Maze-Man

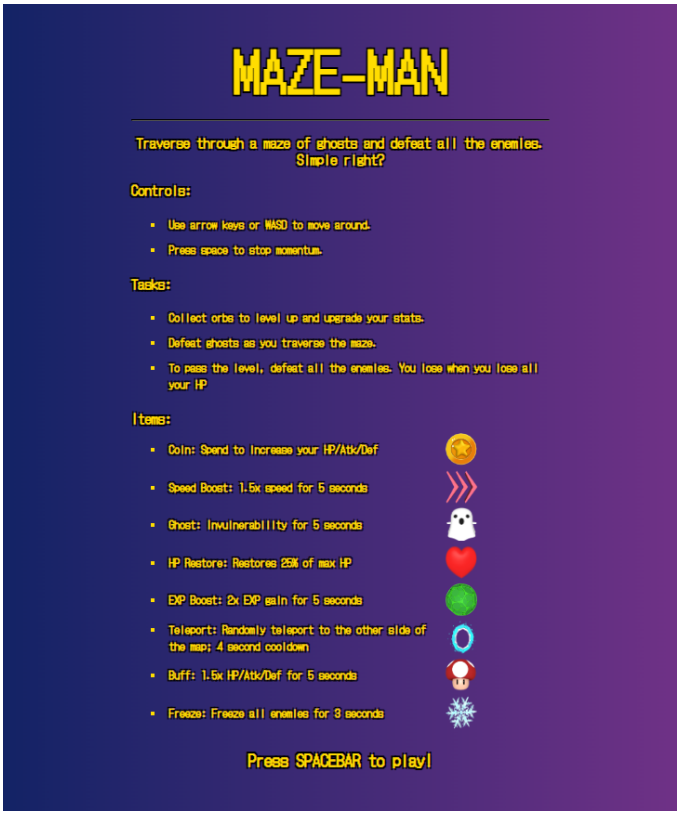
Daniel Park, Brian Lou, Bryant Zeng, Roshaan Khalid

Abstract

Our project is Maze-Man, an isometric Pac-Man-inspired rogue-like game built using Three.js. The game consists of three levels of increasing difficulty, and the objective is to defeat all the enemies/ghosts on each level. Colliding with ghosts head-on will decrease your HP, while attacking them from behind is safe. Level up to increase your stats and collect items to power up your character!

Goals

Our MVP for this project was to create an isometric game using Three.js with a procedurally generated maze, experience points on the ground and level system, basic enemies that damage the player, and a win/lose condition with a final score. Our stretch goals for this project included an item system, boss enemies with multiple levels, coins the player could use to increase their stat points, sound effects, and an endless mode. Our goal was to create a fun and enjoyable game for people of all audiences and skill levels, while including an element of replayability so the game didn't get too boring. In Maze-Man, this comes in the form of random maze generation and item/player/ghost spawns.



Maze-Man Title Screen

Previous Work

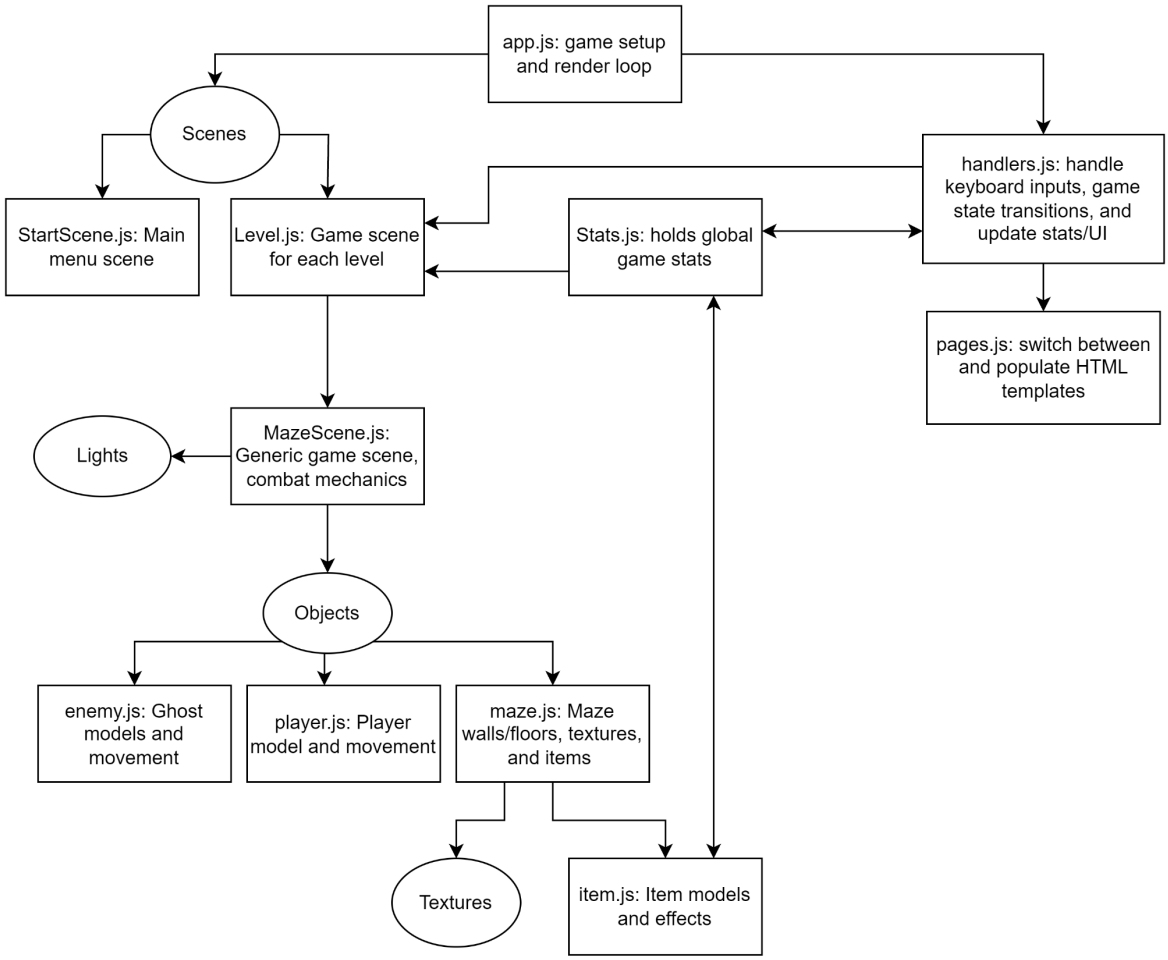
Our game is heavily inspired by the classic Pac-Man, an arcade game in which the player tries to collect all the dots in the maze while avoiding four ghosts. There are power-ups the player can collect in order to scare off the ghosts and defeat them temporarily. We adapt the basic ideas of Pac-Man, including the maze, ghosts, and dots, to create something entirely different. Instead of running away and collecting all the dots, we created a rogue-like game in which the player has to level up their stats and defeat all the ghosts. The ghosts can only safely be attacked from behind or the half side closer to the behind, so the player has the tricky task of evading the ghosts while preparing to attack.

As an isometric rogue-like game, our project also adapts ideas from other games. Isometric games like Crossy Road inspired us to adopt a 2.5D viewpoint, in which the player moves around a 2D map with a 3D perspective. This gives the player a feeling of a 3D landscape while we only have to worry about implementing features in two dimensions. Rogue-like games such as Slay the Spire and The Binding of Isaac inspired us to have multiple levels, where the focus is on increasing the player's stats/items in order to have enough power to defeat the boss. Maze-Man is an amalgamation of these various ideas, and we think they work well together to create an engaging and replayable game.

Approach

Our approach was to use Three.js as our framework to implement game elements. Our maze was represented as a 2D matrix, which contained various elements such as wall tiles and items. Our approach was to first figure out maze procedural generation all together, followed by individually splitting up to work on separate components of the game, such as the combat system, the player movement system, the item system, and miscellaneous items such as menus. We believed that this approach would work well due to the time constraints of the project. Since each of us worked on areas we were confident in, we believed this was the best approach allowing us to complete as many areas as necessary.

Methodology



Code structure

Character Movement:

To implement user controls and character movement, we utilized the fact that our map was entirely contained within a 2D matrix to our advantage. There were 2 possible implementations that we considered: (1), the method we ended up using, using strict logic to bound movement to integer coordinates, and (2) allowing any player movement but using bounding boxes and collision detection between the player and walls to restrict movement and prevent wall clipping. The disadvantage of the second approach was that there were too many possible bugs or edge cases we foresaw occuring; for example, what if the player's speed was large enough to almost completely phase through the wall's bounding box? This approach was a reactive approach, which we did not believe was the best one.

Thus, we decided to use approach (1), where we used strict logic and coordinate controls to limit allowed player movement and prevent wall clipping. To this end, we employed the following invariant: For every character (player and ghosts), one of their coordinates must

always be an integer. Using this invariant and if statements, we were able to successfully restrict movement and prevent any wall clipping.

Maze Generation

There were 3 approaches we considered to maze generation. First, we considered using wave function collapse to procedurally generate wall tiles using certain adjacency rules. However, we decided that this method was uncertain, as none of us had prior experience with such a method. Second, we considered using a convolutional approach, where we convolved windows of certain sizes over the grid, and used randomization to generate walls. However, this method had one downfall, that being that we could not guarantee that we did not have any dead-ends or unreachable tiles in our maze, which we wanted to avoid. We considered using a 1 block buffer around each of the wall tile parts to guarantee that each tile was reachable, but this would have resulted in 2 block wide paths.

Lastly, we ended up with our 3rd approach: to generate the maze, we adapted open source code from pacman-mazegen, an open-source project that procedurally generates 28 by 31 Pacman maps [1]. We modified some parameters and features of this generator code to better suit our project. Primarily, we (1) expanded the map size, (2) removed the middle ghost spawning box, and (3) removed exterior tunnels which normally connect both sides of a Pacman map.

Collision Handling

To handle collisions / attacking enemies, we used bounding boxes (Box3) in Three.js which encapsulate the player and enemies. Every frame, we check if the player's bounding box intersects with the enemies' bounding boxes, and if it does, we handle attacking logic using a combination of the player and enemy's coordinates and direction of movement. This approach was the most straightforward and obvious implementation, as there are no obvious downsides in terms of computational complexity or implementation difficulty.

Map Levels

To implement multiple map levels, we abstracted our code to use a Level() class, which allowed us to customize the enemies' stats as well as recreate multiple mazes for each level. Additionally, this abstraction allowed for us to use different wall / floor textures for each level. This was the most straightforward and obvious approach, and is what we decided to use. handlers.js makes sure levels get generated and set to the scene when necessary.

Map Loading

We try to load each level in the menu, which gives a clean transition to the scene when the game is started. However, in the case of moving to the next level, we add a small delay to load the scene before the countdown starts. This delay is a constant, however users may encounter a black/partially loaded screen or a bit of gameplay before the cooldown starts, depending on the computer. We also encountered some problems with lag and character clipping, as map loading seemed very variable, especially when many other processes

were running on a machine. We experimented with a variety of constants and optimizations to the scene, and we think the game is fairly smooth on a decent computer, though we weren't able to fix these issues entirely.

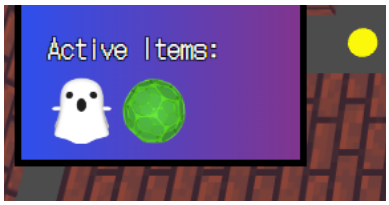
Stats

We tracked several player stats, such as health, damage, defense, maxHealth, player and enemy movement speed, as well as other non-player specific stats such as score, map level and number of enemies defeated. All stats were kept in stats.js, which houses a number of global objects. BaseStats is the player's starting stats, BonusStatsFromLevels is the stat boost per attribute per level, BonusStatsMisc comes from coin spending and items, StatsMultipliers comes from items, and Stats contains the final calculation from each of these objects. This file also houses the enemy's stats and increase per level, which is used in level generation.

Keeping the stats in independent, global objects and updating them in handlers.js was a great way to modularize our application. As seen in the code diagram, stats.js is used by Level.js, handlers.js, and items.js, and this decouples the application logic in a way that allowed us to work on our own tasks seamlessly.

Items

We implemented nine different types of items, each with unique appearances and effects. To implement items, we used 3D models found online to create Three.Object3Ds for each type of item, which were then rescaled to fit in a tile. To add rotation, we added each object to the parent maze scene, which calls the update() function on each item to rotate it slightly in each time-step. On maze generation, each item is randomly generated in an available square, starting with teleporters and ending with exp orbs for any free spaces. The maze object keeps track of which item is at each location, so that when player movement is updated, we can get the item at that location and "collect" it by applying its effect and turning it invisible.



Active Items Displayed

Each item has a different effect, but most serve to temporarily increase the player's stats. This is done by modifying Stats and StatsMultipliers in stats.js, which are global objects used by the UI and combat system. When multiple of the same item are collected at the same time, the expected behavior is to increase the item's duration. We do this by keeping track of ActiveItemCounts in stats.js, and the item's effect is only removed/applied when this count increases from 0 or decreases to 0. We also use ActiveItemCounts to update the UI with the player's active items. One notable item is teleporters, which are generated

at every dead-end in the map and randomly teleports the player to the other side of the map. Each teleporter comes with a 2-second immunity period and 4-second cooldown so they can't be abused by the player. This logic is implemented with JS's setTimeout() and ActiveItemCounts.teleport, which is set to 2 when the player is immune, 1 when teleporters are on cooldown, and 0 if not in use.

We also implement other logic, such as clearing items on game win/lose/quit/restart and between levels. Creating Item objects that contain the 3D model and its state worked well for us, but the logic is kind of hard-coded. Each item has a type property, which is a string describing what item it is. Each of the item functions has a large switch statement which does something different to each item based on its type string, and this might be hard to maintain as the item list grows larger. We considered separate classes for each item, but thought that was too much boilerplate, and our method ended up being very successful in the end.

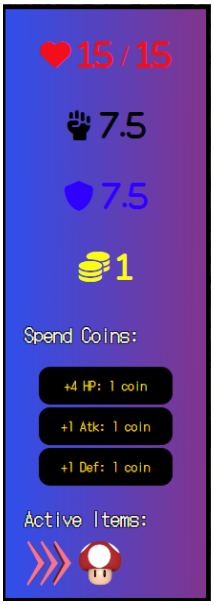
We also wanted to add visual effects depending on the type of item collected (ex. for ghost, turns the player opacity down). However, the player model's opacity made no difference to its appearance and we ran out of time to implement other effects. We did manage to add a basic sound effect for collecting items, however.

UI

For the UI, we implemented several menus: a main menu, the actual game screen, a pause menu, a countdown popup, a lose screen, a win screen, and a next level screen. We transitioned between the different screens by modifying the DOM, much similar to previous year's *Glider* from COS426 gallery. To keep track of the screens, a dictionary of menu names as keys and booleans as values is used. When the screen is being displayed, its value is set to true. Screens change based on keypress, most of the menus are triggered by pressing 'spacebar' but the pause menu is activated by pressing Esc or 'p'. However, when the pause menu is up, there are buttons which when clicked either resumes the game, restarts the level, or exits back to the main menu screen. Also, the ending screens, such as lose, win or next level are handled based on the player's stats. For example, losing all health leads to a lose game screen. Event handlers were used to listen for certain keypresses and mouse clicks on certain buttons to transition to different screens.

Here's a quick runover of how the menus are handled: To begin, the game starts on the main menu screen, however, when space is pressed, the screen becomes the game screen, along with the countdown popup, which appears only for three seconds. The countdown popup appears every time the game screen is loaded. Then if the player loses all their health, then the lose game screen appears. If the player manages to defeat all the enemies, then the nextLevel game screen appears. Finally after beating three levels, the player goes to the win screen.

From left to right: main menu, countdown, game screen, pause menu, lose, next level, win screens



Statbox

On the game screen, all the stats and items are displayed on the upper left hand side in the statbox (shown on the left). Player stats such as health, attack, defense and coins are paired with font awesome icons. In addition, a shop menu allows for players to buy stats using coins. These buttons are handled by event listeners which are initialized when the game screen is activated. Active items are also displayed when the players collect them.

Scoring:
Similar to Pacman, we implemented the score to be the number of yellow dots collected plus the HP of the enemies killed. In addition, score could be translated into exp, which can level up the player. Score is not displayed on the screen, however the equivalent exp is a green exp progress bar that displays the current level of the player and the progress until the next level is reached.

Exp Bar Display

We also show the enemies remaining in the level right below the exp bar so that the player can keep track of progress among the level.

Audio
Our game includes background music and other sound effects, such as for Game Over, Game Win, Player Level up, Map Level pass, Item pickup, and Enemy death. To handle Audio, we downloaded royalty-free audio files and used Three.js's Audio classes to play the sounds. Sound effects were added in the home stretch of the project (last day) so we were unable to add more sound effects or settings to adjust the volume or mute the sound.

Results
We measured success in terms of the original objectives we created, the number of bugs present in the final game, as well as the overall player experience. To evaluate our project and perform experiments, we did trials of the game, debugging and walkthroughing as we progressed through the development, making sure everything was working correctly and the results were as we intended. In the end, our final game looks beautiful and has everything as we originally planned and drafted. All the features, looks, and menus behave just as we intended and there are no apparent bugs, except for choppy movement when the computer lags (which is somewhat expected of a 3D game). But overall, these results indicate that our project was a resounding success.

Discussion and Conclusion
Overall, our approach proved to be very successful. Our game looks and behaves better than originally planned in our bare-bones MVP, with smooth player movement and randomly generated textures for each level. As a team, we learned a lot from building this game, gaining experience in game development using Three.js and computer graphics techniques. Moreover, we were able to successfully communicate and work efficiently in a monorepo without too many conflicts.
Future work includes improving upon the path-finding algorithm so that the npcs can take the player's location into account and have unique pathings. In addition, more variety

among the enemies could allow for a more enriching experience. The maze generation could also be improved upon so that the levels have more variety in maze design, and performance optimization could be another consideration.

Contributions

Daniel

At the start of the project, I worked with Roshaan on creating the initial maze scene using Three.js boxes and textures. My most impactful contribution came in the form of the item system, which I implemented from start to finish. I'm very satisfied with the item models and effects, which are bug-free and impactful to the player. I also made key contributions to other areas of the project, especially in handler.js (handling game state transitions and loading), as well as in the UI by adding elements and linking them together to our backend (ex. active items, coin GUI, enemies left counter).

Bryant

At the start of the project, I helped Brian on researching and analyzing and testing different possible procedural maze generation algorithms, eventually settling on the tetris piece method. However my main contribution was creating the UI for the game. I worked on the framework for handling DOm elements. I also created most of the different menus, such as the main menu, lose, win. For the actual game screen UI, I worked on the exp bar and pause menu. I handled the logic for transitioning between different pages, handling both keypresses as well as mouse clicks for the pause menu.

Roshaan

In the beginning of the project, I did the initial rendering of the map which was procedurally generated. Furthermore, once the map was created, the next step was to render npc ghosts which I added and gave them movement so that they could roam around the map, changing directions as they encountered obstacles. Additionally, I chose music and added background music to the game.

Brian

At the start of the project, I worked with Bryant on procedural maze generation. After we got that working, I worked on player movement and controls by myself. For this, I implemented smooth player turning. After this was completed, I worked with Roshaan to help adapt my player movement code to enemy NPC movement. Next, I worked on the combat system and collision detection with enemies, and also worked with Daniel to integrate player stats with items. After this was done, I worked on creating multiple levels by abstracting our code and adding more wall textures. Near the end, I helped out with music and collaborated closely with all of my team members to put the final game together.

Works Cited

See ReadMe for the full list
[1] https://github.com/shaunlebron/pacman-mazegen