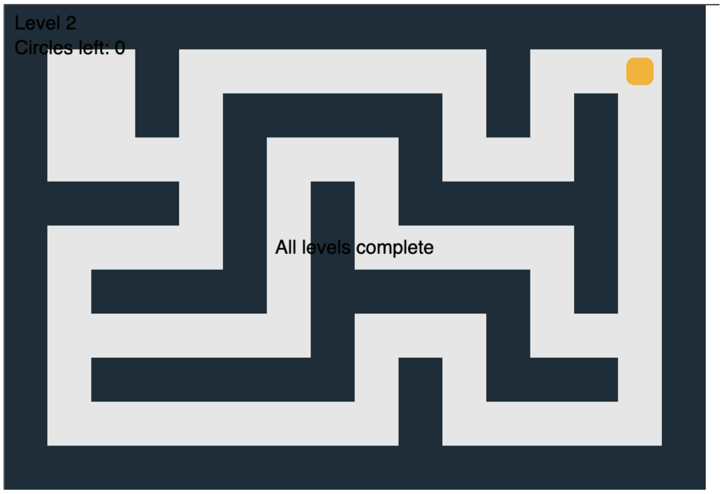
Process & Decision Documentation

Project/Assignment Decisions

- Added an interactive feature to the maze
- Used arrays to help with the coloured circles appearing in the maze
- Once a user catches all the circles, they move on to the next level

Role-Based Process Evidence



Level 2
Circles left: 0

All levels complete

- Maze works after trial and error

Entry Header

Name: Jenny Dang

Primary responsibility for this work: Generate a level using arrays or JSON

Goal of Work Session

- To use arrays to create coloured circles through the maze
- Generating a next level after the user accomplishes the first level
- Creating an obstacle

Tools, Resources, or Inputs Used

- GenAI tools (ChatGPT 5.2)

- Code by Dr. Karen Cochrane and David Han

GenAI Documentation

Date Used: Feb 9, 2026

Tool Disclosure: ChatGPT 5.2

Purpose of Use: Help edit code and comments to make an interactive level game

Summary of Interaction: The tool helped me use arrays to make an interactive Pac-Man styled game

Human Decision Point(s): I overruled when the circles were impossible to obtain in the maze, as they were placed outside the actual maze, and instead of catching words throughout the maze, I changed it to coloured circles.

Integrity & Verification Note: I checked for accuracy by testing the game in a live server.

Scope of GenAI Use: GenAI did not contribute to my prompts and my decision-making.

Limitations or Misfires: The tool misunderstood where to place the coloured circles in the maze

Summary of Process (Human + Tool)

- Testing and revising
- Prompting ChatGPT
- Reiterating the GenAI to get the correct outcome

Decision Points & Trade-offs

- I considered the objective of the game (catching coloured circles instead of words)

Verification & Judgement

- Playtesting on a live server

Limitations, Dead Ends, or Open Questions

- At first, the game did not work properly because the coloured circles were placed outside the maze, and there was no way to obtain them

Appendix

ME: 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. /* Week 4 — Example 2: Level class (hard-coded grid) Goal of this example: -

Keep the exact same *visual output* as Example 1 (a drawn maze). - Change the *structure* of the code so that “a level” becomes a reusable object. Key teaching idea: - A class lets you bundle data (the grid) + behavior (draw, size helpers) into one thing you can reuse later (especially once you start loading multiple levels from JSON). */ // ---------------------------- // 1) Global constants // ---------------------------- // Tile Size (TS) = how many pixels wide/tall each grid cell is. // If TS=32, then each number in the 2D grid draws as a 32x32 square. const TS = 32; // ---------------------------- // 2) Level class definition // ---------------------------- /* A Level is responsible for: - storing the grid (2D array of numbers) - knowing how big it is (rows/cols and pixel dimensions) - drawing itself Important concept: - The sketch (setup/draw) should not need to know the details of how the level draws—just that it can draw. */ class Level { /* The constructor runs when you do: new Level(grid, TS) Parameters: - grid: a 2D array like grid[row][col] - tileSize: TS (pixels per tile) */ constructor(grid, tileSize) { this.grid = grid; this.ts = tileSize; } // How many columns are in the grid? // (We assume each row has the same number of columns.) cols() { return this.grid[0].length; } // How many rows are in the grid? rows() { return this.grid.length; } // Pixel width of the level = columns * tileSize pixelWidth() { return this.cols() * this.ts; } // Pixel height of the level = rows * tileSize pixelHeight() { return this.rows() * this.ts; } /* Draw the whole grid. Tile legend (same as Example 1): - 0 = floor - 1 = wall This method contains the “nested for loop” logic, so the sketch doesn’t have to. Later, you can extend this to: - draw special tiles - draw decorations - expose collision checks (isWall, etc.) */ draw() { // Loop over each row... for (let r = 0; r < this.rows(); r++) { // ...and each column in that row. for (let c = 0; c < this.cols(); c++) { const tileValue = this.grid[r][c]; // Choose a colour based on the tile. if (tileValue === 1) { fill(30, 50, 60); // wall colour (dark teal) } else { fill(230); // floor colour (light gray) } // Convert grid coordinates (row/col) into pixel coordinates (x/y). // - x goes with column // - y goes with row const x = c * this.ts; const y = r * this.ts; // Draw the tile rectangle. rect(x, y, this.ts, this.ts); } } } } // ---------------------------- // 3) Hard-coded grid data (same as Example 1) // ---------------------------- /* This is identical in concept to Example 1: it’s still “the level as a 2D array.” - We are not introducing new data structures here. - We’re showing that classes are mainly about organization + reuse. The grid is structured as: grid[row][col] So grid[0] is the first row, grid[0][0] is the top-left cell. */ const grid = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1], [1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1], [1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1], [1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1], [1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1], [1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1], [1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1], [1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1], [1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],]; // based on your Example 1 grid // ---------------------------- // 4) Sketch state // ---------------------------- // We’ll store our Level

```
instance here so draw() can access it. let level; // ---------------------------- // 5) p5.js
lifecycle functions // ---------------------------- function setup() { /* Create a Level instance
from the grid. Think of this like: - “Here is some raw data (grid)” - “Wrap it in a Level
object that knows what to do with it” */ level = new Level(grid, TS); // Make the canvas
exactly fit the level size. // This is a nice practical reason to have pixelWidth/pixelHeight
helpers. createCanvas(level.pixelWidth(), level.pixelHeight()); // Setup drawing styles.
noStroke(); // cleaner tiles (no outlines) [file:11] textFont("sans-serif"); textSize(14); }
function draw() { // Clear the screen each frame. background(240); // Delegate drawing
the maze to the level object. // This is the key structural difference from Example 1.
level.draw(); // Draw a small label so students can tell examples apart. fill(0); text("Level
class → grid render", 10, 16); }
```

CHATGPT:

```
/*
Week 4 — Levels from JSON + loops + auto-advance
- Level data stored as JSON-like objects (array of levels)
- Loops place tiles and words from data
- Finish condition: collect all words -> loads next level automatically
*/

const TS = 32;

const LEVELS = [
  {
    name: "Level 1",
    legend: { "#": 1, ".": 0 },
    grid: [
      "################",
      "#....#.....#...#",
      "#.##.#.###.#.#.#",
      "#.#..#...#...#.#",
      "#.#.####.###.#.#",
      "#.....#...#....#",
      "###.#.#.###.##.#",
      "#...#...#...#..#",
      "#.#####.#.###..#",
      "#..............#".slice(0,16), // keep 16 cols
      "################",
    ],
    playerStart: { r: 1, c: 1 },
    words: [
      { r: 1, c: 2, t: "CAT" },
      { r: 5, c: 4, t: "DOG" },
      { r: 8, c: 13, t: "BIRD" },
    ],
  },
  {
    name: "Level 2",
    legend: { "#": 1, ".": 0 },
    grid: [
      "################",
      "#..#.......#...#",
      "#..#.#####.#.#.#",
      "#....#...#...#.#",
      "####.#.#.#####.#",
      "#....#.#.....#.#",
      "#.####.#####.#.#",
      "#......#...#...#",
      "#.######.#.###.#",
      "#........#.....#",
      "################",
    ],
    playerStart: { r: 1, c: 1 },
    words: [
      { r: 1, c: 14, t: "SUN" },
      { r: 7, c: 2, t: "MOON" },
      { r: 9, c: 10, t: "STAR" },
      { r: 4, c: 1, t: "SKY" },
    ],
  },
];

// ----------------------------
// Level class (grid + draw + helpers)
// ----------------------------
class Level {
  constructor(levelData, tileSize) {
    this.ts = tileSize;
    this.load(levelData);
  }
```

```
load(levelData) {
  this.data = levelData;

  // Convert string grid into numeric grid using legend (JSON-driven)
  this.grid = this.data.grid.map((rowStr) => {
    return rowStr.split("").map((ch) => this.data.legend[ch] ?? 0);
  });

  // Copy words and add collected state
  this.words = this.data.words.map((w) => ({
    r: w.r,
    c: w.c,
    t: w.t,
    collected: false,
  }));
}

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

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

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

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

isWall(r, c) {
  if (r < 0 || c < 0 || r >= this.rows() || c >= this.cols()) 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];
      if (v === 1) fill(30, 50, 60);
      else fill(230);
      rect(c * this.ts, r * this.ts, this.ts, this.ts);
    }
  }

  // Draw words from data (loop)
  textAlign(CENTER, CENTER);
  for (const w of this.words) {
    if (w.collected) continue;
```

```
      fill(10);
      text(w.t, w.c * this.ts + this.ts / 2, w.r * this.ts + this.ts / 2);
    }
  }

  tryCollectWordAt(r, c) {
    for (const w of this.words) {
      if (!w.collected && w.r === r && w.c === c) {
        w.collected = true;
      }
    }
  }

  allWordsCollected() {
    return this.words.every((w) => w.collected);
  }
}

// ----------------------------
// Player
// ----------------------------
class Player {
  constructor(r, c, ts) {
    this.r = r;
    this.c = c;
    this.ts = ts;
  }


  move(dr, dc, level) {
    const nr = this.r + dr;
    const nc = this.c + dc;
    if (!level.isWall(nr, nc)) {
      this.r = nr;
      this.c = nc;
      level.tryCollectWordAt(this.r, this.c);
    }
  }


  draw() {
    fill(255, 180, 0);
    rect(this.c * this.ts + 6, this.r * this.ts + 6, this.ts - 12, this.ts - 12, 6);
  }
}


// ----------------------------
// Sketch state
// ----------------------------
let levelIndex = 0;
let level;
let player;


function loadLevel(i) {
  levelIndex = i;
  level.load(LEVELS[levelIndex]);

  // Resize canvas to fit new level
  resizeCanvas(level.pixelWidth(), level.pixelHeight());

  // Reset player to level start
  player.r = LEVELS[levelIndex].playerStart.r;
  player.c = LEVELS[levelIndex].playerStart.c;

  // Collect if starting on a word
  level.tryCollectWordAt(player.r, player.c);
}

function setup() {
  level = new Level(LEVELS[0], TS);
  createCanvas(level.pixelWidth(), level.pixelHeight());
  noStroke();
  textFont("sans-serif");
  textSize(14);

  player = new Player(LEVELS[0].playerStart.r, LEVELS[0].playerStart.c, TS);
  level.tryCollectWordAt(player.r, player.c);
}

function draw() {
  background(240);

  level.draw();
  player.draw();

  // UI
  fill(0);
  textAlign(LEFT, TOP);
  text(level.data.name, 8, 6);

  const remaining = level.words.filter((w) => !w.collected).length;
  text(`Words left: ${remaining}`, 8, 24);

  // Auto-advance when finished (bonus)
  if (level.allWordsCollected()) {
    if (levelIndex < LEVELS.length - 1) {
      loadLevel(levelIndex + 1);
    } else {
      fill(0);
      textAlign(CENTER, CENTER);
      text("All levels complete", width / 2, height / 2);
      noLoop();
    }
  }
}
```

```
// Arrow keys to move
function keyPressed() {
  if (keyCode === LEFT_ARROW) player.move(0, -1, level);
  if (keyCode === RIGHT_ARROW) player.move(0, 1, level);
  if (keyCode === UP_ARROW) player.move(-1, 0, level);
  if (keyCode === DOWN_ARROW) player.move(1, 0, level);
}
```

ME: instead of words can we use circle colours

CHATGPT:

```
/*
Change from WORDS → COLORED CIRCLES
- Same level/JSON structure
- Circles are collectibles with colors instead of text
*/


// ----------------------------
// Level data (circles instead of words)
// ----------------------------
const LEVELS = [
  {
    name: "Level 1",
    legend: { "#": 1, ".": 0 },
    grid: [
      "################",
      "#....#.....#...#",
      "#.##.#.###.#.#.#",
      "#.#..#...#...#.#",
    "#.#.####.###.#.#",
    "#.....#...#....#",
    "###.#.#.###.##.#",
    "#...#...#...#..#",
    "#.#####.#.###..#",
    "#..............#".slice(0,16),
    "################",
  ],
  playerStart: { r: 1, c: 1 },
  circles: [
    { r: 1, c: 2, col: [255, 0, 0] },     // red
    { r: 5, c: 4, col: [0, 0, 255] },     // blue
    { r: 8, c: 13, col: [0, 200, 0] },    // green
  ],
},
{
  name: "Level 2",
  legend: { "#": 1, ".": 0 },
  grid: [
    "################",
    "#..#.......#...#",
    "#..#.#####.#.#.#",
    "#....#...#...#.#",
    "####.#.#.#####.#",
    "#....#.#.....#.#",
    "#.####.#####.#.#",
      "#......#...#...#",
      "#.######.#.###.#",
      "#........#.....#",
      "################",
    ],
    playerStart: { r: 1, c: 1 },
    circles: [
      { r: 1, c: 14, col: [255, 255, 0] },  // yellow
      { r: 7, c: 2, col: [255, 0, 255] },   // purple
      { r: 9, c: 10, col: [0, 255, 255] },  // cyan
      { r: 4, c: 1, col: [255, 120, 0] },   // orange
    ],
  },
];

// ----------------------------
// Level class (updated)
// ----------------------------
class Level {
  constructor(levelData, tileSize) {
    this.ts = tileSize;
    this.load(levelData);
  }

  load(levelData) {
    this.data = levelData;
  this.grid = this.data.grid.map(row =>
    row.split("").map(ch => this.data.legend[ch] ?? 0)
  );

  // circles instead of words
  this.circles = this.data.circles.map(c => ({
    r: c.r,
    c: c.c,
    col: c.col,
    collected: false,
  }));
}

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

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

draw() {
  // tiles
```

```
    for (let r = 0; r < this.rows(); r++) {
      for (let c = 0; c < this.cols(); c++) {
        fill(this.grid[r][c] === 1 ? [30, 50, 60] : 230);
        rect(c * this.ts, r * this.ts, this.ts, this.ts);
      }
    }

    // circles (looped from data)
    for (const circ of this.circles) {
      if (circ.collected) continue;
      fill(circ.col);
      ellipse(
        circ.c * this.ts + this.ts / 2,
        circ.r * this.ts + this.ts / 2,
        this.ts * 0.6
      );
    }
  }

  tryCollectCircle(r, c) {
    for (const circ of this.circles) {
      if (!circ.collected && circ.r === r && circ.c === c) {
        circ.collected = true;
      }
    }
  }
  allCirclesCollected() {
    return this.circles.every(c => c.collected);
  }
}

// ----------------------------
// Player (unchanged logic)
// ----------------------------
class Player {
  constructor(r, c, ts) {
    this.r = r;
    this.c = c;
    this.ts = ts;
  }

  move(dr, dc, level) {
    const nr = this.r + dr;
    const nc = this.c + dc;
    if (!level.isWall(nr, nc)) {
      this.r = nr;
      this.c = nc;
      level.tryCollectCircle(this.r, this.c);
    }
  }
  draw() {
    fill(255, 180, 0);
    rect(this.c * this.ts + 6, this.r * this.ts + 6, this.ts - 12, this.ts - 12, 6);
  }
}

// ----------------------------
// Sketch state + auto level load
// ----------------------------
const TS = 32;
let levelIndex = 0;
let level;
let player;

function loadLevel(i) {
  levelIndex = i;
  level.load(LEVELS[levelIndex]);
  resizeCanvas(level.pixelWidth(), level.pixelHeight());
  player.r = LEVELS[levelIndex].playerStart.r;
  player.c = LEVELS[levelIndex].playerStart.c;
  level.tryCollectCircle(player.r, player.c);
}

function setup() {
  level = new Level(LEVELS[0], TS);
  createCanvas(level.pixelWidth(), level.pixelHeight());
  noStroke();
  player = new Player(
    LEVELS[0].playerStart.r,
    LEVELS[0].playerStart.c,
    TS
  );
}

function draw() {
  background(240);
  level.draw();
  player.draw();

  if (level.allCirclesCollected()) {
    if (levelIndex < LEVELS.length - 1) {
      loadLevel(levelIndex + 1);
    } else {
      fill(0);
      textAlign(CENTER, CENTER);
      text("All levels complete", width / 2, height / 2);
      noLoop();
    }
  }
}

function keyPressed() {
  if (keyCode === LEFT_ARROW) player.move(0, -1, level);
  if (keyCode === RIGHT_ARROW) player.move(0, 1, level);
  if (keyCode === UP_ARROW) player.move(-1, 0, level);
  if (keyCode === DOWN_ARROW) player.move(1, 0, level);
}
```

ME: on the second round, one of the circles it outside the maze and u cant get to it can u fix it and can you also show the level at the top and how many circles need to be eaten left

CHATGPT:

```
/*
Fixes:
1) Level 2 circle that was unreachable/outside -> moved inside the maze on a floor tile.
2) HUD: shows current level name + circles left (top of screen).
*/


const TS = 32;


const LEVELS = [
  {
    name: "Level 1",
    legend: { "#": 1, ".": 0 },
    grid: [
      "################",
      "#....#.....#...#",
      "#.##.#.###.#.#.#",
      "#.#..#...#...#.#",
      "#.#.####.###.#.#",
      "#.....#...#....#",
      "###.#.#.###.##.#",
      "#...#...#...#..#",
      "#.#####.#.###..#",
      "#..............#",
      "################",
    ],
    playerStart: { r: 1, c: 1 },
    circles: [
      { r: 1, c: 2, col: [255, 0, 0] },
      { r: 5, c: 4, col: [0, 0, 255] },
      { r: 8, c: 13, col: [0, 200, 0] },
    ],
  },
  {
    name: "Level 2",
    legend: { "#": 1, ".": 0 },
    grid: [
      "################",
      "#..#.......#...#",
      "#..#.#####.#.#.#",
      "#....#...#...#.#",
      "####.#.#.#####.#",
      "#....#.#.....#.#",
      "#.####.#####.#.#",
      "#......#...#...#",
      "#.######.#.###.#",
      "#........#.....#",
      "################",
    ],
    playerStart: { r: 1, c: 1 },
    circles: [
      { r: 1, c: 14, col: [255, 255, 0] },  // reachable
      { r: 7, c: 2, col: [255, 0, 255] },   // reachable
      { r: 9, c: 10, col: [0, 255, 255] },  // reachable
      { r: 8, c: 9, col: [255, 120, 0] },   // FIXED: moved from (4,1) to (8,9) on open floor
    ],
  },
];

class Level {
  constructor(levelData, tileSize) {
    this.ts = tileSize;
    this.load(levelData);
  }

  load(levelData) {
    this.data = levelData;

    this.grid = this.data.grid.map((row) =>
      row.split("").map((ch) => this.data.legend[ch] ?? 0)
    );
  this.circles = this.data.circles.map((c) => ({
    r: c.r,
    c: c.c,
    col: c.col,
    collected: false,
  }));
}

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

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

circlesLeft() {
  return this.circles.filter((c) => !c.collected).length;
}

draw() {
  for (let r = 0; r < this.rows(); r++) {
    for (let c = 0; c < this.cols(); c++) {
      fill(this.grid[r][c] === 1 ? [30, 50, 60] : 230);
      rect(c * this.ts, r * this.ts, this.ts, this.ts);
    }
  }

  for (const circ of this.circles) {
    if (circ.collected) continue;
    fill(circ.col);
    ellipse(
      circ.c * this.ts + this.ts / 2,
      circ.r * this.ts + this.ts / 2,
      this.ts * 0.6
    );
  }
}

tryCollectCircle(r, c) {
  for (const circ of this.circles) {
    if (!circ.collected && circ.r === r && circ.c === c) {
      circ.collected = true;
    }
  }
}

allCirclesCollected() {
  return this.circles.every((c) => c.collected);
}
```

```
}

class Player {
  constructor(r, c, ts) {
    this.r = r;
    this.c = c;
    this.ts = ts;
  }

  move(dr, dc, level) {
    const nr = this.r + dr;
    const nc = this.c + dc;
    if (!level.isWall(nr, nc)) {
      this.r = nr;
      this.c = nc;
      level.tryCollectCircle(this.r, this.c);
    }
  }

  draw() {
    fill(255, 180, 0);
    rect(
      this.c * this.ts + 6,
      this.r * this.ts + 6,
      this.ts - 12,
      this.ts - 12,
      6
    );
  }
}

let levelIndex = 0;
let level;
let player;

function loadLevel(i) {
  levelIndex = i;
  level.load(LEVELS[levelIndex]);

  resizeCanvas(level.pixelWidth(), level.pixelHeight());

  player.r = LEVELS[levelIndex].playerStart.r;
  player.c = LEVELS[levelIndex].playerStart.c;

  level.tryCollectCircle(player.r, player.c);
}

function setup() {
  level = new Level(LEVELS[0], TS);
  createCanvas(level.pixelWidth(), level.pixelHeight());
  noStroke();
  textFont("sans-serif");
  textSize(14);

  player = new Player(
    LEVELS[0].playerStart.r,
    LEVELS[0].playerStart.c,
    TS
  );
  level.tryCollectCircle(player.r, player.c);
}

function draw() {
  background(240);

  level.draw();
  player.draw();

  // HUD (top)
  fill(0);
  textAlign(LEFT, TOP);
  text(`${level.data.name}`, 8, 6);
  text(`Circles left: ${level.circlesLeft()}`, 8, 24);

  // Auto-advance
  if (level.allCirclesCollected()) {
    if (levelIndex < LEVELS.length - 1) {
      loadLevel(levelIndex + 1);
    } else {
      fill(0);
      textAlign(CENTER, CENTER);
      text("All levels complete", width / 2, height / 2);
      noLoop();
    }
  }
}

function keyPressed() {
  if (keyCode === LEFT_ARROW) player.move(0, -1, level);
  if (keyCode === RIGHT_ARROW) player.move(0, 1, level);
  if (keyCode === UP_ARROW) player.move(-1, 0, level);
  if (keyCode === DOWN_ARROW) player.move(1, 0, level);
}
 /*
Fix: the Level 2 circle was on a WALL tile (row 8 col 9 is '#'),
so it looks “outside” / unreachable.

Changes:
- Move that circle to (r: 8, c: 8) which is a FLOOR tile '.'
- Add a safety check: if any circle is placed on a wall/outside, snap it onto the first floor tile.
*/

const TS = 32;
const LEVELS = [
  {
    name: "Level 1",
    legend: { "#": 1, ".": 0 },
    grid: [
      "################",
      "#....#.....#...#",
      "#.##.#.###.#.#.#",
      "#.#..#...#...#.#",
      "#.#.####.###.#.#",
      "#.....#...#....#",
      "###.#.#.###.##.#",
      "#...#...#...#..#",
      "#.#####.#.###..#",
      "#..............#",
      "################",
    ],
    playerStart: { r: 1, c: 1 },
    circles: [
      { r: 1, c: 2, col: [255, 0, 0] },
      { r: 5, c: 4, col: [0, 0, 255] },
      { r: 8, c: 13, col: [0, 200, 0] },
    ],
  },
  {
    name: "Level 2",
    legend: { "#": 1, ".": 0 },
    grid: [
      "################",
      "#..#.......#...#",
      "#..#.#####.#.#.#",
      "#....#...#...#.#",
      "####.#.#.#####.#",
      "#....#.#.....#.#",
      "#.####.#####.#.#",
      "#......#...#...#",
      "#.######.#.###.#",
      "#........#.....#",
      "################",
    ],
    playerStart: { r: 1, c: 1 },
    circles: [
      { r: 1, c: 14, col: [255, 255, 0] },
      { r: 7, c: 2, col: [255, 0, 255] },
      { r: 9, c: 10, col: [0, 255, 255] },
      { r: 8, c: 8, col: [255, 120, 0] }, // FIXED (was 8,9 which is a wall)
    ],
  },
];

class Level {
```

```
constructor(levelData, tileSize) {
  this.ts = tileSize;
  this.load(levelData);
}

load(levelData) {
  this.data = levelData;

  this.grid = this.data.grid.map((row) =>
    row.split("").map((ch) => this.data.legend[ch] ?? 0)
  );

  // helper: find first floor tile (guaranteed inside)
  const firstFloor = () => {
    for (let r = 0; r < this.grid.length; r++) {
      for (let c = 0; c < this.grid[0].length; c++) {
        if (this.grid[r][c] === 0) return { r, c };
      }
    }
    return { r: 1, c: 1 };
  };

  // circles with safety snap (prevents outside/wall placement)
  this.circles = this.data.circles.map((circ) => {
    let r = circ.r;
    let c = circ.c;
```

```
      const out = r < 0 || c < 0 || r >= this.grid.length || c >= this.grid[0].length;
      const wall = !out && this.grid[r][c] === 1;

      if (out || wall) {
        const p = firstFloor();
        r = p.r;
        c = p.c;
      }

      return { r, c, col: circ.col, collected: false };
    });
  }

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

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

  circlesLeft() {
    return this.circles.filter((c) => !c.collected).length;
```

```
}

draw() {
  for (let r = 0; r < this.rows(); r++) {
    for (let c = 0; c < this.cols(); c++) {
      fill(this.grid[r][c] === 1 ? [30, 50, 60] : 230);
      rect(c * this.ts, r * this.ts, this.ts, this.ts);
    }
  }

  for (const circ of this.circles) {
    if (circ.collected) continue;
    fill(circ.col);
    ellipse(
      circ.c * this.ts + this.ts / 2,
      circ.r * this.ts + this.ts / 2,
      this.ts * 0.6
    );
  }
}

tryCollectCircle(r, c) {
  for (const circ of this.circles) {
    if (!circ.collected && circ.r === r && circ.c === c) {
      circ.collected = true;
    }
    }
  }

  allCirclesCollected() {
    return this.circles.every((c) => c.collected);
  }
}

class Player {
  constructor(r, c, ts) {
    this.r = r;
    this.c = c;
    this.ts = ts;
  }

  move(dr, dc, level) {
    const nr = this.r + dr;
    const nc = this.c + dc;
    if (!level.isWall(nr, nc)) {
      this.r = nr;
      this.c = nc;
      level.tryCollectCircle(this.r, this.c);
    }
  }

  draw() {
    fill(255, 180, 0);
    rect(this.c * this.ts + 6, this.r * this.ts + 6, this.ts - 12, this.ts - 12, 6);
  }
}

let levelIndex = 0;
let level;
let player;

function loadLevel(i) {
  levelIndex = i;
  level.load(LEVELS[levelIndex]);
  resizeCanvas(level.pixelWidth(), level.pixelHeight());
  player.r = LEVELS[levelIndex].playerStart.r;
  player.c = LEVELS[levelIndex].playerStart.c;
  level.tryCollectCircle(player.r, player.c);
}

function setup() {
  level = new Level(LEVELS[0], TS);
  createCanvas(level.pixelWidth(), level.pixelHeight());
  noStroke();
  textFont("sans-serif");
  textSize(14);

  player = new Player(LEVELS[0].playerStart.r, LEVELS[0].playerStart.c, TS);
  level.tryCollectCircle(player.r, player.c);
}

function draw() {
  background(240);

  level.draw();
  player.draw();

  // HUD (top)
  fill(0);
  textAlign(LEFT, TOP);
  text(level.data.name, 8, 6);
  text(`Circles left: ${level.circlesLeft()}`, 8, 24);

  // Auto-advance
  if (level.allCirclesCollected()) {
    if (levelIndex < LEVELS.length - 1) loadLevel(levelIndex + 1);
    else {
      fill(0);
      textAlign(CENTER, CENTER);
      text("All levels complete", width / 2, height / 2);
      noLoop();
    }
  }
}
```

```
function keyPressed() {
  if (keyCode === LEFT_ARROW) player.move(0, -1, level);
  if (keyCode === RIGHT_ARROW) player.move(0, 1, level);
  if (keyCode === UP_ARROW) player.move(-1, 0, level);
  if (keyCode === DOWN_ARROW) player.move(1, 0, level);
}
```