ABSTRACT

In this paper we present a tool that procedurally generates a video game dungeon-esque graph for a user and find the best path from start to finish. The best path is determined based on several variables selected by the user. This tool is particularly useful for game developers who strive to create usable, challenging and fun games. Our tool can help designers create unique and complex dungeons. Unlike other procedurally generated dungeons, our approach is hierarchical and has a minimum connectivity feature to prevent dead ends from being created. Our pathfinding component finds optimal paths based on the user’s choice. Users can choose whether to find paths based on amount of gold, difficulty level or time to complete the room.

INTRODUCTION

To determine whether or not a video game is playable is a difficult question to answer. There are not many tools or methods for evaluating the usability of a game [1]. Not only does a game have to be usable, but users prefer challenging and fun gameplay [2]. This can cause stress upon designers, who have to consider both points when developing a video game. In order to solve the latter, many designers are using procedural generation.

Procedural generation is a way to create content algorithmically rather than manually [3]. Often used in gaming, procedural generation will tend to result in smaller file sizes, less memory usage, and reduced work for developers and artists. For the user side, procedural generation will give a more random and unpredictable experience for the
player. When designing content that does not utilize procedural generation, development can be expensive and non-scalable [4]. van der Linden [5] lists four benefits for using procedural generation in video games: (i) rapid generation of content that fills the requirements of designers, (ii) possible diversity of generated content, (iii) amount of time and money saved by a designer/publisher, and (iv) the fact that procedural generation can adapt to the player.

Procedural generation can be associated with video game dungeons. A dungeon in video games usually refers to a labyrinth-like maze of many levels that a player has to traverse in order to obtain rewards [3]. Popularized by the 1980 procedurally generated dungeon crawling game Rogue, many successful video games such as the Diablo and Elder Scrolls series have implemented procedural generation to create unique dungeons and mazes of their own [6].

One implementation of procedural generation is Mike Anderson's dungeon-building algorithm for roguelike games[7]. Anderson's algorithm [7] creates a Euclidean set of interconnected rooms, doors, and tunnels, and also features an entrance and exit. Every room must also be reachable by the player. Forsyth [8] furthered Anderson's research by removing the confines of a two dimensional world, and utilizing procedural generation in a non-Euclidean space. Forsyth's algorithm [8] is defined by two stages: (i) Pick a random room, pick a wall in that room, then attach a room with a random size to that wall. (ii) Enforce minimum degree of connectivity by searching for a node that does not have minimum degree of connectivity, and attaching it to another node that has not met the minimum degree of connectivity.

Our procedural generation algorithm is based on Forsyth's algorithm [8], but we chose to use a hierarchical approach or a parent-child relationship with our algorithm., In Forsyth's dungeon generation [8], multiple floors are created, while in ours, it is generated on one floor. We also have randomly generated properties for each of our rooms: gold, difficulty, and time to complete the room. Using these variables, we are able to find the best path based on the parameter a user chooses.

We chose to use a modified version of Dijkstra's algorithm [9] in order to obtain a shortest path for a given variable. This would pose a problem if one wanted to find the greatest value for gold, since Dijkstra's algorithm would attempt to actually find the lowest value. We mitigated this problem as will be explained in the next section.

IMPLEMENTATION

Procedurally Generated Dungeon

We generate the dungeon based on a modified version of the algorithm presented by Forsyth in [8]. Like Forsyth's algorithm, we represent the dungeon in non-Euclidean space. However, we don't pick walls with our generation algorithm, but rather use a parent-child tree relationship to generate the dungeon.

The procedural generation of the dungeon is done by expanding a tree downwards based on preset changeable constraints. The tree is generated by picking a random number between one and the set limit, and adding that many children to the root. One of those children are chosen, and the same process is repeated on that child. This process repeats until the total number of nodes in the tree is one more than the requested number.
of rooms. At the end of initial generation, the dungeon will have a main path to the exit, and each layer will split off into dead ends if the number of nodes at that depth is greater than one.

The gold value, difficulty, and time are randomly generated values based on constraints. The layer value is based on the hierarchical nature of the generation, and represents the depth in the tree. The root node is considered as the start node, and has a depth of zero. The start node and exit node do not count towards the number of rooms. The number of nodes that the tree contains does not include the start node. Each node has an ID associated with it. The start node has an ID of zero. Each node generated after it has an ID of one greater than the previous node. The last node generated will have an ID of one greater than the number of rooms requested. This is how this node is identified as the exit. This also means that the exit node can be at the same depth as other nodes.

The process for enforcing minimum connectivity rules is as follows. We pick the first node that isn't the start node or the exit node. We add up the total number of parents and children of this node, and that gives us the total number of connections that this node has. We then check that number with the minimum required connections. If it is equal to or greater than that number, we then move on to the next node in iterative storage. If it is not, we then pick a random other node that is not the start, exit, or a node that is already connected. Depending on the depth relationship with this chosen node and the original node, it is either added as a parent or child node. Unlike Forsyth's algorithm [8], we don't allow nodes to connect to themselves. Rooms with self-connections don't make sense in the flow of a dungeon, so we decided to not allow it. This means that if the number of rooms in the tree is less than or equal to the required minimum connectivity, and the minimum required connections is greater than two, it will get in an infinite loop and be unable to meet minimum connectivity rules. By default, the minimum degree of connectivity is two, and should never reach an infinite loop, as a zero room dungeon doesn't have minimum connectivity rules applied, and a one room dungeon has 2 degrees of connectivity, the start node and the exit node. Since Forsyth's algorithm [8] allows for node self connection, it is possible that a dungeon will be generated with dead ends, which our algorithm avoids entirely.

Our algorithm can completely stop dead ends from being created. The spacial relationship between two nodes doesn't matter, as we are in non-Euclidean space. The random nature of node selection when trying to achieve minimum connectivity requirements leads to some interesting possibilities. In a large dungeon, there could be a node near the start that connects to a node near the end, making a short path. There could also be a node that has multiple connections, which will create a centralized "Hub" room with many paths to different nodes across the dungeon.

We decided to use an object oriented design, in order to focus on clean and modular code. The objects that we created are as follows: RoomTree, which represents the nodes and holds the gold, difficulty and time values. The Dungeon class, which holds all the RoomTrees and connects them to other RoomTrees.

**Optimal Pathfinding**

For the pathfinding component of our algorithm, we considered the values that are
held within the nodes of the graph, rather than the edges. This was done by taking the
value for the node, and considering that to be the weight of the edge to that node. So for
example, if we have a node with the lowest difficulty value of the fringe nodes, we will
consider the value of the node to be the weight of the edge to it, and then add it to the tree
vertices. Depending on the
configuration of the search, this could be either gold, difficulty or time. In order to find
the greatest gold value, we multiplied all gold values by negative one when optimizing
for gold. This posed another problem as Dijkstra's algorithm does not allow negative
cycles [9]. We were able to mitigate this issue by implementing a 'visited' flag for each
node in the graph. For example, if a node has been added to tree cycle or 'visited', then
it is flagged and will not be placed in the tree vertices again.

RESULTS

In this section, we present some examples of dungeons that could be generated with
our tool.

![Simplest Dungeon](image)

Figure 1: Simplest Dungeon
Figure 2: One Room Dungeon

Figure 1 shows the simplest dungeon that can be generated: a zero room dungeon, while Figure 2 shows a single room dungeon. The gold value, difficulty, and time are randomly generated values based on constraints. The layer value is based on the hierarchical nature of the generation, and represents the depth in the tree. The root node is considered as the start node, and has a depth of zero. The start node and exit node do not count towards the number of rooms. The number of nodes that the tree contains does not include the start node. Each node has an ID associated with it. The start node has an ID of zero. Each node generated after it has an ID of one greater than the previous node. The last node generated will have an ID of one greater than the number of rooms requested. This is how this node is identified as the exit. This also means that the exit node can be at the same depth as other nodes.

Figure 3: Fifteen Room Dungeon with dead ends

Figure 3 shows a fifteen room dungeon that demonstrates the hierarchical nature of the tree. The single path to the exit node can be seen, as well as the split into dead ends at each layer. After the initial generation of the dungeon, we create an iterative storage
that allows for quick access to all nodes without requiring recursion, and helps prevent infinite loops.

Figure 4: Fifteen Room Dungeon With No Dead Ends

To ensure preventing dead ends, we use the minimum connectivity algorithm we described above. Figure 4 shows a fifteen room dungeon with minimum connectivity rules. In Figure 4, the degree of connectivity is two but can be changed to any value no greater than the number of rooms. It is important to note that with the high interconnectivity of the graph at this point, it is hard to display as a standard tree because of the many overlaps. The hierarchical nature still exists and is not affected by enforcing minimum connectivity, but is not displayed as such in Figure 4. As a result of initial generation, there will always be a path to the end by following in order the depth of each node.
Figures 5 and 6 illustrate how the optimal path is found. On every iteration of our pathfinding algorithm, the smallest node is stored in the 'smallest' variable. This smallest variable is considered to be a tree vertex, and the path to each fringe node is found. Again, this path value is found from the value found within a node, and is dependent on which type the user has selected. After this, the cost for that path is stored if it is less than a path that was stored before. After iterating through every object until the goal is found, we will eventually obtain the optimal path for the problem. Figure 5 shows the optimal path using the gold parameter while Figure 6 shows the optimal path using the time parameter for a 10 room dungeon.
CONCLUSION

In this research we have developed a tool that procedurally generates a graph with a video game dungeon-esque paradigm in mind. Each node or "room" will have values such as gold to be found, difficulty and the time that it would take to complete the room. Unlike other procedural generation algorithms, our algorithm uses a hierarchical approach, where each level generates a certain number of children based on the connectivity set by the user.

After procedurally generating the dungeon, we determine the optimal path from the start to the end node based on several parameters. We tested our algorithm in multiple scenarios and verified that it provides the correct optimal path. Our tool is very beneficial for video game designers to generate maps or dungeons. Game designers would not only have the graph design generated, but also the optimal path for the player, so the designer could add indicators for the user on which path they would receive the best output from.
REFERENCES


