PART 3 - MAZE

(LINK)

This project realizes an interactive maze game leveraging core programming concepts such as arrays, loops, conditionals, recursion, and event-driven input, implemented in p5.js. The maze is stored as a two-dimensional array (`maze`), where each cell object maintains a `walls` array of four booleans representing the presence of walls on its top, right, bottom, and left edges. The recursive maze generation uses Depth-First Search (`generateMaze` function) by marking cells as visited (`visited` flag) and systematically removing walls between adjacent cells, ensuring a perfect maze without cycles. Additional complexity is introduced by randomly removing walls between cells (`addExtraOpenings`), which adjusts difficulty dynamically.

Player position is managed with grid coordinates (`cx`, `cy`) and pixel coordinates (`px`, `py`), facilitating smooth movement animation within the `draw` loop. Movement direction is stored as a vector (`movingDir`) with components set in the `keyPressed` event handler, responding to arrow key inputs. Collision checks are performed by indexing the current cell's walls array with a direction index mapped by `getDirectionIndex(dx, dy)` to prevent moving through walls. The movement interpolation uses vector arithmetic to advance the player's pixel position incrementally, providing fluid visual feedback.

A notable feature is the pathfinding component implemented via Breadth-First Search (`findSolutionBFS`), which utilizes a queue and parent arrays to explore reachable cells efficiently and reconstruct the shortest path from start to exit. This path is animated as a "ghost" using the `startGhost` function, where the ghost's position advances along the solution path at controlled intervals (`ghostStepDelay`), with rendering handled by drawing partial paths and ghost ellipses in distinct colors.

Rendering leverages p5.js drawing functions: `line()` to depict walls with precise positioning based on cell coordinates and dimensions (`cellW`, `cellH`), `rect()` for start and exit zones with transparency for user guidance, and `ellipse()` for player and ghost representations. The player's trail is implemented as an array of pixel positions (`trail`) that is updated intelligently in each movement step to avoid redundant points, enhancing the visual feedback on the path taken.

User input is managed in dedicated handlers `keyPressed()` and `keyReleased()`, supporting movement control, level reset, difficulty adjustment through numeric keys, and toggling the ghost visualization with the 'P' key. The use of booleans such as `gameWon` and `ghostActive` enables precise state management, ensuring that game logic and animations respond correctly to user actions.

The game maintains a responsive layout by recalculating maze dimensions and cell sizes in the `setup()` function and responding to browser window changes via `windowResized()`. Persistent storage of best times through `localStorage` is incorporated for performance tracking across sessions, with timers implemented by tracking `millis()` at start and completion events.

CODE

1. Define global variables
2. - cols, rows: maze grid dimensions
3. - cellW, cellH: pixel size of each cell
4. - maze: 2D array of maze cells with walls and visited flags
5. - player, exitPos: player position and maze exit
6. - movingDir: current player movement direction
7. - gameWon: flag indicating if level is won
8. - animSpeed: player movement speed

9. - ghostPath, ghostIndex, ghostActive, ghostPos, ghostLastStepTime, ghostStepDelay: ghost animation variables
10. - showInstructions, fadeInstructions, instructionAlpha: instruction display control variables

11. - level: current difficulty level
12. - timerStart, timerRunning, bestTime: timing variables
13. - minMazeSize, maxMazeSize: maze size constraints

14. Function setup
15. - Create fullscreen canvas
16. - Calculate maze size based on level (constrained by min and max)
17. - Calculate cell pixel width and height
18. - Initialize maze array with all walls intact and cells unvisited
19. - Generate maze recursively starting from cell (0,0)
20. - Add extra random openings to reduce difficulty, fewer openings as level increases
21. - Initialize player at cell (0,0), calculate centered pixel position
22. - Set exit position at bottom-right corner
23. - Initialize player trail with starting pixel position
24. - Set movement direction to null, gameWon and ghostActive flags to false
25. - Show instructions fully opaque
26. - Load best time from localStorage if available
27. - Set player animation speed increasing with level

28. Function draw (main loop)
29. - Clear background with dark color
30. - For each cell in maze
31. -- Draw walls if present
32. - Highlight start cell with translucent green fill
33. - Highlight exit cell with translucent red fill
34. - If player trail length > 1
35. -- Draw blue glowing trail line following player path
36. - If ghost animation active
37. -- Draw ghost path up to current ghost step in red
38. -- If ghost not arrived and step delay elapsed
39. --- Advance ghost animation one step
40. --- Update ghost pixel position
41. -- Draw red ghost circle at ghost pixel position
42. - Draw yellow circle at player pixel position
43. - If player is moving and game not won
44. -- If first player move
45. --- Start fading instructions
46. --- Start timer
47. -- Calculate destination cell based on movement direction
48. -- Check if move is valid (within maze bounds and no wall blocking)

49. -- If move valid
50. --- Calculate destination pixel coordinates
51. --- Move player smoothly toward destination by animSpeed
52. --- If player reached destination
53. ---- Update player grid position
54. ---- Update trail by adding or removing points intelligently
55. -- Else
56. --- Stop player movement
57. - Check if player reached exit cell
58. -- If yes and game not won
59. --- Set gameWon flag true
60. --- Stop timer
61. --- Calculate elapsed time
62. --- Update and save best time if improved
63. - If gameWon
64. -- Display victory message and controls for next level or restart
65. - If instructions visible or fading
66. -- Draw centered semi-transparent instructions box
67. -- Draw instructions text centered inside box
68. -- Decrease opacity gradually if fading
69. - Draw HUD box top-right with centered texts:
70. -- Current level number
71. -- Elapsed time or placeholder if timer off
72. -- Best time or placeholder if none

73. Function drawCell(cell)
74. - Calculate cell pixel position
75. - For each wall (top, right, bottom, left)
76. -- If wall present
77. --- Draw wall line

78. Function getDirectionIndex(dx, dy)
79. - Map movement vector to wall index:
80. -- Up → 0
81. -- Right → 1
82. -- Down → 2
83. -- Left → 3
84. - Return corresponding wall index or -1 if invalid

85. Function generateMaze(cx, cy) [recursive]
86. - Mark current cell visited
87. - Shuffle directions array randomly
88. - For each direction
89. -- Calculate neighbor cell coordinates
90. -- If neighbor inside maze and unvisited
91. --- Remove walls between current cell and neighbor
92. --- Recursively call generateMaze on neighbor

93. Function addExtraOpenings(count)
94. - Repeat count times
95. -- Pick random cell inside maze
96. -- Pick random direction
97. -- Calculate neighbor cell
98. -- If neighbor valid

99. --- Remove walls between the two cells

100. Function findSolutionBFS(sx, sy, ex, ey)
101. - Initialize queue, visited and parent arrays
102. - Enqueue start cell and mark visited
103. - While queue not empty
104. -- Dequeue current cell
105. -- If current is exit cell, break loop
106. -- For each neighbor without wall blocking
107. --- If neighbor inside maze and not visited
108. ---- Mark visited and record parent
109. ---- Enqueue neighbor
110. - Reconstruct path from exit to start using parents
111. - Reverse path for start-to-exit order
112. - Return path array

113. Function startGhost()
114. - Compute shortest path using findSolutionBFS
115. - Reset ghost animation index and activate animation
116. - Set ghost position to start of path
117. - Record current time as last step time

118. Function ghostArrived()
119. - Return true if ghost animation reached end of path

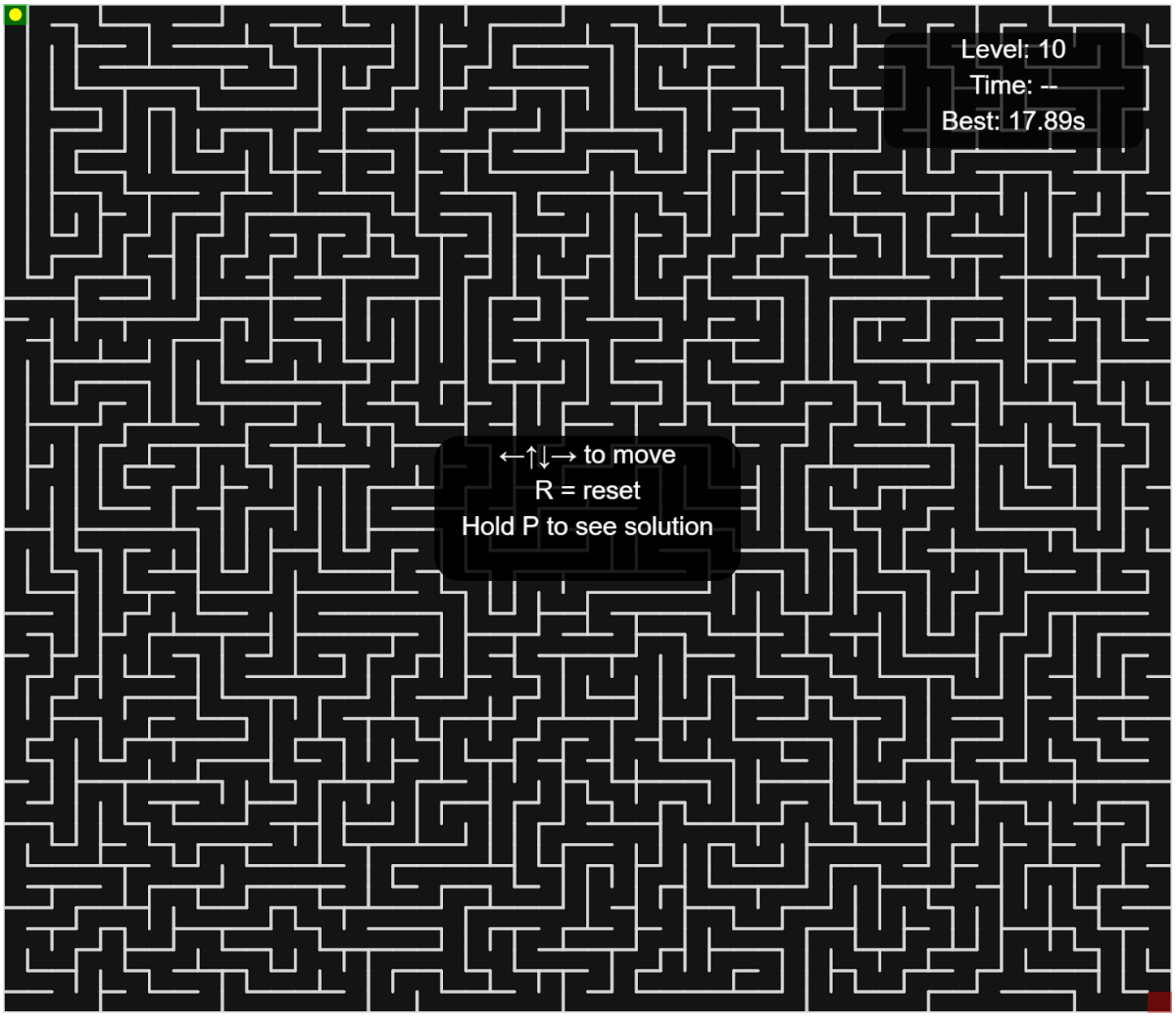
120. Function keyPressed()
121. - If gameWon
122. -- If ENTER pressed
123. --- Increment level and reset game
124. -- If R pressed
125. --- Reset current level
126. - If number key 0-9 pressed
127. -- Set level accordingly (0 means 10) and reset game
128. - If arrow keys pressed
129. -- Set movingDir accordingly
130. - If R pressed
131. -- Reset game
132. - If P pressed and ghost not active and game not won
133. -- Start ghost animation

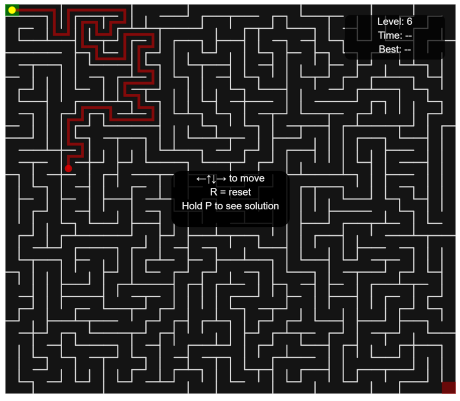
134. Function keyReleased()
135. - Stop player movement
136. - If P released
137. -- Stop ghost animation and reset ghost index

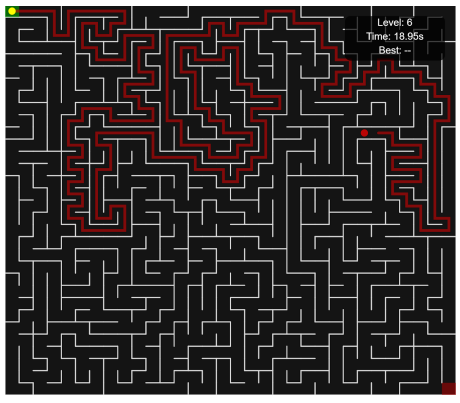
138. Function windowResized()
139. - Reset game by calling setup

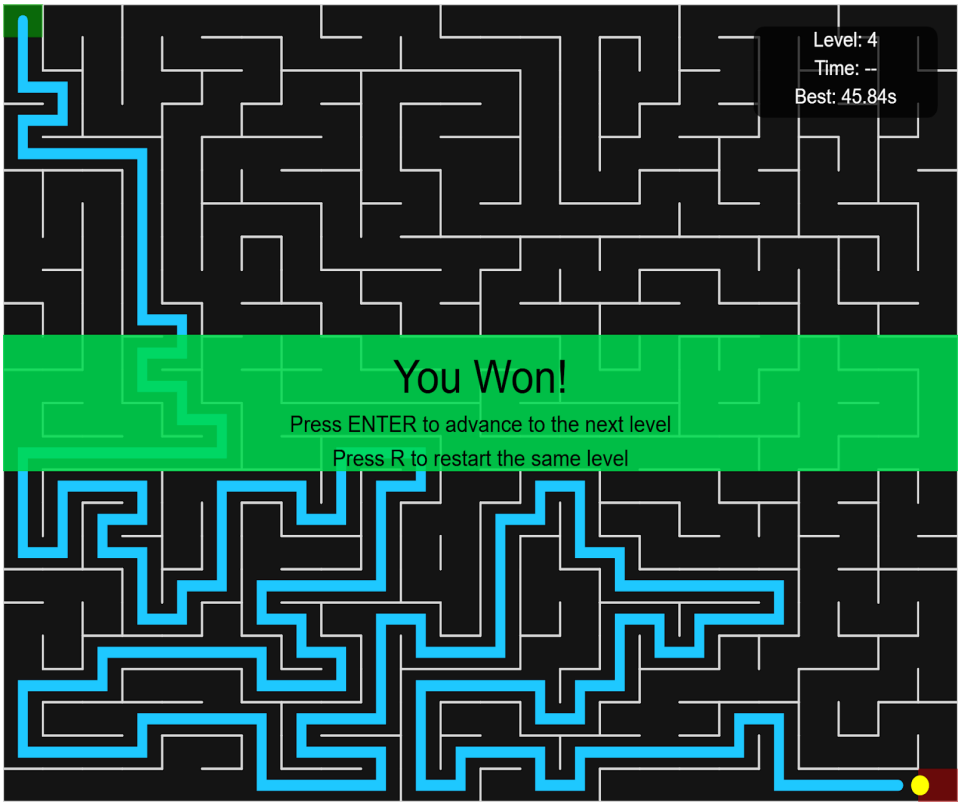
CODE

```
// Maze game with player movement, animated solution (ghost), levels, timer, and best time tracking

// Global variables
let cols, rows;             // Number of columns and rows in the maze grid
let cellW, cellH;           // Width and height of each maze cell in pixels
let maze = [];              // 2D array holding maze cells and their walls info
let player, exitPos;        // Player's position (grid and pixel), and exit cell position
let moveDir = null;         // Player's current moving direction as {dx, dy}, or null if stopped
let gameWon = false;        // Flag to track if the current level is won
let moveSpeed = 5;          // Speed of the player's movement in pixels per frame, changes with level

// Variables for the ghost animation (shows the solution path)
let ghostPath = [];         // Array of cells representing the shortest path solution
let ghostIndex = 0;         // Current step index in the ghost animation path
let ghostActive = false;    // Flag indicating whether the ghost animation is active
let ghostPos = null;        // Current pixel position of the ghost circle
let ghostLastStepTime = 0;  // Timestamp (ms) when the ghost last moved one step
let ghostStepDelay = 60;    // Delay in milliseconds between ghost animation steps

// Instruction display variables
let showInstructions = true; // Show instructions text box at game start
let fadeInstructions = false; // Start fading instructions when player moves
let instructionAlpha = 255;  // Opacity level (0-255) of instructions text box

// Level and timing variables
let level = 1;              // Current game level number
let timerStart = 0;         // Timestamp (ms) when timer started (player's first move)
let timerRunning = false;   // Flag to track if timer is currently running
let bestTime = null;        // Best recorded time for the current level, from localStorage

// Maze size limits for dynamic difficulty
let minMazeSize = 12;       // Minimum maze dimension (cols and rows)
let maxMazeSize = 50;       // Maximum maze dimension

function setup() {
  // Create canvas that fills the browser window
  createCanvas(windowWidth, windowHeight);

  // Calculate maze size based on current level:
  // Maze grows larger by 4 cells per level but capped by min and max limits
  let mazeSize = constrain(minMazeSize + (level - 1) * 4, minMazeSize, maxMazeSize);
  cols = mazeSize;
  rows = mazeSize;

  // Calculate pixel dimensions for each maze cell
  cellW = width / cols;
  cellH = height / rows;

  // Initialize the maze array: 2D grid with each cell having all walls intact and marked unvisited
  maze = [];
  for (let y = 0; y < rows; y++) {
    maze[y] = [];
    for (let x = 0; x < cols; x++) {
      maze[y][x] = {
        x: x,           // Cell grid X coordinate
        y: y,           // Cell grid Y coordinate
        walls: [true, true, true, true], // Walls in order: top, right, bottom, left
        visited: false // Flag used during maze generation
      };
    }
  }

  // Generate maze paths recursively, starting from top-left corner (0,0)
  generateMaze(0, 0);

  // Add extra random openings (removing walls) to reduce difficulty,
  // number of openings increases as level increases for more challenge
  let extraOpenings = max(3, 15 - level * 1.5 + 2);
  addRandomOpenings(extraOpenings);

  // Initialize player at starting cell (0,0) with no pixel position yet
  player = {
    cx: 0,      // Player's grid X coordinate
    cy: 0,      // Player's grid Y coordinate
    px: 0,      // Player's pixel X coordinate (will be set to center of cell)
    py: 0,      // Player's pixel Y coordinate
    trail: []   // Array of pixel positions to draw the player's path trail
  };

  // Exit cell is fixed at bottom-right corner of maze
  exitPos = createVector(cols - 1, rows - 1);

  // Convert player's grid position to pixel coordinates centered in cell
  player.px = player.cx * cellW + cellW / 2;
  player.py = player.cy * cellH + cellH / 2;

  // Initialize player's trail with starting pixel position
  player.trail = [{ px: player.px, py: player.py }];

  moveDir = null;     // Player starts stationary
  frameRate(60);      // Set target frame rate
  gameWon = false;    // Reset game won flag
  ghostActive = false; // Reset ghost animation

  showInstructions = true;  // Show instructions at the start
  fadeInstructions = false; // Do not fade yet
  instructionAlpha = 255;   // Fully opaque instructions

  timerRunning = false;     // Timer not running at start

  // Retrieve best time for this level from localStorage, if it exists
  let stored = localStorage.getItem("bestTime_level_" + level);
  bestTime = stored ? parseFloat(stored) : null;

  // Adjust player animation speed: faster as level increases, capped at 12 px/frame
  moveSpeed = constrain(5 + (level - 1) * 0.7, 5, 12);
}

function draw() {
  background(30);  // Clear background with dark color

  // Draw the maze cells with their walls
  stroke(220);     // Light gray stroke for walls
  strokeWeight(3);
  for (let y = 0; y < rows; y++) {
    for (let x = 0; x < cols; x++) {
      drawCell(maze[y][x]); // Draw the walls for each cell
    }
  }

  // Highlight start cell with translucent green fill
  noStroke();
  fill(0, 200, 0, 120);
  rect(0, 0, cellW, cellH);

  // Highlight exit cell with translucent red fill
  fill(200, 0, 0, 120);
  rect(exitPos.x * cellW, exitPos.y * cellH, cellW, cellH);

  // Draw player's trail as a glowing blue line (only if trail has multiple points)
  if (player.trail && player.trail.length > 1) {
    stroke(50, 200, 255);
    strokeWeight(cellW * 0.25);
    noFill();
    beginShape();
    for (let p of player.trail) {
      vertex(p.px, p.py); // Add each trail point vertex
    }
    endShape();
  }

  // Animate ghost solution path if active (red trail and ghost dot)
  if (ghostActive && ghostPath.length > 1) {
    if (ghostIndex > 0 && ghostPos) {
      stroke(255, 0, 0, 150); // Red glowing stroke for ghost path
      strokeWeight(cellW * 0.2);
      noFill();
      beginShape();
      // Draw ghost path up to current ghost step
      for (let i = 0; i <= ghostIndex; i++) {
        vertex(ghostPath[i].x * cellW + cellW / 2, ghostPath[i].y * cellH + cellH / 2);
      }
      endShape();
    }

    // Step forward in ghost animation if enough time has passed
    if (!gameWon && millis() - ghostLastStepTime > ghostStepDelay) {
      ghostIndex++;
      ghostLastStepTime = millis();
      if (ghostIndex < ghostPath.length) {
        ghostPos = { x: ghostPath[ghostIndex].x, y: ghostPath[ghostIndex].y };
      }
    }

    // Draw the ghost circle at its current position (red, semi-transparent)
    fill(255, 0, 0, 180);
    noStroke();
    ellipse(
      ghostPos.x * cellW + cellW / 2,
      ghostPos.y * cellH + cellH / 2,
      min(cellW, cellH) * 0.6
    );
  }

  // Draw player as a yellow circle at current pixel position
  fill(255, 255, 0);
  noStroke();
  ellipse(player.px, player.py, min(cellW, cellH) * 0.4);

  // Handle smooth player movement if game not won and moveDir is set
  if (!gameWon && moveDir) {
    // On player's first movement, hide instructions and start timer
    if (showInstructions) {
      showInstructions = false;  // Hide instructions
      fadeInstructions = true;   // Start fade-out animation
      timerStart = millis();     // Start the timer
      timerRunning = true;       // Flag timer as running
    }

    // Calculate the grid coordinates of the destination cell the player is moving toward
    let destCx = player.cx + moveDir.dx;
    let destCy = player.cy + moveDir.dy;

    // Get current cell to check for walls
    let currCell = maze[player.cy][player.cx];

    // Get wall index corresponding to movement direction (0:top, 1:right, 2:bottom, 3:left)
    let dirIdx = getDirectionIndex(moveDir.dx, moveDir.dy);

    // Check if move is within bounds and no wall blocks the movement
    if (
      destCx >= 0 &&
      destCx < cols &&
      destCy >= 0 &&
      destCy < rows &&
      currCell.walls[dirIdx] == false
    ) {
      // Convert destination cell grid position to pixel position (center)
      let destPx = destCx * cellW + cellW / 2;
      let destPy = destCy * cellH + cellH / 2;

      // Compute distance vector and magnitude from current pixel position to destination
      let dx = destPx - player.px;
      let dy = destPy - player.py;
      let dist = sqrt(dx * dx + dy * dy);

      // Move player smoothly towards destination by an increment of moveSpeed pixels per frame
      if (dist > moveSpeed) {
        player.px += (dx / dist) * moveSpeed;
        player.py += (dy / dist) * moveSpeed;
      } else {
        // Snap player precisely to destination cell center pixel coordinates
        player.px = destPx;
        player.py = destPy;

        // Update player trail by adding or removing points intelligently
        let last = player.trail[player.trail.length - 1];
        if (!last || last.px != destPx || last.py != destPy) {
          // If moving backward on the path, remove the last trail point instead of adding new
          if (
            player.trail.length > 1 &&
            dist(destPx, destPy, player.trail[player.trail.length - 2].px, player.trail[player.trail.length - 2].py) < 0.5
          ) {
            player.trail.pop();
          } else {
            // Otherwise add new point to trail to follow player movement
            player.trail.push({ px: destPx, py: destPy });
          }
        }

        // Update player's grid cell coordinates to new position
        player.cx = destCx;
        player.cy = destCy;
      }
    } else {
      // Movement blocked by wall or outside maze, stop player movement
      moveDir = null;
    }
  }

  // Check if player has reached exit cell (based on pixel position)
  let exitX = exitPos.x * cellW;
  let exitY = exitPos.y * cellH;
  if (
    player.px > exitX &&
    player.px < exitX + cellW &&
    player.py > exitY &&
    player.py < exitY + cellH
  ) {
    gameWon = true;  // Mark game as won
    timerRunning = false; // Stop timer

    // Calculate elapsed time in seconds since timer started
    let currentTime = (millis() - timerStart) / 1000;

    // Update best time if current is better or no best time recorded yet
    if (bestTime == null || currentTime < bestTime) {
      bestTime = currentTime;
      // Save best time in localStorage for this level
      localStorage.setItem("bestTime_level_" + level, bestTime);
    }
  }

  // Display victory message with options when game is won
  if (gameWon) {
    fill(0, 150, 0, 230);  // Semi-transparent green background bar
    rect(0, height / 2 - 60, width, 120);

    fill(0);               // Black text color
    textAlign(CENTER, CENTER);
    textSize(40);
    text("You Won!", width / 2, height / 2 - 20);

    textSize(20);
    text("Press ENTER to advance to the next level", width / 2, height / 2 + 20);
    text("Press R to restart the same level", width / 2, height / 2 + 50);
  }

  // Draw instructions text box with fade-out animation at the start of the game
  if (showInstructions || (fadeInstructions && instructionAlpha > 0)) {
    let instructions = "Use arrow keys or WASD to move\nR to restart\nP to see solution";
    textSize(16);
    let lines = instructions.split("\n");
    textFont('sans-serif');

    // Calculate width and height of text box based on text size and padding
    let longest = lines.reduce((a, b) => a.length > b.length ? a : b);
    let tw = textWidth(longest) + 30; // add padding
    let th = lines.length * 20 + 30;  // add padding

    // Center the text box on the canvas
    let tx = (width - tw) / 2;
    let ty = (height - th) / 2;

    // Fade instructions by decreasing alpha gradually
    if (fadeInstructions) {
      instructionAlpha -= 5;
      if (instructionAlpha <= 0) {
        instructionAlpha = 0;
        fadeInstructions = false; // Stop fading after fully invisible
      }
    }

    // Draw background rectangle with current alpha
    fill(0, 220 * (instructionAlpha / 255));
    rect(tx, ty, tw, th, 10);

    // Draw each instruction line centered horizontally inside the box
    fill(255, instructionAlpha);
    noStroke();
    textAlign(CENTER, CENTER);
    for (let i = 0; i < lines.length; i++) {
      text(lines[i], width / 2, ty + 30 + i * 20);
    }
  }

  // Draw Heads-Up Display (HUD) box at top-right for level, time, and best time
  fill(0, 150);
  rect(width - 160, 20, 140, 80, 10);

  fill(255);
  textAlign(CENTER, CENTER);
  textSize(18);

  let hudX = width - 90;
  let hudY = 40;
  let lineHeight = 25;

  // Display current level number
  text("Level: " + level, hudX, hudY + lineHeight / 2);

  // Display elapsed time or dashes if timer not running
  if (timerRunning) {
    let elapsed = (millis() - timerStart) / 1000;
    text("Time: " + nf(elapsed, 1, 1) + "s", hudX, hudY + lineHeight * 1.5);
  } else {
    text("Time: --", hudX, hudY + lineHeight * 1.5);
  }

  // Display best time or dashes if no best time recorded yet
  if (bestTime != null) {
    text("Best: " + nf(bestTime, 1, 1) + "s", hudX, hudY + lineHeight * 2.5);
  } else {
    text("Best: --", hudX, hudY + lineHeight * 2.5);
  }
}

// Draw maze cell walls as lines
function drawCell(cell) {
  stroke(220);
  strokeWeight(3);

  // Calculate top-left pixel position of the cell
  let x = cell.x * cellW;
  let y = cell.y * cellH;

  // Draw top wall if it exists
  if (cell.walls[0]) line(x, y, x + cellW, y);

  // Draw right wall if it exists
  if (cell.walls[1]) line(x + cellW, y, x + cellW, y + cellH);

  // Draw bottom wall if it exists
  if (cell.walls[2]) line(x + cellW, y + cellH, x, y + cellH);

  // Draw left wall if it exists
  if (cell.walls[3]) line(x, y + cellH, x, y);
}

// Convert movement delta dx, dy to wall index (0 to 3)
function getDirectionIndex(dx, dy) {
  if (dx == 0 && dy == -1) return 0; // moving up corresponds to top wall
  if (dx == 1 && dy == 0) return 1;  // moving right corresponds to right wall
  if (dx == 0 && dy == 1) return 2;  // moving down corresponds to bottom wall
  if (dx == -1 && dy == 0) return 3; // moving left corresponds to left wall
  return -1; // invalid movement
}

// Recursive depth-first search algorithm to generate maze paths
function generateMaze(cx, cy) {
  let cell = maze[cy][cx];
  cell.visited = true; // mark current cell as visited

  // Randomize directions to create a unique maze each run
  let dirs = shuffle([0, 1, 2, 3]);

  for (let i = 0; i < dirs.length; i++) {
    let d = dirs[i];

    // Calculate neighbor cell coordinates based on direction
    let nx = cx;
    let ny = cy;
    if (d == 0) ny -= 1; // up
    if (d == 1) nx += 1; // right
    if (d == 2) ny += 1; // down
    if (d == 3) nx -= 1; // left

    // Continue if neighbor is within bounds and not visited
    if (nx >= 0 && nx < cols && ny >= 0 && ny < rows && !maze[ny][nx].visited) {
      // Remove walls between current cell and neighbor to create a path
      cell.walls[d] = false;
      maze[ny][nx].walls[(d + 2) % 4] = false;

      // Recursively generate maze starting from neighbor cell
      generateMaze(nx, ny);
    }
  }
}

// Add extra random openings in the maze to reduce difficulty
function addRandomOpenings(count) {
  for (let i = 0; i < count; i++) {
    let x = int(random(1, cols - 1)); // random cell inside maze, avoiding edges
    let y = int(random(1, rows - 1));
    let dir = int(random(4));         // random direction to remove wall

    let nx = x, ny = y;

    if (dir == 0) ny -= 1;
    if (dir == 1) nx += 1;
    if (dir == 2) ny += 1;
    if (dir == 3) nx -= 1;

    // Remove wall if neighbor cell exists
    if (nx > 0 && nx < cols - 1 && ny > 0 && ny < rows - 1) {
      maze[y][x].walls[dir] = false;
      maze[ny][nx].walls[(dir + 2) % 4] = false;
    }
  }
}

// Breadth-first search to find shortest path from start to exit
function findSolutionBFS(sx, sy, ex, ey) {
  let queue = [];
  let visited = Array(rows).fill().map(() => Array(cols).fill(false));
  let parent = Array(rows).fill().map(() => Array(cols).fill(null));

  queue.push({ x: sx, y: sy });  // Start from starting cell
  visited[sy][sx] = true;

  // Movement directions: up, right, down, left
  let dirs = [
    { dx: 0, dy: -1 },
    { dx: 1, dy: 0 },
    { dx: 0, dy: 1 },
    { dx: -1, dy: 0 }
  ];

  while (queue.length > 0) {
    let curr = queue.shift();  // Dequeue front of queue

    if (curr.x == ex && curr.y == ey) break; // Exit found

    let cell = maze[curr.y][curr.x];

    // Explore neighbors if no wall blocks path
    for (let d = 0; d < 4; d++) {
      if (!cell.walls[d]) {
        let nx = curr.x + dirs[d].dx;
        let ny = curr.y + dirs[d].dy;

        // If neighbor valid and not visited, enqueue it
        if (
          nx >= 0 && nx < cols &&
          ny >= 0 && ny < rows &&
          !visited[ny][nx]
        ) {
          visited[ny][nx] = true;
          parent[ny][nx] = { x: curr.x, y: curr.y };
          queue.push({ x: nx, y: ny });
        }
      }
    }
  }

  // Reconstruct shortest path from exit to start using parent links
  let path = [];
  let px = ex, py = ey;
  while (px != -1 && py != -1) {
    path.push({ x: px, y: py });
    let par = parent[py][px];
    if (!par) break; // No path found
    px = par.x;
    py = par.y;
  }
  path.reverse();  // Reverse to start->exit order

  return path;
}

// Start the ghost animation showing the solution path
function startGhost() {
  ghostPath = findSolutionBFS(0, 0, exitPos.x, exitPos.y); // Compute shortest path
  ghostIndex = 0;                 // Reset ghost animation step index
  ghostActive = true;             // Activate ghost animation
  ghostPos = { x: ghostPath[0].x, y: ghostPath[0].y }; // Start ghost at maze start
  ghostLastStepTime = millis();   // Initialize last step timestamp
}

// Returns true if ghost animation reached the end of the path
function ghostArrived() {
  return ghostIndex == ghostPath.length - 1;
}

// Keyboard input handler
function keyPressed() {
  // If game won, check for enter or R keys to advance or restart
  if (gameWon) {
    if (keyCode == ENTER || keyCode == RETURN) {
      level++;  // Next level
      setup();  // Restart game state
      return;
    }
    if (key == 'r' || key == 'R') {
      setup();  // Restart current level
      return;
    }
  }

  // Level selection via number keys 1-9 and 0 for 10
  if (key >= '0' && key <= '9') {
    let newLevel = key == '0' ? 10 : parseInt(key);
    if (newLevel >= level) {
      level = newLevel;
      setup();
      return;
    }
  }

  // Arrow keys control player movement direction
  if (keyCode === RIGHT_ARROW) moveDir = { dx: 1, dy: 0 };
  else if (keyCode === LEFT_ARROW) moveDir = { dx: -1, dy: 0 };
  else if (keyCode === DOWN_ARROW) moveDir = { dx: 0, dy: 1 };
  else if (keyCode === UP_ARROW) moveDir = { dx: 0, dy: -1 };

  // Reset game on R
  if (key === 'r' || key === 'R') {
    setup();
  }

  // Hold P to activate ghost animation showing the solution path
  if (key === 'p' || key === 'P') startGhost();
}

// Keyboard release handler: stop player movement and ghost animation on P release
function keyReleased() {
  moveDir = null;
  if (key === 'p' || key === 'P') {
    ghostActive = false;
    ghostIndex = 0;
  }
}

// Recreate maze on window resize
function windowResized() {
  setup();
}
```


Level: 10
Time: --
Best: 17.89s
←↑↓→ to move
R = reset
Hold P to see solution


Level: 6
Time: --
Best: --
←↑↓→ to move
R = reset
Hold P to see solution


Level: 6
Time: 18.95s
Best: --


Level: 4
Time: --
Best: 45.84s
You Won!
Press ENTER to advance to the next level
Press R to restart the same level