Rooms and Mazes: A Procedural Dungeon Generator

↩ ↪

December 21, 2014 code dart game-dev roguelike

Several months ago I promised a follow-up to my previous blog post about turn-based game loops in my roguelike. Then I got completely sidetracked by self-publishing my book, *Game Programming Patterns*, and forgot all about it. I totally left you hanging.

Well, I finally got some time to think about my roguelike again and today, I'm here to… keep you hanging. Alas, you are at the mercy of my wandering attention span! Instead of game loops, today we're going to talk about possibly the most fun and challenging part of making a roguelike: generating dungeons!

Go ahead and click the little box below to see what we end up with:

Pretty neat, huh? If you want to skip the prose, the code is here.

One of my earliest memories of computing is a maze generator running on my family's Apple IIe. It filled the screen with a grid of green squares, then incrementally cut holes in the walls. Eventually, every square of the grid was connected and the screen was filled with a complete, perfect maze.

My little home computer could create something that had deep structure—every square of the maze could be reached from any other—and yet it seemed to be chaotic—it carved at random and every maze was different. This was enough to blow my ten-year-old mind. It still kind of does today.

What's in a dungeon?

Procedural generation—having the game build stuff randomly instead of using hand-authored content—is amazing when it works well. You get a ton of replayability because the game is different every time. As the person implementing the game, you also get the critical feature of not knowing what you're going to get even though you wrote the code. The game can surprise *you* too.

People get into procedural generation because it seems easier. Hand-authoring content is obviously a lot of work. If you want your game to have a hundred levels, you have to make a hundred things. But make one little random level generator and you can have a hundred levels, a thousand, or a million, for free!

Alas, it doesn't *quite* work out that way. You see, *defining the procedure* is a hell of a lot harder than just sitting down and banging out some content. You have to take some very nebulous, artistic chunk of your brain, figure out precisely what it's doing, and translate that to code. You're coding a simulation of yourself.

It must balance a number of technical and aesthetic constraints. For mine, I focused on:

- It needs to be **fairly efficient.** The generator only runs when the player enters a new level, so it doesn't have to be as *super* fast, but I still don't want a several second pause giving the player time to question whether they should be playing a game or doing something more productive with their life.
- The dungeon needs to be **connected.** Like the mazes on my old green-screen Apple, that means from any point in the dungeon, there is a way—possibly circuitous—to any other point.

  This is vital because if player has to complete a quest like "find the magic chalice" or "kill the cockatrice", it's pretty cruel if the dungeon drops that in some walled-off room the player can't get to. It also avoids wasting time generating and populating areas the player can never see.
- Moreso, I want dungeons to **not be perfect.** "Perfect" in the context of mazes and graphs (which are synonymous) means there is *only one* path between any two points. If you flatten out all of the windy passages, you'll discover your twisty maze is really just a tree all crumpled up. Passageways branch but never merge. *Im* -perfect mazes have loops and cycles—multiple paths from A to B.

  This is a gameplay constraint, not a technical one. You could make a roguelike with perfect dungeons, and many simple roguelikes do that because generators for those are easier to design and implement.

  But I find them less fun to play. When you hit a dead end (which is often), you have to do a lot of backtracking to get to a new area to explore. You can't circle around to avoid certain enemies, or sneak out a back passage. Neither can the bad guys, for that matter.

  Fundamentally, games are about making decisions from a set of alternatives. At a literal level, perfect dungeons only give you one path to choose from.
- I want **open rooms.** I could make dungeons just be nothing but mazes of narrow passages, but then you could never get surrounded by a

horde of monsters. It would feel claustrophic and kill a bunch of interesting combat tactics.

Wide open areas are critical for area effect spells, and big dramatic battles. They also provide space for interesting decorations and themed areas. Vaults, pits, traps, treasure rooms, etc. Rooms are the high points of the hero's journey.

- I want **passageways.** At the same time, I don't want the dungeon to *just* be rooms. There are some games that create levels this way where doors directly join room to room. It works OK, but I find it a bit monotonous. I like the player feeling confined part of the time, and having narrow corridors that the player can draw monsters into is a key tactic in the game.

- All of this needs to be **tunable.** Many roguelikes have one huge multi-floor dungeon where depths vary in difficulty but not much else. My game is different. It has a number of different *areas* . Each has its own look and feel. Some may be small and cramped, others spacious and orderly.

  I solve this partially by having multiple distinct dungeon generation algorithms. Outdoor areas use an entirely different process. (I should probably write about that too sometime. Look, another unfulfilled promise!) But coding a new dungeon generator from scratch for *every* area is a huge time sink. Instead, I want the generator to have a bunch of knobs and levers I can tweak so I can make a number of areas that share the same code but have their own feel.

A room with a view

I've been working on this game pretty much forever (it's gone through four different implementation languages!) and I've tried a number of different dungeon generators. My main source of inspiration is a game called Angband . The only thing I've sunk more of my life into than working on my game is playing that one.

Angband is fantastically old. When it forked off of Moria, Nancy Kerrigan had just taken a round of melee damage from a club-wielding troll. On machines of that time, it was much harder to make a fast dungeon generator, and Angband's is pretty simple:

1. Sprinkle a bunch of randomly located, non-overlapping rooms.

2. Draw random corridors to connect them.

To ensure rooms don't overlap, I just discard a room if it collides with any previously placed one. To avoid a possible infinite loop, instead of trying until a certain number of rooms are successfully *placed* , I do a fixed number of *attempts* to place rooms. Failure becomes more common as the dungeon gets fuller—after all, you can only fit so many rooms in a given area—but tuning this gives you some control over room density, like so:

Attempts: 200

A dark and twisty passageway

Most of the dungeon generators I've written start with this. The hard part, by far, is making good passageways to connect them. That's really what this post is about—a neat way to solve that problem.

Angband's solution is brute force but surprisingly effective. It picks a pair of rooms—completely ignoring how far apart they are—and starts a passageway that wanders randomly from one (hopefully) to the other. It's got a few clever checks to keep things from overlapping too much but passageways can and do cut through other rooms, cross other passages or dead end.

I tried implementing that a number of times but (likely failures on my part) never got to something I really liked. The corridors I ended up with always looked too straight, or overlapped other stuff in unattractive ways.

Then, a few months ago, I stumbled onto a description of a dungeon generator by u/FastAsUcan on the /r/roguelikedev subreddit. His generator, Karcero , is based on Jamis Buck's dungeon generator. If you've ever done any procedural dungeon generation, you know—or should know—who Buck is. He's got a ton of great articles on random mazes.

Years ago, I remember seeing an actual dungeon generator he wrote for use with pen-and-paper Dungeons & Dragons. Unlike most of his maze stuff, this had actual rooms, and the results looked great.

But, at the time, I didn't know how it *worked* . How do you go from mazes to open winding corridors and rooms? I tucked this open question away in the corner of my mind and immediately forgot about it.

The post by FastAsUcan provides the answer. It works like so:

1. Make a perfect maze. There are a number of different algorithms for this, but they're all fairly straightforward.

2. Make the maze *sparse* . Find dead end passages and fill them back in with solid rock.

3. Pick some of the remaining dead ends and cut holes in them to adjacent walls. This makes the maze imperfect. (Remember, this is a good thing!)

4. Create rooms and find good locations to place them. "Good" here means not overlapping the maze but *near* it so you can add a door and connect it.

The magic step, and the piece I was missing, is *sparseness* . A normal maze fills every single square of the world, leaving no areas where you can fit a room. The trick that Jamis and FastAsUcan do here is to carve the whole maze and then *uncarve* the dead ends.

Doing that is actually pretty easy. A dead end is just a tile that has walls on three sides. When you find one of those, you fill that tile back in. That may in turn make the tile it connects to a dead end. Keep doing this until you run out of dead ends and you'll end up with lots of solid area where rooms can be placed.

Of course, if you do that starting with a perfect maze and run to completion, you'll erase the whole maze! A perfect maze has no loops so *everything* is a dead end if you follow passages long enough. Jamis' solution is to not erase *all* of the dead ends, just some. It stops after a while. Something like this:

Corridors to leave: 1000

Once you do that, you can start placing rooms. The process Jamis uses for this is interesting. He picks a room size and then tries to place it on every single location in the dungeon. Any location that overlaps a room or passageway is discarded. The remaining positions are "ranked" where rooms that are near passageways are better. It then picks the best position and places the room there, and puts some doors between the room and the passage.

Rinse, lather, repeat and you've got yourself a dungeon.

Rooms *then* mazes

I went ahead and coded this up exactly as described. It went OK, but I found that the process of placing rooms was pretty slow. It works well for dungeons of the small size you do for a tabletop role-playing game, but not so much at the scale of a computer roguelike.

So, I did some tinkering and came up with a slight variation. My contribution is pretty minor, but I thought it would be worth writing down. (Honestly, I just think it's fun to watch animated dungeon generators, and the prose is pure fluff.)

Where Buck and Karcero start with the maze and then add the rooms, mine does things in the opposite order. First, it places a bunch of random rooms. Then, it iterates over every tile in the dungeon. When it finds a solid one where an open area *could* be, it starts running a maze generator at that point.

Maze generators work by incrementally carving passages while avoiding cutting into an already open area. That's how you ensure the maze only has one solution. If you let it carve into existing passages, you'd get loops.

This is conveniently exactly what you need to let the maze grow and fill the odd shaped areas that surround the rooms. In other words, a maze generator is a randomized flood fill algorithm. Run this on every solid region between the rooms and we're left with the entire dungeon packed full of disconnected rooms and mazes.

Each color here represents a different region of connected tiles.

Looking for a connection

All that remains is to stitch those back together into a single continuous dungeon. Fortunately, that's pretty easy to do. The room generator chooses odd sizes and positions for rooms so they are aligned with the mazes. Those in turn fill in all of the unused area, so we're assured that each unconnected region is only a single tile away from its neighbors.

After filling in the rooms and mazes, we find all of those possible *connectors* . These are tiles that are:

1. Solid rock.
2. Adjacent to two regions of different colors.

Here they are highlighted:

We use these to tie the regions together. Normally we think of the entire dungeon as a graph with each tile a vertex, but we're going to go up a level of abstraction. Now, we treat each *region* of tiles as a single vertex and each connector as an edge between them.

If we use *all* of the connectors, our dungeon would be way too densely connected. Instead, we carve through just the connectors we need to get each region connected to the whole *once* . In fancy terms, we're finding a *spanning tree* .

The process is pretty straightforward:

1. **Pick a random room to be the main region.**

2. **Pick a random connector that touches the main region and open it up.** In the demo, it does that by placing a door, but you can do an open passageway, locked door, or magical wardrobe. Be creative.

   Note that this process is agnostic about rooms and mazes. It just deals in "regions". That means it can connect rooms directly to other rooms sometimes. You can avoid that if you want, but I find the resulting dungeons more fun to play.

3. **The connected region is now part of the main one. Unify it.** In the demo, I use a little flood fill algorithm to color in the newly merged region because it looks pretty. In a real implementation, you don't need to mess with tiles. Just make a little data structure that tracks "region X is now merged".

4. **Remove any extraneous connectors.** There are likely other existing connectors that connect the two regions that just merged. Since they no longer connect two separate regions and we want a spanning tree, discard them.

5. **If there are still connectors left, go to #2.** Any remaining connectors imply that there is still at least one disconnected region. Keep looping until all of the unconnected regions are merged into the main one.

Earlier, I said that I don't want a perfect dungeon because they make for crappy gameplay. But, since this creates a spanning tree, that's exactly what we've got. We only allow a single connector between any two regions so our dungeon *is* a tree and there's only a single path between any two points.

Fixing that is pretty simple. In step 3, when we cull the unneeded connectors, we give them a *slight* chance of being opened up. Something like:

```
if (rng.oneIn(50)) _carve(pos, CELL_DOOR);
```

This occasionally carves an extra opening between regions. That gives us the imperfect loops we want to make the dungeon more fun to play in. Note that this is also easily tunable. If we make the chance more likely, we get more densely connected dungeons.

Uncarving

If we stop here, we'll get dungeons that are packed chock full of maze corridors, most of which are dead ends. That has a certain sadistic appeal, but isn't exactly what I'm going for. The last remaining step is the "sparseness" pass described earlier.

Now that we've got all of our rooms connected to each other, we can remove all of the dead ends in the maze. When we do that, the mazes are reduced to just the winding set of passageways needed to connect the rooms to each other. Every corridor is guaranteed to go somewhere interesting.

What we ended up with

In summary:

1. Place a bunch of random non-overlapping rooms.

2. Fill in the remaining solid regions with mazes.

3. Connect each of the mazes and rooms to their neighbors, with a chance to add some extra connections.

4. Remove all of the dead ends.

I'm pretty happy with it so far. It's not perfect, though. It tends to produce annoyingly windy passages between rooms. You can tune that by tweaking your maze generation algorithm, but making the passageways less windy tends to make them wander to the edge of the dungeon, which has its own strange look.

The fact that rooms and mazes are aligned to odd boundaries makes things simpler and helps fill it in nicely, but it does give the dungeon a bit of an artificially aligned look. But, overall, it's an improvement over what I had before, and the dungeons it makes seem to be pretty fun to play.

If you want to see for yourself, you can play the game right in your browser. The code for these demos is here, but it's pretty gnarly. Making them animated adds a lot of complexity. Instead, here is the much cleaner implementation my game uses.

As a bonus for making it this far, here's a super dense giant dungeon. I find it hypnotic:

Hi! I'm **Bob Nystrom**, the one on the left.

I wrote a book called **Game Programming Patterns**.

You can email me at `robert` at this site or follow me on twitter at `@munificentbob`.

Elsewhere

- Code at github
- Code at bitbucket
- Tweets at twitter
- Photos at flickr
- Video at vimeo
- Posts at google+

Categories

- code 67
- language 41
- magpie 24
- c-sharp 13
- dart 13
- game-dev 12
- java 10
- cpp 8
- game-patterns 6
- parsing 6
- roguelike 6
- design 5
- go 5
- js 4
- book 3
- c 3
- finch 3
- python 3
- ruby 3
- blog 2
- f-sharp 2
- lua 2
- ai 1
- beta 1
- blogofile 1
- game 1
- jasic 1
- javascript 1
- music 1
- oop 1
- optimization 1
- oscon 1
- politics 1
- scheme 1
- typescript 1
- visualization 1

All 73 articles …

This blog is built using jekyll. The source repo for it is here.