Fractal Coordinates for Incremental Procedural Content Generation

PETER MAWHORTER, Wellesley College, USA

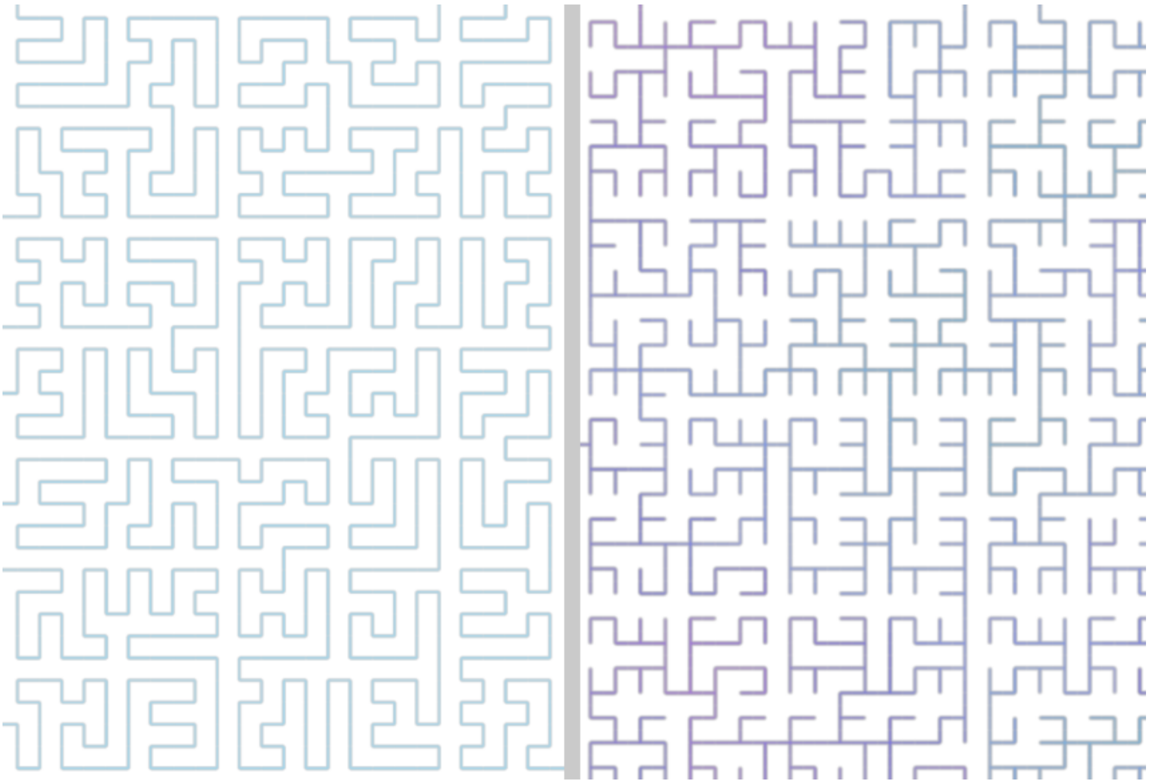
Fig. 1. Left: a section from an infinite grid-filling path. Right: a section from an infinite maze that includes cycles at multiple scales.

Incremental procedural content generation (IPCG) has been used in games such as Minecraft to provide a unique flavor of gameplay, but requires that parts of the world can be generated independently of one another in any order and will always fit together in the end. Noise functions such as simplex noise are a very popular building block for IPCG systems, because they support this property, but noise functions alone have their limitations, one of which being an inability to create non-local structures or continuity. To combat this, noise functions are usually applied at several scales to provide fractal continuity and recognizable features similar to real world geography's fractal complexity. This paper describes a system of fractal coordinates suitable for use with IPCG that generalize this idea of multiple layers of structure at different scales, and demonstrates how it can be used to achieve some interesting effects.

CCS Concepts: • **Applied computing** → **Media arts**; • **Computing methodologies** → Texturing.

Additional Key Words and Phrases: procedural content generation, fractals, coordinate systems, incremental procedural content generation

ACM Reference Format:
Peter Mawhorter. 2021. Fractal Coordinates for Incremental Procedural Content Generation. In *Foundations of Digital Games 2021, August 03–06, 2021, Montréal, Québec, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472538.3472576

1 INTRODUCTION

Popularized in games in large part thanks to the success of Minecraft [14], and earlier in computer graphics by Perlin's work on noise functions [16, 17] incremental procedural content generation (IPCG) involves generating content piece-by-piece, with the advantage (in the games context) that an enormous implicit world can exist and only the parts actually explored by players need to be generated in detail and/or retained in memory (e.g., storing all of a standard Minecraft world in memory would require dozens of petabytes [2]). At the same time, this places some interesting constraints on the generation algorithms used: they must be able to independently generate adjacent pieces of the world without information sharing (which would create a dependency leading to infinite regress), and they must typically do so quite quickly, so that the player may move freely in the world without loading times limiting movement.

This paper presents a system of fractal coordinates composed of a height value and Cartesian coordinates at that height which can be used to achieve a variety of different multi-scale coherence effects in IPCG systems. This power does come at the cost of substituting logarithmic-time not-quite-local generation for constant-time truly-local generation, but the ability to impose hierarchical constraints creates interesting new possibilities, and this paper includes two examples of abstract generative systems that make use of fractal coordinates. First, a simple fractal loop-based system that generates maze structures which regularly connect back to previously visited areas, and second, a more complex fractal system for generating a chaotic infinite path through a square grid which visits every grid point without branching or intersecting itself. At a general level, this fractal coordinate system should aid in the adaption of algorithms designed for fixed-size generation (e.g., subdivision algorithms, cellular automata, search-based algorithms, etc.) to an indefinite-sized incremental space, because of the way that it organizes indefinite space into finite-sized chunks with customizable sizes, while still allowing for some hierarchical constraints to be present between those chunks via fractal regress. Accordingly, this system has the potential to expand the design space of incremental PCG algorithms by offering a workaround to the restrictive constraints of true incremental generation.

The remainder of this paper explains the fractal coordinate system, compares it with existing IPCG techniques, provides examples of what can be achieved using it, and discusses connections to other research.

2 DISCRETE RECTANGULAR FRACTAL COORDINATE SYSTEMS

The core idea behind fractal coordinates is to take an infinite discrete grid (e.g., integer Cartesian coordinates) and identify each grid cell instead using a fractal coordinate system that identifies both grid cells at the base level and collections of grid cells that form grids at larger scales. For simplicity in this paper discussions are confined to two dimensions, but the same technique can be applied straightforwardly to three or more dimensions. It's also worth noting that the coordinate systems presented here are discrete rather than continuous, in part because most digital PCG outputs will eventually be discretized as pixels or samples, and so dealing with a discrete space is inevitable, although in theory continuous fractal coordinates could also be used with some adaptations to the scheme presented here.

To understand the idea of a multi-scale grid, imagine grouping every 2×2 region of an underlying integer Cartesian coordinate grid into its own cell, which now forms another grid. We will refer to these groupings of coordinates as **tiles** and the parts of each tile as **cells**; for the base grid, each tile includes only a single cell so the terms may be used interchangeably. To identify a tile *either* in the underlying grid or the larger-scale grid, one could add a third ”height” coordinate indicating which grid one was talking about to the standard two Cartesian coordinate values for a 2-dimensional grid. For example, using height/(x, y) notation, in the lower grid (height 0) the coordinates $0/(0,0)$, $0/(1,0)$, $0/(0,1)$ and $0/(1,1)$ form a 2×2 square, and that entire square as a single tile can also be specified by the coordinates $1/(0,0)$. For any coordinate value, we can identify minimum and maximum x and y coordinates at a specific lower height: For example, the tile $1/(1,0)$ spans x coordinates 2 to 3 and y coordinates 0 to 1 at height 0. This 2-scale grid system is not yet fractal coordinates, but it has many of the same properties: both individual smallest-scale grid cells and larger collections of those cells may be identified by separate coordinates, for example.

Extending this multi-scale grid notion to three or four layers of grid is straightforward: at each new layer, we establish a grouping of cells from the lower layer into tiles, and give each tile its own coordinate value. The insight of fractal coordinates is that this process can be continued indefinitely, allowing the specification of arbitrarily large tiles based on a fixed multiple of smaller tiles at each height. Of course, one could devise similar schemes for, e.g., a triangle/simplex grid instead of a rectangular grid, but for simplicity, we focus on rectangular grids here. Accordingly, the fractal coordinate systems that we use throughout this paper are each characterized by a scaling factor between different height layers (2 in the example above), which could conceivably follow an arbitrary pattern of scales instead of being constant, and in fact there could be arbitrary rectangular groupings happening at each scale, although again we will keep things simple by considering squares.

One small wrinkle in extending our grid system to an arbitrary number of layers involves the relative alignment of grids at different heights. For reasons that we will see shortly, it is undesirable to have an infinite number of grid edges aligned with each other across different scales, which would happen if we aligned the bottom-left corner of each $h/(0,0)$ tile at $0/(0,0)$ at all heights h. To avoid this, we can align the center of each $h/(0,0)$ tile with the $h-1/(0,0)$ origin tile below it, as shown in Fig. 2 (picking alternating centers at different heights if the scale factor is even). This alignment strategy provides a useful property: for any tile $0/(x,y)$ in the base grid, there is a unique **origin height** h_o at which the origin tile $h_o/(0,0)$ contains that base tile as a strict sub-region (the strict part means that for origin tiles themselves, their origin height is one larger than their actual height). Furthermore, the origin height of any base-level tile scales as the logarithm of the tile's maximum coordinate value (either x or y), with the base of the logarithm being the scale factor of the fractal coordinate system.

2.1 Fractal Trace Coordinates

In addition to the $h/(x,y)$ notation for specifying a fractal tile, there is an over-specific notation which is useful for identifying tiles. Although this alternate notation is completely optional and takes up more space than the equivalent basic coordinates, it has some useful properties which might make it appealing in some situations. These ”trace coordinates” consist of the origin height h_o followed by a variable-length list of numbers specifying a series of sub-tiles to access to find the specified tile. Each number in this trace is between 0 and $S^2 - 1$ where S is the scale factor for the fractal coordinate system, and identifies a sub-tile in bottom-to-top, left-to-right order, so that given within=tile coordinates (t_x, t_y), the trace number $n_t = St_y + t_x$. As a concrete example, the tile $1/(-1,-1)$ in the fractal coordinate system with scale 4 has trace coordinates $2/[5]$: it can be found by going up to the origin tile $2/(0,0)$ and then descending into the sub-tile at $(t_x, t_y) = (1,1)$ within that tile. Similarly, the tile $0/(2,3)$ in scale 4 would have trace coordinates $2/[14,3]$, whereas the tile $0/(2,2)$ has trace coordinates $1/[15]$.

Trace coordinates are useful because it is very easy to identify the parent (and all ancestors) of a given tile: simply remove the last element of the trace portion of the coordinates, or if there's only one element in the trace, increase

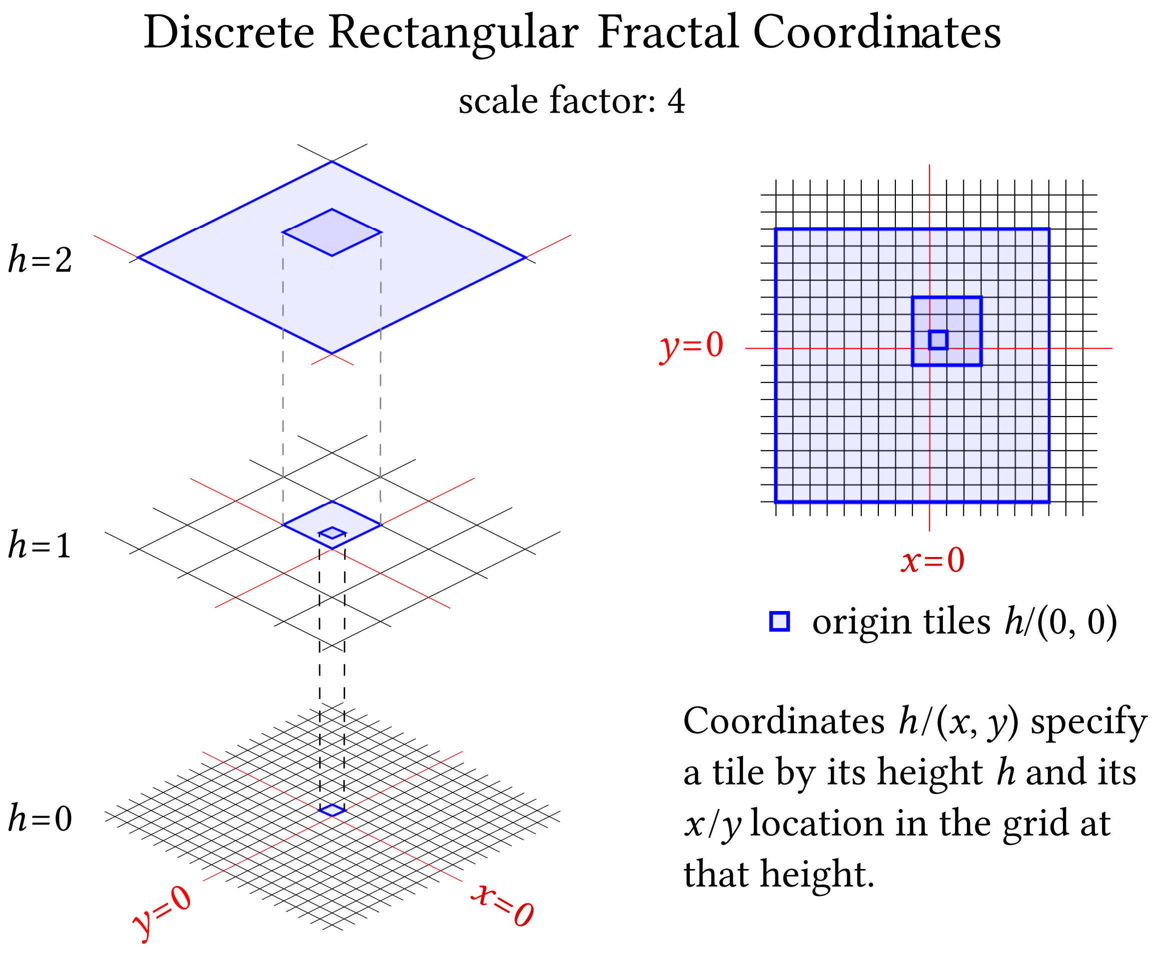
Discrete Rectangular Fractal Coordinates
scale factor: 4
h=2
h=1
h=0
y=0
x=0
y=0
x=0
origin tiles $h/(0, 0)$
Coordinates $h/(x, y)$ specify a tile by its height h and its x/y location in the grid at that height.

Fig. 2. A discrete rectangular fractal coordinate system with scale factor 4.

the origin height by 1 and then figure out the new origin trace location. Similarly, trace coordinates make it easy to identify all children of a given tile: they're the tiles (S^2 in total) with trace coordinates identical to the parent except for a single extra entry at the end of the trace. When processing recursive constraints, trace coordinates also make it very clear which sequence of parent tiles' constraints might need to be considered at this level. Of course, trace coordinates have drawbacks too: they require a number of trace entries that scales with the logarithm of a tile's distance from the origin, and they make it more difficult to figure out the coordinates of neighboring tiles at the same height. Note that in theory, a nonstandard trace coordinate could use a height higher than a tile's h_o origin height as the height value and add additional trace entries that specify origin tiles at the beginning of a trace.

Although the definition of a fractal coordinate system given here is complete, it is not immediately apparent what the usefulness of such coordinate systems is for IPCG. The next section includes two concrete examples of how such coordinate systems can be used to create interesting incrementally-generatable domains, and the related work section makes further connections and suggestions for possible applications.

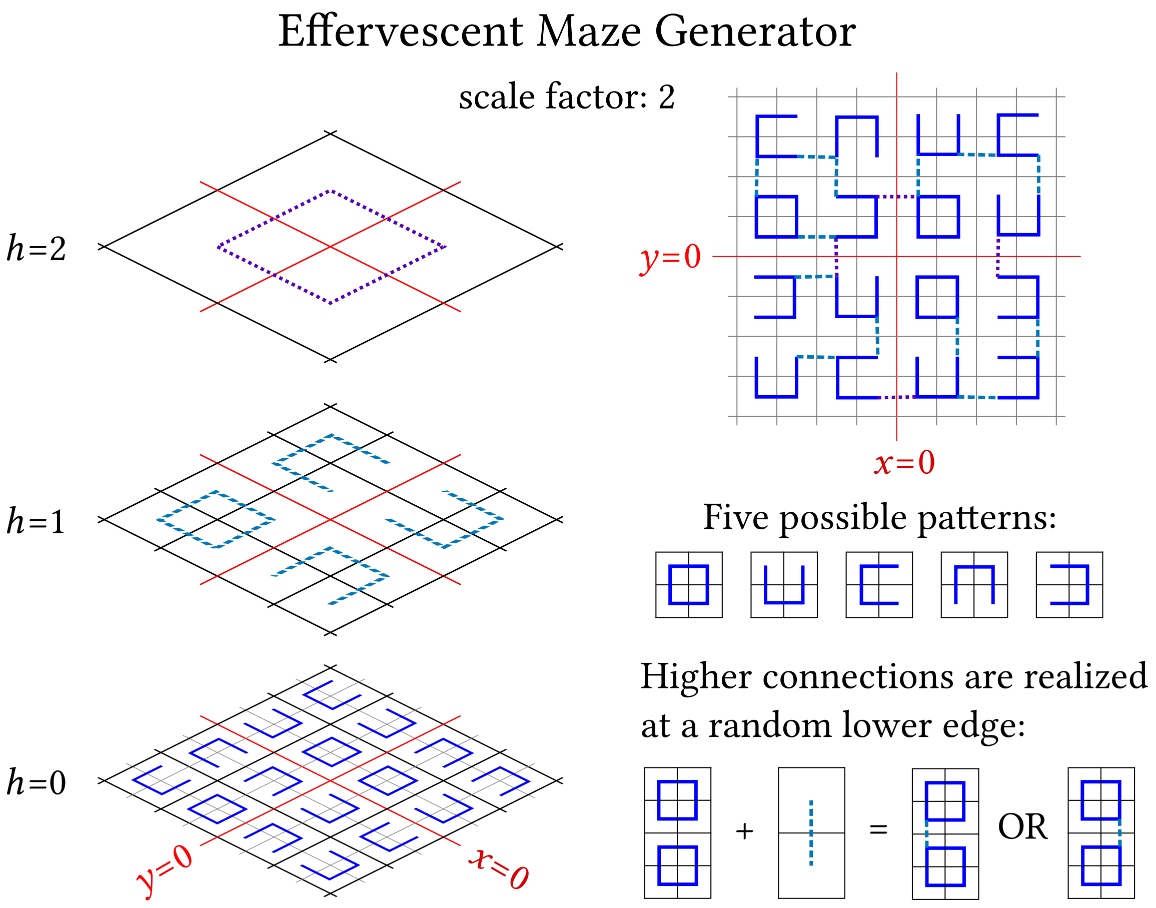
Effervescent Maze Generator
scale factor: 2
h=2
h=1
h=0
y=0
x=0
Five possible patterns:
Higher connections are realized at a random lower edge:
+
=
OR

Fig. 3. A fractal maze created using a 2×2 fractal coordinate system. Each 2×2 region is fully connected in a U or O shape (or rotated U). The left side of the diagram shows how multiple layers of random connection patterns create a maze (result at top right). As the diagram states, connections between tiles at a higher height are realized at random locations in the base grid.

3 EXAMPLE APPLICATIONS

Figure 3 shows one example application of fractal coordinates, and it's actually a relatively simple one. Before diving into the details, some general notes about PCG using fractal coordinates:

- Because fractal coordinates identify arbitrarily-sized tiles, which are made up of smaller cells that are also tiles (except at height 0), using some kind of grid-based generation technique is reasonable.
- Since tiles at all height values occupy the same space, whatever is generated at each level needs to synch up with things above and below it. This can get tricky, but it's also how the system gets its power to enforce complex constraints.
- Because every cell in the base grid is part of an origin cell at some height, the following procedure can be used to pass information between tiles at different scales without infinite regression:

(1) Without any information input, start by generating the height-zero origin tile, which has no origin tile below it.

(2) For higher origin tiles, use information from the origin tile below to constrain the results of the parent.
(3) When generating non-origin tiles, first generate their parent tile, and use information from the parent tile to constrain the results in the child tile.

These three rules give us a recursive means of generating information for any tile that can be related to information for tiles above or below it, and the total number of tiles whose generation is required (i.e., all parents until one is an origin tile, and then all children at the origin down to height 0) will scale as the logarithm of the distance from the origin.

- In many ways, it's convenient to think of the generation procedures for a fractal coordinate space as a recursive function, and to think about properties you want to guarantee as being assured via induction. Based on the rules above, the height-0 origin tile (at $0/(0, 0)$) is the base case, while generation for all other tiles recurses to a case for either a parent tile or a child tile.
- The time/memory tradeoffs involved should in most cases make it worthwhile to use memoization to cache generated tile information, and the algorithms presented here all do so. As with most incremental PCG systems, we rely on limitations on the player's speed of exploration to manage cache sizes for generated information.

3.1 *Effervescent*: Fractal/Cyclic Indefinite Mazes

The *Effervescent* system which generated the maze shown in Fig. 3 treats each cell in each tile as a cell in a maze which may or may not be connected in any of the four cardinal directions. A demo can be found online at: (the code is open-source but is *not* organized into a library for convenient use). The demo displays paths such as those shown in the teaser figure, which it generates on-the-fly, and it allows automatic or manual panning and zooming to explore the generated pattern. The generation system ensures that any cell which has an outgoing direction in one direction will have a corresponding incoming connection from that neighbor in the opposite direction, although the data structure does support non-reciprocated connections. Accordingly, for each cell in a tile, the information generated is four boolean values for whether that tile connects to the neighbor in each cardinal direction. *Effervescent* uses the following rules:

(1) For all tiles, we start with fully-connected O shape, where each cell connects with two neighbors to form a ring (actually a square) out of the tile (each tile is 2×2).
(2) With some random probability per-tile, we cut one edge of that O shape to form a U shape in some random rotation. Note that cutting one edge retains traversibility.
(3) For each connection in the tile, the edge that the connection spans is made up of 2^h edges in the base-level grid. We pick just one of those edges at random to actually realize the connection, and store that choice with the tile info.

Note that in this setup, there are no constraints which have to be respected between parent and child tiles, because child tiles only determine connectivity for their internal edges, and the internal edges of parent tiles are always external edges for their children. Note also that this is why our coordinate system does not simply use the base grid's origin (0, 0) as the southwest corner of the origin cell at every height, but instead centers origin cells within each other: if we had aligned all origins to share a southwest cell at (0, 0), the edge from (0, 0) to (0, -1) would not be an internal edge at *any* any height.

Given these rules, to figure out whether an individual edge in the base grid is connected or not, we first generate the unique tile for which that edge is (part of) an *internal* edge. In the worst case, the height of that tile will be the $O(\log h_o)$, where h_o is the larger origin height of the cells on either side of the edge we're interested in. Having identified the tile where that edge is internal, we look up (or generate) tile info for that tile based on the rules above and check if that internal edge is connected, and if so whether that connection is realized through the edge we're interested in.

With a procedure for finding out whether a given base-grid edge is connected, to draw the maze we simply check our viewport and ask whether each edge in that viewport is connected, drawing those that are (caching keeps this process reasonable). Of course, as the area of interest gets further and further from the origin, we end up having to generate higher and higher origin tiles to include the area we're looking at, and the total number of ancestors we need to generate increases as the logarithm of our distance from the origin. However, given this caveat, the algorithm is fully incremental: we can ask it to generate edge at any point in the 2D grid, and it can do so independently of generating edges elsewhere (except in that it generates some ancestor tiles for some other edges). The properties of the maze we generate include:

- Every cell in the grid is reachable from any other cell. This is true by induction: at the tile level, we only generate O and U shapes, so there's always a path between any two sub-tiles. Those sub-tiles have the same property, so we'll never get stuck inside of them.
- By controlling the probability of creating U or O shapes (and conditioning that on tile height) we can have some control over how often we expect exploration processes to circle back and re-encounter a previously visited location. Distinct sub-areas are cut off from one another by narrow single connections, and this happens fractally at multiple scales.

Although *Effervescent* takes advantage of fractal coordinates to both guarantee complete reachability and to create its fractal structure that includes both cycles and dense/sparse connectivity contrasts, it notably does not have any dependencies between parent and child layers. Generation at each layer is a simple matter of randomly picking an O or U shape, and of determining exactly where high-level edge connections are realized, and both processes can be competed in isolation without any additional constraints. In contrast, the next example application we will explore makes heavy use of such constraints, but achieves much more elusive special properties as a result.

3.2 *Labyrinfinite*: An Indefinite Labyrinth

To demonstrate the power of recursive constraints in a fractal coordinate system, the *Labyrinfinite* system generates a Hamiltonian path through the infinite graph that connects all points on a 2D integer grid with their orthogonal neighbors to form an infinite square grid. In other words: it generates an infinite path that twists and turns, but never branches or crosses itself (an infinite labyrinth), which eventually visits every point on an infinite square grid. It also generates this path incrementally: you can ask for any part of the path and it will give it to you without generating the rest of the path (although technically, it does generate some necessary constraints on the rest of the path in the process). In fact, these paths also form a family of space-filling curves [5, 18] which incorporate a natural level of visual variety.

Figure 4 shows the result, and includes a diagram showing how information in higher and lower tiles is connected. A demo can be viewed online at: (again, the code is open-source but not packaged/polished for distribution). The demo displays paths such as those shown in the teaser figure, which it generates on-the-fly, and it allows automatic or manual panning and zooming to explore the generated pattern. The same dependency strategy mentioned at the start of this section is at play here: origin tiles generate based on the tiles below them, while all other tiles generate based on their parent tiles.

How does *Labyrinfinite* work? In terms of basic principles, it sets up the following fractal structure:

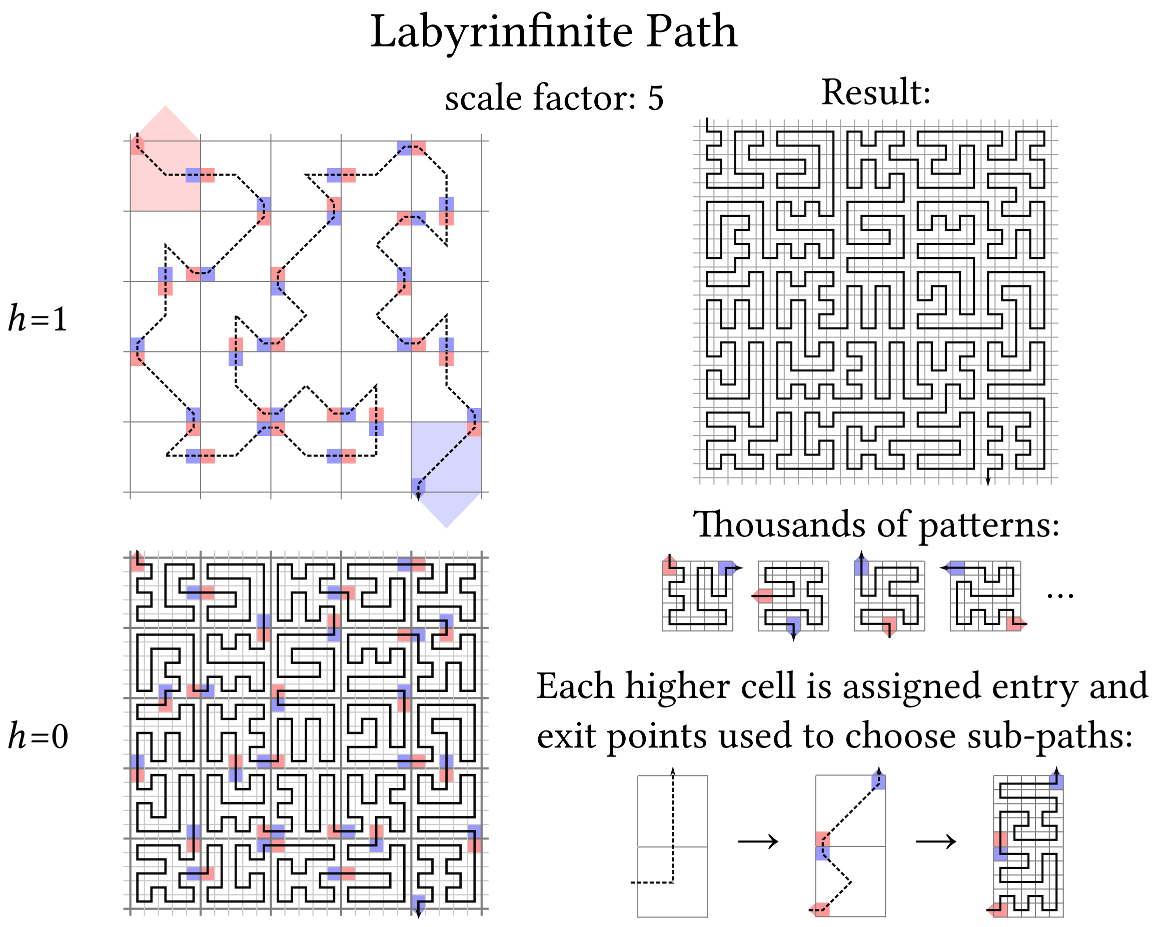
Labyrinfinite Path
scale factor: 5
Result:
h=1
h=0
Thousands of patterns:
...
Each higher cell is assigned entry and exit points used to choose sub-paths:

Fig. 4. A fractal path created using a 5 × 5 fractal coordinate system. The line visits every point in each 5 × 5 region before moving on to the next, and the same is true fractally (e.g., for 25 × 25 regions). The result (top right) is actually constrained by multiple layers (two shown at left).

(1) Each 5 × 5 region is traversed by a path that enters at one edge and exits at another, randomly chosen from all possible 5 × 5 paths which enter and exit at those coordinates (taking advantage of symmetries, there are less than two thousand such paths, and we simply store all of them in a cache indexable by entry/exit points).
(2) Each of those 5 × 5 regions is actually a single cell in a larger path, and the sides on which the smaller path enters and exits its region match up with the connectivity of that cell in the larger path.
(3) At the smaller level, the precise entrance and exit positions at which each traversal starts and ends are coordinated with their neighbors, so that neighbors actually do connect with each other.
(4) These three properties are recursive, so that there is an infinite succession of even-larger layers that also enforce them.

Some important points to consider:

- Based on 5×5 tiles, for a complete traversal to be possible, the path must enter at positions 1, 3, or 5 along an edge: entering a positions 2 or 4 makes it impossible to traverse every cell in the tile.
- Positions 1 and 5 of an edge overlap with positions 1 and/or 5 of two other edges (e.g., position 1 on the north edge counting from the west is the same cell as position 1 on the west edge counting from the north). Since you cannot both enter and exit from the same cell, this means that we can't independently choose entry/exit points for each edge like *Effervescent* did, although it's notable that if we enter at position 3, no matter which edge we exit from (as long as it's a different edge than the one we entered on) there are solutions which exit at every possible position on that edge.

How does *Labyrinfinite* actually enforce the recursive properties detailed above? It starts by using the same system of recursion described above: each tile waits to be generated until its parent has been generated, and uses information in the parent to make sure the rules above will be respected. Of course, this can't apply to all tiles, as there would be no base case, so origin tiles are generated differently: They rely on information from the origin tile below them (which is generated first) instead of relying on information from their parent. The base case is the origin tile at height 0, which is generated without any reliance on outside information. The three generation cases break down like this:

(1) For the origin tile at height 0, we choose random entrance and exit edges and random entrance and exit positions on those edges. Then we pick a pattern from our pattern library that has entrances/exits on the appropriate edges at the appropriate positions, and that becomes the traversal pattern for the height-0 origin tile.
(2) For origin tiles above height 0, they will only be generated once the origin tile below them has been. Therefore, they know both the entry/exit edges and entry/exit positions on those edges for the origin tile that lies at their center. These higher-level origin tiles first filter the pattern library to include only patterns whose central tile is entered and exited from the edges dictated by the origin tile below them, so the random choices made there become constraints at the level above. There are no entry/exit choices at the lower level that have zero corresponding patterns in the pattern library, since the pattern library contains all possible traversals of a 5×5 grid. After filtering the pattern library, the higher-level origin tile picks a pattern at random, thus determining both the edges and edge positions at which the path must enter/exit that tile. Two problems remain: constraints imposed by entry/exit positions of sub-tiles yet-to-be-generated within this tile, and constraints on entry/exit positions of lower-level tiles at the borders of this tile. To ensure that when the path leaves this tile it will enter the next tile at the same location (regardless of the height of this tile), we pick random edge positions (1, 3 or 5) to impose as a constraint on both of our child tiles that are at our edges. We do this using a seed based on the position and span of that edge in the base grid, so that the tile on the other side of that edge will be able to follow the same process and pick a matching random entry/exit position to constrain its sub-tile. To ensure that there are no long-term dependency issues within this tile, where entry/exit position (not edge) constraints conflict with one another, we constrain the entry/exit positions for the other end of the cells where edge constraints exist so far (our entrance and exit cells and our central cell) to be position 3, which as we noted above then places no further constraints on the entry/exit position paired with it. Since entry/exit *edges* are determined for each cell (i.e., sub-tile) by the pattern we picked, we proceed to pick specific entry/exit *positions* for each of our cells, incrementally expanding them from the three cells which already have constraints placed on them: our entry cell, our exit cell, and our central cell. At each step, we simply find a cell which has one edge that's already decided but another edge that's not, and pick and random entry/exit position for that other edge, among the positions which are compatible with the existing constraint. Because of the position-3 restrictions

we made before this filling-in process, the filing-in process will never meet a contradiction, and does not need to back-track its decisions. At this point, we finally have a full specification for the given origin cell: a traversal pattern that dictates entry/exit *edges* for each cell, and beyond that, specific entry/exit *positions* for each cell. Furthermore, the entry/exit positions chosen at random by the origin cell below this one have been respected as constraints at this level.

(3) Finally, the procedure for generating any non-origin tile is guaranteed to be able to access information from its parent tile, because that parent tile is always generated before it. Based on the algorithm for generating origin tiles, the information available from parent tiles is:
   (a) The entrance and exit *edges* for each child tile (i.e., cell in the parent).
   (b) The entrance and exit *positions* for each child tile.

   All the child tile has to do is filter the pattern library to pick out a pattern which respects the constraints provided by the tile above it; that pattern determines the entry/exit edges for each of its children. Like the origin tiles, it sets the entry/exit positions for the tiles where the path enters and exits it based on seeded random decisions which will be matched by its neighbors, and then sets the entry/exit positions for the other end of those two tiles to be 3. Lastly, it fills in entry/exit positions for all of the rest of its tiles one at a time using random selection, just like the process for the origin tile. This time, there are no pre-determined constraints on the central tile, and the same guarantee can be made that greedy random selection of positions will work as long as we obey constraints within one cell at a time. Now the non-origin tile has selected entrance and exit edges and positions for each of its child tiles, and thus can fulfil its inductive bargain to provide that information to its children in turn.

As detailed in the algorithm description above, except for the height-0 origin tile, each tile when generated specifies the entrance and exit edges and positions of all of its child tiles. Since that information is everything that the generation algorithm needs from a parent or child tile, the entire recursive system stands up, anchored by the simple random choices made in the base case for the height-0 origin tile. Furthermore, the selection of entry/exit positions in cells at tile edges can proceed without depending on a sibling-tile being generated first (which would be an infinite regress), because seeding based on the edge's absolute position and size on the base grid allows both siblings to make complementary decision about which entry/exit position should be used. The end result is that given a number of generated tiles that scales as the logarithm base 5 of the distance from the origin, a single tile anywhere can be generated incrementally without needing to generate its siblings or the full path from it to the origin, and yet we also have a recursive guarantee that the entire path both travels everywhere and never branches or crosses itself. These two properties: incremental generation and complex global dependence, are the true measure of the power of fractal coordinates. Although they do cheat a little compared to strictly incremental algorithms by pre-generating some amount of abstract higher-level context in order to generate a specific patch of the world, they only require logarithmic time/memory use as distance from the origin increases, and they can achieve outcomes that might seem to only be possible with truly detailed global coordination.

4 RELATION TO EXISTING IPCG TECHNIQUES

As hinted at in the end of the previous section, fractal coordinate generation techniques which use the depend-on-parent-or-origin-child recursive structure outlined above require time and/or memory expenditures that grow as one gets farther from the origin, with formulae that are in $O(\log_S d)$ where d is distance from the origin and S is the scale factor for the coordinate system. In contrast a strictly incremental system would generate each piece of content completely independently of any other piece, and the time- and memory-per-piece would be constant for all pieces irrespective of position. However, logarithms grow slowly, and even in "infinite" worlds, there are usually technical constraints on how far a player can travel since coordinate systems run into issues when the limits of integer and/or floating point math are reached (e.g., the "Far Lands" in Minecraft [7]). Concretely, for example, the logarithm base 5 of 2^{32} is only 13.782, so at the point where a generation system would need to worry about integer overflow, a size-5 fractal coordinate system is still only generating a few dozen tiles to come up with concrete results for height-0 tiles.

Perlin's original and simplex noise functions are a good point of comparison [16, 17] (see [9] for a clear explanation of simplex noise). These algorithms are hugely popular in IPCG systems because they are incremental and generate noise that has coherence on multiple scales. Similar to fractal coordinates, typical Perlin noise systems use multiple "octaves" of noise at different scales, and combine them by simple addition or other operations to get a more complex manifold for, e.g., terrain generation. In fact, from one angle Perlin noise might be viewed as an application of fractal coordinates, because multi-octave noise generates values at different scales and combines them. The uses that this paper advocates however diverge from what Perlin noise does by using discrete grids instead of continuous mathematical functions, and by illustrating how hard constraints can be passed back and forth between different fractal layers, instead of relying on operations that preserve continuity and generating layers independently.

In general, the recursive scheme for satisfying fractal constraints explored here could be applied in a number of ways that allow much more complex structures to emerge than in a strictly incremental system where generation of neighboring areas is truly independent. Of course, many IPCG systems combine something like Perlin noise with local generative processes that use more traditional (often additive) techniques to build coherent structures with complex organization (e.g., dungeon generation in Minecraft). This strategy has several drawbacks that fractal coordinates could help address:

- Although individual complex structures like dungeons or villages can be generated, the generation processes for each are unaware of each other and cannot share nay information, to keep the whole system incremental.
- As a result, these landmarks must either be strictly separated spatially, or allowed to interfere with each other in ways that can produce artifacts (e.g., Minecraft embraces this with things like dungeons allowed to intersect features like ravines, which leads to interesting structures but also issues like floating blocks that might be undesirable).
- The fixed-area generation systems for complex structures are not themselves incremental, and thus these structures have limited size and complexity, since they need to generate as the player approaches them, but if that generation process takes too much time it will cause lag for the player. Alternatively, a fixed number of such structures could be generated when the world is generated to front-load the required generation time and also guarantee the player's ability to find them (see e.g., [1]). However, this approach abandons incrementality, and therefore limits world scale (which could be relevant in, e.g., a massively-multiplayer game like Noctis [6] or Elite: Dangerous [4]).

Fractal coordinates could address the issues above by providing a means for coordinated generation between nearby structures (it's a partial and principled retreat from strict incrementality). Furthermore, the information available in fractal coordinate tiles makes it possible to distribute generated content (e.g., strongholds in Minecraft) with strict regional constraints on frequency (e.g., roll for a small # within a larger region and distribute them among child tiles recursively), while at the same time allowing a game system to efficiently locate the nearest such content to the player in order to provide hints or direct AI actions, while retaining the core benefits of incremental generation.

A concrete example of these difficulties comes up in [10], in which cellular automata are used to generate caves, and separate sections of an infinite grid are generated one-by-one to provide an indefinite gameplay experience. A limitation of this system is that in order to avoid harsh seams, and to ensure full traversibility, 50×50 tiles are generated in batches of 5, including a central tile plus its four neighboring tiles. After running a specific number of automaton steps on all 5 tiles, connectivity is checked from the central tile to its neighbors, and open paths are added if this check fails, after which extra cellular automaton iterations are performed to smooth things out. The difficulty is that these extra steps may have to be repeated when one of the four adjacent tiles has its turn to be fully generated, finds a neighbor that it isn't connected to, and thus has to be re-smoothed. Since this re-smoothing cannot extend to the base tile (which would create an infinite regress), it may create some seam artifacts as the number of automaton steps of the base and adjacent tiles get out of sync. This fix-up approach may also result in ordering effects, where the final appearance of a tile depends on which adjacent tiles were generated in what order, and thus the terrain generated depends on how the player moves through the map; such ordering effects can cause additional seaming issues if the player moves on a path which moves away from an explored area and then reconnects with it. Fractal coordinates could help resolve these issues by establishing a larger multi-tile context in which to coordinate edge connectivity and smoothing. For example in a 4×4 fractal coordinate system, each tile could be provided with pre-determined edges by its parent tile, could run automata rules on the edge regions of its internal edges to be able to provide the same to its child tiles, and could then delegate full generation of the child tiles with the constraint that the pre-determined edges should not change. Any required connectivity rules could be satisfied by these pre-determined edge regions plus operations which only affect the interior of a tile, without the need to modify neighboring tiles. Although there is not space here to fully flesh out this approach, [10] does provide a good example of some of the seam issues in incremental generation, and fractal coordinates offer one possible path towards dealing with them.

Fractal coordinates and even the specific recursive constraint propagation strategy outlined here are more of a general approach than a specific generation algorithm, and the goal of this manuscript is to help those interested in incremental PCG systems to understand their benefits and limitations. As the next section explores in detail, there are lots of potential connections to existing techniques that could be adapted to fractal coordinate spaces. The core insights offered here are the center-based origin determination for fractal tiles at successive heights which prevents infinite regress because every edge becomes internal to a specific tile at some height, and the recursive strategy for constraint propagation between tiles which allows all non-origin tiles to rely on their parent being already generated, while origin tiles rely on a central child being generated, with the height-0 origin as the base case.

5 RELATED WORK AND POTENTIAL CONNECTIONS

There are lots of research avenues that are related to this work, and even more with potential future synergy. The main inspiration for this generation strategy was taken from noise functions like Perlin noise [16] and Worley noise [25] (a survey can be found at [12]). Their incremental properties made possible games like Noctis [6] and Minecraft [14], which is also the genre to which the techniques in this paper are most applicable. The unique constraints of incremental world generation in particular make it a difficult setting in which to apply many existing PCG techniques, and it poses unique challenges for PCG systems based on search, evolutionary algorithms, and other AI or machine-learning strategies that for various reasons require the ability to compare concrete generation results (see e.g., [20, 21]). While such systems can be applied to individual world chunks, they themselves don't deal with the problem of information sharing detailed above which fractal coordinates helps tackle directly. However, non-incremental maze-generation systems like [24] and [13] are closely related to this work. [24] in particular includes the idea of stitching multiple graph segments together, and one could imagine applying the same minimum-spanning-tree technique over a fractal graph with potentially pleasing results; there might even be a way to modify it to work incrementally. Similarly, [13] also incorporate space-filling curves into their work; interestingly they go to great lengths to overcome the regularity that the use of a Hilbert curve introduces into their maps while recognizing that the connectivity it ensures is desirable; *Labyrinfinite*'s randomized space-filling curve would seem a useful substitute in their system. Macedo and Chaimowicz' work also incorporates cellular automata (CA) as did the previously mentioned work of Johnson, Yannakakis, and Togelius ([10]), hinting at a rich area of future work in incremental CA systems.

There are also threads of related work in mathematics (e.g., [5, 15]), and physics (e.g., [15]) although this work does not directly apply any recent ideas from those fields. In particular, the discrete fractal coordinates presented here are far less complex than the notion of fractal coordinates present in [15]. More connections likely remain to be made in these areas however by specialists who have overlapping expertise.

In terms of future work, there are some avenues of research that seem promising. In addition to adapting many existing algorithms that work with fixed-size grids to work on fractal grids (e.g., [23]) and investigating how constraint propagation between fractal layers could add interesting effects, the recently popular Wave Function Collapse (WFC) algorithm [8] seems like a prime target for adaptation to a fractal grid. As detailed in [11], the core of WFC is constraint satisfaction, and the recursive constraint chaining strategy proposed here for fractal grids offers a way to propagate constraints efficiently between tiles at different level of the grid. At the same time, one of the limitations of WFC is the scaling of the algorithm to large grids, along with in some cases the increasing chances and costs of forced restarts at larger grid sizes. Although problems with between-tile compatibility might be highly domain-specific which might limit the generality of a fractal approach, the use of fractal coordinates for WFC seems like a natural area for future work.

Another overlap is with space partition algorithms, whose use dates all the way back to Rogue in 1980 [22]. Good summaries of several algorithms can be found in [3, 19], including variable-size partition schemes like random binary space partitioning. Since fractal coordinates can be used as a kind of space partition, it would be interesting to investigate potential applications of variable-size fractal coordinate systems, although this would obviously complicate several key coordinate transformation functions (e.g., finding the parent of a tile).

6 CONCLUSION

The examples presented here demonstrate the interesting forms that the use of fractal coordinates can produce, but they barely scratch the surface of its potential applications. As mentioned in the previous section, there would seem to be fertile ground for combining fractal coordinates with at least a few different existing algorithms, and of course more concrete generative systems along the lines of those presented as examples here could push fractal coordinates further. The combination of an ability to provide global and multi-scale strong constraints while also being an incremental approach that doesn't place limits on world size is what stands out about the fractal coordinate approach, and the graph structures that are generated by the systems presented here could be used not only as level geometry but also for more abstract connections (e.g., trade routes or quest progression paths). A big downside is the complex design work of figuring out how to enforce one's desired constraints within the fractal coordinate structure, as at least the recursive scheme for constraint propagation presented here requires that dependencies line up in a specific way. For example, the gesture made earlier towards seam-constraints for cellular automata might lead to a solution to that issue, but there are non-trivial details to be hashed out there about how such seam constraints would function. Rather than a one-size-fits-all solution, fractal coordinates are a useful tool that still requires deep expertise to deploy effectively. However, the potential of fractal coordinates to make new kinds of constraints possible, and the ability to compute the resulting structures incrementally with only a logarithmic time/memory overhead has potential to create new possibilities in indefinite generated worlds such as that of Minecraft.

ACKNOWLEDGMENTS

This work was done on the traditional land of the Massachusett, Wampanoag, and Nipmuck peoples, and the author is grateful for their stewardship of that land in the past, in the present, and in the future, despite its present circumstances.

REFERENCES

[1] Anonymous, Ajc 1254, et al. 2021. Minecraft Wiki: Stronghold. https://minecraft.gamepedia.com/Stronghold.
[2] AntVenom. 2021. MAXIMUM FILE-SIZE for a FULL MINECRAFT WORLD?! https://www.youtube.com/watch?v=ZmfopT9Vupo.
[3] Tommy Bacher. 2018. Procedural Level Generation Algorithms. http://www.jthomasbacher.com/bacherJuniorISWriting.pdf. Independent Study Project Report.
[4] Frontier Developments Ltd. 2014. Elite: Dangerous. Frontier Developments Ltd.. PC Edition.
[5] HIROSHI Fukuda, MICHIO Shimizu, and GISAKU Nakamura. 2001. New Gosper space filling curves. In *Proceedings of the International Conference on Computer Graphics and Imaging (CGIM2001)*, Vol. 34. 38.
[6] Alessandro Ghignola. 2000. Noctis. Home Sweet Pixel. PC.
[7] GmMkr11260, Jonnay23, Tjb0607, et al. 2021. Minecraft Wiki: Java Edition Far Lands. https://minecraft.gamepedia.com/Java_Edition_Far_Lands.
[8] Maxim Gumin. 2016. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse. GitHub Repository.
[9] Stefan Gustavson. 2005. *Simplex noise demystified*. Technical Report. Linköping University, Linköping, Sweden. http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf
[10] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. 2010. Cellular Automata for Real-Time Generation of Infinite Cave Levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (Monterey, California) *(PCGames '10)*. Association for Computing Machinery, New York, NY, USA, Article 10, 4 pages. https://doi.org/10.1145/1814256.1814266
[11] Isaac Karth and Adam M Smith. 2017. WaveFunctionCollapse is Constraint Solving in the Wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. 1–10.
[12] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, David S Ebert, John P Lewis, Ken Perlin, and Matthias Zwicker. 2010. A survey of procedural noise functions. *Computer Graphics Forum* 29, 8 (2010), 2579–2600.
[13] Yuri PA Macedo and Luiz Chaimowicz. 2017. Improving procedural 2D map Generation based on multi-layered cellular automata and Hilbert curves. In *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 116–125.
[14] Mojang. 2009. Minecraft. Mojang. PC; full version released 2011.
[15] Laurent Nottale. 2011. *Scale Relativity And Fractal Space-Time: A New Approach to Unifying Relativity and Quantum Mechanics*. Imperial College Press, London, England.
[16] Ken Perlin. 1985. An image synthesizer. *ACM Siggraph Computer Graphics* 19, 3 (1985), 287–296.
[17] Ken Perlin. 2002. Noise Hardware. In *Real-time Shading Languages*. Ch. 2. https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf
[18] Hans Sagan. 2012. *Space-filling curves*. Springer Science & Business Media.
[19] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. 2016. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games*. Springer, 31–55.
[20] Nathan Sturtevant and Matheus Ota. 2018. Exhaustive and Semi-Exhaustive Procedural Content Generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 109–115.
[21] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural content generation via machine learning (PCGML). *IEEE Transactions on Games* 10, 3 (2018), 257–270.
[22] Michael Toy and Glenn Wichman. 1980. Rogue. Mastertronic Group Ltd.. PC.
[23] Breno MF Viana and Selan R dos Santos. 2019. A Survey of Procedural Dungeon Generation. In *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 29–38.
[24] Bartosz von Rymon Lipinski, Simon Seibt, Johannes Roth, and Dominik Abé. 2019. Level graph–incremental procedural generation of indoor levels using minimum spanning trees. In *2019 IEEE Conference on Games (CoG)*. IEEE, 1–7.
[25] Steven Worley. 1996. A cellular texture basis function. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 291–294.