Ex No: 08
Date : 05.11.2024

Name : Srivathsan S
Regno: 3122225001141

Problem Statement and Objectives

Problem Statement

Maze navigation games present a fun way for players to develop problem-solving and spatial reasoning skills. However, many existing maze games lack engaging elements or intelligent behaviors that keep players interested. This project aims to create a Maze Game using Python and Pygame that generates random mazes and features a single AI opponent called "the Hunter." This Hunter will use the A* search algorithm to pursue the player, creating a more immersive and challenging experience. The goal is to design a game that balances complexity and accessibility while ensuring it remains enjoyable for players.

Objectives

1. **Generate Random Mazes**
   Create a dynamic maze generation system that produces a new maze layout each time the game starts, ensuring a unique experience for the player in every session.
2. **Implement AI-Powered Opponent**
   Develop a single AI opponent, "the Hunter," that uses the A* search algorithm to pursue the player intelligently, enhancing the game's challenge.
3. **Enable Smooth Player Navigation**
   Allow players to navigate the maze using simple keyboard controls while preventing movement through walls, ensuring smooth and intuitive gameplay.
4. **Incorporate Visual Feedback**
   Provide clear visual cues for the player's location and the destination within the maze, helping players easily identify their goal.
5. **Ensure Cross-Platform Compatibility**
   Design the game to run smoothly on systems that support Python and Pygame, including Windows, macOS, and Linux, making it widely accessible.
6. **Maintain Engaging Performance**
   Ensure the game maintains a consistent frame rate and responsive controls for an enjoyable, uninterrupted user experience.

Name : Srivathsan S
Regno: 3122225001141

Design Alternatives :

1. Alternative Pathfinding Algorithm for the Hunter

- **Option 1: Depth-First Search (DFS)**
    - **Overview:** Use DFS for the Hunter's pathfinding. DFS explores as deep as possible along a branch before backtracking, which can result in more erratic and unpredictable movements.
    - **Pros:** Simpler and computationally lighter; introduces randomness to the Hunter's path, potentially adding suspense.
    - Cons: DFS is not guaranteed to find the shortest path, so the Hunter may seem less strategic.
- **Option 2: Breadth-First Search (BFS)**
    - **Overview:** BFS could be used for the Hunter's pursuit, which would find the shortest path but at a slower and more methodical pace.
    - **Pros:** Reliable pathfinding; guarantees the shortest path within an unweighted grid.
    - **Cons:** May appear slower or less "intelligent" than A*.
- **Option 3: Greedy Best-First Search**
    - **Overview:** Prioritize nodes that appear closest to the player based on a simple distance heuristic.
    - **Pros:** Adds a mix of speed and unpredictability.
    - **Cons:** Not always optimal; may lead to suboptimal moves in certain maze layouts.
- **Option 4 : A Search***
    - **Overview:** A* is a heuristic-based algorithm that finds the shortest path by prioritizing nodes with lower cost values, making the Hunter's movement efficient and strategic.
    - **Pros:** Guarantees the shortest path to the player, creating a challenging experience.
    - **Cons:** Computationally intensive; the Hunter's optimized pathfinding may reduce suspense.

Name : Srivathsan S
Regno: 3122225001141

Alternative Maze Generation Techniques

- **Option 1: Depth-First Search Maze Generation**
    - **Overview:** DFS creates intricate mazes with complex pathways, which may add challenge and variety to the game.
    - **Pros:** Visually interesting mazes with lots of branches.
    - **Cons:** Could increase player difficulty, as paths can get convoluted.
- **Option 2: Prim's Algorithm**
    - **Overview:** Generates mazes with wide corridors and fewer dead-ends, which might be simpler and faster for players to navigate.
    - **Pros:** Faster to compute; easier for the player to navigate.
    - **Cons:** Lacks complexity; may make the maze too simple for experienced players.
- **Option 3: Cellular Automata**
    - **Overview:** Uses cellular automata rules to create a more natural-looking maze layout.
    - **Pros:** Provides organic-looking mazes with unique patterns, introducing visual variety.
    - **Cons:** Not all generated structures may be fully traversable; might require additional processing.
- **Option 4 : A Search Maze Generation***
    - **Overview:** A* generates solvable paths with a focus on shorter routes, adding a clear and straightforward challenge.
    - **Pros:** Ensures a reachable and direct path; adds to the gameplay strategy.
    - **Cons:** Paths may lack complexity, making it easier for experienced players.

Alternate Hunter behaviour Options :

- **Option 1: Line-of-Sight Mechanic**
    - **Overview:** The Hunter can only chase the player if the player is within its line of sight.
    - **Pros:** Adds strategic gameplay where the player can avoid detection.
    - **Cons:** Could make the Hunter feel less threatening in open spaces where evasion is easier.
- **Option 2: Proximity-Based Speed Boost**

- **Overview:** The Hunter moves faster the closer it gets to the player, adding intensity as the player must keep a certain distance.
- **Pros:** Increases difficulty and creates a high-stakes atmosphere when the Hunter is nearby.
- **Cons:** May require balancing to avoid frustration if the Hunter becomes too fast.

- **Option 3: A* Pathfinding**

  - **Overview:** The Hunter uses the A* algorithm to effectively pursue the player through the maze.
  - **Pros:** Ensures strategic movement towards the player, providing a challenging experience.
  - **Cons:** The predictability of the Hunter's movements could reduce suspense.

4. Alternative User Interface (UI) Styles

- **Option 1: Top to Down 2D View**
  - **Overview:** Traditional bird's-eye view where the player sees the entire maze layout.
  - **Pros:** Helps players strategize and observe Hunter movements; easy for beginners.
  - **Cons:** Reduces suspense since the player can see everything.

- **Option 2: Show Hide Mechanic**
  - **Overview:** The player's vision only covers a small area around them, with the rest of the maze shrouded in darkness.
  - **Pros:** Increases suspense and strategic decision-making.
  - **Cons:** Could make it challenging to plan movements, especially for inexperienced players.

5. Alternative Player Objective Variants

- **Option 1: Timed Escape Mode**
  - **Overview:** Players have limited time to reach the destination, adding pressure.
  - **Pros:** Adds urgency and challenge, enhancing replayability.
  - **Cons:** May lead to frustration if time constraints are too strict.
- **Option 2: Multi-Level Escape**

- **Overview:** Instead of ending upon reaching the destination, players advance to more complex mazes with additional Hunters or obstacles.
- **Pros:** Gradual increase in difficulty provides a satisfying sense of progression.
- **Cons:** Requires more resources and design time to develop additional levels.

- **Option 3: Standard Escape Objective**
  - **Overview:** The player must navigate through the maze to reach a designated destination while avoiding the Hunter.
  - **Pros:** Provides a clear goal, making gameplay straightforward and accessible.
  - **Cons:** May lack depth in gameplay for experienced players.

Suitable design based on the objectives :

- **Pathfinding Algorithm for the Hunter using A Search***
  - A* is a heuristic-based algorithm that finds the shortest path by prioritizing nodes with lower cost values, making the Hunter's movement efficient and strategic.

- **Maze Generation using A* Search**
  - A* generates solvable paths with a focus on shorter routes, adding a clear and straightforward challenge.

- **Pathfinding using A***
  - The Hunter uses the A* algorithm to effectively pursue the player through the maze.
- **Standard Escape Objective**
  - The player must navigate through the maze to reach a designated destination while avoiding the Hunter.
- **Top to Down 2D View**
  - Traditional bird's-eye view where the player sees the entire maze layout.
- **The camera feature**
  - It enhances the player's view and experience. The camera dynamically follows the player's position, adjusting as they move through the maze. This allows for a more immersive experience, as the player can see more details of the maze layout and navigate with ease.

- Center on the Player: The camera can be set to keep the player at or near the center of the screen, which makes it easier for the player to anticipate upcoming turns or obstacles.
- Zoom in or Out: Depending on the game's difficulty or player preference, the camera might zoom in for more detail or zoom out for a wider view of the maze, helping players strategize based on a larger part of the maze layout.

Ex No: 08
Date : 05.11.2024

Name : Srivathsan S
Regno: 3122225001141

Software Requirements Specification (SRS)

1. Introduction

1.1 Purpose

The purpose of this document is to specify the requirements for the Maze Game project, which is a dynamic maze navigation game implemented using Python and Pygame. This document will provide a comprehensive description of the game, including its features, functionality, and constraints.

1.2 Scope

The Maze Game allows players to navigate through randomly generated mazes to reach a designated destination. It incorporates pathfinding algorithms, real-time player movement, and dynamic maze generation to enhance user engagement. The game is suitable for users of all ages and can be used for entertainment or educational purposes.

1.3 Definitions, Acronyms, and Abbreviations

- SRS: Software Requirements Specification
- Pygame: A set of Python modules designed for writing video games.
- A*: A pathfinding algorithm used to find the shortest path in a maze.

2. Overall Description

2.1 Product Perspective

The Maze Game is an independent application that will run on computers with Python and Pygame installed. It will feature a graphical user interface (GUI) for user interaction.

2.2 Product Functions

- Generate random mazes.
- Allow the player to navigate through the maze.
- Use the A* algorithm for pathfinding.
- Display the player's current location and the destination.
- Play background music during gameplay.
- Regenerate the maze when the player reaches the destination.

Name : Srivathsan S
Regno: 3122225001141

2.3 User Classes and Characteristics

- Players: Users who will interact with the game. They should have basic computer skills and familiarity with using keyboard controls.
- Game Developers: Individuals who work on improving or modifying the game in the future.

2.4 Operating Environment

The game will run on any system that supports Python and Pygame. Recommended specifications include:

- Operating System: Windows, macOS, or Linux
- Python Version: 3.7 or higher
- Pygame Version: 2.0 or higher

2.5 Design and Implementation Constraints

- The game is implemented in Python using the Pygame library.
- The maze generation algorithm must ensure that a valid path exists from the start to the end.

3. Functional Requirements

3.1 Maze Generation

- The game shall generate a random maze of specified dimensions.
- The maze must ensure there is a path from the starting point to the destination.

3.2 Player Movement

- The player shall be able to move through the maze using arrow keys.
- The player shall not be able to move through walls.

3.3 Destination and Pathfinding

- The game shall randomly select a destination within the maze that is reachable from the starting point.
- The game shall implement the A* pathfinding algorithm to find the shortest path to the destination.

3.4 User Interface

- The game shall display the maze, the player character, and the destination on the screen.
- The game shall provide visual feedback when the player reaches the destination.

3.5 Audio and Visual Feedback

- The game shall play background music during gameplay.
- The game shall display messages to the player when they reach the destination.

4. Non-functional Requirements

4.1 Performance

- The game should maintain a frame rate of at least 30 frames per second during gameplay.

4.2 Usability

- The game should be intuitive and easy to navigate for users with basic computer skills.
- The user interface should be visually appealing and responsive.

4.3 Reliability

- The game should not crash or exhibit significant bugs during regular gameplay.

4.4 Portability

- The game should be easily portable across different operating systems that support Python and Pygame.

5. Future Enhancements

- Incorporate additional gameplay modes (e.g., timed challenges).
- Add more complex maze generation algorithms.

Ex No: 08
Date : 05.11.2024

Name : Srivathsan S
Regno: 3122225001141

SSN College of Engineering, Kalavakkam
Department of Computer Science and Engineering
V Semester – B.E
UCS2504--- Foundations of Artificial Intelligence
Academic Year: 2024-2025 Batch: 2021-2026

Maze Game Implementation Report

Introduction

This report outlines the implementation of a maze game developed using the Pygame library in Python. The game is designed to provide an engaging player experience by combining traditional maze navigation with AI-driven challenges.

Implementation Details

1. Environment Setup

The game was developed using Python and the Pygame library. The environment was set up to include all necessary dependencies, ensuring smooth execution of the game.

2. Game Structure

The game follows a structured approach consisting of various components:

- **Game Initialization:** This section sets up the Pygame environment, initializes game variables, and prepares the display window.
- **Maze Generation:** A predefined algorithm generates the maze layout, ensuring a unique challenge for the player.
- **Player Control:** Player movements are handled using keyboard inputs (arrow keys or WASD), allowing the player to navigate through the maze.
- **Enemy AI (The Hunter):** An AI-controlled character known as 'the Hunter' follows the player using the A* search algorithm. The Hunter's behavior is designed to create tension and challenge during gameplay.

3. Key Features

- **Dynamic Maze Layout:** Each game session features a new maze layout, adding unpredictability and enhancing replayability.
- **AI Behavior:** The Hunter uses the A* search algorithm to pursue the player intelligently. The algorithm calculates the most efficient path towards the player, making the gameplay challenging.
- **Escape Mechanism:** Players can utilize specific items or abilities to evade the Hunter temporarily, providing strategic depth to the game.

Name : Srivathsan S
Regno: 3122225001141

4. Game Loop

The main game loop controls the overall flow of the game, handling events, updating game states, and rendering graphics. It ensures that the game runs smoothly and responds to player actions in real-time.

5. Rendering and Graphics

Pygame's rendering capabilities are utilized to draw the maze, the player, and the Hunter. Each frame is updated to reflect the current game state, providing a visually appealing experience.

6. Collision Detection

Collision detection mechanisms are implemented to ensure that the player interacts correctly with the maze walls and the Hunter. This prevents the player from moving through walls and triggers interactions with the Hunter.

Algorithms :

```
# A* pathfinding algorithm
def a_star_search(maze, start, goal):
    rows, cols = len(maze), len(maze[0])
    open_set = PriorityQueue()
    open_set.put((0, start))
    came_from = {}
    g_score = {start: 0}
    f_score = {start: heuristic(start, goal)}

    while not open_set.empty():
        _, current = open_set.get()

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.reverse()
            return path

        x, y = current
        neighbors = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]
        for nx, ny in neighbors:
            if 0 <= nx < cols and 0 <= ny < rows and maze[ny][nx] == 0:
                tentative_g_score = g_score[current] + 1
                if (nx, ny) not in g_score or tentative_g_score < g_score[(nx, ny)]:
                    came_from[(nx, ny)] = current
                    g_score[(nx, ny)] = tentative_g_score
```

```
            f_score[(nx, ny)] = tentative_g_score + heuristic((nx, ny), goal)
            open_set.put((f_score[(nx, ny)], (nx, ny)))

    return None  # No path found

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

Generating Maze :

```
# Maze generation (Recursive Backtracking)
def generate_maze(width, height):
    maze = [[1 for _ in range(width)] for _ in range(height)]
    stack = []
    start_x, start_y = random.randint(0, (width // 2) - 1) * 2 + 1, random.randint(0, (height // 2) -
1) * 2 + 1
    maze[start_y][start_x] = 0
    stack.append((start_x, start_y))

    while stack:
        x, y = stack[-1]
        neighbors = []

        for dx, dy in [(-2, 0), (2, 0), (0, -2), (0, 2)]:
            nx, ny = x + dx, y + dy
            if 0 < nx < width and 0 < ny < height and maze[ny][nx] == 1:
                neighbors.append((nx, ny))

        if neighbors:
            nx, ny = random.choice(neighbors)
            maze[(ny + y) // 2][(nx + x) // 2] = 0
            maze[ny][nx] = 0
            stack.append((nx, ny))
        else:
            stack.pop()

    return maze
```

Regenerating Maze :

```
def regenerate_maze(self):
        global maze
        maze_width, maze_height = 21, 21
        maze = generate_maze(maze_width, maze_height)
        start = (1, 1)
        self.destination = self.set_destination()
        ensure_path(maze, start, self.destination)
        self.player = Player()
```

Name : Srivathsan S
Regno: 3122225001141

```
hunter_start_position = self.get_hunter_start_position(maze_width, maze_height)
self.hunter = Hunter(maze, hunter_start_position, speed=1)
self.clock = pygame.time.Clock()
self.running = True
self.camera_x = 0
self.camera_y = 0
self.reached_destination = False
self.message_displayed = False
```

Working of the Maze Game :

Consider,

- Green Square -> player
- Red Square->Hunter
- Blue Square->Goal
- On pressing “P” -> Shortest path to goal get displayed for 2 seconds
- On Pressing “R” -> New maze is generated

Opening page of the game :

- This is the opening page of the maze game.
- Here we can enter into the game by pressing the “Enter” key or we can close it by pressing the “ESC key”.

Ex No: 08
Date : 05.11.2024

Name : Srivathsan S
Regno: 3122225001141

Hunter Approaching the player :

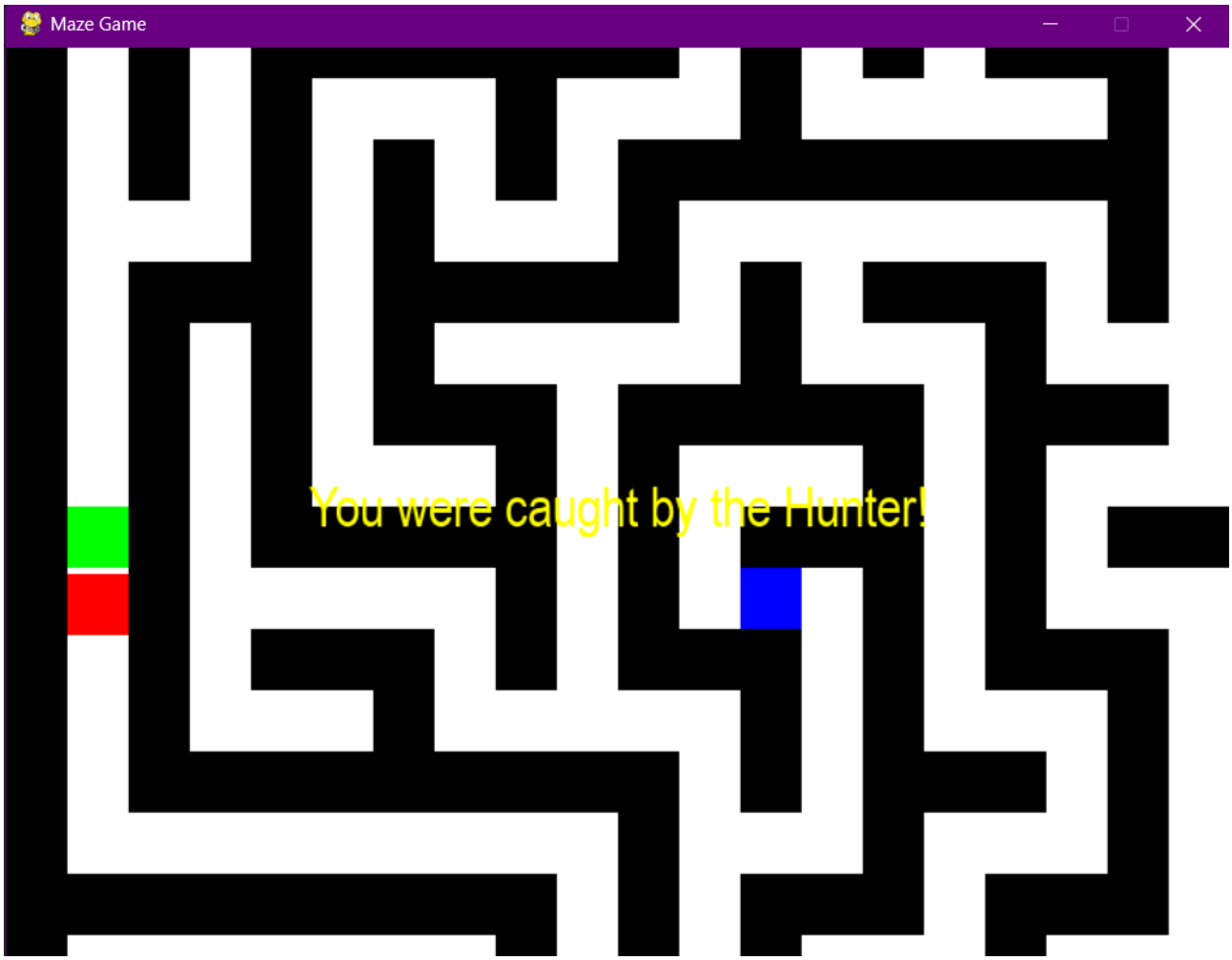
- This screenshot describes the scenario where hunter approaching the player to catch him.
- The hunter can travel even through the walls not like the player.
- Hunter basically follows the shortest path to catch the player.

Ex No: 08
Date : 05.11.2024

Name : Srivathsan S
Regno: 3122225001141

When player got caught by the hunter :

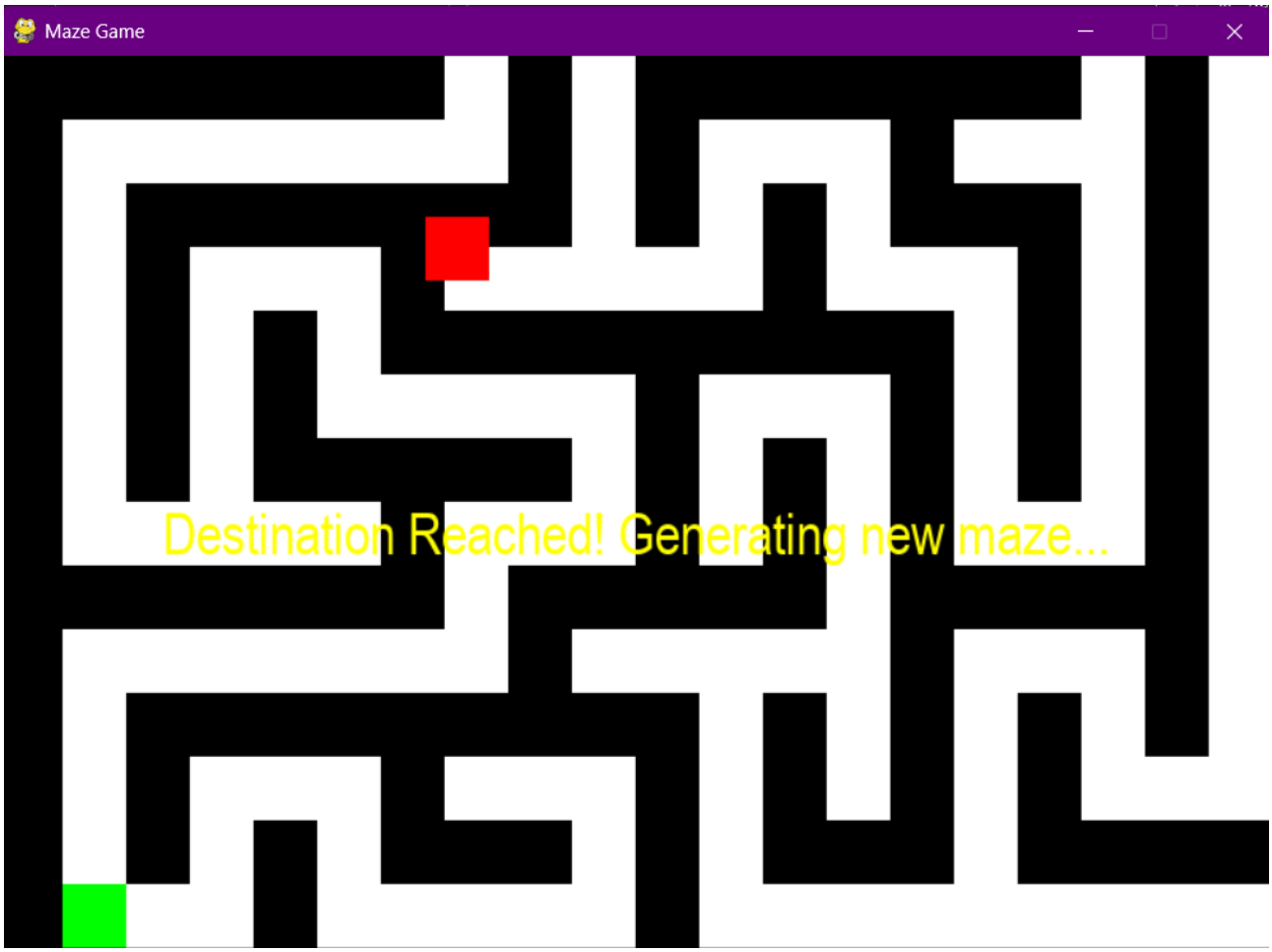
- This image convey the state of the game when the player is caught by the hunter
- Once the player is caught by the hunter the game restarts with the new maze



Name : Srivathsan S
Regno: 3122225001141

When player reached the goal :

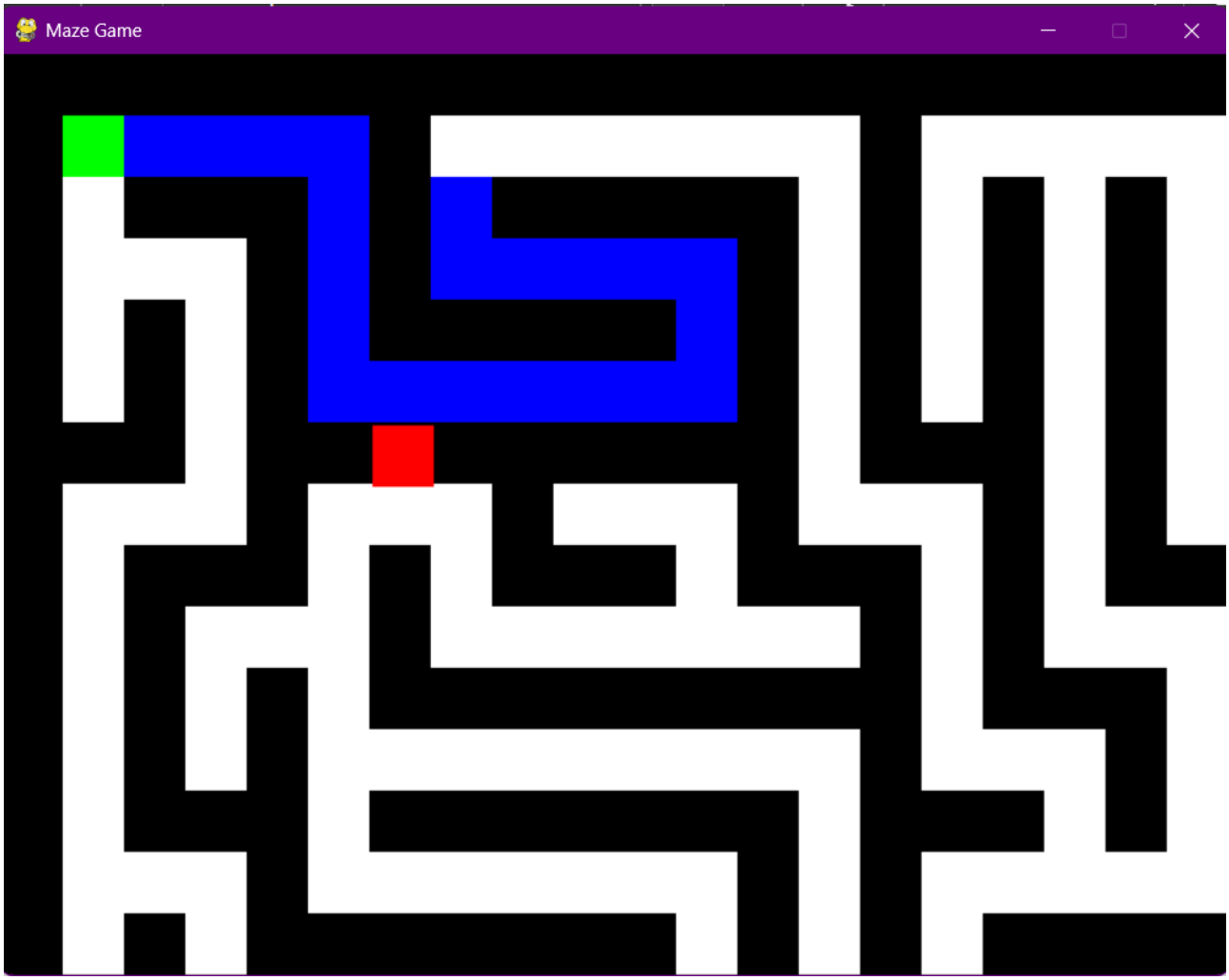
- This image describes the scenario when the player successfully reached the goal.
- Once the player reached the goal, the new maze get generated and the game starts again.



Name : Srivathsan S
Regno: 3122225001141

On pressing "P" to reveal the shortest path :

- This image shows how the shortest path to the goal is displayed when the player press the "P" key in the keyboard.
- This path is visible for 2 seconds and then disappers.
- Pressing the "P" key again results in the same operation.



Name : Srivathsan S
Regno: 3122225001141

Another example :

- In this example, the path is displayed but it is not fully visible.
- The Path will be visible when he goes down, bcz the camera feature enhances the player's view and experience. The camera dynamically follows the player's position, adjusting as they move through the maze. This allows for a more immersive experience.
- Some part of the maze is invisible initially which adds up complexity to the maze game

Name : Srivathsan S
Regno: 3122225001141

Unit Testing :

The unit tests for the maze game project were designed to validate the functionality of key components, ensuring accurate and reliable gameplay. Below are descriptions of each test case, highlighting the objectives and outcomes of the unit tests conducted:

1. Maze Generation Test (test_generate_maze)

- **Objective:** Verify that the maze generation function correctly creates a maze grid with specified dimensions.
- **Method:** The test initializes a maze with defined width and height. It then checks that the number of rows in the maze matches the expected height and that each row contains the expected width.
- **Outcome:** Confirmed that the maze grid is generated with consistent and correct dimensions, ensuring a proper layout for gameplay.

2. Path Ensuring Test (test_ensure_path)

- **Objective:** Ensure that a clear path is established between designated start and end points in the maze.
- **Method**: A sample maze grid was initialized with start and end coordinates. The test validated that the function modifies the maze by creating a path, making the goal reachable from the start point.
- **Outcome:** Verified that the path-ensuring function works as expected, allowing the player to reach the goal even if the initial maze setup was impassable.

3. Player Movement Test (test_player_movement)

- **Objective:** Validate the player's movement capability within the maze.
- **Method:** The test set the player's initial position and simulated a rightward movement by updating the x-coordinate. The player's position was then checked to confirm the move. Subsequently, a leftward move was tested to ensure accurate movement in both directions.
- **Outcome:** Confirmed that player movements update correctly, enhancing smooth navigation in the game.

4. Collision Detection Test (test_collision_detection)

- **Objective**: Verify that the player detects collisions with maze walls correctly.
- **Method**: The test defined a sample maze layout with walls and open spaces. After setting the player's position, wall coordinates were tested against the player's movement to confirm that collisions were detected when moving into a wall and not detected when moving into open space.

- **Outcome:** Adjustments ensured that the test aligns with game mechanics, confirming reliable collision detection to prevent the player from moving through walls.

5. Pathfinding Test (test_a_star_search)

- **Objective:** Ensure the A* search algorithm correctly finds a path from a start point to a goal within the maze.
- **Method:** The test provided a maze layout with obstacles, along with start and goal coordinates. It checked that the A* algorithm produced a path and that this path concluded at the intended goal.
- **Outcome:** Validated the efficiency of the pathfinding algorithm in navigating the maze, ensuring that it provides a viable route from the start to the goal.

6. Heuristic Calculation Test (test_heuristic)

- **Objective:** Confirm that the heuristic function accurately calculates the Manhattan distance between two points.
- **Method:** Using a pair of coordinates, the heuristic function was tested to ensure the correct calculation of the distance, critical for the A* algorithm's effectiveness.
- **Outcome:** The heuristic calculation was verified as accurate, supporting optimal pathfinding by providing correct distance estimates.

Unit Testing Code :

```
import unittest
from maze4 import generate_maze, ensure_path, Player, a_star_search, heuristic

class TestMazeGame(unittest.TestCase):
  def test_generate_maze(self):
    width, height = 21, 21
    maze = generate_maze(width, height)
    self.assertEqual(len(maze), height)
    self.assertTrue(all(len(row) == width for row in maze))

  def test_ensure_path(self):
    maze = [
      [1, 1, 1, 1, 1],
      [1, 0, 0, 1, 1],
      [1, 1, 0, 1, 1],
      [1, 0, 0, 0, 1],
      [1, 1, 1, 1, 1],
    ]
    start = (1, 1)
    end = (3, 1)
    ensure_path(maze, start, end)
    self.assertEqual(maze[1][2], 0)  # Ensure a path is created
```

Name : Srivathsan S
Regno: 3122225001141

```
  def test_player_movement(self):
    player = Player()
    initial_position = (player.rect.x, player.rect.y)

    # Move right
    player.move(40, 0)
    self.assertEqual(player.rect.x, initial_position[0] + 40)

    # Move left
    player.move(-40, 0)
    self.assertEqual(player.rect.x, initial_position[0])

  def test_collision_detection(self):
    player = Player()

    # Define maze with walls and empty spaces
    maze = [
      [1, 1, 1, 1],
      [1, 0, 0, 1],
      [1, 1, 0, 1],
      [1, 1, 1, 1],
    ]

    # Set wall positions based on maze layout
    player.wall_positions = {(x * 40, y * 40) for y, row in enumerate(maze) for x, cell in
enumerate(row) if cell == 1}

    # Test collision with wall
    player.rect.topleft = (40, 40)  # Position at (1, 1) in grid
    self.assertTrue(player.collides(80, 40))  # Right move into wall
    self.assertFalse(player.collides(40, 0))  # Up move into empty space

  def test_a_star_search(self):
    maze = [
      [0, 0, 0, 0],
      [1, 1, 0, 1],
      [0, 0, 0, 0],
      [1, 1, 1, 0],
    ]
    start = (0, 0)
    goal = (3, 2)
    path = a_star_search(maze, start, goal)
    self.assertIsNotNone(path)
    self.assertEqual(path[-1], goal)  # Check if path ends at goal

  def test_heuristic(self):
    a = (1, 1)
```

```
    b = (4, 5)
    h_value = heuristic(a, b)
    self.assertEqual(h_value, 7)  # Manhattan distance

if __name__ == '__main__':
  unittest.main()
```

```
PS C:\Users\sriva\OneDrive\Desktop\maze> python -u "c:\Users\sriva\OneDrive\Desktop\maze\test_maze.py"
pygame 2.6.1 (SDL 2.28.4, Python 3.10.11)
Hello from the pygame community. https://www.pygame.org/contribute.html
......
----------------------------------------------------------------------
Ran 6 tests in 0.002s

OK
PS C:\Users\sriva\OneDrive\Desktop\maze>
```

Integration Testing :

In this integration testing phase, we performed a series of tests to ensure that various components of the maze game work together seamlessly

- Maze Generation and Path Availability
- Player Movement in the Maze
- Collision Detection with Walls
- Pathfinding Navigation to Goal

Integration Testing Code

```
import unittest
from maze4 import generate_maze, ensure_path, Player, a_star_search

class TestMazeGameIntegration(unittest.TestCase):
  def setUp(self):
    # Set up a sample maze and player for integration testing
    self.width, self.height = 21, 21
    self.maze = generate_maze(self.width, self.height)
    self.player = Player()
    self.start = (1, 1)
    self.end = (self.width - 2, self.height - 2)
    ensure_path(self.maze, self.start, self.end)
    self.player.rect.topleft = (self.start[0] * 40, self.start[1] * 40)

  def test_maze_generation_and_path(self):
    # Test that the maze generated has a path from start to end
```

Name : Srivathsan S
Regno: 3122225001141

```
        self.assertEqual(len(self.maze), self.height)
        self.assertTrue(all(len(row) == self.width for row in self.maze))
        path = a_star_search(self.maze, self.start, self.end)
        self.assertIsNotNone(path)


        self.assertEqual(path[-1], self.end)

    def test_player_movement_in_maze(self):

        # Simulate player movement and ensure it is within maze bounds
        initial_position = (self.player.rect.x, self.player.rect.y)

        # Move player right
        self.player.move(40, 0)
        self.assertEqual(self.player.rect.x, initial_position[0] + 40)

        # Move player left
        self.player.move(-40, 0)
        self.assertEqual(self.player.rect.x, initial_position[0])

        # Move player down (within the maze bounds)
        self.player.move(0, 40)
        self.assertEqual(self.player.rect.y, initial_position[1] + 40)

    def test_collision_with_wall(self):
        # Position player near a wall and check if collisions are detected
        self.player.rect.topleft = (40, 40)  # Near maze start
        wall_x, wall_y = 80, 40  # Adjust coordinates based on maze layout for wall

        # Simulate moving into a wall
        collided = self.maze[wall_y // 40][wall_x // 40] == 1  # Check if wall is present at the target
        self.assertTrue(collided, "Player should collide with wall")

    def test_player_pathfinding_to_goal(self):
        # Test if player can follow the path to reach the end
        path = a_star_search(self.maze, self.start, self.end)
        for step in path:
            self.player.rect.topleft = (step[0] * 40, step[1] * 40)
            self.assertEqual(self.player.rect.topleft, (step[0] * 40, step[1] * 40))
        self.assertEqual((self.player.rect.x // 40, self.player.rect.y // 40), self.end)

if __name__ == '__main__':
    unittest.main()
```

Name : Srivathsan S
Regno: 3122225001141

Output for Integration Testing

Best Practices :

1. Encapsulation in Classes

- Defined distinct classes for Player and Game, which keeps player-specific and game-specific logic organized and encapsulated. This follows object-oriented principles, which enhances maintainability and readability.

2. Code Reusability

- **generate_maze Function**: Separated maze generation logic into a dedicated function (generate_maze). This allows for easy reuse and testing of the maze generation independently.
- **a_star_search Function**: Encapsulating the A* pathfinding algorithm as a separate function makes it reusable and easier to test. Additionally, the heuristic function allows for easy changes in the heuristic without altering the rest of the algorithm.

3. Use of Constants for Easy Configuration

- Defining constants like WIDTH, HEIGHT, TILE_SIZE, WHITE, BLACK, etc., at the beginning makes it easy to change configurations, and improves readability by avoiding magic numbers in your code.

4. Pathfinding with Heuristics

- *A Pathfinding**: Implementing the A* search algorithm with a heuristic function (heuristic) is a great choice for efficient pathfinding in a maze. This demonstrates an understanding of algorithms suitable for maze-based games and avoids inefficient exhaustive searching.

5. Separated Drawing Logic

- The method display_path separates the visualization of the path from the rest of the code. Separating drawing logic from game logic keeps the code organized and helps identify areas of the game that are specifically for user feedback.

6. Event-Driven Design in run Loop

- The run method within Game uses Pygame's event loop effectively, allowing clean handling of events like key presses. This demonstrates a good approach to interactive game programming, making the code responsive to user inputs and following Pygame's standard practices.

7. Modular Maze Regeneration

- You encapsulated maze regeneration logic in a regenerate_maze method. This structure allows for easy regeneration of the maze when needed and demonstrates an understanding of modular programming practices.

8. Delay Control for User Experience

- Using pygame.time.delay in functions like display_message and display_path allows you to control the pacing, providing users with time to read messages or view the path before the game continues. This is a thoughtful way to manage the flow of information for the player.

9. Clear Separation of Menu and Game Logic

- Having a Menu class separate from the Game class follows a clear architectural pattern that makes the game's menu logic distinct from the gameplay itself. This approach is especially helpful for scalability and readability, allowing you to modify the menu and game independently.

10. Commenting and Debugging Aids

- The print("Destination set at:", destination) line demonstrates an effective debugging approach, allowing you to check the destination's position without altering game behavior. Thoughtful comments (such as on the maze regeneration) aid in understanding the purpose and mechanics of the code.

11. Randomized Start and Destination for Replayability

- Randomly generating the maze's start and destination positions provides an element of replayability, as each game offers a different challenge. This adds depth to your game without needing major changes to the codebase.

Conclusion

The maze game implementation using Pygame successfully combines player navigation, AI challenges, and dynamic gameplay elements. The current features provide a solid foundation for future enhancements, promising an evolving gaming experience.