

Process & Decision Documentation

Project/Assignment Decisions

I decided to create a simple grid-based maze game with three levels of increasing complexity. Initially, I considered adding more interactive elements like collectible items or enemies, but I simplified the scope to focus on correctly implementing the core Week 4 concepts: JSON data loading, Level class structure, and dynamic tile placement using loops.

I chose to implement the bonus feature (automatic level progression) because it created a more seamless gameplay experience and demonstrated my understanding of game state management. This required adding a goal-checking method in the Player class that triggers level transitions.

One key decision was adjusting the UI text display. After testing, the instructions were difficult to read against the game background, so I added a semi-transparent white background box behind the text to improve visibility.

Role-Based Process Evidence

This section documents how the Group Charter development process unfolded using GenAI assistance, including iterations, decisions, and human oversight.

Name: Rini Lu

Role(s): Developer

Primary responsibility for this work: Setting up the project structure and designing level layouts

Goal of Work Session

Create the JSON file with level data and establish the basic Level class structure to dynamically render tiles from the JSON data.

Tools, Resources, or Inputs Used

- Week 4 lecture slides and example code (Example 3: JSON Levels + Level class)

- Claude (GenAI) for structure planning and debugging
- VS Code with Live Server extension
- p5.js reference documentation

GenAI Documentation

Date Used: February 5, 2026

Tool Disclosure: Claude (Anthropic), Sonnet 4.5

Purpose of Use:

Initial code structure planning for Level and Player classes

Debugging assistance when level transitions weren't working correctly

Help writing clear code comments

Summary of Interaction: I asked Claude how to structure a Level class that loads data from JSON and how to implement automatic level progression. Claude provided example code structures and explained the logic flow for checking win conditions and loading the next level.

Human Decision Point(s):

- Claude suggested using a more complex state management system, but I simplified it to just use a `loadLevel()` function with a level index
- Changed the tile colors from Claude's suggestions to ones that matched my visual preference (beige floors instead of gray)
- Modified the player size to be slightly larger than suggested for better visibility

Integrity & Verification Note

I tested each function individually in Live Server to ensure the logic worked correctly. I compared the class structure against the Week 4 examples to verify I was following the patterns taught in class.

Scope of GenAI Use

GenAI did not contribute to: the specific level layouts in the JSON file (I designed these myself), the choice to implement the bonus feature, or the final visual styling

decisions. Approximately 40% of the code structure came from GenAI suggestions, while 60% was written by me based on class examples.

Limitations or Misfires

Claude initially suggested a more complex level transition system with fade effects, but this was beyond the scope of a simple side quest. I had to clarify I wanted a minimal implementation.

Summary of Process (Human + Tool)

1. Started by reviewing Week 4 lecture examples, particularly Example 3 (JSON loading)
2. Created the `levels.json` file with three level layouts, testing each one's dimensions
3. Asked Claude for help structuring the Level class with proper methods
4. Wrote the Player class myself, referencing the blob example from Week 2
5. Debugged collision detection by testing in Live Server and using `console.log` statements
6. Iterated on level designs after realizing the first versions were too simple

Verification & Judgement

- Tested the game in Live Server after each major change
- Verified that all three levels loaded correctly and had playable paths to the goal
- Confirmed the game works in Google Chrome as required by the assignment
- Asked a roommate to playtest briefly - they confirmed the controls were intuitive

Appendix

How should I structure a Level class in p5.js that loads data from JSON for a grid-based maze game?

Structuring a Level class is a smart move. It keeps your sketch.js clean and makes it much easier to swap maps, change themes, or add new mechanics later without rewriting your entire logic.

Here is a clean, scalable structure for a grid-based maze game in p5.js.

1. The JSON Data Structure

Before writing the class, you need a consistent data format. A simple 2D array (grid) is usually the most intuitive approach.

JSON

```
{  
  "levelNumber": 1,  
  "rows": 5,  
  "cols": 5,  
  "tileSize": 40,  
  "grid": [  
    [1, 1, 1, 1, 1],  
    [1, 0, 0, 2, 1],  
    [1, 0, 1, 0, 1],  
    [1, 3, 0, 0, 1],  
    [1, 1, 1, 1, 1]  
  ]  
}
```

- 0: Empty Path
- 1: Wall
- 2: Start Point
- 3: Goal

2. The Level Class Structure

The class should handle loading, storing, and rendering the data. By passing the JSON object directly into the constructor, you keep the class decoupled from the file loading process.

JavaScript

```
class Level {  
    constructor(data) {  
        this.grid = data.grid;  
        this.rows = data.rows;  
        this.cols = data.cols;  
        this.tileSize = data.tileSize;  
    }  
  
    // Draw the maze based on the grid values  
    display() {  
        for (let r = 0; r < this.rows; r++) {  
            for (let c = 0; c < this.cols; c++) {  
                let x = c * this.tileSize;  
                let y = r * this.tileSize;  
  
                if (this.grid[r][c] === 1) {  
                    fill(50); // Wall color  
                } else if (this.grid[r][c] === 3) {  
                    fill(0, 255, 0); // Goal color  
                } else {  
                    fill(255); // Path color  
                }  
  
                stroke(200);  
                rect(x, y, this.tileSize, this.tileSize);  
            }  
        }  
    }  
  
    // A helper to check if a specific grid coordinate is a wall  
    isWall(col, row) {  
        // Boundary check
```

```
if (col < 0 || col >= this.cols || row < 0 || row >= this.rows) {
    return true;
}
return this.grid[row][col] === 1;
}
```

3. Implementation in sketch.js

Since `loadJSON` is asynchronous, you should call it in `preload()` to ensure the data is ready before the game starts.

JavaScript

```
let levelData;
let currentLevel;

function preload() {
    // Load your JSON file
    levelData = loadJSON('level1.json');
}

function setup() {
    createCanvas(400, 400);
    // Initialize the class with the loaded data
    currentLevel = new Level(levelData);
}

function draw() {
    background(220);
    currentLevel.display();
}
```

Pro-Tips for Scalability

- Asset Mapping: Instead of hardcoding colors (`fill(50)`), add an `assets` object to your class that maps numbers to images (e.g., `1: wallImage`).
- The "Tile" Class: If your maze tiles need to do things (like animate or hurt the player), consider making a Tile class and filling this.`grid` with Tile objects instead of raw numbers.
- Collision Helper: Your `isWall()` method is the "secret sauce." When your player tries to move, calculate their potential grid position and ask `currentLevel.isWall(newCol, newRow)` before allowing the move.

Would you like me to show you how to integrate a Player class that interacts with this Level's boundaries?