Procedural Level Design in Eldritch

David Pittman
Minor Key Games

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

About me

SMU GUILDHALL

gearbox software

2K MARIN

Minor Key Games

I've been making games professionally for about 9 years.

My background in the industry is as a programmer. I started as an intern at Gearbox while attending SMU Guildhall. When I graduated, I moved out to the Bay Area and eventually landed at 2K Marin, where I was an AI programmer on BioShock 2 and (for a time) lead AI programmer on The Bureau.

About two years ago (in 2013), I left to co-found Minor Key Games with my twin brother Kyle, and I made a game called Eldritch.

What's Eldritch?

- "A Lovecraftian action roguelike."
- "A first-person action game inspired by roguelikes, immersive sims, and H. P. Lovecraft."
- We're all professionals here, right?
  What I really meant was…

While making Eldritch, I had trouble describing it to friends and family. It wasn't an especially unique or original game, but a hybrid of genres that could be difficult to envision.

I tended to describe it in terms of my inspirations in roguelikes and immersive sims, but what I was really trying to say was…

Dishonored…

If you mix first person combat, stealth, and magic

+ Spelunky…

With procedural levels and permanent player death

+ Cthulhu…

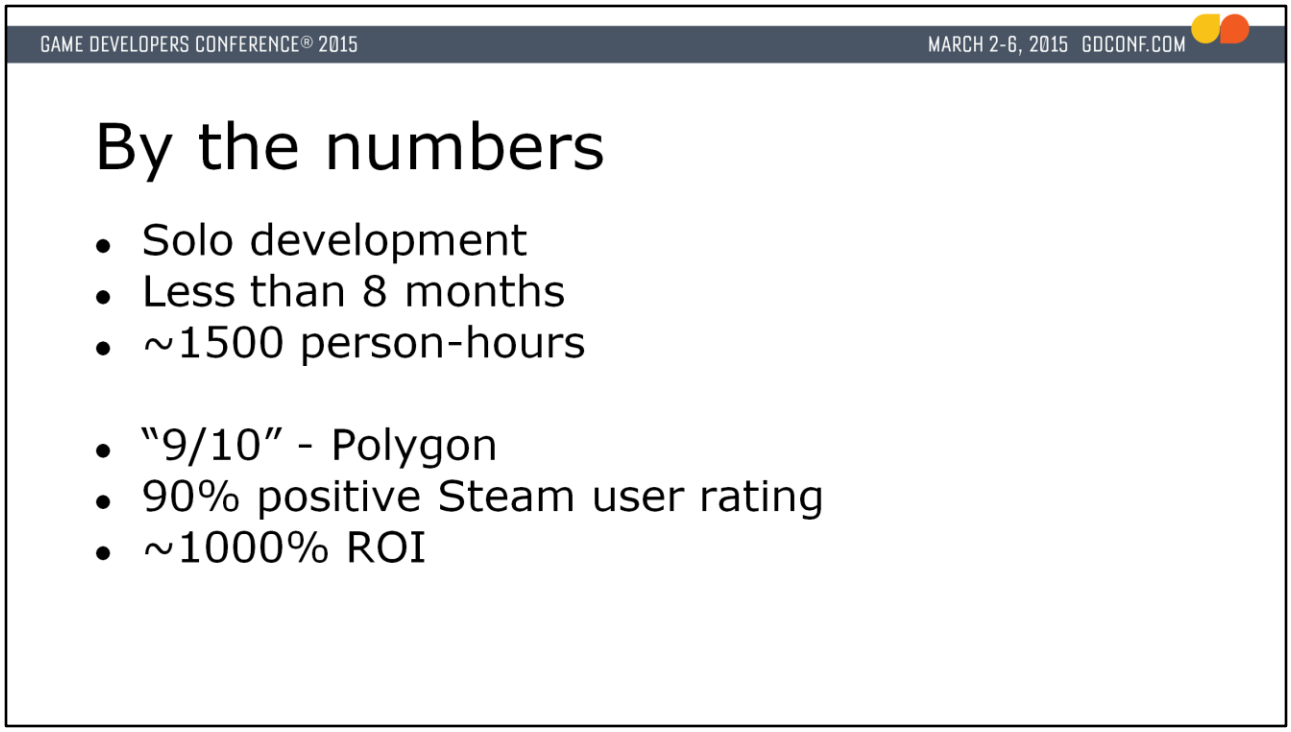
And dress it up in Lovecraftian fiction

- Money…

And make it on a shoestring budget

= Eldritch

You get something that looks like Eldritch.



Minor Key Games is a two-person company, but we make individual projects. So I made Eldritch mostly by myself, in a span of slightly less than 8 months.

I don't track my hours, but a back of the envelope calculation says it took about 1500 hours to make Eldritch.

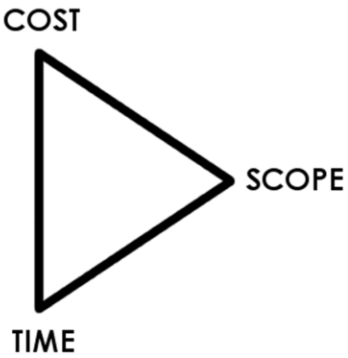
For comparison, a conservative big-budget game—say, 100 people working for 2 years without crunching—would be 400,000 hours. So Eldritch took a fraction of a percent of the work of a big-budget game, by this measure. I don't say this to brag (because Eldritch definitely shows its hasty construction in the simple art and rough edges), but to explain the constraints I was working within.

Despite its speedy development, Eldritch was fairly well received. It made a tenfold return on investment (and is still selling). That's not because it sold millions, but because development was so lean that it was fairly easy to recoup its cost.

Level design goals

Why procedural level design?

- Perennial discovery
- Project constraints
- Team strengths
- Content reuse
- Multiple ways to use it
  - Runtime vs. offline
  - Supplement handcrafted le
- Procedural level generatic



Roguelikes and other genres depend on generated levels to provide an aspect of perennial discovery.

Project constraints often encourage generation. In the case of Eldritch, the “cost vs. time vs. scope” triangle flattens into a line between time and scope, because I couldn’t spend more to finish the game faster. (That almost never works anyway, but that’s a separate talk.)

My background as a programmer meant I would be much more efficient developing a generation algorithm than building levels by hand.

Level generation (and the replayable nature of roguelikes) encourages content reuse.

Levels can be generated at runtime or offline. For example, an open world game could generate its terrain and then populate points of interest on it by hand. Generated levels can also supplement handcrafted content. A 10-hour campaign could be followed by a generated endless mode to extend a game’s lifespan.

Procedural level generation IS design. It’s not strictly the domain of engineering. In order to write a good generation program, we need to understand what makes a good level so we can evaluate the output and tweak the system to give the desired results.

Eldritch level goals

- Compact
- Handcrafted
- Continuous
- Semi-coherent

Lovecraft reference

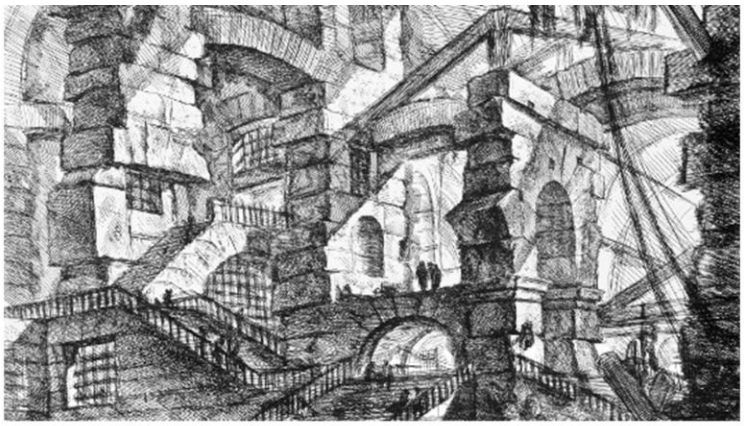
He talked of his dreams in a strangely poetic fashion; making me see with terrible vividness the damp Cyclopean city of slimy green stone—whose geometry, *he oddly said, was* all wrong…

"The Call of Cthulhu"

Art reference

Marc Simonetti, *R'lyeh*

Art reference



Giovanni Battista Piranesi, *Carceri Plate XIV*

Art reference

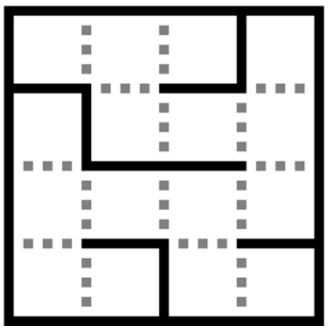
- Right angles
- Cyclopean masonry
- Complex layouts
- Verticality
- Atmosphere

Results

Overview

Overview

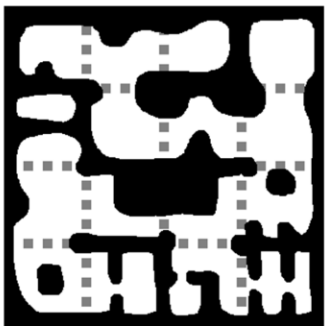
- Build a 3D maze



I'm visualizing the maze here in 2D because it shows better on the slides, but the maze in Eldritch is a 4x4x3 volume with paths up and down as well as North South East West.

Overview

- Build a 3D maze
- Fill it with room modules



Overview

- Build a 3D maze
- Fill it with room modules
- Add enemies and items

Overview

- Build a 3D maze
- Fill it with room modules
- Add enemies and items

Links:
http://eldritchgame.com/about.html#source
http://tinysubversions.com/spelunkyGen/

This presentation does not show any source code, but for those interested, I have released the code for Eldritch under a permissive license.

This process owes a lot to the Spelunky level generator and similar maze+module algorithms. Spelunky's generator was written by Derek Yu and was thoroughly documented with an interactive implementation by Darius Kazemi at the second link.

Maze generation

"Maze" is a four letter word

Describing a level as a maze usually has bad connotations. It implies a confusing structure, with unclear direction to the goal, and a homogeneous environment that makes every corridor look the same as every other corridor.

For Eldritch, a confusing structure is desired. It contributes to the Lovecraftian fantasy of getting lost in an inconceivably old and complex city.

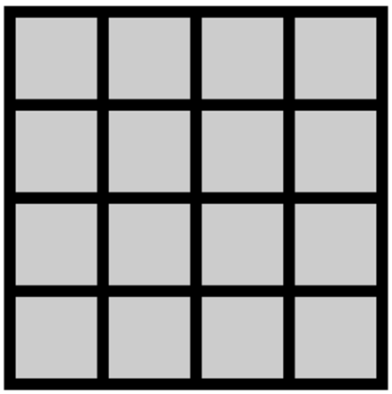
For roguelikes, unclear direction is part of the central mechanics. There is no explicit wayfinding; instead, the player is expected to explore and eventually uncover the exit on their own.

But the homogeneity of mazes was something to be avoided. I wanted the mazelike structure, but I wanted it decorated with memorable visual landmarks to help the player distinguish between parts of the map.

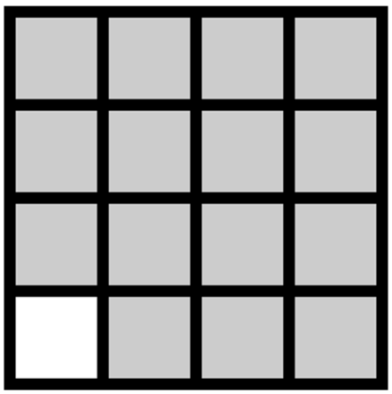
Maze generation

- Prim's algorithm/growing tree
  - Start from a single visited cell
  - Randomly open a path to any adjacent unvisited cell
  - Repeat until all cells are visited

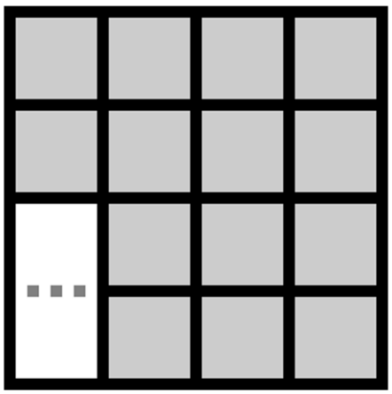
Maze generation



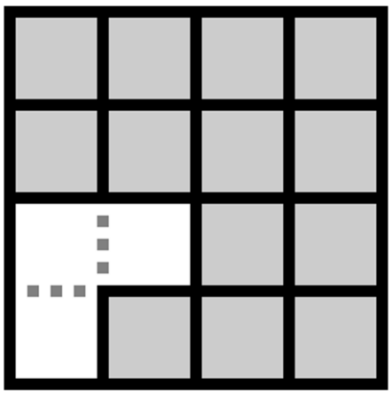
Maze generation



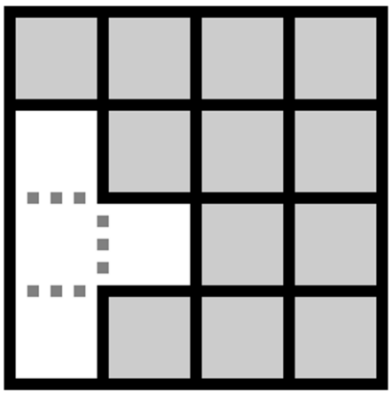
Maze generation



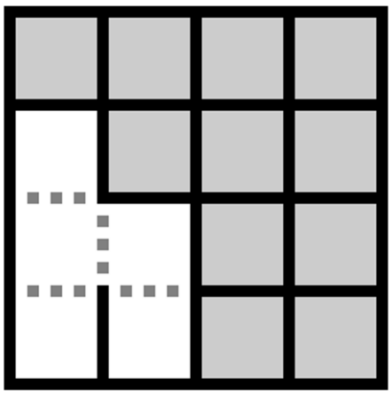
Maze generation



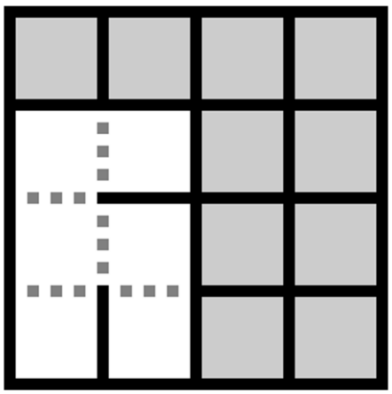
Maze generation



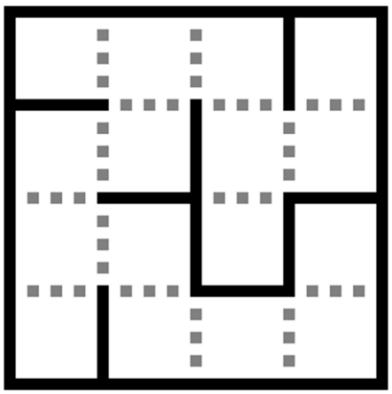
Maze generation



Maze generation



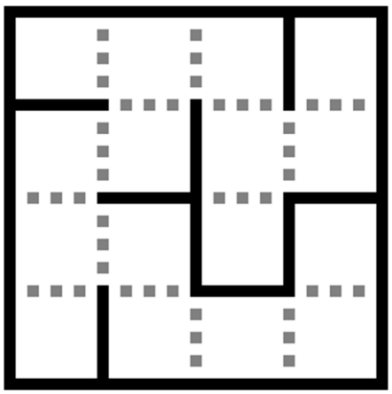
Maze generation



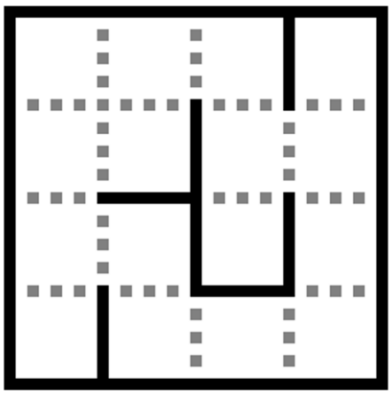
Modifications

- Bias expansion from most recent room
- Balance horizontal/vertical paths
- Randomly knock down walls

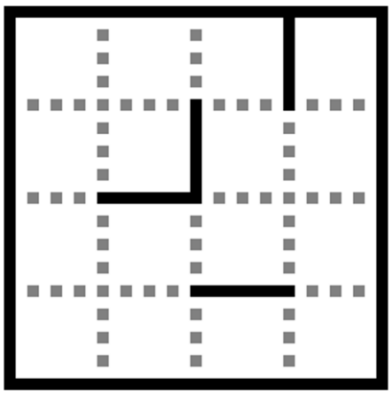
Modifications



Modifications



Modifications



Prescribed rooms

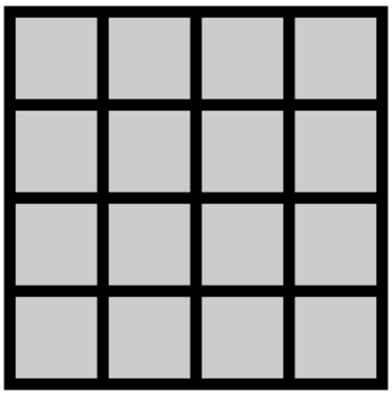
- Not all space is alike
- Don’t search maze for fit
- Seed maze with prescribed rooms

“Prescribed rooms” are rooms whose configuration are determined by the special features that will occupy them. For example, a store in Eldritch always has one entrance, so it must fit into a dead end cell in the maze.

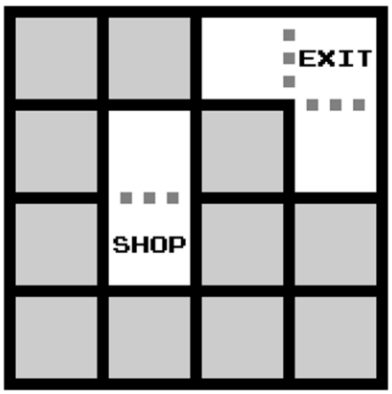
For more complicated configurations, there is no guarantee that we can find a matching cell in the maze; and even if we do, there is no guarantee about where in the maze it will be.

So instead of searching the maze for these configurations, we seed the maze with prescribed rooms, and then forbid Prim’s algorithm from modifying those cells further.

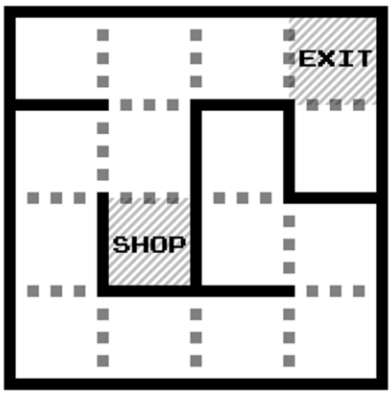
Prescribed rooms



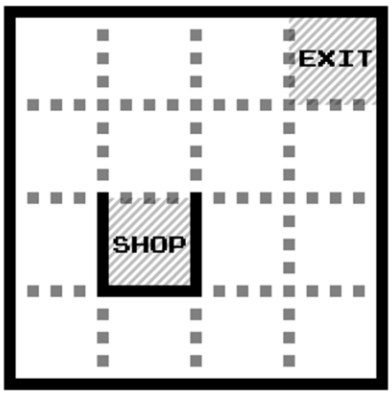
Prescribed rooms



Prescribed rooms



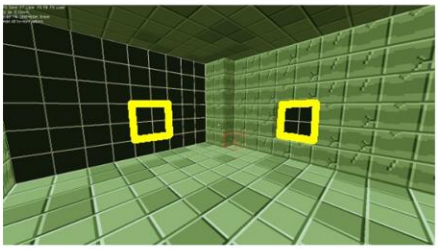
Prescribed rooms

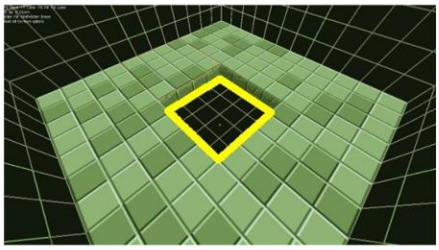


Module design

Module design

- Handbuilt voxel modules
- 6 exits = 64 configurations
- Must connect to any other module

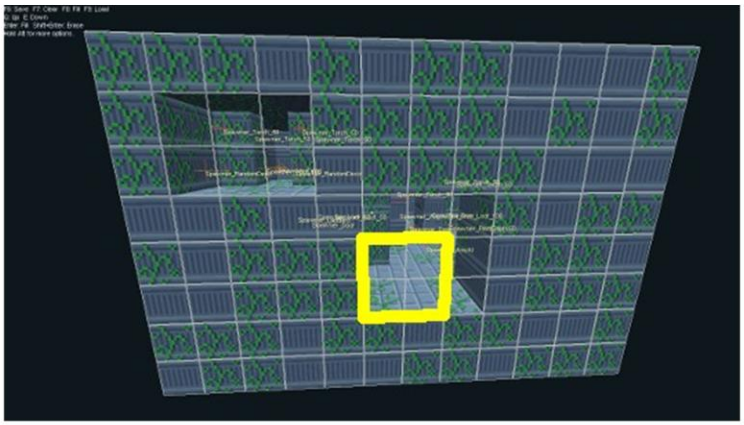




Room modules are built from voxels in a simple in-engine tool. Each module is 12 x 12 x 8 voxels (at a 1m resolution).

To guarantee that any two adjacent modules are traversable, modules are required to have an opening in at least a safe region (shown here in yellow).

Module design



When I put openings outside the safe region, interesting results emerged.

Module design

This space typically appears as a balcony overlooking the adjacent room.

Module design

But sometimes, it collides with adjacent features. Here, it looks like it was part of a neighboring feature that is crumbling away.

Module design

The low resolution of the voxel grid means that features tend to join in desirable ways. Here, the upper level of the adjacent module appears to continue seamlessly into that space.

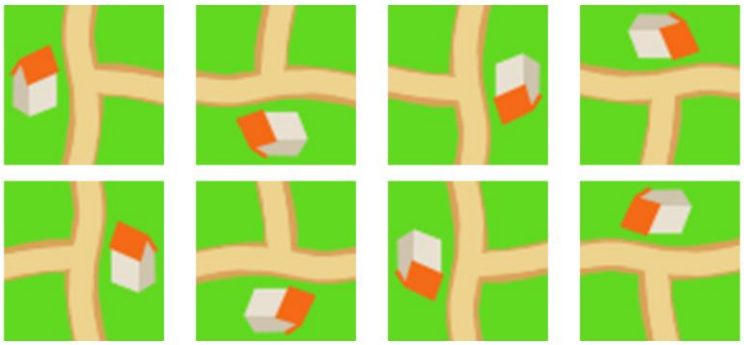
Module design

Sometimes, the adjacent module has this region walled off. In this case, the player may open the door and find themselves facing a blank wall.

So I put a guaranteed item spawner on the ground here. Now when the player finds this structure, it appears to have been intentionally designed as a loot closet.

Module transforms

- Rotation about z-axis
- Reflection about x- or y-axis



Modules may be rotated about the vertical axis to fit into the maze. So a module with exits to the North South and East could be spun into any of the positions on the top row.

Modules can also be mirrored about the horizontal axes. This produces redundant configurations (note that each configuration on the bottom row also exists on the top), so it isn't strictly necessary. But mirrored modules are visibly different from unmirrored ones, so doing this stretches our modules a bit further.

Mirroring also adds a small constraint; I couldn't, for example, spell a word out on voxels because the word would read backward when the module was mirrored.

Module transforms

- Rotation about z-axis
- Reflection about x- or y-axis
- 24 configurations instead of 64
- Statistical report of configuration frequency
- ~60 modules per world

Using these transforms reduces the set of 64 configurations to 24. (This is left as an exercise for the reader.)

To avoid homogeneity, I wanted multiple variations on each configuration; for example, multiple T-junction modules so that every T-junction doesn't look alike.

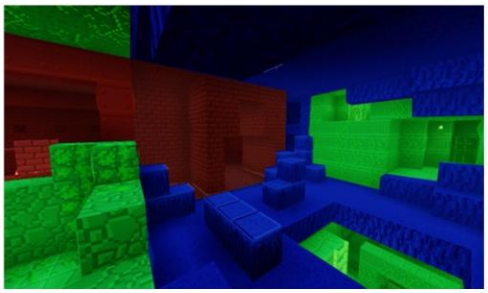
I wrote a tool that ran through 10000 iterations of the generator and dumped a spreadsheet of the frequency of each configuration. This provided a comparative metric of modules. For example, L-junctions might appear 3x as often as T-junctions.

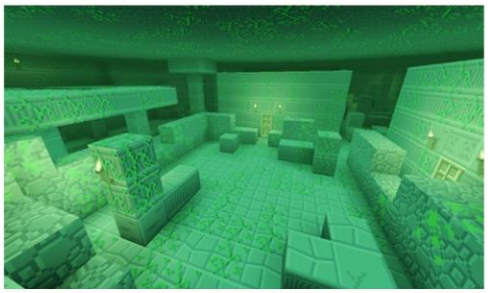
There was no absolute metric for how many to build. I chose that if a configuration appeared N times in a map (on average), then I would build N modules to fit that configuration. For configurations which appear fewer than once per map (on average), such as the very rare room with exits to the North South East West Up and Down, I still needed one module.

Between the core modules and "feature" modules (shops, vaults, shrines, etc.), each world had about 60 modules, and the base game shipped with around 250 modules.

Themes

- Visual coherency
- Partial module sets





Each world had a few visual themes in its modules. The first world is loosely based on The Shadow Over Innsmouth, and is meant to represent an ancient sunken city beneath a New England town. I illustrated this with red brick warehouse modules, purple cavern modules, and white stone ruin modules.

But throwing these all into the mix together produced visually incoherent levels. In the image on the left, we have warehouses and caverns and ruins all visible from a single vantage point. It's a mess.

I didn't want to build a complete module set for each of these themes (warehouse, caverns, ruins), because that would approximately triple the module count.

Instead, I explicitly declared the theme of each module, and made the generator elect a primary theme for each level. It then prefers to use modules from that theme when they are available; but it can fall back to using out-of-theme modules when needed.

This produces visually coherent images like the one on the right, while still requiring only partial module sets for each theme. And when the occasional warehouse or cavern shows up in a ruins map, it provides a landmarking contrast instead of being so much visual noise.

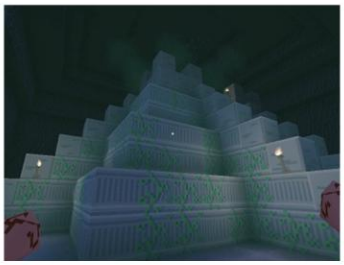
Feature modules

- Complement of prescribed rooms

```
[Feature_Water_Bank]
SliceMinX    = 0
SliceMaxX    = -1
SliceMinY    = 0
SliceMaxY    = -1
SliceMinZ    = 1
SliceMaxZ    = -1
Chance       = 0.666
NumRoomDefs = 2
RoomDef0     = "Rooms/Water/bank-n.eldritchroom"
RoomDef1     = "Rooms/Water/bank-n-2.eldritchroom"
```

Feature modules

- Complement of prescribed rooms
- Not generated elsewhere
- Shops, banks, shrines, "hero rooms"



Feature modules are the modules which go into prescribed rooms in the maze. (In fact, in the game's data, they are defined in the same place.)

This way, the generator won't put any random dead-end module into a cell that was marked for a shop; and likewise, the feature modules are kept out of the mix so a shop can't be put into any random dead-end cell.

In addition to systemic rooms like shops, I built a handful of "hero rooms" for each world: uniquely challenging rooms like the heavily guarded throne room in the left image. These had a somewhat rare chance of appearing, and were intended to create a memorable experience when discovered.

I also composed large features out of multiple modules to create unique locations. The ziggurat in the right image appears at the end of the first world. It fills 8 modules (in a 2x2x2 arrangement) and is therefore larger than any other singular feature in the level could possibly be.

Entity population

Entity goals

- Challenge (enemies, traps)
- Incentive to explore (loot)
- Placed in sensible locations
- Controlled group populations

In addition to the common goals for entities in a level (to challenge the player and provide incentive to explore), Eldritch had goals related to its procedural nature. Entities needed to appear in logical places, and I needed a way to control group populations so that I could ramp up the number of enemies over the course of the game.

Spawners

Developing placement algorithms for every kind of entity would have been unnecessarily difficult. Instead, it made sense to place entities by hand in each module.

But I wanted a random element, so that the player couldn't easily predict that Room X always has Enemy Y and Loot Z.

So instead of placing entities directly, I placed entity *spawners* in the modules. During level generation, some rules determine what things are actually spawned at those locations.

Spawners

- Basic

```
[Spawner_Player]
Entity  = "Player"
OffsetZ = 0.3078125
```

The most basic spawner is one which spawns a certain entity 100% of the time. For example, the player spawner: we always need a player.

Spawners

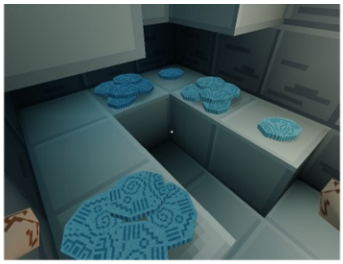
- Basic
- Random chance

```
[Spawner_Snare_33]
Chance = 0.333
Entity = "TrapBolt"
```

I could give spawners a random chance to spawn their entity. For example, traps usually had a 1:2 or 1:3 chance to spawn. This way, I could cover a room with trap spawners but get random sparser arrangements when the level was generated.

Spawners

- Basic
- Random chance
- Weighted random entity



```
[Spawner_BankArtifacts]
Chance             = 0.75
NumSubSpawners     = 2
SubSpawner0        = "Spawner_Artifact"
SubSpawner0Weight  = 2.0
SubSpawner1        = "Spawner_Artifact5"
SubSpawner1Weight  = 3.0
```

I could also change what entity was spawned, using a weighted list. In this example, the coins which a player finds in a bank vault have a 75% chance of spawning at all, and a 3:2 chance to be a big pile of coins instead of a single coin.

I also used weighted randomness for enemies. I would place generic enemy AI spawners, and then write a weighted table of enemies to get a balanced distribution of grunts and tougher enemies.

Resolve groups

- Control population
- Enemy AIs, healing fountains
- Minimum distance from player spawn

To control populations of certain groups of entities, I introduced a concept called “resolve groups”. The generator builds a set of all the spawners belonging to a group, then culls them and resolves it into a subset of spawners to actually use.

Resolve groups were used to control enemy population. Every enemy AI spawner was tagged as part of the “AI” group and gathered into a large set. Spawners too close to the player spawn were removed from the set to avoid the case where the player starts a level face-to-face with an enemy. Then a number of the remaining spawners were chosen at random to spawn their entities.

Spawner placement

- Loot in dead ends (1 exit)
- Traps in hallways (2 exits)
- Enemies at junctions (3+ exits)

At first, I distributed entities somewhat evenly across all modules. During playtesting, I realized that I could use entities more effectively to sculpt the pacing of a level.

Players are great at identifying patterns, and will quickly recognize a dead end module. If the player has sufficient health and ammo, they will ignore a dead end and proceed through the level. By pushing loot into dead ends instead of scattering it randomly throughout the maze, I encouraged players to explore that space which would otherwise have been wasted.

I started putting traps into hallways (or more generally, any room with 2 exits). These modules exist only to be traversed; traps make that traversal more interesting. (I also used certain kinds of traps to guard loot in dead ends.)

Enemies in Eldritch wander randomly, so I was less strict about their placement. But the ideal place for them was in junctions (modules with 3 or more exits), open spaces where the player might find more interesting ways to fight or circumvent the enemy entirely.

If I'd been a more experienced designer who already knew how to use level features to shape the player's experience, I probably would have been thinking about this sort of thing sooner. And I think this is where procedural levels have a ton of potential to grow and improve.

"Procedural level generation IS design."

Thank you!

- http://minorkeygames.com
- @dphrygian
- david.pittman@gmail.com

Bonus Slides

These are some slides I cut from the presentation due to time and format.

Navigation

- Path search on voxel grid
- Collidable entities rasterized into grid
- Flags for [locked] doors, refcount
- AI capabilities: jump, fly, doors

AI pathfinding is done on the voxel grid. This vastly simplified the problem, as there wasn't any offline path network to generate and no need to stitch path networks of modules together.

It also supported pathfinding for flying enemies, as it is a fully volumetric structure (compared to conventional ground-based networks).

Collidable entities are rasterized into the grid, with a refcount so they can be cleared when all destructible entities in the space have been destroyed.

Visibility and culling

- Maze provides coarse vis info…
- …but geo can be removed during play
- Worst case: every room is visible
- Lazy solution: always use the worst case
- View frustum and distance culling only

This slide is pretty self explanatory. Basically, I did no occlusion culling for rendering, and no culling of any kind for the sim. It's lazy, and Eldritch tended to perform poorly on older machines because of this.

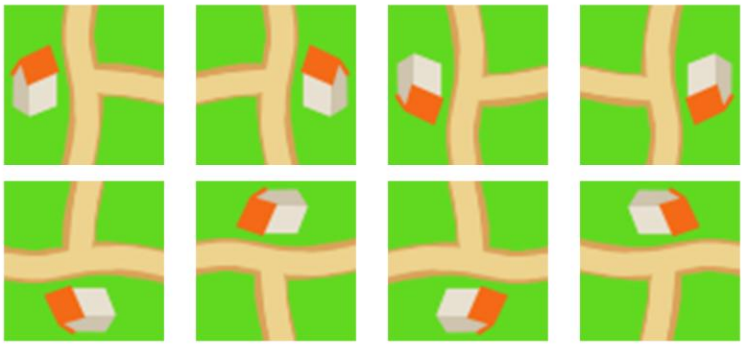
Performance

- Maze generation is negligible
- Module placement is negligible
- Spawner selection is negligible
- Constructing world mesh is expensive
- Spawning entities is expensive

The generator costs virtually nothing to run. The only parts of generating a level that take any significant time are building the voxel hull and constructing entities (both of which happen whether generating a new level or loading into an old one).

Module transforms

0: 000b
1: 001b
2: 010b
3: 011b
4: 100b
5: 101b
6: 110b
7: 111b



001b: Mirror X
010b: Mirror Y
100b: Rotate 90

Uniquely enumerating the module transforms was necessary in order to iterate through them and test each possible way a module could fit into the maze.

Each of the eight configurations can be generated from the ordered application of three atomic transforms, chosen to prevent redundant results. For Eldritch, I chose mirror X, mirror Y, and rotate by 90 degrees (counter-clockwise), but there are other transforms which also work for this problem.

Then the binary form of the enumeration for a transformed module actually describes the steps to achieve that transformation. For example, “101b” means to do the first and third steps (mirror X and rotate 90) and skip the second step (mirror Y).