IMPLEMENTATION

Bake Your Own 3D Dungeons With Procedural Recipes

by Marcin Seredynski 8 Feb 2014 20 Comments

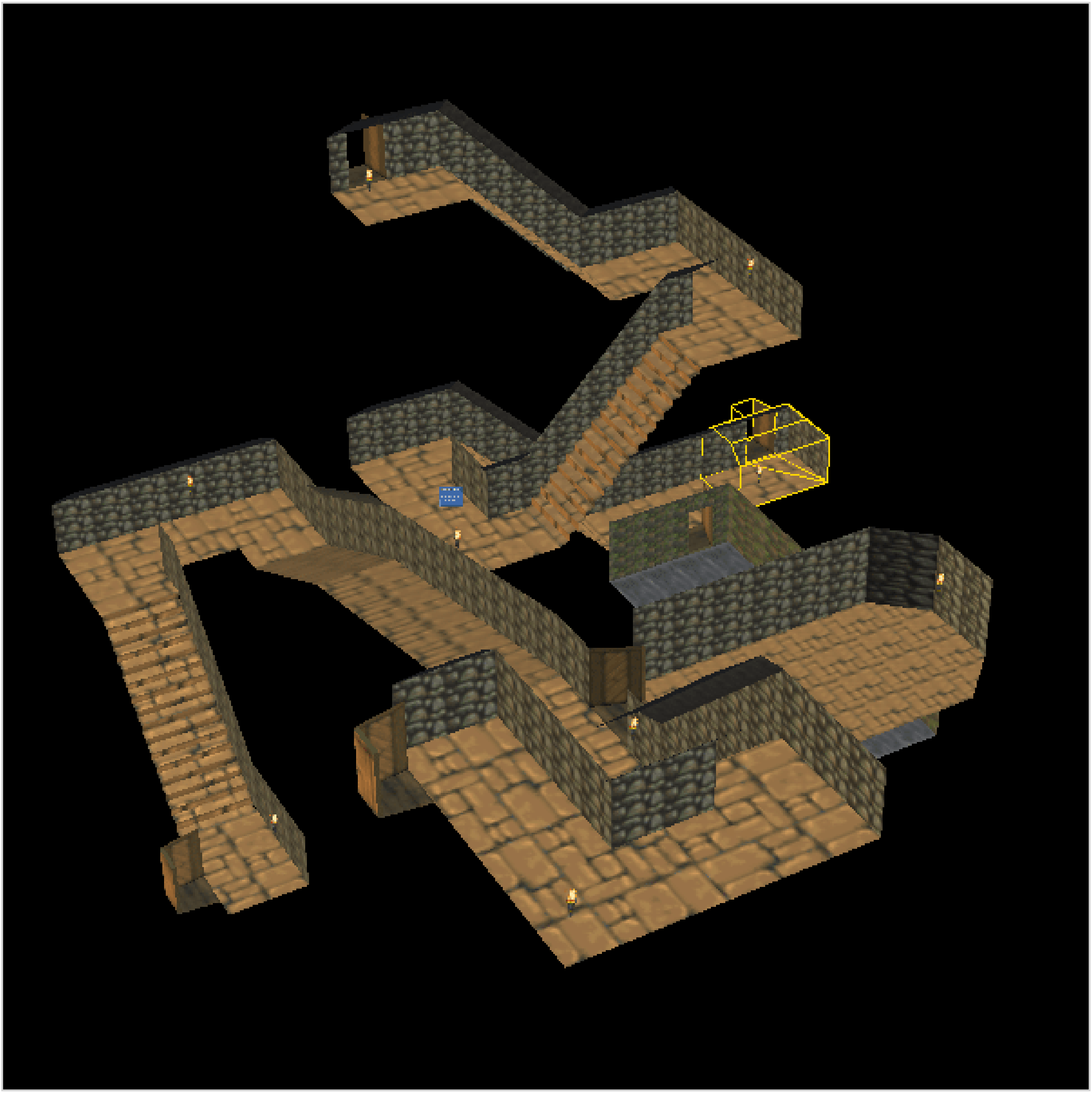
In this tutorial, you will learn how to build complex dungeons from prefabricated parts, unconstrained to 2D or 3D grids. Your players will never run out of dungeons to explore, your artists will appreciate the creative freedom, and your game will have better replayability.

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.

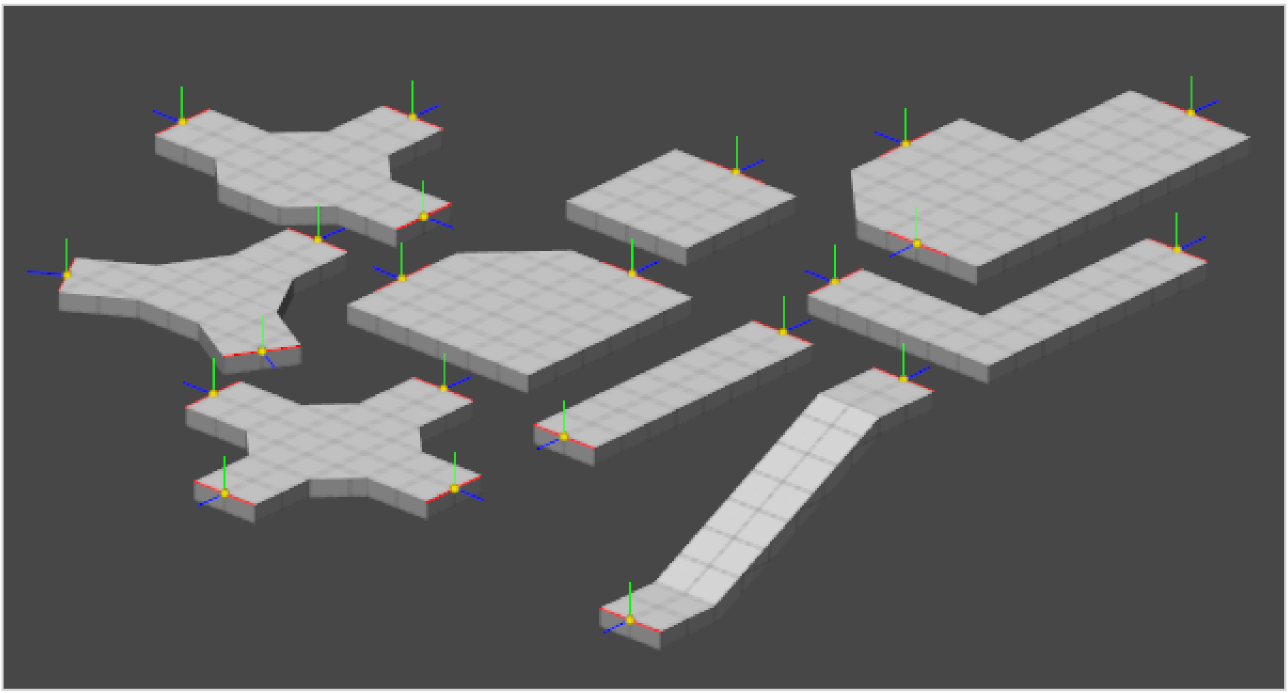
We will focus on generating dungeons similar to Daggerfall's.

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.

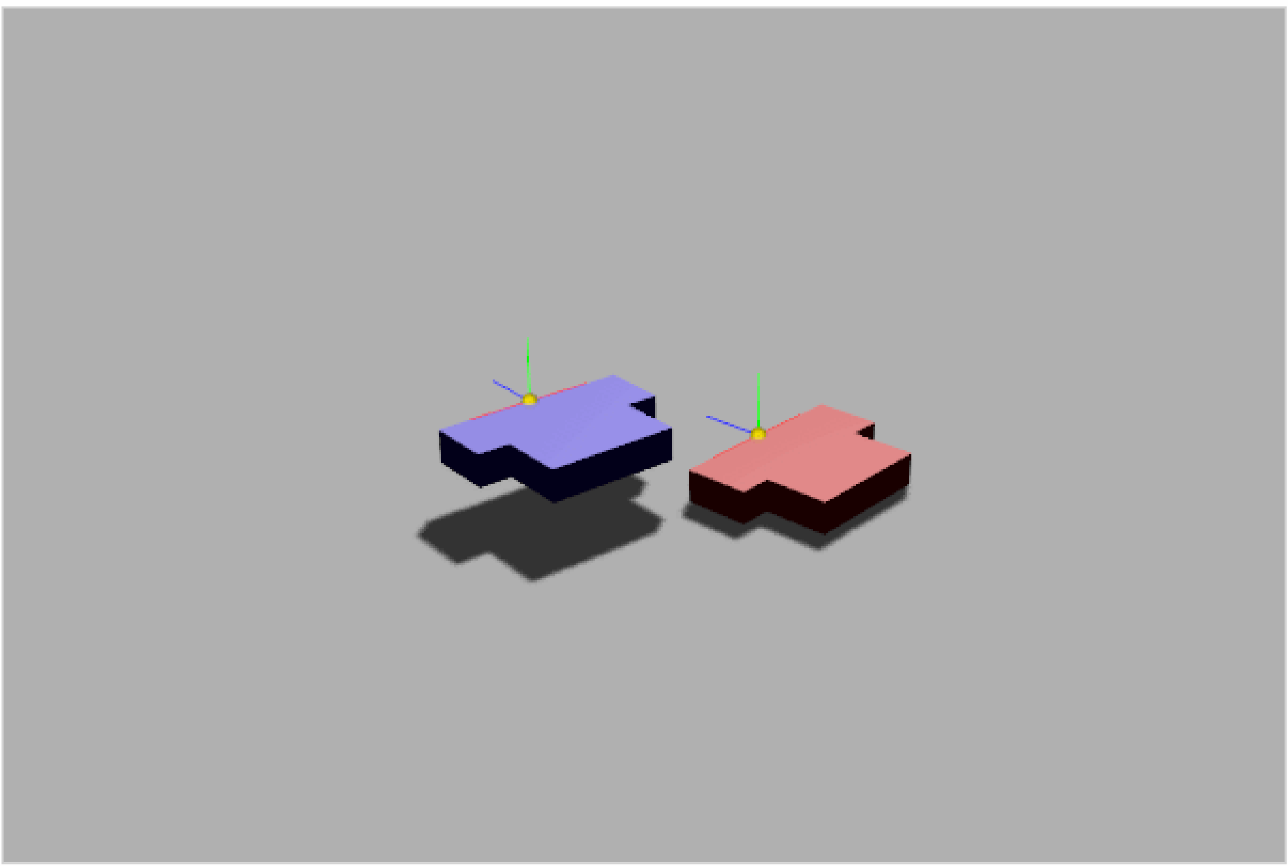
The detailed process of connecting two modules together is:

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.

```
def generate_dungeon(starting_module_prefab, module_prefabs, iter
    starting_module = instantiate(starting_module_prefab)
    pending_exits = list(starting_module.get_exits())

    while iterations > 0:
        new_exits = []
        for pending_exit in pending_exits:
            tag = random.choice(pending_exit.tags)
            new_module_prefab = get_random_with_tag(module_prefab
            new_module_instance = instantiate(new_module_prefab)
            exit_to_match = random.choice(new_module_instance.exi
            match_exits(pending_exit, exit_to_match)
            for new_exit in new_module_instance.get_exits():
                if new_exit != exit_to_match:
                    new_exits.append(new_exit)
        pending_exits = new_exits
        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:

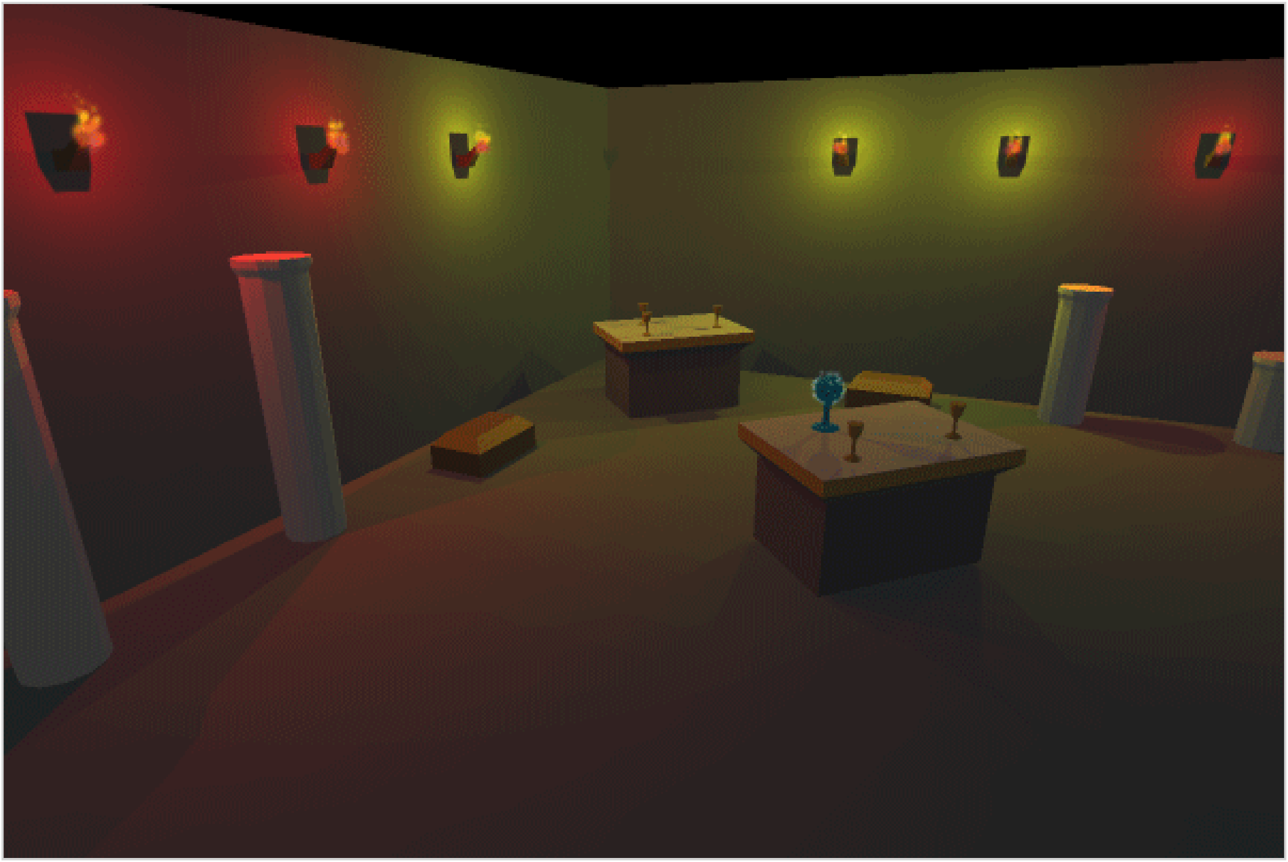
```
def match_exits(old_exit, new_exit):
    new_module = new_exit.parent
    forward_vector_to_match = old_exit.backward_vector
    corrective_rotation = azimuth(forward_vector_to_match) - azim
    rotate_around_y(new_module, new_exit.position, corrective_rot
    corrective_translation = old_exit.position - new_exit.positio
    translate_global(new_module, corrective_translation)

def azimuth(vector):
    # Returns the signed angle this vector is rotated relative to
    forward = [0, 0, 1]
    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 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.

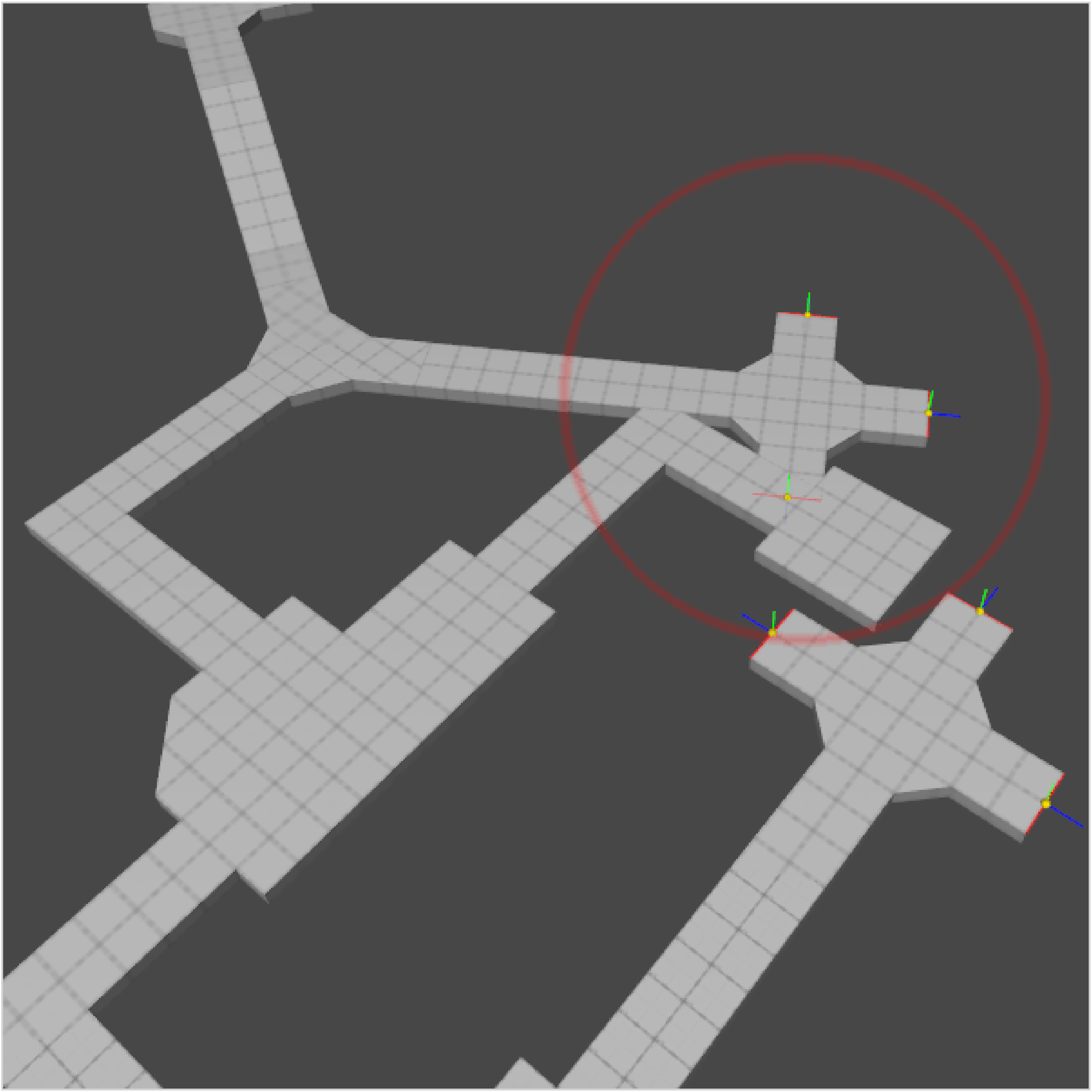
The same algorithm may be used to create procedural items. If you would like to create a sword, you could define its grip as a starting module. The grip would connect to pommel and to the cross-guard. The cross-guard would connect to the blade. By having just three versions of each of the sword parts,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

Difficulty:

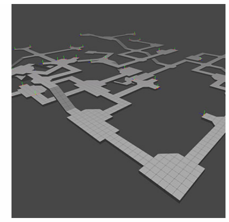
Intermediate

Length:

Short

Categories:

Implementation



Translations:

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

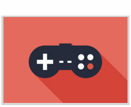
Translate this post Powered by native

Download Attachment

About Marcin Seredynski

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

Related Tutorials

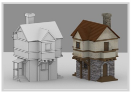


An Introduction to GameplayKit: Part 3

Code

Render an SVG Globe

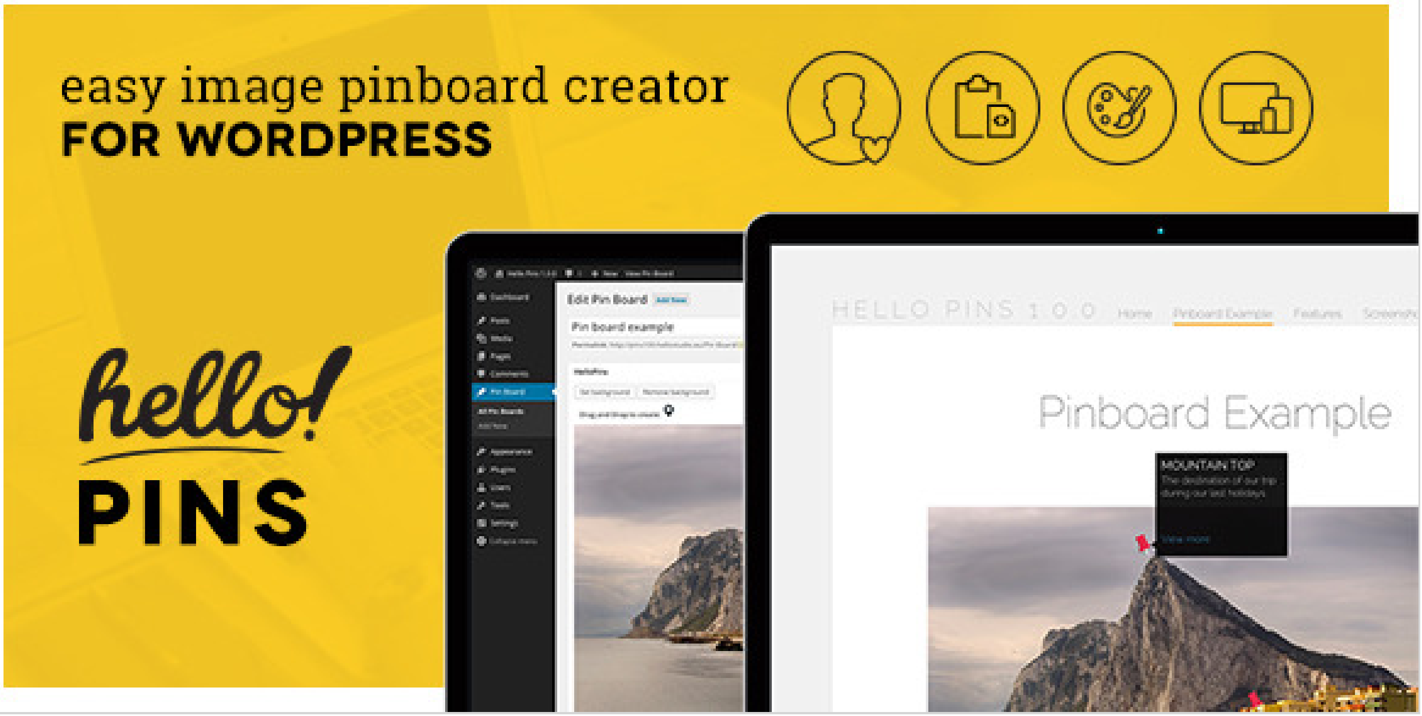
Code



Creating a Low Poly Medieval House in Blender: Part 1

3D & Motion Graphics

Envato Market Item



What Would You Like to Learn?

Suggest an idea to the content editorial team at Tuts+.

tuts+

Teaching skills to millions worldwide.

20,391 Tutorials **613** Video Courses

Follow Us

Help and Support

FAQ
Terms of Use
Contact Support
About Tuts+
Advertise
Teach at Tuts+
Translate for Tuts+
Meetups

Email Newsletters

Get Tuts+ updates, news, surveys & offers.

Email Address

Subscribe

Privacy Policy