



GAME DEVELOPMENT > PROGRAMMING

Bake Your Own 3D Dungeons With Procedural Recipes

by [Marcin Seredynski](#) 8 Feb 2014Difficulty: Intermediate Length: Medium Languages: English

Programming



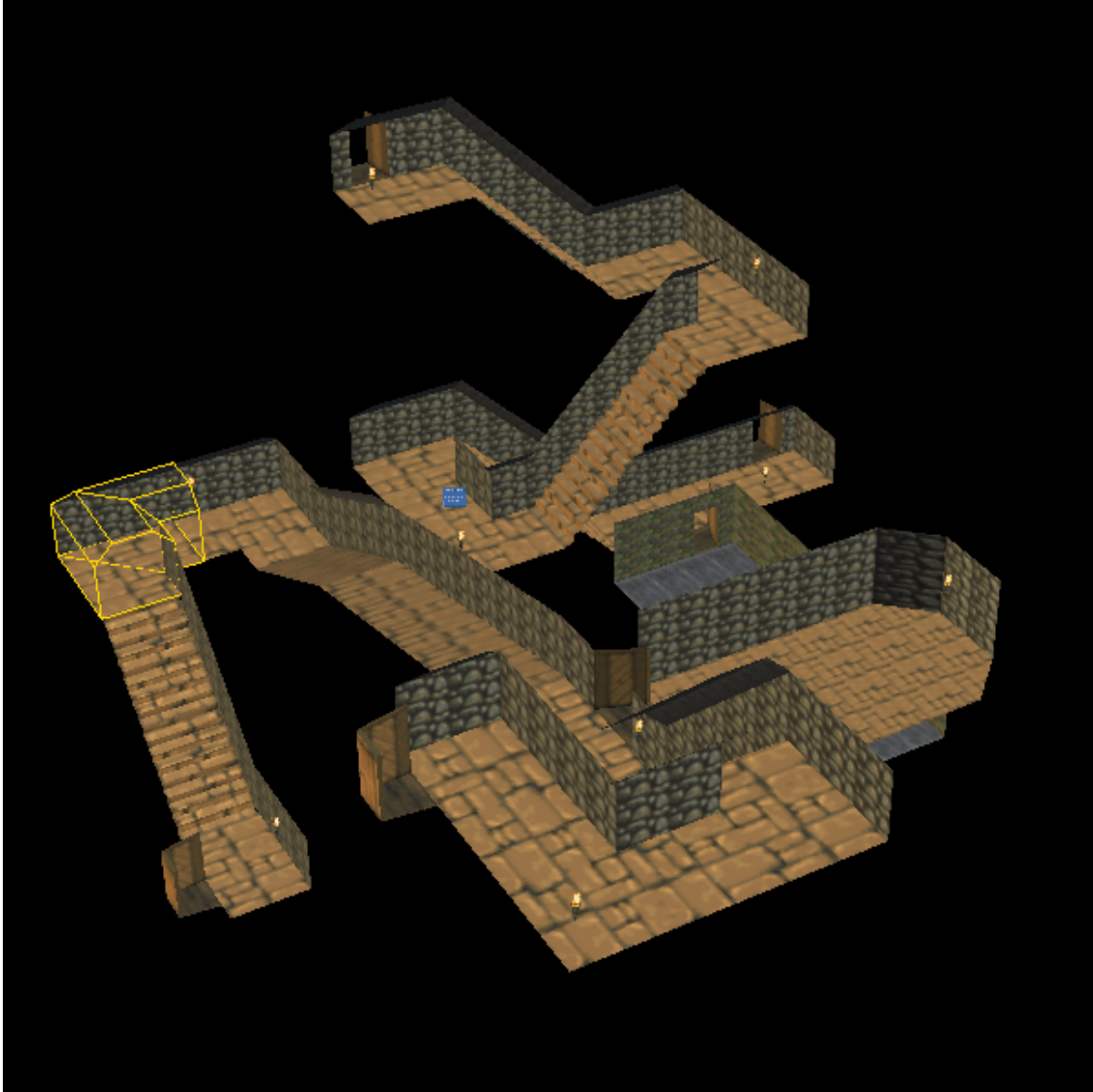
In this tutorial, you will learn how to build complex dungeons from prefabricated parts, unconstrained to 2D or 3D grids. Your players will never run out of dungeons to explore, your artists will appreciate the creative freedom, and your game will have better replayability.

To benefit from this tutorial, you need to understand basic 3D transformations and feel comfortable with [scene graphs](#) and [entity-component systems](#).

A Bit of History

One of the earliest games to use procedural world generation was [Rogue](#). Made in 1980, it featured dynamically generated, 2D, grid-based dungeons. Thanks to that no two playthroughs were identical, and the game has spawned a whole new genre of games, called "roguelikes". This type of dungeon is still quite common over 30 years later.

In 1996, [Daggerfall](#) was released. It featured procedural 3D dungeons and cities, which allowed the developers to create thousands of unique locations, without having to manually build them all. Even though its 3D approach offers many advantages over the classic 2D grid dungeons, it isn't very common.



This image shows a part of a larger dungeon, extracted to illustrate what modules have been used to build it. The image was generated with "Daggerfall Modelling", downloaded from dfworkshop.net.

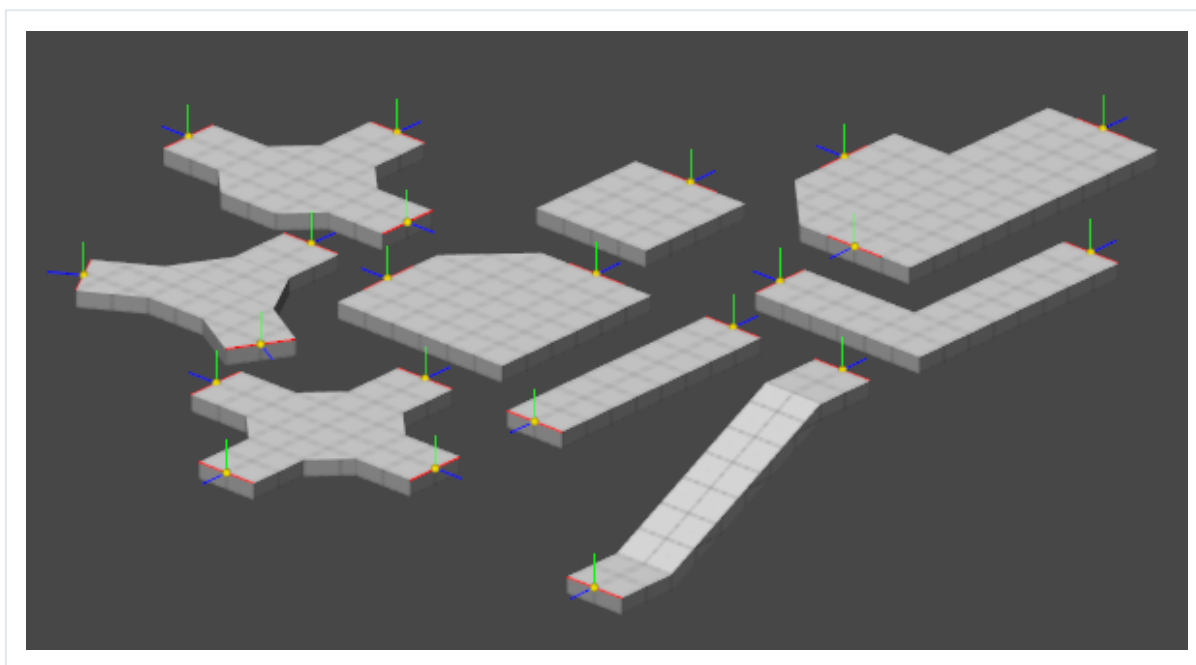
We will focus on generating dungeons similar to Daggerfall's.

How to Build a Dungeon?

In order to build a dungeon, we need to define what a dungeon is. In this tutorial, we will define a dungeon as a set of modules (3D models) connected to each other according to a set of rules. We will use *rooms* connected by *corridors* and *junctions*:

- A *room* is a large area that has one or more exits
- A *corridor* is a narrow and long area that may be sloped, and has exactly two exits
- A *junction* is a small area that has three or more exits

In this tutorial, we will use simple models for the modules—their meshes will only contain floor. We will use three of each: rooms, corridors and junctions. We will visualize exit markers as axis objects, with the -X/+X axis being red, +Y axis green, and +Z axis blue.



Modules used to build a dungeon

Notice that the orientation of exits isn't constrained to 90 degree increments.

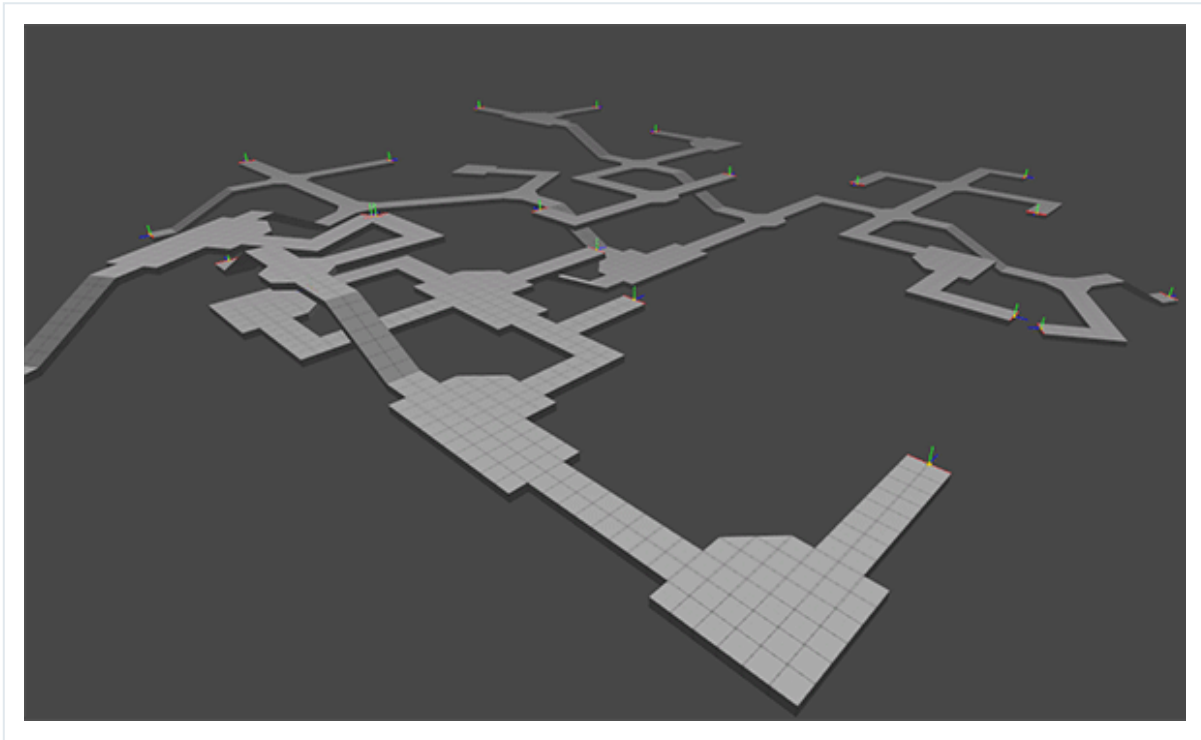
When it comes to connecting the modules, we will define the following rules:

- Rooms can connect to corridors
- Corridors can connect to rooms or junctions
- Junctions can connect to corridors

Each module contains a set of *exits*—marker objects with a known position and rotation. Each module is tagged to say what kind it is, and each exit has a list of tags it can connect to.

At the highest level, the process of building the dungeon is as follows:

1. Instantiate a starting module (preferably one with a larger number of exits).
2. Instantiate and connect valid modules to each of the unconnected exits of the module.
3. Rebuild a list of unconnected exits in the whole dungeon so far.
4. Repeat the process until a large enough dungeon is built.



A look at the iterations of the algorithm at work.

The detailed process of connecting two modules together is:

1. Pick an unconnected exit from the old module.
2. Pick a prefab of a new module with tag matching tags allowed by the old module's exit.
3. Instantiate the new module.
4. Pick an exit from the new module.

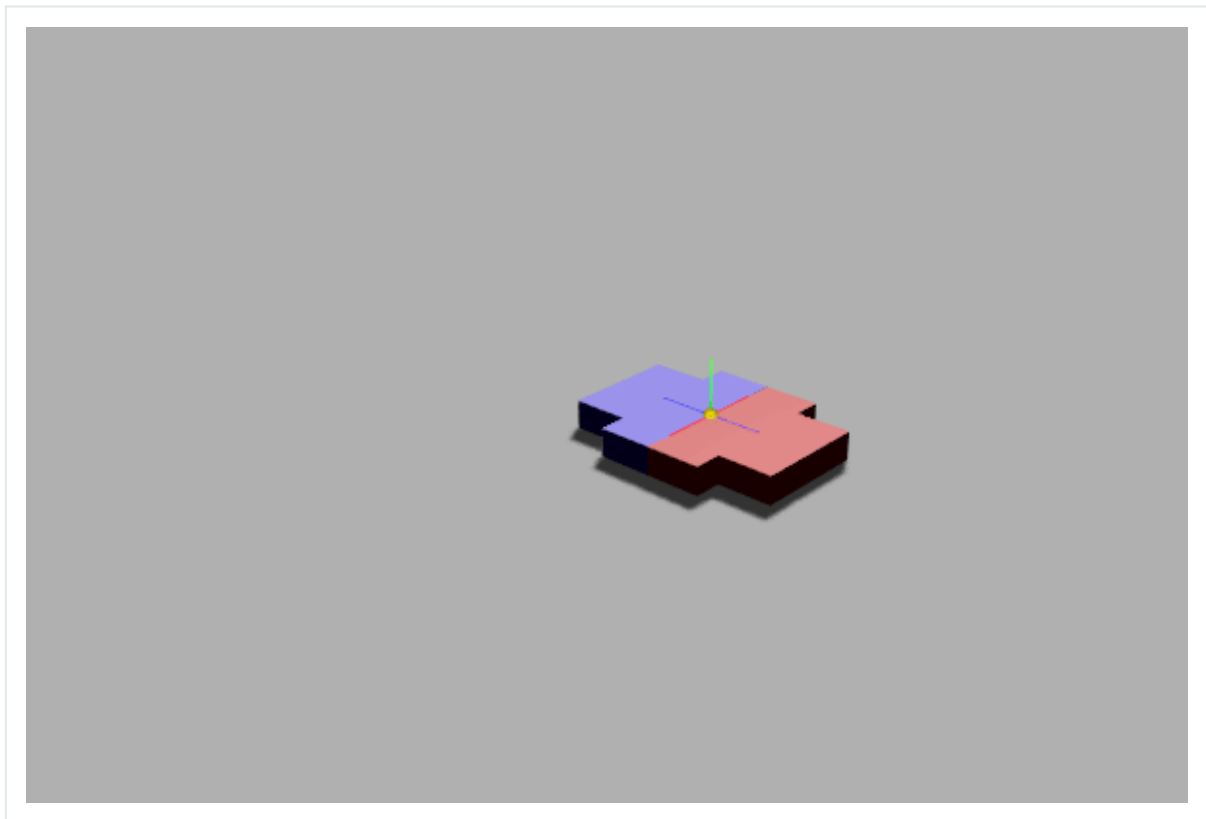
5. Connect the modules: match the new module's exit to the old one's.
6. Mark both exits as connected, or simply delete them from the scene graph.
7. Repeat for the rest of the old module's unconnected exits.

To connect two modules together, we have to align them (rotate and translate them in 3D space), so that an exit from the first module matches an exit from the second module.

Exits are *matching* when their position is the same and their +Z axes are opposite, while their +Y axes are matching.

The algorithm to do this is simple:

1. Rotate the new module on the +Y axis with the rotation origin at the new exit's position, so that the old exit's +Z axis is opposite to the new exit's +Z axis, and their +Y axes are the same.
2. Translate the new module so that the new exit's position is the same as the old exit's position.



Connecting two modules.

Implementation

The pseudo-code is Python-ish, but it should be readable by anyone. [The sample source code](#) is a Unity project.

Let's assume we're working with an entity-component system that holds entities in a scene graph, defining their parent-child relationship. A good example of a game engine with such a system is Unity, with its game objects and components. Modules and exits are entities; exits are children of modules. Modules have a component that defines their tag, and exits have a component that defines the tags they're allowed to connect to.

We'll deal with the dungeon generation algorithm first. The end constraint we will use is a number of iterations of dungeon generation steps.

```

01 def generate_dungeon(starting_module_prefab, module_prefabs, iterations):
02     starting_module = instantiate(starting_module_prefab)
03     pending_exits = list(starting_module.get_exits())
04
05     while iterations > 0:
06         new_exits = []
07         for pending_exit in pending_exits:
08             tag = random.choice(pending_exit.tags)
09             new_module_prefab = get_random_with_tag(module_prefabs, tag)
10             new_module_instance = instantiate(new_module_prefab)
11             exit_to_match = random.choice(new_module_instance.exits)
12             match_exits(pending_exit, exit_to_match)
13             for new_exit in new_module_instance.get_exits():
14                 if new_exit != exit_to_match:
15                     new_exits.append(new_exit)
16         pending_exits = new_exits
17         iterations -= 1

```

The `instantiate()` function creates an instance of a module prefab: it creates a copy of the module, together with its exits, and places them in the scene. The `get_random_with_tag()` function iterates through all the module prefabs, and picks one at random, tagged with the provided tag. The `random.choice()` function gets a random element from a list or an array passed as a parameter.

The `match_exits` function is where all the magic happens, and is shown in detail below:

```

01 def match_exits(old_exit, new_exit):
02     new_module = new_exit.parent
03     forward_vector_to_match = old_exit.backward_vector
04     corrective_rotation = azimuth(forward_vector_to_match) - azimuth(new_exit.fc
05     rotate_around_y(new_module, new_exit.position, corrective_rotation)
06     corrective_translation = old_exit.position - new_exit.position
07     translate_global(new_module, corrective_translation)
08
09 def azimuth(vector):
10     # Returns the signed angle this vector is rotated relative to global +Z axis
11     forward = [0, 0, 1]
12     return vector_angle(forward, vector) * math.copysign(vector.x)

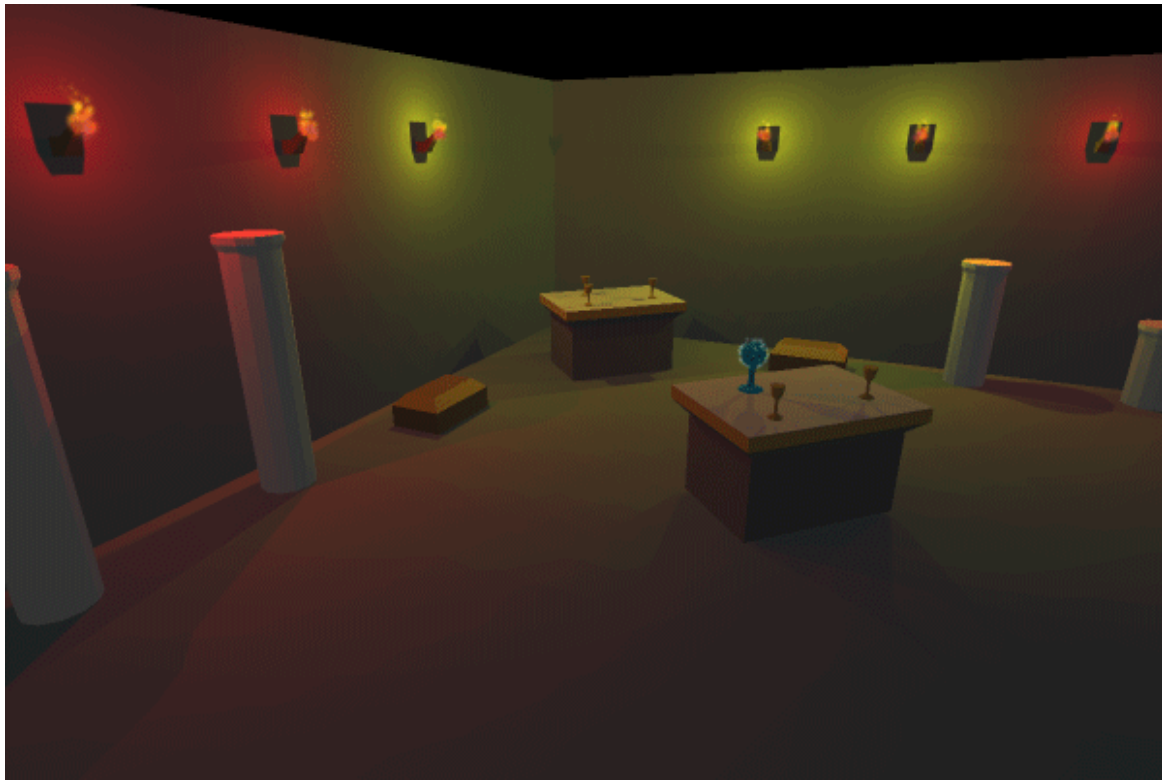
```

The `backward_vector` property of an exit is its -Z vector. The `rotate_around_y()` function rotates the object around a +Y axis with its pivot at a provided point, by a specified angle. The `translate_global()` function translates the object with its children in the global (scene) space, regardless of any child relationship it may be a part of. The `vector_angle()` function returns an angle between two arbitrary vectors, and finally, the `math.copysign()` function copies the sign of a provided number: `-1` for a negative number, `0` for zero, and `+1` for a positive number.

Extending the Generator

The algorithm can be applied to other types of world generation, not just dungeons. We can extend the definition of a module to cover not only dungeon parts such as rooms, corridors and junctions, but also furniture, treasure chests, room decorations, etc. By placing the exit markers in the middle of a room, or on a room's wall, and tagging it as a `loot`, `decoration`, or even `monster`, we can bring the dungeon to life, with objects that you can steal, admire or kill.

There's only one change that needs to be done, so that the algorithm works properly: one of the markers present in a placeable item has to be marked as `default`, so that it always gets picked as the one that will be aligned to existing scene.



In the above image, one room, two chests, three pillars, one altar, two lights, and two items have been created and tagged. A room holds a set of markers that reference other models' tags, such as `chest`, `pillar`, `altar`, or `wallLight`. An altar has three `item` markers on it. By applying the dungeon generation technique to a single room, we can create numerous variations of it.

The same algorithm may be used to create procedural items. If you would like to create a [sword](#), you could define its grip as a starting module. The grip would connect to pommel and to the cross-guard. The cross-guard would connect to the blade. By having just three versions of each of the sword parts, you could generate 81 unique swords.

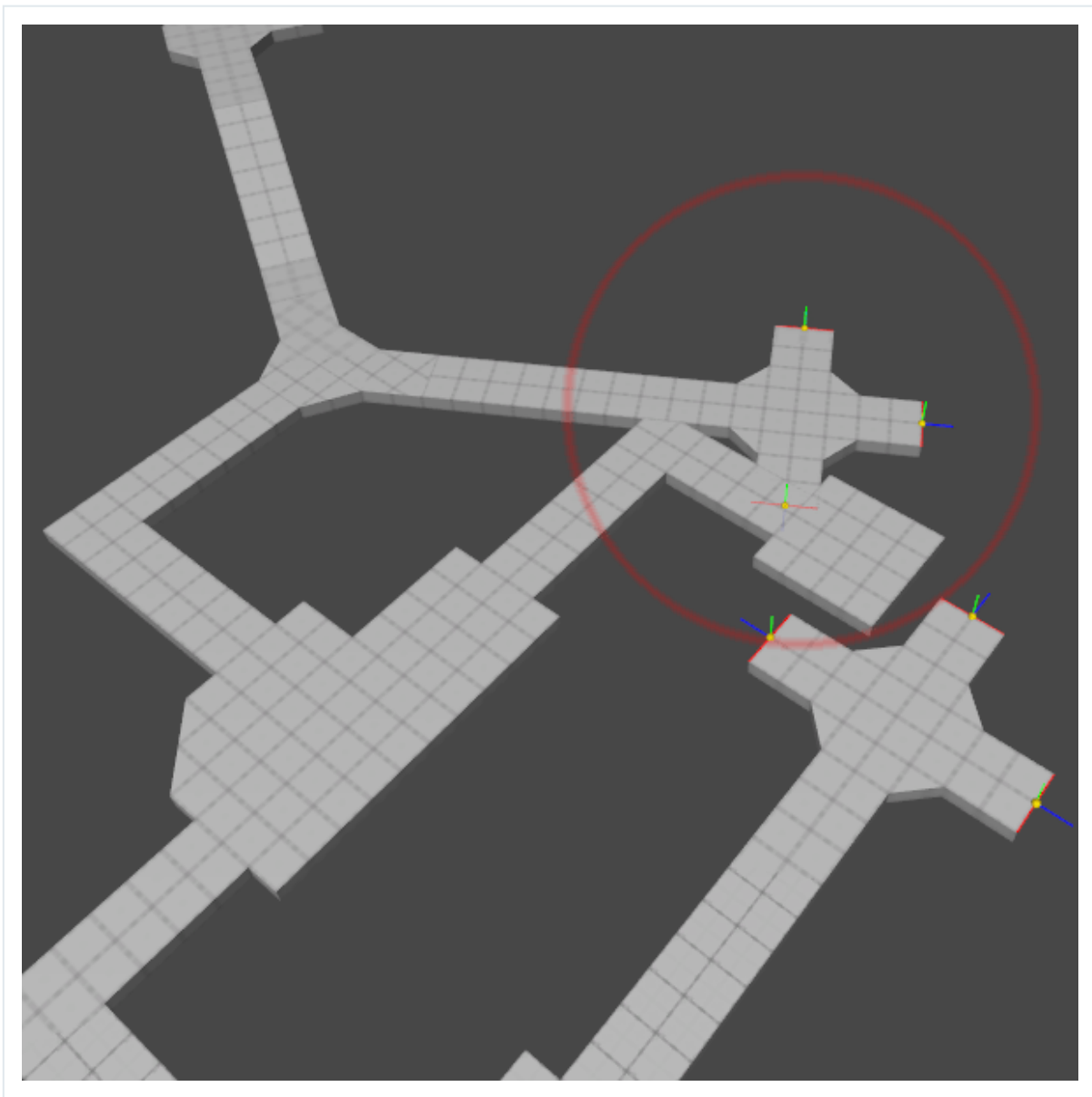
Caveats

You have probably noticed some problems with the way this algorithm works.

The first problem is that the simplest version of it builds dungeons as a tree of modules, with its root being the starting module. If you follow any branch of the structure of the dungeon, you are guaranteed to hit a dead-end. There tree's branches aren't

interconnected, and the dungeon will lack loops of rooms or corridors. One way to address this would be setting aside some of the module's exits for later processing, and not connecting new modules to these exits. Once the generator went through enough iterations, it would pick a pair of exits at random, and would try to connect them with a set of corridors. There's a bit of algorithmic work that would need to be done, in order to find a set of modules and a way to interconnect them in a way that would create a passable path between these exits. This problem in itself is complex enough to deserve a separate article.

Another problem is that the algorithm is unaware of spatial features of modules it places; it only knows the tagged exits, and their orientations and locations. This causes the modules to overlap. An addition of a simple collision check between a new module to be placed around existing modules would allow the algorithm to build dungeons that don't suffer from this problem. When the modules are colliding, it could discard the module it tried to place and would try a different one instead.



The simplest implementation of the algorithm with no collision checks causes modules to overlap.

Managing the exits and their tags is another problem. The algorithm suggests defining tags on each exit instance, and tagging all the rooms—but this is quite a lot of maintenance work, if there's a different way of connecting the modules you would like to try. For example, if you would like to allow rooms to connect to corridors and junctions instead of just corridors, you would have to go through all the exits in all the room modules, and update their tags. A way around this is to define the connectivity rules on three separate levels: dungeon, module and exit. Dungeon level would define rules for the entire dungeon—it would define which tags are allowed to interconnect. Some of the rooms would be able to override the connectivity rules, when they're processed. You could have a "boss" room that would guarantee that there's always a "treasure" room behind it.

Certain exits would override the two previous levels. Defining tags per exit allows the greatest flexibility, but sometimes too much flexibility isn't that good.

Floating point math isn't perfect, and this algorithm relies heavily on it. All the rotation transformations, arbitrary exit orientations, and positions will add up, and may cause artifacts as seams or overlaps where exits connect, especially further from the world's center. If this would be too noticeable, you could extend the algorithm to place an extra prop where the modules meet, such as a door frame or a threshold. Your friendly artist will surely find a way to hide the imperfections. For dungeons of a reasonable size (smaller than 10,000 units across), this problem is not even noticeable, assuming enough care was taken when placing and rotating the exit markers of the modules.

Conclusion

The algorithm, despite some of its shortcomings, offers a different way to look at the dungeon generation. You will be no longer constrained to 90-degree turns and rectangular rooms. Your artists will appreciate the creative freedom this approach will offer, and your players will enjoy the more natural feel of the dungeons.

Advertisement



Marcin Seredynski

Game Developer nightly, .NET Developer daily. Fan of voxels. Rants as @vigrd on Twitter. Struggles to build his own website at gamination.com

 FEED  LIKE  FOLLOW  FOLLOW

Weekly email summary

Subscribe below and we'll send you a weekly email summary of all new Game Development tutorials. Never miss out on learning about the next big thing.

Update me weekly

Advertisement

Download Attachment

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by  native

30 Comments

Gamedevtuts+

 Login ▾

 Recommend 2  Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Junian • 4 years ago

Thank you for your awesome tutorial, sir. This one is deserves to be shared to the world.

3 ^ | v • Reply • Share ›



Andrew Kabakwu • 4 years ago

Thank you for a great tutorial.

I downloaded the unity sample source code and tried it out. Wonderful.

I have a few questions -

Where are the mesh assets used in the modules?

And where are the modules created?

If they are somehow built into the unity scene file, how did you do that?

1 ^ | v • Reply • Share ›



Chase Collins • 4 years ago

Thanks a lot Marcin, You should write more, This was a superb tutorial

1 ^ | v • Reply • Share ›



Fuminori Kitayama • 10 months ago

This article helped me so much, I know this is an old post but I can't help to appreciate you for writing such a concise codes. so thank you very much.

^ | v • Reply • Share ›



STAR_WARS_NERD • a year ago

Has anyone worked out to do the 'basic' Collision check? I keep having the problem that the meshes are barely touching when they are generated, and keeps detecting a collision. A simple algorithm would be awesome

^ | v • Reply • Share ›



everlasting17 • a year ago

Thx for tutorial ! It helps me to wrote a generator for my game .

^ | v • Reply • Share ›



Lea Maria • a year ago

Thank you for this awesome tutorial! I'm new to procedural generation (and no programmer^^) and i wonder if i have to create this exit markers as empty group or locator objects in my 3D Software or better create them in Unity if this is possible? Also i would like to know if there is anything else an artist should know about creating assets for this kind of objects. Besides of making the exit objects and textures seamless :)

^ | v • Reply • Share ›



xigolle • 2 years ago

I know this is a very very old post but I kinda need this kind of room generating in my current project :)

But I got a few questions hoping that there is still someone who can help me out :)

My 2 questions are how can I make this project to work without the corridor's so only create rooms randomly attached to each other?

Also is it possible to set walls everywhere except by the exit ?

Hope somebody can help me I am really new to Unity

^ | v • Reply • Share ›



Marcin Seredynski ➔ xigolle • 2 years ago

Simply don't feed the corridor prefab to the generator's asset pool. Prepare your room assets in such a way that they can be connected together door-to-door.

As much as I know procedural generation is exciting, I think that if you're really new to Unity, you'll be probably better off starting with building your scenes manually and learning the core concepts of the engine first.

Here's a few links to get you started:

* <https://unity3d.com/learn/t...>

* <http://gamedevelopment.tuts...>

* <http://gamedevelopment.tuts...>

^ | v • Reply • Share ›



xigolle → Marcin Seredynski • 2 years ago

Thanks a lot for the reply :)

I thought I made sure I didn't feed it to the generator. I examined your code and I think I understands how it works but my expectations doesn't sum it what it returns. I removed the corridor modules from his parent as also made the amount of items smaller and only put the elements in it which it should need to generate the data. Will try to make some good assets so that it connects.

The problem I only am using Unity for a visual demo of an other application. But will review the links you gave me in the hope I get it working :)

Still strange why the script just doesn't leave the corridor out of the generator.

^ | v • Reply • Share ›



Thomas • 2 years ago

What is the license on the scripts in the Unity example?

^ | v • Reply • Share ›



James • 2 years ago

This tutorial is just what I was looking for. I've played with it an expanded upon it. However, I'm having trouble making rooms loop back (instead of sprawling unendingly across the scene). Can you provide another tutorial, or some insight as to how to approach this?

^ | v • Reply • Share ›



Dipankar Jana • 2 years ago

Why python???

^ | v • Reply • Share ›



Michael James Williams Mod → Dipankar Jana • 2 years ago

It makes for good pseudo-code because it's fairly easy to read, regardless of what other language you might know. That makes the tutorial more platform-agnostic. There's C# Unity code in the demo files though! <https://github.com/tutsplus...>

^ | v • Reply • Share ›



TK • 3 years ago

I was wondering the same thing as Andrew Kabakwu. I was hoping to play with this a bit more, but I have no idea where the meshes are for the modules, or how to create them (googling didnt seem to help). Would really appreciate it if you can point me to some resources! Apart from that, good work!

^ | v • Reply • Share ›



Michael James Williams Mod → TK • 3 years ago

I will ping the instructor and ask :)

^ | v • Reply • Share ›



Marcin Seredynski → Michael James Williams • 3 years ago

I am so happy that this post is still being read :)

In hindsight, I should have approached meshes differently. I used ProBuilder asset to create them, and then scrapped ProBuilder scripts, leaving the meshes in the scenes. I had to do this to ensure I don't redistribute ProBuilder as it requires a license.

However, if you need to access the meshes, you can do so by going to the example scenes, checking the "dungeon modules" and "room modules" inactive objects. Underneath them you will see a number of modules, and below them - connectors. You need to have a look at the modules' "Mesh Filter" component.

If you want to play with creating custom modules, feel free to use any editor and format and import it into the Scene. The modules need to have a "Module" component attached, and ModuleConnector children objects below them in the hierarchy.

I hope this helps!

^ | v • Reply • Share ›



Rikard Carlsson • 3 years ago

great tutorial. thank you! I just have to ask tho. i am looking at your sample script and cant figure out what "TItem" is in "ModularWorldGenerator.cs". How can i make a random generator without the TItem?

^ | v • Reply • Share ›



Michael James Williams Mod → Rikard Carlsson • 3 years ago

I believe using TItem in this sense is just a way of telling C# what *type* of item to return; in particular, by specifying that the parameter of the function is TItem[] and the return type is TItem, it means that you pass it an array of any type of object, and it will return exactly one object of that type. So, pass it an array of strings and it'll return a string; pass it an array of ints and it'll return an int, and so on.

The GetRandom() function, then, will return a random element from whatever array you pass to it, regardless of what type of object is in the array.

Hope that helps!

^ | v • Reply • Share ›



Marcin Seredynski → Michael James Williams • 3 years ago

Michael, thank you for clarifying this to Rikard, this is exactly what it does.

Anyone interested in learning more about C# Generics will find excellent documentation here: <https://msdn.microsoft.com/...>

^ | v • Reply • Share ›



Harrison Travers → Marcin Seredynski • 2 years ago

Hey Marcin, just a quick question, is it possible to generate a dungeon with items onto with all the on the single script. Because I'm have trouble with just that. I have my Room Modules which have their moduleconnecters with the tags "Room" However, I have moduleconnecters with tags like "Entity" or "Enemy" but for what ever reason, if random generation favours the annoyance of a "room" generating on the "Entity" tag... It will. Any clarification on why this is happening, not sure if it's functioning as intended or not.

PS: Problem has occurred with default both checked and unchecked.

2 ^ | v • Reply • Share ›



Oliver Müller • 3 years ago

Thank you!

^ | v • Reply • Share ›



Paul M • 3 years ago

This was a really helpful article. I played around with the unity files but I'm having trouble grasping how some of the rotations / matching the exits is done in C#. I'm a beginner coder so I know that is part of the problem. Is it possible to get the scripts with commented lines to help with tearing it apart to understand?

^ | v • Reply • Share ›



Michael James Williams Mod → Paul M • 3 years ago

I'll see what we can do :) Is there anything in particular you'd like more detailed comments on?

^ | v • Reply • Share ›



Sorrien • 3 years ago

Is there any way you could post the models you used to test this with to make it easier to get started with the actual coding?

^ | v • Reply • Share ›



Sorrien • 3 years ago

It could be useful to try to recreate this in a blueprint in UE4: <https://www.unrealengine.co...>

^ | v • Reply • Share ›



Juan • 3 years ago

Thanks for this.

^ | v • Reply • Share ›



Nicholas • 4 years ago

I've been researching procedural generation and seeing if I could translate its methods into a 3d space, and this tutorial really does wonders for me. I think I might try your sample project and expand on it for my game. Thank you so much.

^ | v • Reply • Share ›



Andrew Kabakwu • 4 years ago

Thank you, you should write more tutorials :)

^ | v • Reply • Share ›






Marcin Seredynski • 4 years ago

Hello Andrew,

Apologies for a late reply. I have created the meshes in ProBuilder, and stripped the game objects of the ProBuilder scripts. These game objects / modules are built in the Unity scene, all marked as active, but with an inactive parent. The approach would work as well for prefabs, which you could reference from the project hierarchy.

I hope this helps. If it doesn't - ask more :)

^ | v • Reply • Share ›

 [Subscribe](#)  [Add Disqus to your site](#)[Add Disqus](#)  [Privacy](#)

Advertisement

[ENVATO TUTORIALS+](#)

[JOIN OUR COMMUNITY](#)

[HELP](#)[tuts+](#)

25,123

Tutorials

1,088

Courses

18,640

Translations

[Envato.com](#) [Our products](#) [Careers](#)

© 2017 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.

[Follow Envato Tuts+](#)

