VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY

THE INTERNATIONAL UNIVERSITY

ALGORITHM & DATA STRUCTURE

ITIT013IU

FINAL REPORT

Group Member

1. Bùi Gia Bảo ITITIU22019
2. Hồ Gia Trân ITITIU22020
3. Nguyễn Hải Thanh ITITIU22146
4. Nguyễn Hoàng Giang ITITIU22046

Table of Contents

Table of Figure

LAST DAY ON EARTH

I. INTRODUCTION

1. Abstract

This project examines the development of a 2D game (named Last Day on Earth) implemented in Java, focusing on the application of data structures and algorithms as part of a Data Structure and Algorithm course project. The game features a player character with movement and shooting capabilities, enemy warriors that dynamically spawn and pursue the player using algorithm for pathfinding, employing a Raycasting algorithm to detect the player's visibility without obstructions before chasing. Multiple map sizes ("Small," "Medium," "Big") are supported to test the efficiency of the path finding algorithm and incorporating collision detection. Key data structures such as ArrayLists were utilized to manage dynamic entities like bullets and warriors, while a 2D array represented the game map for efficient tile-based collision checks. Algorithms were employed to ensure optimal pathfinding for enemy movement, demonstrating practical use of graph traversal techniques. The resulting game successfully integrates those elements, providing a playable experience where players can engage enemies and achieve victory. Although there are still some limitations, the game can be improved further in the future. This project highlights the effective application of data structures and algorithms in game development, thereby helping programmers strengthen their Java and DSA skills.

2. About Last Day on Earth

Last Day on Earth is a 2D survival action game developed as a project for the Data Structure and Algorithm course. The game falls into the survival and action genres, where players must defend themselves against relentless waves of zombie-like warriors and a formidable boss in a post-apocalyptic world. The core gameplay revolves around a lone survivor (the player character) who uses a gun to fend off enemies, utilizing strategic movement and shooting across dynamically generated maps of varying sizes ("Small," "Medium," "Big").

The objective is to survive by eliminating enemies, including warriors that pursue the player using the A* search algorithm for pathfinding, and a powerful boss that appears after a set time. The boss utilizes a Raycasting algorithm to detect the player's position, initiating pursuit only when a clear line of sight exists without obstructions, adding a layer of strategic depth. Bullets can be fired in four directions (left, right, up, down) to combat threats, with the game concluding in victory upon defeating the boss or in defeat if the player is overwhelmed. Tile-based maps are managed by a 2D array for collision detection. ArrayLists are utilized to handle dynamic entities such as bullets and warriors, while A* search algorithm and Raycasting ensure efficient enemy movement and behavior. The design highlights the application of data structures and algorithms, making Last Day on Earth a practical demonstration of algorithmic problem-solving in a gaming context.

II. SOFTWARE REQUIREMENTS

- **Java Development Kit (JDK) 11**: Essential for compiling and running the Java-based game, providing the core runtime environment and libraries such as Java Swing and AWT for graphical user interface development.
- **Visual Studio Code (IDE)**: Employed as the integrated development environment to write, debug, and manage Java source code, offering extensions for enhanced coding support and version control integration.
- **GitHub Desktop**: Utilized for version control, enabling commits, branch management, and merging to maintain a structured development workflow and track changes in the project repository.
- **UML Making Tools**: Used to design and document the game's architecture, such as class diagrams and sequence diagrams, ensuring a clear representation of data structures and algorithmic flows.
- **Canva**: Applied for editing images and designing backgrounds, facilitating the creation of visual assets such as maps, character sprites, and user interface elements to enhance the game's aesthetic appeal.

III. GAME MECHANICS

The Last Day on Earth game incorporates a set of mechanics designed to deliver a survival action experience, leveraging data structures and algorithms to enhance gameplay dynamics. The game first start at Main Menu, click on "Start Game" to enjoy the game.

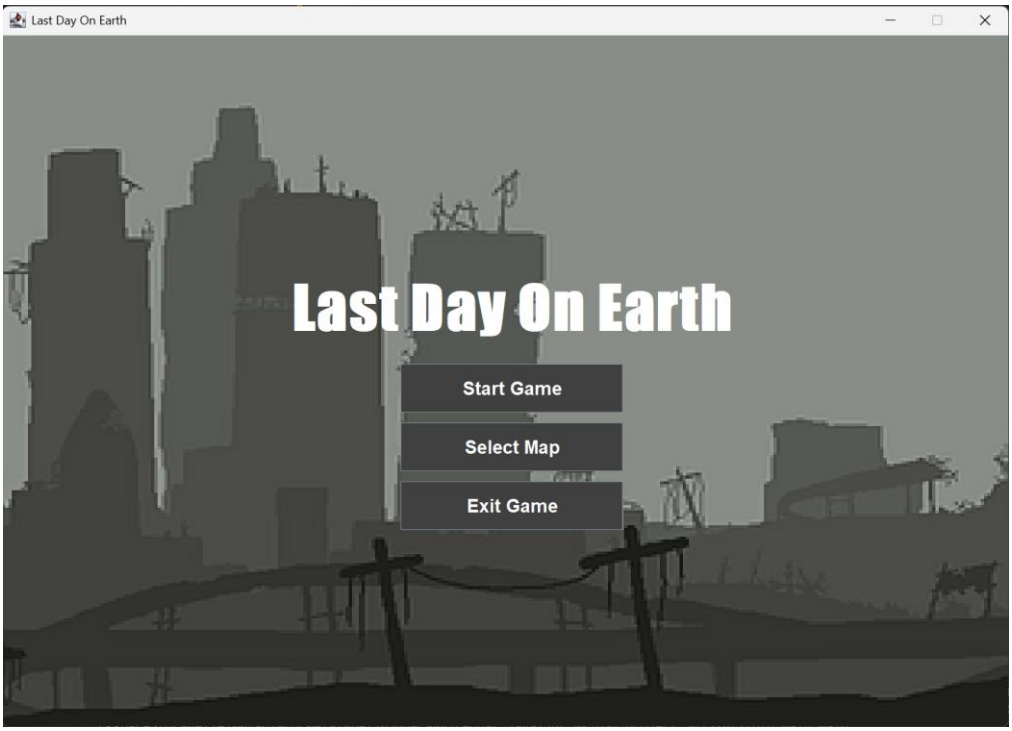


Figure 1. Main Menu

The core mechanic centers on a player character equipped with a gun, capable of moving in four directions (left, right, up, down) using the ADSW key and firing bullets to combat enemies using the arrow keys. Bullets are managed dynamically using an ArrayList, with their movement and collision tracked across the map. The game features enemy warriors that spawn at regular intervals, pursuing the player through A* search algorithm applied to the 2D array, ensuring optimal pathfinding from their current position to the player's location. A boss enemy appears after a predetermined time, employing a Raycasting algorithm to detect the player's visibility. The Raycasting technique casts rays from the boss's position to the player, checking for obstructions using the 2D array, and initiates pursuit only when a clear line of sight is established.

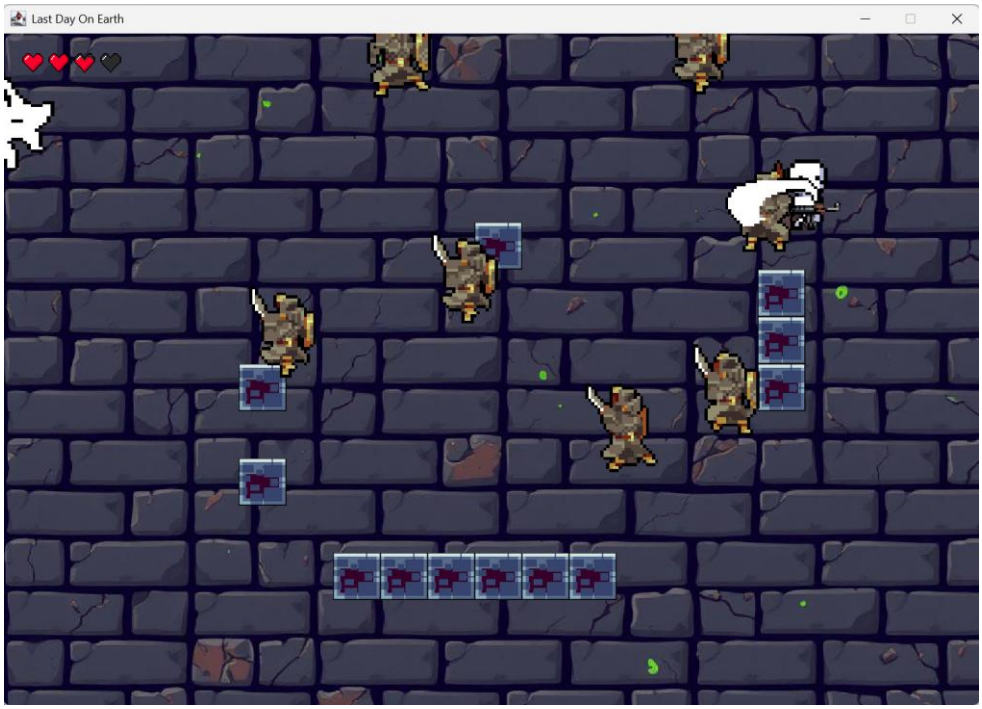


Figure 2. Player with Warrior

Figure 3. Player with Boss

Gameplay objectives include surviving waves of warriors and defeating the boss to achieve victory, with defeat occurring if the player's health is depleted.

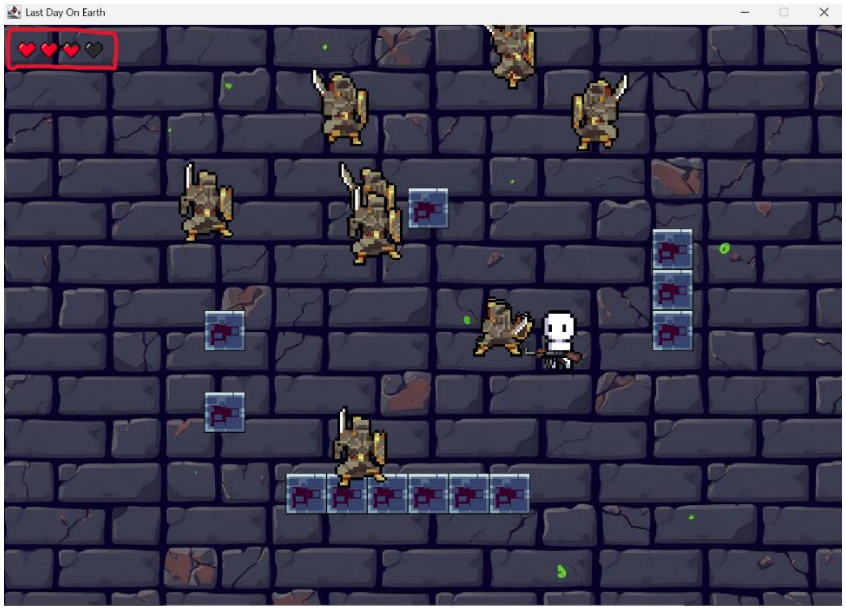


Figure 4. Heart status

When the player defeats the Boss, a "Victory" message will appear in the middle of the screen and allow the player to press Enter to restart the game.

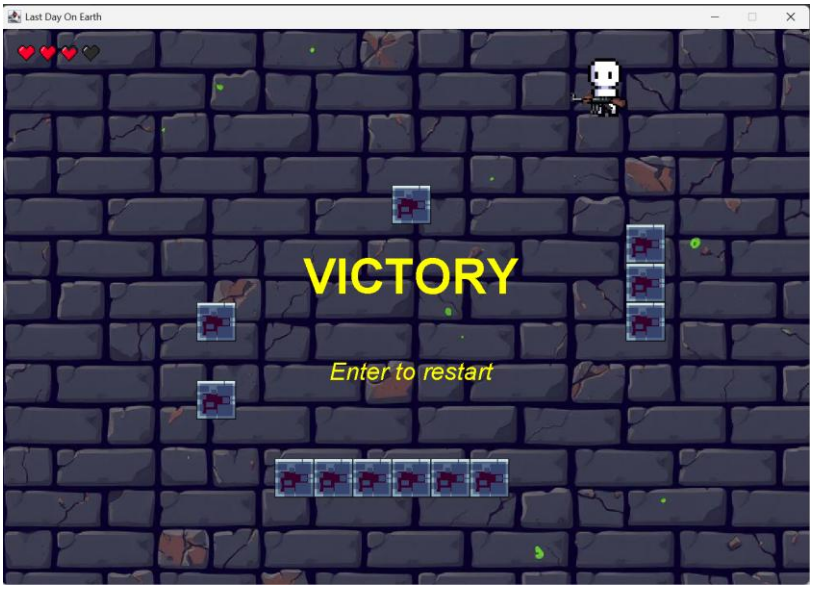


Figure 5. Announcing the winning player

When the player's health is depleted, a "Game Over" message will appear in the middle of the screen and the player will be given options to either play again or return to the Main Menu.

Figure 6. Game Over Panel

IV. DESIGN

1. UMLs

A UML class diagram was developed to represent the relationships and structure of classes:

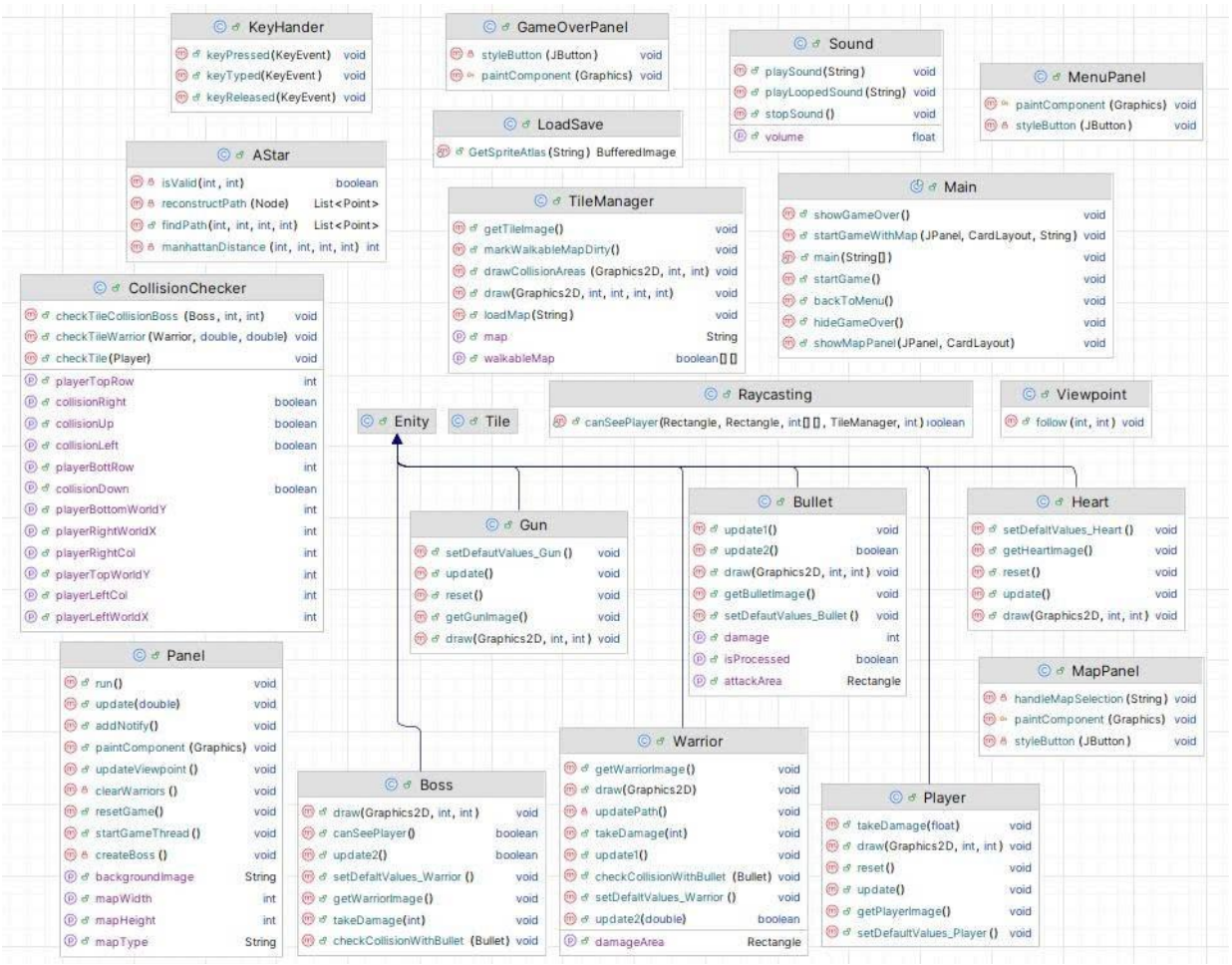


Figure 7. UML Class Diagram

2. Entity Class

a. Warrior

- Warriors are zombie-like enemies that spawn periodically to pursue and attack the player, serving as the primary threat in early gameplay. They use the A* search algorithm to navigate the map, ensuring efficient pathfinding to the player's position.

- Statistics:

    - Health: 2 (takes 2 bullet hits to destroy).
    - Damage: 1 (deals 1 damage to the player on contact).

Figure 8. Warrior

- Speed: 700 pixels per second (adjusted by deltaTime).
- Attack Range: 70 pixels.
- Spawn Interval: Every 1 second.

b. Boss

- The Boss is the ultimate enemy, appearing after a set duration to challenge the player. It employs a Raycasting algorithm to detect the player's position, initiating pursuit only when a clear line of sight exists, adding strategic depth to the encounter.

- Statistics:

  - Health: 5 (takes 5 bullet hits to defeat, 0.5 damage if not visible).
  - Damage: 1 (deals 1 damage on contact or attack object hit).
  - Speed: 25 pixels per frame (base), 2 pixels per frame (follow).
  - Attack Range: 70 pixels.
  - Attack Interval: 5 seconds (attack object).
  - Visibility Range: Determined by Raycasting, up to map boundaries.

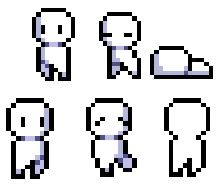
Figure 9. Boss

c. Player

- The Player represents the main character, a lone survivor in a post-apocalyptic world tasked with eliminating enemies to survive. It navigates the map, shoots bullets to combat threats, and aims to defeat the boss to achieve victory.

- Statistics:

  - Health: 10 (represented by hearts, loses 1 per enemy attack, max 4 initially).
  - Damage: 1 (dealt by bullets fired from the gun).



- Speed: 4 pixels per frame (movement speed).
- Initial Position: Center of map (boardWidth/2, boardHeight/2).

Figure 10. Player

d. Bullets

- Bullets are projectiles fired by the player's gun, used to damage and destroy enemies. They travel in four directions (left, right, up, down) and are removed upon hitting an enemy or exceeding map boundaries.

- Statistics:

  - Damage: 1 (deals 1 damage to enemies on impact).
  - Speed: 8 pixels per frame (movement speed, ensuring fast travel across the map).
  - Range: Limited by map boundaries.

Figure 11. Bullet

e. Gun

- The Gun is attached to the player, responsible for spawning bullets based on the player's direction. It ensures accurate firing mechanics, enabling the player to combat enemies effectively.

- Damage: 1 (deals 1 damage to enemies).

Figure 12. Gun

f. Heart

- The Heart visualizes the player's health status, providing feedback on the player's survival state. It updates its display based on the player's health, depleting each enemy attack.

- Count: 4 (initial maximum health)

Figure 13. Heart

V. IMPLEMENTATION

The implementation of Last Day on Earth translates the design into functional code, leveraging data structures and algorithms to support the game's mechanics. This section details the data structures used, the implementation of the A* search algorithm and Raycasting algorithms, and their respective time complexities, aligning with the educational goals of the Data Structure and Algorithm course.

1. The data structures used

- ArrayList:

  The Panel class employs ArrayLists to manage dynamic collections of entities, such as bullets (ArrayList<Bullet>) and warriors (ArrayList<Warrior>). This structure supports efficient addition, removal, and iteration, essential for updating and rendering entities in each frame. The space complexity is O(n), where n is the number of elements, with an average time complexity of O(1) for adding or removing elements, though removal by index can be O(n) due to shifting.

- 2D Array:

  The TileManager class uses a 2D array (mapTileNum: int[][]) to store tile data and a walkableMap: boolean[][] for pathfinding and collision detection. This provides constant-time access to tile information, critical for rendering and navigation. For a map of rows × cols (e.g., 30×15 for the medium map), the space complexity is O(rows × cols), with O(1) access time per element.

- Rectangle:

The Rectangle class from java.awt defines collision and attack areas (attackArea, damageArea, collisionArea) for entities like Warrior and Boss. The intersects method enables O(1) time collision detection per check, supporting real-time entity interactions.

2. Algorithms Implemented

a. A* search algorithm:

The **updatePath** method calculates the shortest path from the warrior's current tile to the player's tile using A*. It uses the **walkableMap** 2D array to determine traversable areas, with the path updated every 0.5 seconds (controlled by lastPathUpdate and PATH_UPDATE_INTERVAL). A* uses a priority queue (implicitly supported by a min-heap structure) to explore nodes, prioritizing those with the lowest estimated total cost (f = g + h), where g is the cost from the start node, and h is the heuristic distance to the goal. The resulting path is stored as a List<Point> and traversed incrementally in the update2 method, adjusting the warrior's position based on speed and delta time.

```
private void updatePath() {
    int startTileX = (int) (x / panel.tileSize) ;
    int startTileY = (int) (y / panel.tileSize) ;
    int goalTileX = (int) (player.x / panel.tileSize);
    int goalTileY = (int) (player.y / panel.tileSize);

    AStar aStar = new AStar(tileM.getWalkableMap());
    path = aStar.findPath(startTileX, startTileY, goalTileX, goalTileY);
    pathIndex = 0;
}
```

Figure 14. updatePath method

```
if (path != null && pathIndex + 1 < path.size() &&
!collisionOn) {
    Point nextStep = path.get(pathIndex + 1);
    double targetX = nextStep.x * panel.tileSize + panel.
    tileSize / 2.0;
    double targetY = nextStep.y * panel.tileSize + panel.
    tileSize / 2.0;

    double deltaX = targetX - x;
    double deltaY = targetY - y;
    double distance = Math.sqrt(deltaX * deltaX + deltaY *
    deltaY);
    if (distance == 0) distance = 1;

    double moveDistance = speed * deltaTime;
    if (distance <= moveDistance) {
        x = (int) targetX;
        y = (int) targetY;
        pathIndex++;
    } else {
        x += (int) ((deltaX / distance) * moveDistance);
        y += (int) ((deltaY / distance) * moveDistance);
    }

    worldX = x;
    worldY = y;
}
```

Figure 15. The path traversal logic

- Data Structures:
    - walkableMap: boolean[][] from TileManager defines the grid.
    - path: List<Point> stores the computed path.
    - An implicit priority queue (within AStar) manages node exploration.
- Time Complexity:

  The worst-case time complexity is O((rows × cols) log (rows × cols)) for a grid of rows × cols (e.g., 30×15 = 450 tiles), due to potential exploration of all nodes. The priority queue adds O(log V) per operation, where V ≤ rows × cols. In practice, the heuristic reduces

explored nodes, making it more efficient (e.g., O(450) for the medium map). Space complexity is O(rows × cols) for the grid and path.

b. Raycasting:

The **canSeePlayer** method in the Boss class initiates the Raycasting process by creating Rectangle objects for the boss (source) and player (target), then delegates to **Raycasting.canSeePlayer**. In Raycasting.java, the algorithm uses a digital differential analyzer (DDA)-like approach to trace a ray from the boss's center (x0, y0) to the player's center (x1, y1). It calculates the step direction (sx, sy) and error term (err) based on the differences (dx, dy), iterating through tiles while checking for collisions against the mapTileNum 2D array. If a non-walkable tile (indicated by tileManager.tile[tileNum].collision) is encountered or the ray goes out of bounds, it returns false. If the ray reaches the target, it returns true, influencing the boss's damage (full damage if visible, 0.5 if hidden) and pursuit behavior. A distance limit of 120 pixels is also enforced to restrict the range.

```
public boolean canSeePlayer() {
    if (panel.tileM.mapTileNum == null) {
        throw new IllegalStateException(s:"mapTileNum is null when calling
        canSeePlayer.");
    }

    Rectangle bossRect = new Rectangle(this.x, this.y, this.width, this.
    height);
    Rectangle playerRect = new Rectangle(player.x, player.y, player.width,
    player.height);
    return Raycasting.canSeePlayer(bossRect, playerRect, panel.tileM.
    mapTileNum, panel.tileM, panel.tileSize);
}
```

Figure 16. canseePlayer method in Boss.java

```
if (canSeePlayer()) {
    double newX = x + speedX2;
    double newY = y + speedY2;
    panel.cChecker.checkTileCollisionBoss(this, (int)speedX2, (int)
    speedY2);
    if (!collisionOn) {
        x = (int) newX;
        y = (int) newY;

        x = Math.max(a:0, Math.min(x, mapWidth - width));
        y = Math.max(a:0, Math.min(y, mapHeight - height));
        worldX = x;
        worldY = y;
    } else {
        int[][] diagonalDirs = {
                {1, 1}, {-1, 1}, {1, -1}, {-1, -1}
        };
        int attempts = 0;
        do {
            int index = rand.nextInt(diagonalDirs.length);
            directionX = diagonalDirs[index][0];
            directionY = diagonalDirs[index][1];
            attempts++;
        } while (
                directionX == -lastDirectionX && directionY ==
                -lastDirectionY &&
                        attempts < 10
        );
        lastDirectionX = directionX;
        lastDirectionY = directionY;
    }
} else {
```

Figure 17. Implement Raycasting Algorithm in update method

```java
public static boolean canSeePlayer(Rectangle source, Rectangle target, int[]
[] mapTileNum, TileManager tileManager, int tileSize) {
    if (mapTileNum == null) {
        throw new IllegalArgumentException(s:"mapTileNum must not be null.
        Ensure it is initialized correctly in TileManager.");
    }
    if (tileManager == null) {
        throw new IllegalArgumentException("tileManager must not be null. Ensure it is initialized correctly.");
    }
    double distance = source.getLocation().distance(target.getLocation());
    if (distance > 120) return false;

    int x0 = source.x + source.width / 2;
    int y0 = source.y + source.height / 2;
    int x1 = target.x + target.width / 2;
    int y1 = target.y + target.height / 2;

    int dx = Math.abs(x1 - x0), dy = Math.abs(y1 - y0);
    int sx = x0 < x1 ? 1 : -1, sy = y0 < y1 ? 1 : -1;
    int err = dx - dy;

    while (true) {
        int tileX = x0 / tileSize;
        int tileY = y0 / tileSize;
```

```java
        // **Bounds Check**: Ensure tileX and tileY are within bounds
        if (tileX < 0 || tileY < 0 || tileX >= mapTileNum.length || tileY
        >= mapTileNum[0].length) {
            return false; // Out of bounds, cannot see target
        }

        int tileNum = mapTileNum[tileX][tileY];
        if (tileManager.tile[tileNum].collision) {
            return false;
        }

        if (x0 == x1 && y0 == y1) break;

        int e2 = 2 * err;
        if (e2 > -dy) {
            err -= dy;
            x0 += sx;
        }
        if (e2 < dx) {
            err += dx;
            y0 += sy;
        }
    }
    return true;
}
```

Figure 18. Raycasting.java

- Data Structures:
    - mapTileNum: int[][] from TileManager defines the map layout.

- Rectangle objects (source, target) represent the boss and player positions.

- Time Complexity:

The time complexity is O(d), where d is the number of tiles along the ray (e.g., maximum diagonal distance in a 30×15 map is approximately 34 tiles). Each tile check is O(1), making the worst-case complexity O($\sqrt{(rows^2 + cols^2)}$), or O(34) for the medium map. Space complexity is O(1), as no additional data structures are stored.

VI. CONCLUSION AND FUTURE WORK

The Last Day on Earth project successfully demonstrates the application of data structures and algorithms in a survival action game, aligning with the objectives of the Data Structure and Algorithm course. This section summarizes the project's achievements, identifies its limitations, and proposes directions for future development, including the integration of additional algorithms and data structures to enhance gameplay and performance.

The project achieved its primary goal of developing a functional 2D survival action game. Key entities such as the Player, Warrior, and Boss were implemented with distinct roles and behaviors, supported by a modular class architecture (Panel, TileManager, CollisionChecker). The game loop in Panel ensures smooth updates and rendering, while the TileManager leverages a 2D array to manage the map efficiently. The Warrior class employs the A* algorithm for intelligent pathfinding, enabling zombies to pursue the player effectively, and the Boss class uses Raycasting to enhance strategic gameplay through line-of-sight mechanics. These implementations showcase practical applications of data structures (ArrayLists, 2D arrays, Rectangle) and algorithms, providing an interactive experience where players can navigate, fight enemies, and engage with a boss encounter.

Beside its achievements, the project has a number of flaws that affect its depth and polish. Visual effects are basic, lacking advanced animations or particle effects, which reduces immersion. There is no high score saving system, limiting replayability.

Additionally, while defeating the boss triggers a win, the game does not transition to new levels, and the Viewpoint system for dynamic camera movement remains unimplemented.

In the future, we plan to make Last Day on Earth more appealing by introducing more enemy types, improving the UI, optimizing performance, and implementing the Viewpoint system. Additional algorithms and data structures can be applied to support these improvements, such as spatial partitioning techniques to enhance collision detection efficiency, alternative pathfinding algorithms for varied enemy behaviors, and state management systems to improve AI complexity. These enhancements would deepen the game's mechanics and further demonstrate the value of Data Structure and Algorithm concepts in game development.

Conclusion, Last Day on Earth roughly showcases the role of data structures and algorithms in creating an engaging survival action game. Despite its limitations, the project provides a foundation for future enhancements, offering opportunities to apply advanced techniques to improve gameplay and performance, in line with the principles of the Data Structure and Algorithm course.