLinks between tiles in the final dungeon so that agents can traverse between rooms. If disabled, agents will be confined to the room they start in (unless runtime baking causes the rooms to be connected). Checking this option allows us to have a fully functioning pathing solution that was baked in-editor and has no runtime generation cost.

Distance from Doorway	How far from either side of the doorway should the NavMeshLink connection be placed?
Agent Types Link Info	A separate link can be created for each agent type if desired. We can add new entries to this list to create multiple links. Agent Type – The type of agent this link is meant for (defined in Unity's Agent Settings). Area – What type of surface area should this link have? (defined in Unity's Agent Settings). This can allow for special traversal behaviour when passing through the doorway. Disable When Door is Closed? – If checked, this link will be un-traversable when the door is closed.

It's also possible to implement 3rd party pathfinding solutions into DunGen. Check the Integration section for any included integrations; if one doesn't already exist, it's possible to integrate it as an adapter by deriving from the `BaseAdapter` base class and implementing the abstract *Run* method.

Additional Features

DunGen Character Component

The DunGen Character component (*Add Component > DunGen > Character*) is a small helper component with no settings that can be used in code to receive events when a collider it is attached to passes into another tile, for example:

```
private void Start()
{
	var character = GetComponent<DungenCharacter>();
	character.OnTileChanged += OnCharacterTileChanged;
}

private void OnCharacterTileChanged(DungenCharacter character, Tile previousTile, Tile
newTile)
{
	// Do Something...
}
```

Post-Process Step

Sometimes, it's helpful to be able to apply our own post-processing to the dungeon generation. This is possible using a method in the `DungeonGenerator` class: *RegisterPostProcessStep*

This method accepts three arguments:

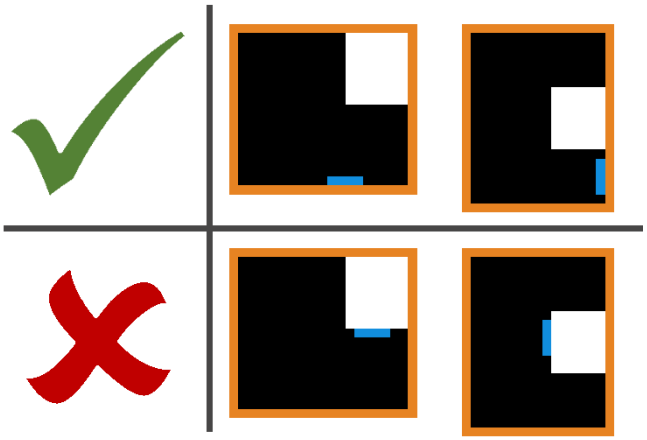
`Action<DungeonGenerator>` **postProcessCallback**	The method to be called when this post-process step is handled by DunGen
`int` **priority**	The priority of this step. Higher priority steps execute first.
`PostProcessPhase` **phase**	Determines in which phase the post-process step should run: before or after DunGen's built-in post-processing is completed.

Alternatively, we can write a custom adapter component derived from `BaseAdapter` and implement the abstract *Run* method. DunGen will run any adapter that is attached to the same GameObject as its **Runtime Dungeon** component whenever dungeon generation is invoked.

Limitations & Considerations

Doorway Placement Restrictions

Collision checking between tiles in DunGen is handled using an AABB (axis-aligned bounding box) around each tile. For this reason, doorways must be placed on the edge of the tile in order to work – the placement doesn't need to be exact - DunGen will correctly handle doorways that are slightly off, but doorways absolutely must not appear on concave portions of a tile.



The above image demonstrates correct doorway placement where the **orange** box represents the tile's AABB and the **blue** line represents the doorway position.

If the tiles in your dungeon are rectangular, you'll never have to consider this restriction; but if you're using more exotic shapes, some additional thought needs to be put into ensuring the doorways are accessible.

Nested Prefabs

While not a limitation inherent to DunGen, it's an issue to be aware of nonetheless. Unit in its current form doesn't correctly handle nested prefabs – that is, when a prefab is placed inside another prefab, the nested prefab is no longer linked to the original and will not have updates applied to it as it should.

As each tile in DunGen must be a prefab, any prefabs that we place in our rooms will be affected by this issue and it makes creating compelling room content that much more tedious. Unfortunately, we don't have a concrete timeframe for when Unity will finally implement nested prefab support so workarounds must be used for the moment.

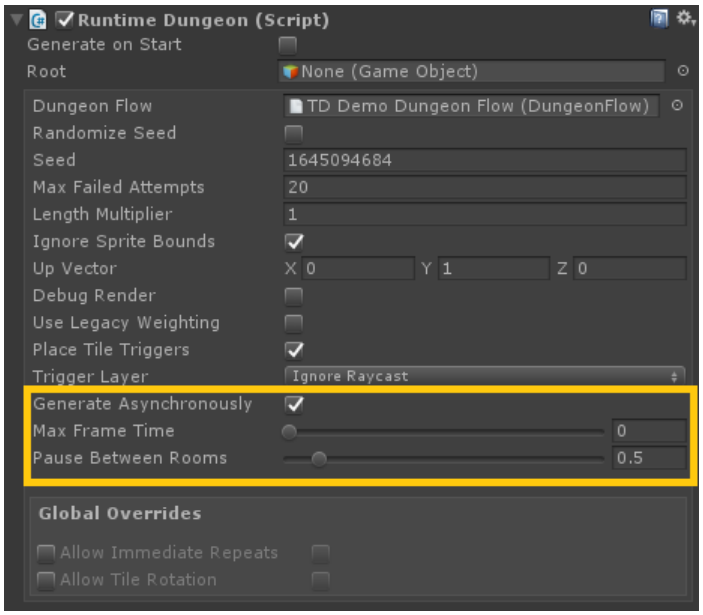
There are several solutions such as Prefab Evolution on the asset store that can be used to alleviate this problem.

Analysing Our Dungeon

While not as important as it was in earlier versions of DunGen, it's still nice to be able to analyse the performance and effectiveness of our dungeon generation. A **Runtime Analyzer** component is supplied (*Add Component > DunGen > Analysis > Runtime Analyzer*) which can be used to repeatedly generate a dungeon and provide us with a collection of pertinent statistics such as how many times the dungeon generation failed, how long each step took on average, etc.

The easiest way to make use of this feature is to use the existing scene provided in the demo "Assets/DunGen/Demo/Analysis Scene" and on the **Analyzer** GameObject, substitute your dungeon flow asset in place of the demo dungeon. The simply hit play and wait until it's finished.

It's also now possible to use the new ability to generate dungeons asynchronously to visualise the dungeon generation process. This can be achieved by changing some settings in the dungeon generator.



Ensure "Generate Asynchronously" is checked, and set an appropriate "Pause Between Rooms" time (in seconds).

Now when you hit play, the dungeon will be generated one room at a time, pausing for the specified number of seconds between each room. This makes it much easier to visualise exactly what DunGen is doing and what might be causing it to fail to generate the dungeon correctly.

INTEGRATION

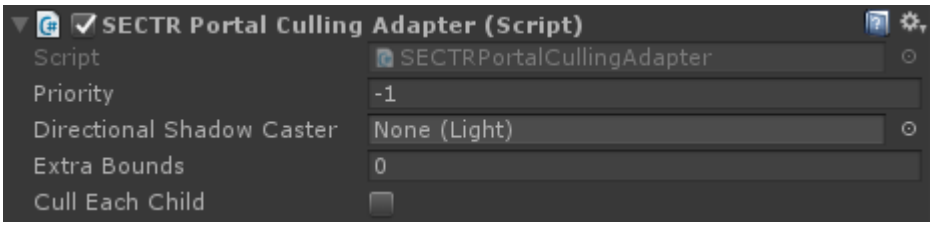
DunGen comes with built-in integration for some popular 3rd party assets from the Asset Store. It's also possible to write your own adapter that DunGen runs during the post-processing step. More information on this process can be found in the Post-Process Step section.

SECTR VIS Portal Culling

Uses SECTR VIS to automatically apply portal culling to our generated dungeon. Causing rooms to be culled if the player cannot see into them to improve rendering performance.

1. First, we need to extract the package at "Assets/DunGen/Integration/SECTR_VIS.unitypackage" into our project.

2. On the same GameObject as our **Runtime Dungeon** component, add a new **SECTR VIS Portal Culling Adapter** component (*Add Component > DunGen > Culling > SECTR VIS Portal Culling Adapter*). The settings on this adapter should mimic what you're used to seeing from SECTR VIS, more information on these settings can be found in the SECTR documentation.



3. Add a **SECTR Culling Camera** component to your player camera

That's it! DunGen will now automatically generate sectors and portals for your dungeons for use with SECTR VIS portal culling.

As mentioned in a previous section, doors placed by DunGen (including through the Lock & Key system) will have a **Door** component attached which can be used to control through code whether the room beyond the doorway should be culled or not. Simply set the *IsOpen* property from your own door script to toggle the portal state.

PlayMaker

Double-clicking the package at “Assets/DunGen/Integration/PlayMaker” will unpack the required files into your project; additional DunGen-related PlayMaker actions should now be available.

Action Nodes

There are currently three PlayMaker actions implemented for use with DunGen:

Generate	Generates a new dungeon layout using settings from an existing **Runtime Dungeon** component in the scene.
Generate with Settings	Generates a new dungeon layout using settings that you specify in the PlayerMaker UI. It’s not necessary to place a **Runtime Dungeon** component in the scene first as one will be created for you if one does not already exist, however you can specify additional settings on an existing **Runtime Dungeon** component in the scene.
Clear	Removes a dungeon layout created by an existing **Runtime Dungeon** component in the scene.

RAIN Navigation Mesh

DunGen can use RAIN's runtime navigation mesh generation function to automatically create a navigation mesh for our dungeon.

Setup

1. Ensure both DunGen and RAIN are imported into your project
2. Double-click the package at "Assets/DunGen/Integration/RAIN.unitypackage" to unpackage the files we'll need
3. Add a **RAIN NavMeshRig** component somewhere in your scene. This can be on any GameObject as long as it's **not** one that holds DunGen's **Runtime Dungeon** component. It's best if this is on an otherwise empty GameObject.
4. Add a **RAIN NavMesh Generator** component to the same GameObject as your **Runtime Dungeon**.

That's it! DunGen will automatically use RAIN to generate a navigation mesh for your dungeon. You can visualise the mesh (in the scene view only) by setting "Display Mode" to "Navigation Mesh" in the **RAIN NavMeshRig** component (the mesh will only be rendered while this GameObject is selected).

> **NOTE:** DunGen's RAIN integration doesn't yet handle opening/closing doors.

A* Pathfinding Project Pro

DunGen can use the runtime navigation mesh generation function of A* Pathfinding Project Pro to automatically create a navigation mesh for our dungeon.

Setup

1. Ensure both DunGen and A* Pathfinding Project Pro are imported into your project
2. Double-click the package at "DunGen/Integration/AStarPathfindingProjectPro.unitpackage" to unpackage the files we'll need
3. Make sure you have an **Astar Path** component (*Add Component > Pathfinding > Pathfinder*) somewhere in your scene. Add a "Recast Graph" to the component.
4. Add a **A* Pathfinding NavMesh Generator** component to the same GameObject as your **Runtime Dungeon** component.

That's it! The navigation mesh will be generated when the dungeon generation is complete.

Handling Doors

Changing path walkability by opening & closing doors can be handled automatically by DunGen with some simple setup:

1. Make a new layer that will be ignored when generating the NavMesh
2. In the **Astar Path** component, make sure your new layer is not selected in the "Layer Mask" for your recast graph
3. Make sure all door prefabs use the new layer you made in step 1
4. In the **A* Pathfinding NavMesh Generator** component, choose the Open and Closed door tags you'd like to use. Any A* AI you add should not be able to traverse the tag you define as your "Closed Door" tag.