Overview

When starting the game, the player sees the intro with falling rocks in front of a cave opening and the player can choose from two different levels. The player is taken to cave and must find an exit without dying in order to win the game.

Requirements

MazeWars was designed to be a 2D top down shooter game. The player must survive mobs inside the cave and be able to gather all three artifacts in order to win the game. We used the asteroids.cpp program as the main framework for our game. The player controls the shooter and their objective is to kill enemies through the cave, collect artifacts, and try to find an exit to complete the game.

The player has a limited amount of bullets that is measured by an ammo bar. They also have a health bar similar to the ammo bar. If the health bar becomes empty, the player loses one life out the four lives the player start with. The monsters have a fixed walking pattern, and when the
player comes close to the monster, the monster starts to pursue the player. If the monster touches
the player, the player loses their health. If there are multiple monsters touching the player, the
player’s health depletes faster. If the player loses all their lives, the player is prompt with a game
over screen advising the player failed and can restart the game. To complete the game, the player
must have all three artifacts. The player is then taken to a parallax screen where the player can go
left and right, watching the names of the creators of the game with a beautiful scenery of
mountains, clouds, and terrain and the game ends by showing the statistics of their game play.

Design

Our software development was based off of agile and extreme programming. We used the
asteroids framework Professor Gordon gave us as a basis for our game. We had many design
ideas we wanted to create. For example, we wanted features such as speed boost and power ups.
Through the whole process we would code our design and build piece by piece after that.

Gordon's asteroids framework.

Next, we removed the asteroids features in his framework to have a beginning framework
for our game to build off of. We started working on character control, and kept the WASD and
Cameron added mouse controls. During this time, Matthew worked on adding controls for an Xbox controller and later Cameron made the control menu.

Matthew’s Xbox Controls. Cameron’s Control Menu.

After the controls were done, the GUI was created for the game. Cameron made the design of the HUD. Nearing the end of the 10 weeks, our design for the HUD changed.

Beginning HUD design. Final HUD design.

Then we focused on the enemy design and what they would do. When the enemy is near the player, the player gets hurt and their health decreases.
The enemies.

The maze design took the longest for our game project. David used the cellular automaton to generate our maze. The cellular automaton follows a set of rules for the cells to follow. Each square is called a “cell” and each cell has two possible states, black and white, which are walls and floors for our game.

David's cellular automaton.
For our final design for our game, we designed more features, such as sound, special keys, credits page, and etc. to the game play.

Game Menu. Special key for ~.

Implementation

Coding

When writing the code, our team mostly used Extreme Programming. Our team would develop small designs for the game so we won't focus on a certain area for too long. Some of us would also participate in pair programming to help each other understand certain logic of the code and revise the code to prevent any errors from occurring. We had several guidelines in order to have good readability of the code and tried to make sure to have good naming conventions.

Coding Guidelines:

• One of the major guidelines for coding was using the K&R style to have high readability, spacing, and consistency throughout the code.

• Variable names should have a lot of meaning and self-documentation so it can be
easily understood by other programmers.

- Function names should describe their behavior as explicitly as possible
- The code itself should be as self-documenting as possible to avoid distracting and redundant comments. The comments should be used to describe snippets of code that isn't as easy to understand.

Testing

During the course of the project people would push code that would break functionality. I took it upon myself to fix a lot of issues in the code many times throughout the project. We had issues where there was a black box rendered around the character, an issue with the way the enemies reacted to the player when using the Xbox controller, and an issue where the button presses were being toggled multiple times when trying a button press. To solve these issues I took what I learned from Alex’s presentation and tried to test our project accordingly. The biggest help from Alex’s presentation was when he said to narrow down what you’re testing, that way you’re only testing a specific feature at a time. This helped tremendously when there were bugs nested into another group member’s source code that are affecting code in a separate place in the code.

Git/GitHub Source Control

Nobody in our group had ever used GitHub, or any other form of source control, before this project. One of our biggest issues using GitHub is that myself, and a few other members of our team would write code and not push it immediately, or we would be editing files at the same time. Then when it came for us to upload our code, there would be many merge conflicts that we
didn’t know how to solve other than by cloning a new copy of or code, and copy/pasting our changes into the new clone before re-pushing the code. This gave us issues a few times as some members would unknowingly copy/paste and push old versions of the code that would overwrite previously pushed changes.

**XWindows and OpenGL graphics**

I enjoyed using OpenGL throughout this project. It was fun to do research on how exactly to accomplish a particular task in OpenGL and trying different things to get it working. A few times this would suck when I was researching a certain task in OpenGL but only finding code examples that applied to a library, such as GLFW. One thing our group couldn’t figure out how to do, was render objects that had semi transparency. We wanted to create a fog of war effect that would only unveil parts of the map that the player visited, and only illuminate the immediate area surrounding the player. We couldn’t accomplish this the way we wished because we couldn’t figure out the semi transparency on rendered objects. Our first “solution” to the fog of war problem, was that our team member Cameron, wrote code for a diamond that displayed the area around the player. It effectively solved two problems, limiting the player’s viewport as well as covering up an issue we had where the textures at the ~10% edges of the screen would pop-in textures. Later in development another team member, David wrote code to essentially just extend the HUD to surround all of the edges of the screen, almost as if it was the edges of our
playable character’s helmet. This looked drastically better than the diamond rendered previously and made our HUD look as if it was part of the character’s helmet.

Parallax

The design of the parallax ending fixes the player at the center of the screen while the images in the background move at different speeds to simulate the player's movement. The character has a running animation and a resting pose and there are two crates which the player can interact and manipulate. Implementing the interaction with the crates proved to be fairly simple, it required finding only the two edges of the crates not touching the ground or pointing up and testing if the player's position was within the boundaries and if it was then the player's movement Boolean was false, if the player jumped on top of the box another Boolean would turn true and the player could only fall so far down the screen while they are within the two outer boundaries of the crate, for the crate you could move, the logic was almost identical but instead of restricting movement of the player altogether, the players speed is cut in half and added to the position of the crate as well to make it appear as if the player is moving the crate. The wooden crate can move as far as the player in either direction and is interact able at every point in the level because the collision is based on the relative position of the crate to the player.
Delivery/Maintenance

The game is run fully compiled with three folders including /images, /parallax, and /sounds.

It can be delivered in a package as small as 20Mb uncompressed. Maintenance on the code and project would be performed by all members of the team and done regularly, because we had an unusual file structure the maintenance might be a little more difficult if it were to be handled by different programmers without extensive documentation.
Implementation

The game concept of a maze crawler introduces some abstractions, such as two-dimensional game space, and a variety of objects that require rendering and interaction with each other. To achieve this, a simple engine was created to handle game events and object manipulation.

Several structures were created to describe each object in the game. One base struct, common to all, was “Stats.” Movement handling, rendering position, and velocity were all tied to each objects’ Stats member, which allowed a uniform handling of every object across the board.

In the beginning phase of development, many objects were referenced by pointers; this complicated the handling of movement and rendering. To solve this, template functions were created to handle special events, in the case that interaction occurred between a pointer and a value by reference, and vice-versa.

Game position was implemented using an array of three members: x coordinate, y coordinate, and z coordinate (where z was unimplemented). The translation from game position to screen position was dependent on the focal point of the game camera. Here, the camera was fixated on the player. By checking if the game position of an object was within the screen resolution of the camera, by checking if the x and y coordinates were within a certain game distance, the rendering would correctly display the game space and all objects in it.

The map generation was the most difficult implementation. The original requirement was to create a start and end-point of the Maze in which the player would visit a random number of rooms, hallways, and other parts of the Maze that he or she encounters enemies and objectives. The size of the map is determined by game length, and from there it randomly generates a set of specifications to create the number of hallways and room types within the maze. However, my difficulty was with discovering a suitable algorithm for achieving these results. My first attempts included a recursive function which would backtrack “events” (known as DFork in the code) to retry attempts to reach the endpoint; this ended in heavy calculations, though, due to the randomness of the event generation.

I abandoned the original requirements and adopted a new random generation technique – cellular automata. By adopting a set of rules, each “cell” in the two-dimensional array would adopt a value based on neighboring cell values. I seeded the cell values by randomly generating “hallways” in the initial two-dimensional array, before applying the filter based on the set of rules. This filter occurs for each cell in the array, then return all the values to a “new” array acting as a “step” through the automation. I repeated this process one hundred times to smoothen out each array, giving every map a new, random area for the player to travel through.