An Approach to Maze Generation AI, and Pathfinding in a Simple Horror Game

Matthew Cooke and Aaron Uthayagumaran

McGill University

I. Introduction

We set out to create a game that utilized many fundamental topics from class. Our game is based on a generated maze that continuously expands in size as the player navigates the terrain. We wanted to efficiently generate new lengths of maze while having enemies pathfind through the maze. The player is dropped into a closed, dark, corridor with only a flashlight; very little information is given to the player. The objective of the game is to navigate the maze while avoiding enemies, in order to ultimately find a treasure.

The initial perfect maze is generated using backtracking techniques in order to allow for branching while preventing loops or unreachable sections of the maze to be developed. Furthermore, the Game Manager stores a list of Maze instances. As the player traverses the maze, there exists boundaries that when crossed will signal the Game Manager to create an adjacent maze in the direction represented by the boundary. This allows for the maze to be perceived as infinite in size as the player will not be able to distinguish the interior of the mazes a part. The player is controlled with a first-person view of a camera, using lighting tricks to give the user the illusion of holding a flashlight. When the maze is generated, invisible monsters as well as a treasure chest are placed into random corridors, which the player does not know the location of. Both the enemies and the treasure chest use sound as a means of warning the player of their location. The player must then attempt to navigate to the treasure while avoiding enemies. If the player successfully reaches the chest, the game ends and the player wins. If the player is caught by an enemy, the game ends and the player loses. We needed to be able to generate mazes, allow for one or more enemies to use pathfinding techniques, and use dynamic lighting efficiently.

We succeeded in generating the mazes during runtime, as well as placing AI enemies in these generated mazes. The game acts as a seamless experience to the player; they do not notice the transition from old to newly generated mazes. Once the player reaches a threshold, the monsters follow the player effectively. Although we initially planned on using A* search for the monster to simply move towards the player. This was later modified to use a pseudo-occupancy map approach for target tracking. Moreover, the enemy moves cell to cell depending on the probability of the player being at the cell, where the probability is defined based on the enemy's field of view. If the enemy were to see the player via ray casting then the first cell towards the player will be given a probability of 100%, with adjacent cells acquiring a probability based upon the defined diffusion rate of the probability. This allowed for a somewhat intelligent enemy that the player must escape from, as the game ends once the enemy captures the player.

II. Background

We focused on 3 main topics during this project: a) Procedural generation of our maze, b) AI, and c) pathfinding.

a) Procedural generation has been a topic studied extensively in modern computer game development. Many algorithms have been proposed to handle terrain generation, in 2D and 3D, as well as maze generation. (Okamoto, 2009)

There are various types of mazes defined in modern games, our game specifically focusing on perfect mazes.

b) AI is currently one of the most researched fields in computer science. AI algorithms are implemented almost everywhere we look. From data-analysis (Fayad, 1996), to complex aviation software (Mckeown, 2007), to modern video games (Russel & Norvig, 1995). Although many complex algorithms are used, game AI has ultimately one goal, to give the agents the illusion of intelligence. We focused on two different methods when testing this project, a probabilistic search (Isla, 2013), and a simple automated walk (Quinlan, 1986).

c) Pathfinding has been another major topic in computer science, especially the video game industry. (Bulitko, 2010). A common pathfinding technique is A* search in which the path is defined by the optimal f-value. The f-value being defined by the heuristic and cost. Using Manhattan distance as a consistent measure for both the heuristic, the distance from the enemy to the player as well as the cost the distance travelled, A* was best suited to complement the maze and AI. We adjusted this pathfinding however, when the player leaves the sight of the enemy, to diffuse possible locations probabilistically. This allows the enemy to search more naturally for a player who has exited line of sight.

III. Methodology

The game we designed, encapsulates various techniques and algorithms that allow for a promising user experience. Utilizing basic backtracking techniques, perfect mazes were generated seamlessly. Exploring the many features supplied by Unity, a comfortable, more user engaging interface is constructed. Additionally, the concept of occupancy graphs allowed for the enemy AI to have more realistic feel to search for the player.

III.I. Maze Generation

The game begins with the player perceived to be trapped in a maze, which seems to have no end. However, unknown to the player the ir perception is correct, as the maze itself expands in size whenever the player crosses a certain threshold. Moreover, a new Maze instance is instantiated to the current maze if necessary. Furthermore, with the additional aesthetics the maze adapts an eerie dungeon-like feel for the player to traverse to add to our horror themed game.

The GameManager script will instantiate the initial Maze prefab, where the maze is to be constructed as a 15 x 15 grid of MazeCell objects. We will refer the the 15 x 15 mazes as a “Maze Space”. Each MazeCell is about 3 x 3 units in size, which contains a set of booleans to represent if there is a wall at each side of the cell (ie. North, East, South, West). To ensure that no loops are created and that there are no unreachable cells a backtracking algorithm is utilized. Firstly, a random MazeCell is initialized, this cell is arbitrarily selected within the 15 x 15 array of MazeCells in the Maze Space. The cell is then marked as visited and stored in a stack of cells that have been visited. Next, a random direction is selected that does not already have a wall, a new adjacent cell is initialized and is pushed to the top of the stack. With each iteration, the algorithm continues to traverse through the maze generating each cell. However, if the random direction chosen has already been initialized, both cells mark their respective direction to one another as a wall and a new direction is chosen, to prevent loops. Once all directions for cell is occupied by a wall or is a path to another cell, the MazeCell object is popped from the list as it no longer has any viable direction, to allow for branching. With both branching and no loops in the Maze, we have successfully generated a perfect maze. It should be noted there are two special case of MazeCell, the boundary cells which have a wall to enclose the maze and a subset of boundary cells, the four enterance cells which do not have a wall for their respective side, (0, 7), (14, 7), (7, 0), and (7, 14) to allow for movement between mazes.

After the MazeCells have been initialized, four boundary columns are instantiated, two vertical at ±12 units from the center and two horizontal at ±12 units from the center. These boundaries use box colliders with triggers attached to respond when a RigidBody, which is the Player object intersects the the boundary. Depending on the boundary crossed an adjacent maze will be constructed with the same rules that the initial maze followed. Recall, that the mazes have 4 entrance as such the entrance are now adjacent allowing the player to transition between mazes without noticing at all.

Lastly, to allow for the player to experience the overall horror aspect of the game several assets available on Unity ere used. An detailed stone material was used to allow the interior of the maze to have an overall dungeon-like appearance, which was aided with the lack of a lighting source. A fog aset was added to aid with the element of surprise so that the player cannot see down long corridors.

III.II. The Player

The player is controlled with a first-person controller from Unity. This controller allows for 'W' 'A' 'S' 'D' movement as well as using shift as a modifier to run. It also allows us to hook the mouse and use it to look around. We attached this controlled to a camera, in order to get a first person view. There is no physical representation of the player's body, only the view from the camera. The camera is the parent of two lighting effects, one to generate a diffuse spatial light and the other for the point "flashlight". These are children of the camera, and looking around therefore changes the angle of light.

III.III. The Enemy

We designed our enemy to be located solely using sound; it is therefore invisible. The player can hear the enemy as they approach and must decide which path through the maze they will take. The enemies are placed pseudo-randomly in each generated maze. These enemies then walk around their environment using basic AI. Enemies use a raycast field-of-view system to determine line of sight to the player, and if line of sight is established, they use target tracking to the player.

a) AI

The AI for the enemy involves basic random movement of the enemy. After the AI is randomly placed in the maze it will continually move to a single adjacent cell in a random direction. This will repeat until the player is spotted in the field of view. This is achieved by using probability values for adjacent cells equal to 0.1 and not 0 (Isla, 2013), allowing the enemy to search its immediate vicinity.

b) Field of View

The enemy's field of view is calculated using raycasts in Unity. While the enemy is wandering its area, it is sending out invisible rays at each update. These raycasts collide with any collider in the environment, and disappear. If they come into contact with the player, they call a function the checks the player's location and implements pathfinding to it. Rays are computationally light in Unity, and therefore can be cast regularly without major CPU impact. (see Unity Physics.Raycast)

c) Pathfinding via Target Tracking

If the AI has not seen the Player, moves at random throughout the maze. More specifically the AI moves to the cell which holds a greater probability. In the case of a tie, one is chosen at random. Initially, all cells are given a probability of 1%, however, once a ray from the cell hits the player the probabilities will change. As the enemy now sees the player, the first cell in the direction of the player gets a probability of 100% as such the enemy will move towards that direction. All cell probabilities diffuse to cells adjacent to it by a factor of alpha, the enemy will continue to move in the direction of the player. In addition, if the player were to leave the enemy's field of view, the enemy will still have an idea where to go as the probability diffuses among adjacent cells. This allows for the player to experience as if the enemy is chasing them even when the player escaped a visual range.

III.IV. The Treasure

The goal of our game is to ultimately find a large treasure chest. This chest is located somewhere on one of the generate Maze spaces, but never the initial 15x15 maze space. Like our enemies, the treasure play an audio cue to alert the player to their location. When the player is alerted of the location of the treasure, they may explore the immediate surroundings in order to find it.

III.V. Ending the Game

The game ends in one of two ways: the player gets caught by an enemy, or the player successfully reaches the treasure without getting caught. Each of these scenarios are handle with colliders: one in the enemy and one in the treasure chest. If the player collides with the enemy, the Unity scene is changed to a Game Over screen, with an audio cue. Alternatively, if the player reaches the chest, the Unity scene is changed to a You Win screen.

IV. Results

To determine the effectiveness of the pathfinding algorithm experiments were conducted involving both the player and enemy. To determine the effectiveness of the pathfinding algorithm, the maze was modified in order to place both the enemy and player in 1 x 10 cell corridor. The player would move around the corner of the corridor in order to get out of the enemy's line of sight. (Figure 4) Five experiments were conducted in which the enemy was placed certain cell blocks away from the player each experiment had 5 trials.

Cell Distance	Player Was Caught
7	p=0.6, n=5
6	p=0.6, n=5
5	p=1.0, n=5
4	p=1.0, n=5
3	p=1.0, n=5

As the results indicate nearly always within 5 cell distance the player will get caught even after they turn the corner. With a distance of 6 and 7 the enemy caught the player most times however, certain times the enemy may move backwards, before ultimately catching the target as the probability diffused as such.

V.I. Conclusion

We started our project with the goal of creating an infinitely large procedurally generated maze, in which we would host a horror game. We wanted to create a scary atmosphere that could be expanded upon indefinitely. Our game is by no means finished, and we have many more ideas we would like to implement. Working through this project has taught us a lot, both about game design principles and the Unity engine itself. We intend to keep working together in the Unity development environment after the course, working on this and other projects.

V.II. Future Work

There are many improvements that can be done to our game. Firstly, although our game is playable, it is not fully fleshed out. We would like to add different maps and textures, new enemies, new objectives, and a story. It is also important to implement some sort of minimap feature to keep track of where the player has explored. We would like to implement the Occupancy map algorithm (Isla, 2013) fully, and improve the enemy pathfinding. The enemy's artificial intelligence is also very basic, as they random walk around their local area. This can be improved by using decision trees to perform different actions depending on environmental information. Our maze generation can also be improved; adding terrain generation and complexity and creating a true 3D maze, with elevation changes, is planned. More optimisation can also be done in order to allow for smoother gameplay on lower-end machines.

VI. Supplemental Materials

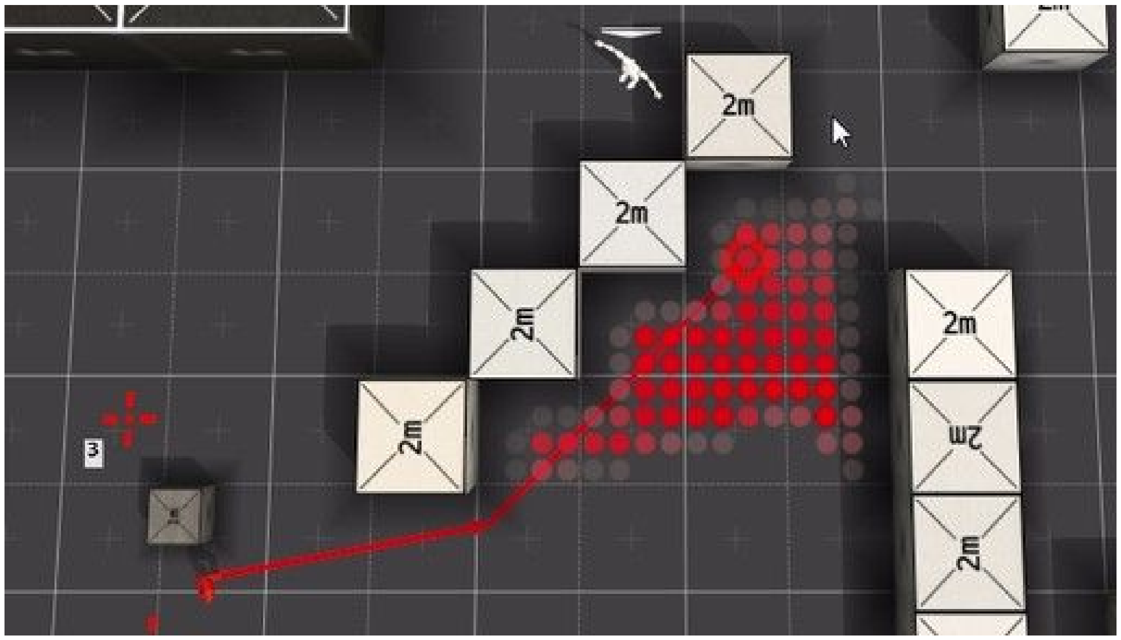

Figure 1. Isla Probability Dispersion; AIGameDev.com

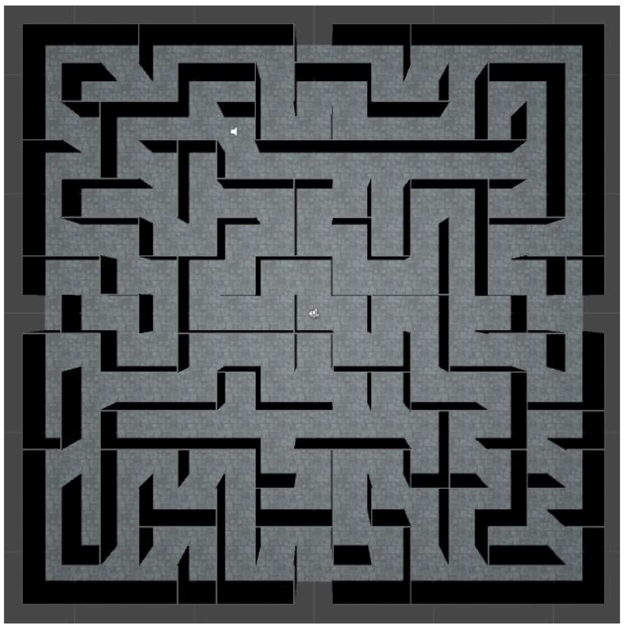

Figure 2. Initial Maze Space

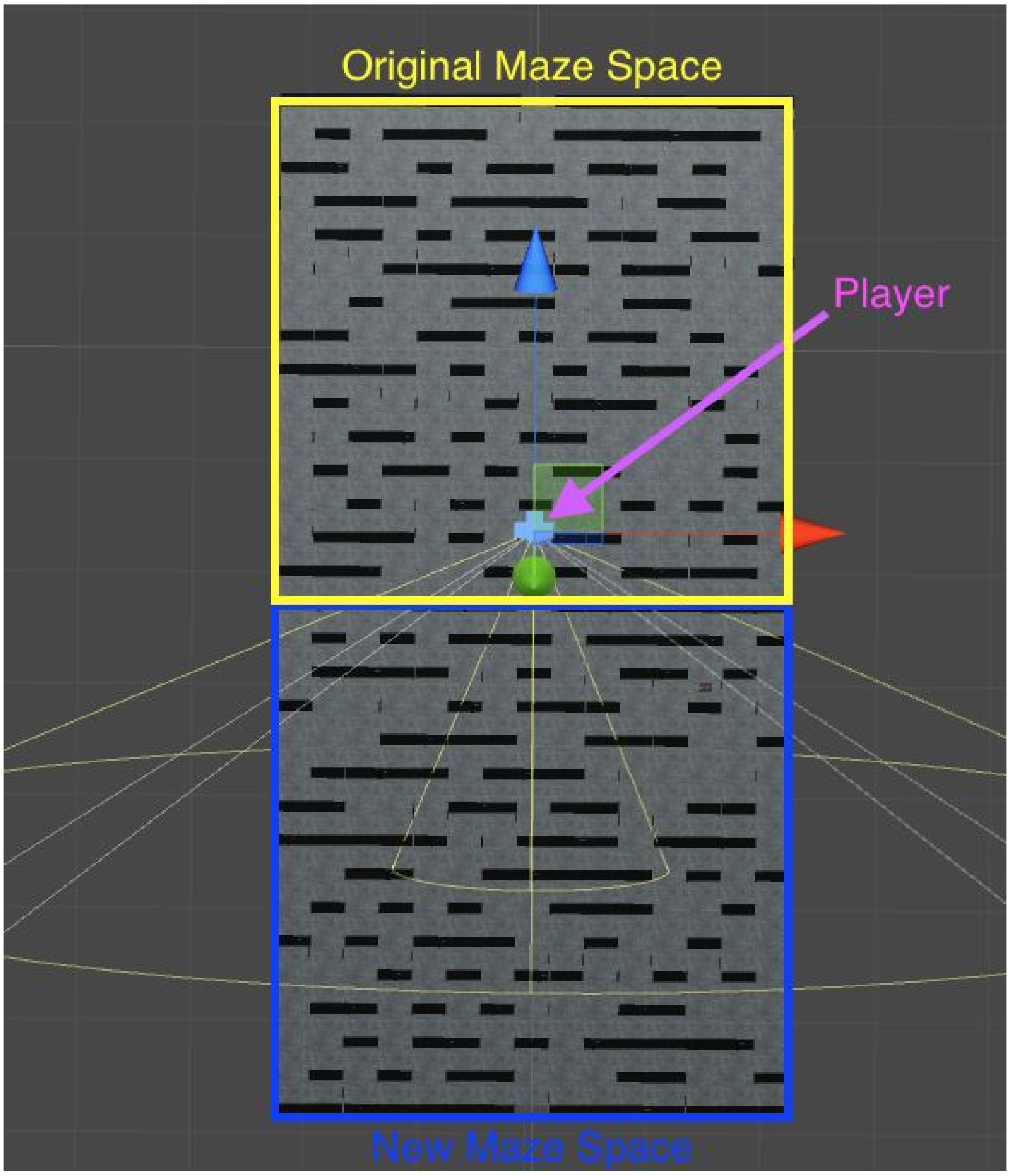

Figure 3. Player reached threshold to generate next Maze Space

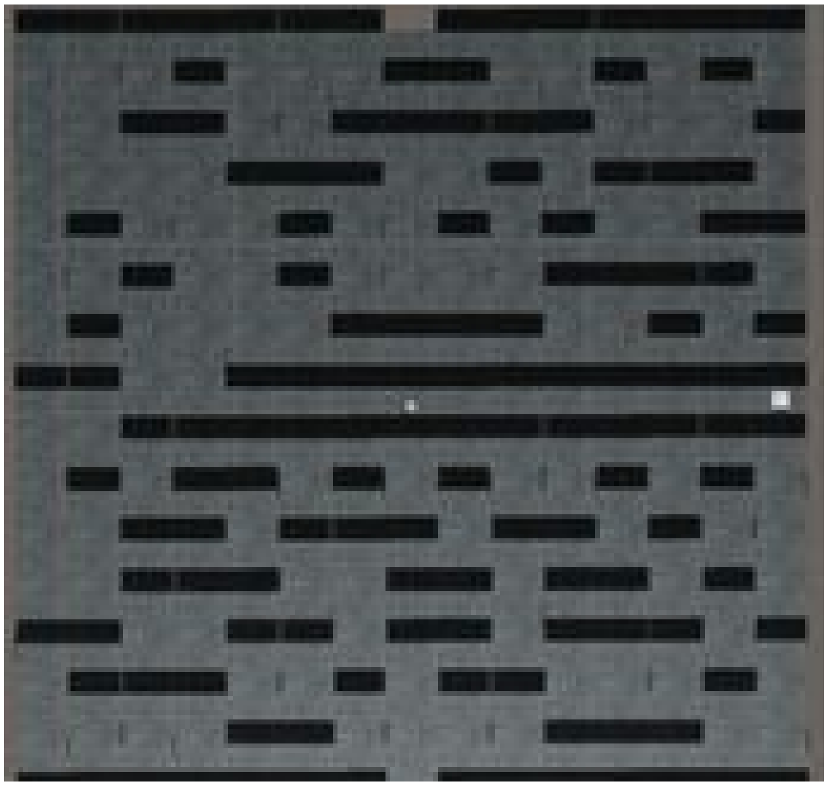
Figure 4. Player and enemy in testing corridor

VII. References

Artificial Intelligence; A Modern Approach. Stuart J. Russell and Peter Norvig. 1995.

Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding. Vadim Bulitko, Yngvi Bjornsson, and Ramon Lawrence. 2010.

From Data Mining to Knowledge Discovery in Databases. Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. 1996.

How to make a picturesque maze. Yoshio Okamoto and Ryuhei Uehara. 2009.

Induction of Decision Trees, Quinlan, J. Mach. 1986.

Occupancy Grids and Emergent Search Behavior for NPCs. Occupancy Grids and Emergent Search Behavior for NPCs. AiGameDev.com. 2017.

The Role of Artificial Intelligence in the Integration of Remotely Sensed Data with Geographic Information Systems. David M. Mckeown. 2007.

Third Eye Crime: Building a Stealth Game Around Occupancy Maps. Damián Isla. 2013.