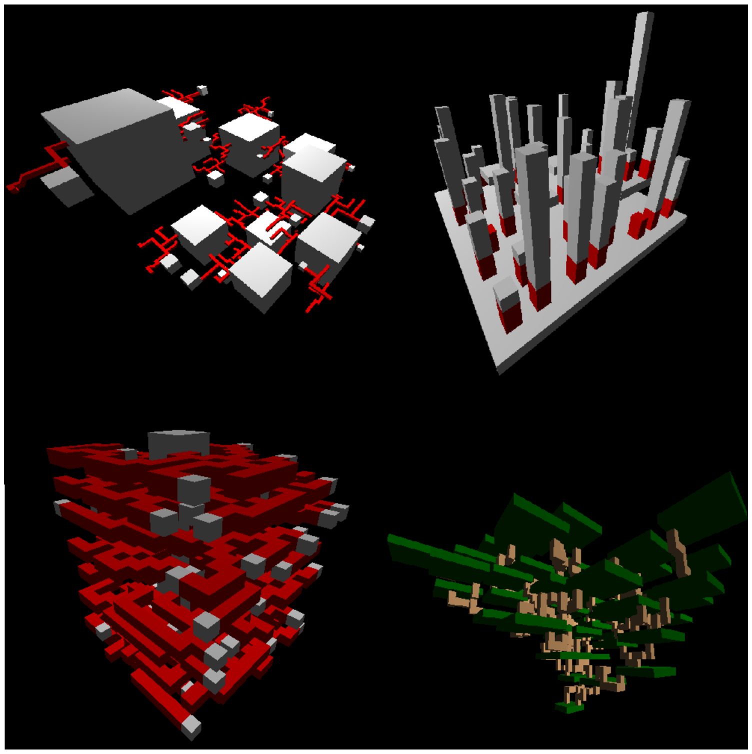


Dungeon Generation for Use in Interactive Media

David Koloski, Dwight Naylor

Rensselaer Polytechnic Institute

May 11, 2015

Abstract

Procedurally Generated Content (PGC) has been a staple of interactive media for many years, allowing a designer to provide experiences that express their intent without hand-crafting all the individual parts of the environment. Dungeons were one of the first applications of dynamically generated content and remain a popular way to provide unique user experiences. However, the methods of generating these have remained relatively unexplored and remain implemented on a case-by-case basis, often with little regard for formalization or rigor. We aim to provide a solid theoretical foundation for discussing dungeon generation as well as formulate a generic and robust algorithm for doing so.

Introduction

Procedurally generated content (PGC) has existed in interactive media since its inception through games like The Legend of Zelda *(Nintendo, 1986)*, Rogue *(Troy and Wichman, 1980)*, The Elder Scrolls II: Daggerfall *(Bethesda Softworks, 1996), and* Diablo *(Blizzard Entertainment, 1998)*. These games showcased the versatility and potential of PGC, in particular through dungeon generation. By providing a consistent experience every time but allowing the implementation to vary, they provided replayability and eased the work required to create new assets. The demand for dungeon and dungeon-like procedural content has not decreased since, and seen a recent resurgence in roguelikes. However, while the demand for high quality procedurally-generated content has grown, the techniques for generating it have not grown any more advanced.

Other fields related to procedural generation have seen significant advancement explicitly for use in interactive media. SpeedTree, a suite of tools for generating high quality virtual foliage, has seen wide use in both the film industry and the game industry. Having been used in such high-budget productions as Avatar *(20th Century Fox, 2009)* and Avengers: Age of Ultron *(Marvel Studios, 2015)*, SpeedTree proves that dedicated, sophisticated, and high-quality PGC is in demand and applicable to a wide variety of situations. However, PGC has attempted only to fill an aesthetic space and need only look good, dungeons have a particularly interesting set of requirements.

As both a narratological and ludological tool, dungeons must not just look good, but also provide proper progression, create a navigable and interesting space for gameplay, and conform to restrictions regarding game narrative. In order to do so, dungeon generation must provide the flexibility for designers to govern exactly how generation should proceed as well as the conciseness for dynamic content to be created such that it conforms robustly and completely to a set of rules. We aim to explore the approaches for dungeon generation and formulate a limited approach of our own to generate high quality dungeons with significant flexibility at high speed.

Summary of Background Research

Papers on dungeon generation remain in limited quantity, but Linden, Lopes, and Bidarra provided an overview of many different techniques for generating dungeons and dungeon-like PGC and analyzed the best environment for use as well as the strengths and limitations of many approaches. Although there were many alternative methods, the primary ones discussed were cellular automata, generative grammars, genetic algorithms, and constraint-based generation. These cover the majority of cases, and each has clear strengths and weaknesses.

Cellular automata use game-of-life style rules and a discretized space to generate dungeons. By designing the rules properly, emergent behavior such as clumping can emerge naturally. The complexity of the ruleset is bounded by the size of the neighborhood of the cell, which is in many cases simple the Moore neighborhood of size one. Additionally, multiple generations of cellular automata are iterated to provide increasingly improved results. Starting with a grid of random data, the cellular automata determine how and where the space ‘evolves’ and places features on a per-tile basis. This approach works well for situations where a high degree of randomness and uniformity is desired, but is unsuitable for most other applications which require some degree of intent. Generating specific features on a tile-by-tile basis is easy, but coordinating action across multiple cell is difficult, if not impossible. Increasing the neighborhood size is possible and may provide proper results, but ultimately the process is plagued by a disconnect between the designed rules and the desired output. It would not be easy for someone without a background in programming or formal logic to quickly or easily design a good, working cellular automata scheme. Therefore, cellular automata, though usable in some circumstances, is not suited for our applications.

Generative grammars are more advanced than automata. Rather than starting with a space-first approach which considers the topology and evolution of the result over time, generative grammars focus on providing complexity at the level of player action. Most grammars used for dungeon generation start with a set of possible player actions and use a generator to build an action graph for the dungeon. From there, a renderer must take the generated grammar and create dungeon topology and features. While this approach has much more flexibility and allows designers to quickly and easily specify the actions taken in a dungeon, it suffers by abstracting out the renderer and not giving much consideration. In general, the problem of creating a rendering of a graph without violating basic topological rules is NP-complete in the best case. Ensuring that no edges cross is a particularly difficult problem that has no easy or fast solution. Although generative grammars offer much more than automata, they are still not suitable for incorporation in interactive media, which may require that dungeons be built while the user is interacting. Moreover, they do not provide a solution to the problem of creating geometry with intent, only of creating a series of actions with intent.

Genetic algorithms have held much promise, and are robust enough to work in a variety of circumstances. Ashlock *et. al* demonstrated uses of genetic algorithms with respect to dungeon generation, and have shown that they provide interesting, varied topologies that push conformity toward specific features. There are chromosome encodings that specify the exact layout of the dungeon, but the majority of other representations use an indirect approach which encodes the dungeon as a number of features rather than a direct representation. This is a relatively minor setback in the grand scheme of things, but given that genetic algorithms also require a significant amount of time to run and create a new dungeon from scratch, it’s likely that the running time required to use genetic algorithms will be prohibitive in interactive media. For offline generation, they have an improved status, but cannot cover all of the use cases we require.

Finally, constraint-based algorithms approach the generation problem as a series of constraints. Fundamentally, constraint-based approaches are similar to inverse kinematics (IK), which find solutions under a series of constraints. Given a criteria for suitable outputs, find a solution which satisfies hard constraints (eg. limbs of an armature must not intersect) and soft ones (eg. minimizing the angles of each armature joint). We have chosen to follow this approach to generation because it provides several advantages over other methods. The first and most important is that it allows the fast, easy, and intuitive specification of the results desired. This enables generation to be handled by a non-expert, allowing for a more even distribution of work. Secondly, constraint-based approaches are relatively feature-agnostic, and allow for both large-scale and small-scale variety and specification in a wide variety of environments. Finally, the geometric interpretation of a dungeon is tightly coupled to the input criteria, which makes rendering part of the same process as generating.

Other results have been achieved using a variety of other techniques. In particular, hybrid approaches that combine a core constraint-based approach with genetic post-generation have achieved some very nice results, but suffer again from the runtime of genetic algorithms, their high complexity, and low control. Occupancy-regulated extensions to generation designs a dungeon from a library of level chunks, creating the dungeon from sections specifically designed by artists. Real-world data has also been used in generative approaches, though these are limited in scope to primarily real-world generation. For more robust and flexible generation, using real-world data is not an option.

Some results by Roden and Parberry have shown that constraint-based approaches are robust and applicable to two and three dimensions, as well as showing some results of generation using relatively simple criteria. Their dungeons are best suited for what Linden, Lopes, and Bidarra call ‘underground’ dungeons, in that they require primarily local coordination of features and no great amount of cohesion. We aim to expand upon the technique and show more results of three dimensional dungeon generation using a constraint-based approach.

Goals

Our goal is to provide a means for efficient, interesting dungeon generation for games and simulations, in a manner which is highly customizable and suited for the needs of game designers. Our ideal dungeon generator would capture the maximum amount of user input while simultaneously minimizing the effort necessary for the user to generate a dungeon from said input. We seek to provide dungeons that are entertaining, customizable, and easy to generate. These broad constraints are testable through the following metric approaches:

- Constraint satisfaction: A dungeon must be generated to satisfy all hard constraints and follow all soft constraints given by a user. In the event of failure, the generator should fail gracefully, and provide output that helps the designer diagnose and solve the problem.
- Customizability: The generator must provide a means by which the user can generate a variety of different dungeon types with little to no change to the code for the generator itself. We aim to provide this by offering a wide variety of constraint parameters, the customization of which can capture a large swathe of desirable dungeon arrangements.
- Dungeon Complexity: The dungeon must generate interesting, complex features that are not explicitly specified by user input. Our main approach for doing this is through the addition of "fake hallways", which are simply pathways not leading to any part of the initial timeline.
- Fast Generation: Dungeons must be generated and rendered fast enough to allow for rapid testing and development of different dungeon types.

Place in the Design Pipeline

A principal part of the design of our algorithm is determining exactly what the scope of our work is going to be. To do this, we describe here the exact portion of the graphics pipeline we expect to replace, as well as which parts we intend to leave unmodified.

The traditional Design Pipeline for a world containing dungeons within a game or simulation is as follows:

1. The designer brings several concepts for a dungeon into the process, including thematic elements, structural ideas, and general broad concepts of the dungeons appearance and progression.

2. The designer works (with assistance from other parts of the project team) to construct a timeline or progression of the game that is to be executed within the dungeon.
3. The designer begins building and modeling the dungeon based on the aforementioned criteria, resulting in a final product consisting of a general dungeon outline, without significant detail or reproducibility.
4. Artists and other designers modify this framework to construct a final dungeon for implementation in the game or simulation.

Our goal is to replace step (3) above with an automated dungeon generator, which will be able to take the broad concepts from step (1) and the timeline from step (2) in some standard format, and use both to automatically construct an interesting, semi-detailed dungeon geometry from a random seed.

This modified procedure has a wide variety of benefits. It allows for a faster process by reducing the time the designer must spend designing, allows for repeated generation of dungeons quickly to allow for testing and analysis of various dungeon concepts, allows for easier integration with artists and other designers by providing a skeletal dungeon framework, and allows for fulfillment of arbitrary constraint changes without a manual redesign of the dungeon from scratch.

Our approach

Using a constraint-based model, we have divided our constraints into the broad categories of ‘hard’ and ‘soft’ constraints. Hard constraints are those rules which cannot be violated at any time during generation and lead to unusable or illegal dungeons. Some examples of hard constraints are:

- Geometry of the dungeon must not self-intersect
- Rooms and corridors must not be too close to each other
- A bounding box exists for the entire dungeon, which rooms and corridors must not be generated outside of
- Rooms must be of the requested size
- Vertical corridors can be legal or illegal

These are some of the constraints implemented in our generator, and are not violated. Other constraints are more of guiding principles that should be attempted as best as possible but are not as necessary for proper construction as hard constraints. Some of the soft constraints implemented are:

- The chance of generating stairways
- The chance of a hallway going in the same direction
- Hallway length (a minimum is provided, but no maximum)
- Which general direction hallways should progress in

Through a combination of hard and soft constraints, we have found that dungeons are relatively easy to express, and that their articulations are clear and specific. Soft constraints can be said to enforce the ‘niceness’ of a dungeon, while hard ones can be said to enforce their ‘sanity’.

Architecture

Our core representation of a dungeon consists of several rooms connected by hallways or passageways. We have built our system within an infinite 3d integer grid of tiles. Each spot in the grid can either have a tile or not, and each tile can be given various properties, including which room or hallway it is a part of.

Our dungeon's expansion and creation is

centered around the concept of "joints" within the dungeon. A joint is simply a tuple, consisting of a location and a cardinal direction. A joint is meant to represent the passageway from one cell of the dungeon to a neighboring cell, determined by the direction of the joint. Regardless of the specific algorithm by which it is done, our methodology for dungeon generation then consists of adding additional rooms or hallways from available joints (which in turn create more available joints), until such a time as the dungeon is "complete".

The Initial Algorithm

Our algorithm for maze generation went through one major upheaval. The algorithm we eventually settled on is described in the next section. The initial algorithm we tried was a far more broad, powerful algorithm, but turned out to be completely intractable to implement, even allowing for a very broad range of error from the specifications.

Initially, we intended to allow the user to specify a series of rooms and their locations. In addition to this, the user could specify a series of "adjacency sets", which are sets of joints wherein all joints in the same set must eventually connect to one-another, while all joints in differing sets must never connect to one another. Keep in mind that for two joints to "connect", the joints must eventually have hallways or rooms constructed between them in such a way as to allow the player to get from one joint to the next without exiting the maze. At this point, it is important to keep in mind that in addition to this main constraint on the structure of the maze, there are a wide variety of additional constraints included in the maze specifications that must also be fulfilled.

Our first approach was to randomly build from the initial joints as if the adjacency sets weren't a concern, and to just hope that the sets would be properly satisfied eventually. Perhaps obviously, this solution hardly ever worked, and it was rare for even the simplest problem (one set consisting of two doorways) to properly meet the given constraints.

The obvious opposite approach to the problem was to perform a brute-force search of some given bounding box around the input joints, branching out the maze in every possible way until a solution is reached that satisfies the initial constraints. This algorithm is also obviously impractical, leading to an exponential blowup in runtime for even small search spaces, as well as producing mazes which both appeared unnatural and would be either too easy or too difficult for an agent to navigate.

The next step in our solution was to form something of a combination between the two previous approaches. We would begin randomly branching dungeon paths according to the rules set forth by the dungeon parameters, but at each creation of a hallway or room, we would check to make sure that we had not violated the adjacency requirements, either by "blocking off" a connection between two joints of the same set, or by accidentally connecting two joints of different sets. If we had done either of these things, the algorithm would backtrack one step and choose a different option. This appeared to mitigate the exponential blowup, and allowed us to generate moderately large dungeons in a fairly short time.

Unfortunately, this approach remains intractable, and is actually NP-Complete (provably reducible to the "Vertex Cover" problem). Even this might not be enough to abandon the approach, however, and we were able to find several optimizations which reduce the search time significantly. Even with these optimizations, the program still takes several minutes to generate even 2d mazes of size 100x100, which proves unacceptable for use in real-world game scenarios, where the generation may have to be as fast as real-time. The dungeons generated also often have unnatural features, such as "spirals" where the generator desperately tried to get two hallways to meet whilst maintaining the dungeon generation criteria. For these reasons, we were eventually forced to abandon the aforementioned input, and rewrote our system as described below.

The Final Algorithm

The primary change to our system involved relaxing our constraints by removing the user's ability to choose room locations and (to some extent) room connectivity. We have replaced the input with a tree of rooms, constructed by the user. This removes several of the prime problems with the previous approach, namely the arbitrary connectivity, room location, and possibility of failure on branching. This allows us to greatly reduce the complexity of our algorithm.

Our new algorithm is the following: iterate over the input tree breadth-first, constructing the given room at each node, and building out a hallway for each edge. These two stages (building rooms and building hallways) both differ significantly from the previous model, as described here.

For room generation, the initial algorithm was actually far simpler; simply place the room where the user defined it. In the new algorithm, we have to place the room at the end of a given hallway, which involves searching for a spot, placing the room if possible, and either continuing or destroying the hallway if the room failed to be placed. This problem can be reduced to the following task:

*Given a rectangle of size (w,h), and a grid of size (n*w, n*h) where some tiles are marked "unplaceable", find all possible placement locations of the rectangle.*

This converts a linear-time placement of the old algorithm to a quadratic-time placement in the current algorithm (both relative to the volume of the room). We are able to reduce this to approximately $O(n_{1.5})$ through optimizing the search algorithm, which is good enough that it is not a significant bottleneck on the algorithm.

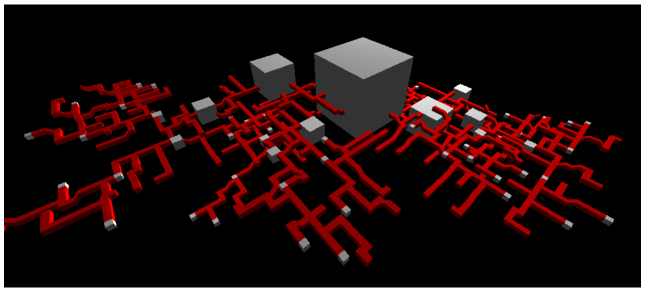
The hallway-generation algorithm is the main point of improvement of our improved algorithm over the initial approach. Rather than performing what amounts to an exhaustive search of all hallways to enforce arbitrary connectivity, we simply construct a hallway from some available joint of a room, and "dig" the hallway out until we are ready to place a room. If the hallway reaches a point where it is unable to expand or place a room, we delete the hallway and dig another. This is repeated breadth-first over the input tree until all rooms have been constructed.

Afterwards, we randomly generate a series of "fake" hallways from some portion of available joints in order to make the dungeon more maze-like and complex. This is another area where the initial approach was superior, as it would automatically generate these fake hallways simply by having failed passages within its search-space.

Overall, our new approach removes the exponential slowdown of the initial algorithm, while capturing all the benefits of the generation. All constraints are fulfilled during hallways and room placement, and the tree-input is able to properly capture most of the desired maze timelines.

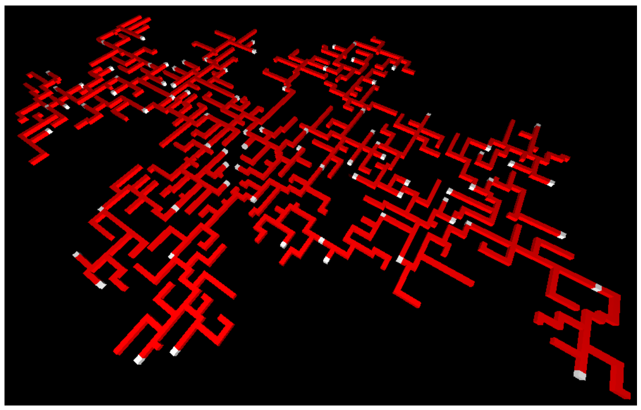
Algorithm Examples

In general, the larger blocks are rooms and the smaller thinner segments are corridors and they will often be color coded. Along with the geometry generated, we give the skeleton used, any significant heuristics, and the time required to generate in milliseconds. All benchmarks were done on a machine with 16 GiB of RAM on an Intel Core i7-2630-QM CPU @ 2 GHz.



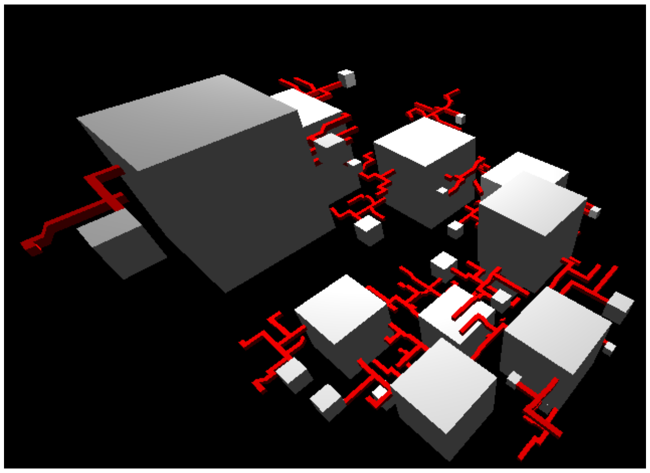
A traditional maze generated using our system. The skeleton for this is a binary tree, with the largest rooms at the root.

Generated in ~570 ms.



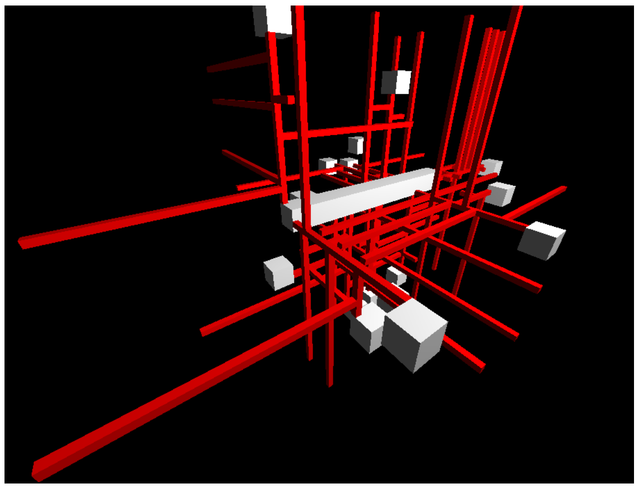
A two-dimensional maze generated by adding random rooms off of existing rooms. The skeleton is dynamically generated.

Generated in ~850 ms.



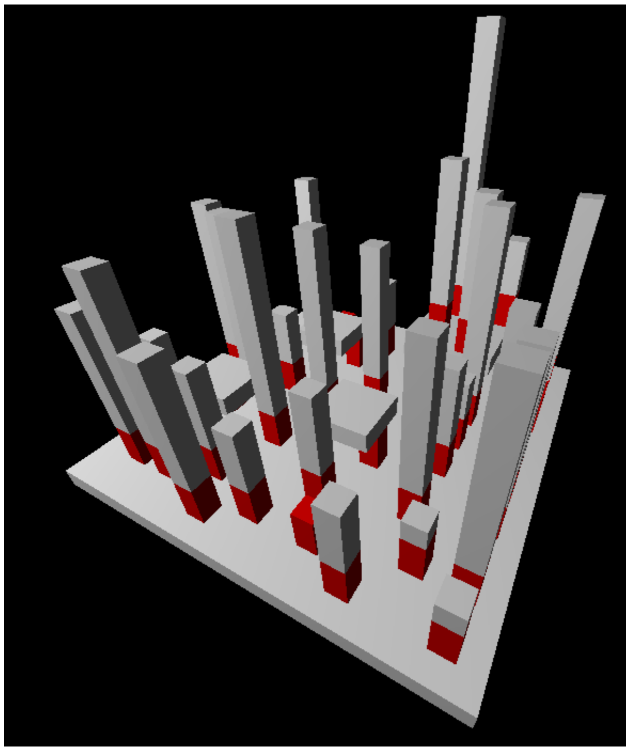
A dungeon generated using a skeleton with a backbone off of which the smaller rooms grow, shaped somewhat like a half-ladder.

Generated in ~1400 ms.



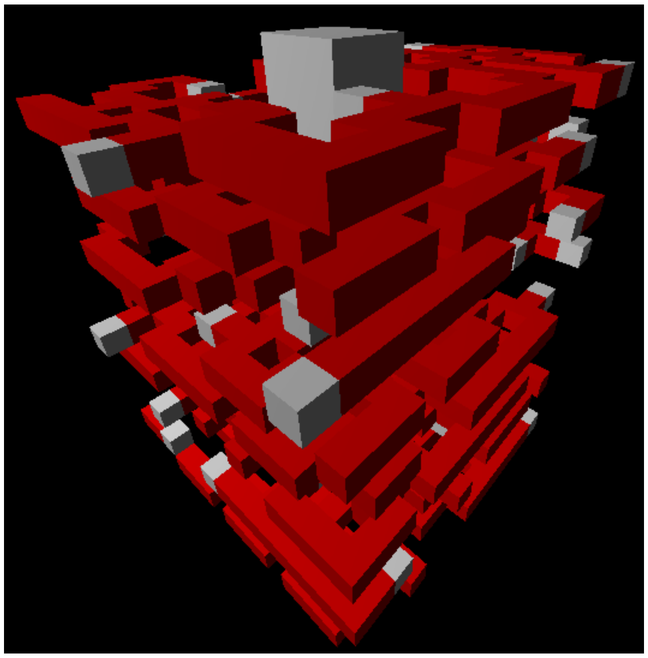
A mining base underground. The skeleton used was a network star, with a central hub and many nodes attached. This also features generation of corridors not attached to rooms, to add meaningful space to the dungeon.

Generated in ~1300 ms.



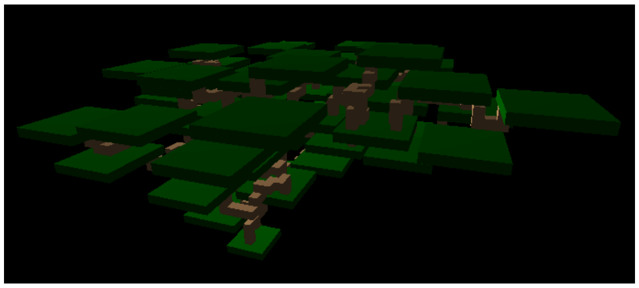
A ‘city’ generated by enforcing that rooms only grow upwards, and appear generally like skyscrapers except the base. The skeleton is also a star.

Generated in ~35 ms.



This dungeon also contains a central backbone off of which the other rooms grow. Down the center, the backbone rooms grow straight down, while offshoots grow out in planes from the center.

Generated in ~140 ms.



A tree generated with rooms and corridors growing from the base. The skeleton is again a binary tree, but exhibits nice behavior when placing upper branches.

Generated in ~210 ms.

Conclusion

Our generator was able to quickly generate interesting dungeons that fulfilled all of our specifications. We demonstrate that it is able to produce results across a variety of parameters, yielding dungeons that are entirely different from one another, but are all able to be generated from the same framework, with minimal modification to the initial parameters. Therefore, we conclude that the project was a success. Clearly, we faced a very significant obstacle in our redesign, and actually had to completely trash hundreds of lines of code when moving to the new algorithm, but ultimately, it resulted in a stronger end product.

Future Work

There are many avenues along which our work may be expanded. Particularly, many of our constraints are not usable or intuitive for non-expert users. Designing a library of possible constraints and allowing the user to specify them in a standard input syntax would alleviate this problem and expand the usefulness of our work to a broader audience. Additionally, encoding our constraints using a standard logical syntax would allow for us to feed dungeon constraints into classical constraint solvers which specialize in finding solutions under given constraints.

We did not use any more than simple probabilities for any of our soft constraints, but acknowledge that allowing for multiple types of distributions (in particular, Gaussian) would enable generation that can be fit to many more situations than currently supported. Finally, because only the intermediate step that builds geometry from the specification of a dungeon is given, adding a preprocessing step that builds trees from a set of specifications and a postprocessing step that adds backwards generation and decorates the geometry would prove useful. The possibilities are numerous, and many more techniques can be used to extend our work.

Two simple, but very fruitful modifications that we have begun work on but not included here are multiple generation and compound generation. Multiple generation is simply generating multiple dungeons and placing them inside the same map; a feature that takes no customization and is absolutely necessary for any real-world application. Compound generation is more complex, but much more rewarding.

Essentially, after generating a dungeon (for example, the "city" demo above), we can begin regenerating another dungeon within the rooms of the previous one (for example, putting a building within each room of the city). There are also other methods of compounding, such as adding sub-dungeons instead of just hallways for noise. This general methodology allows for a much more complex dungeon representation, and eliminates most of the limits of the basic algorithm.

References

R. Linden, R. Lopes, and R. Bidarra, “Procedural Generation of Dungeons,” in *IEEE Transactions on Computational Intelligence and AI in Games*

L. Johnson, G. N. Yannakakis, and J. Togelius, “Cellular automata for real-time generation of infinite cave levels,” in *PCG ‘10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*

R. van der Linder, R. Lopes, and R. Bidarra, “Designing procedurally generated levels,” in *Proceedings of the second workshop on Artificial Intelligence in the Game Design Process*

D. Ashlock, C. Lee, and C. McGuinness, “Search-based procedural generation of maze-like levels,” *IEEE Transactions on*

Computational Intelligence and AI in Games

T. Roden and I. Parberry, “From artistry to automation: a structured methodology for procedural content creation,” in *Proceedings of the 3rd International Conference on Entertainment Computing*