Maze Game

A Pygame Project

A complete guide to building an interactive maze game using Python and Pygame library.

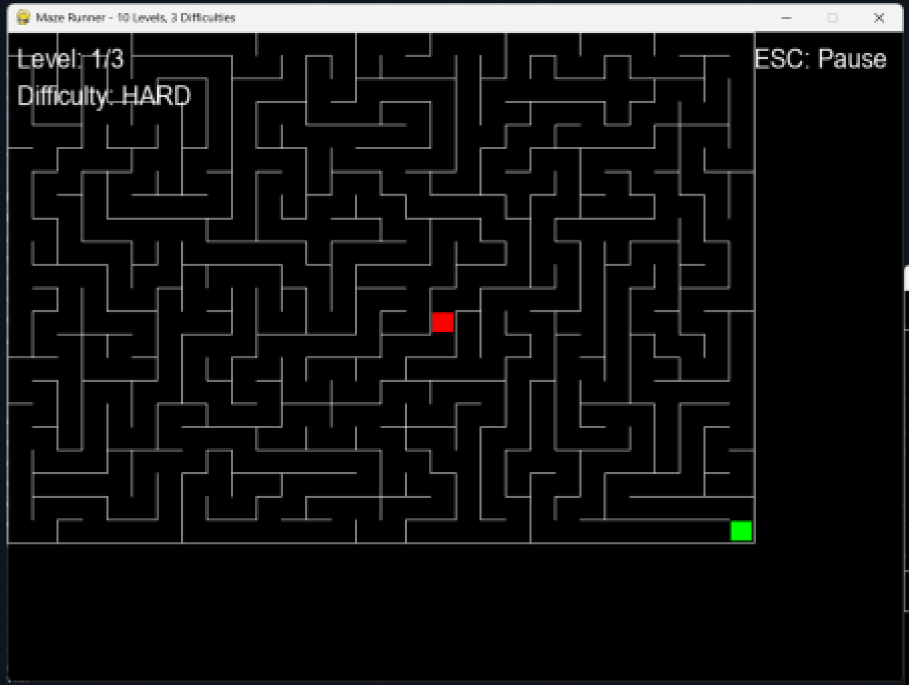
Project Team Members

- Prince Sharma (11024210118)
- Aditya Gairola (11024210095)
- Anirudh Sharma (11024210110)
- Divyansh Choudhary (11024210101)

What is This Project?

Game Overview

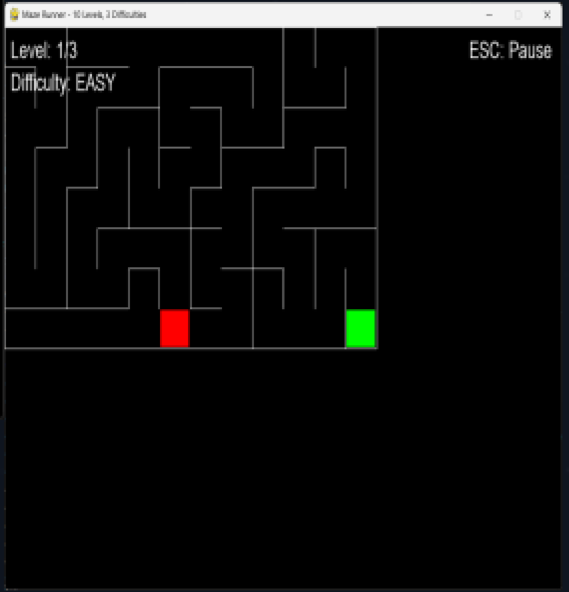
This is a fun maze game where the player moves through different levels. Each level has a maze that gets harder as you progress. The player starts at one corner and must reach the green square at the other end.



Maze Runner - 10 Levels, 3 Difficulties

Level: 1/3
Difficulty: HARD

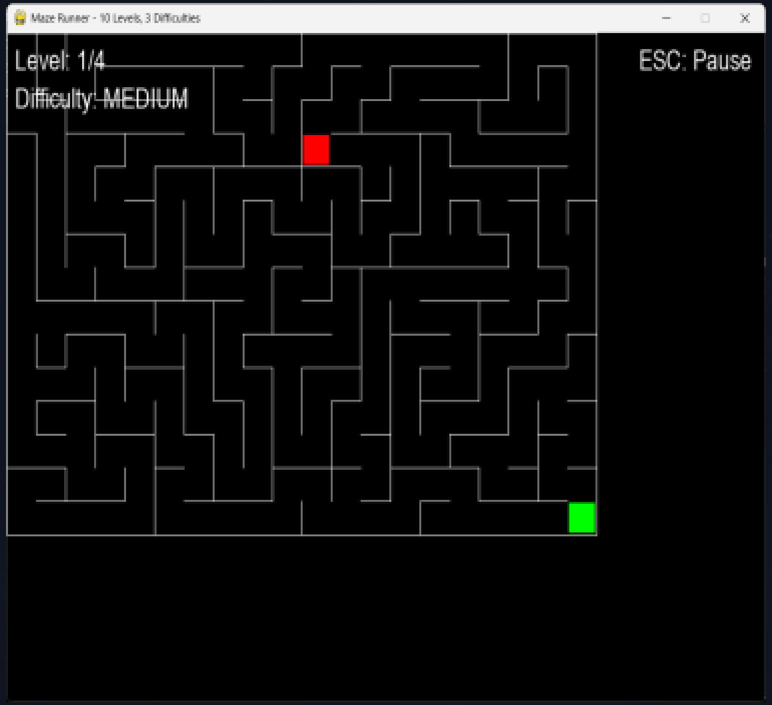
ESC: Pause



Maze Runner - 10 Levels, 3 Difficulties

Level: 1/3
Difficulty: EASY

ESC: Pause



Maze Runner - 10 Levels, 3 Difficulties

Level: 1/4
Difficulty: MEDIUM

ESC: Pause

Understanding the Libraries

Main Libraries Used

Pygame

A Python library for making games. It handles graphics, events, and game timing. It lets us draw shapes, display text, and respond to keyboard input.

Sys

A system library that helps exit the program when the user closes the window or clicks the exit button.

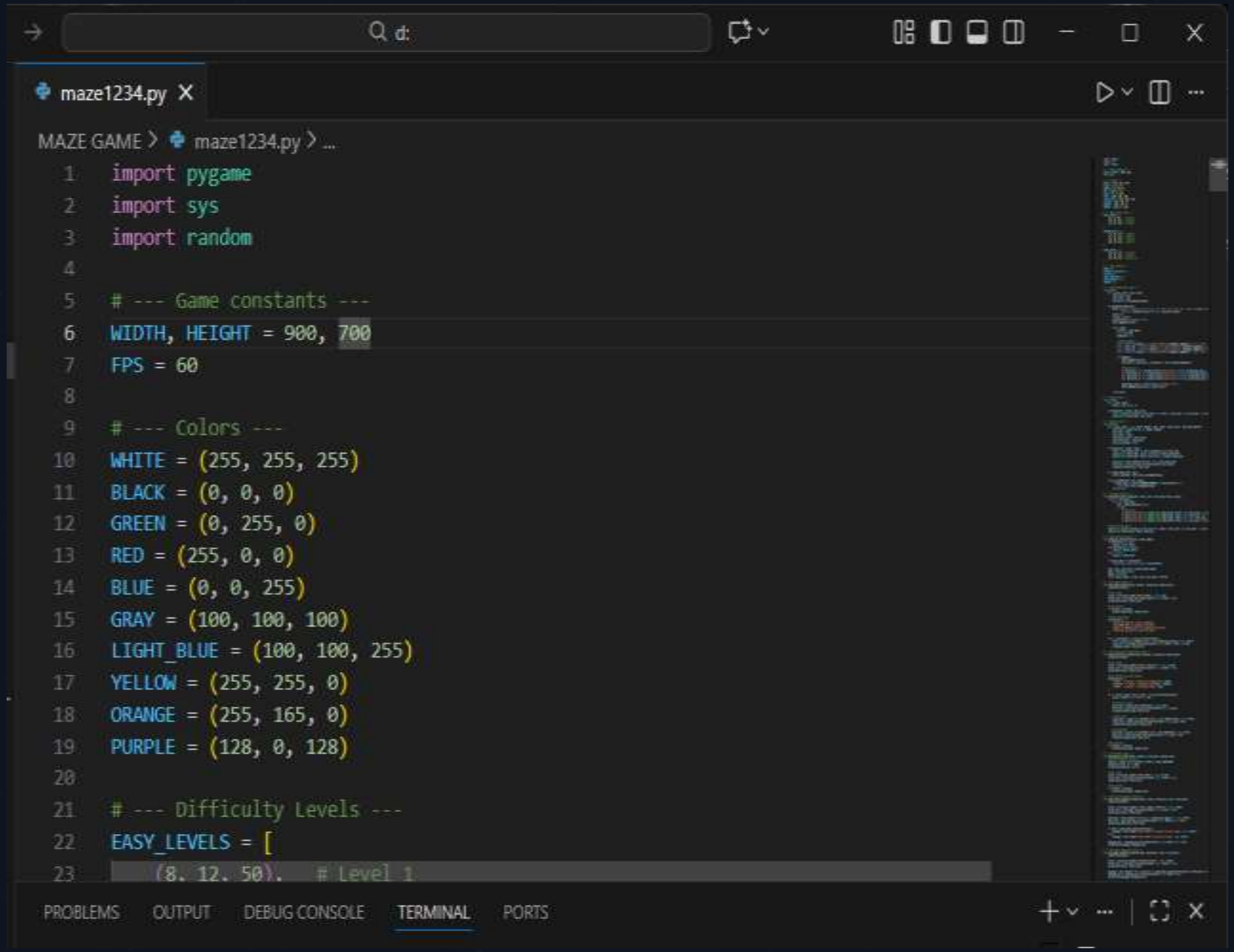
Random

This library picks random choices from a list. We use it to randomly generate the maze paths.

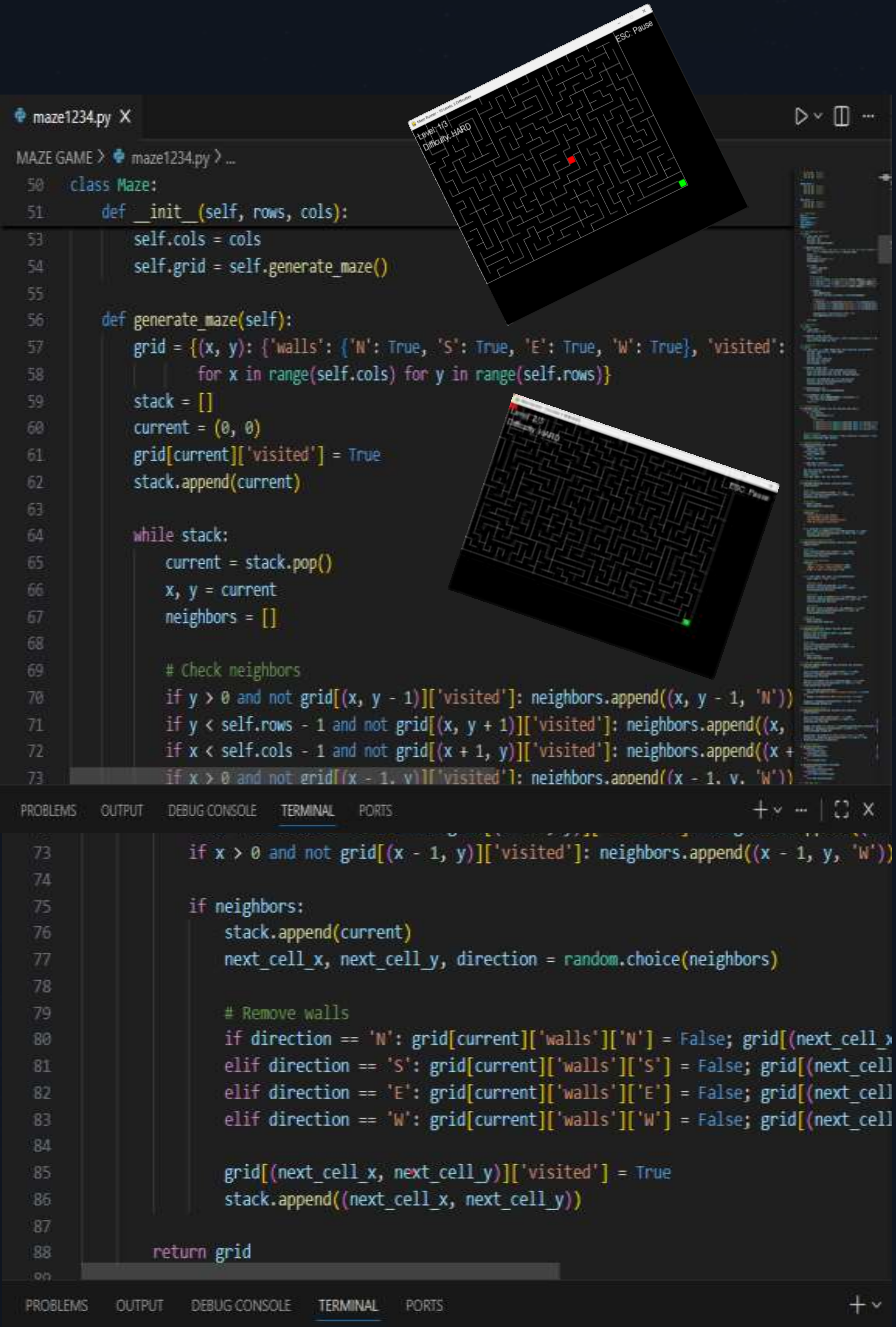
Game Constants and Colors

Game Setup Values

The game window is 900 pixels wide and 700 pixels tall. The game runs at 60 frames per second (FPS), which makes everything smooth and responsive.



```python
import pygame
import sys
import random

# --- Game constants ---
WIDTH, HEIGHT = 900, 700
FPS = 60

# --- Colors ---
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
BLUE = (0, 0, 255)
GRAY = (100, 100, 100)
LIGHT_BLUE = (100, 100, 255)
YELLOW = (255, 255, 0)
ORANGE = (255, 165, 0)
PURPLE = (128, 0, 128)

# --- Difficulty Levels ---
EASY_LEVELS = [
    (8, 12, 50),   # Level 1
```



```python
class Maze:
    def __init__(self, rows, cols):
        self.cols = cols
        self.grid = self.generate_maze()

    def generate_maze(self):
        grid = {(x, y): {'walls': {'N': True, 'S': True, 'E': True, 'W': True}, 'visited':
                for x in range(self.cols) for y in range(self.rows)}
        stack = []
        current = (0, 0)
        grid[current]['visited'] = True
        stack.append(current)

        while stack:
            current = stack.pop()
            x, y = current
            neighbors = []

            # Check neighbors
            if y > 0 and not grid[(x, y - 1)]['visited']: neighbors.append((x, y - 1, 'N'))
            if y < self.rows - 1 and not grid[(x, y + 1)]['visited']: neighbors.append((x,
            if x < self.cols - 1 and not grid[(x + 1, y)]['visited']: neighbors.append((x +
            if x > 0 and not grid[(x - 1, y)]['visited']: neighbors.append((x - 1, y, 'W'))
```

```python
            if x > 0 and not grid[(x - 1, y)]['visited']: neighbors.append((x - 1, y, 'W'))

            if neighbors:
                stack.append(current)
                next_cell_x, next_cell_y, direction = random.choice(neighbors)

                # Remove walls
                if direction == 'N': grid[current]['walls']['N'] = False; grid[(next_cell_x
                elif direction == 'S': grid[current]['walls']['S'] = False; grid[(next_cell
                elif direction == 'E': grid[current]['walls']['E'] = False; grid[(next_cell
                elif direction == 'W': grid[current]['walls']['W'] = False; grid[(next_cell

                grid[(next_cell_x, next_cell_y)]['visited'] = True
                stack.append((next_cell_x, next_cell_y))

        return grid
```

How the Maze is Created

Maze Generation Process

The game creates a new random maze for each level using an algorithm called "Depth-First Search" or DFS. Think of it like carving paths through a maze from top-left to bottom-right.

Start Point

Begin at position (0,0) in the top-left corner of the grid

Create Stack

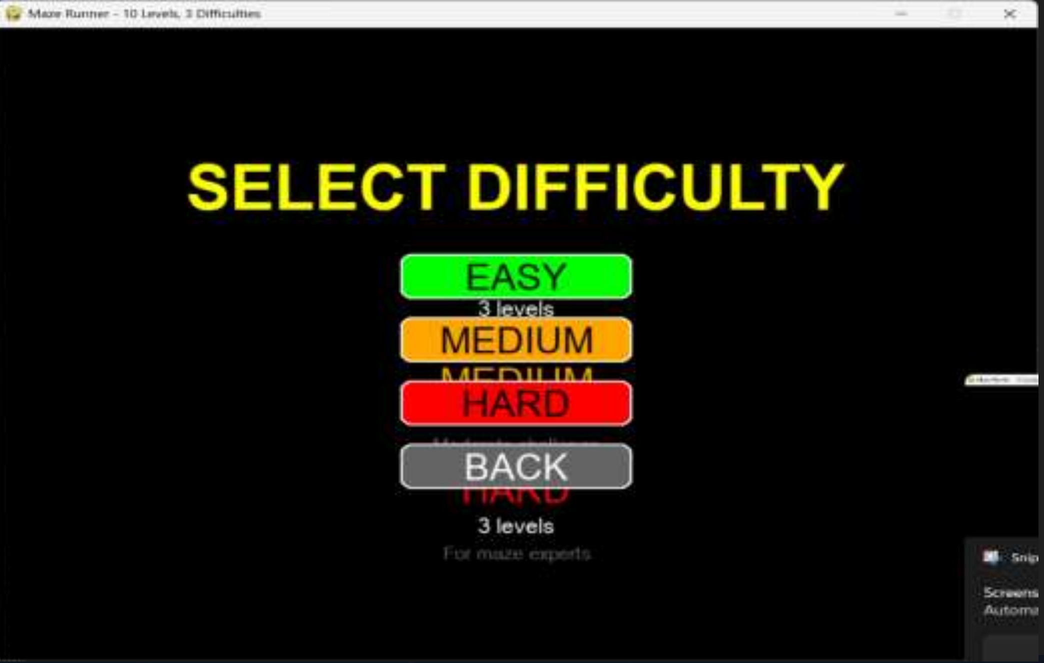
Keep track of all visited cells using a stack (like a pile of cards)

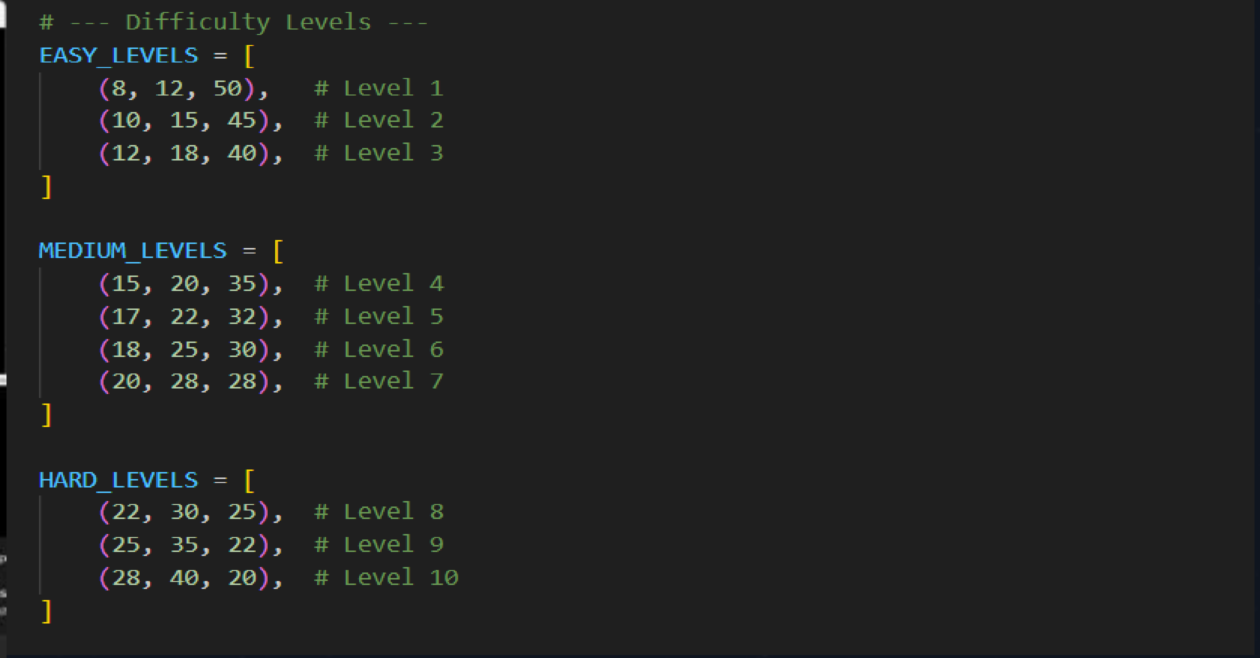
Remove Walls

Randomly pick neighbours and remove walls between cells to create paths

Complete Maze

Continue until all cells are visited and you have one complete maze





```
# --- Difficulty Levels ---
EASY_LEVELS = [
    (8, 12, 50),    # Level 1
    (10, 15, 45),   # Level 2
    (12, 18, 40),   # Level 3
]

MEDIUM_LEVELS = [
    (15, 20, 35),   # Level 4
    (17, 22, 32),   # Level 5
    (18, 25, 30),   # Level 6
    (20, 28, 28),   # Level 7
]

HARD_LEVELS = [
    (22, 30, 25),   # Level 8
    (25, 35, 22),   # Level 9
    (28, 40, 20),   # Level 10
]
```

Understanding Difficulty Levels

Three Difficulty Modes

Easy Mode

3 small mazes. Perfect for beginners to learn the game.

Medium Mode

4 medium-sized mazes. Good challenge for regular players.

Hard Mode

3 very large and complex mazes. Only for expert players!

Each level increases in difficulty by making the maze bigger and more complicated.

Key Game Loops and Functions

Main Loop

The main loop runs 60 times every second. It checks for keyboard input, updates player position, checks if the player reached the end, and draws everything on screen.

1 **Event Loop**

Checks what keys the player pressed and what buttons they clicked

2 **Game Logic**

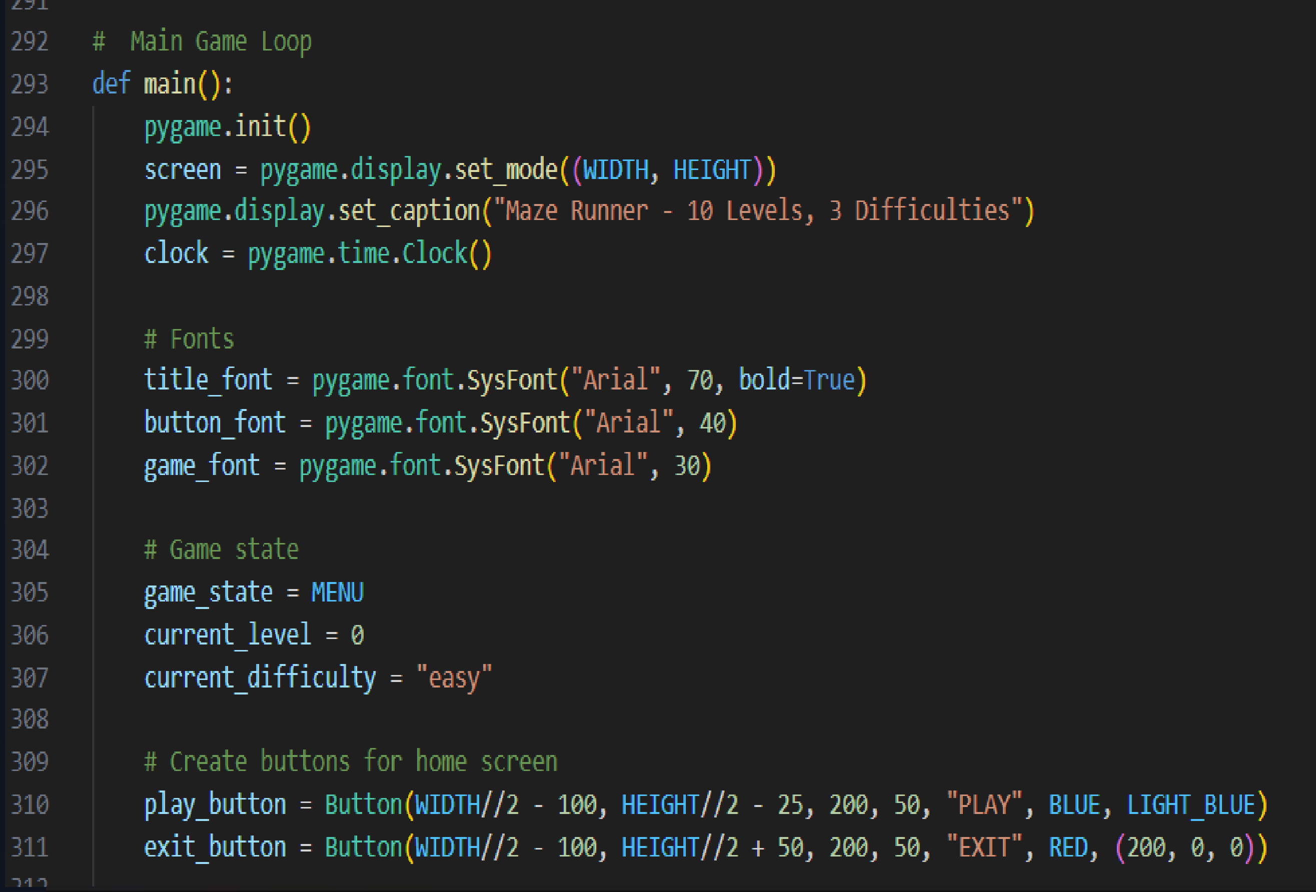
Moves the player and checks if they hit walls or reached the goal

3 **Drawing Loop**

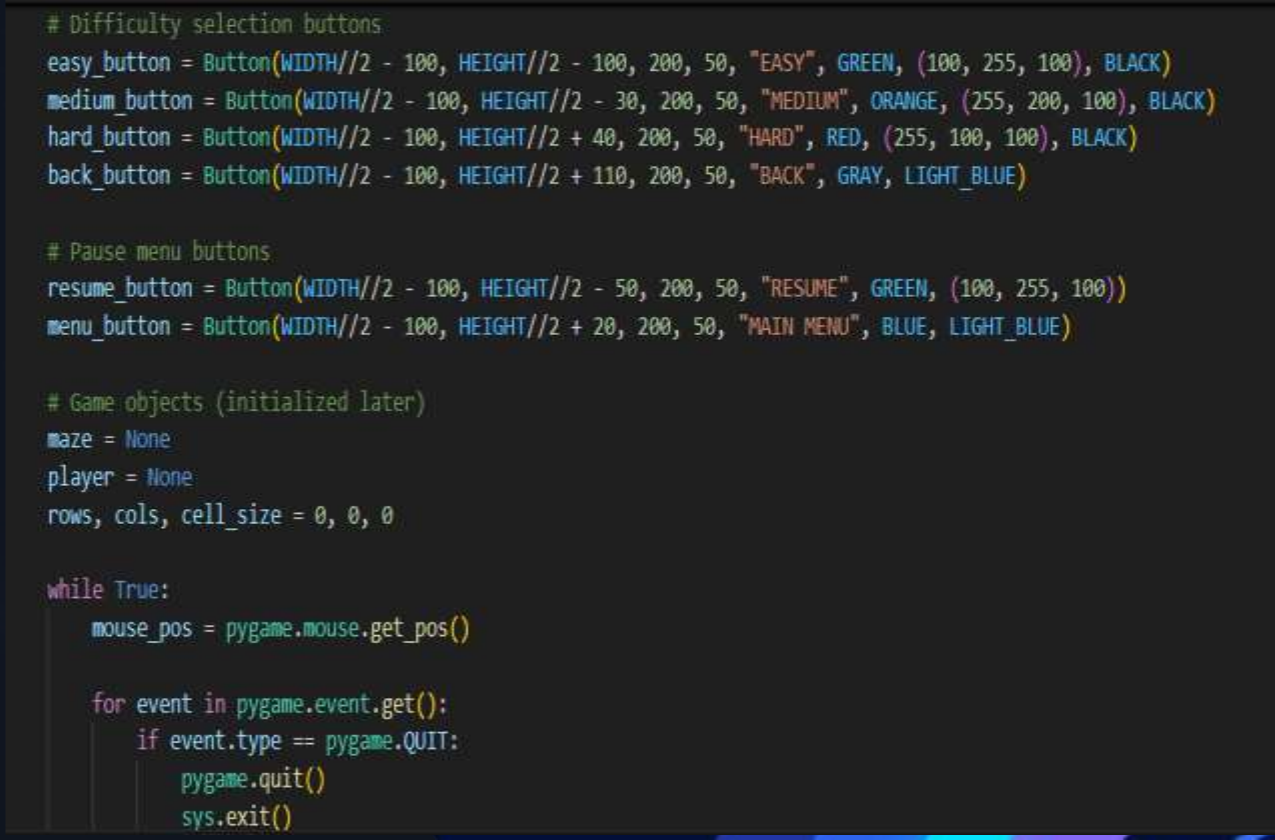
Shows the maze, player position, and game information on screen

4 **Frame Control**

Uses clock.tick(60) to keep the game running at exactly 60 FPS



```
#  Main Game Loop
def main():
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Maze Runner - 10 Levels, 3 Difficulties")
    clock = pygame.time.Clock()

    # Fonts
    title_font = pygame.font.SysFont("Arial", 70, bold=True)
    button_font = pygame.font.SysFont("Arial", 40)
    game_font = pygame.font.SysFont("Arial", 30)

    # Game state
    game_state = MENU
    current_level = 0
    current_difficulty = "easy"

    # Create buttons for home screen
    play_button = Button(WIDTH//2 - 100, HEIGHT//2 - 25, 200, 50, "PLAY", BLUE, LIGHT_BLUE)
    exit_button = Button(WIDTH//2 - 100, HEIGHT//2 + 50, 200, 50, "EXIT", RED, (200, 0, 0))
```

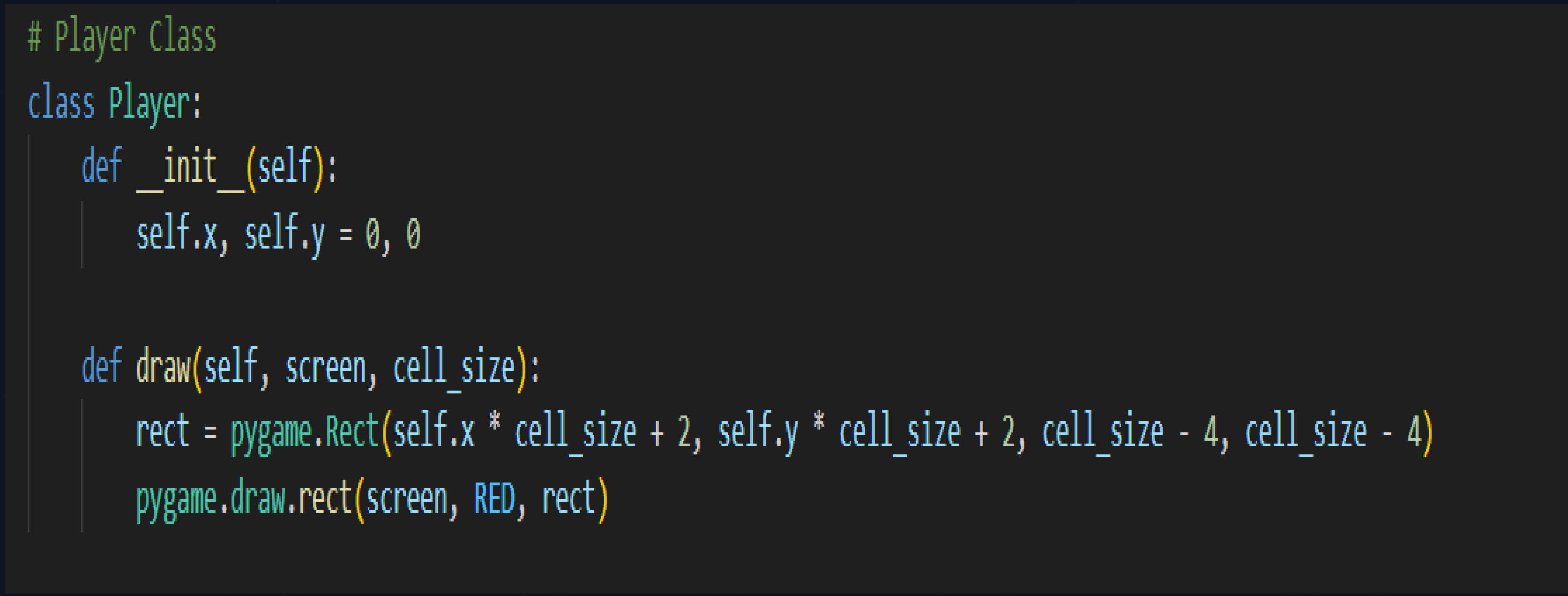


```
# Difficulty selection buttons
easy_button = Button(WIDTH//2 - 100, HEIGHT//2 - 100, 200, 50, "EASY", GREEN, (100, 255, 100), BLACK)
medium_button = Button(WIDTH//2 - 100, HEIGHT//2 - 30, 200, 50, "MEDIUM", ORANGE, (255, 200, 100), BLACK)
hard_button = Button(WIDTH//2 - 100, HEIGHT//2 + 40, 200, 50, "HARD", RED, (255, 100, 100), BLACK)
back_button = Button(WIDTH//2 - 100, HEIGHT//2 + 110, 200, 50, "BACK", GRAY, LIGHT_BLUE)

# Pause menu buttons
resume_button = Button(WIDTH//2 - 100, HEIGHT//2 - 50, 200, 50, "RESUME", GREEN, (100, 255, 100))
menu_button = Button(WIDTH//2 - 100, HEIGHT//2 + 20, 200, 50, "MAIN MENU", BLUE, LIGHT_BLUE)

# Game objects (initialized later)
maze = None
player = None
rows, cols, cell_size = 0, 0, 0

while True:
    mouse_pos = pygame.mouse.get_pos()

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

Classes Used in the Code

Player Class

Stores the player's X and Y position in the maze. It has methods to draw the red square on screen at the correct location.

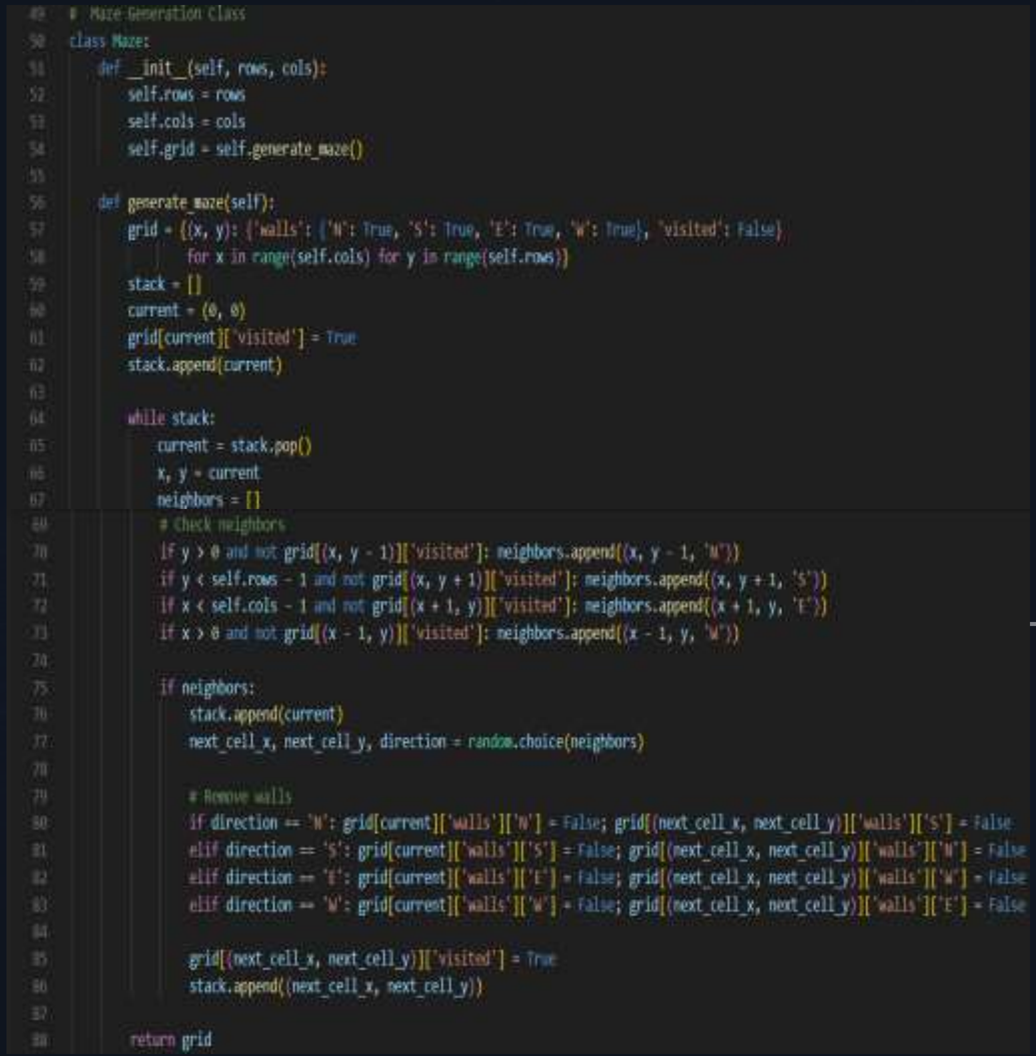


```
# Player Class
class Player:
    def __init__(self):
        self.x, self.y = 0, 0

    def draw(self, screen, cell_size):
        rect = pygame.Rect(self.x * cell_size + 2, self.y * cell_size + 2, cell_size - 4, cell_size - 4)
        pygame.draw.rect(screen, RED, rect)
```

Maze Class

Creates the maze structure using the DFS algorithm.

It stores all walls and empty spaces in a grid that the game uses to check valid moves.



```
# Maze Generation Class
class Maze:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.grid = self.generate_maze()

    def generate_maze(self):
        grid = {(x, y): {'walls': {'N': True, 'S': True, 'E': True, 'W': True}, 'visited': False}
                for x in range(self.cols) for y in range(self.rows)}
        stack = []
        current = (0, 0)
        grid[current]['visited'] = True
        stack.append(current)

        while stack:
            current = stack.pop()
            x, y = current
            neighbors = []
            # Check neighbors
            if y > 0 and not grid[(x, y - 1)]['visited']: neighbors.append((x, y - 1, 'N'))
            if y < self.rows - 1 and not grid[(x, y + 1)]['visited']: neighbors.append((x, y + 1, 'S'))
            if x < self.cols - 1 and not grid[(x + 1, y)]['visited']: neighbors.append((x + 1, y, 'E'))
            if x > 0 and not grid[(x - 1, y)]['visited']: neighbors.append((x - 1, y, 'W'))

            if neighbors:
                stack.append(current)
                next_cell_x, next_cell_y, direction = random.choice(neighbors)

                # Remove walls
                if direction == 'N': grid[current]['walls']['N'] = False; grid[(next_cell_x, next_cell_y)]['walls']['S'] = False
                elif direction == 'S': grid[current]['walls']['S'] = False; grid[(next_cell_x, next_cell_y)]['walls']['N'] = False
                elif direction == 'E': grid[current]['walls']['E'] = False; grid[(next_cell_x, next_cell_y)]['walls']['W'] = False
                elif direction == 'W': grid[current]['walls']['W'] = False; grid[(next_cell_x, next_cell_y)]['walls']['E'] = False

                grid[(next_cell_x, next_cell_y)]['visited'] = True
                stack.append((next_cell_x, next_cell_y))

        return grid
```

Game States and Flow

Six Different Game States

MENU	SELECT
PLAYING	**PAUSED**
COMPLETE	GAME END

The game moves between these states based on player actions. Press ESC to pause, SPACE to continue, and arrow keys to move in the maze.

References

- Pygame Official Documentation: pygame.org
- Python.org - Official Python Documentation
- GeeksforGeeks - Python and Game Development Tutorials
- YouTube - Pygame Tutorial Series

Thank You!

We hope you enjoyed learning about our Maze Game project. This game teaches important programming concepts like object-oriented programming, game loops, algorithms, and user interface design.

Key Takeaways

- Games need many parts working together - graphics, input, logic, and timing
- Classes help organize code into manageable pieces
- Algorithms like DFS can create interesting random content
- Game loops are the heart of all real-time games

Thank you for your attention and support!