

Process & Decision Documentation

Side Quests and A4 (Individual Work)

One significant change I made was to the level 3 word block error code, the original code that GenAI created was a code that blocked off one entire column. This made it so the user was not able to pass through to reach the end goal, this created a problem as users were not able to get off this page as they had no way to end the game. I then went back to edit the code so that the error codes would appear in a better arrangement so that the game was complete and had a start to finish. It definitely made the game less frustrating as it wasn't stuck on one page but it was a lot smoother to transition from level 3 back to 1.

GenAI Documentation

Date Used: February 8, 2026

Tool Disclosure: Gemini 3

Purpose of Use: I used GenAI to edit the original code to add/replace to get the desired arrangement of word blocks. I also used it to write the comments for the new code.

Summary of Interaction: The tool contributed to the writing and editing of the basic code and also writing comments for the new updated code.

Human Decision Point(s): From the original code, the text was off, and the word blocks were in weird places. I edited the visuals in terms of text and also rearranged the code for a better arrangement.

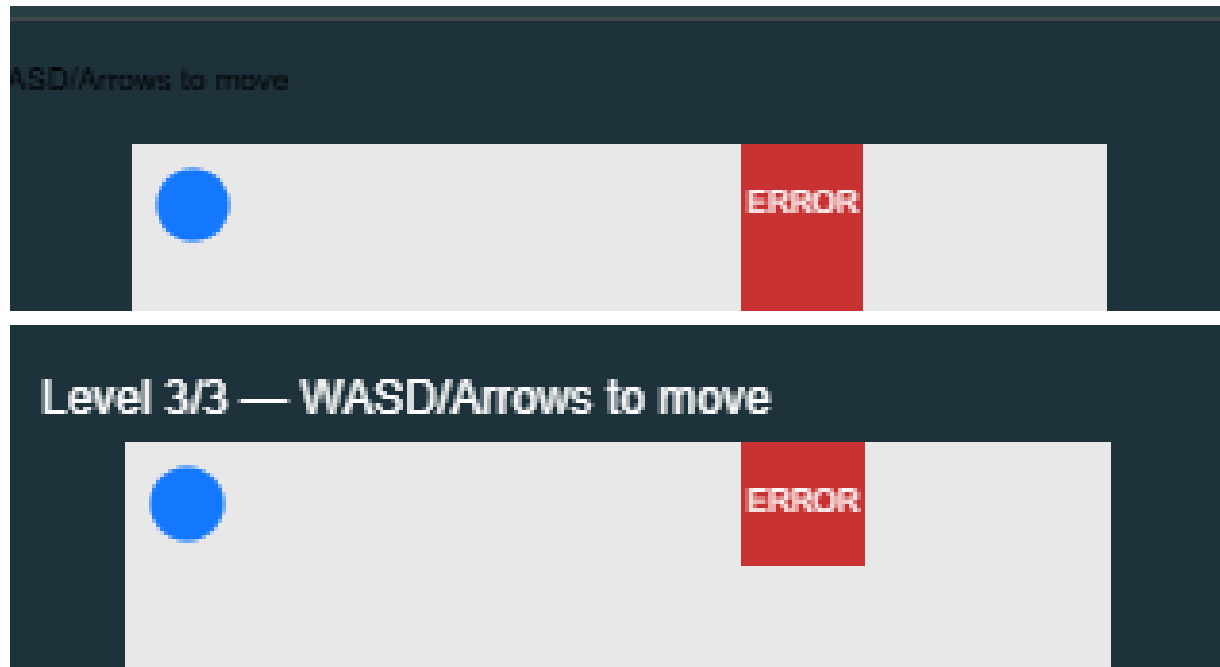
Integrity & Verification Note: For GenAI accuracy, I tested the code out using the live server several times to see what the arrangement of the maze looked like and repeated this process several times. I also did it based on whether it fulfilled the assignment and also personal visual bias.

Scope of GenAI Use: AI did not contribute to the reiterating of the newest code for the level 3 word blocks, I edited it after it sent the basic code. The tool also did not edit the README file or edit the way the text appeared.

Limitations or Misfires: AI did not account for text and visual problems. It also did not consider if the level 3 maze can be successful in the user being able to reach the end.

Summary of Process (Human + Tool)

I did several iterations, testing, and revising of the code. In terms of the text, I played around with the text size, positioning, and colour. I also did several testing processes using the live server to the level 3 maze as the original code gave me a blocked off column that didn't allow for the user to even pass that column making it impossible for the user to pass. I tested the original code, played around with the numbers, rows, and columns and kept revising based on what it looked like on the live server. Sometimes I would end up with only a few word blocks or too many, I had to keep revising to ensure the maze was possible to complete.



Decision Points & Trade-offs

I changed the text size and colour to become more visible, I changed it to white and capped the size at 12, this was to make the text more visible and also because the original code cut off the text in level 3. Another decision I made was to make level 3 more difficult than what the code gave. It gave me a maze that would not allow the user to go from start to finish, so I decided to break down parts of that wall to make it possible for the user to complete the game.

Verification & Judgement

My change was appropriate based on the assignment criteria that the code had to work, if the maze was not possible to complete then the game does not function as it should. Seeing this, I had to make the change to ensure the maze was complete and would be able to go from start to finish as it can be frustrating and confusing if it didn't.

Limitations, Dead Ends, or Open Questions

Some things that didn't work out was the way all the error blocks were displayed, some were intentional while others were not, I left this as I felt the game was complete even with the random misplaced word blocks. Something that did not work as expected was the code for the

level 3 maze, it took a while for me to figure out how it worked but I managed to through trial and error through the live server. I tried my best to edit the word block code to appear in a more intentional way but I just kept whatever may have been unintentional especially since it was made through a loop.

Appendix

What I Gave GenAI:

For this code, Generate a level using JSON data; use loops to dynamically place word blocks like (ERROR) for some areas to make it more difficult to complete the maze in level 3. Use the base code and add/replace where needed to fulfill this. The following is the code with its page labels: Index

```
<html lang="en">
```

```
<head>
```

```
<meta charset="utf-8" />
```

```
<meta name="viewport" content="width=device-width, initial-scale=1" />
```

```
<title>Example 4 — Playable Maze (JSON + Classes)</title>
```

```
<link rel="stylesheet" href="style.css" />
```

```
</head>
```

```
<body>
```

```
<!-- Load p5 first -->
```

```
<script src="https://cdn.jsdelivr.net/npm/p5@1.9.0/lib/p5.min.js"></script>
```

```
<!-- Load your classes BEFORE sketch.js so they exist when sketch runs -->
```

```
<script src="level.js"></script>
```

```
<script src="player.js"></script>
```

```
<!-- Then load the main sketch (preload/setup/draw) -->
```

```
<script src="sketch.js"></script>
```

```
</body>
```

```
</html> jsconfig.json {
```

```
  "include": [
```

```
    "*.js",
```

```
    "libraries/*.js",
```

```
    "c:\\Users\\david\\.vscode\\extensions\\wmcicompsci.cs30-p5-1.9.3\\p5types\\global.d.ts"
```

```
  ]
```

```
} level.js /*
```

```
Level.js
```

A Level represents ONE maze grid loaded from levels.json.

Tile legend (from your original example):

0 = floor
1 = wall
2 = start
3 = goal

Responsibilities:

- Store the grid
- Find the start tile
- Provide collision/meaning queries (isWall, isGoal, inBounds)
- Draw the tiles (including a goal highlight)

*/

```
class Level {
  constructor(grid, tileSize) {
    // Store the tile grid and tile size (pixels per tile).
    this.grid = grid;
    this.ts = tileSize;

    // Start position in grid coordinates (row/col).
    // We compute this by scanning for tile value 2.
    this.start = this.findStart();

    // Optional: if you don't want the start tile to remain "special"
    // after you've used it to spawn the player, you can normalize it
    // to floor so it draws like floor and behaves like floor.
    if (this.start) {
      this.grid[this.start.r][this.start.c] = 0;
    }
  }

  // ----- Size helpers -----

  rows() {
    return this.grid.length;
  }

  cols() {
    return this.grid[0].length;
  }

  pixelWidth() {
    return this.cols() * this.ts;
  }
}
```

```
pixelHeight() {  
    return this.rows() * this.ts;  
}
```

```
// ----- Semantic helpers -----
```

```
inBounds(r, c) {  
    return r >= 0 && c >= 0 && r < this.rows() && c < this.cols();  
}
```

```
tileAt(r, c) {  
    // Caller should check inBounds first.  
    return this.grid[r][c];  
}
```

```
isWall(r, c) {  
    return this.tileAt(r, c) === 1;  
}
```

```
isGoal(r, c) {  
    return this.tileAt(r, c) === 3;  
}
```

```
// ----- Start-finding -----
```

```
findStart() {  
    // Scan entire grid to locate the tile value 2 (start).  
    for (let r = 0; r < this.rows(); r++) {  
        for (let c = 0; c < this.cols(); c++) {  
            if (this.grid[r][c] === 2) {  
                return { r, c };  
            }  
        }  
    }  
}
```

```
// If a level forgets to include a start tile, return null.  
// (Then the game can choose a default spawn.)  
return null;  
}
```

```
// ----- Drawing -----
```

```
draw() {  
    /*
```

Draw each tile as a rectangle.

Visual rules (matches your original logic):

- Walls (1): dark teal
- Everything else: light floor
- Goal tile (3): add a highlighted inset rectangle

```
*/
for (let r = 0; r < this.rows(); r++) {
  for (let c = 0; c < this.cols(); c++) {
    const v = this.grid[r][c];

    // Base tile fill
    if (v === 1) fill(30, 50, 60);
    else fill(232);

    rect(c * this.ts, r * this.ts, this.ts, this.ts);

    // Goal highlight overlay (only on tile 3).
    if (v === 3) {
      noStroke();
      fill(255, 200, 120, 200);
      rect(
        c * this.ts + 4,
        r * this.ts + 4,
        this.ts - 8,
        this.ts - 8,
        6
      );
    }
  }
}
}
}
}
}
level.json {
  "levels": [
    [
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
      [1, 2, 0, 0, 0, 1, 0, 0, 0, 1],
      [1, 0, 1, 1, 0, 1, 0, 1, 0, 1],
      [1, 0, 1, 0, 0, 0, 0, 1, 0, 1],
      [1, 0, 1, 0, 1, 1, 0, 1, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 3, 1],
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
    ],
  ],
}
```

```

[
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
  [1, 2, 0, 0, 1, 0, 0, 0, 3, 1],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 1],
  [1, 0, 1, 0, 0, 0, 1, 0, 0, 1],
  [1, 0, 0, 0, 1, 0, 0, 1, 0, 1],
  [1, 1, 1, 0, 1, 0, 0, 1, 0, 1],
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]
]
}
player.js /*
Player.js

```

A Player stores the avatar position in grid coordinates (row/col) and knows how to:

- draw itself
- attempt a move (tile-by-tile) with collision rules

The Player does NOT:

- load JSON
- switch levels

Those are "game orchestration" responsibilities that belong in sketch.js.
*/

```

class Player {
  constructor(tileSize) {
    this.ts = tileSize;

    // Current grid position (row/col).
    this.r = 0;
    this.c = 0;

    // Movement throttle (so a key press doesn't move 60 tiles per second).
    this.movedAt = 0;
    this.moveDelay = 90; // ms
  }

  // Place the player at a specific grid location (e.g., the level's start).
  setCell(r, c) {
    this.r = r;
    this.c = c;
  }
}

```

```
// Convert grid coords to pixel center (for drawing a circle).
pixelX() {
  return this.c * this.ts + this.ts / 2;
}
```

```
pixelY() {
  return this.r * this.ts + this.ts / 2;
}
```

```
draw() {
  // Same "simple high-contrast avatar" idea as your original.
  fill(20, 120, 255);
  circle(this.pixelX(), this.pixelY(), this.ts * 0.6);
}
```

```
/*
Try to move by (dr, dc) tiles.
```

Inputs:

- level: a Level instance, used for bounds + wall collision + goal detection
- dr/dc: desired movement step, typically -1,0,1

Returns:

- true if the move happened
- false if blocked or throttled

```
*/
```

```
tryMove(level, dr, dc) {
  // Throttle discrete movement using millis()
  const now = millis();
  if (now - this.movedAt < this.moveDelay) return false;
```

```
  const nr = this.r + dr;
  const nc = this.c + dc;
```

```
  // Prevent walking off the map.
  if (!level.inBounds(nr, nc)) return false;
```

```
  // Prevent walking into walls.
  if (level.isWall(nr, nc)) return false;
```

```
  // Movement is allowed, so commit.
  this.r = nr;
  this.c = nc;
  this.movedAt = now;
```



```
    return true;
  }
}
```

sketch.js /*

Week 4 — Example 4: Playable Maze (JSON + Level class + Player class)

Course: GBDA302

Instructors: Dr. Karen Cochrane and David Han

Date: Feb. 5, 2026

This is the "orchestrator" file:

- Loads JSON levels (preload)
- Builds Level objects
- Creates/positions the Player
- Handles input + level switching

It is intentionally light on "details" because those are moved into:

- Level.js (grid + drawing + tile meaning)
- Player.js (position + movement rules)

Based on the playable maze structure from Example 3

*/

```
const TS = 32;
```

```
// Raw JSON data (from levels.json).
```

```
let levelsData;
```

```
// Array of Level instances.
```

```
let levels = [];
```

```
// Current level index.
```

```
let li = 0;
```

```
// Player instance (tile-based).
```

```
let player;
```

```
function preload() {
```

```
  // Ensure level data is ready before setup runs.
```

```
  levelsData = loadJSON("levels.json");
```

```
}
```

```
function setup() {
```

```
  /*
```

```

Convert raw JSON grids into Level objects.
levelsData.levels is an array of 2D arrays.
*/
levels = levelsData.levels.map((grid) => new Level(copyGrid(grid), TS));

// Create a player.
player = new Player(TS);

// Load the first level (sets player start + canvas size).
loadLevel(0);

noStroke();
textFont("sans-serif");
textSize(14);
}

function draw() {
  background(240);

  // Draw current level then player on top.
  levels[li].draw();
  player.draw();

  drawHUD();
}

function drawHUD() {
  // HUD matches your original idea: show level count and controls.
  fill(0);
  text(`Level ${li + 1}/${levels.length} — WASD/Arrows to move`, 10, 16);
}

function keyPressed() {
  /*
  Convert key presses into a movement direction. (WASD + arrows)
  */
  let dr = 0;
  let dc = 0;

  if (keyCode === LEFT_ARROW || key === "a" || key === "A") dc = -1;
  else if (keyCode === RIGHT_ARROW || key === "d" || key === "D") dc = 1;
  else if (keyCode === UP_ARROW || key === "w" || key === "W") dr = -1;
  else if (keyCode === DOWN_ARROW || key === "s" || key === "S") dr = 1;
  else return; // not a movement key

```

```

// Try to move. If blocked, nothing happens.
const moved = player.tryMove(levels[li], dr, dc);

// If the player moved onto a goal tile, advance levels.
if (moved && levels[li].isGoal(player.r, player.c)) {
  nextLevel();
}
}

```

// ----- Level switching -----

```

function loadLevel(idx) {
  li = idx;

  const level = levels[li];

  // Place player at the level's start tile (2), if present.
  if (level.start) {
    player.setCell(level.start.r, level.start.c);
  } else {
    // Fallback spawn: top-left-ish (but inside bounds).
    player.setCell(1, 1);
  }

  // Ensure the canvas matches this level's dimensions.
  resizeCanvas(level.pixelWidth(), level.pixelHeight());
}

```

```

function nextLevel() {
  // Wrap around when we reach the last level.
  const next = (li + 1) % levels.length;
  loadLevel(next);
}

```

// ----- Utility -----

```

function copyGrid(grid) {
  /*
  Make a deep-ish copy of a 2D array:
  - new outer array
  - each row becomes a new array

```

Why copy?

- Because Level constructor may normalize tiles (e.g., replace 2 with 0)
- And we don't want to accidentally mutate the raw JSON data object.

```
*/
return grid.map((row) => row.slice());
}
style.css html, body {
  margin: 0;
  padding: 0;
}

canvas {
  display: block;
}
```

What GenAI gave me:

To achieve this, we need to make three specific updates:

1. `level1.json`: Add a third grid to the levels array.
2. `level1.js`: Update the `draw()` method to detect a new tile type (e.g., 4) and render the "ERROR" text.
3. `player.js`: Update the `tryMove` logic so that the player cannot walk through these "ERROR" blocks, treating them as obstacles.

Here are the updated files:

1. `level1.json`

Add the third level. I've designed this one with more open space so the "ERROR" blocks (represented by the number 4) can be placed dynamically via loops in the code, or manually here.

JSON

```
{
  "levels": [
    [
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
      [1, 2, 0, 0, 0, 1, 0, 0, 0, 1],
      [1, 0, 1, 1, 0, 1, 0, 1, 0, 1],
      [1, 0, 1, 0, 0, 0, 0, 1, 0, 1],
      [1, 0, 1, 0, 1, 1, 0, 1, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 3, 1],
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
    ]
  ]
}
```

```

],
[
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
  [1, 2, 0, 0, 1, 0, 0, 0, 3, 1],
  [1, 0, 1, 0, 1, 0, 1, 0, 1, 1],
  [1, 0, 1, 0, 0, 0, 1, 0, 0, 1],
  [1, 0, 0, 0, 1, 0, 0, 1, 0, 1],
  [1, 1, 1, 0, 1, 0, 0, 1, 0, 1],
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
],
[
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
  [1, 2, 0, 0, 0, 0, 0, 0, 0, 1],
  [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
  [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
  [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
  [1, 0, 0, 0, 0, 0, 0, 0, 3, 1],
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]
]
}

```

2. `level.js` (Updated)

We add a helper to check for the error block and update the `draw` loop to render the text.

JavaScript

```

class Level {
  constructor(grid, tileSize) {
    this.grid = grid;
    this.ts = tileSize;
    this.start = this.findStart();

    if (this.start) {
      this.grid[this.start.r][this.start.c] = 0;
    }
  }

  rows() { return this.grid.length; }
  cols() { return this.grid[0].length; }
  pixelWidth() { return this.cols() * this.ts; }
  pixelHeight() { return this.rows() * this.ts; }
}

```

```
inBounds(r, c) { return r >= 0 && c >= 0 && r < this.rows() && c < this.cols(); }  
tileAt(r, c) { return this.grid[r][c]; }
```

```
// Detect both Walls (1) and Error Blocks (4) as obstacles
```

```
isWall(r, c) {  
  const v = this.tileAt(r, c);  
  return v === 1 || v === 4;  
}
```

```
isGoal(r, c) { return this.tileAt(r, c) === 3; }
```

```
findStart() {  
  for (let r = 0; r < this.rows(); r++) {  
    for (let c = 0; c < this.cols(); c++) {  
      if (this.grid[r][c] === 2) return { r, c };  
    }  
  }  
  return null;  
}
```

```
draw() {  
  for (let r = 0; r < this.rows(); r++) {  
    for (let c = 0; c < this.cols(); c++) {  
      const v = this.grid[r][c];  
  
      if (v === 1) fill(30, 50, 60); // Wall  
      else if (v === 4) fill(200, 50, 50); // Error Block Background (Reddish)  
      else fill(232); // Floor  
  
      rect(c * this.ts, r * this.ts, this.ts, this.ts);  
  
      // Render "ERROR" text for tile type 4  
      if (v === 4) {  
        fill(255);  
        textAlign(CENTER, CENTER);  
        textSize(8);  
        text("ERROR", c * this.ts + this.ts / 2, r * this.ts + this.ts / 2);  
      }  
  
      if (v === 3) {  
        noStroke();  
        fill(255, 200, 120, 200);  
        rect(c * this.ts + 4, r * this.ts + 4, this.ts - 8, this.ts - 8, 6);  
      }  
    }  
  }  
}
```

```
}  
}  
}  
}
```

3. `sketch.js` (Updated)

We use a loop inside `setup()` to inject the "(ERROR)" blocks into Level 3 dynamically.

JavaScript

// ... (previous variables remain the same)

```
function setup() {  
  levels = levelsData.levels.map((grid) => new Level(copyGrid(grid), TS));
```

```
  // --- DYNAMIC ERROR BLOCK GENERATION FOR LEVEL 3 ---
```

```
  // We target index 2 (Level 3)
```

```
  let level3 = levels[2];
```

```
  // Example: Fill every second column in row 3 with ERROR blocks
```

```
  for (let c = 2; c < level3.cols() - 2; c += 2) {
```

```
    level3.grid[3][c] = 4;
```

```
  }
```

```
  // Example: Add a vertical "glitch" wall in column 6
```

```
  for (let r = 1; r < level3.rows() - 1; r++) {
```

```
    if (r !== 3) { // Leave a small gap
```

```
      level3.grid[r][6] = 4;
```

```
    }
```

```
  }
```

```
  // -----
```

```
  player = new Player(TS);
```

```
  loadLevel(0);
```

```
  noStroke();
```

```
  textFont("sans-serif");
```

```
}
```

```
// ... (rest of the sketch.js functions remain the same)
```

What changed?

- **The Logic:** In `Level.js`, `isWall` now checks for `v === 4`. Since `Player.js` already calls `level.isWall()`, the player is automatically blocked by the "ERROR" tiles without needing to change the Player class at all.
- **The Loop:** In `sketch.js`, we used two `for` loops to modify `levels[2].grid`. This allows you to create patterns (like a checkerboard or a row of obstacles) without having to type every "4" into the JSON file manually.

Would you like me to add a "glitch" effect where the ERROR blocks move or change positions every few seconds?

What I Gave to GenAI:

For the portion of code under "function setup()" {" write corresponding comments const TS = 32;

```
// Raw JSON data (from levels.json).
```

```
let levelsData;
```

```
// Array of Level instances.
```

```
let levels = [];
```

```
// Current level index.
```

```
let li = 0;
```

```
// Player instance (tile-based).
```

```
let player;
```

```
function preload() {
```

```
  // Ensure level data is ready before setup runs.
```

```
  levelsData = loadJSON("levels.json");
```

```
}
```

```
function setup() {
```

```
  levels = levelsData.levels.map((grid) => new Level(copyGrid(grid), TS));
```

```
  // --- DYNAMIC ERROR BLOCK GENERATION FOR LEVEL 3 ---
```

```
  // Index 2 is Level 3
```

```
  let level3 = levels[2];
```

```
  let rows = level3.rows();
```

```
  let cols = level3.cols();
```

```
  // 1. ORIGINAL: Fill every second column in row 3
```

```
  for (let c = 1; c < cols - 2; c += 3) {
```



```

    level3.grid[3][c] = 4;
}

// 2. ORIGINAL: Vertical "glitch" wall in column 6
for (let r = 1; r < rows - 1; r += 3) {
    if (r !== 2) level3.grid[r][3] = 4;
}

for (let r = 2; r < rows - 2; r += 2) {
    if (r !== 3) level3.grid[r][5] = 4;
}

for (let r = 1; r < 2; r++) {
    for (let c = cols - 1; c < cols - 2; c += 2) {
        level3.grid[r][c] = 4;
    }
}

// 4. NEW: Checkerboard "Corridor" (Bottom Section)
// Places error blocks in a checkerboard pattern near the bottom
for (let c = 1; c < cols - 1; c++) {
    if (c % 2 === 0) {
        level3.grid[rows - 3][c] = 4;
    }
}
// -----

player = new Player(TS);
loadLevel(0);
noStroke();
textFont("sans-serif");
}

function draw() {
    background(240);

    // Draw current level then player on top.
    levels[li].draw();
    player.draw();

    drawHUD();
}

function drawHUD() {

```

```

// HUD matches your original idea: show level count and controls.
fill(255);
textSize(12);
textAlign(LEFT, TOP);
text(`Level ${li + 1}/${levels.length} — WASD/Arrows to move`, 10, 16);
}

function keyPressed() {
  /*
  Convert key presses into a movement direction. (WASD + arrows)
  */
  let dr = 0;
  let dc = 0;

  if (keyCode === LEFT_ARROW || key === "a" || key === "A") dc = -1;
  else if (keyCode === RIGHT_ARROW || key === "d" || key === "D") dc = 1;
  else if (keyCode === UP_ARROW || key === "w" || key === "W") dr = -1;
  else if (keyCode === DOWN_ARROW || key === "s" || key === "S") dr = 1;
  else return; // not a movement key

  // Try to move. If blocked, nothing happens.
  const moved = player.tryMove(levels[li], dr, dc);

  // If the player moved onto a goal tile, advance levels.
  if (moved && levels[li].isGoal(player.r, player.c)) {
    nextLevel();
  }
}

// ----- Level switching -----

function loadLevel(idx) {
  li = idx;

  const level = levels[li];

  // Place player at the level's start tile (2), if present.
  if (level.start) {
    player.setCell(level.start.r, level.start.c);
  } else {
    // Fallback spawn: top-left-ish (but inside bounds).
    player.setCell(1, 1);
  }
}

```

```

    // Ensure the canvas matches this level's dimensions.
    resizeCanvas(level.pixelWidth(), level.pixelHeight());
}

```

```

function nextLevel() {
    // Wrap around when we reach the last level.
    const next = (li + 1) % levels.length;
    loadLevel(next);
}

```

// ----- Utility -----

```

function copyGrid(grid) {
    /*
    Make a deep-ish copy of a 2D array:
    - new outer array
    - each row becomes a new array

    Why copy?
    - Because Level constructor may normalize tiles (e.g., replace 2 with 0)
    - And we don't want to accidentally mutate the raw JSON data object.
    */
    return grid.map((row) => row.slice());
}

```

What GenAI Gave Me:

Here is the updated `setup()` function with detailed comments explaining the logic behind the dynamic level generation and the initialization steps.

JavaScript

```

function setup() {

    // Convert the raw nested arrays from levelsData into an array of Level objects.

    // We use copyGrid to ensure we don't accidentally modify the original JSON data.

    levels = levelsData.levels.map((grid) => new Level(copyGrid(grid), TS));

    // --- DYNAMIC ERROR BLOCK GENERATION FOR LEVEL 3 ---

    // Select the third level (index 2) to inject extra obstacles programmatically.

```

```
let level3 = levels[2];
```

```
let rows = level3.rows();
```

```
let cols = level3.cols();
```

```
// 1. Horizontal Obstacles: Iterate through columns and place an ERROR block (4)
```

```
// every 3 tiles along row index 3 to create a broken barrier.
```

```
for (let c = 1; c < cols - 2; c += 3) {
```

```
    level3.grid[3][c] = 4;
```

```
}
```

```
// 2. Vertical Obstacles (Set A): Create a dashed vertical line in column 3.
```

```
// The 'if' statement ensures a gap is left at row 2 so the player isn't fully trapped.
```

```
for (let r = 1; r < rows - 1; r += 3) {
```

```
    if (r !== 2) level3.grid[r][3] = 4;
```

```
}
```

```
// 3. Vertical Obstacles (Set B): Create another dashed line in column 5.
```

```
// This uses a different step (r += 2) and skips row 3 to create staggered movement paths.
```

```
for (let r = 2; r < rows - 2; r += 2) {
```

```
    if (r !== 3) level3.grid[r][5] = 4;
```

```
}
```

```
// This loop is currently configured to target row 1, but the inner loop condition
```

```
// (cols - 1 < cols - 2) is logically false, so it acts as a placeholder for
```

```
// future top-row obstacles.
```

```
for (let r = 1; r < 2; r++) {  
  for (let c = cols - 1; c < cols - 2; c += 2) {  
    level3.grid[r][c] = 4;  
  }  
}
```

```
// 4. NEW Checkerboard "Corridor": Adds obstacles to every even column in
```

```
// the second-to-last row, forcing the player to "zigzag" to reach the goal.
```

```
for (let c = 1; c < cols - 1; c++) {  
  if (c % 2 === 0) {  
    level3.grid[rows - 3][c] = 4;  
  }  
}  
  
// -----
```

```
// Initialize the Player instance with the defined tile size.
```

```
player = new Player(TS);
```

```
// Set the game state to start at Level 1 (index 0).
```

```
loadLevel(0);
```

```
// Global visual settings for the P5.js canvas.
```

```
noStroke();
```

```
    textFont("sans-serif");  
}
```

Key Takeaways for Level 3:

- **Logical Barriers:** By using `if (r !== 2)` or specific step increments like `+= 3`, you ensure that the maze remains solvable while still looking "glitched" or chaotic.
- **Grid Reference:** Remember that `level3.grid[row][col]` is the standard access pattern.
- **Scalability:** Since we use `rows` and `cols` variables instead of hardcoded numbers like `10`, these loops will still work even if you change the size of Level 3 in your `level.json` later.

Would you like me to show you how to make these "ERROR" blocks flicker or change colors randomly during the `draw()` loop?