Constrained Procedural Map Generation for Two-Dimensional Games

Gavin Scott
Computer Science Department
California Polytechnic State University
San Luis Obispo, CA
goscott@calpoly.edu

Daniel Toy
Computer Science Department
California Polytechnic State University
San Luis Obispo, CA
dltoy@calpoly.edu

ABSTRACT

We propose a system for the generation of levels for two-dimensional games that allows the designer to specify a flexible set of constraints and generation parameters. These allow for fine-tuned controlling of both the placement of impassable walls and temporarily impassible obstacles to slow the player's progress and increase player engagement. The system generates only one map that is guaranteed to meet the criteria rather than the traditional method of generating and testing many maps, which can lead to a prohibitively long generation process.

Keywords

Procedural content generation, 2D maps, obstacle placement, constructive PCG

1. INTRODUCTION

Procedural map generation is a popular area within Procedural Content Generation (PCG) because it can create a wide variety of maps. These benefits can help developers avoid having to hand-create each map and increase the replay value of a game since each map will be unique. However, developers must ensure that the PCG algorithm used is properly constrained in order to ensure that the output generates maps that are playable and enjoyable. A map is considered 'playable' if the objective of the game can be accomplished in the map. For example, a map created using PCG would be considered unplayable if the exit to the map was blocked off on all sides by walls and unreachable by the player. There are two ways in which to ensure that a map is playable. One traditional option is to continuously generate new maps and test them to see if they fit the constraints. This can be very time-consuming, and if a limit is placed on the generation time, the process might result in an imperfect map. The other strategy, called constructive-PCG, is to constrain the algorithm such that only valid solutions are ever generated. This is the concept we chose to employ.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

In this paper, we analyze multiple PCG algorithms for creating 2D maps. This includes the generation of walls in the map as well as the placement of obstacles throughout the map for the player to interact with. The algorithms we analyze in this paper are combinations of different techniques that are constrained to create a valid map every time, in a reasonable and predictable amount of time. Specifically, we analyze the use of variations on cellular automata, quadtrees, and digger algorithms for the generation of the maps' walls. We focused primarily on increasing the quality of the map by requiring more player strategy and minimizing unreachable space, while also trying to maintain an intuitive placement of the obstacles.

2. RELATED WORK

Cellular Automata (CA) [7] are systems that consist of a grid of cells, each with a distinct state. Each cell is controlled by a set of rules that determine the cell's state based on the states of its neighboring cells, allowing the states to change with each new generation of cells, or round of processing. John Conway's "Game of Life" [6] demonstrated that interesting emergent behavior could result from these independent automata, even when constrained by a very simple set of rules. CA has also been integrated into video-game level generation by using automata with two states representing whether or not the cell will be accessible to the player and to create well-made and interesting maps, often with the intent of emulating 'organic' structures like caves [3]. However, the resulting maps are still unpredictable, and to create a level that meets a set of designer-specified constraints often requires multiple passes through the generation process until an acceptable result is generated. This has the potential to take a very long time, especially if the rules for the finished map are very strict.

This type of map generation can be referred to as "generate-and-test," where maps are created until one meets the requirements set by either the user or the game designer [5]. Some research has been done towards improving the results of the generated results through a search-based system that scores each generated map and uses previous results to generate better ones in future generations [2]. This, however, requires the generation of multiple levels, which could still take a prohibitive amount of time. Our system attempts to bypass this restriction by using "constructive PCG" [5], creating only one level which is guaranteed to meet the specifications while still allowing for a number of flexible constraints and types of maps.

3. GENERATION CONSTRAINTS

To make the map-creation system more generic and useful, we included a set of constraints from which a map is generated. This list is not exhaustive, and more constraints could easily be added in the future. The constraints are as follows: the width and height of the map, a list of points that are required to be in one continuous zone of walkable space (ignoring temporary obstacles), boolean conditions specifying whether or not an open path should cross the map left-to-right sides or top-to-bottom, the percentage of the map that should be walls, which generation algorithm to use, as well as a list of parameters for the placement of temporary obstacles.

All the images of generated maps are 50x50 tiles, with the player's starting location (near the bottom left) and points near the three other corners specified as "required open."

4. PCG ALGORITHMS

In the following sections, we describe the different PCG algorithms we used to generate different types of maps. Images of each map are included at the end of the section.

4.1 Pathing Algorithms

Many wall-placement algorithms require the generation of random paths between two or more points. To accomplish this a degree of randomness is provided and a path is created by generating a set of 'moves' required to get from one point to another. Because diagonal movement was disallowed in our example game, the initial set of moves consisted of the minimum required horizontal or vertical steps connecting the two points. Then, based on the provided randomness parameter, a number of other moves are inserted into the list, which together cancel each other out (if a move to the left is added, so is a move to the right, etc.). Multiple paths are created to connect multiple points, connecting each new point with a random one on the existing path. As a final step, the list of moves is randomized.

This was intended initially as a helper function for the other algorithms, but we found that with a high-enough randomness the resulting paths were often sufficiently interesting to be used as maps on their own. However, because the paths move from point to point, the required points were often placed at the ends of long tunnels in the map; even with relatively high randomness the paths would often not veer too far in a direction with no required point on the way.

4.2 Cellular Automata

Cellular Automata (CA) is a PCG algorithm commonly used to create maps that have a very 'organic' feel, as if they could have been made in nature. The algorithm divides the world into tiles and determines if each tile is 'alive' depending on the state of its neighbors. These rules are run some number of times to produce the map's walls. The parameters in CA can be tuned to produce maps that have more or fewer walls [3, 2]. This process does not easily allow for creating maps that fit a set of constraints outside of a generate-and-test system; CA was used instead as a component to many of the other PCG systems rather than a level generation tool in it's own right.

4.3 Cellular Automata Modifications

As previously stated, CA is exceptional at creating maps that look as if they were naturally formed. However, they have the drawback of not always creating usable maps with different parts being unreachable. As we have the constraint of wanting certain points on the map to be connected, we took several approaches to ensure that maps were usable but still retained the CA feel of being naturally formed.

Our simplest approach, 'Cellular Automata Path Overlay,' was to generate a map using regular CA rules and then generate paths (using the algorithm explained above) between each of the specified points we wanted to be connected and cleared any walls that were on that path. The main benefit of this technique is that it is simple and quick. However, it can break the natural feel of the map because walls will have random paths cut into them making it more random. Also, if paths are generated that have too much randomness the final map will have more open space than intended; if the randomness is too low the paths will be very straight and stand out against the organic feel of the rest of the map.

Too improve upon this we first limited the amount of pathing required by only connecting points that were not already connected by walkable tiles, rather than generating a full path and inserting it into the map on top of CA. This improved the results, but for points that were near the edge or far apart (as our required points were), the paths were still long enough to suffer from the same drawbacks as the basic overlay, often creating maps where the borders consisted of no walls. Our final version of this algorithm, 'Zoned Path Overlay,' identifies the smallest number of paths required to connect all the given points, and generates them between locations in the given point's zones, rather than the points themselves. This ensures connection while limiting the lengths of the paths and preventing paths from being carved out that allow the player to walk directly from one required point to another.

4.4 Quadtree

In this approach, we use two different PCG techniques together to create maps. First, we apply the Quad Tree algorithm to break the map into smaller blocks, starting with the entire map as a single, large block. As mentioned above, we generated maps that were 50 by 50 tiles and set a threshold area for when to stop splitting blocks (our default was 300). Then, a horizontal or vertical split is chosen along with a random point in the block in order to divide the map into two smaller blocks. This process continues until the blocks are of a certain size [4]. Next, in each block, we used CA to generate the walls. In order to ensure more free space in each block, we tweaked our default CA parameters to create areas with larger open space. The result is a map that has a good distribution of open space and wall formations that look natural. However, this approach has two downsides. Firstly, it does not ensure that all specified points are connected or even available to walk on. The second negative is that it tends to create many areas in the map that are unreachable. We address the first issue in a similar manner to our CA adjustments and the second in Section 5 regarding obstacles.

In order to ensure that all points that are required to be open and connected, we took two approaches. Our first approach, Quad Tree Path Overlay, is very much like Cellular Automata Path Overlay in that it overlays a path on top of the generated map. This path will similarly be a random path between the two points. This path however has a tendency to cut through the natural-looking wall formations

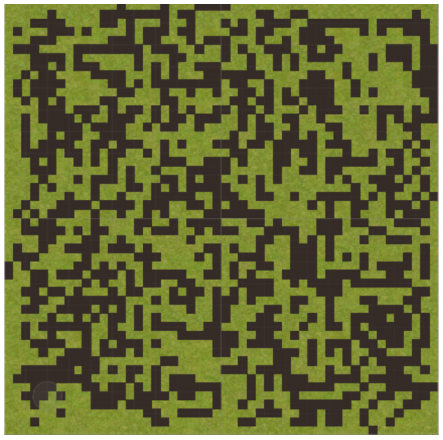

Figure 1: Normal Cellular Automata

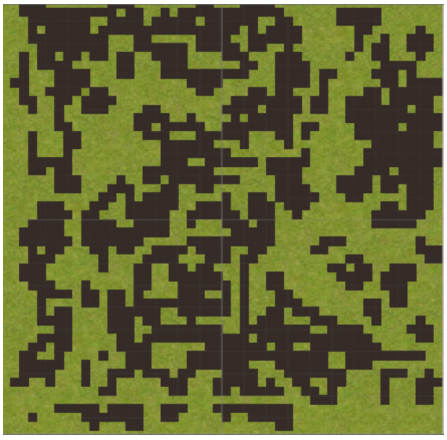

Figure 2: Basic Cellular Automata Path Overlay

and can be recognized.

Our other approach was to make use of pathing algorithms to ensure that there existed a path between each point. We named this approach Quad Tree Connect. With this approach we go through each point and ensure that it is connected to the previous listed point using A* but make walls impassible. If it is not connected, we perform A* again, this time allowing it to traverse walls with a very large penalty. We then remove any walls encountered on this path. This algorithm ensures minimal wall removal while maintaining the maps natural feel.

4.5 Agent-Based

We also used digger algorithms which randomly traverse through the map, creating rooms dispersed throughout the level [4]. We implemented two types of behaviors for the digger agents. The first is a random digger that simply digs in a direction with an increasing chance to switch direction each time it digs. It also randomly creates rooms of a fixed size, stopping after creating a set number of rooms. The other digger algorithm is a look-ahead digger which performs similarly to the random digger except that it does not create rooms if they intersect with another room. This creates dungeons that have rooms connected by hallways. To ensure that each specified point is open, we spawn a room at each of them and then connect them with a straight path to the nearest open tile. The downside of this algorithm is that it does not look as naturally formed as CA and also tends to build rooms along the edges of the map. In addition, there is a large amount of walls in the map and less open space.

5. OBSTACLE PLACEMENT

To make the maps more interesting to the player we also populated them with trees, which served as temporary obstacles. Trees block the movement of both the player and enemies, but the player can chop them down to permanently remove them from the map. This requires the player to choose between increasing their freedom of movement or keeping them away from danger, making the game-play require more strategy. We implemented four algorithms for tree placement: random (which randomly changes available tiles to trees), clustering (which generates a given number of clusters of a given average size), 'zone-connecting,' and 'zone-separating.' All placement algorithms can be controlled by parameters and any subset of them can be used in the generation of a map. Images of maps generated using each of these algorithms, including one using a combination of all the techniques, are shown below.

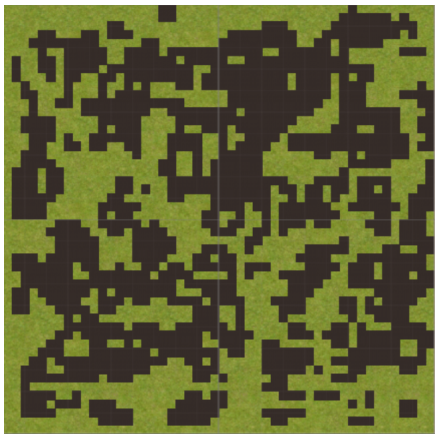

Figure 3: Quad Tree Overlay

5.1 Zone-Connecting

One of the problems we encountered with wall-placement (which is endemic to many map-generation algorithms) was that a large amount of open space was unreachable by the player. This wastes resources by generating more content than the player can interact with, and can limit the entertainment-value of the map as well. One way to solve this would have been to remove walls until enough of the map was reachable; this could lead to large amounts of open space in the map, limiting the the time the player could spend exploring. Instead, we addressed this in our 'zone-connecting' placement algorithm. It takes as input a number of tiles, and ensures that there is no walkable zone of the map containing more than that number of tiles that is not connected to the player's area. It chooses a random path between the two zones and replaces all wall tiles on that path with trees. This opens up the map for more exploration while retaining the level of 'closed-ness' present in the original wall-placement (at least, before the player starts removing trees). An ex-

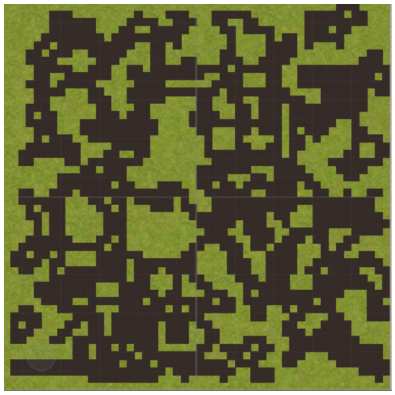

Figure 4: Quad Tree Connect

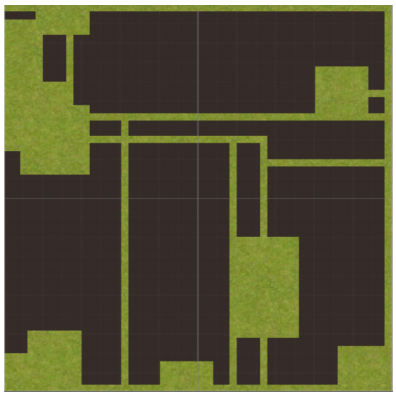

Figure 5: Blind Digger

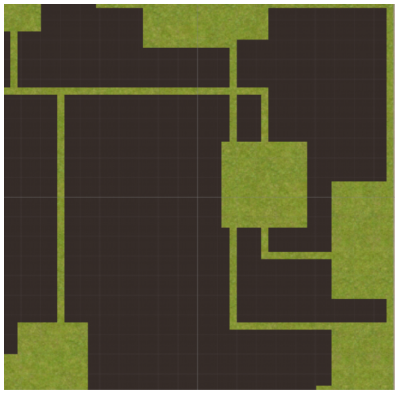

Figure 6: Look Ahead Digger

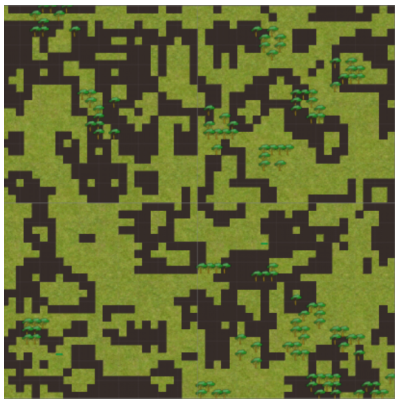

Figure 7: Clusters of Trees

ample map with trees placed using this algorithm is shown in figure 8.

5.2 Zone-Separating

Another problem also presented itself; walls and obstacles could happen to be generated in a way that allows the player to quickly reach their destination without presenting them with any obstacles, again wasting large portions of the map. To combat this we added another obstacle placement algorithm that splits zones of walkable tiles into multiple subzones by separating them with obstacles. This was done by taking in a list of pairs of points and starting to 'grow' a zone from each pair. When the two zones connected, trees were placed along the border to ensure that the player would be required to chop down trees to get from one to another. Random points were used to make the border-tree placements seem more natural while still restricting player movement, but it could also be used to ensure the separation of the player from some important point on the map. Figure 9 shows a map generated using zone-separation for tree placement.

6. EVALUATION METHODS

In order to evaluate our maps, we created a game, "Lord of Maps," in Unity [1]. The goal of Lord of Maps is to explore the map, find a zombie spawner, and destroy it. The spawner slowly creates zombies that wander the map and attack the player if they are nearby. In addition, the map is initially shrouded in fog of war and rendered black until the player has explored it. The maps are also populated with trees the player can chop down.

We implemented each of the algorithms discussed previously; however, for our study we had our players play the regular cellular automata and zoned automata in the interest of time. We chose these two in order to focus on the difference between regular cellular automata and zoned automata which is much more heavily constrained in order to create valid maps each time. For tree placement, we chose to use random placement in cellular automata and all four of our obstacle placement algorithms for zoned automata. It is also important to note that we had to use a slightly different algorithm for choosing the spawner location in the normal cellular automata map because each generated map is not playable if we chose a random location for the spawner (it may be unreachable). Thus we decided to choose the farthest possible location that the player could reach as the spawner's location. This lead to some users having to restart the game due to the spawner's location being too close.

We had a total of nine users participate in our study, all of which were fellow Computer Science students in our Graduate Game AI class. After playing each map, we asked our participants to answer a short survey to give feedback on each map. We designated the map generated with cellular automata to be Map A and zoned automata to be Map B. We detail our results in the following section.

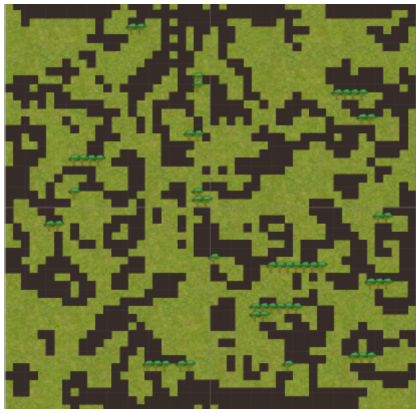

Figure 8: Zone-Connection Obstacles (min size of 5 tiles)

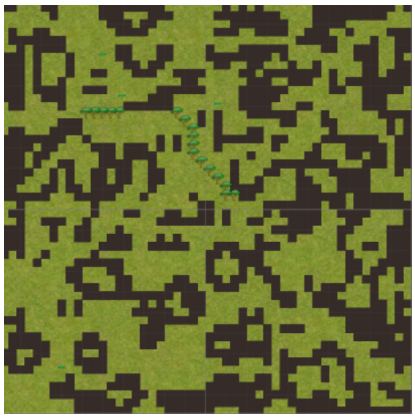

Figure 9: Zone-Separation, separating top-right and bottom left

7. RESULTS & CONCLUSIONS

From our results we found that a majority of our participants preferred the second map that was generated using zoned cellular automata. The main questions in our survey focused on determining whether or not the maps were able to retain their natural, organic feel, which cellular automata is particularly good at creating. The results of these surveys are displayed in Figure 11. A larger percentage of users felt that the map generated using zoned automata and our wall placement algorithms was more enjoyable and natural.

For wall placement, a majority of participants noted that in Map A, the walls did not feel natural. Some mentioned that they felt a little random. This can be partially attributed to the parameters we used for the cellular automata algorithm which we used to induce a more scattered set of walls as seen in Figure 1. For Map B, participants were much more split. Some said that the map seemed to be random while others felt that it was similar to a cave or valley. This can be partially attributed to the randomness inherent with the map generation algorithm.

Participants found tree placement in both maps to be natural. This is rather interesting as we spawned trees completely randomly in Map A. However, upon analyzing the feedback comments on Map A we find that several realized they were random and also thought that they seemed

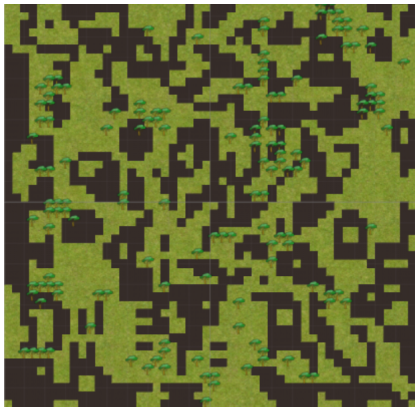

Figure 10: A combination of all obstacle placement algorithms

natural because they blocked their path and were forced to chop them down. In their short answers, several participants noted that trees were clumped together more often but also that trees still felt randomly placed. Both of these responses reflect the object placement algorithms that we used to generate trees throughout the map which indicates that they were effective.

Next, we asked participants to describe any differences they noticed between the two maps after they completed playing the game. In most of these responses, participants noted that Map B was much larger in terms of space that they had to explore. Many players noted that map B felt much more like a level because they spent more time trying to find the spawner and had to cut down trees. This was due in large part to the issue mentioned earlier in which we were forced to place the spawner in the farthest reachable tile which was oftentimes rather close. This feedback reinforces that using cellular automata is useful for creating a natural feel but requires constraints to create a playable, more challenging level.

Lastly, we asked our participants if they believed that this tool would be useful to developers looking to generate maps. Almost all of our participants agreed that this would be useful with 1 being neutral. We feel that our map generation can thus be of use to developers particularly looking to generate 2D maps in Unity.

We noticed some limitations that may have affected our results.We feel that some of player's ability to focus on the maps were partially hindered by the game's playability. Several players noted that the game's controls were difficult while some also felt some of our graphics were distracting particularly with our sprites not being centered. Our other problem stems from our definition of 'natural.' We initially felt that using the term 'natural' in our survey would aptly convey the organic feel of the map and used it in our survey by asking them to rate how natural different aspects of the map were. However, in retrospect, we should have provided a more clear definition of the word. We were asked for our definition during the study and several participants interpreted it differently as evidenced by some of the short response answers.

Figure 11 shows the results of our survey. 'Natural Walls A' represents the participant's answers when asked if the wall placement was 'natural' in Map A (vanilla CA). 'Nat-

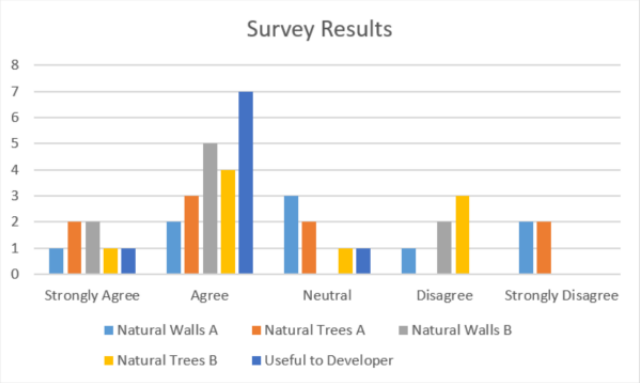

Figure 11: Survey Results

ural Trees A' represents their answer about tree placement in Map A, etc.

We feel we were able to make marginal improvements on the cellular automata algorithm. In addition, we built a tool that would be useful to developers in level design for 2D dungeon games as it is able to generate maps with walls and obstacles that can be set with a variety of different parameters to design it. Our analysis of different strategies using cellular automata yields that it is successful in creating quality maps and that can further increase this system's usefulness by constraining it in such a way to ensure that the maps maintain playability.

8. FUTURE WORK

Currently, our system supports a number of generation constraints that can be met by any of the core generation algorithms; in the future we would like to expand the number of available constraints to make the system more flexible. For example, a closer integration between the placement of temporary obstacles and the placement of the walls could potentially lead to more interesting maps than our current two-phase system. It would also be interesting to apply these strategies to three-dimensional maps or infinitely-scrolling maps, particularly because procedural level generation has traditionally been focused primarily on only fixed-size two-dimensional games.

9. REFERENCES

[1] Unity. https://unity3d.com/.

[2] Mike Preuss Julian Togelius and Georgios N. Yannakakis. Towards multiobjective procedural map generation. In *Proceedings of the FDG workshop on Procedural Content Generation*, 2010.

[3] Georgios N. Yannakakis Lawrence Johnson and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the FDG workshop on Procedural Content Generation*, 2010.

[4] Julian Togelius Noor Shaker and Mark J. Nelson. Procedural content generation in games: A textbook and an overview of current research. In *Springer*, 2016.

[5] Mohammad Shaker, Noor Shaker, Julian Togelius, and Mohamed Abou-Zleikha. A progressive approach to content generation. In *European Conference on the Applications of Evolutionary Computation*, pages 381–393. Springer, 2015.

[6] Stanford. A discussion of the game of life. http://web.stanford.edu/ cdebs/GameOfLife/.

[7] John Von Neumann and Arthur W. Burks. Theory of self-reproducing automata. In *IEEE Transactions on Neural Networks 5.1*, 1966.