Optimal Path Search with Procedural Generation in a Video Game

Liz Demin

May 9, 2018

Contents

Optimal Path Search with Procedural Generation in a Video Game

Liz Demin

May 9, 2018

Abstract

This project focuses on implementing a video game that uses procedural generation to generate levels that can always be beaten by the player. The game is a top-down, 2D rougelike-style exploration game where a player must traverse a maze to reach an end point without running out of Energy. The project focuses on creating an algorithm that generates an optimal path through a maze and distributes items that increase the player's Energy attribute in the maze environment in a way that the player can always reach the next Energy item without Energy reaching zero.

1 Introduction

Procedural generation is a common method of algorithmically creating content in video games. Procedural generation takes a finite number of inputs, which are often randomly generated, and uses them to create game elements. For games that consist of repeated elements and gameplay mechanics, procedural generation is an effective way to generate large amounts of unique content for the player to experience. Procedural generation has the potential to generate content that would otherwise be unfeasible or difficult for a developer to create manually due to complexity or size. Procedural generation allows players to experience new gameplay with each play-through, adding longevity and freshness to games that would otherwise be repetitive or short.

One issue with procedural generation is ensuring that a generation algorithm will produce consistent content that will result in levels that can be completed with the rules of the game. Many games deal with this issue by imposing a large number of restraints on the gaming world; however, this approach has a tendency of creating repetitive, uninteresting, or predictable content. Other games impose too few constraints and result in levels that are mostly empty and contain very little objectives for the player to complete. Although both approaches result in consistent gameplay, they are a heuristic way of ensuring that a level can be completed and their effectiveness depends largely on the player's gaming preferences.

The project's goal is to use a set of algorithms to procedurally generate levels for a video game while ensuring that each level can always be completed without the use of additional modifications or heuristic algorithms. The implemented game is a top-down, 2D roguelike-style game that has grid-based movement mechanics. In the game, the player navigates a 2D maze by moving up, down, left, and right with the goal of reaching an end point within the grid. The player has an "Energy" attribute that decreases every time the player moves or encounters an enemy, and the player must reach the end of the maze without the Energy attribute reaching zero. The level that the player navigates has Energy items that replenish the player's Energy attribute when the player picks them up. The procedural generation algorithms developed for the game produce levels based on finite, numerical information about the level's layout and seek to minimize the inconsistent gameplay that results when imprecise algorithms are used to create similar games.

2 Approach

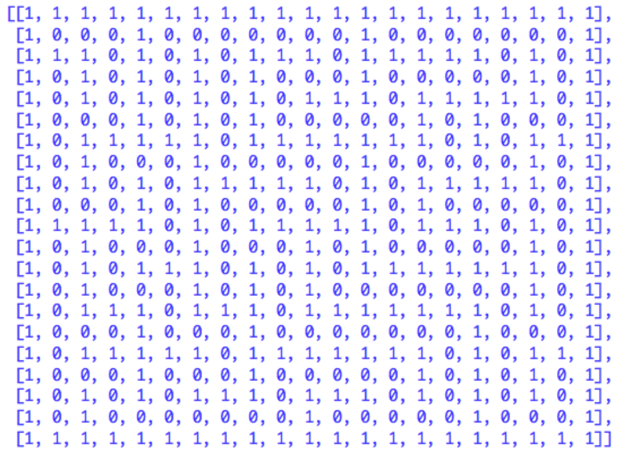
The primary mechanism of the game is the player's "Energy" attribute, which decreases every time the player moves or encounters an enemy and is replenished when the player picks up an Energy item. The Energy items replenish a set amount of Energy units and the enemies subtract a randomized amount of Energy units, which is determined by their numerical HP (hit points).

The game generates a different maze for each level and algorithmically places Energy items and enemies along the optimal path leading to the end point, ensuring that the player is always be able to complete the level by keeping track of the net Energy gain and loss. The player completes the level upon reaching the "Portal" item, which marks the end point of the maze. After, a new level is generated and the player is reset. The game's core algorithmic component consists of maze generation, maze solving, and the Energy item and enemy distribution algorithm.

2.1 Maze Generation

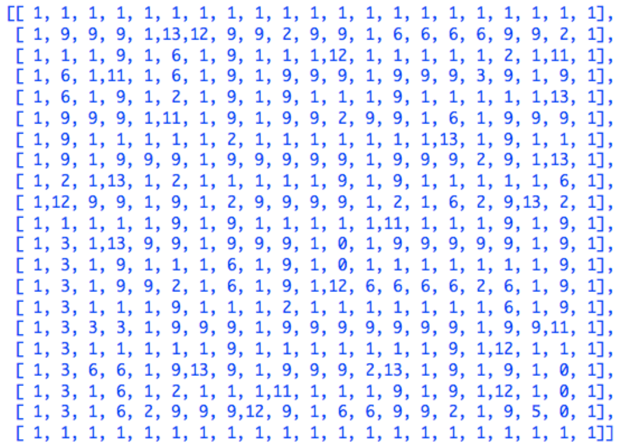
The game engine is set up so that many maze generation algorithms can be utilized the maze generation portion of the program can take a text file as an input, meaning that any maze generation algorithm in any language that produces the correct ASCII representation of a maze can be used. This makes it possible to generate mazes externally using other languages and technologies and then utilize those mazes to create game levels. The final version of the game utilizes a selected backtracking maze generation algorithm that is implemented within the program. Since the maze generation algorithm was not developed by the author, it will not be discussed here.

The generated maze is stored initially as a string array ASCII representation. Next, the generated ASCII maze is parsed into a 2D int array, in which every unique integer represents a specific tile. Immediately after parsing, the maze is empty with a 1 representing a wall tile and a 0 representing a blank tile. Next, the 2D int array is populated with different integers representing different the different elements of the game: enemies, Energy items, and portals. The population of the 2D int array is discussed in section 2.3.



```
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
 [1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1],
 [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1],
 [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1],
 [1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1],
 [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1],
 [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1],
 [1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1],
 [1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1],
 [1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
 [1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
 [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
 [1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1],
 [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

Figure 1: The maze upon generation, unpopulated



```
[[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [ 1, 9, 9, 9, 1,13,12, 9, 9, 2, 9, 9, 1, 6, 6, 6, 6, 9, 9, 2, 1],
 [ 1, 1, 1, 9, 1, 6, 1, 9, 1, 1, 1,12, 1, 1, 1, 1, 1, 2, 1,11, 1],
 [ 1, 6, 1,11, 1, 6, 1, 9, 1, 9, 9, 9, 1, 9, 9, 9, 3, 9, 1, 9, 1],
 [ 1, 6, 1, 9, 1, 2, 1, 9, 1, 9, 1, 1, 1, 9, 1, 1, 1, 1, 1,13, 1],
 [ 1, 9, 9, 9, 1,11, 1, 9, 1, 9, 9, 2, 9, 9, 1, 6, 1, 9, 9, 9, 1],
 [ 1, 9, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1,13, 1, 9, 1, 1, 1],
 [ 1, 9, 1, 9, 9, 9, 1, 9, 9, 9, 9, 9, 1, 9, 9, 9, 2, 9, 1,13, 1],
 [ 1, 2, 1,13, 1, 2, 1, 1, 1, 1, 1, 9, 1, 9, 1, 1, 1, 1, 1, 6, 1],
 [ 1,12, 9, 9, 1, 9, 1, 2, 9, 9, 9, 9, 1, 2, 1, 6, 2, 9,13, 2, 1],
 [ 1, 1, 1, 1, 1, 9, 1, 9, 1, 1, 1, 1, 1,11, 1, 1, 1, 9, 1, 9, 1],
 [ 1, 3, 1,13, 9, 9, 1, 9, 9, 9, 1, 0, 1, 9, 9, 9, 9, 9, 1, 9, 1],
 [ 1, 3, 1, 9, 1, 1, 1, 6, 1, 9, 1, 0, 1, 1, 1, 1, 1, 1, 1, 9, 1],
 [ 1, 3, 1, 9, 9, 2, 1, 6, 1, 9, 1,12, 6, 6, 6, 6, 2, 6, 1, 9, 1],
 [ 1, 3, 1, 1, 1, 9, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 6, 1, 9, 1],
 [ 1, 3, 3, 3, 1, 9, 9, 9, 1, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9,11, 1],
 [ 1, 3, 1, 1, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 9, 1,12, 1, 1, 1],
 [ 1, 3, 6, 6, 1, 9,13, 9, 1, 9, 9, 9, 2,13, 1, 9, 1, 9, 1, 0, 1],
 [ 1, 3, 1, 6, 1, 2, 1, 1, 1,11, 1, 1, 1, 9, 1, 9, 1,12, 1, 0, 1],
 [ 1, 3, 1, 6, 2, 9, 9, 9,12, 9, 1, 6, 6, 9, 9, 2, 1, 9, 5, 0, 1],
 [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

Figure 2: The maze after population

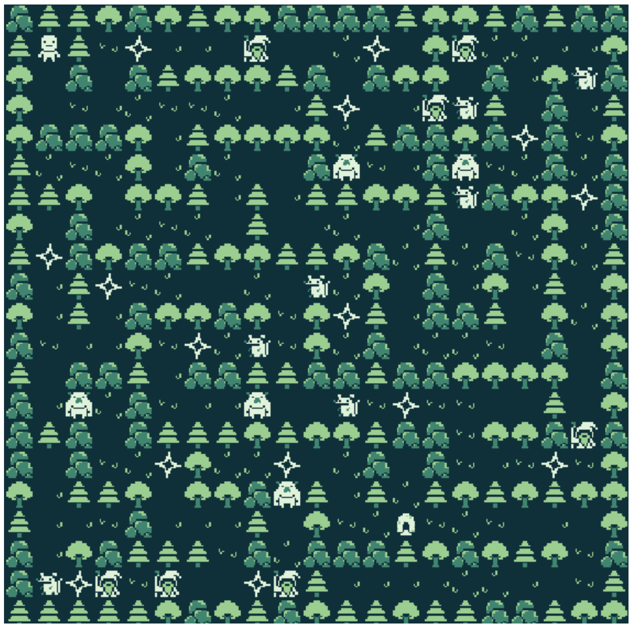


Figure 3: The generated maze represented graphically

2.2 Tile Mapping and Collision Detection

After the 2D int array is populated with items, it is iterated through to create the graphical representation of the generated level. As the iteration looks at each integer in the grid, a MapTile object is created for the specific element that the integer represents and placed at the same location in a 2D MapTile array, called the TileGrid. The MapTile object stores the type of the tile, its coordinates, along with any specific information for its particular type. The player itself is stored as a MapTile object, which is used for collision detection.

The collision detection is done by comparing the player's MapTile's position (which is stored as x,y coordinates) to the MapTile at the coordinates of the desired move space on the TileGrid. There are 5 types of collisions possible, and are handled as follows:

```
if the MapTile is a wall, the player cannot move to it
if the MapTile is a blank space:
        the player can move to it
        1 Energy unit is removed
if the MapTile is an Energy item:
        the player can move to it
        X Energy units are added
if the MapTile is an enemy:
        the enemy's HP is removed from the player's Energy
        if the player's Energy is now less than 0, the player dies
```

```
                else the player can move to it
        if the MapTile is a portal:
                the player can move to it
                the level is completed and a new level is generated
```

Since the collision detection is done entirely through the TileGrid, it eliminates overhead from checking for collisions in the graphical space. All changes to the game's map (player movement, addition/removal of items) are done on the TileGird array level first and then simply drawn onto the screen, removing the need for physical collision detection with hit boxes or colliders.

2.3 Maze Solving and Item Distribution

The central challenge of the game's procedural generation is to place Energy items along the optimal path to the end point, therefore ensuring that the player will always have enough Energy to reach the end. Enemies also must be placed along the optimal path and factored into the net gain and loss of Energy as the player moves through the maze. The player loses one unit of Energy with every 1 square that is moved, so distance between each Energy item and the next must be tracked to guarantee that the player does not run out of Energy before replenishing it.

However, to add a degree of difficultly and depth of gameplay, the Energy items are not placed linearly on the optimal path. This forces the player to explore the maze and pursue potential dead ends while utilizing Energy for the payoff of an Energy gain. Therefore, the procedural generation is a step above a maze that is simply solved from point A to point B instead, the algorithm intentionally results in the illusion of being non-linear for the player.

2.3.1 Primary Path

First, the maze is ”solved” using a backtracking algorithm which generates the optimal path, which is called the primary path. The primary path is stored as a list of coordinates. The backtracking solving algorithm uses a recursive function which iterates through coordinates and explores neighbors in a clockwise direction. If a neighbor is blank and has not been visited before, the recursive function is called on the neighbor and so on until either the end point is found or the path hits a dead end (meaning all neighbors are either walls or have been visited previously) after which the recursion bottoms out and proceeds to the next set of coordinates in the primary path. The attempted paths that resulted in a dead end are not included in the primary path, so the resulting primary path is a simple path from the starting point to the ending point that solves the maze.

2.3.2 Secondary Path

Next, a second divergent, or secondary, path is determined using the primary path, which includes the addition of sidetracking paths that lead from the primary path toward a dead end and then back. The secondary path is where the non-linearity is introduced into the level generation. First, the algorithm loops through the list of primary path coordinates and looks for "junctions". Junctions are coordinates where it is possible to move to a neighboring blank tile that is not part of the primary path. After a junction is found, a modified version of the recursive backtracking algorithm is used to create an "out-and-back" path – the junction is pursued for an x amount of steps out and back to create a branch from the primary path. This branch is then added to the secondary path. This is repeated for all junctions and the result is a branching, non-linear secondary path. With this distribution approach, the player must not only solve the maze linearly but also discover the necessary sidetracking paths in order to maintain Energy levels.

2.3.3 Distribution of Enemies and Energy items

Finally, items are distributed by the algorithm by looping through the coordinates in the secondary path. The loop mirrors the player's ideal traversal of the maze, and distributes enemies and Energy items as such by keeping track of the player's total Energy used at every set of coordinates. Each set of coordinates can be either a blank spot, an enemy, or an Energy item.

The loop places enemies at random by using a random number generator; for example, if a 5 is rolled, the algorithm attempts to place an enemy at the current coordinates in the loop. It does this by generating a random amount of hit points for an enemy and then checking if the player has the equivalent or more Energy at that point in the maze. If the player does, it is determined that the player can move past that enemy and the enemy is placed at those coordinates. That enemy's hit points are added to the total Energy used. If the enemy cannot be placed, the current coordinates are treated as a blank space and 1 unit of Energy is added to the total Energy used. Whenever the total Energy used is equal to the player's total Energy, an Energy item is placed (since at that set of coordinates, the player will be out of Energy and it will need to be replenished).

2.4 Technology Used

The game was written in Python using the Pygame library. Python and Pygame were chosen due to the author's familiarity with them and also due to the simplicity of utilization of the Pygame library. Python enabled for rapid prototyping and easy visualization of game levels.

The game was initially started development in the Unity 2D engine with C# scripting, but the author determined that too much overhead was presented by Unity's graphical UI. Next, the author attempted to write the game in Java using the Swing library, which presented difficulty in drawing the necessary components on the screen easily. The graphics for the game were made with the online editor Piskel. A portion of the graphics were drawn from the Scroll-O-Sprites sprite pack and modified and recolored (svh440, 2013).

3 Related Work

3.1 Traveling Salesman Problem

Determining the placement of Energy items is similar to the Traveling Salesman Problem, in which an entity must visit a certain amount of locations that are a certain distance away from each other in the shortest amount of time (Rajesh Matai, 2014). Although the traveling salesman problem has a different set of constraints, it was still used as an aid to develop the Energy item mechanic. The traveling salesman problem has variable distances between nodes with a constant number of nodes whereas the Energy item distribution has a fixed distance between each node but a variable number of nodes.

3.2 Sokoban and Puzzle Games

Puzzle games in which each level is presented as a static environment were an influence for the game. One particular game that consists of unchanging levels the popular Japanese game Sokoban. Sokoban involves the player pushing around boxes in a grid-based level in order to attempt to create a path to an end point. Although Sokoban levels are traditionally made by hand, some work has been done on attempting to procedurally generate them by hand (Joshua Taylor, 2011). The work by Taylor and Parberry aims to ensure that every generated Sokoban level can be solved, which is the same motivation as this project.

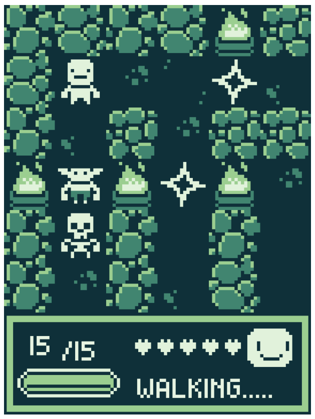
3.3 Stamina in Video Games

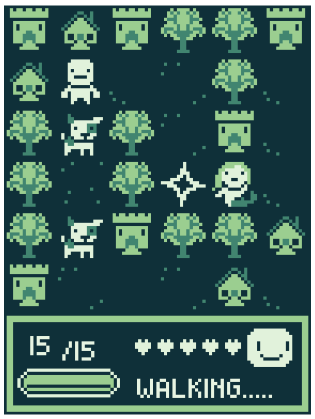
Many video games used the concept of ”stamina” or a measure of finite energy in order to set gameplay constraints. One such game is The Legend of Zelda: Breath of the Wild, in which the player has a limited amount of stamina which prevents them from reaching certain areas until stamina is increased with items (Nintendo, 2018). The use of finite energy, when implemented correctly, is a dynamic way to impact the player's navigation through a level or environment. The Legend of Zelda: Breath of the Wild is set in an open-world environment and thus is not as constrained as the game for this project, but the effects are similar in both games.

4 Results

The final game consists of five distinct level types along with a title screen, player select screen, and end game screen. Each of the five levels have a different cosmetic appearance and contain different enemies and environment tiles. Varying levels were included to showcase different types of graphics and to add an element of randomization to the game progression, as the type of level is randomly selected when the player moves to a new level. The title screen includes the name of the game along with several sprites used in the game levels. The player select screen allows the player to choose one of two characters. Finally, the end game screen shows the player some statistics about their particular play-through, which include number of levels completed, total number of enemies beaten, and maximum Energy capacity.

The goal of each level, as stated previously, is to get from the start point to an end point which is marked with a portal tile. The player is able to play each level five times before dying, so the goal of the game is to beat as many levels as possible in sequence without the game ending. A heads-up display (HUD) is shown during gameplay that contains useful information for the player, including current and total amount of Energy (shown both numerically and as a graphical bar), number of attempts remaining, and a status both in text and emote form.





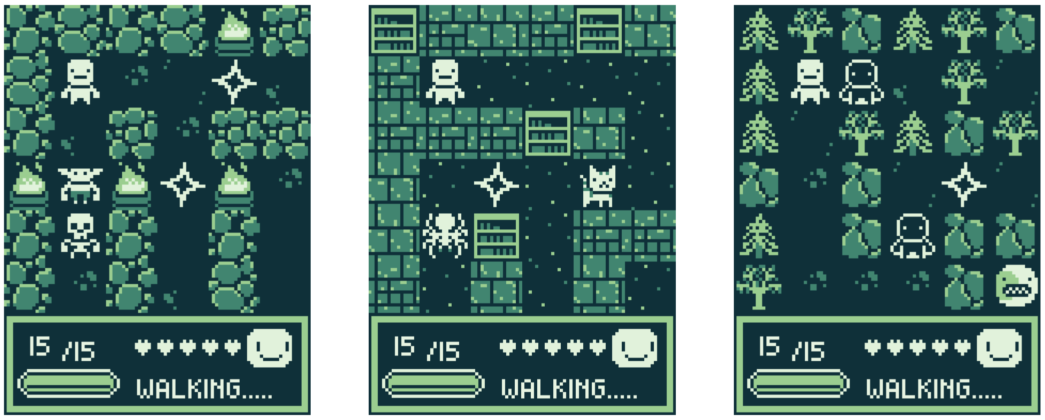


Figure 4: Examples of generated levels

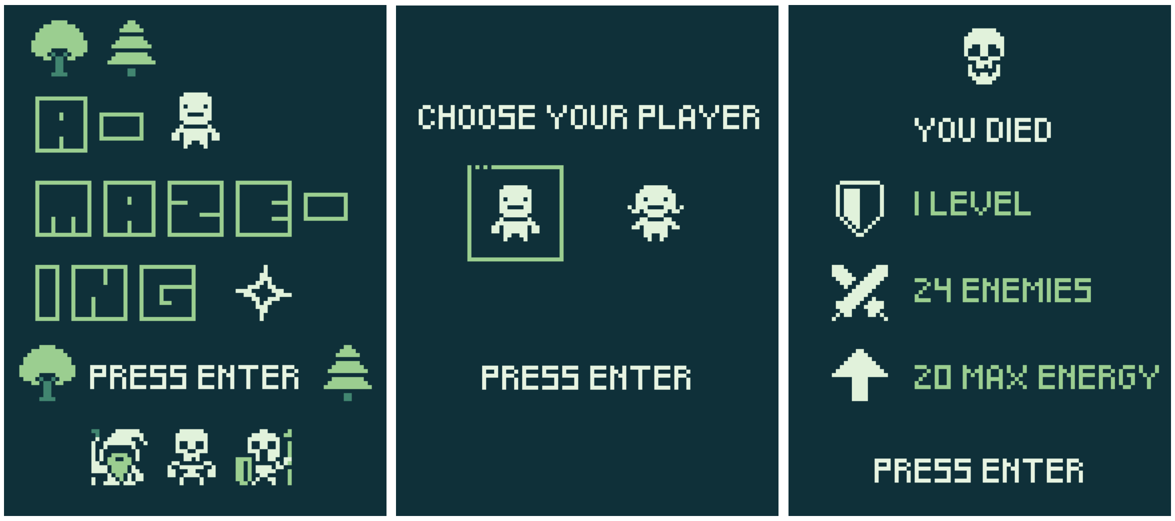


Figure 5: Title screen, character select screen, and end screen

5 Discussion and Future Work

Ultimately, it was possible to develop a set of algorithms that generate levels based on finite, numerical information. Because the algorithm populates each level by keeping track of exact amount of player Energy along with the distance in tiles between each item, it is possible to ensure that following the secondary path will allow the player to beat each level every time. However, the clarity of the secondary path is not apparent to the player, which adds the gameplay aspect and makes the game interesting for the player.

As future work, the author would like to port the game to a mobile platform by rewriting it in Java and publishing it to the iOS App Store. Before publishing, the author would like to make improvements such as level transitions, minor sprite fixes, and the addition of sound effects and music.

References

Joshua Taylor, I. P. (2011). Sokoban generator. Available via the Web, http://larc.unt.edu/ian/research/sokoban/. **Paper on procedurally generating Sokoban levels.**

Nintendo (2018). The legend of zelda: Breath of the wild. Available via the Web, https://www.zelda.com/breath-of-the-wild/. **The website for the game.**

Rajesh Matai, Surya Singh, M. M. (2014). Traveling salesman problem: an overview of applications, formulations, and solution approaches. Available via the Web, https://www.researchgate.net/publication/221909777_Traveling_Salesman_Problem_an_Overview_of_Applications_Formulations_and_Solution_Approaches. **Paper about the traveling salesman problem.**

svh440 (2013). Scroll-o-sprites. Available via the Web, https://imgur.com/a/uHx4k. **The sprite sheet I used.**