

Process & Decision Documentation

Side Quests and A4 (Individual Work)

- I decided to refactor the collision detection logic in the Level class from an "allow-list" (only allowing movement on tile 0) to a "block-list" (only blocking movement on tile 1).
- When I introduced the Lava tile (4), the original logic treated it as a wall because it wasn't a floor (0), making it impossible for the player to step on it and trigger the reset mechanic.
- This allowed the player to walk onto hazardous tiles, enabling the sketch.js game loop to detect the specific tile value under the player and trigger the "reset to start" function.

GenAI Documentation

Date Used: February 5, 2026

Tool Disclosure: Gemini 3

Purpose of Use: Debugging logic errors (specifically why the player wouldn't reset upon touching lava) and generating comments for the final code.

Summary of Interaction: I provided my sketch.js and level.js files to the tool. It identified that my isWall boolean was returning true for lava tiles and provided corrected logic to find the start position (2) dynamically in the setup loop.

Human Decision Point(s): Gemini initially suggested a JSON structure that didn't match my specific "raw array" format. I rejected that suggestion and forced the code to work with my existing levels.json structure by modifying how the array length was calculated in the loops.

Integrity & Verification Note: I verified the output by playtesting the game. I physically moved the character onto the red tile to ensure the reset happened immediately and changed a bit of the visual of the maze with grid.

Scope of GenAI Use: The core Object-Oriented structure (Player class, Level class) was derived from the course slides (Week 4). GenAI was only used to implement the specific "Lava" interaction logic and the level-loading function.

Limitations or Misfires: The tool initially failed to account for the order of operations in my HTML file, causing a blank screen until I manually reordered the script tags to load the libraries before the classes.

Summary of Process (Human + Tool)

- Started by implementing the static grid from the Week 4 curriculum examples.
- Manually edited the levels.json file to introduce the new integer 4 for Lava.

- Encountered a "logic bug" where the Lava rendered correctly (red) but functioned like a wall; used GenAI to diagnose the Player.tryMove function.
- Iterated on the sketch.js file to ensure the game could handle multiple levels, adding a check to load the next array index when the goal was reached.

Verification & Judgement

- I ran the game multiple times, intentionally trying to break the walls and walking into lava from different angles to ensure the collision box was accurate.
- I also reviewed the Week 4 slides to ensure my use of loadJSON in preload() matched the "Data Separation" pattern taught in class.

Limitations, Dead Ends, or Open Questions

- The game currently assumes the start position is always tile 2; if I forget to put a 2 in the JSON, the game crashes.
- I originally wanted moving enemies, but due to the complexity. I just simplified the hazard to static Lava tiles.

Appendix

I am a student want to build a tile-based maze game in p5.js using Object Oriented Programming. Before start coding can you explain what is the main logic in this?

Based on the "Week 4 - part 2" lecture slides, building a tile-based maze game using Object-Oriented Programming (OOP) relies on breaking the game into distinct "responsibilities" and separating your data from your code.

Here is the main logic explained in four key parts:

1. The World is a Grid (2D Array)

The fundamental logic of a tile-based game is representing the world as a 2D array of numbers, where each number represents a specific type of terrain.

- **The Data:** You use a grid where 1 might be a wall, 0 might be a floor, and other numbers (like 3) can represent goals or start points.

+1

- **The Visuals:** Instead of drawing random rectangles, your code loops through this array. If it sees a 1, it draws a wall color; if it sees a 0, it draws a floor color.

2. Classes: Organizing by Responsibility

Instead of writing all your code in the main `draw()` function (which creates "spaghetti code"), you use OOP to create specific objects for different parts of the game.

+1

- **The Level Class:**

- **Responsibility:** It handles the environment. It bundles the grid data and the behavior to draw that grid into one object.

+1

- **Logic:** It takes the grid data in its constructor and has a `draw()` method that iterates over the rows and columns to render the tiles.

+1

- **The Player Class:**

- **Responsibility:** It handles the character. It keeps track of its own position (row/column or x/y) and handles movement logic.

+1

- **Interaction:** The player object is separate from the level object, making it easier to manage and modify.

3. JSON: Separating Data from Code

A major part of the logic is moving the level designs out of your JavaScript code and into a separate JSON file.

- **Why:** This allows you to add or edit levels (like changing the wall layout) without rewriting your actual code logic.
- **How:** You use `loadJSON` in the `preload()` function to import an array of level configurations. You then feed this data into your `Level` class when the game starts.

+1

4. The Game Loop & Collision Logic

The gameplay logic connects the **Player** and the **Level**:

- **Rendering:** In your main `draw()` loop, you simply tell the current level object to draw itself, and then tell the player object to draw itself.
- **Movement & Collision:** When a key is pressed, the `Player` class checks the `Level` class's grid data before moving.

- *Logic*: "If I try to move right, is the tile at grid[current_row][next_col] a wall (1)?" If yes, the move is blocked.
- **Level Switching**: If the player moves onto a goal tile (e.g., value 3), the game logic detects a "win" and loads the next grid from the JSON data.

+1

Summary of the Architecture

The slide deck summarizes this "Single Responsibility" pattern as follows:

- **Level Class**: Holds the map.
- **Player Class**: Holds the character.
- **JSON**: Holds the raw data (configurations).
- **Sketch.js**: Acts as the conductor, passing data from JSON to the classes and managing the game state.

Would you like to see a code example of how the **Level Class** constructor handles the JSON data?

I have an example for it. Now, may you please draft an interesting game play based on the current array I have?

Based on the "Week 4" slides and the standard tile-based maze structure you are using, here is a draft for a gameplay mechanic called "**The Floor is Lava**".

This concept uses your existing **Array/Grid system** but adds a new number (4) to represent hazardous terrain. This forces the player to move carefully rather than just spamming keys.

The Gameplay Concept: "Volcano Escape"

- **Goal**: Reach the green tile (Exit/3) to escape.
- **Constraint**: You cannot touch the walls (1), and if you touch the red tiles (4), you are sent back to the start (2).
- **Layout**: A narrow winding path surrounded by "Lava" tiles.

Part 1: Update Your Data (levels.json)

We will modify your JSON array to include the new number 4. Copy this block into your levels.json file.

The Design Strategy:

- The path is safe (0) but narrow.
- The danger (4) is everywhere else.

- We use 2 for Start and 3 for the Goal.

JSON

```
{
  "levels": [
    {
      "rows": 10,
      "cols": 10,
      "grid": [
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 2, 0, 4, 4, 4, 0, 0, 3, 1],
        [1, 1, 0, 4, 4, 4, 0, 1, 1, 1],
        [1, 0, 0, 0, 0, 4, 0, 4, 4, 1],
        [1, 0, 4, 4, 0, 4, 0, 4, 4, 1],
        [1, 0, 4, 4, 0, 0, 0, 4, 4, 1],
        [1, 0, 4, 1, 1, 1, 1, 4, 4, 1],
        [1, 0, 0, 0, 0, 0, 0, 0, 4, 1],
        [1, 1, 1, 1, 1, 1, 1, 4, 4, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
      ]
    }
  ]
}
```

Part 2: Update Your Logic

You only need to make two small changes to your classes to make this work.

1. Visuals: Update level.js

In your Level class, inside the draw() method, add a condition to color the lava tiles.

JavaScript

```
// Inside level.js -> draw()
for (let r = 0; r < this.rows; r++) {
  for (let c = 0; c < this.cols; c++) {
    let tile = this.grid[r][c];

    if (tile === 1) fill(50);      // Wall (Grey)
    else if (tile === 0) fill(255); // Floor (White)
    else if (tile === 2) fill(0, 0, 255); // Start (Blue)
    else if (tile === 3) fill(0, 255, 0); // Goal (Green)

    // --- NEW CODE ---
    else if (tile === 4) fill(255, 50, 50); // Lava (Red)
    // -----

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

2. Interaction: Update player.js (or sketch.js)

You need to check what tile the player is standing on *after* they move.

If you are handling movement in sketch.js (like in the slide examples), modify your keyPressed function:

JavaScript

```
function keyPressed() {  
  // ... existing movement logic ...  
  
  // Get the tile value at the player's new position  
  let currentTile = level.grid[player.r][player.c];  
  
  // Check for Lava  
  if (currentTile === 4) {  
    console.log("OUCH! Lava!");  
    // Reset player to start position (you might need to store startR and startC)  
    player.r = level.startR;  
    player.c = level.startC;  
  }  
  
  // Check for Goal (Standard win condition)  
  if (currentTile === 3) {  
    console.log("You Win!");  
    // Load next level...  
  }  
}
```

Why this is "Interesting" (OOP Perspective)

- **Scalability:** You didn't write "draw a red square at x=50, y=50". You created a *rule* for 4. Now you can paint "lava" all over your levels in the JSON file without writing any more code.
- **Data-Driven Design:** You are now designing the game in the *data* (JSON), not the logic. This is how professional game engines work.

Next Step: Would you like to know how to add a "moving enemy" class that patrols the 0 tiles?

What about the index.html, currently nothing is showing up, how would I arrange the order

I have the correct order, but isn't working. To debug check in the level.js draw function you previously created, something is incorrect here:

```
// Inside level.js -> draw()  
  
for (let r = 0; r < this.rows; r++) {  
  for (let c = 0; c < this.cols; c++) {
```

```

let tile = this.grid[r][c];

if (tile === 1) fill(50);    // Wall (Grey)
else if (tile === 0) fill(255); // Floor (White)
else if (tile === 2) fill(0, 0, 255); // Start (Blue)
else if (tile === 3) fill(0, 255, 0); // Goal (Green)

// --- NEW CODE ---
else if (tile === 4) fill(255, 50, 50); // Lava (Red)
// -----

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

```

You are absolutely right to spot that. The issue is in the loop conditions: `this.rows` and `this.cols` likely do not exist as properties on your object unless you manually calculated and assigned them in the constructor.

In JavaScript (and the examples from your slides), arrays determine their own size. You should ask the **grid array** for its length, not the class object.

The Fix

Change the loop conditions to use `this.grid.length` (for rows) and `this.grid[0].length` (for columns).

Here is the corrected `draw()` function for your `level.js`:

JavaScript

```

// Inside level.js

draw() {
  // FIX 1: Use 'this.grid.length' to get the total number of rows
  for (let r = 0; r < this.grid.length; r++) {

    // FIX 2: Use 'this.grid[r].length' to get the columns in the current row
    for (let c = 0; c < this.grid[r].length; c++) {

```

```

let tile = this.grid[r][c];

// Set the color based on the tile number
if (tile === 1) {
  fill(50);      // Wall
} else if (tile === 0) {
  fill(255);     // Floor
} else if (tile === 2) {
  fill(0, 0, 255); // Start
} else if (tile === 3) {
  fill(0, 255, 0); // Goal
} else if (tile === 4) {
  fill(255, 50, 50); // Lava
}

// Draw the rectangle
// x position = column * tileSize
// y position = row * tileSize
rect(c * this.ts, r * this.ts, this.ts, this.ts);
}
}
}

```

Why the original code failed

In the slides, the dimensions are accessed via methods or properties of the array itself (`this.grid.length`). If you try to use `this.rows` without defining it in your constructor (e.g., `this.rows = grid.length`), it returns `undefined`. In a loop, `0 < undefined` is `false`, so the loop never runs, and nothing draws.

There isn't Lava displayed. why will this occur?

Based on the files you provided and your recent code changes, the reason the "Lava" is not appearing is that **your map data (levels.json) does not contain the number 4**.

You updated the *logic* to draw red tiles when it sees a 4, but you updated your `levels.json` with a map that only contains 0 (floor), 1 (wall), 2 (start), and 3 (goal). The computer is drawing exactly what is in the file.

Here is how to fix it:

1. Update `levels.json` to include Lava

You need to manually change some of the 0s (floor) into 4s (lava) in your file.

Copy and paste this into your `levels.json`:

(I have modified your map to include the number 4)

JSON


```

{
  "levels": [
    [
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
      [1, 2, 0, 4, 4, 4, 0, 0, 3, 1],
      [1, 1, 0, 1, 1, 1, 0, 1, 1, 1],
      [1, 0, 0, 0, 0, 4, 0, 4, 4, 1],
      [1, 0, 4, 4, 0, 4, 0, 4, 4, 1],
      [1, 0, 4, 4, 0, 0, 0, 4, 4, 1],
      [1, 0, 4, 1, 1, 1, 1, 4, 4, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 4, 1],
      [1, 1, 1, 1, 1, 1, 1, 4, 4, 1],
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
    ]
  ]
}

```

2. Verify level.js Logic

Ensure your draw() function inside level.js has the correct loop logic and the color check for 4.

JavaScript

```

// Inside level.js -> draw()
// Note: Use 'this.grid.length' to avoid errors with the new JSON format
for (let r = 0; r < this.grid.length; r++) {
  for (let c = 0; c < this.grid[r].length; c++) {
    let type = this.grid[r][c];
    if (type === 1) {
      fill(100); // Wall
    } else if (type === 0) {
      fill(255); // Floor
    } else if (type === 2) {
      fill(0, 255, 0); // Start
    } else if (type === 3) {
      fill(255, 0, 0); // Goal
    } else if (type === 4) {
      fill(255, 100, 0); // <--- LAVA COLOR (Orange/Red)
    }
    rect(c * this.ts, r * this.ts, this.ts, this.ts);
  }
}

```

3. Verify sketch.js Loading

Since you changed your JSON format to a "raw array" (removing the "grid": label), you must ensure sketch.js does NOT look for .grid.

Correct setup logic in sketch.js:

JavaScript

```

// Inside setup()
let currentLevel = 0; // or whichever level index
let levelData = levelsData.levels[currentLevel]; // DIRECT ACCESS (No .grid)

```

```
// If you previously had: levelsData.levels[currentLevel].grid  
// CHANGE IT TO: levelsData.levels[currentLevel]
```

```
world = new Level(levelData, tileSize);
```

4. Important: Clear Cache

If you make these changes and still don't see red tiles, your browser is "remembering" the old JSON file.

- **Fix:** Press Ctrl + Shift + R (Windows) or Cmd + Shift + R (Mac) to force the new map to load.

Please fix If the player moves onto a lava tile (value `4`), they are reset to the start position.

Here is the fixed sketch.js code.

sketch.js

JavaScript

```
let levelsData; // JSON data loaded from the file  
let world;      // The current Level object  
let player;     // The Player object  
let currentLevel = 0; // Index of the current level  
let tileSize = 40;  // Size of each tile in pixels  
  
// Global variables: track the start position (2) for the current level  
// We need this to reset the player if they hit lava  
let startRow = 1;  
let startCol = 1;  
  
function preload() {  
  // Load the JSON file before the game starts  
  levelsData = loadJSON("levels.json");  
}  
  
function setup() {  
  createCanvas(400, 400);  
  
  // 1. Create the Player instance (only need to do this once)  
  player = new Player(tileSize);  
  
  // 2. Load the first level  
  loadCurrentLevel();  
}  
  
function draw() {  
  background(220);  
  
  // Draw the world first (background), then the player (foreground)  
  if (world) world.draw();  
  if (player) player.draw();  
}
```

```

function keyPressed() {
  // If the world or player hasn't loaded yet, do nothing
  if (!world || !player) return;

  // 1. Attempt to move
  // The player class handles wall collisions (Tile 1) internally
  let moved = false;
  if (keyCode === UP_ARROW) moved = player.tryMove(world, -1, 0);
  if (keyCode === DOWN_ARROW) moved = player.tryMove(world, 1, 0);
  if (keyCode === LEFT_ARROW) moved = player.tryMove(world, 0, -1);
  if (keyCode === RIGHT_ARROW) moved = player.tryMove(world, 0, 1);

  // 2. Logic to check what tile the player is standing on after moving
  if (moved) {
    let currentTile = world.grid[player.r][player.c];

    // --- CASE A: Stepped on Lava (4) ---
    if (currentTile === 4) {
      console.log("Stepped on lava! Resetting to start...");
      player.setCell(startRow, startCol); // Reset position to the start of the level
    }

    // --- CASE B: Reached the Goal (3) ---
    if (currentTile === 3) {
      console.log("Level Complete! Loading next level...");

      // Increment level index
      currentLevel++;

      // Check if there are more levels in the JSON data
      if (currentLevel < levelsData.levels.length) {
        loadCurrentLevel(); // Load the new level data
      } else {
        console.log("Congratulations! You have finished all levels!");
        noLoop(); // Stop the game loop
        // Optional: You could reset currentLevel = 0 here to loop the game
      }
    }
  }
}

// --- Helper Function: Handles loading level data ---
function loadCurrentLevel() {
  // 1. Get the grid array for the current level index
  let levelGrid = levelsData.levels[currentLevel];

  // 2. Create a new Level object with this grid
  world = new Level(levelGrid, tileSize);

  // 3. Resize canvas to fit the new level dimensions (Optional but recommended)
  // This ensures the whole map is visible if levels have different sizes
  resizeCanvas(world.grid[0].length * tileSize, world.grid.length * tileSize);

  // 4. Find the Start Position (2) in the new grid
  // We scan the array to find where the player should start

```

```
for (let r = 0; r < world.grid.length; r++) {  
  for (let c = 0; c < world.grid[r].length; c++) {  
    if (world.grid[r][c] === 2) {  
      startRow = r;  
      startCol = c;  
    }  
  }  
}
```

```
// 5. Move the player to the found start position  
player.setCell(startRow, startCol);
```