Version: 2.9.1

Contents

Terminology

Here are some of the phrases used throughout the documentation and their meaning:

Tile - The building blocks used to create the dungeon. These are the small modular pieces that you design for DunGen to piece together. Users coming from version 1.x will know these as "Tiles".

Tile Set - An arbitrary collection of *Tiles.* You can organize these however you like. Properly utilizing *Tile Sets* will become very important for accomplishing more complex tasks with DunGen.

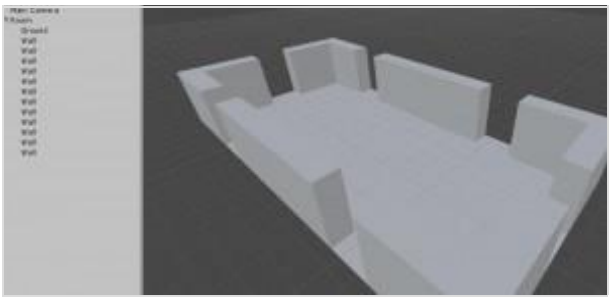
Dungeon Archetype - A description of a type of dungeon (which *Tile Sets* can be used, how the dungeon branches, etc.)

Dungeon Flow - Describes the flow of the dungeon through the use of a graph. A *Dungeon Flow* can contain multiple *Dungeon Archetypes*. You can think of the *Dungeon Flow* as like a prefab of which a randomized instance will be placed in the scene.

Dungeon - The actual layout that has been generated by the *Dungeon Flow* and placed in the scene.

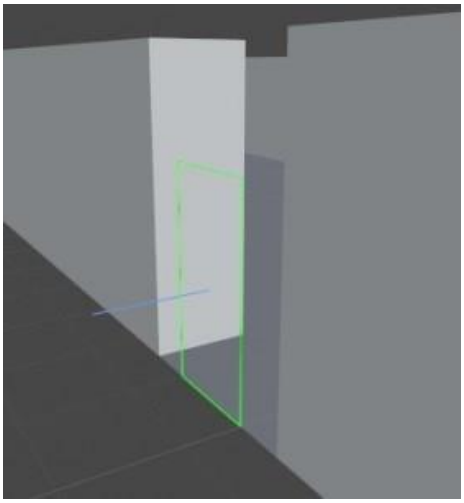
Making a Tile

First, create a simple room, leaving gaps in the walls for where you want potential doorways to be. Parent all objects in the room under a single GameObject and give it a name.



Optionally, you may want to add a *Tile* component (*Component* > *DunGen* > *Tile*) to your parent GameObject. This will allow you to modify any settings for the tile. Currently, the only setting to modify is *Allow Rotation* which is used to determine if the tile should be allowed to rotate to fit into place in the dungeon. This is useful if you have a fixed perspective camera and would like some rooms to have the camera-blocking wall removed.

Now we need to tell DunGen where we want potential doorways to be. To do this we will create an empty GameObject and call it "Doorway" and parent it to the room GameObject. Add a Doorway component (*Component* > *DunGen* > *Doorway*) to this new GameObject and position it where you want your doorway to be.

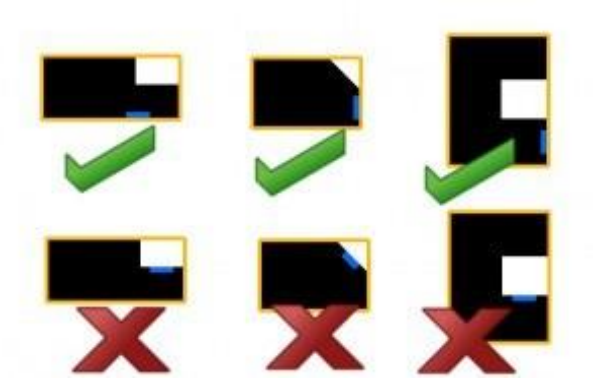


The doorway must be centred and on the very edge of the room with its local Z-axis (shown by the blue line in the editor) pointing outwards.

The green rectangle represents the extents of the doorway and currently has no use outside of visualizing where the doorways are. In later versions of DunGen, this rectangle will be used in the portal culling system. It's useful to have this properly match your doorway geometry so we're going to adjust the size now - which can be done from the inspector.

Important Note:

Due to the way overlapping tiles are handled in DunGen (using an axis-aligned bounding box around the tile to check for collisions), doorways must be placed on the edge of the bounds of the tile in order to work. The image below demonstrates correct doorway placement where the **orange** box represents the tile's bounds and the **blue** line represents the doorway position.



If all of the tiles in your dungeon are rectangular, you'll never have to worry about this limitation. But if you're using more exotic shapes, some additional thought needs to be put into it to ensure that the doorways are accessible.

Different doorway types

The doorway has a *Socket Group* field which is used to match certain doorway types together. Each doorway can only connect to another doorway with a matching *Socket Group*. This allows you to make sure that larger doorways aren't incorrectly connected to smaller ones. If all of your doorways are the same size, you can safely leave this at the default value for every doorway. If you need additional doorway types, you'll have to modify the *DoorwaySocketType* enum in "Assets/DunGen/DoorwaySocket.cs"

Adding/removing doorway-related objects

When the doorway is not in use, you will want some object to be spawned in its place to block the player. This can be a closed door mesh, a pile of debris from a collapsed ceiling, a bookcase, or anything else that can be used to give the player a feeling that the area is larger than it really is. The doorway component has two arrays of GameObjects, "*Add when in use*" and "*Add when NOT in use* which add objects in the tile when the doorway has a connection to another tile, or when it does not.

We'd like the doorway to be blocked when it's not in use (not connected to another tile) so first we add a GameObject to the scene to block it. Then we simple click the "Add New" button under the "*Add when NOT in use*" header in our Doorway component and drag our blocking object into the newly created slot. This blocking object will be in place if the doorway is not connected to another, but will be removed if it is, opening up the doorway. Conversely, we can place objects in the

"*Add when in use*" array if we want objects to only be present when the doorway is attached to another tile. For example, you might want pillars to be present on either side of a doorway only when the way is open.

We need to repeat this process for each potential doorway in the tile. Each tile should have at least two doorways, preferably more, in order to give the tool the best chance at successfully generating a dungeon. Try to keep the number of tiles with only one or two doorways to a minimum.

There's one final thing we need to do before our basic tile is ready, we need to create a prefab from it. Drag the root GameObject of the tile from the hierarchy into the project panel, be sure to give the new prefab a descriptive name.

Setting up the persistent scene

Now that we have at least one tile, we can set up the scene that we're going to generate the dungeon in, but first we need to create a few new assets.

The Tile Set

Create a new Tile Set (*Assets* > *Create* > *DunGen* > *Tile Set*) and give it a name. This asset will be used to hold all of our tiles in this simple example but later you might want to organize your tiles into multiple tile sets, more on why you would want to do that later.

Select the newly created tile set and in the inspector, click the "Add New Tile" button to add a new slot. Drag your tile prefab into appropriate field. There are also a number of fields for values related to weighting, we'll ignore those for now and come back to them later.

The Dungeon Archetype

Create a new Dungeon (*Assets* > *Create* > *DunGen* > *Dungeon Archetype*) and give it a name. This asset acts as a template which describes the rules for generating a particular style of dungeon. Select the newly created asset to see its settings in the inspector, we'll go through each of these settings in-turn.

Branch Depth: Two values representing the minimum and maximum depth of the branches. Branch tiles are tiles that do not lie on the main path. They are used to add multiple branching paths to the dungeon to make it feel more expansive.

Branch Count: Two values representing the minimum and maximum number of branches that can come off a single tile. The minimum and maximum branch count can both be set to zero to produce a dungeon with only a single route with no branching if you prefer.

Tile Sets: This is where you define which tiles this dungeon archetype uses. Click "Add New" and drag your tile set asset into the newly created slot.

The Dungeon Flow

We have one final asset to create before we can continue. Create a new Dungeon Flow (Assets > Create > DunGen > Dungeon Flow) and give it a name. This asset will define how you want the dungeon to be laid out. Select your newly created dungeon flow asset to see its settings in the inspector. The length is fairly self-explanatory, it holds the minimum and maximum possible length for the main path of the dungeon. We'll skip over the "Global Props" section for now and jump straight into the flow editor. Click the "Open Flow Editor" button.

You'll be presented with a simple graph with two nodes: start and goal, connected by a single line. In this graph, a node represents a single tile in the final dungeon, whereas a line represents a string of tiles connecting the nodes together. Right-clicking on the line gives you the option to add a new node, split the line into segments, or delete the current segment. Right-clicking a node allows you to delete it, while clicking and dragging moves the node (start and goal nodes cannot be moved or deleted).

In your own game, you may want to play around with the graph. For example, you may want to make a graph with a "boss" node just before the goal, but for our purposes, the default layout is fine.

Clicking on a node or line in the graph will give you some settings in the inspector. For nodes, you are asked for a collection of tile sets - we'll just add our only tile set to both the start and the goal nodes for now. Selecting the line will present you with the option at add dungeon archetypes - again, we'll add the only archetype we have to the line between the two points.

The Layout Generator

Now that the dungeon archetype is set up, we need to be able to generate it.

For generating at runtime: Add a new empty GameObject to the scene and attach a *Runtime Dungeon* component to it (*Components* > *DunGen* > *RuntimeDungeon*).

For generating in the editor: Open the *New Dungeon* window (*Window* > *DunGen* > *Generate Dungeon*)

Now we need to drag our dungeon flow asset into the *Dungeon Flow* slot in the inspector. There are a couple of settings here which we'll go through now.

Randomize Seed: Whether to randomize the seed the first time the dungeon is generated. If false, you will be able to set the seed manually (see below).

Seed: This value determines the dungeon that is generated. The layout generation is random, but the same seed will always produce the same layout.

Max Failed Attempts: The maximum number of times the generator will try to create a dungeon before failing with an error.

Generate on Start (runtime only): If true, a dungeon layout will be generated as soon as the scene is run in game mode. When false, you must manually call the component's *Generate()* function.

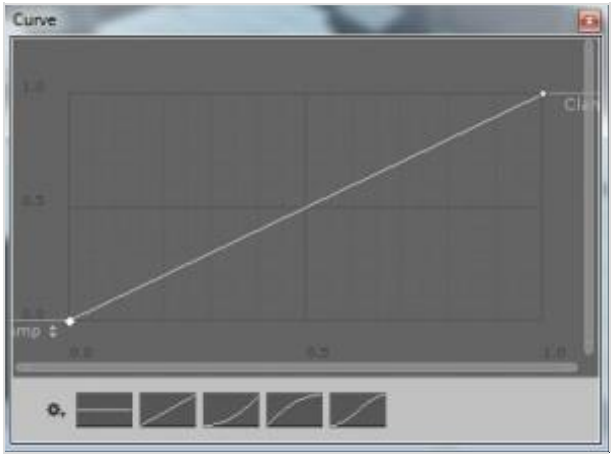
With *Generate on Start* set to true we can just hit the play button and your dungeon layout should be generated automatically.

Weights

Before going on to explain the different types of prop randomization available, we need to talk about weights. Weights are a way of controlling the chance that something will happen - in the case of DunGen, weights control the chance that tiles and props will spawn in certain places. Weights in DunGen have three components, the *Main Path Weight*, the *Branch Path Weight*, and the *Depth Scale*. The main and branch path weights allow you to specify different values based on whether the tile/object lies on the main path or a branch - giving you the option to give each object a different chance to spawn based on its location. Additionally, there is a curve called the *Depth Multiplier* - this allows you to alter the weights based on how deep into the dungeon the object is. The curve's X-axis represents the normalized depth in the dungeon (the start tile has a depth of 0, the end tile has a depth of 1) - for branch nodes, the depth starts at 0 for tiles attached to the main path and reach 1 at the maximum branching depth.

Example:

Suppose we want an object to spawn only on the main path, with an increasing chance to spawn the deeper into the dungeon we get. At the end of the dungeon the object should be twice as likely to spawn as other objects in the set.
We would set the *Branch Weight* to zero since we never want it to spawn in a branch tile. We would set the *Main Path Weight* to two and give it the following *Depth Multiplier* curve:



With this curve, the object has a linearly increasing chance to spawn as the dungeon progresses. Starting at zero in the first Tile (*Depth Multiplier* of 0 × *Weight* of 2) to a value of two in the last Tile (*Depth Multiplier* of 1 × *Weight* of 2)

Adding variation to tiles

Now we have a dungeon being generated from one or more tiles, but how do we make it feel more varied? One option is to just add more tiles, but this quickly gets to become a lot of work. Another option is to vary the props in your existing tiles. Fortunately, DunGen provides several methods of adding this kind of variation.

Local Prop Sets

A local prop set is a set of objects in a Tile of which a certain number are picked at random. Local prop sets define which props are present on a per-tile basis, so that each instance of a tile in your dungeon can appear different. Adding a local prop set to a tile can be done by attaching a *Local Prop Sets* component (*Components > DunGen > Random Props > Local Prop Sets*) to any GameObject in the tile.

Clicking the *Add New Prop* button will add a new slot to the set in which an object from the scene can be dragged.

The *Local Prop Sets* component also has a *Count* field with a minimum and maximum value. A random number of props (between the min and max values

specified) from this local set are chosen to be present in the tile, the others are discarded.

Random Prefabs

A *Random Prefab* component allows you to select any number of object prefabs and have one of them spawn in place of the GameObject this script is attached to. Clicking the *Add New Prefab* button will add an object slot for a prefab to be selected from the project window. As usual, these objects have their own weights.

Global Props

The final prop type is called a *Global Prop*. While it's nice to have the ability to spawn random objects into the world, for many situations it's vital to limit the number of a certain object that can appear in a dungeon - this is where *Global Props* come in. Attaching a *Global Prop* component to a GameObject (*Components > DunGen > Random Props > Global Prop*) allows you to control the prop on a per-dungeon basis. Attaching the component alone doesn't do anything, we need to make some changes in the dungeon flow first.

In the inspector window for the *Global Prop* component, you'll notice that, along with the weight configuration like every other prop type, there is also a *Group ID*. This *Group ID* allows us to have multiple sets of objects to limit individually.

Going back to the dungeon flow (find and select the dungeon flow asset you made earlier), you'll notice a *Global Props* section in the inspector clicking *Add New* will give you another set to work with. From here, you can choose a *Group ID* (matching the *Group ID* in the *Global Prop* component) and a *Range*. The *Range* dictates the minimum and maximum number of objects in that group that can be in the dungeon.

Example:

You have a prefab for a shrine which allows players to regain health/mana but you want no more than one to appear in the generated dungeon. You would attach a *Global Prop* component, giving it a unique *Group ID* number and place an instance of the prefab inside several different tiles. You would then add a new entry into the *Global Props* section of the dungeon flow, giving it a matching *Group ID* and setting the min and max of the *Range* to 1.

Lock & Key System

The lock & key system requires some programming in order to integrate it into your own game. This section of the tutorial will look at a simple implementation (the one used in the demo scene) but you'll likely want your own way of handling keys which is why DunGen exposes some simple interfaces to implement.

Creating the Key Manager

First, we need a KeyManager asset (*Assets > Create > DunGen > Key Manager*) to handle all of the keys we'll be making for our dungeon. With the newly created key manager asset selected, click the "Add 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 the screen when picking up keys or approaching locked doors for example.

Prefab – A prefab representation of the key in the scene. This would be used if you want the key itself to be placed as an object in the scene.

Colour – For colour-coding your keys if you want the association between keys & locks to be more apparent visually.

Once you've added all of the keys you want, we can move on to setting them into the dungeon flow. For the demo scene, we have three key types named Red, Blue and Green with the corresponding colour set.

Placing Keys into the Dungeon Flow

Select the *Dungeon Flow* asset that you want to add locks & keys to. In the inspector, there is a field called "Key Manager" - this is where you put the KeyManager asset you just created. Once that's done, you can open up the flow editor.

In DunGen, keys & locks can be added separately to either nodes or lines on the dungeon flow graph. Adding a lock to a line on the graph will also allow you to choose a range to specify the number of locked doorways that can be applied to that segment of the dungeon. Adding a lock to a node on the graph gives you the option of where the lock should be placed on that node (Entrance, Exit, or Both).

You can add possible key spawn points in the same way. When a locked door is created, DunGen will look through all tiles that come before it in the dungeon path, and pick from only the parts of the dungeon that you have marked as being a possible spawn point for a key of the same type. If no key can be found, the locked door is removed.

Tying it into your Game

Now we need a way for the keys to actually work with your game. DunGen doesn't have any idea how the keys are to be handled so you're not stuck with one way of doing things. This also means it's up to you to integrate the lock & key system into your game. Luckily, we've made it as easy as possible, all you have to do is implement a few interfaces.

IKeySpawnable – This is how DunGen knows where to spawn keys. This interface has a single function: *SpawnKey(Key key, KeyManager manager)* 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 all tiles that come before the doorway in the dungeon path that you have marked as being able to contain a "Red" key. From these tiles, it will pick a component at random that implements the IKeySpawnable interface. Some possible implementations include:

Key Spawn Point – You could implement a script that acts as a key spawn point (as we have done in the demo scene) that simply creates an instance of the key prefab at its current location when the SpawnKey function is called.

Inventory Spawn – You could have a monster inventory component for your game. You might then want to implement IKeySpawnable on that component which would add an item to that monster's inventory that can be used as a key to open the locked door.

Locks and keys are placed in the dungeon after all of the props are processed. This means that you can have chests as random props and DunGen can properly pick them as potential key spawn points.

IKeyLock – This interface has one function: *OnKeyAssigned(Key key, KeyManager manager)* 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 function is called on any component that implements IKeyLock anywhere in the locked door prefab and any of its children.

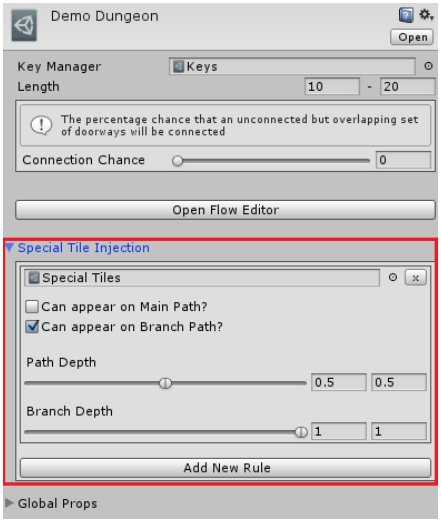
Similarly, when a key prefab is spawned, the OnKeyAssigned function is called on any component that implements IKeyLock anywhere in the key prefab and any of its children.

For the demo scene, we used this interface in a KeyColour script to change the colour of keys and locked doorways to match the key type that was assigned to them.

It is also completely up to you how you want to handle players picking up keys, checking if the player has a key, and unlocking doors if they do. The demo scene provides an example in which keys are placed in the scene as objects that can be walked into and placed into a (very) simple inventory. Locked doors will open if approached while you have the corresponding key in your inventory.

Injecting Special Tiles

In some cases, you might want specific tiles to spawn only once through the dungeon (or other more specialized logic). DunGen handles this by allowing you to "inject" tiles into the generation process; for simple cases, this can be done through the inspector for your *DungeonFlow* asset.



Press the "Add New Rule" button to add a new tile injection rule to the dungeon.

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

Path Depth is the range of depths on the main path that this tile can possibly be placed on (where 0 = start, and 1 = goal).

Branch Depth works in the same way as Path Depth, except for determining how deep into a branch the tile can spawn (where 0 is the first tile in the branch, 1 is the last). This is only used if "Can appear on Branch Path?" is true.

Tile Injection Using Code

For more advanced cases, it's more desirable to handle rules by code. To do this, simply make a new function matching the TileInjectionDelegate, and add this to the *DungeonGenerator*'s TileInjectionMethods property. The function itself should add new instances of InjectedTile to the list that's passed as an argument, like so:

```
1.  private void InjectTiles(System.Random randomStream, ref List<InjectedTile> tilesToInject)
2.  {
3.      const bool isOnMainPath = false;
4.      const float pathDepth = 0.5f;
5.      const float branchDepth = 1.0f;
6.
7.      tilesToInject.Add(new InjectedTile(TileSet, isOnMainPath, pathDepth, branchDepth));
8.  }
```

The above method will add a tile from the specified TileSet 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 not using the simple UI method, but helps to show how to inject tiles into DunGen through code.

SECTR VIS Integration

DunGen comes with built-in support for SECTR VIS portal culling. If you own this asset, you can enable DunGen to use it for portal culling easily by following these simple steps:

1. Double-click the package at "DunGen/Integration/SECTR_VIS.unitypackage" to extract it.

2. In your dungeon settings (in the RuntimeDungeon inspector if you're using that component) – under the "Portal Culling" category - select "SECTR VIS" from the drop-down menu

3. Add a "SECTR Culling Camera" to your player camera

Now, when setting up your dungeon (either with the *Runtime Dungeon* component or through the *Generate Dungeon* window), you will be presented with a new set of options. Ensure that "Enabled" is checked in order for portal culling to work. The remaining settings should look familiar to anyone who has used SECTR before, more information on these settings can be found in the SECTR documentation.

That's it! DunGen will now automatically generate sectors and portals for your runtime or in-editor dungeons for use with SECTR Vis portal culling.

Doors placed by DunGen (including through the Lock & Key system) will have a *Door* component attached which can be used to control 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 Integration

If you also own PlayMaker, you can enable integration by double-clicking the package at "Dungen/Integration/PlayMaker" to extract it. Integration should just work from there.

Action Nodes

There are currently three PlayMaker actions implemented within DunGen.

Generate – Generates a new dungeon layout using settings from an existing *RuntimeDungeon* component in the scene.

Generate with Settings – Generates a new dungeon layout using settings that you can specify through the PlayMaker UI. It's not necessary to place a *RuntimeDungeon* component in the scene first as one will be created for you.

Clear – Removed a dungeon layout created by an existing *RuntimeDungeon* component in the scene

Navigation Mesh Generation

DunGen doesn't provide runtime NavMesh generation out of the box, but it does come with integration with both RAIN AI and A* Pathfinding Project Pro.

RAIN Integration

Setup

1. Ensure both DunGen and RAIN are imported into your project
2. Double-click the package at "DunGen/Integration/RAIN.unitypackage" and select "import" when prompted.
3. Make sure you have a RAIN NavMeshRig component somewhere in your scene. This can be on any GameObject as long as it's not the same one that holds DunGen's RuntimeDungeon component. It's best if this is on an otherwise empty GameObject.
4. Add a "DunGen/NavMesh/RAIN NavMesh Generator" component to the same GameObject that contains the RuntimeDungeon.
5. That's it! The NavMesh will be generated when the dungeon is complete. 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 drawn while this GameObject is selected).

Handling Doors

DunGen's RAIN integration doesn't yet handle opening/closing doors.

A* Pathfinding Project Pro Integration

Setup

1. Ensure both DunGen and A* Pathfinding Project Pro are imported into your project
2. Double-click the package at "DunGen/Integration/AStarPathfindingProjectPro.unitypackage" and select "import" when prompted.
3. Make sure you have an Astar Path component (Pathfinding/Pathfinder) somewhere in your scene. Add a "Recast Graph" to the component.
4. Add a "DunGen/NavMesh/A* Pathfinding NavMesh Generator" component to the same GameObject that contains the RuntimeDungeon.
5. That's it! The NavMesh will be generated when the dungeon 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 "DunGen/NavMesh/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"

Limitations

Aside from the limitation regarding doorway placement discussed earlier, there is one more problem to be aware of. As each tile in the dungeon must be a prefab, and Unity doesn't handle nested prefabs properly, placing prefab instances inside a tile will not work properly (the instance is placed correctly, but updating the prefab itself does not updated the instance if placed inside another prefab). This is a problem that Unity have said they will be fixing but it's not clear when this will happen. For now, there are several work-arounds out there on the Asset Store. There's also the "poor mans nested prefabs" script by Nicholas Francis which you can find on his website.

Analysing Your Dungeon

Once you've created your dungeon, you may want to test it to see how it performs in terms of success rate, generation times and other measurable statistics. DunGen provides an option to do this automatically. In place of the *Runtime Dungeon* component you would use to generate a dungeon for your game, you can use a *Runtime Analyser* component (Components > DunGen > Analysis > Runtime Analyser) to gain information about your dungeon.

The *Runtime Analyser* component has most of the properties of the dungeon generator with a couple of additions:

Iterations: How many dungeons it should generate to gain statistics. Obviously, we can't generate all possible layouts; that would take too long. Instead, we can set the iteration count to give us a reasonable sample size (around 1000 – 10000 should suffice). The greater the number of iterations, the longer the analysis will take.

Maximum Analysis Time: The time (in seconds) that the analysis is allowed to run for. If it hasn't reached its target iteration count by this time, the analysis will be stopped early. In most cases, you'll want to leave this set to zero (meaning there is no time limit) as the analysis can be stopped at any time by leaving play mode.

Per-Frame Analysis Time: How long (in seconds) the analysis can spend on each frame. DunGen will attempt to generate as many layouts as possible in a single frame but will split the generation over multiple frames to keep Unity responsive. The default value should be fine in most cases.

When the analysis is complete, you'll have access to a large amount of information about the generation of your dungeon. The most important of which will be the percentage of dungeons that were successfully generated – ideally, you'll want this as close to 100% as possible. If you have a high percentage of failed dungeons, this might indicate that your rooms could use more (or more easily accessible) doorways.

The analysis also gives information on the time taken to generate the layouts (split up into each individual stage), the number of rooms that were generated and the number of retries (also an important statistic to be aware of as a retry count above zero means that a dungeon failed to generate the first time and had to be restarted.

Each of these statistics is broken down in to a minimum, maximum and average (mean) as well as a standard deviation (how much the value varies) to give you as much information as possible to analyse your dungeon and make changes where necessary.