Maze Game with Procedural Generation

BUILT WITH PYTHON AND PYGAME

PRESENTER'S: ICT GROUP IV

Introduction

- **What is the Maze Game**?
  - A game where players navigate a procedurally generated maze to reach a goal. Collect coins along the way to maximize the score.
- Key Features:

  Procedural maze generation.

  Dynamic wall placement.

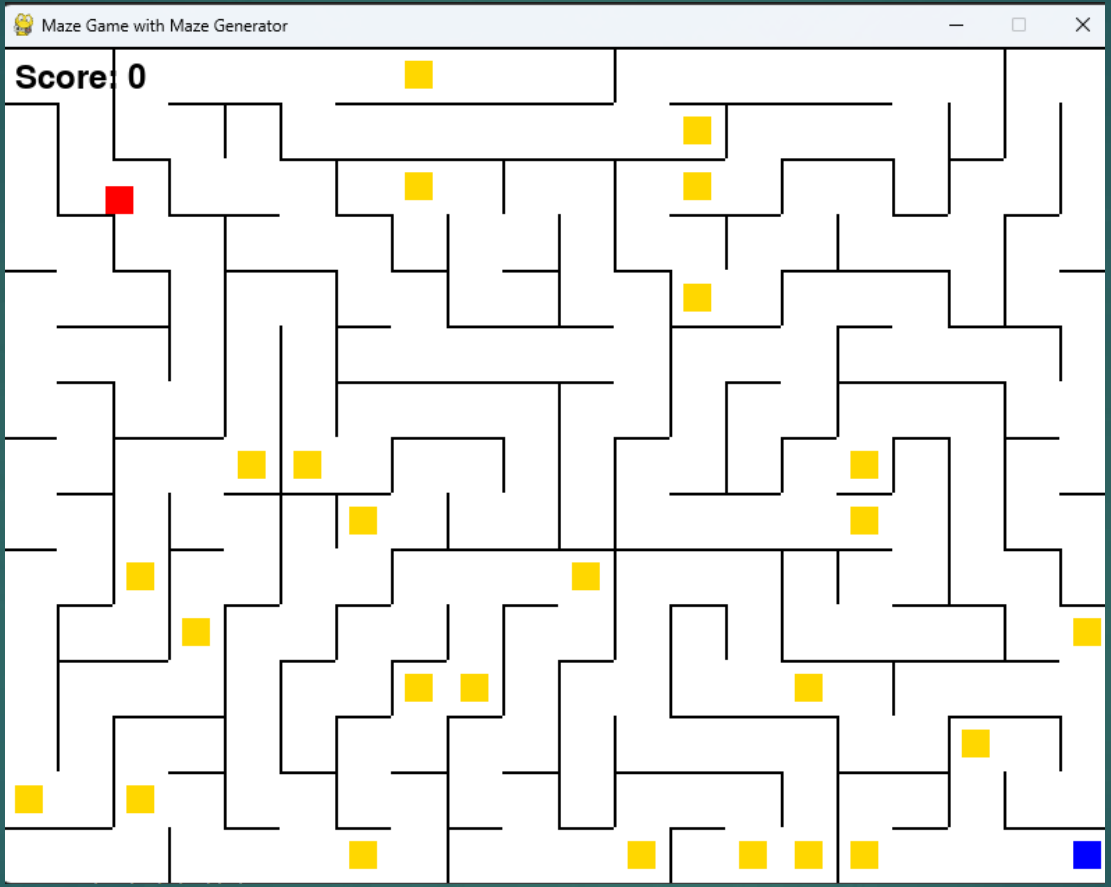
  Collectible coins.
- Winning condition: Reach the goal.

Game Development Tools

- **Language**: Python
- **Library**: Pygame (used for rendering graphics, handling input, and game logic)
- **Why Pygame?**
  - Simple to use.
  - Powerful for 2D games.

How the Game Works

- **Objective:**
  - Navigate through the maze, avoiding walls.
  - Collect coins to increase your score.
  - Reach the goal to win.
- **Gameplay:**
  - Player controls a red square.
  - The goal is a blue square located at the opposite corner of the maze.
  - Coins are scattered throughout the maze and appear as golden squares.


Maze Game with Maze Generator
Score: 0

How the Maze is Generated

- **Recursive Backtracking Algorithm**:
  - Starts from the top-left corner.
  - Randomly explores neighboring cells.
  - Removes walls between cells to create paths.
  - Ensures that all areas of the maze are reachable.
- **Dynamic Walls**:
  - Based on the maze grid, walls are generated to outline paths.
- **Visual Representation:**
  - Grid cells represent potential paths and walls.
  - Walls are drawn along the edges of blocked cells.

Recursive Backtracking?

- Recursive backtracking is an algorithmic approach to generating mazes. It works by simulating a "carving" process, where paths are created between cells in a grid until a complete maze is formed. Here's a step-by-step explanation:
- Key Concepts of Recursive Backtracking
  - Maze as a Grid
    - A maze is represented as a grid of cells, where each cell can have walls on its four sides (top, bottom, left, right). o The goal is to "carve out" paths between cells by removing walls while ensuring that there are no disconnected regions
  - Recursive Backtracking:
    - This approach involves exploring all possible paths from the current cell until the maze is fully carved.
    - If the algorithm reaches a "dead end" (a cell with no unvisited neighbors), it backtracks to the previous cell and tries other possible paths.
    - The algorithm calls itself to explore neighboring cells, making it recursive. o The "backtracking" occurs when there are no valid moves left from the current cell, and the function returns to the previous call to continue carving

Algorithm in Steps

- Start at a Random Cell:
  - Begin carving the maze at a random starting point (commonly the top-left corner (0, 0)).
- Mark the Current Cell as Visited:
  - Keep track of which cells have been visited to avoid revisiting and overwriting paths
- Check for Unvisited Neighbors
  - Look at all four potential neighbors of the current cell (up, down, left, right). o If a neighbor is unvisited and within bounds, it's a valid candidate for carving.
- Choose a Random Neighbor:
  - Randomly select one of the valid neighbors to ensure that the maze is unpredictable and unique each time.

- Remove the Wall:
  - Remove the wall between the current cell and the selected neighbor, creating a path between them.
- Move to the Neighbor
  - Recursively call the algorithm on the neighbor to continue carving paths.
- Backtrack When Stuck:
  - If a cell has no unvisited neighbors, backtrack to the previous cell in the path and explore other directions. o This is where the "stack" in the algorithm comes in—it keeps track of the path so you can retrace your steps when needed.
- Repeat Until All Cells Are Visited:
  - Continue this process until every cell in the maze has been visited and connected.

- Why It Works for Maze Generation
  - Connectivity: Recursive backtracking ensures that all cells are connected, meaning there are no isolated areas in the maze.
  - Single Solution: This algorithm generates mazes with a single unique path between any two points, making it perfect for puzzle-like mazes.
  - Randomness: The random selection of neighbors introduces variability, so every maze generated is different

Code Breakdown

- **Game Initialization**:

```
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
```

- Initializes the Pygame library.
- Sets up the screen with the specified dimensions

- **Player and Goal Setup**:

```
player_size = CELL_SIZE // 2
player_pos = [CELL_SIZE // 4, CELL_SIZE // 4]
goal = pygame.Rect(WIDTH - CELL_SIZE + CELL_SIZE // 4, HEIGHT - CELL_SIZE + CELL_SIZE // 4, CELL_SIZE // 2, CELL_SIZE // 2)
```

- Defines the player’s size and starting position.
- Defines the goal’s size and position.

- **Maze Generator**:

```
def generate_maze():
    ...
```

- Recursive functions creates paths by removing walls
- Converts a grid into playable walls

- **Wall and Coin Placement**:

```
def create_walls_from_maze():
    ...
```

- Converts the maze into walls visible on the screen
- Randomly places coins with a 10 % chance per cell

- Game Loop
  - Handles User Input
  - pygame.key.get_pressed()
- Collision Detection

```
def check_collision(player_rect, walls):
  for wall in walls:
      if player_rect.colliderect(wall):
          return True
```

  - Ensures the player cannot move through walls.
- **Coin Collection**:

```
for coin in coins[:]:
   if player_rect.colliderect(coin):
       coins.remove(coin)
       score += 1
```

  - Checks if the player collects a coin and updates the score.

- **Winning and Scoring**
  - Winning Condition
    - player must reach the goal without hitting walls.
  - Scoring
    - Each coin collected adds to the score.
    - Final score is displayed when the player reaches the goal.

- Gameplay Demo

- Challenges and Learning
  - **Challenges**:
    - Implementing the maze generator.
    - Collision detection for walls and coins.
    - Keeping the game balanced and fun.

- **What I Learned**:
  - Python programming skills.
  - Game design and logic.
  - Procedural generation algorithms.

- THANK YOU