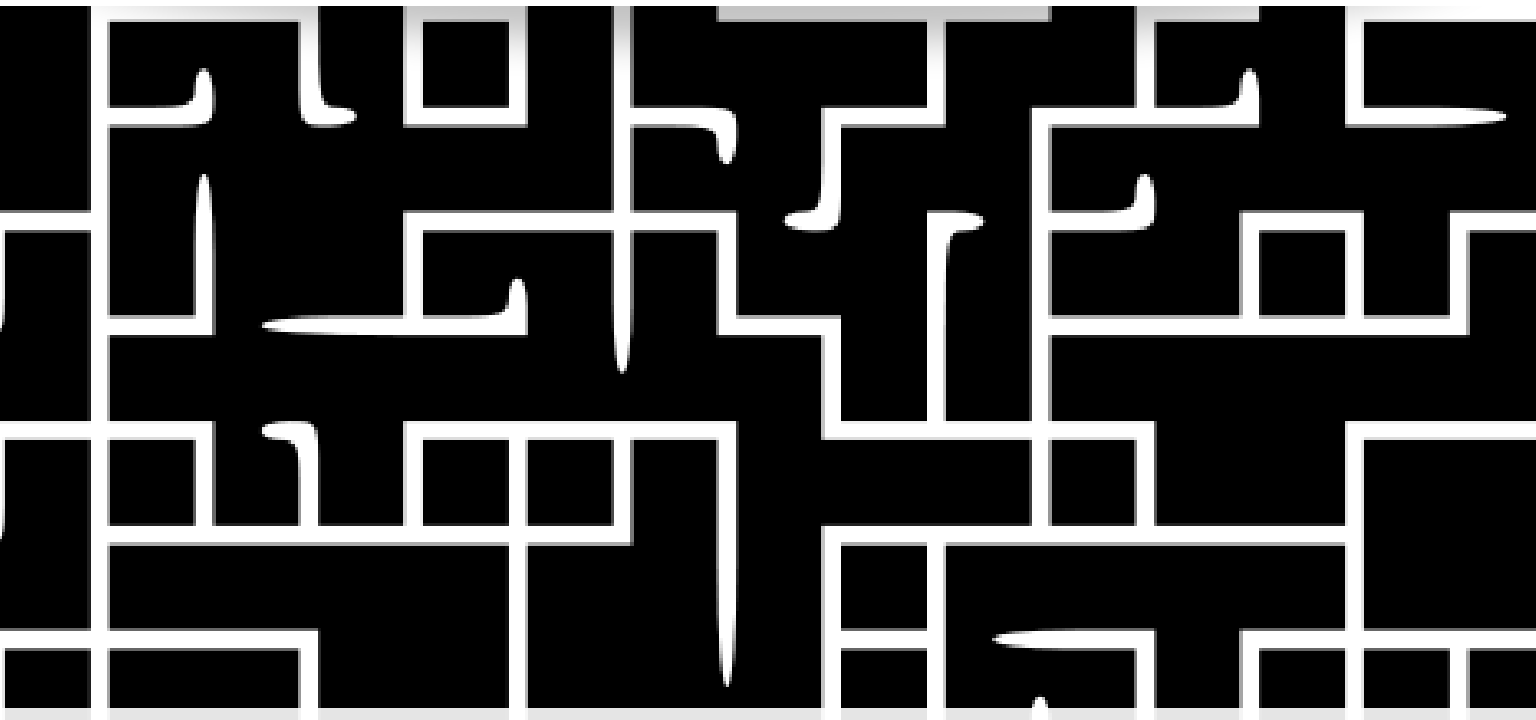
MAZE RUNNER



MOLLY O'CONNOR
ANDREW DENEGAR
NICK CLOUSE

TABLE OF CONTENTS

User Section

TABLE OF CONTENTS

Developer Section

Manual

OUR VISION

“

Maze runner is a gripping puzzle/action game for teens or young adults that blends simplicity and depth resulting in a suspenseful and strategic gaming experience.

”

---------- Controls ----------

Keyboard Commands

Move Player Upward

Move Player to the Right

Move Player to the Left

Move Player Downward

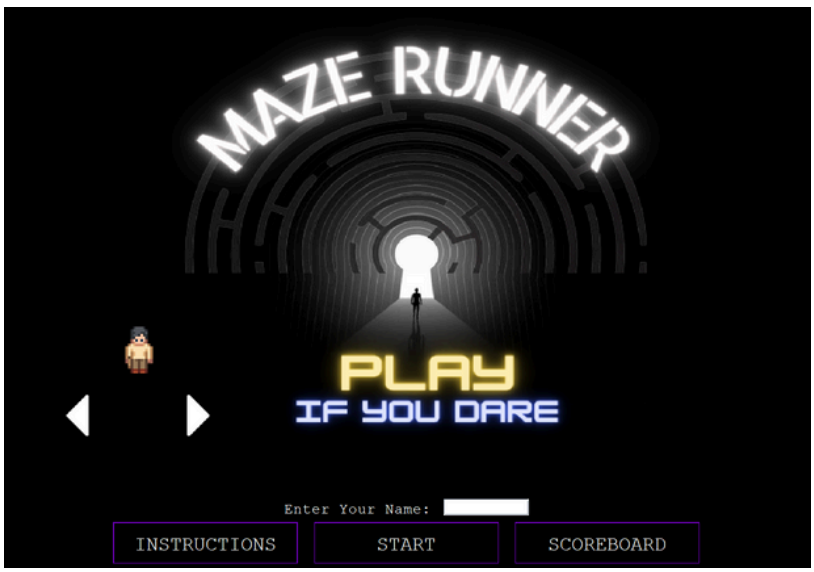
Attack

-------- How To Play --------

Homescreen

In this game, the objective is to navigate through three challenging levels of mazes, find the exit, and escape before the time runs out. Here is how it works!

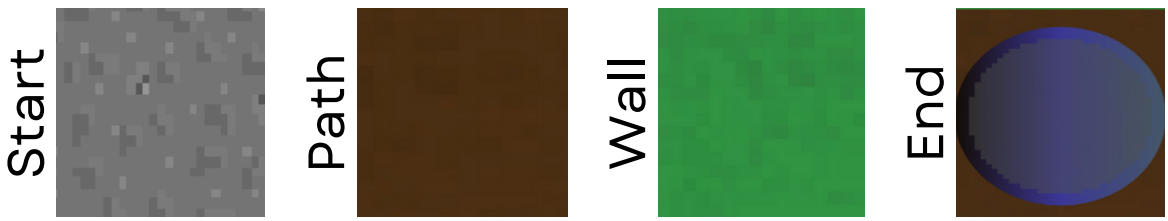
This is the game homescreen, listed are three options, instructions, start, and scoreboard. If you choose “INSTRUCTIONS” the basic rules to the game will be displayed for the user to understand. If “START” is clicked, then the first level is loaded and will begin. If “SCOREBOARD” is clicked, a list of the best maze scores and the player who received the time will be displayed. Before you can start the game, a player name must be entered that is less than 10 characters long. You also have the option to choose different character skins.



Levels

Once a player starts the game, they are directed to level 1. Each category of level (1,2,3) has 5 versions, this makes it so the player can't memorize all the mazes (unless they have way too much time on their hands). Both pictures above, are level 1, just a different version. Each time a level is started the player will be loaded on the "start block" and have to navigate the maze until they find the "end block". There are a total of 3 levels, with each level increasing in difficulty. Level 1 has the square dimension of 20x20 blocks. Level 2 is 30x30, and the final level is 40x40 blocks big. As the player progresses through the game, the maze increases in size.

Types of Blocks



Timing

For each level the player has 2 minutes to complete the maze. The time will not start until the player presses an arrow key and a player movement is detected. After the time starts the player can see there remaining time in the top left corner of the screen .

Initial Time

Updated Time

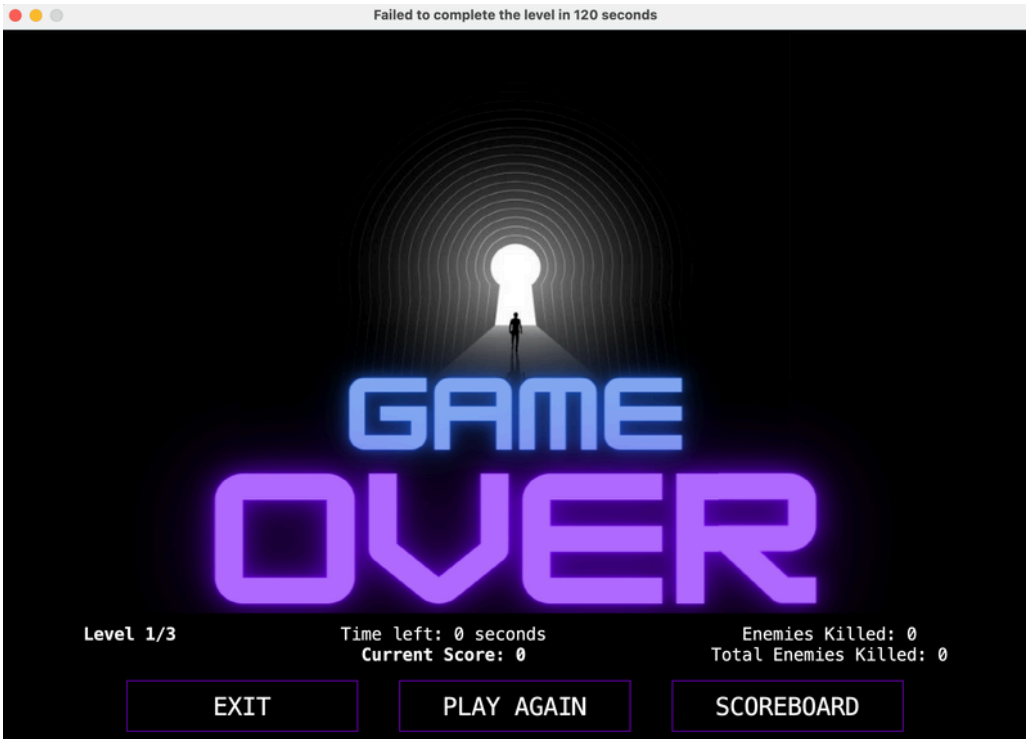
The time will decrease until it reaches 0, which is when the player loses. But there is one way that the player can add more time to their total. Throughout the maze there are enemies roaming around. If a player kills an enemy, 15 seconds will be added to their time.

Face an Enemy

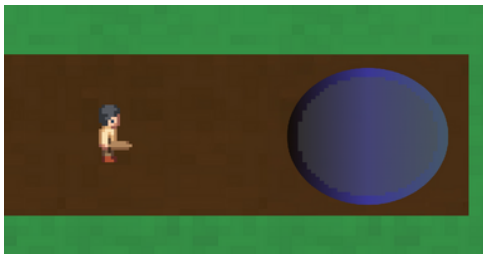
Successfully Kill Enemy

Game Over

When the timer reaches 0, the player loses and the Game Over Screen will display on the system. The player has 3 options shown on this screen. If they choose “EXIT” the system will shut down. If they choose “PLAY AGAIN” the player restarts at level 1. Even if the player was currently on level 3 when they died they will still be reset at level 1 and their score will start over. Their player name will stay the same as what they initially put it as on the Home screen. If the player chooses “SCOREBOARD” they will see the scoreboard with the best high scores from other players.



Winning



If the player makes it through three mazes without ever running out of time, they have won! After they reach the end of the final level, their time and the amount of enemies they kill will be calculated and turned into a final score.

If their score is good enough to make the scoreboard, the leaderboard will be updated and they will see the screen shown below. Which tells the player they have made the scoreboard.

If their score isn't good enough to make the scoreboard, then nothing on the leaderboard is changed and the player will see the screen below. Which tells them that their score wasn't good enough and encourages them to play again and receive a better score.

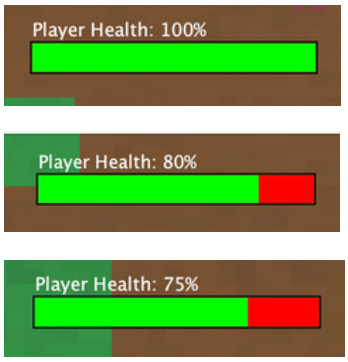
---- Enemy Interactions ----

Attacking Enemies

As mentioned earlier, along the way, you will encounter enemies lurking in the maze. If the player gets within a certain distance of the enemy, the enemy will attack you, and the player can choose to fight or flight. If they choose to fight, they must hit the enemy three times in order to kill them. If they accomplish this, they will be granted and additional 15 seconds to complete the maze.

Types of Enemies

There are two types of enemies seen in the maze, each enemy requires that the player hits them 3 times with their weapon in order to kill it. The two different enemies have different effects on player health. If enemy 1 comes in contact with the player their health will decrease by 20, if enemy 2 comes in contact with the player, the player health will decrease by 25.



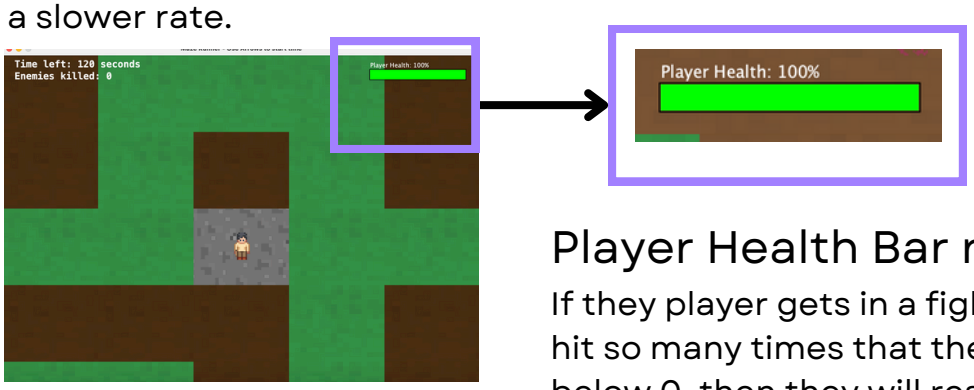
Ghost
Player Health Hit = 20%
Hit Count = 3

Mage
Player Health Hit = 25%
Hit Count = 3

Losing to an Enemy

Player Health Bar

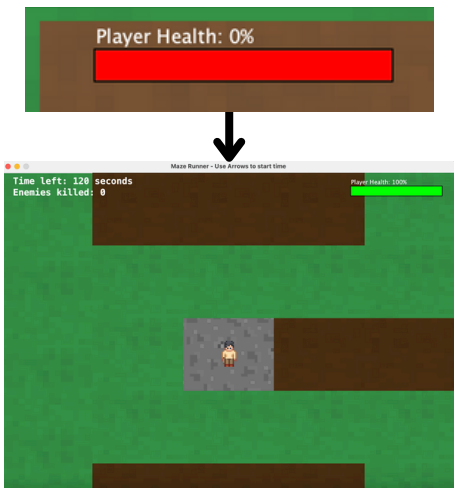
If the player chooses to fight an enemy the player does take the chance of losing to the enemy and losing their current progress in that maze. As a player, you have a health bar which keeps track of your current health located in the top right of the gaming screen. At the beginning of each level the player health will be reset to 100. If the player comes in contact with an enemy their health bar will decrease the given player health hit that that enemy has. Once a player escape the fight with an enemy, either by killing the enemy, or just running away, the player health bar will incrementally increase over time. If your health bar reached 50% after fighting an enemy, your health bar will slowly increases as you traverse the rest of the maze. Your healths increasing speed decreases as the levels get higher. By the time you reach level 3 your health bar regnerates at a slower rate.

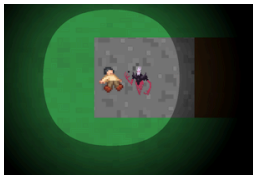


Player Health Bar reaches 0

If they player gets in a fight with an enemy and gets hit so many times that their player health goes below 0, then they will restart the maze. After dying, the player health bar is reset to 100 and they are respawned at the start location at the beginning of the maze they are currently trying to complete. The time will not be reset and will continue counting down from where they left off when they were killed by the enemy. If the player killed any other enemies throughout the level and then gets respawned after dying, those enemies will remain dead.

Health = 0 -> Level Resets

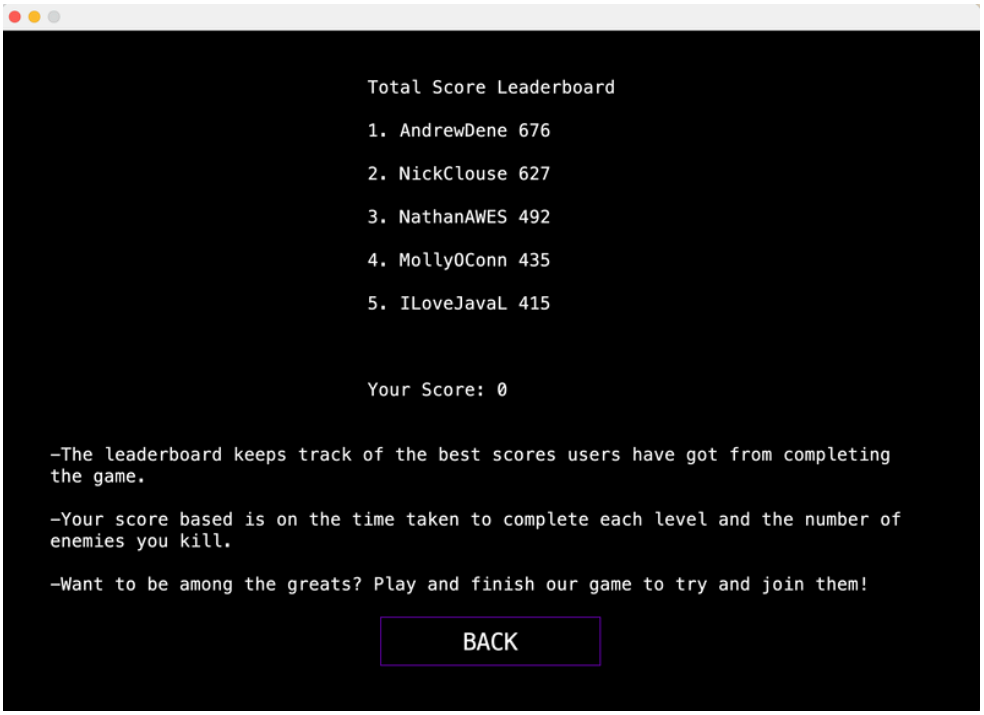




-------- Scoreboard --------

How the Scoreboard Works

At the beginning of the game, in between levels, at the game over screen, and at the you've won screen, the player is given the option to look at the scoreboard. The scoreboard keeps track of the 5 best players to play Maze Runner and will be updated if you beat one of the scores and become one of the greats.



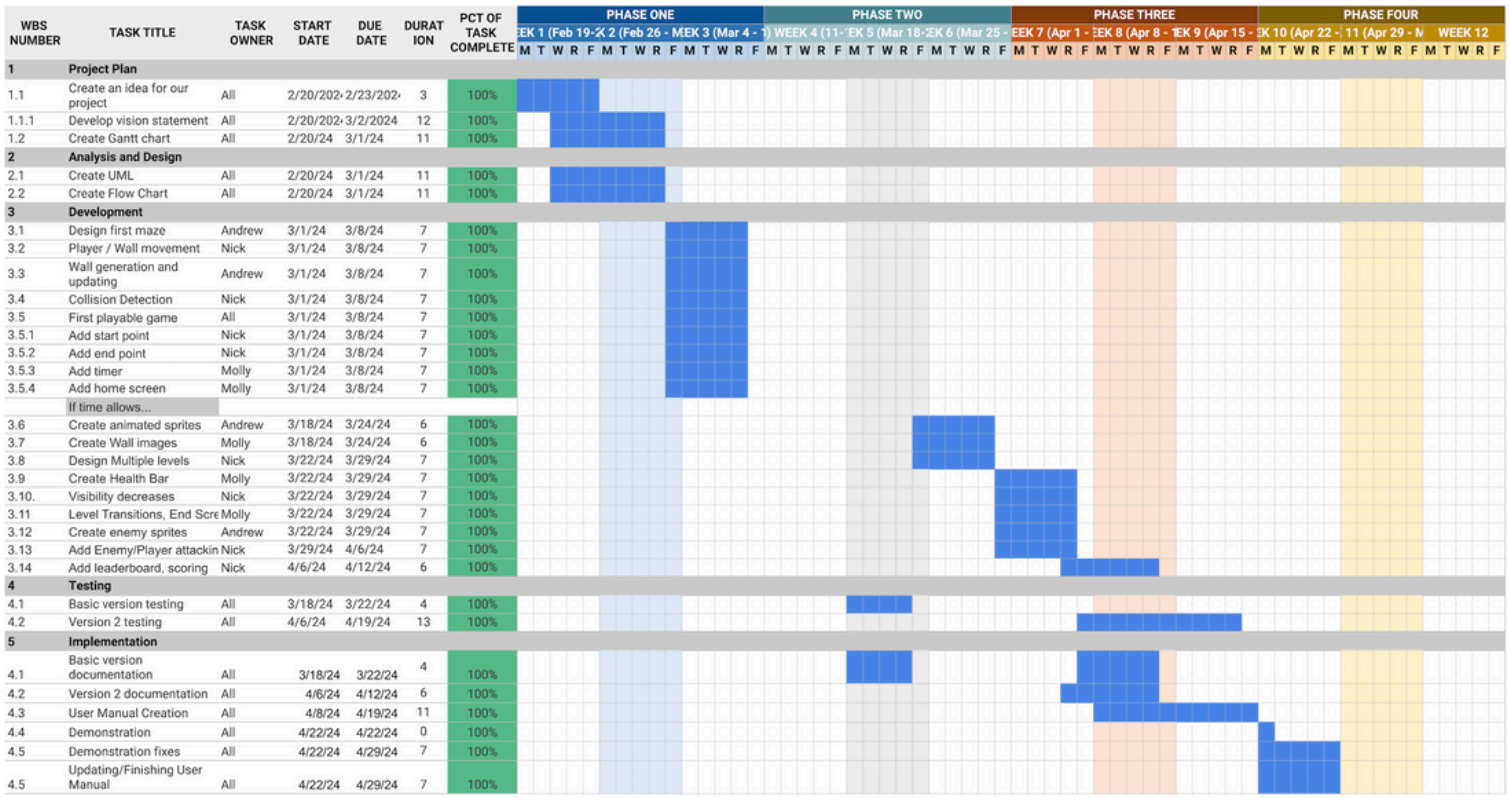
How Scoring Works

There is a scoreboard for each level and your overall time for three levels. You can view this scoreboards after completing a level. Your score is the time you had remaining. So if you finish the game faster and kill more enemies you will receive a higher score!

Developer Section

Schedule

Gantt Chart



WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
1	**Project Plan**					
1.1	Create an idea for our project	All	2/20/202	2/23/202	3	100%
1.1.1	Develop vision statement	All	2/20/202	3/2/2024	12	100%
1.2	Create Gantt chart	All	2/20/24	3/1/24	11	100%
2	**Analysis and Design**					
2.1	Create UML	All	2/20/24	3/1/24	11	100%
2.2	Create Flow Chart	All	2/20/24	3/1/24	11	100%
3	**Development**					
3.1	Design first maze	Andrew	3/1/24	3/8/24	7	100%
3.2	Player / Wall movement	Nick	3/1/24	3/8/24	7	100%
3.3	Wall generation and updating	Andrew	3/1/24	3/8/24	7	100%
3.4	Collision Detection	Nick	3/1/24	3/8/24	7	100%
3.5	First playable game	All	3/1/24	3/8/24	7	100%
3.5.1	Add start point	Nick	3/1/24	3/8/24	7	100%
3.5.2	Add end point	Nick	3/1/24	3/8/24	7	100%
3.5.3	Add timer	Molly	3/1/24	3/8/24	7	100%
3.5.4	Add home screen	Molly	3/1/24	3/8/24	7	100%
	If time allows...					
3.6	Create animated sprites	Andrew	3/18/24	3/24/24	6	100%
3.7	Create Wall images	Molly	3/18/24	3/24/24	6	100%
3.8	Design Multiple levels	Nick	3/22/24	3/29/24	7	100%
3.9	Create Health Bar	Molly	3/22/24	3/29/24	7	100%
3.10.	Visibility decreases	Nick	3/22/24	3/29/24	7	100%
3.11	Level Transitions, End Scre	Molly	3/22/24	3/29/24	7	100%
3.12	Create enemy sprites	Andrew	3/22/24	3/29/24	7	100%
3.13	Add Enemy/Player attackin	Nick	3/29/24	4/6/24	7	100%
3.14	Add leaderboard, scoring	Nick	4/6/24	4/12/24	6	100%
4	**Testing**					
4.1	Basic version testing	All	3/18/24	3/22/24	4	100%
4.2	Version 2 testing	All	4/6/24	4/19/24	13	100%
5	**Implementation**					
4.1	Basic version documentation	All	3/18/24	3/22/24	4	100%
4.2	Version 2 documentation	All	4/6/24	4/12/24	6	100%
4.3	User Manual Creation	All	4/8/24	4/19/24	11	100%
4.4	Demonstration	All	4/22/24	4/22/24	0	100%
4.5	Demonstration fixes	All	4/22/24	4/29/24	7	100%
4.5	Updating/Finishing User Manual	All	4/22/24	4/29/24	7	100%

Our actual schedule turned out to be very similar to our proposed schedule. However, we were able to add more features than expected, and some of the documentation/testing took longer than anticipated. Some of the later dates are slightly different, and there are several tasks that weren't on the proposed schedule.

Testing Criteria

Testing example for Leaderboard class

```java
/**
 * Main method, used for testing.
 *
 * @param args Arguments passed.
 */
public static void main(String[] args) {
    boolean allPassed = true;

    final String fileName = "leaderboards/overall_time_leaderboard.txt";
    final Leaderboard leaders = new Leaderboard(fileName);

    final Entry[] entries = Arrays.copyOf(leaders.getleaderboard(), leaders.getleaderboard().length);
    final String name = leaders.getleaderboardName();

    if (name == null || entries == null) {
        System.err.println("Failed to load leaderboard!");
        allPassed = false;
    }

    // This entry should be added
    leaders.addEntry("MazeRunner", 300);

    Entry[] newEntries = leaders.getleaderboard();
    // Make sure no entries have been changed
    for (int i = 0; i < entries.length; i++) {
        if (!entries[i].equals(newEntries[i])) {
            System.err.println("Leaderboard was updated when it shouldn't have been!");
            allPassed = false;
        }
    }
```

All of our classes used a similar testing layout. The main method of each class is used for testing. They test all methods in the class, and if any of them fail, the boolean allPassed is set to false, and a error message is printing to the console, which shows what failed. After all tests have concluded, a final message is shown in the console.

```java
if (allPassed) {
    System.out.println("All cases passed! :)");
} else {
    System.err.println("At least one case failed! :(");
}
```

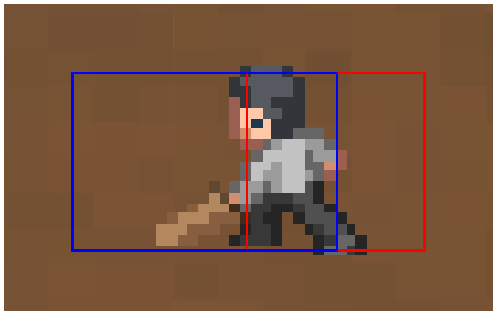
---------- Sprites ----------

Overview & Players

Overview

Our game currently has two types of sprites (players and enemies). Players and enemies are independent of each other (not connected by a sprite parent class). Both classes use BufferedImages. Details on sprite combat are found in the Attacking sections and are not addressed here. This section will focus on drawing sprites and class structures.

Player

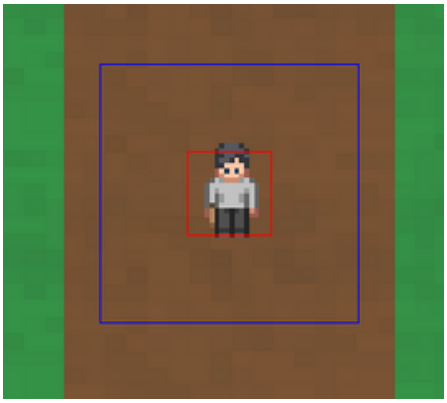


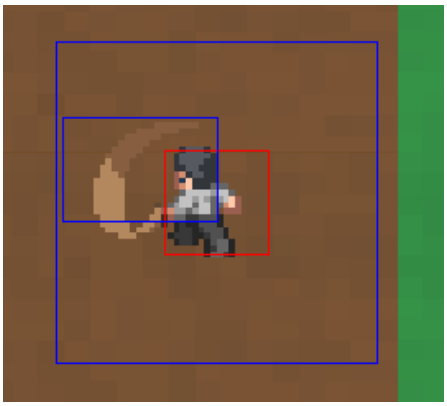
Player States: During the game the player can be facing one of 8 directions (N, NE, E, SE, S, SW, W, NW) and can be either idle, moving, or attacking which are both stored as enumerations. The image on the left shows currentFacing = Facing.W, currentState = State.Attacking, and shows the attack range (blue) and hitbox (red).

Image Loading: There are five sets of character images: “Knight”, “Civilian1”, “Civilian1(black)”, “Civilian2”, “Civilian2(black)”. Civilian2 is not an option because there are no weapon images. Currently, sprites start out with a weapon although future work could start the player out with no weapon. All images are loaded similarly based on the number and orientation of frames in the sprite sheet.

Player Cont. & Enemies

Player cont.





Drawing Images: Each image is drawn larger than the actual hitbox of the character (see image to the right). This was done so that attacking images could extend beyond the hitbox of the player. A constant, FRAMESPERSWITCH determines how many frames will be drawn before switching to the next image in the sequence (eg. after 6 frames switch to the next attacking image). When the player starts attacking the direction and state (attacking) are fixed until the animation is completed.

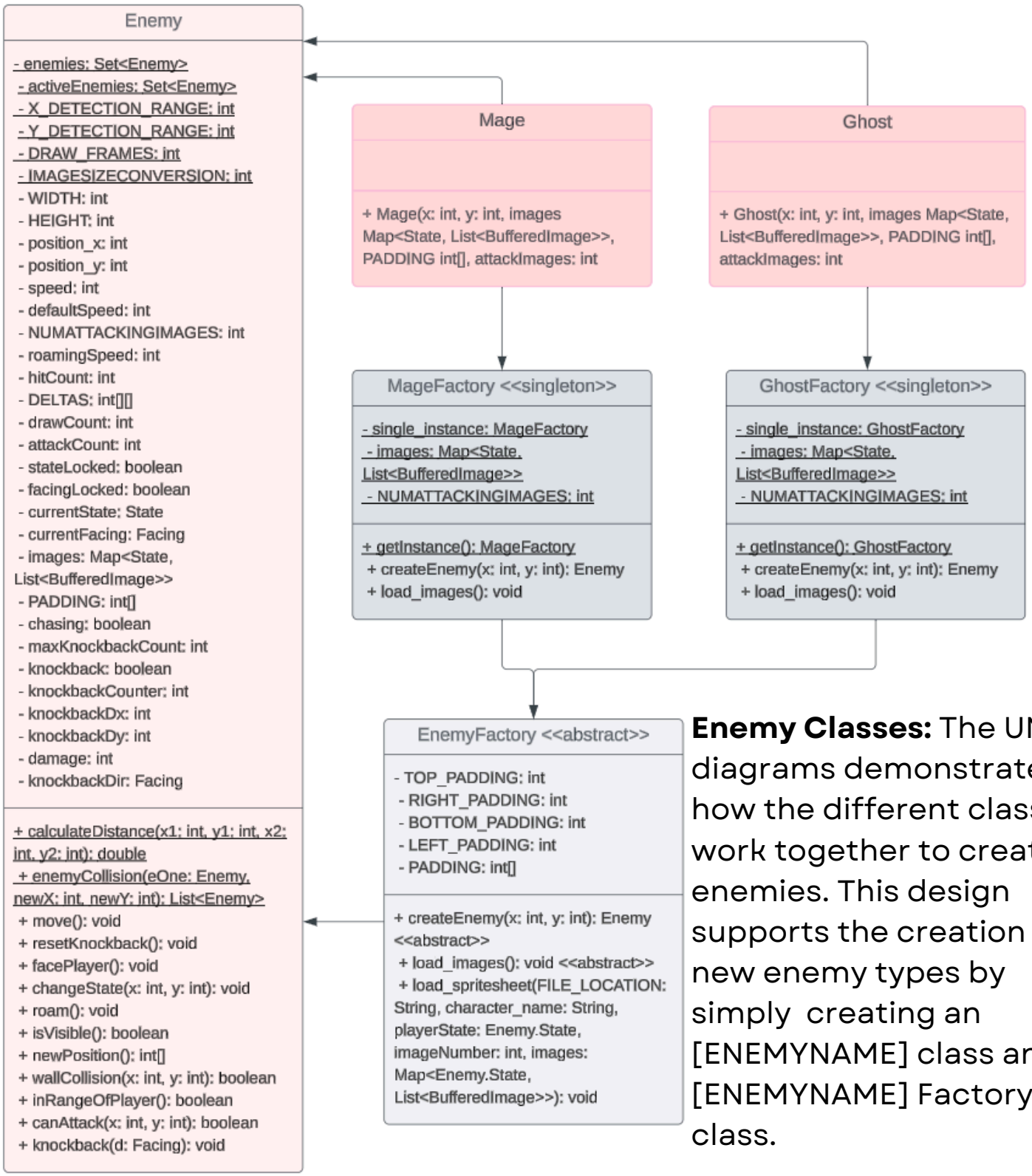
Enemies

Factory Creation Pattern: To create an enemy, you should first create an EnemyFactory for the enemy type. With the EnemyFactory (MageFactory or GhostFactory) you can call createEnemy() with parameters: int x, int y to specify the enemy's starting location. An example of creating two mages and two ghosts is shown below.

```
// Create factories that will create our images.
EnemyFactory mageCreator = MageFactory.getInstance();
EnemyFactory ghostCreator = GhostFactory.getInstance();

// Create enemy instances using 'magic' numbers for x and y positions.
Enemy merlin = mageCreator.createEnemy(10, 10);
Enemy spooky = ghostCreator.createEnemy(10, 110);
Enemy casper = ghostCreator.createEnemy(110, 10);
Enemy gandalf = mageCreator.createEnemy(110, 110);
```

UML Enemy Class Relationships



Enemy Classes: The UML diagrams demonstrate how the different classes work together to create enemies. This design supports the creation of new enemy types by simply creating an [ENEMYNAME] class and [ENEMYNAME] Factory class.

Enemies Cont.

Enemy Position: There are two components that go into drawing enemy position: ChunkManager offset and the enemy's relative position. The ChunkManager offset represents the movement of the player from the original location while the relative position is the position of the enemy on the map. The code below shows the position that the Enemy will be drawn.

```
public void draw(Graphics2D g) {
    // Store position based on movement of the map
    final int final_x = ChunkManager.xOffset + position_x;
    final int final_y = ChunkManager.yOffset + position_y;
```

2-directional limitation: The enemy asset pack we acquired only has the enemies facing in 1 direction (2 directions when we flip the image) compared to our player that faces 8 directions. Our enemy can still attack in 8 directions.

Opportunities:

- Finding another asset pack or designing enemies that face in 8 directions could improve the aesthetics of the game.
- Adding knockback animations.
- There are 3 enemies that we did not have time to implement: Salamander, Beaver, and Necromancer.

Enemy Movement

If the enemy is not actively attacking or being knocked back, they can move toward the player or roam. Each enemy has a range that dictates when they start moving toward the player. If the player is out of range, the enemy patrols back and forth. If the enemy's distance from the player is less than a specified detection range (defined in GameVariables), it starts chasing the player.

When chasing, the enemy calculates the distance to the player using the distance formula and considers eight directions to move. If moving in a direction would result in hitting a wall or another enemy, it cannot move that way. If a valid direction is found, the enemy updates its position.

Pseudocode for moving a enemy

```
move() {
  if (knockback) {
    Knockback enemy
  } else if (canAttackPlayer(currentX, currentY)) {
    Attack player
  } else {
    if (playerInRange(currentX, currentY)) {
      Chase player
    } else {
      Roam around
    }
  }
}
```

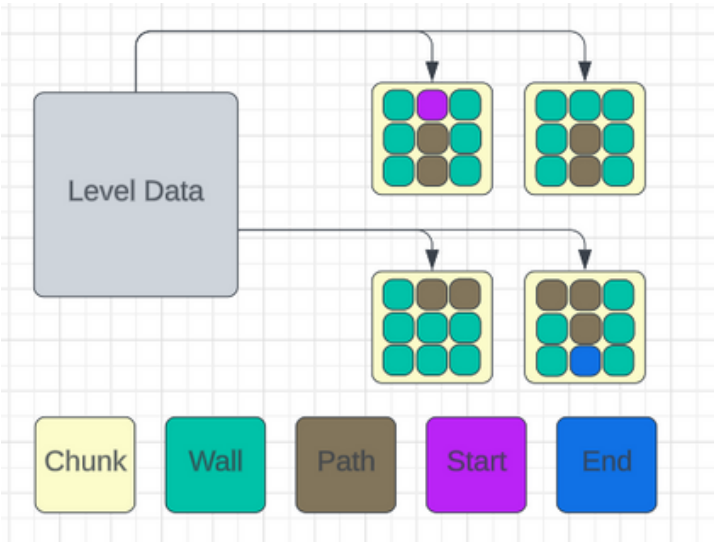
------- Level Loading -------

Overview

Our level data is ultimately organized into three hierarchical levels: ChunkManager, Chunk(s), and PositionBlock(s). ChunkManager loads our data in from a text file, splits our data into Chunk(s) and PositionBlock(s), and is created and called by GamePanel during runtime. In this section we explore these steps and components in more detail.

Level File Structure (example shown below):
Description: <FILE DESCRIPTION>
dimension: <X chunks by X chunks (ex. 4x4)
chunk_size: <X PositionBlocks by X PositionBlocks> (ex. 10x10)
<LEVEL DATA>

```
1 Description: 1x2 chunks, where
2 dimension:1x2
3 chunk_size:10x10
4 1111111111
5 1000000001
6 1111011111
7 0001010000
8 0001010000
9 0001010000
10 0001010000
11 0001010000
12 0001010000
13 0001010000
14 1111011111
15 1200400031
16 1111011111
17 0001010000
18 0001410000
19 0001010000
20 0001010000
21 0001110000
22 0000000000
23 0000000000
```

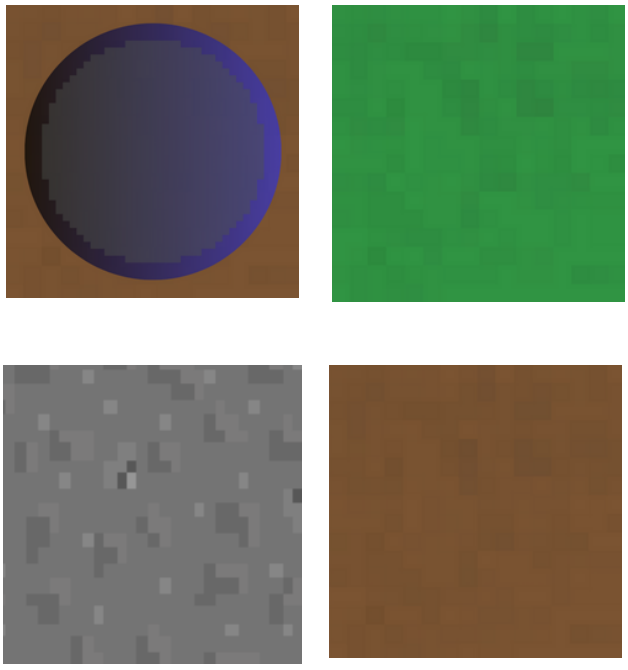


Chunk Manager & Blocks

ChunkManager: This component is responsible for loading level data into chunks and positioning blocks. It randomly selects one of five pre-designed level layouts for each level. For instance, level 1 will have a 2x2 chunk map, level 2: 3x3 chunks, and level 3: 4x4 chunks. To display the level data, GamePanel triggers the draw(Graphics2d) method on the ChunkManager instance, which then cascades the draw operation through the chunks and position blocks. At any given point, only a few chunks will be visible on the screen so we keep track of the active chunks to decide which chunks are drawn.

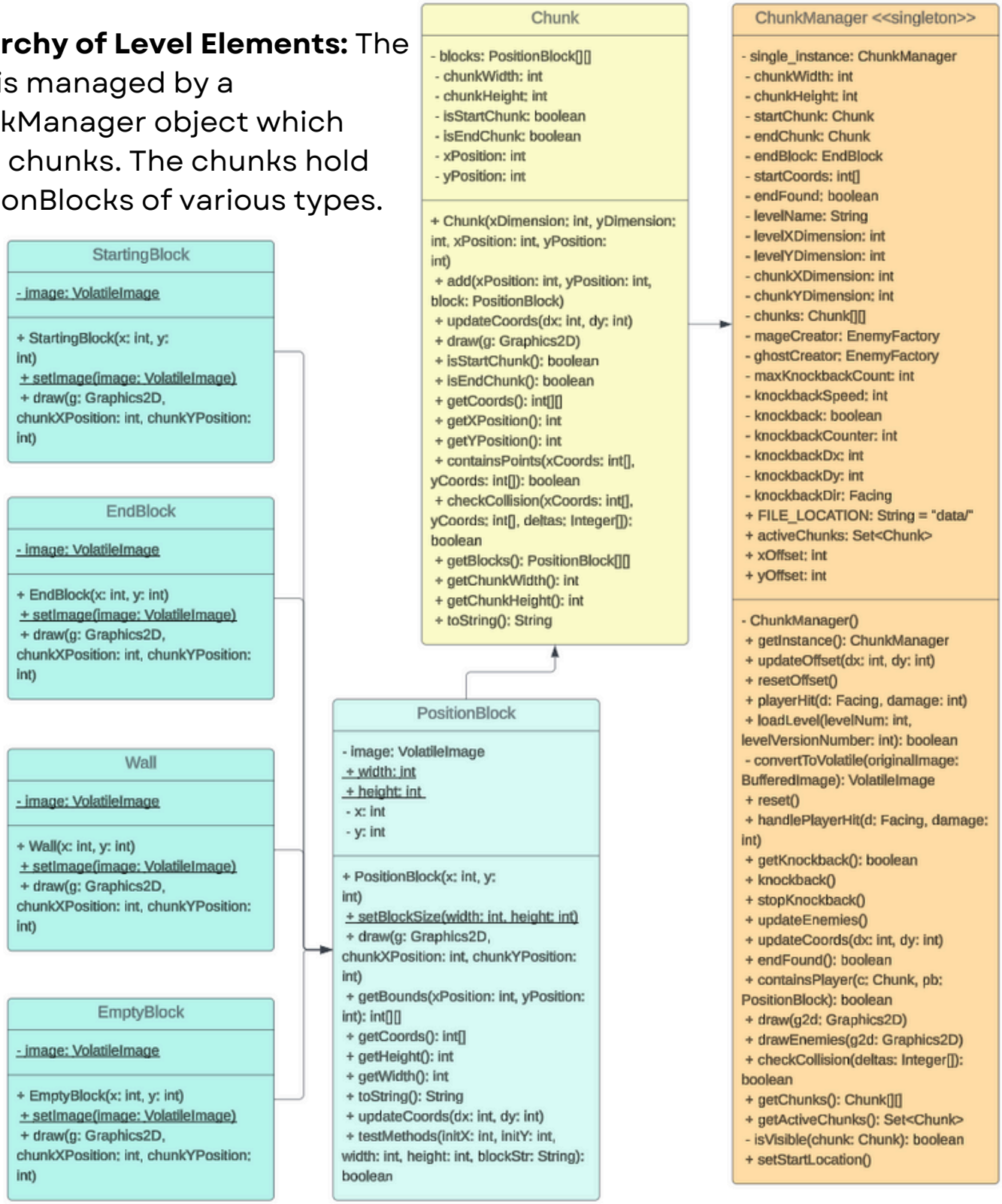
Chunks: Chunks are the building blocks of the level. Each chunk holds a two-dimensional array of position blocks.

PositionBlock: There are 4 types of PositionBlock: EmptyBlock, EndBlock, StartingBlock, and Wall. Each PositionBlock uses VolatileImage(s) which are stored statically for each type of PositionBlock. To set the image for a particular type, call setImage(VolatileImage). The images to the right show an EndBlock, Wall, StartingBlock, and EmptyBlock in the top left, top right, bottom left, and bottom right respectively.



UML Level Structure

Hierarchy of Level Elements: The level is managed by a ChunkManager object which holds chunks. The chunks hold PositionBlocks of various types.



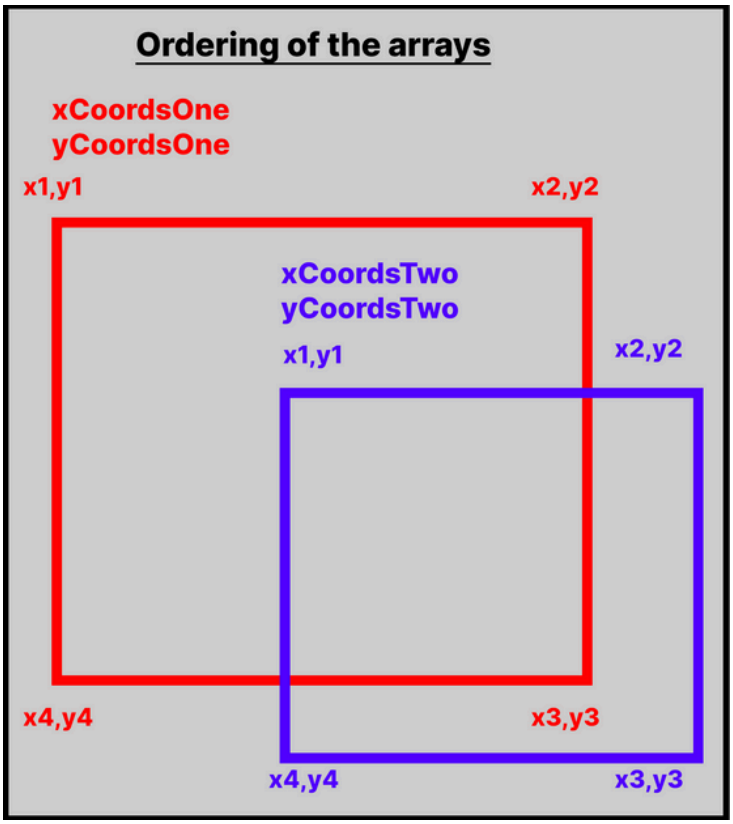
---- Collision Detection ----

Detecting Collisions

- Detecting collisions is a very important component of our game. We ended up making a class called *CollisionDetection*, which has two static functions, *getCollision()* and *fullCollision()*, so different classes like *Player*, *Enemy*, and *ChunkManager*, and *Chunk* can called one of those functions when they need to.

- The functions check if any part of each square is intersecting. If so, *getCollision()* returns true. If there is an intersection, *fullCollision()* gets the overlap of each side, and checks if all sides of CoordsTwo are inside of CoordsOne and is in the general center of CoordsOne, and returns true if so. This is considered a 'full collision,' and we mainly used this when detecting if the player is has found the end square.

- The first two arrays passed are the x and y coordinates of the first object, and the second two arrays are the x and y coordinates of the second object. Each array needs to be a length of four, and should represent a rectangle or square.



Detecting Sprites hitting Walls

- When checking if the player is hitting a wall, we use the *getCollision()* function. For each direction the player could move (Up,Down,Left,Right), we check if moving in that direction would result in hitting a wall. If it would, then the player isn't allowed to move in that direction.

- The *ChunkManager* class handles checking wall collisions. It calls the *checkCollision()* function on each chunk in activeChunks, which contains all the chunks currently visible on the screen. Each wall has a hitbox, which is slightly bigger than its width and height, so the player stops just before touching the wall.

- We use a similar method with detecting if enemies will hit a wall, but the enemies can move diagonally as well as vertically and horizontally, so we have to check additional directions to see if a collision would occur.

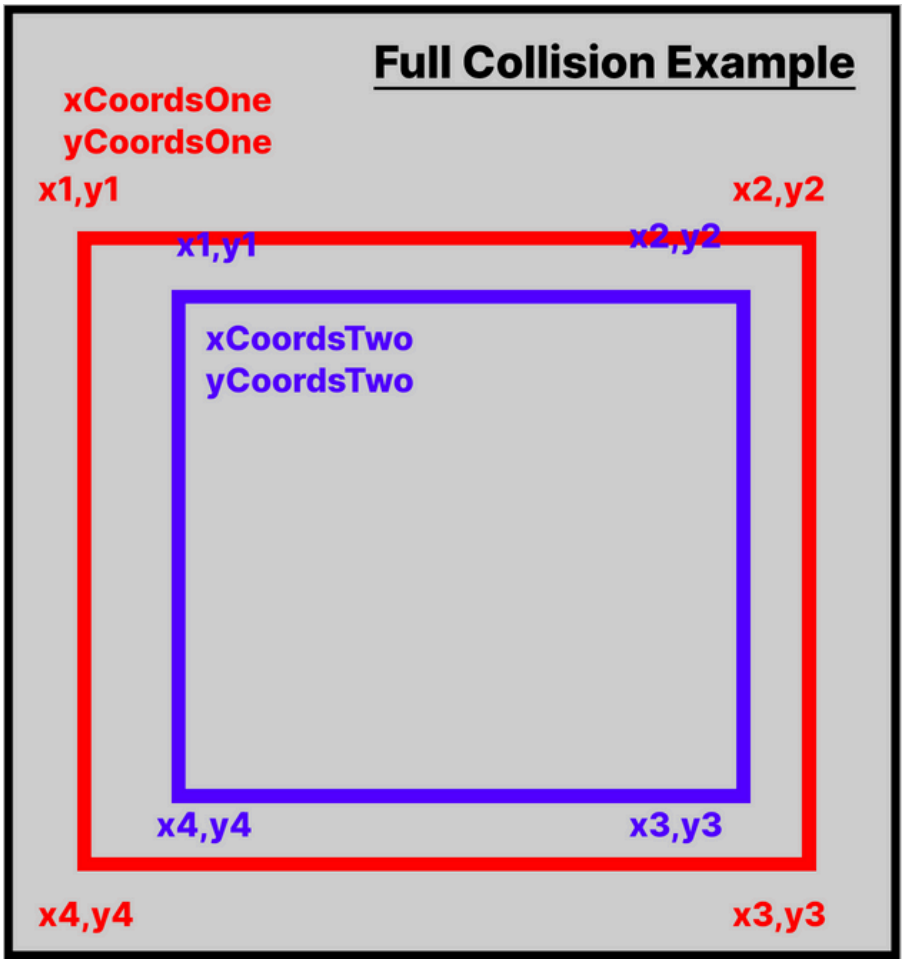
```
/**
 * Checks for collision between the given coordinates, which represent a square.
 * The coordinates are ordered top left, top right, bottom right, bottom left.
 *
 * @param xCoordsOne The first set of x coordinates.
 * @param yCoordsOne The first set of y coordinates.
 * @param xCoordsTwo The second set of x coordinates.
 * @param yCoordsTwo The second set of y coordinates.
 * @return true if there is a collision.
 */
public static boolean getCollision(int[] xCoordsOne, int[] yCoordsOne, int[] xCoordsTwo, int[] yCoordsTwo) {

    // if x1 top left x < x2 bottom right x
    if (xCoordsOne[0] <= xCoordsTwo[2]
            // if x1 bottom right x > x2 top left x
            && xCoordsOne[2] >= xCoordsTwo[0]
            // if y1 top left y < y2 bottom right y
            && yCoordsOne[0] <= yCoordsTwo[2]
            // if y1 bottom right y > y2 top left y
            && yCoordsOne[2] >= yCoordsTwo[0]) {
        return true;
    }

    return false;
}
```

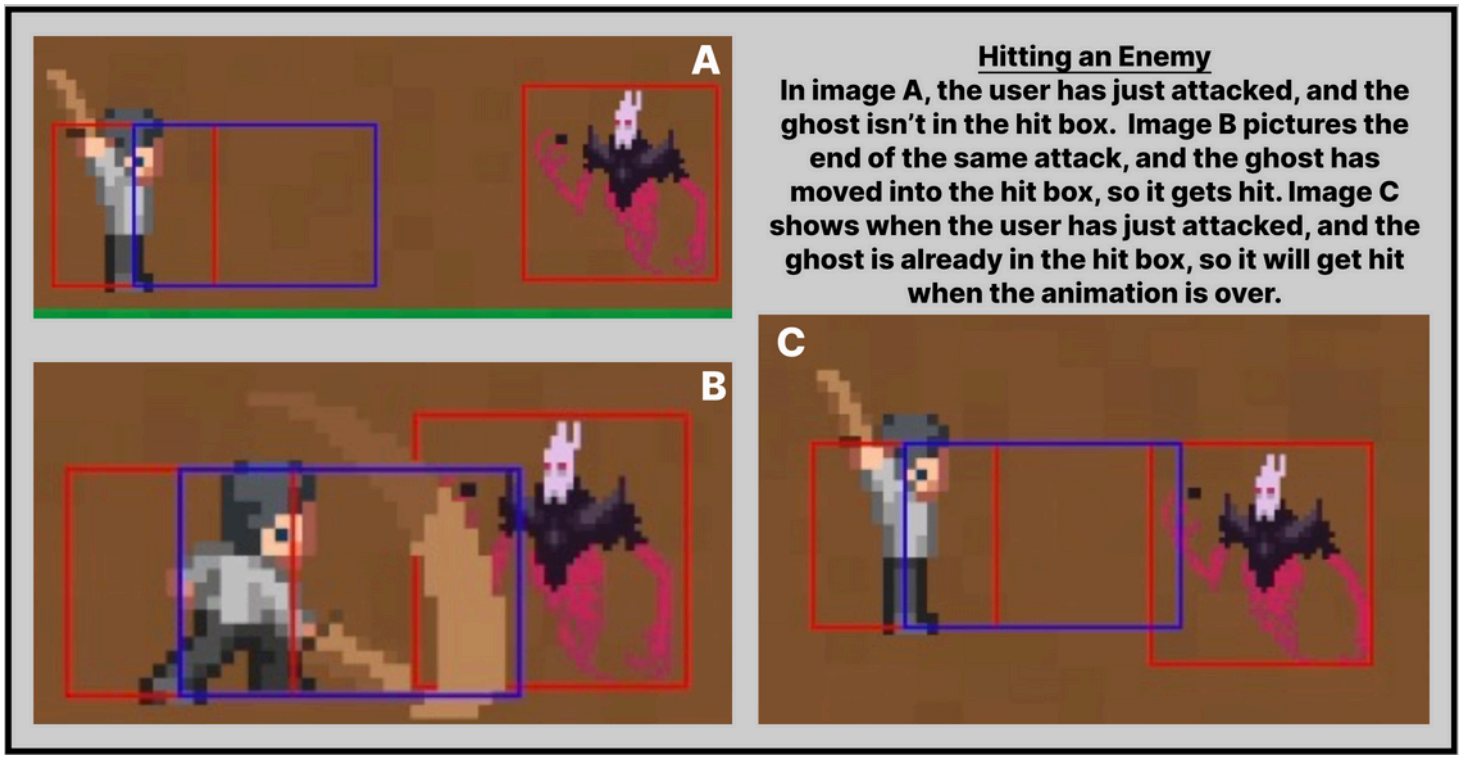
Detecting the End

When a new level is loaded, *ChunkManager* keep track of the *endChunk*, which contains the end of the maze. Each time *ChunkManager* updates the coordinates of each chunk, it checks if the *endChunk* is currently active. If it is, it uses *containsPlayer()* to check if the player is inside the *endBlock*. Using *fullCollision()*, the function returns true if a full collision is detected. We use *fullCollision()* in this instance because we want the player to reach the middle of the *endBlock* instead of just touching the *endBlock*.



Detecting Player-to-Enemy Attacking

When checking if the player hits an enemy, we use the *getCollision()* function. When the user presses the spacebar, the player sprite attacks. While it's attacking, a hitbox is checked, and anything within the hitbox during the attack is hit. Once the animation is over, any enemies found are hit and knocked back. While an enemy is being knocked back, if it hits any other enemies, those enemies get knocked back as well.



Detecting Enemy-to-Player Attacking

While moving an enemy, we check if it can hit the player using *getCollision()*. If the enemy's hitbox is colliding with the player's hitbox, the enemy can attack the player, and does so.

---- AudioPlayer ----

AudioPlayer

The code for this class was first created using the help of Kaarin gaming. (https://www.youtube.com/watch?v=afOcsF2-xbg&list=PL4rzdwizLaxYmltJQRjq18a9gsSyEQQ-0&index=33). Once the basic functions were made and understood we altered them to better fit our needs. The AudioPlayer class is simple and only has five functions. With these functions you can call a .wav audio file and play the sound once, play sound repeatedly, stop the sound, set the volume of the sound, or check if the sound is already playing.

This code shows the loading and start of a sound and then plays it continuously until the sound is stopped.

```
/**
 * Plays a song continuously.
 * @param fileName The name of the song file to play.
 */
public void playSong(String fileName) {
    try {
        AudioInputStream audioInputStream = AudioSystem.getAudioInputStream(getClass().getResourceAsStream(fileName));
        clip = AudioSystem.getClip();
        clip.open(audioInputStream);
        clip.loop(Clip.LOOP_CONTINUOUSLY); // Loop the clip continuously
        isPlaying = true;
    } catch (Exception e) {
        System.err.println("Failed to load audio file " + fileName);
    }
}
```

All sounds were found on the Pixabay where all content is released under the Pixabay license, which makes the content safe to use without asking permission or giving credit to the artist, even for commercial purposes.

Sounds

- Player Attacking (swinging sword)
- Player Walking (gravel path noises)
- Player death (gaming death)
- Player hitting an Enemy (punch noise)
- Enemy hitting a player (shooting fire noise)
- Completing Level (gaming level up)
- Granting additional time (ding ding)
- Menu Song (intense music)
- Level Play (intense music)
- Game Over (womp womp)
- Winning Screen (Happy valiant music)

---- Running the Game ----

Main

The Main class is the core of our game. It manages top-level game logic, user interactions with the screen and buttons, and screen display. Key features include timing player sessions, displaying game elements, managing player input, tracking scores, and updating leaderboards. It controls game flow, from starting the game to handling wins, losses, and restarts.

Main JavaDoc

Modifier and Type	Field	Description
static boolean	addedToLeaderboard	Keeps track if user has been added to leaderboard.
static boolean	addTime	Keeps track of if time should be added.
private static BufferedImage	backgroundImage	Background image that is the same as the maze walls.
private static final Cursor	defaultCursor	Default Cursor, used when player is in between levels or at home/end screen.
static int	enemiesKilled	Number of enemies killed for current level.
private static GamePanel	gamePanel	The main game panel where the game is rendered.
private static HomeScreen	homePanel	Home screen.
private static GameOverWIN	nextLevel	Win screen when user reaches end of a level.
private static String	playerName	Stores player's name.
static int	seconds_left	Remaining seconds left for the player on current level.
static int	timeAmount	Time elapsed for level.
private static GameOverLOSE	timeOut	Lose screen when user runs out of time.
private static Timer	timer	Timer object used for timing how long the player has.
static int	totalEnemiesKilled	Total enemies killed by the player.
static int	totalTimePlayed	Total time the player takes to beat all levels.
private static final Cursor	transparentCursor	Create a transparent cursor.
private static JFrame	window	Window used to display the game.
private static finalWinScreen	winner	Win screen.

Modifier and Type	Method	Description
static void	addScoreToLeader()	Adds player score to respective leaderboard.
static void	addTime(int t)	Adds more time to the time player has left.
static void	closeMainWindow()	Closes the main window.
static void	disablePanels()	Removes the game over screen, used when the player chooses play again after running out of time in a level.
static void	enemyKilled()	Adds 1 to the players enemy kill count
static void	gameOverPanel(boolean show)	Disables the other panels and displays the "Game Over" game panel, when player runs out of time
static int	getLevel()	Gets the current level the player is on.
static int	getScore()	Gets the score of the player.
static void	main(String[] args)	Main method to start the game.
static boolean	otherPanelRunning()	Returns true if the next level panel is still running and being seen
static void	resetTime()	Resets the game timer and associated variables.
static void	restartGame()	Resets the game and removes the old window and starts a new one
static void	runMainCode()	Creates window and starts game.
static void	showFinalWinScreen(boolean show)	Disables the other panels and displays the "You win" game panel
static void	showGamePanel()	Disables the other panels and displays the main game panel
static void	showNextLevelPanel(boolean show)	Disables the other panels and displays the "Next Level" game panel
static void	stopTime()	Stops the timer
static void	updateTotalTimeAndEnemies()	Updates the total time and the total enemies a player has killed in a level

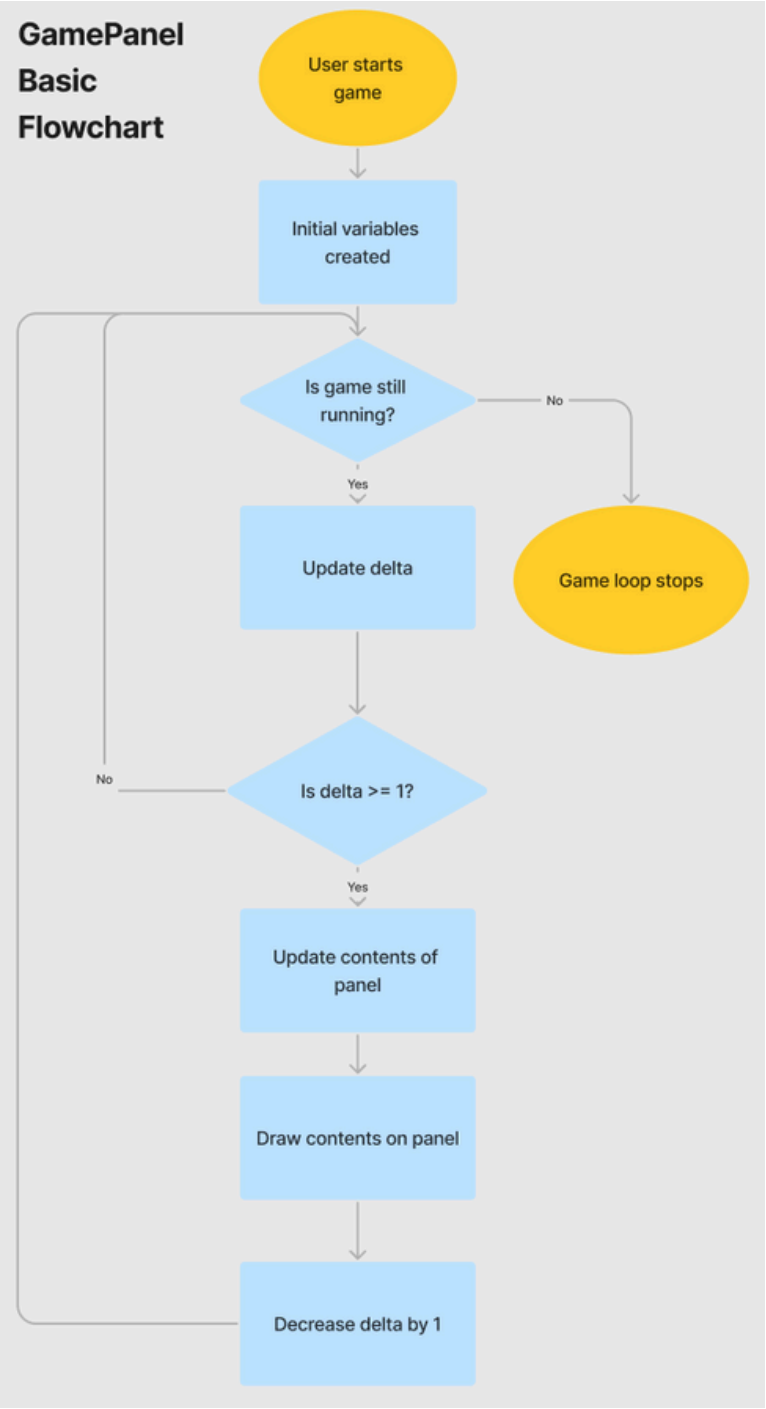
GamePanel

GamePanel handles creating the game screen and contains the game loop. It manages the top-level updating and rendering of all objects on the screen.

GamePanel's update() function first checks if the end has been found or if the player's health is at 0. It uses a *ChunkManager* object to update the player and enemies position depending on the keys pressed by the user.

After everything is updated, the *repaint()* function draws the maze, player, enemies, health bar, and time left/enemies killed on the screen

Our game loop uses a concept called "delta time" to update game logic and render frames. Delta time represents the elapsed time between frames, and it's used to ensure that the game runs smoothly and consistently regardless of the frame rate.



---- Final Thoughts ----

Challenges and Opportunities

Challenges Faced:

- **Github** - Hard learning curve
- **Project Management** - We did general planning for our schedule and ideas, but we should have planned more specific components, and assigned tasks better so we didn't always have to wait for tasks to be done before we could work on other features.
- **File Organization** - Faced challenges on converting how our files were stored across all of our devices.
- **Computer differences** - Our game would become very laggy with some features depending on the computer.

Future Opportunities:

- Our collision methods could be more efficient, by using Java's built in methods
- Find an efficient way to incorporate the effect of decreasing visibility into the game. Right now it happens when the player dies, but we originally wanted the players visibility to decrease as time went on
- Implement more enemy sprites
- Improve enemy movement, they can get stuck on walls pretty easily
- Animations for when player/enemy is knocked back
- Secret passages through the maze
- Storyline
- Enemy health bar

Thank you!

Thank you for reading our manual!

GitHub link

Includes our source code and JavaDoc, and other relevant information.