Process & Decision Documentation

Side Quests

Focus on:

This game is a tile-based puzzle game built using p5.js. The player controls a green circle to navigate through randomized mazes, avoiding walls to reach the red goal and advance to the next stage.

If GenAI was used (keep each response as brief as possible):

Date Used: Feb 7, 2026

Tool Disclosure: Gemini 3

Purpose of Use: Generate game code and make improvements.

Summary of Interaction: Iteratively developed a tile-based game from a JSON template, adding player movement, win conditions, and interface elements.

Human Decision Point(s): Designed all the game rules, add the specific control scheme (WASD/Arrows), the visual layout (centering), and the requirement for smooth vs. grid-based movement.

Integrity & Verification Note: Code was manually tested in a browser to verify that collision physics and level transitions worked correctly.

Scope of GenAI Use: Generated the majority of the sketch.js logic, including the Game Loop, Level Class, and collision algorithms.

Limitations or Misfires: Initial collision code caused the player to "snag" on wall edges, need use another prompt to adjust the hitbox size.

Summary of Process (Human + Tool)

- I provided the initial example code (index.html, levels.json) and specific gameplay goals. Gemini generated the JavaScript logic, after manual testing and multiple iterations, the game has successfully completed all requirements.

Decision Points & Trade-offs

- I chose a simple "bounding box" collision system instead of a complex physics engine to keep the code lightweight and readable.

- I decided to loop the levels infinitely rather than creating a "Game Over" screen to keep the gameplay flow continuous.

Verification & Judgement

- The primary verification method is playtesting. Initially, when the green circle became stuck against the wall, playtesting showed the collision logic was too strict.
- Additionally, manually completing the level ensures the “red square” trigger correctly resets the player and loads the next map.

Limitations, Dead Ends, or Open Questions

- The code initially generated by Gemini did not achieve the expected results, resulting in a rather chaotic game interface. After several iterations, this issue was successfully resolved.

Appendix

Appendix: GenAI Transcript

Tool Used: Gemini **Date:** Feb 7, 2026

User Prompt 1:

Based on this code, give me a small game code. place a red square on the tiles, along with a small green circle that can move freely. When the green square moves onto the red square, it means the level is cleared. make the second level that automatically loads after clearing the first. In each level, the positions of the red and green squares are randomized.

[uploaded index.html, jsconfig. json, levels.json, sketch.js, style.css]

Gemini:

Here is the updated `sketch.js` code. I have modified your existing structure to include a `resetLevel()` function that randomizes positions, a player object with collision detection, and logic to handle the win condition.
You can replace your entire `sketch.js` content with this code.

sketch.js
JavaScript

```
/*
Week 4 — Example 3 Extension: Random Placement Game
Based on provided levels.json and Level class structure.
*/

// ----------------------------
// Globals
// ----------------------------

let levelsData;
let levels = [];
let current = 0;
const TS = 32; // Tile Size

// Player object
let player = {
  x: 0,
  y: 0,
  size: 16,
  speed: 3
};

// Goal object
let goal = {
  c: 0, // grid column
  r: 0  // grid row
};

// ----------------------------
// Preload
// ----------------------------
function preload() {
  levelsData = loadJSON("levels.json");
}

// ----------------------------
// Level Class
// ----------------------------
class Level {
  constructor(grid, tileSize) {
    this.grid = grid;
    this.ts = tileSize;
  }

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

  // Helper to check if a specific grid cell is a wall
  isWall(c, r) {
    // Check bounds
    if (c < 0 || r < 0 || c >= this.cols() || r >= this.rows()) return true;
```

```
    // Return true if tile is 1 (wall)
    return this.grid[r][c] === 1;
  }

  draw() {
    for (let r = 0; r < this.rows(); r++) {
      for (let c = 0; c < this.cols(); c++) {
        const v = this.grid[r][c];
        // Draw walls (dark) and floors (light)
        noStroke();
        fill(v === 1 ? color(30, 50, 60) : color(220));
        rect(c * this.ts, r * this.ts, this.ts, this.ts);
      }
    }
  }
}

// ---------------------------
// Setup
// ---------------------------
function setup() {
  // Convert JSON data to Level objects
  levels = levelsData.levels.map((grid) => new Level(grid, TS));

  // Initialize the first level placement
  resetLevel();

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

// ---------------------------
// Game Logic: Reset / Randomize
// ---------------------------
function resetLevel() {
  // 1. Resize canvas to fit current level
  let lvl = levels[current];
  resizeCanvas(lvl.pixelWidth(), lvl.pixelHeight());

  // 2. Find all "Floor" spots (where value is NOT 1)
  let validSpots = [];
  for (let r = 0; r < lvl.rows(); r++) {
    for (let c = 0; c < lvl.cols(); c++) {
      if (!lvl.isWall(c, r)) {
        validSpots.push({ c, r });
      }
    }
  }

  // 3. Pick a random spot for the Goal
  // random(validSpots) returns one object from the array
  let goalSpot = random(validSpots);
  goal.c = goalSpot.c;
```

```
  goal.r = goalSpot.r;

  // 4. Pick a random spot for the Player (ensure it's not the same as goal)
  let playerSpot = random(validSpots);
  // Simple check to try again if we landed on the exact same tile
  while(playerSpot.c === goal.c && playerSpot.r === goal.r && validSpots.length > 1) {
    playerSpot = random(validSpots);
  }

  // Set player pixel position to the CENTER of that tile
  player.x = playerSpot.c * TS + TS / 2;
  player.y = playerSpot.r * TS + TS / 2;
}

// ---------------------------
// Draw Loop
// ---------------------------
function draw() {
  background(240);

  let lvl = levels[current];

  // 1. Draw the map
  lvl.draw();

  // 2. Draw the Goal (Red Square on the tile)
  fill(200, 50, 50);
  rect(goal.c * TS, goal.r * TS, TS, TS);

  // 3. Handle Player Movement (Free movement)
  handleInput(lvl);

  // 4. Draw Player (Green Circle)
  fill(50, 200, 50);
  circle(player.x, player.y, player.size);

  // 5. Check Win Condition
  // Calculate center of goal in pixels
  let goalPixelX = goal.c * TS + TS / 2;
  let goalPixelY = goal.r * TS + TS / 2;

  // Check distance between player and goal center
  let d = dist(player.x, player.y, goalPixelX, goalPixelY);

  // If distance is less than half a tile, consider it "arrived"
  if (d < TS / 2) {
    levelComplete();
  }
}

// ---------------------------
// Movement & Collision
// ---------------------------
```

```
function handleInput(lvl) {
  let nextX = player.x;
  let nextY = player.y;

  if (keyIsDown(LEFT_ARROW))  nextX -= player.speed;
  if (keyIsDown(RIGHT_ARROW)) nextX += player.speed;
  if (keyIsDown(UP_ARROW))    nextY -= player.speed;
  if (keyIsDown(DOWN_ARROW))  nextY += player.speed;

  // Collision Detection:
  // Convert the *next* pixel position to a Grid Coordinate
  let col = floor(nextX / TS);
  let row = floor(nextY / TS);

  // Only move if the center point is NOT hitting a wall
  // (This is a simple point-collision check for "smooth" feel)
  if (!lvl.isWall(col, row)) {
    player.x = nextX;
    player.y = nextY;
  } else {
    // Optional: Slide along walls logic could go here.
    // For now, we just stop if the new center is invalid.

    // Try moving just X?
    col = floor(nextX / TS);
    row = floor(player.y / TS);
    if(!lvl.isWall(col, row)) player.x = nextX;

    // Try moving just Y?
    col = floor(player.x / TS);
    row = floor(nextY / TS);
    if(!lvl.isWall(col, row)) player.y = nextY;
  }
}

// ----------------------------
// Win State
// ----------------------------
function levelComplete() {
  // Advance level index
  current++;

  // Wrap around if we pass the last level
  if (current >= levels.length) {
    current = 0;
    console.log("Game Restarted!");
  }

  // Load the new level
  resetLevel();
}
```

Key Changes Made:

1. **resetLevel() Function:** This scans the JSON grid for "Floor" tiles (any tile that isn't 1). It creates a list of valid coordinates and picks two random ones: one for the Player and one for the Goal.
2. **Player Movement (handleInput):** Instead of jumping tile-by-tile, the player moves by pixels (player.speed). We calculate which grid cell the player *would* be in if they moved; if it's a wall, we block the movement.
3. **Win Condition:** We check the distance (dist) between the player and the center of the red square. If they get close enough, levelComplete() is called.
4. **Flow:** levelComplete() automatically increments the current index and calls resetLevel() immediately, creating a continuous loop.

User Prompt 2:

Make the green circle larger and must not touch the surrounding obstacles.

Gemini:

Here is the updated sketch.js.
I have increased the **player size to 26** (fitting snugly inside the 32-pixel tiles) and updated the movement logic to check the **edges of the circle** against the walls. This ensures the green circle slides along obstacles without visually overlapping or "touching" them.

sketch.js

JavaScript

```
/*
Week 4 — Example 3 Extension: Large Player & Strict Collision
Based on provided levels.json and Level class structure.
*/

// ---------------------------
// Globals
// ---------------------------

let levelsData;
let levels = [];
let current = 0;
const TS = 32; // Tile Size

// Player object
let player = {
  x: 0,
  y: 0,
  size: 26,     // INCREASED SIZE (Fits in 32px tile with 3px padding)
  speed: 3
};
```

```
// Goal object
let goal = {
  c: 0,
  r: 0
};

// ----------------------------
// Preload
// ----------------------------
function preload() {
  levelsData = loadJSON("levels.json");
}

// ----------------------------
// Level Class
// ----------------------------
class Level {
  constructor(grid, tileSize) {
    this.grid = grid;
    this.ts = tileSize;
  }

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

  // Helper to check if a specific grid cell is a wall
  isWall(c, r) {
    // Check bounds
    if (c < 0 || r < 0 || c >= this.cols() || r >= this.rows()) return true;
    return this.grid[r][c] === 1;
  }

  draw() {
    for (let r = 0; r < this.rows(); r++) {
      for (let c = 0; c < this.cols(); c++) {
        const v = this.grid[r][c];
        noStroke();
        fill(v === 1 ? color(30, 50, 60) : color(220));
        rect(c * this.ts, r * this.ts, this.ts, this.ts);
      }
    }
  }
}

// ----------------------------
// Setup
// ----------------------------
function setup() {
  levels = levelsData.levels.map((grid) => new Level(grid, TS));
  resetLevel();
```

```
  noStroke();
}

// ---------------------------
// Game Logic
// ---------------------------
function resetLevel() {
  let lvl = levels[current];
  resizeCanvas(lvl.pixelWidth(), lvl.pixelHeight());

  // Find valid floor spots
  let validSpots = [];
  for (let r = 0; r < lvl.rows(); r++) {
    for (let c = 0; c < lvl.cols(); c++) {
      if (!lvl.isWall(c, r)) {
        validSpots.push({ c, r });
      }
    }
  }

  // Randomize Goal
  let goalSpot = random(validSpots);
  goal.c = goalSpot.c;
  goal.r = goalSpot.r;

  // Randomize Player (ensure distinct from goal)
  let playerSpot = random(validSpots);
  while(playerSpot.c === goal.c && playerSpot.r === goal.r && validSpots.length > 1) {
    playerSpot = random(validSpots);
  }

  // Place player at CENTER of the tile
  player.x = playerSpot.c * TS + TS / 2;
  player.y = playerSpot.r * TS + TS / 2;
}

// ---------------------------
// Draw Loop
// ---------------------------
function draw() {
  background(240);
  let lvl = levels[current];

  // 1. Draw Level
  lvl.draw();

  // 2. Draw Goal (Red Square)
  fill(200, 50, 50);
  rect(goal.c * TS, goal.r * TS, TS, TS);

  // 3. Move Player
  handleInput(lvl);
```

```
  // 4. Draw Player (Large Green Circle)
  fill(50, 200, 50);
  circle(player.x, player.y, player.size);

  // 5. Win Check
  let goalPixelX = goal.c * TS + TS / 2;
  let goalPixelY = goal.r * TS + TS / 2;

  if (dist(player.x, player.y, goalPixelX, goalPixelY) < TS / 2) {
    levelComplete();
  }
}

// ----------------------------
// Movement & strict Collision
// ----------------------------
function handleInput(lvl) {
  // We handle X and Y separately to allow "sliding" along walls

  // -- Try moving X --
  let nextX = player.x;
  if (keyIsDown(LEFT_ARROW))  nextX -= player.speed;
  if (keyIsDown(RIGHT_ARROW)) nextX += player.speed;

  // Only update X if no collision at the new X position
  if (!checkCollision(nextX, player.y, lvl)) {
    player.x = nextX;
  }

  // -- Try moving Y --
  let nextY = player.y;
  if (keyIsDown(UP_ARROW))    nextY -= player.speed;
  if (keyIsDown(DOWN_ARROW))  nextY += player.speed;

  // Only update Y if no collision at the new Y position
  if (!checkCollision(player.x, nextY, lvl)) {
    player.y = nextY;
  }
}

/*
  Checks if the player (at position x, y) overlaps any wall.
  It checks 4 points around the player's bounding box.
*/
function checkCollision(x, y, lvl) {
  let r = player.size / 2; // Radius

  // The 4 edges/corners of the player to check
  // We add a tiny buffer (-1 or +1) to be safe, but exact radius is usually fine
  let pointsToCheck = [
    { x: x - r, y: y - r }, // Top Left
    { x: x + r, y: y - r }, // Top Right
    { x: x + r, y: y + r }, // Bottom Right
```

```
    { x: x - r, y: y + r }  // Bottom Left
  ];

  for (let p of pointsToCheck) {
    let c = floor(p.x / TS);
    let row = floor(p.y / TS);

    // If any of the 4 corners is inside a wall, return true (Collision!)
    if (lvl.isWall(c, row)) {
      return true;
    }
  }
  return false;
}

// ---------------------------
// Win State
// ---------------------------
function levelComplete() {
  current++;
  if (current >= levels.length) {
    current = 0;
  }
  resetLevel();
}
```

User Prompt 3:

Add another controls: Arrow keys and WASD keys (W-forward, A-left, S-backward, D-right).

Gemini:

Here is the updated sketch.js.
I have updated the handleInput() function to accept both **Arrow keys** and **WASD** keys (W=87, A=65, S=83, D=68). The rest of the game logic (collision, random placement, level switching) remains the same.

sketch.js

JavaScript

```
/*
Week 4 — Example 3 Extension: Dual Controls (Arrow + WASD)
Based on provided levels.json and Level class structure.
*/

// ---------------------------
// Globals
// ---------------------------
```

```
let levelsData;
let levels = [];
let current = 0;
const TS = 32; // Tile Size

// Player object
let player = {
  x: 0,
  y: 0,
  size: 26,     // Large size (fits in 32px tile with padding)
  speed: 3
};

// Goal object
let goal = {
  c: 0,
  r: 0
};

// ----------------------------
// Preload
// ----------------------------
function preload() {
  levelsData = loadJSON("levels.json");
}

// ----------------------------
// Level Class
// ----------------------------
class Level {
  constructor(grid, tileSize) {
    this.grid = grid;
    this.ts = tileSize;
  }

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

  isWall(c, r) {
    if (c < 0 || r < 0 || c >= this.cols() || r >= this.rows()) return true;
    return this.grid[r][c] === 1;
  }

  draw() {
    for (let r = 0; r < this.rows(); r++) {
      for (let c = 0; c < this.cols(); c++) {
        const v = this.grid[r][c];
        noStroke();
        fill(v === 1 ? color(30, 50, 60) : color(220));
        rect(c * this.ts, r * this.ts, this.ts, this.ts);
      }
```

```
    }
  }
}

// ---------------------------
// Setup
// ---------------------------
function setup() {
  levels = levelsData.levels.map((grid) => new Level(grid, TS));
  resetLevel();
  noStroke();
}

// ---------------------------
// Game Logic
// ---------------------------
function resetLevel() {
  let lvl = levels[current];
  resizeCanvas(lvl.pixelWidth(), lvl.pixelHeight());

  // Find valid floor spots
  let validSpots = [];
  for (let r = 0; r < lvl.rows(); r++) {
    for (let c = 0; c < lvl.cols(); c++) {
      if (!lvl.isWall(c, r)) {
        validSpots.push({ c, r });
      }
    }
  }

  // Randomize Goal
  let goalSpot = random(validSpots);
  goal.c = goalSpot.c;
  goal.r = goalSpot.r;

  // Randomize Player (ensure distinct from goal)
  let playerSpot = random(validSpots);
  while(playerSpot.c === goal.c && playerSpot.r === goal.r && validSpots.length > 1) {
    playerSpot = random(validSpots);
  }

  // Place player at CENTER of the tile
  player.x = playerSpot.c * TS + TS / 2;
  player.y = playerSpot.r * TS + TS / 2;
}

// ---------------------------
// Draw Loop
// ---------------------------
function draw() {
  background(240);
  let lvl = levels[current];
```

```
  // 1. Draw Level
  lvl.draw();

  // 2. Draw Goal (Red Square)
  fill(200, 50, 50);
  rect(goal.c * TS, goal.r * TS, TS, TS);

  // 3. Move Player
  handleInput(lvl);

  // 4. Draw Player (Large Green Circle)
  fill(50, 200, 50);
  circle(player.x, player.y, player.size);

  // 5. Win Check
  let goalPixelX = goal.c * TS + TS / 2;
  let goalPixelY = goal.r * TS + TS / 2;

  if (dist(player.x, player.y, goalPixelX, goalPixelY) < TS / 2) {
    levelComplete();
  }
}

// ----------------------------
// Movement & Collision (WASD + ARROWS)
// ----------------------------
function handleInput(lvl) {
  // We handle X and Y separately to allow "sliding" along walls

  // -- Try moving X --
  let nextX = player.x;

  // Left: Arrow Left or 'A' (65)
  if (keyIsDown(LEFT_ARROW) || keyIsDown(65))  nextX -= player.speed;

  // Right: Arrow Right or 'D' (68)
  if (keyIsDown(RIGHT_ARROW) || keyIsDown(68)) nextX += player.speed;

  // Only update X if no collision at the new X position
  if (!checkCollision(nextX, player.y, lvl)) {
    player.x = nextX;
  }

  // -- Try moving Y --
  let nextY = player.y;

  // Up: Arrow Up or 'W' (87)
  if (keyIsDown(UP_ARROW) || keyIsDown(87))    nextY -= player.speed;

  // Down: Arrow Down or 'S' (83)
  if (keyIsDown(DOWN_ARROW) || keyIsDown(83))  nextY += player.speed;

  // Only update Y if no collision at the new Y position
```

```
  if (!checkCollision(player.x, nextY, lvl)) {
    player.y = nextY;
  }
}

/*
  Checks if the player (at position x, y) overlaps any wall.
  It checks 4 points around the player's bounding box.
*/
function checkCollision(x, y, lvl) {
  let r = player.size / 2; // Radius

  // The 4 edges/corners of the player to check
  let pointsToCheck = [
    { x: x - r, y: y - r }, // Top Left
    { x: x + r, y: y - r }, // Top Right
    { x: x + r, y: y + r }, // Bottom Right
    { x: x - r, y: y + r }  // Bottom Left
  ];

  for (let p of pointsToCheck) {
    let c = floor(p.x / TS);
    let row = floor(p.y / TS);

    if (lvl.isWall(c, row)) {
      return true;
    }
  }
  return false;
}

// ---------------------------
// Win State
// ---------------------------
function levelComplete() {
  current++;
  if (current >= levels.length) {
    current = 0;
  }
  resetLevel();
}
```

User Prompt 4:

And display “Level 1” at the top of the game. After completing this level, the level increases by one, such as Level 2. Display gameplay instructions at the bottom of the game. Center the game display on the screen instead of the top left corner.

Gemini:

Here are the updated files.

To achieve the layout changes (centering) and the UI updates (text at top/bottom), I have updated **style.css** to handle the screen positioning and **sketch.js** to draw the interface bars and track the "Level" number as it increases indefinitely.

1. style.css

Update your CSS file to this. This uses Flexbox to perfectly center the game on the screen and gives it a neutral background.

CSS

```
html, body {
  margin: 0;
  padding: 0;
  height: 100vh; /* Full viewport height */
  background-color: #202020; /* Dark background to make game pop */

  /* Flexbox to center the canvas */
  display: flex;
  justify-content: center;
  align-items: center;
  font-family: sans-serif;
}

canvas {
  display: block;
  box-shadow: 0 0 20px rgba(0,0,0,0.5); /* Optional: adds a nice shadow */
}
```

2. sketch.js

Update your JS file. I have added a displayLevel variable to track progress (1, 2, 3...) separately from the map data (which loops). I also added top and bottom margins to the canvas to hold the text.

JavaScript

```
/*
Week 4 — Example 3 Extension: UI, Centering, and Infinite Level Counter
*/

// ----------------------------
// Globals
// ----------------------------

let levelsData;
let levels = [];
let current = 0;          // Which map index to load (loops 0, 1, 0, 1...)
let displayLevel = 1;     // The level number displayed to the player (1, 2, 3...)

const TS = 32;            // Tile Size
const UI_TOP = 40;        // Height of the top header bar
const UI_BOTTOM = 40;     // Height of the bottom instruction bar
```

```
let player = {
  x: 0,
  y: 0,
  size: 26,
  speed: 3
};

let goal = { c: 0, r: 0 };

// ---------------------------
// Preload
// ---------------------------
function preload() {
  levelsData = loadJSON("levels.json");
}

// ---------------------------
// Level Class
// ---------------------------
class Level {
  constructor(grid, tileSize) {
    this.grid = grid;
    this.ts = tileSize;
  }

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

  // These return the dimensions of the GAME GRID only
  pixelWidth() { return this.cols() * this.ts; }
  pixelHeight() { return this.rows() * this.ts; }

  isWall(c, r) {
    if (c < 0 || r < 0 || c >= this.cols() || r >= this.rows()) return true;
    return this.grid[r][c] === 1;
  }

  draw() {
    for (let r = 0; r < this.rows(); r++) {
      for (let c = 0; c < this.cols(); c++) {
        const v = this.grid[r][c];
        noStroke();
        fill(v === 1 ? color(30, 50, 60) : color(220));
        rect(c * this.ts, r * this.ts, this.ts, this.ts);
      }
    }
  }
}

// ---------------------------
// Setup
// ---------------------------
function setup() {
```

```
  levels = levelsData.levels.map((grid) => new Level(grid, TS));
  resetLevel();
  noStroke();
  textAlign(CENTER, CENTER); // Center text alignment for UI
}

// ---------------------------
// Game Logic
// ---------------------------
function resetLevel() {
  let lvl = levels[current];

  // Resize canvas to fit Grid + UI Bars
  resizeCanvas(lvl.pixelWidth(), lvl.pixelHeight() + UI_TOP + UI_BOTTOM);

  // Find valid floor spots
  let validSpots = [];
  for (let r = 0; r < lvl.rows(); r++) {
    for (let c = 0; c < lvl.cols(); c++) {
      if (!lvl.isWall(c, r)) {
        validSpots.push({ c, r });
      }
    }
  }

  // Randomize Goal
  let goalSpot = random(validSpots);
  goal.c = goalSpot.c;
  goal.r = goalSpot.r;

  // Randomize Player
  let playerSpot = random(validSpots);
  while(playerSpot.c === goal.c && playerSpot.r === goal.r && validSpots.length > 1) {
    playerSpot = random(validSpots);
  }

  player.x = playerSpot.c * TS + TS / 2;
  player.y = playerSpot.r * TS + TS / 2;
}

// ---------------------------
// Draw Loop
// ---------------------------
function draw() {
  // 1. Draw Background (The UI bars color)
  background(255);

  // 2. Draw Top Header (Level Counter)
  fill(30); // Dark Gray bar
  rect(0, 0, width, UI_TOP);

  fill(255); // White text
  textSize(18);
```

```
  text("LEVEL " + displayLevel, width / 2, UI_TOP / 2);

  // 3. Draw Game Area
  // We shift the coordinate system down so (0,0) is below the header
  push();
  translate(0, UI_TOP);

    let lvl = levels[current];
    lvl.draw();

    // Draw Goal
    fill(200, 50, 50);
    rect(goal.c * TS, goal.r * TS, TS, TS);

    // Update & Draw Player
    handleInput(lvl);
    fill(50, 200, 50);
    circle(player.x, player.y, player.size);

    // Win Check
    let goalPixelX = goal.c * TS + TS / 2;
    let goalPixelY = goal.r * TS + TS / 2;
    if (dist(player.x, player.y, goalPixelX, goalPixelY) < TS / 2) {
      levelComplete();
    }

  pop(); // Restore coordinates for the footer

  // 4. Draw Bottom Footer (Instructions)
  fill(30); // Dark Gray bar
  rect(0, height - UI_BOTTOM, width, UI_BOTTOM);

  fill(200); // Light Gray text
  textSize(12);
  text("Controls: WASD or Arrow Keys", width / 2, height - UI_BOTTOM / 2);
}

// ---------------------------
// Movement & Collision
// ---------------------------
function handleInput(lvl) {
  let nextX = player.x;

  // X Axis
  if (keyIsDown(LEFT_ARROW) || keyIsDown(65))  nextX -= player.speed;
  if (keyIsDown(RIGHT_ARROW) || keyIsDown(68)) nextX += player.speed;

  if (!checkCollision(nextX, player.y, lvl)) {
    player.x = nextX;
  }

  // Y Axis
  let nextY = player.y;
```

```
  if (keyIsDown(UP_ARROW) || keyIsDown(87))    nextY -= player.speed;
  if (keyIsDown(DOWN_ARROW) || keyIsDown(83))  nextY += player.speed;

  if (!checkCollision(player.x, nextY, lvl)) {
    player.y = nextY;
  }
}

function checkCollision(x, y, lvl) {
  let r = player.size / 2;
  let pointsToCheck = [
    { x: x - r, y: y - r },
    { x: x + r, y: y - r },
    { x: x + r, y: y + r },
    { x: x - r, y: y + r }
  ];

  for (let p of pointsToCheck) {
    let c = floor(p.x / TS);
    let row = floor(p.y / TS);
    if (lvl.isWall(c, row)) return true;
  }
  return false;
}

// ---------------------------
// Win State
// ---------------------------
function levelComplete() {
  current++;
  displayLevel++; // Always increase the displayed number (Level 1 -> 2 -> 3)

  // Wrap the array index so we don't crash
  if (current >= levels.length) {
    current = 0;
  }

  resetLevel();
}
```

User Prompt 5:

The green circle sometimes get stuck and fail to move, possibly due to its edge touching an obstacle. please solve this issue.

Gemini:

This is a very common issue in grid-based movement. It happens because the "mathematical" square checking for collisions is exactly the same size as your player. If

your player is touching a wall on the **left**, and you try to move **up**, the computer checks your top-left corner, sees it is technically touching the wall, and blocks the movement. To fix this, we make the **collision hitbox slightly smaller than the visual circle**. This creates a tiny "buffer zone" that allows the player to slide along walls without getting snagged on the edges.

Here is the fixed sketch.js.

sketch.js

JavaScript

```
/*
Week 4 — Example 3 Extension: Smoother Movement & UI
*/

// ----------------------------
// Globals
// ----------------------------

let levelsData;
let levels = [];
let current = 0;
let displayLevel = 1;

const TS = 32;           // Tile Size
const UI_TOP = 40;       // Top bar height
const UI_BOTTOM = 40;    // Bottom bar height

let player = {
  x: 0,
  y: 0,
  size: 26,    // Visual size
  speed: 3
};

let goal = { c: 0, r: 0 };

// ----------------------------
// Preload
// ----------------------------
function preload() {
  levelsData = loadJSON("levels.json");
}

// ----------------------------
// Level Class
// ----------------------------
class Level {
  constructor(grid, tileSize) {
    this.grid = grid;
    this.ts = tileSize;
  }

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

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

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

  isWall(c, r) {
    if (c < 0 || r < 0 || c >= this.cols() || r >= this.rows()) return true;
    return this.grid[r][c] === 1;
  }

  draw() {
    for (let r = 0; r < this.rows(); r++) {
      for (let c = 0; c < this.cols(); c++) {
        const v = this.grid[r][c];
        noStroke();
        fill(v === 1 ? color(30, 50, 60) : color(220));
        rect(c * this.ts, r * this.ts, this.ts, this.ts);
      }
    }
  }
}

// ---------------------------
// Setup
// ---------------------------
function setup() {
  levels = levelsData.levels.map((grid) => new Level(grid, TS));
  resetLevel();
  noStroke();
  textAlign(CENTER, CENTER);
}

// ---------------------------
// Game Logic
// ---------------------------
function resetLevel() {
  let lvl = levels[current];

  resizeCanvas(lvl.pixelWidth(), lvl.pixelHeight() + UI_TOP + UI_BOTTOM);

  let validSpots = [];
  for (let r = 0; r < lvl.rows(); r++) {
    for (let c = 0; c < lvl.cols(); c++) {
      if (!lvl.isWall(c, r)) {
        validSpots.push({ c, r });
      }
    }
  }

  let goalSpot = random(validSpots);
  goal.c = goalSpot.c;
  goal.r = goalSpot.r;
```

```
  let playerSpot = random(validSpots);
  while(playerSpot.c === goal.c && playerSpot.r === goal.r && validSpots.length > 1) {
    playerSpot = random(validSpots);
  }

  player.x = playerSpot.c * TS + TS / 2;
  player.y = playerSpot.r * TS + TS / 2;
}

// ---------------------------
// Draw Loop
// ---------------------------
function draw() {
  background(255);

  // 1. Top Header
  fill(30);
  rect(0, 0, width, UI_TOP);
  fill(255);
  textSize(18);
  text("LEVEL " + displayLevel, width / 2, UI_TOP / 2);

  // 2. Game Area
  push();
  translate(0, UI_TOP);

    let lvl = levels[current];
    lvl.draw();

    // Goal
    fill(200, 50, 50);
    rect(goal.c * TS, goal.r * TS, TS, TS);

    // Player
    handleInput(lvl);
    fill(50, 200, 50);
    circle(player.x, player.y, player.size);

    // Win Check
    let goalPixelX = goal.c * TS + TS / 2;
    let goalPixelY = goal.r * TS + TS / 2;
    if (dist(player.x, player.y, goalPixelX, goalPixelY) < TS / 2) {
      levelComplete();
    }

  pop();

  // 3. Bottom Footer
  fill(30);
  rect(0, height - UI_BOTTOM, width, UI_BOTTOM);
  fill(200);
  textSize(12);
  text("Controls: WASD or Arrow Keys", width / 2, height - UI_BOTTOM / 2);
```

```
}

// ----------------------------
// Movement & Collision
// ----------------------------
function handleInput(lvl) {
  // Move X
  let nextX = player.x;
  if (keyIsDown(LEFT_ARROW) || keyIsDown(65))  nextX -= player.speed;
  if (keyIsDown(RIGHT_ARROW) || keyIsDown(68)) nextX += player.speed;

  if (!checkCollision(nextX, player.y, lvl)) {
    player.x = nextX;
  }

  // Move Y
  let nextY = player.y;
  if (keyIsDown(UP_ARROW) || keyIsDown(87))    nextY -= player.speed;
  if (keyIsDown(DOWN_ARROW) || keyIsDown(83))  nextY += player.speed;

  if (!checkCollision(player.x, nextY, lvl)) {
    player.y = nextY;
  }
}

function checkCollision(x, y, lvl) {
  // COLLISION FIX:
  // We subtract a small "padding" from the radius for collision checks only.
  // This makes the physics body slightly smaller than the visual circle.
  // It ensures that if we are visually "touching" a wall, we aren't
  // mathematically "inside" it, allowing us to slide freely.

  const hitBoxPadding = 3;
  let r = (player.size / 2) - hitBoxPadding;

  let pointsToCheck = [
    { x: x - r, y: y - r },
    { x: x + r, y: y - r },
    { x: x + r, y: y + r },
    { x: x - r, y: y + r }
  ];

  for (let p of pointsToCheck) {
    let c = floor(p.x / TS);
    let row = floor(p.y / TS);
    if (lvl.isWall(c, row)) return true;
  }
  return false;
}

// ----------------------------
// Win State
// ----------------------------
```

```
function levelComplete() {
  current++;
  displayLevel++;

  if (current >= levels.length) {
    current = 0;
  }

  resetLevel();
}
```

User Prompt 6:

Here is my side quest requirements:

Generate a level using arrays or JSON data; use loops to dynamically place tiles,obstacles, or words. Bonus: Add a second level that loads automatically when the first is finished.

Does the current code meet all requirements?

Gemini:

Yes, the current code **fully meets and exceeds** all the requirements for your side quest.
Here is the breakdown of how your code satisfies each specific point:

1. Generate a level using arrays or JSON data

- **Status:** ✅ **Met**
- **Evidence:** You are using `loadJSON("levels.json")` in the `preload()` function. The data inside `levels.json` is an array of arrays (a grid), which determines your level layout.

2. Use loops to dynamically place tiles, obstacles, or words

- **Status:** ✅ **Met**
- **Evidence:** Inside the `Level` class, the `draw()` method uses nested `for` loops (`for (let r...` and `for (let c...`) to iterate through your JSON arrays. It dynamically draws a dark wall tile if the value is `1` and a floor tile otherwise.

3. Bonus: Add a second level that loads automatically

- **Status:** ✅ **Met**
- **Evidence:** The `levelComplete()` function increments the `current` level index (`current++`) and immediately calls `resetLevel()`. This automatically loads the next grid from your JSON file without restarting the program.

You are good to go! The code is functional, modular, and fulfills the assignment criteria perfectly.