

Tab 1

GBDA 302 – Global Digital Project 2

Jowan Manjooran Jomon

jmanjoor **21105035**

Side Quest Week 3 – Process & Decision Documentation

1. What I was trying to do

For Week 3, I aimed to use the provided multi-screen p5.js starter structure to build a short branching interactive story. Instead of relying on random outcomes, my goal was to have player choices determine which scenes and ending they reached, inspired by the game *Dispatch*.

2. One small decision or change I made

I replaced the original random win/lose outcome in the game screen with a two-step decision structure, where the player makes an initial choice, encounters a follow-up scenario, and then reaches an ending based on those decisions.

3. Evidence of the change

- Random win/lose logic was removed from game.js
- Player choices are tracked using simple variables
- Different endings are reached based on decisions rather than chance

4. GenAI Use

I used GenAI (ChatGPT) to help explain the starter code, generate draft code for a branching decision structure, and refine on-screen instructional and narrative text. I

reviewed, tested, and modified the generated code to ensure it aligned with the course template and worked correctly within the existing state-based structure.

Appendix: GenAI Transcript

Tool Used: ChatGPT

Prompt

```
startt.js file // NOTE: Do NOT add setup() or draw() in this file

// setup() and draw() live in main.js

// This file only defines:

// 1) drawStart() → what the start/menu screen looks like

// 2) input handlers → what happens on click / key press on this screen

// 3) a helper function to draw menu buttons

// -----
// Start screen visuals
// -----
// drawStart() is called from main.js only when:
// currentScreen === "start"

function drawStart() {

    // Background colour for the start screen
    background(180, 225, 220); // soft teal background
```

```
// ---- Title text ----  
  
fill(30, 50, 60);  
  
textSize(46);  
  
textAlign(CENTER, CENTER);  
  
text("Win or Lose", width / 2, 180);  
  
  
  
// ---- Buttons (data only) ----  
  
// These objects store the position/size/label for each button.  
  
// Using objects makes it easy to pass them into drawButton()  
  
// and also reuse the same information for hover checks.  
  
const startBtn = {  
  
    x: width / 2,  
  
    y: 320,  
  
    w: 240,  
  
    h: 80,  
  
    label: "START",  
  
};  
  
  
  
const instrBtn = {  
  
    x: width / 2,  
  
    y: 430,
```

```
w: 240,  
h: 80,  
label: "INSTRUCTIONS",  
};  
  
// Draw both buttons  
drawButton(startBtn);  
drawButton(instrBtn);  
  
// ---- Cursor feedback ----  
// If the mouse is over either button, show a hand cursor  
// so the player knows it is clickable.  
const over = isHover(startBtn) || isHover(instrBtn);  
cursor(over ? HAND : ARROW);  
}  
  
// -----  
// Mouse input for the start screen  
// -----  
// Called from main.js only when currentScreen === "start"  
function startMousePressed() {
```

```
// For input checks, we only need x,y,w,h (label is optional)

const startBtn = { x: width / 2, y: 320, w: 240, h: 80 };

const instrBtn = { x: width / 2, y: 430, w: 240, h: 80 };

// If START is clicked, go to the game screen

if (isHover(startBtn)) {

    currentScreen = "game";

}

// If INSTRUCTIONS is clicked, go to the instructions screen

else if (isHover(instrBtn)) {

    currentScreen = "instr";

}

}

// -----
// Keyboard input for the start screen
// -----
// Provides keyboard shortcuts:

// - ENTER starts the game

// - I opens instructions

function startKeyPressed() {
```

```
if (keyCode === ENTER) {  
  
    currentScreen = "game";  
  
}  
  
  
  
if (key === "i" || key === "l") {  
  
    currentScreen = "instr";  
  
}  
  
}  
  
  
  
// -----  
  
// Helper: drawButton()  
  
// -----  
  
// This function draws a button and changes its appearance on hover.  
  
// It does NOT decide what happens when you click the button.  
  
// That logic lives in startMousePressed() above.  
  
//  
  
// Keeping drawing separate from input/logic makes code easier to read.  
  
function drawButton({ x, y, w, h, label }) {  
  
    rectMode(CENTER);  
  
  
  
    // Check if the mouse is over the button rectangle
```

```
const hover = isHover({ x, y, w, h });

noStroke();

// ---- Visual feedback (hover vs not hover) ----

// This is a common UI idea:
// - normal state is calmer
// - hover state is brighter + more "active"
//

// We also add a shadow using drawingContext (p5 lets you access the
// underlying canvas context for effects like shadows).

if (hover) {
    fill(255, 200, 150, 220); // warm coral on hover

    // Shadow settings (only when hovered)
    drawingContext.shadowBlur = 20;
    drawingContext.shadowColor = color(255, 180, 120);

} else {
    fill(255, 240, 210, 210); // soft cream base

    // Softer shadow when not hovered
}
```

```
drawingContext.shadowBlur = 8;  
drawingContext.shadowColor = color(220, 220, 220);  
}  
  
// Draw the rounded rectangle button  
rect(x, y, w, h, 14);  
  
// Important: reset shadow so it does not affect other drawings  
drawingContext.shadowBlur = 0;  
  
// Draw the label text on top of the button  
fill(40, 60, 70);  
textSize(28);  
textAlign(CENTER, CENTER);  
text(label, x, y);  
}
```

Response

Good – your `start.js` is already correct structurally. For “Dispatch vibe,” you only need **small text + label changes** (optional: one extra subtitle). You do **not** need to touch the routing logic unless you want different keyboard shortcuts.

Here’s what to change **inside this file** (no new architecture):

✓ Changes to make in `drawStart()`

1) Change the title

Replace:

- "Win or Lose"

With something like:

- "DISPATCH: Night Shift"
or
- "DISPATCH / NIGHT SHIFT"

Why: instantly sets theme (Design & Clarity points).

2) Add a subtitle line (optional but makes it feel “real”)

Under the title, add **one short line** like:

- "02:13 AM • Storm warning • Lines unstable"

This is just a `text()` call, no logic. Makes it Dispatch.

3) Rename the START button label (optional)

In `startBtn`, change label from:

- "START"

To:

- "BEGIN SHIFT" (best)
or
- "TAKE CALLS"

Keep the instructions button as-is.

Everything else in `drawStart()` stays.

Changes to make in `startMousePressed()` / `startKeyPressed()` (optional)

Option A (recommended): leave logic exactly the same

Keep:

- Start → `currentScreen = "game"`
- Instructions → `currentScreen = "instr"`
- ENTER starts, I opens instructions

This is fine and consistent with the template.

Option B (if you want Dispatch keyboard feel)

You can add shortcuts like:

- `S` for start shift
- `H` for help/instructions

But it's not required. Don't overcomplicate.

What NOT to change

- Don't add `setup()` or `draw()` (main.js owns that).
 - Don't change `isHover()` usage (it's shared and correct).
 - Don't add new screens here.
-

Quick “before/after” checklist

After edits, your start screen should read like:

- **DISPATCH: Night Shift**
- "02:13 AM • Storm warning • Lines unstable"
- Button: **BEGIN SHIFT**
- Button: INSTRUCTIONS

That's it ✅

If you paste your `instructions.js` next, I'll tell you exactly what text to change there so it matches the branching story (still no code changes beyond text unless you want).

Prompt

```
// NOTE: Do NOT add setup() or draw() in this file

// setup() and draw() live in main.js

// This file only defines:

// 1) drawStart() → what the start/menu screen looks like

// 2) input handlers → what happens on click / key press on this screen

// 3) a helper function to draw menu buttons

// -----
// Start screen visuals
// -----
// drawStart() is called from main.js only when:
// currentScreen === "start"

function drawStart() {
```

```
// Background colour for the start screen  
  
background(180, 225, 220); // soft teal background  
  
  
// ---- Title text ----  
  
fill(30, 50, 60);  
  
textSize(46);  
  
textAlign(CENTER, CENTER);  
  
  
  
text("02:13 AM • Tornado warning • Lines unstable", width / 2, 180);  
  
  
  
// ---- Buttons (data only) ----  
  
// These objects store the position/size/label for each button.  
  
// Using objects makes it easy to pass them into drawButton()  
  
// and also reuse the same information for hover checks.  
  
const startBtn = {  
  
    x: width / 2,  
  
    y: 320,  
  
    w: 240,  
  
    h: 80,  
  
    label: "BEGIN SHIFT",  
  
};
```

```
const instrBtn = {  
    x: width / 2,  
    y: 430,  
    w: 240,  
    h: 80,  
    label: "INSTRUCTIONS",  
};  
  
// Draw both buttons  
  
drawButton(startBtn);  
  
drawButton(instrBtn);  
  
// ---- Cursor feedback ----  
  
// If the mouse is over either button, show a hand cursor  
  
// so the player knows it is clickable.  
  
const over = isHover(startBtn) || isHover(instrBtn);  
  
cursor(over ? HAND : ARROW);  
}  
  
// -----
```

```
// Mouse input for the start screen

// -----
// Called from main.js only when currentScreen === "start"

function startMousePressed() {

    // For input checks, we only need x,y,w,h (label is optional)

    const startBtn = { x: width / 2, y: 320, w: 240, h: 80 };

    const instrBtn = { x: width / 2, y: 430, w: 240, h: 80 };



    // If START is clicked, go to the game screen

    if (isHover(startBtn)) {

        currentScreen = "game";

    }

    // If INSTRUCTIONS is clicked, go to the instructions screen

    else if (isHover(instrBtn)) {

        currentScreen = "instr";

    }

}

// -----
// Keyboard input for the start screen

// -----
```

```
// Provides keyboard shortcuts:  
  
// - ENTER starts the game  
  
// - I opens instructions  
  
function startKeyPressed() {  
  
    if (keyCode === ENTER) {  
  
        currentScreen = "game";  
  
    }  
  
  
    if (key === "i" || key === "I") {  
  
        currentScreen = "instr";  
  
    }  
  
}  
  
  
// -----  
  
// Helper: drawButton()  
  
// -----  
  
// This function draws a button and changes its appearance on hover.  
  
// It does NOT decide what happens when you click the button.  
  
// That logic lives in startMousePressed() above.  
  
//  
  
// Keeping drawing separate from input/logic makes code easier to read.
```

```
function drawButton({ x, y, w, h, label }) {  
  rectMode(CENTER);  
  
  // Check if the mouse is over the button rectangle  
  const hover = isHover({ x, y, w, h });  
  
  noStroke();  
  
  // ---- Visual feedback (hover vs not hover) ----  
  // This is a common UI idea:  
  // - normal state is calmer  
  // - hover state is brighter + more "active"  
  //  
  // We also add a shadow using drawingContext (p5 lets you access the  
  // underlying canvas context for effects like shadows).  
  if (hover) {  
    fill(255, 200, 150, 220); // warm coral on hover  
  
    // Shadow settings (only when hovered)  
    drawingContext.shadowBlur = 20;  
    drawingContext.shadowColor = color(255, 180, 120);
```

```
    } else {  
  
        fill(255, 240, 210, 210); // soft cream base  
  
        // Softer shadow when not hovered  
  
        drawingContext.shadowBlur = 8;  
  
        drawingContext.shadowColor = color(220, 220, 220);  
  
    }  
  
    // Draw the rounded rectangle button  
  
    rect(x, y, w, h, 14);  
  
    // Important: reset shadow so it does not affect other drawings  
  
    drawingContext.shadowBlur = 0;  
  
    // Draw the label text on top of the button  
  
    fill(40, 60, 70);  
  
    textSize(28);  
  
    textAlign(CENTER, CENTER);  
  
    text(label, x, y);  
  
} change the code accordingly and also make sure you give a line break after title
```

Response

You're close. Two fixes:

1. You accidentally replaced the **title** with the **subtitle** (you need *both*).
2. You want a **line break** after the title → easiest is: **draw title and subtitle as two separate `text()` calls** at different Y positions (p5 doesn't auto-line-break unless you use `\n`, but spacing with two calls looks cleaner).

Here's the updated `start.js` with:

- Title: **DISPATCH: Night Shift**
- Subtitle on the next line: **02:13 AM • Tornado warning • Lines unstable**
- A clear line break/spacing between them
- Everything else unchanged (buttons + logic)

```
// NOTE: Do NOT add setup() or draw() in this file

// setup() and draw() live in main.js

// This file only defines:

// 1) drawStart() → what the start/menu screen looks like

// 2) input handlers → what happens on click / key press on this screen

// 3) a helper function to draw menu buttons

// -----
// Start screen visuals
// -----
// drawStart() is called from main.js only when:
// currentScreen === "start"

function drawStart() {
```

```
// Background colour for the start screen  
  
background(180, 225, 220); // soft teal background  
  
  
// ---- Title + subtitle text ----  
  
fill(30, 50, 60);  
  
textAlign(CENTER, CENTER);  
  
  
// Title (top line)  
  
textSize(50);  
  
text("DISPATCH: Night Shift", width / 2, 160);  
  
  
  
// Subtitle (line break / next line)  
  
textSize(18);  
  
text("02:13 AM • Tornado warning • Lines unstable", width / 2, 205);  
  
  
  
// ---- Buttons (data only) ----  
  
const startBtn = {  
    x: width / 2,  
    y: 320,  
    w: 240,  
    h: 80,
```

```
    label: "BEGIN SHIFT",  
};  
  
const instrBtn = {  
    x: width / 2,  
    y: 430,  
    w: 240,  
    h: 80,  
    label: "INSTRUCTIONS",  
};  
  
// Draw both buttons  
drawButton(startBtn);  
drawButton(instrBtn);  
  
// ----- Cursor feedback -----  
const over = isHover(startBtn) || isHover(instrBtn);  
cursor(over ? HAND : ARROW);  
}  
  
// -----
```

```
// Mouse input for the start screen

// -----
function startMousePressed() {

    const startBtn = { x: width / 2, y: 320, w: 240, h: 80 };

    const instrBtn = { x: width / 2, y: 430, w: 240, h: 80 };

    if (isHover(startBtn)) {

        currentScreen = "game";

    } else if (isHover(instrBtn)) {

        currentScreen = "instr";

    }

}

// -----
// Keyboard input for the start screen
// -----
function startKeyPressed() {

    if (keyCode === ENTER) {

        currentScreen = "game";

    }

}
```

```
if (key === "i" || key === "l") {  
  
    currentScreen = "instr";  
  
}  
  
}  
  
// -----  
  
// Helper: drawButton()  
  
// -----  
  
function drawButton({ x, y, w, h, label }) {  
  
    rectMode(CENTER);  
  
    const hover = isHover({ x, y, w, h });  
  
    noStroke();  
  
    if (hover) {  
  
        fill(255, 200, 150, 220);  
  
        drawingContext.shadowBlur = 20;  
  
        drawingContext.shadowColor = color(255, 180, 120);  
  
    } else {  
  
        fill(255, 240, 210, 210);  
  
    }  
}
```

```
drawingContext.shadowBlur = 8;  
drawingContext.shadowColor = color(220, 220, 220);  
}  
  
rect(x, y, w, h, 14);  
  
drawingContext.shadowBlur = 0;  
  
fill(40, 60, 70);  
textSize(28);  
textAlign(CENTER, CENTER);  
text(label, x, y);  
}
```

If you paste your `instructions.js` next, I'll rewrite the instruction text so it matches the Dispatch decision tree (and remove the "random result" wording).

Prompt

```
this is win // NOTE: Do NOT add setup() or draw() in this file  
// setup() and draw() live in main.js  
// This file only defines:
```

```
// 1) drawWin() → what the win screen looks like  
// 2) input handlers → how the player returns to the start screen  
  
//  
  
// This file is intentionally very similar to lose.js.  
  
// The goal is to show that win/lose screens are often  
// simple “end states” with minimal logic.
```

```
// -----  
  
// Main draw function for win screen  
// -----  
  
// drawWin() is called from main.js  
// only when currentScreen === "win"  
  
function drawWin() {  
  
    // Green-tinted background to communicate success  
    background(200, 255, 200);  
  
    fill(0);  
    textAlign(CENTER, CENTER);  
  
    // Main success message  
    textSize(40);
```

```
text("You Win!", width / 2, 300);

// Instruction text

textSize(20);

text("Click or press R to return to Start.", width / 2, 360);

}
```

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

```
// Mouse input for win screen
```

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

```
// Any mouse click returns the player to the start screen
```

```
function winMousePressed() {
```

```
    currentScreen = "start";
```

```
}
```

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

```
// Keyboard input for win screen
```

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

```
// R is commonly used for "restart" in games
```

```
function winKeyPressed() {
```

```
    if (key === "r" || key === "R") {
```

```
    currentScreen = "start";  
}  
}
```

Response

```
// NOTE: Do NOT add setup() or draw() in this file  
  
// setup() and draw() live in main.js  
  
// This file only defines:  
  
// 1) drawWin() → what the win screen looks like  
  
// 2) input handlers → how the player returns to the start screen  
  
//  
  
// This file is intentionally very similar to lose.js.  
  
// The goal is to show that win/lose screens are often  
  
// simple “end states” with minimal logic.
```

```
// -----  
  
// Main draw function for win screen  
  
// -----  
  
// drawWin() is called from main.js  
  
// only when currentScreen === "win"  
  
function drawWin() {  
  
    // Calm green-tinted background (success / relief)
```

```
background(200, 255, 200);

fill(0);

textAlign(CENTER, CENTER);

// ---- Title (top line) ----

textSize(44);

text("CONTROLLED RESOLUTION", width / 2, 260);

// ---- Outcome (line break / next line) ----

textSize(18);

const outcome =

"Rescue arrives prepared.\n" +

"No lives lost.\n" +

"You traded urgency for judgment.";

text(outcome, width / 2, 335);

// ---- Instruction (separate line) ----

textSize(18);

text("Click or press R to return to Start.", width / 2, 470);

}
```

```
// -----  
// Mouse input for win screen  
// -----  
// Any mouse click returns the player to the start screen  
  
function winMousePressed() {  
  
    currentScreen = "start";  
  
}
```

```
// -----  
// Keyboard input for win screen  
// -----  
// R is commonly used for "restart" in games  
  
function winKeyPressed() {  
  
    if (key === "r" || key === "R") {  
  
        currentScreen = "start";  
  
    }  
  
}
```

Prompt

```
main.js // -----  
// main.js = the "router" (traffic controller) for the whole game
```

```
// -----  
  
//  
  
// Idea: this project has multiple screens (start, instructions, game, win, lose).  
  
// Instead of putting everything in one giant file, each screen lives in its own  
  
// file and defines two main things:  
  
// 1) drawX()      → how that screen looks  
  
// 2) XMousePressed() / XKeyPressed() → how that screen handles input  
  
//  
  
// This main.js file does 3 important jobs:  
  
// A) stores the current screen in a single shared variable  
  
// B) calls the correct draw function each frame  
  
// C) sends mouse/keyboard input to the correct screen handler
```

```
// -----  
  
// Global game state  
  
// -----  
  
// This variable is shared across all files because all files run in the same  
  
// global JavaScript scope when loaded in index.html.  
  
//  
  
// We store the "name" of the current screen as a string.  
  
// Only one screen should be active at a time.
```

```
let currentScreen = "start"; // "start" | "instr" | "game" | "win" | "lose"

// -----
// setup() runs ONCE at the beginning

// -----
// This is where you usually set canvas size and initial settings.

function setup() {

  createCanvas(800, 800);

  // Sets a default font for all text() calls
  // (This can be changed later per-screen if you want.)

  textAlign("center");
  textFont("sans-serif");

}

// -----
// draw() runs every frame (many times per second)

// -----
// This is the core “router” for visuals.

// Depending on currentScreen, we call the correct draw function.

function draw() {

  // Each screen file defines its own draw function:
```

```
// start.js      → drawStart()  
  
// instructions.js → drawInstr()  
  
// game.js       → drawGame()  
  
// win.js        → drawWin()  
  
// lose.js       → drawLose()  
  
  
  
if (currentScreen === "start") drawStart();  
  
else if (currentScreen === "instr") drawInstr();  
  
else if (currentScreen === "game") drawGame();  
  
else if (currentScreen === "win") drawWin();  
  
else if (currentScreen === "lose") drawLose();  
  
  
  
// (Optional teaching note)  
  
// This “if/else chain” is a very common early approach.  
  
// Later in the course you might replace it with:  
  
// - a switch statement, or  
  
// - an object/map of screens  
  
}  
  
  
  
// -----  
  
// mousePressed() runs once each time the mouse is clicked
```

```
// -----
// This routes mouse input to the correct screen handler.

function mousePressed() {

    // Each screen *may* define a mouse handler:

    // start.js      → startMousePressed()

    // instructions.js → instrMousePressed()

    // game.js       → gameMousePressed()

    // win.js         → winMousePressed()

    // lose.js        → loseMousePressed()

    if (currentScreen === "start") startMousePressed();

    else if (currentScreen === "instr") instrMousePressed();

    else if (currentScreen === "game") gameMousePressed();

    // The ?(.) means "call this function only if it exists"

    // This prevents errors if a screen doesn't implement a handler.

    else if (currentScreen === "win") winMousePressed?().;

    else if (currentScreen === "lose") loseMousePressed?().;

}

// -----
// keyPressed() runs once each time a key is pressed
```

```
// -----
// This routes keyboard input to the correct screen handler.

function keyPressed() {

    // Each screen *may* define a key handler:

    // start.js      → startKeyPressed()

    // instructions.js → instrKeyPressed()

    // game.js       → gameKeyPressed()

    // win.js        → winKeyPressed()

    // lose.js       → loseKeyPressed()

    if (currentScreen === "start") startKeyPressed();

    else if (currentScreen === "instr") instrKeyPressed();

    else if (currentScreen === "game") gameKeyPressed?().;

    else if (currentScreen === "win") winKeyPressed?().;

    else if (currentScreen === "lose") loseKeyPressed?().;

}

// -----
// Shared helper function: isHover()

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

```

// Many screens have buttons.

// This helper checks whether the mouse is inside a rectangle.

// Important: our buttons are drawn using rectMode(CENTER),
// meaning x,y is the CENTRE of the rectangle.

// So we check mouseX and mouseY against half-width/half-height bounds.

// Input: an object with { x, y, w, h }

// Output: true if mouse is over the rectangle, otherwise false

function isHover({ x, y, w, h }) {

  return (
    mouseX > x - w / 2 && // mouse is right of left edge
    mouseX < x + w / 2 && // mouse is left of right edge
    mouseY > y - h / 2 && // mouse is below top edge
    mouseY < y + h / 2 // mouse is above bottom edge
  );
}

}

```

Reponse

Your `main.js` is **already correct** and should work with all the changes we made. 

What I checked (and why it's good)

- `currentScreen` values match your screens: "start" | "instr" | "game" | "win" | "lose" ✓
- `draw()` routes correctly to `drawStart`, `drawInstr`, `drawGame`, `drawWin`, `drawLose` ✓
- `mousePressed()` calls the right handlers ✓
- `keyPressed()` calls `gameKeyPressed?.()` which is perfect because your new `game.js` defines `gameKeyPressed()` ✓
- `isHover()` matches your buttons since you draw them with `rectMode(CENTER)` ✓

Only 1 optional improvement (not required)

In `mousePressed()` you call `gameMousePressed()` without `?.()`. That's fine because your `game.js` defines `gameMousePressed()`.

If you ever remove/rename that function later, it would error. If you want it extra safe, you *could* change:

```
else if (currentScreen === "game") gameMousePressed();
```

to:

```
else if (currentScreen === "game") gameMousePressed?();
```

But again: **not necessary** as long as `gameMousePressed()` exists (it does).

Last thing to verify for GitHub Pages

Your `index.html` must load files in this order (screen files first, then `main.js` last):
`start.js, instructions.js, game.js, win.js, lose.js, main.js`

If you paste your `index.html` I'll confirm it's exactly right and you're basically submission-ready.

Prompt

fix the main.js and cleanup redundant code //

// main.js = the "router" (traffic controller) for the whole game

// -----

//

// Idea: this project has multiple screens (start, instructions, game, win, lose).

// Instead of putting everything in one giant file, each screen lives in its own

// file and defines two main things:

// 1) drawX() → how that screen looks

// 2) XMousePressed() / XKeyPressed() → how that screen handles input

//

// This main.js file does 3 important jobs:

// A) stores the current screen in a single shared variable

// B) calls the correct draw function each frame

// C) sends mouse/keyboard input to the correct screen handler

// -----

// Global game state

```
// -----  
  
// This variable is shared across all files because all files run in the same  
// global JavaScript scope when loaded in index.html.  
  
//  
  
// We store the "name" of the current screen as a string.  
  
// Only one screen should be active at a time.  
  
let currentScreen = "start"; // "start" | "instr" | "game" | "win" | "lose"
```

```
// -----  
  
// setup() runs ONCE at the beginning  
  
// -----  
  
// This is where you usually set canvas size and initial settings.
```

```
function setup() {  
  
  createCanvas(800, 800);  
  
  
  // Sets a default font for all text() calls  
  // (This can be changed later per-screen if you want.)  
  
  textAlign(CENTER);  
  textFont("sans-serif");  
  
}  
  
// -----
```

```
// draw() runs every frame (many times per second)

// -----
// This is the core "router" for visuals.

// Depending on currentScreen, we call the correct draw function.

function draw() {

    // Each screen file defines its own draw function:

    // start.js      → drawStart()

    // instructions.js → drawInstr()

    // game.js       → drawGame()

    // win.js        → drawWin()

    // lose.js       → drawLose()

    if (currentScreen === "start") drawStart();

    else if (currentScreen === "instr") drawInstr();

    else if (currentScreen === "game") drawGame();

    else if (currentScreen === "win") drawWin();

    else if (currentScreen === "lose") drawLose();

    // (Optional teaching note)

    // This "if/else chain" is a very common early approach.

    // Later in the course you might replace it with:
```

```
// - a switch statement, or  
  
// - an object/map of screens  
  
}  
  
  
// -----  
  
// mousePressed() runs once each time the mouse is clicked  
  
// -----  
  
// This routes mouse input to the correct screen handler.  
  
function mousePressed() {  
  
    // Each screen *may* define a mouse handler:  
  
    // start.js      → startMousePressed()  
  
    // instructions.js → instrMousePressed()  
  
    // game.js       → gameMousePressed()  
  
    // win.js        → winMousePressed()  
  
    // lose.js       → loseMousePressed()  
  
  
    if (currentScreen === "start") startMousePressed();  
  
    else if (currentScreen === "instr") instrMousePressed();  
  
    else if (currentScreen === "game") gameMousePressed();  
  
    // The ?.() means “call this function only if it exists”  
  
    // This prevents errors if a screen doesn’t implement a handler.
```

```
else if (currentScreen === "win") winMousePressed?();  
else if (currentScreen === "lose") loseMousePressed?();  
}  
  
// -----
```

```
// keyPressed() runs once each time a key is pressed  
// -----  
// This routes keyboard input to the correct screen handler.
```

```
function keyPressed() {  
    // Each screen *may* define a key handler:  
    // start.js      → startKeyPressed()  
    // instructions.js → instrKeyPressed()  
    // game.js       → gameKeyPressed()  
    // win.js        → winKeyPressed()  
    // lose.js       → loseKeyPressed()
```

```
if (currentScreen === "start") startKeyPressed();  
else if (currentScreen === "instr") instrKeyPressed();  
else if (currentScreen === "game") gameKeyPressed?();  
else if (currentScreen === "win") winKeyPressed?();  
else if (currentScreen === "lose") loseKeyPressed?();
```

```
}
```

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

```
// Shared helper function: isHover()
```

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

```
//
```

```
// Many screens have buttons.
```

```
// This helper checks whether the mouse is inside a rectangle.
```

```
//
```

```
// Important: our buttons are drawn using rectMode(CENTER),
```

```
// meaning x,y is the CENTRE of the rectangle.
```

```
// So we check mouseX and mouseY against half-width/half-height bounds.
```

```
//
```

```
// Input: an object with { x, y, w, h }
```

```
// Output: true if mouse is over the rectangle, otherwise false
```

```
function isHover({ x, y, w, h }) {
```

```
    return (
```

```
        mouseX > x - w / 2 && // mouse is right of left edge
```

```
        mouseX < x + w / 2 && // mouse is left of right edge
```

```
        mouseY > y - h / 2 && // mouse is below top edge
```

```
        mouseY < y + h / 2 // mouse is above bottom edge
```

```
});  
}
```

Response

Here's a "cleaner + safer" `main.js` that matches your updated project. Changes I made:

- Uses a **switch** (cleaner than long `if/else`)
- Uses optional chaining `? . ()` for **all** handlers (prevents crashes if something is missing)
- Keeps `isHover()` exactly the same
- No changes to your screen names (`"start", "instr", "game", "win", "lose"`)

Paste this over your `main.js`:

```
// -----  
  
// main.js = the "router" (traffic controller) for the whole game  
  
// -----  
  
//  
  
// This project has multiple screens (start, instructions, game, win, lose).  
  
// Each screen lives in its own file and defines:  
  
// 1) drawX()          → how that screen looks  
  
// 2) XMousePressed/XKeyPressed → how that screen handles input  
  
//  
  
// This main.js file:  
  
// A) stores the current screen  
  
// B) calls the correct draw function each frame
```

```
// C) routes mouse/keyboard input to the correct handler

// -----
// Global game state
// -----

let currentScreen = "start"; // "start" | "instr" | "game" | "win" | "lose"

// -----
// setup() runs ONCE at the beginning
// -----

function setup() {
    createCanvas(800, 800);
    textAlign(CENTER);
    textFont("sans-serif");
}

// -----
// draw() runs every frame
// -----

function draw() {
    switch (currentScreen) {
        case "start":
```

```
    drawStart?().  
  
    break;  
  
    case "instr":  
  
        drawInstr?().  
  
        break;  
  
    case "game":  
  
        drawGame?().  
  
        break;  
  
    case "win":  
  
        drawWin?().  
  
        break;  
  
    case "lose":  
  
        drawLose?().  
  
        break;  
  
    default:  
  
        // Failsafe: if something sets an unknown screen, return to start  
  
        currentScreen = "start";  
  
        break;  
  
    }  
  
}
```

```
// -----
// mousePressed() routes mouse input

// -----

function mousePressed() {

    switch (currentScreen) {

        case "start":

            startMousePressed?.( );

            break;

        case "instr":

            instrMousePressed?.( );

            break;

        case "game":

            gameMousePressed?.( );

            break;

        case "win":

            winMousePressed?.( );

            break;

        case "lose":

            loseMousePressed?.( );

            break;

    }

}
```

```
}

// -----
// keyPressed() routes keyboard input

// -----

function keyPressed() {

    switch (currentScreen) {

        case "start":

            startKeyPressed?.(());

            break;

        case "instr":

            instrKeyPressed?.(());

            break;

        case "game":

            gameKeyPressed?.(());

            break;

        case "win":

            winKeyPressed?.(());

            break;

        case "lose":

            loseKeyPressed?.(());

    }

}
```

```
break;

}

}

// -----
// Shared helper: isHover()

// -----
// Buttons are drawn using rectMode(CENTER).

// This helper checks if mouseX/mouseY are inside the rectangle.

function isHover({ x, y, w, h }) {

  return (

    mouseX > x - w / 2 &&
    mouseX < x + w / 2 &&
    mouseY > y - h / 2 &&
    mouseY < y + h / 2
  );
}

}
```

If you paste your `index.html`, I'll confirm the script order is correct for GitHub Pages (that's the #1 reason these multi-file sketches break).

Tab 2

GBDA 302 – Global Digital Project 2

Jowan Manjooran Jomon

jmanjoor **21105035**

Side Quest Week 4 – Process & Decision Documentation

1. What I was trying to do

For Week 4, I aimed to extend the provided blob platformer starter code to generate levels using external JSON data instead of hardcoded layouts. My goal was to create a system where platforms, coins, and an exit door are dynamically generated using loops, and where completing one level automatically loads the next. This allowed the level structure to be controlled entirely through data rather than manual placement in the code.

2. One small decision or change I made

One key decision I made was to switch from manually defining platforms to using a tilemap array in a JSON file. Each character in the map represents a different game element (platforms, coins, or the door), which the game parses at runtime. This made it easier to adjust level layouts and ensured that multiple levels could be added without changing the core game logic.

3. Evidence of the change

- Level layouts are defined in levels.json using a tilemap array
- Nested loops in WorldLevel iterate through the map to create platforms, coins, and the door dynamically
- A second level is included in the JSON file and loads automatically once all coins are collected and the player reaches the door
- No platform or coin positions are hardcoded in sketch.js

4. GenAI Use

I used GenAI (ChatGPT) to help understand the provided starter code, debug issues related to JSON parsing and collision logic, and refine the level-generation approach. GenAI was also used to help clarify how to structure the tilemap and ensure that the dynamically generated levels met the Week 4 requirements. All generated suggestions were reviewed, tested, and modified to work correctly within the existing class-based framework.

Appendix: GenAI Transcript and Screenshots

Tool Used: ChatGPT

- 1) How do I generate a level using arrays or JSON data? I was thinking of making the game have it where the blob collects coins and passes through a door which takes them to the next level using JSON.**

You're already very close — the files you uploaded are literally a Week 4 "JSON + classes" level loader. You just need to extend the JSON schema to include coins and a door, then check collisions and call loadLevel(next) when the win condition is met.

Below is the clean mental model + the exact edits.

What "generate a level using arrays / JSON" means

Instead of hard-coding platforms like:

```
platforms.push(new Platform(0, 324, 640, 36));
```

You store level geometry/data in JSON:

```
"platforms": [ { "x":0, "y":324, "w":640, "h":36 } ]
```

Then you "generate" the level by looping:

```
this.platforms = levelJson.platforms.map(p => new Platform(p));
```

That's exactly what your WorldLevel already does.

worldLevel

Step 1 – Update levels.json to include coins + a door

Right now your JSON has platforms only.

levels

Add coins and door like this:

{

"schemaVersion": 1,

"levels": [

{

"name": "Intro Steps",

"gravity": 0.65,

"jumpV": -11.0,

```
"theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
```

```
"start": { "x": 80, "y": 220, "r": 26 },
```

"platforms": [

```
{ "x": 0, "y": 324, "w": 640, "h": 36 },
```

```
{ "x": 120, "y": 254, "w": 120, "h": 12 }
```

```
],
```

```
"coins": [
```

```
{ "x": 160, "y": 220, "r": 10 },
```

```
{ "x": 340, "y": 170, "r": 10 }
```

```
],
```

```
"door": { "x": 600, "y": 288, "w": 28, "h": 36 }
```

```
},
```

```
{
```

```
  "name": "Level 2",
```

```
  "gravity": 0.7,
```

```
  "jumpV": -12.0,
```

```
  "theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },
```

```
  "start": { "x": 60, "y": 220, "r": 26 },
```

```
  "platforms": [
```

```
    { "x": 0, "y": 324, "w": 640, "h": 36 },
```

```
{ "x": 260, "y": 260, "w": 140, "h": 12 }

],
"coins": [
{ "x": 300, "y": 230, "r": 10 }

],
"door": { "x": 16, "y": 288, "w": 28, "h": 36 }

}

]
```

That satisfies Week 4 + the bonus “second level loads automatically.”

Side Quests

Step 2 – Add Coin + Door classes (new files or inside existing)

```
coin.js

class Coin {

constructor({ x, y, r }) {

    this.x = x;

    this.y = y;
```

```
this.r = r ?? 10;

this.collected = false;

}

draw() {

if (this.collected) return;

push();

noStroke();

fill(255, 200, 0);

circle(this.x, this.y, this.r * 2);

pop();

}
```

```
tryCollect(player) {

if (this.collected) return false;

const d = dist(this.x, this.y, player.x, player.y);

if (d < this.r + player.r * 0.6) {

this.collected = true;

return true;

}

return false;
```

```
}
```

```
}
```

door.js

```
class Door {
```

```
constructor({ x, y, w, h }) {
```

```
    this.x = x; this.y = y; this.w = w; this.h = h;
```

```
}
```

```
draw(isUnlocked) {
```

```
    push();
```

```
    noStroke();
```

```
    fill(isUnlocked ? 60 : 120);
```

```
    rect(this.x, this.y, this.w, this.h, 6);
```

```
    pop();
```

```
}
```

```
overlapsPlayer(player) {
```

```
// treat player as a box (simple + good enough)
```

```
const box = { x: player.x - player.r, y: player.y - player.r, w: player.r * 2, h: player.r * 2 };
};
```

```
        return (

            box.x < this.x + this.w &&

            box.x + box.w > this.x &&

            box.y < this.y + this.h &&

            box.y + box.h > this.y

        );

    }

}
```

Also: if you create new files, add them in index.html before sketch.js, same as your other classes.

index

Step 3 – Update worldLevel.js to parse coins + door from JSON

Add to WorldLevel constructor:

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

```
this.door = levelJson.door ? new Door(levelJson.door) : null;
```

And update drawWorld() to also draw coins + door:

```
drawWorld() {  
    background(color(this.theme.bg));  
  
    for (const p of this.platforms) p.draw(color(this.theme.platform));  
    for (const c of this.coins) c.draw();  
  
    const allCoinsCollected = this.coins.every((c) => c.collected);  
    if (this.door) this.door.draw(allCoinsCollected);  
}
```

This plugs into your existing world pipeline cleanly.

worldLevel

Step 4 – Update sketch.js to handle coin collecting + auto next level

Add a score variable at the top:

```
let coinsCollected = 0;
```

Inside draw() after player.update(...):

```
// Collect coins
```

```
for (const c of world.coins) {  
    if (c.tryCollect(player)) coinsCollected++;  
}
```

```
const allCoinsCollected = world.coins.every((c) => c.collected);
```

```
// Door win -> next level
```

```
if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {  
    const next = (levelIndex + 1) % data.levels.length;  
    loadLevel(next);  
    coinsCollected = 0; // or keep global if you want  
}
```

And update HUD:

```
text(`Coins: ${coinsCollected}/${world.coins.length}`, 10, 54);
```

2) This is what levels look like rn { "schemaVersion": 1, "levels": [{ "name": "Intro Steps", "gravity": 0.65, "jumpV": -11.0, "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" }, "start": { "x": 80, "y": 220, "r": 26 }, "platforms": [{ "x": 0, "y": 324, "w": 640, "h": 36 }, { "x": 120, "y": 254, "w": 120, "h": 12 }, { "x": 300, "y": 204, "w": 90, "h": 12 }, { "x": 440, "y": 144, "w": 130, "h": 12 }, { "x": 520, "y": 254, "w": 90, "h": 12 }] }] }

Perfect — your JSON is already in the right shape. Now you just need to **add two new keys** inside the level object:

- "coins": [...] (an array)
- "door": { ... } (one object)

Here's your **same level**, updated cleanly:

```
{
  "schemaVersion": 1,
  "levels": [
    {
      "name": "Intro Steps",
      "gravity": 0.65,
      "jumpV": -11.0,
      "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
      "start": { "x": 80, "y": 220, "r": 26 },
      "platforms": [
        { "x": 0, "y": 324, "w": 640, "h": 36 },
        { "x": 120, "y": 254, "w": 120, "h": 12 },
        { "x": 300, "y": 204, "w": 90, "h": 12 },
        { "x": 440, "y": 144, "w": 130, "h": 12 }
      ]
    }
  ]
}
```

```

        { "x": 520, "y": 254, "w": 90, "h": 12 },
    ],
    "coins": [
        { "x": 150, "y": 220, "r": 10 },
        { "x": 330, "y": 170, "r": 10 },
        { "x": 500, "y": 120, "r": 10 }
    ],
    "door": { "x": 600, "y": 288, "w": 28, "h": 36 }
}
]
}

```

What those numbers mean

- **Coin x/y:** where it sits in the level
- **Coin r:** coin radius (size)
- **Door x/y/w/h:** rectangle position + size (easy collision)

This keeps you aligned with Week 4: you're now storing level content in JSON and will generate coins/door with loops when loading the level.

Side Quests

If you want the bonus (Level 2 auto-load)

Add a **second object** inside "levels": [...] like:

```
{
    "name": "Level 2",
    "gravity": 0.7,
    "jumpV": -12.0,
    "theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },
    "start": { "x": 60, "y": 220, "r": 26 },
    "platforms": [

```

```

    { "x": 0, "y": 324, "w": 640, "h": 36 },
    { "x": 240, "y": 260, "w": 160, "h": 12 }
],
"coins": [
  { "x": 280, "y": 230, "r": 10 }
],
"door": { "x": 16, "y": 288, "w": 28, "h": 36 }
}

```

Next immediate step (code side)

Now your code must do:

1. `level.coins.map(...)` → create Coin objects
2. `new Door(level.door)`
3. every frame: collect coins + if all collected and touching door → load next level

If you want, I'll read your `worldLevel.js` and `sketch.js` and tell you **exactly where** to paste each line so nothing breaks.

- 3) I got an expected comma error
- ```
{
 "schemaVersion": 1,
 "levels": [
 {
 "name": "Intro Steps",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": {
 "bg": "#FOFOFO",
 "platform": "#C8C8C8",
 "blob": "#1478FF"
 },
 "start": { "x": 80, "y": 220, "r": 26 },
 "platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 120, "y": 254, "w": 120, "h": 12 },
 { "x": 300, "y": 204, "w": 90, "h": 12 },
 { "x": 440, "y": 144, "w": 130, "h": 12 },
 { "x": 520, "y": 254, "w": 90, "h": 12 }
],
 "coins": [
 { "x": 150, "y": 220, "r": 10 },
 { "x": 330, "y": 170, "r": 10 },
 { "x": 500, "y": 120, "r": 10 }
],
 "door": { "x": 600, "y": 288, "w": 28, "h": 36 }
 },
 {
 "name": "Level 2",
 "gravity": 0.7,
 "jumpV": -12.0,
 "theme": {
 "bg": "#F6F1FF",
 "platform": "#B9A8FF",
 "blob": "#4A3CFF"
 },
 "start": { "x": 60, "y": 220, "r": 26 },
 "platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 240, "y": 260, "w": 160, "h": 12 }
],
 "coins": [
 { "x": 280, "y": 230, "r": 10 }
],
 "door": { "x": 16, "y": 288, "w": 28, "h": 36 }
 }
]
}
```

ep — that error is **100% because you're missing a comma** between the two level objects in the `levels` array.

Right now you have:

```
 }
 {
```

It must be:

```
,
 {
```

Also: your second level object needs to be indented the same, but indentation doesn't matter — commas do.

Here's the **fixed full JSON** (copy–paste exactly):

```
{
 "schemaVersion": 1,
 "levels": [
 {
 "name": "Intro Steps",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
 "start": { "x": 80, "y": 220, "r": 26 },

 "platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 120, "y": 254, "w": 120, "h": 12 },
 { "x": 300, "y": 204, "w": 90, "h": 12 },
 { "x": 440, "y": 144, "w": 130, "h": 12 },
 { "x": 520, "y": 254, "w": 90, "h": 12 }
],

 "coins": [
 { "x": 150, "y": 220, "r": 10 },
 { "x": 300, "y": 204, "r": 10 }
]
 }
]
}
```

```

 { "x": 330, "y": 170, "r": 10 },
 { "x": 500, "y": 120, "r": 10 }
],
}

"door": { "x": 600, "y": 288, "w": 28, "h": 36 }
},

{
 "name": "Level 2",
 "gravity": 0.7,
 "jumpV": -12.0,
 "theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },
 "start": { "x": 60, "y": 220, "r": 26 },

 "platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 240, "y": 260, "w": 160, "h": 12 }
],
}

"coins": [
 { "x": 280, "y": 230, "r": 10 }
],
}

"door": { "x": 16, "y": 288, "w": 28, "h": 36 }
}
]
}

```

If you still get an error after this, it's usually one of these:

- you accidentally left an extra } or ]
- you pasted it inside another JSON block
- there's a hidden character from copying

Next step: once your JSON loads, we wire your code so:

- coins disappear on touch
- door only works when all coins collected
- door loads the next JSON level automatically (Week 4 bonus)  
Side Quests

#### **4) step 3 worldlevel.js is like this rn /\***

#### **WorldLevel.js (Example 5)**

**WorldLevel wraps ONE level object from levels.json and provides:**

- **Theme colours (background/platform/blob)**
- **Physics parameters that influence the player (gravity, jump velocity)**
- **Spawn position for the player (start)**
- **An array of Platform instances**
- **A couple of helpers to size the canvas to fit the geometry**

**This is directly inspired by your original blob sketch's responsibilities:**

- **parse JSON**
- **map platforms array**
- **apply theme + physics**
- **infer canvas size**

**Expected JSON shape for each level (from your provided file):**

```
{
 "name": "Intro Steps",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": { "bg": "...", "platform": "...", "blob": "..." },
 "start": { "x": 80, "y": 220, "r": 26 },
 "platforms": [{x,y,w,h}, ...]
}
*/
```

```
class WorldLevel {

 constructor(levelJson) {

 // A readable label for HUD.

 this.name = levelJson.name || "Level";

 // Theme defaults + override with JSON.

 this.theme = Object.assign(
 { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },
 levelJson.theme || {}
);
 }
}
```

```
// Physics knobs (the blob player will read these).

this.gravity = levelJson.gravity ?? 0.65;

this.jumpV = levelJson.jumpV ?? -11.0;

// Player spawn data.

// Use optional chaining so levels can omit fields safely.

this.start = {

 x: levelJson.start?.x ?? 80,

 y: levelJson.start?.y ?? 180,

 r: levelJson.start?.r ?? 26,

};

// Convert raw platform objects into Platform instances.

this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));

}

/*
If you want the canvas to fit the world, you can infer width/height by
finding the maximum x+w and y+h across all platforms.

*/
```

```
inferWidth(defaultW = 640) {

 if (!this.platforms.length) return defaultW;

 return max(this.platforms.map((p) => p.x + p.w));

}

/*
```

```
inferHeight(defaultH = 360) {

 if (!this.platforms.length) return defaultH;

 return max(this.platforms.map((p) => p.y + p.h));

}
```

```
/*

Draw only the world (background + platforms).

The player draws itself separately, after the world is drawn.

*/

drawWorld() {

 background(color(this.theme.bg));

 for (const p of this.platforms) {

 p.draw(color(this.theme.platform));

 }

}
```

ood — this file is clean. You just need to add 2 lines in the constructor and 1 loop in drawWorld().

- 1) In constructor(levelJson) add this right after the platforms line

Find this line at the bottom of your constructor:

```
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));
```

Immediately under it, add:

```
// Convert raw coin objects into Coin instances.
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

So your constructor ends like:

```
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

- 2) In drawWorld() draw coins after platforms

Your drawWorld() currently:

```
drawWorld() {
 background(color(this.theme.bg));
 for (const p of this.platforms) {
 p.draw(color(this.theme.platform));
 }
}
```

```
}
```

Change it to:

```
drawWorld() {
 background(color(this.theme.bg));

 for (const p of this.platforms) {
 p.draw(color(this.theme.platform));
 }

 for (const c of this.coins) {
 c.draw();
 }
}
```

3) If coins STILL don't appear, it's 99% index.html script order

You must load coin.js before worldLevel.js and sketch.js.

In index.html, make sure you have:

```
<script src="platform.js"></script>
<script src="coin.js"></script>
<script src="worldLevel.js"></script>
<script src="blobPlayer.js"></script>
<script src="sketch.js"></script>
```

If coin.js isn't loaded, you'll either get Coin is not defined OR your code won't run right.

4) Quick check: do you see them but off-screen?

Put one coin right near the spawn in levels.json to confirm:

```
"coins": [
 { "x": 120, "y": 200, "r": 10 }
]
```

If that shows up, your drawing works – you just need to reposition the rest.

If you paste your current index.html script tags (just that section), I'll tell you instantly if the load order is the issue.

**5) coins loaded up but its not being collected**

Coin showing means draw is working. They're not being collected because your game loop never runs any "collect" logic right now – sketch.js only updates the player + draws HUD, no coin collision checks at all

sketch

Do this ↴

1) Add a coin counter (top of sketch.js)

Under your globals:

```
let coinsCollected = 0;
```

(You already have data, levelIndex, world, player there

```
sketch
```

```
.)
```

2) In draw() add the collect loop (RIGHT after player.update(...))

Your draw() currently does:

```
world.drawWorld();
player.update(world.platforms);
```

```
player.draw(world.theme.blob);
```

Add this between player.update(...) and player.draw(...):

```
for (const c of world.coins) {
 if (c.tryCollect(player)) coinsCollected++;
}
```

Why here? Because the player's x/y gets finalized at the end of update()

blobPlayer

, so you want to test coin overlap after that.

3) Show it in the HUD (optional but useful)

Add one more HUD line:

```
text(`Coins: ${coinsCollected}/${world.coins.length}`, 10, 54);
```

(Your HUD currently only prints two lines

sketch

.)

4) Reset coinsCollected when you load a new level

Inside loadLevel(i) add:

```
coinsCollected = 0;
```

because loadLevel() is the “new run” entry point

sketch

If it still doesn't collect after this

Two common gotchas:

A) Your Coin.tryCollect() is never returning true (radius too small)

Increase the overlap threshold in tryCollect() to be more forgiving:

```
if (d < this.r + player.r) { ... }
```

B) world.coins is empty

That means WorldLevel is not parsing coins yet – your posted worldLevel.js only maps platforms

worldLevel

. You need the constructor line:

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

If you paste your current coin.js (especially tryCollect) I'll tell you instantly whether it's a distance/radius issue or a logic issue.

**6) i added player.update and player.draw code could you tell me which file to update for each step too**

According to a document from (DATE): I can't see the files' modified dates in the snippets, but I can tell you exactly which file to change for each step, based on your current project structure.

Here's the map 

Step-by-step + which file to edit

1) Put coin data in JSON

 File: levels.json

Add "coins": [...] inside each level object. Your file currently only has "platforms"

levels

2) Create the Coin class

 File: coin.js (new file you create)

This file defines class Coin with:

draw()

tryCollect(player)

(No existing coin.js is shown in your repo right now, so it must be new.)

3) Make sure the browser loads coin.js

 File: index.html

Right now you load only:

platform.js

worldLevel.js

blobPlayer.js

sketch.js

index

You must insert:

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

between platform.js and worldLevel.js so WorldLevel can use Coin.

#### 4) Parse coins from JSON into objects

File: worldLevel.js

Right now, the constructor only converts platforms into Platform objects

worldLevel

.

Add:

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

right under the platforms mapping.

## 5) Draw coins every frame

File: worldLevel.js

Right now drawWorld() only draws background + platforms

worldLevel

Add:

```
for (const c of this.coins) c.draw();
```

after the platform loop.

6) Actually collect coins (this is why yours aren't collecting)

 File: sketch.js

Your draw() currently only does:

world.drawWorld()

player.update(world.platforms)

player.draw(...)

and then HUD

sketch

So add the collection loop in sketch.js, inside draw():

```
for (const c of world.coins) {
 c.tryCollect(player);
}
```

(If you want a counter, increment when it returns true.)

TL;DR file responsibility chart

levels.json → where coins exist (data)

coin.js → what a coin is (class)

index.html → makes coin.js load

worldLevel.js → converts JSON into Coin objects + draws them

sketch.js → runs “collect” logic each frame (the missing piece)

If you paste your current coin.js (just the tryCollect function), I'll tell you why it isn't triggering (usually distance threshold or player radius mismatch).

7) this is what levels.js looks like { "schemaVersion": 1, "levels": [ { "name": "Intro Steps", "gravity": 0.65, "jumpV": -11.0, "theme": { "bg": "#FOFOFO", "platform": "#C8C8C8", "blob": "#1478FF" }, "start": { "x": 80, "y": 220, "r": 26 }, "platforms": [ { "x": 0, "y": 324, "w": 640, "h": 36 }, { "x": 120, "y": 254, "w": 120, "h": 12 }, { "x": 300, "y": 204, "w": 90, "h": 12 }, { "x": 440, "y": 144, "w": 130, "h": 12 } ] } ] }

```
{ "x": 520, "y": 254, "w": 90, "h": 12 }, "coins": [{ "x": 150, "y": 220, "r": 10 }, { "x": 330, "y": 170, "r": 10 }, { "x": 500, "y": 120, "r": 10 }], "door": { "x": 600, "y": 288, "w": 28, "h": 36 } }, { "name": "Level 2", "gravity": 0.7, "jumpV": -12.0, "theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" }, "start": { "x": 60, "y": 220, "r": 26 }, "platforms": [{ "x": 0, "y": 324, "w": 640, "h": 36 }, { "x": 240, "y": 260, "w": 160, "h": 12 }], "coins": [{ "x": 280, "y": 230, "r": 10 }], "door": { "x": 16, "y": 288, "w": 28, "h": 36 } }] }
```

Your levels.json looks correct now ✅ (coins + door + 2 levels). The reason coins don't collect is not the JSON anymore — it's the code path that checks collection.

Here's the exact "who to edit where" for coin collection + door to next level.

1) worldLevel.js — make sure it creates coins (and door if you're doing it)

Inside constructor(levelJson), you should have both of these:

```
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));
```

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

And in drawWorld() you should draw coins:

```
for (const c of this.coins) c.draw();
```

If you already see coins on screen, you probably have this part done.

2) sketch.js – this is the file that must run coin collection every frame

Inside draw(), after player.update(...) and before HUD, add:

```
for (const c of world.coins) {
 c.tryCollect(player);
}
```

If you want a counter:

```
let coinsCollected = 0; // put at top of sketch.js (global)
```

```
for (const c of world.coins) {
 if (c.tryCollect(player)) coinsCollected++;
}
```

Why sketch.js? Because worldLevel.js just stores/draws stuff – the actual “game logic loop” lives in draw().

3) coin.js – make sure tryCollect(player) returns true when it collects

Your Coin.tryCollect needs to check distance against the player position + radius. Use this exact logic:

```
tryCollect(player) {
 if (this.collected) return false;

 const d = dist(this.x, this.y, player.x, player.y);
 if (d < this.r + player.r) { // <- forgiving threshold
 this.collected = true;
 return true;
 }
 return false;
}
```

If your threshold is too small (like player.r \* 0.2), it will feel “never collects”.

4) index.html – make sure coin.js loads before worldLevel.js

Your script order must include coin.js before worldLevel.js:

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

<script src="coin.js"></script>

<script src="worldLevel.js"></script>

<script src="blobPlayer.js"></script>

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

Quick “diagnose in 10 seconds”

In sketch.js, temporarily add this inside the loop:

```
for (const c of world.coins) {

 if (c.tryCollect(player)) console.log("collected!");

}
```

If you see “collected!” in console → works.

If not → the issue is your tryCollect math OR player.x/player.y aren't what you think.

If you paste your current coin.js (especially tryCollect) I'll point to the exact line causing the non-collection and give you the corrected version.

## 8) This is what worldlevel.js look like /

### WorldLevel.js (Example 5)

**WorldLevel wraps ONE level object from levels.json and provides:**

- Theme colours (background/platform/blob)
- Physics parameters that influence the player (gravity, jump velocity)
- Spawn position for the player (start)
- An array of Platform instances
- A couple of helpers to size the canvas to fit the geometry

**This is directly inspired by your original blob sketch's responsibilities:**

- parse JSON
- map platforms array
- apply theme + physics
- infer canvas size

**Expected JSON shape for each level (from your provided file):**

```
{
 "name": "Intro Steps",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": { "bg": "...", "platform": "...", "blob": "..." },
 "start": { "x": 80, "y": 220, "r": 26 },
 "platforms": [{x,y,w,h}, ...]
}
/
/
```

```
class WorldLevel {
 constructor(levelJson) {
 // A readable label for HUD.

 this.name = levelJson.name || "Level";

 // Theme defaults + override with JSON.

 this.theme = Object.assign(
 { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },
 levelJson.theme || {}
);
}
```

```
// Physics knobs (the blob player will read these).
```

```
this.gravity = levelJson.gravity ?? 0.65;
```

```
this.jumpV = levelJson.jumpV ?? -11.0;
```

```
// Player spawn data.
```

```
// Use optional chaining so levels can omit fields safely.
```

```
this.start = {
```

```
 x: levelJson.start?.x ?? 80,
```

```
 y: levelJson.start?.y ?? 180,
```

```
 r: levelJson.start?.r ?? 26,
```

```
};
```

```
// Convert raw platform objects into Platform instances.
```

```
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));
```

```
// Convert raw coin objects into Coin instances.
```

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

```
}
```

```
/
```

If you want the canvas to fit the world, you can infer width/height by

finding the maximum x+w and y+h across all platforms.

```
/

inferWidth(defaultW = 640) {

 if (!this.platforms.length) return defaultW;

 return max(this.platforms.map((p) => p.x + p.w));

}

//
```

```
inferHeight(defaultH = 360) {

 if (!this.platforms.length) return defaultH;

 return max(this.platforms.map((p) => p.y + p.h));

}

//
```

**Draw only the world (background + platforms).**

**The player draws itself separately, after the world is drawn.**

```
/

drawWorld() {

 background(color(this.theme.bg));

 for (const p of this.platforms) {

 p.draw(color(this.theme.platform));

 }
}
```

```
for (const c of this.coins) {
 c.draw();
}
}
}
```

According to a document from \*\*Feb. 5, 2026\*\*, your project is currently only updating/drawing the \*\*player\*\* and \*\*platforms\*\* in `sketch.js` (no coin-collection logic runs anywhere yet)—so even if coins are visible, they won’t “collect” until you add the loop in the main game loop.

Here’s \*\*exactly which file\*\* to update for each step (and what to paste).

----

## 0) Make sure you actually saved the updated JSON

Your repo’s current `levels.json` (the one your code loads) still shows \*\*only platforms\*\* and only \*\*one level\*\*. Your pasted JSON (with `coins` + `door` + Level 2) is correct—but it only matters if it’s saved into the actual `levels.json` file.

✓ \*\*File:\*\* `levels.json`

→ Paste your updated JSON into the real file and save.

---

## ## 1) Load `coin.js` in the browser

Right now `index.html` loads: `platform.js`, `worldLevel.js`, `blobPlayer.js`, `sketch.js`.

If `coin.js` isn't loaded **before** `worldLevel.js`, you'll get "Coin is not defined" or it won't behave right.

✓ \*\*File:\*\* `index.html`

→ Change your script section to:

```html

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

<script src="coin.js"></script>

<script src="worldLevel.js"></script>

<script src="blobPlayer.js"></script>

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

```

(coin must be before worldLevel).

---

## ## 2) Parse + draw coins from JSON

Your `WorldLevel` in the repo version currently only maps platforms and draws platforms.

You said you already added `this.coins` + drawing—good. Just confirming the required pieces:

✓ \*\*File:\*\* `worldLevel.js`

→ In the constructor (after platforms):

```
```js
```

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

```
```
```

→ In `drawWorld()` (after platform loop):

```
```js
```

```
for (const c of this.coins) c.draw();
```

```
```
```

---

## 3) \*\*THIS is why coins aren't collecting:\*\* you must check coins in `draw()`

Your `sketch.js` `draw()` only does: world draw → player update/draw → HUD.

No coin checks happen, so nothing ever sets `coin.collected = true`.

✓ \*\*File:\*\* `sketch.js`

→ Add a counter at the top (with your globals):

```js

```
let coinsCollected = 0;
```

```

→ Then in `draw()`, put this \*\*right after\*\* `player.update(world.platforms);` and \*\*before\*\* `player.draw(...)`:

```js

```
for (const c of world.coins) {  
    if (c.tryCollect(player)) coinsCollected++;
```

```
}
```

```
~~~
```

Why there? Because player position is finalized at the end of `update()`.

→ Update HUD (optional but helpful). Add a third line:

```
~~~js
```

```
text(`Coins: ${coinsCollected}/${world.coins.length}`, 10, 54);
```

```
~~~
```

Your HUD currently only has two lines.

→ Reset the counter when loading a new level:

✓ **File:** `sketch.js` inside `loadLevel(i)`:

```
~~~js
```

```
coinsCollected = 0;
```

```
~~~
```

That function is your “new level” entry point.

4) Make sure `coin.js` collision math is forgiving

If your `tryCollect()` threshold is too small, it’ll feel like it never collects.

✓ **File:** `coin.js`

Use this overlap rule:

```js

```
const d = dist(this.x, this.y, player.x, player.y);
if (d < this.r + player.r) { ... }
```

---

---

## ### Fast debug (10 seconds)

In `sketch.js`, temporarily do:

```
```js
if (c.tryCollect(player)) console.log("collected");
```
```

```

If you never see "collected", it's either:

- * `tryCollect` math, OR
- * `world.coins` is empty because `levels.json` wasn't saved (common), OR
- * `coin.js` isn't loaded before `worldLevel.js`.

If you paste your current `coin.js` (just the `tryCollect` function), I'll tell you exactly why it isn't triggering and give the corrected version.

9) this is what **worlds.js** looks like /*

WorldLevel.js (Example 5)

WorldLevel wraps ONE level object from levels.json and provides:

- **Theme colours (background/platform/blob)**

- Physics parameters that influence the player (gravity, jump velocity)
- Spawn position for the player (start)
- An array of Platform instances
- A couple of helpers to size the canvas to fit the geometry

This is directly inspired by your original blob sketch's responsibilities:

- parse JSON
- map platforms array
- apply theme + physics
- infer canvas size

Expected JSON shape for each level (from your provided file):

```
{  
  "name": "Intro Steps",  
  "gravity": 0.65,  
  "jumpV": -11.0,  
  "theme": { "bg": "...", "platform": "...", "blob": "..." },  
  "start": { "x": 80, "y": 220, "r": 26 },  
  "platforms": [ {x,y,w,h}, ... ]  
}
```

*/

```
class WorldLevel {  
  
  constructor(levelJson) {  
  
    // A readable label for HUD.  
  
    this.name = levelJson.name || "Level";  
  
    // Theme defaults + override with JSON.  
  
    this.theme = Object.assign(  
  
      { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },  
  
      levelJson.theme || {}  
    );  
  
    // Physics knobs (the blob player will read these).  
  
    this.gravity = levelJson.gravity ?? 0.65;  
  
    this.jumpV = levelJson.jumpV ?? -11.0;  
  
    // Player spawn data.  
  
    // Use optional chaining so levels can omit fields safely.  
  
    this.start = {  
  
      x: levelJson.start?.x ?? 80,  
  
      y: levelJson.start?.y ?? 180,  
    }  
  }  
}
```

```
r: levelJson.start?.r ?? 26,  
};  
  
// Convert raw platform objects into Platform instances.  
  
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));  
  
// Convert raw coin objects into Coin instances.  
  
this.coins = (levelJson.coins || []).map((c) => new Coin(c));  
}
```

/*

**If you want the canvas to fit the world, you can infer width/height by
finding the maximum x+w and y+h across all platforms.**

***/**

```
inferWidth(defaultW = 640) {  
  
if (!this.platforms.length) return defaultW;  
  
return max(this.platforms.map((p) => p.x + p.w));  
}
```

```
inferHeight(defaultH = 360) {  
  
if (!this.platforms.length) return defaultH;  
  
return max(this.platforms.map((p) => p.y + p.h));
```

```
}
```

```
/*
```

Draw only the world (background + platforms).

The player draws itself separately, after the world is drawn.

```
*/
```

```
drawWorld() {
```

```
    background(color(this.theme.bg));
```

```
    for (const p of this.platforms) {
```

```
        p.draw(color(this.theme.platform));
```

```
}
```

```
    for (const c of this.coins) {
```

```
        c.draw();
```

```
}
```

```
}
```

Good – then you're at the **finish line**. You need three things:

1. **Score / points counter updates when a coin is collected**

2. **Door only appears (or unlocks) when ALL coins are collected**

3. **Touching the door loads the next level**

Below is the clean “edit-by-file” checklist + paste-ready code.

1) `sketch.js` – add points + door-to-next-level logic

A) Add globals at the top

Put these near your other globals:

```js

```
let coinsCollected = 0;
```

```
let totalCoins = 0;
```

```

B) Reset counts when loading a level

Inside your `loadLevel(i)` function (right after you create `world = new WorldLevel(...)`), add:

```
```js
```

```
coinsCollected = 0;

totalCoins = world.coins.length;
```

```
```
```

C) In `draw()` update points only when a coin is newly collected

Right after `player.update(world.platforms);` add:

```
```js
```

```
for (const c of world.coins) {

 if (c.tryCollect(player)) {

 coinsCollected++;

 }

}
```

```
```
```

D) Door unlock + next level trigger

Still inside `draw()`, after the coin loop:

```
```js
const allCoinsCollected = coinsCollected >= totalCoins;

if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {
 loadLevel(levelIndex + 1);
}

```
#### E) HUD (points display)
```

Where you draw text, add:

```
```js
text(`Coins: ${coinsCollected}/${totalCoins}`, 10, 54);

```
---
```

2) `door.js` – create the Door class (new file)

Create a new file **door.js**:

```
```js
```

```
class Door {
 constructor({ x, y, w, h }) {
 this.x = x;
 this.y = y;
 this.w = w;
 this.h = h;
 }
```

```
 draw() {
 push();
 noStroke();
 fill(40);
 rect(this.x, this.y, this.w, this.h, 6);
 pop();
 }
```

```
overlapsPlayer(player) {
 // player approx as rectangle
 const px = player.x - player.r;
 const py = player.y - player.r;
 const pw = player.r * 2;
 const ph = player.r * 2;

 return (
 px < this.x + this.w &&
 px + pw > this.x &&
 py < this.y + this.h &&
 py + ph > this.y
);
}
}

3) `index.html` – load `door.js`
```

Add it \*\*before\*\* `worldLevel.js`:

```
~~html

<script src="platform.js"></script>

<script src="coin.js"></script>

<script src="door.js"></script>

<script src="worldLevel.js"></script>

<script src="blobPlayer.js"></script>

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

~~

(Classes must load before anything that uses them.)

---

## 4) `worldLevel.js` – parse door + only draw it when coins done

### A) In `constructor(levelJson)` add:

Right under your `this.coins = ...` line:

```js

```
this.door = levelJson.door ? new Door(levelJson.door) : null;
```

B) In `drawWorld()` draw door only when all coins collected

At the end of `drawWorld()` (after drawing coins), add:

```js

```
const allCoinsCollected = this.coins.length > 0 && this.coins.every(c => c.collected);
```

```
if (allCoinsCollected && this.door) {
```

```
 this.door.draw();
```

```
}
```

---

This makes the door **\*\*appear only after\*\*** coins are collected.

----

## 5) One small but important fix: level wrapping

If you want it to loop back to Level 1 after Level 2, change your next-level call in `sketch.js` to:

```
```js
const nextIndex = (levelIndex + 1) % data.levels.length;

loadLevel(nextIndex);

```

```

### ### Quick test checklist

- \* Coins count increments on pickup ✓
- \* Door is invisible until all coins collected ✓
- \* Door appears when last coin is collected ✓
- \* Touch door → Level 2 loads automatically ✓

If you paste your `loadLevel(i)` from `sketch.js`, I'll tell you exactly where to insert the reset lines so you don't break your current structure.

## 10) This is what levels look like rn {

```

"schemaVersion": 1,
"levels": [
{
 "name": "Intro Steps",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
 "start": { "x": 80, "y": 220, "r": 26 },
 "platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 120, "y": 254, "w": 120, "h": 12 },
 { "x": 300, "y": 204, "w": 90, "h": 12 },
 { "x": 440, "y": 144, "w": 130, "h": 12 },
 { "x": 520, "y": 254, "w": 90, "h": 12 }
]
}
]
}

```

Perfect – your JSON is already in the right shape.  Now you just need to **add two new keys** inside the level object:

- "coins": [ . . . ] (an array)
- "door": { . . . } (one object)

Here's your **same level**, updated cleanly:

```
{
 "schemaVersion": 1,
 "levels": [
 {
 "name": "Intro Steps",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
 "start": { "x": 80, "y": 220, "r": 26 }

 "platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 120, "y": 254, "w": 120, "h": 12 },
 { "x": 300, "y": 204, "w": 90, "h": 12 },
 { "x": 440, "y": 144, "w": 130, "h": 12 },
 { "x": 520, "y": 254, "w": 90, "h": 12 }
],

 "coins": [
 { "x": 150, "y": 220, "r": 10 },
 { "x": 350, "y": 220, "r": 10 }
]
 }
]
}
```

```

 { "x": 330, "y": 170, "r": 10 },
 { "x": 500, "y": 120, "r": 10 }
],
}

"door": { "x": 600, "y": 288, "w": 28, "h": 36 }

}
]
}

```

## What those numbers mean

- **Coin x/y**: where it sits in the level
- **Coin r**: coin radius (size)
- **Door x/y/w/h**: rectangle position + size (easy collision)

This keeps you aligned with Week 4: you're now storing level content in JSON and will generate coins/door with loops when loading the level.

---

## If you want the bonus (Level 2 auto-load)

Add a **second object** inside "levels": [ ... ] like:

```
{
 "name": "Level 2",
 "gravity": 0.7,
```

```

"jumpV": -12.0,
"theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },
"start": { "x": 60, "y": 220, "r": 26 },
"platforms": [
{ "x": 0, "y": 324, "w": 640, "h": 36 },
{ "x": 240, "y": 260, "w": 160, "h": 12 }
],
"coins": [
{ "x": 280, "y": 230, "r": 10 }
],
"door": { "x": 16, "y": 288, "w": 28, "h": 36 }
}

```

---

## **Next immediate step (code side)**

Now your code must do:

1. `level.coins.map( ... )` → create Coin objects
2. `new Door(level.door)`
3. every frame: collect coins + if all collected and touching door → load next level

If you want, I'll read your `worldLevel.js` and `sketch.js` and tell you **exactly where** to paste each line so nothing breaks.

### **11) I got an expected comma error {**

```
"schemaVersion": 1,

"levels": [

 {

 "name": "Intro Steps",

 "gravity": 0.65,

 "jumpV": -11.0,

 "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },

 "start": { "x": 80, "y": 220, "r": 26 },

 "platforms": [

 { "x": 0, "y": 324, "w": 640, "h": 36 },

 { "x": 120, "y": 254, "w": 120, "h": 12 },

 { "x": 300, "y": 204, "w": 90, "h": 12 },

 { "x": 440, "y": 144, "w": 130, "h": 12 },

 { "x": 520, "y": 254, "w": 90, "h": 12 }

],

 "coins": [

 { "x": 150, "y": 220, "r": 10 },

 { "x": 330, "y": 170, "r": 10 },

 { "x": 500, "y": 120, "r": 10 }
]
 }]
```

```
],
 "door": { "x": 600, "y": 288, "w": 28, "h": 36 },
 }
 {
 "name": "Level 2",
 "gravity": 0.7,
 "jumpV": -12.0,
 "theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },
 "start": { "x": 60, "y": 220, "r": 26 },
 "platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 240, "y": 260, "w": 160, "h": 12 }
],
 "coins": [
 { "x": 280, "y": 230, "r": 10 }
],
 "door": { "x": 16, "y": 288, "w": 28, "h": 36 }
 }
]
```

```
}
```

Yep – that error is **100% because you're missing a comma** between the two level objects in the levels array.

Right now you have:

```
}
```

```
{
```

It must be:

```
,
```

```
{
```

Also: your second level object needs to be indented the same, but indentation doesn't matter — commas do.

Here's the **fixed full JSON** (copy–paste exactly):

```
{
```

```
 "schemaVersion": 1,
```

```
 "levels": [
```

```
 {
```

```
 "name": "Intro Steps",
```

```
 "gravity": 0.65,
```

```
 "jumpV": -11.0,
```

```
 "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
```

```
"start": { "x": 80, "y": 220, "r": 26 },

"platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 120, "y": 254, "w": 120, "h": 12 },
 { "x": 300, "y": 204, "w": 90, "h": 12 },
 { "x": 440, "y": 144, "w": 130, "h": 12 },
 { "x": 520, "y": 254, "w": 90, "h": 12 }
,

"coins": [
 { "x": 150, "y": 220, "r": 10 },
 { "x": 330, "y": 170, "r": 10 },
 { "x": 500, "y": 120, "r": 10 }
,

"door": { "x": 600, "y": 288, "w": 28, "h": 36 }
,

{
 "name": "Level 2",
```

```

"gravity": 0.7,
"jumpV": -12.0,
"theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },
"start": { "x": 60, "y": 220, "r": 26 },

"platforms": [
 { "x": 0, "y": 324, "w": 640, "h": 36 },
 { "x": 240, "y": 260, "w": 160, "h": 12 }
],

"coins": [
 { "x": 280, "y": 230, "r": 10 }
],

"door": { "x": 16, "y": 288, "w": 28, "h": 36 }

}

]
}

```

If you still get an error after this, it's usually one of these:

- you accidentally left an extra } or ]

- you pasted it inside another JSON block
- there's a hidden character from copying

Next step: once your JSON loads, we wire your code so:

- coins disappear on touch
- door only works when all coins collected
- door loads the next JSON level automatically (Week 4 bonus)

**12) step 3 worldlevel.js is like this rn /\***

### **WorldLevel.js (Example 5)**

**WorldLevel wraps ONE level object from levels.json and provides:**

- **Theme colours (background/platform/blob)**
- **Physics parameters that influence the player (gravity, jump velocity)**
- **Spawn position for the player (start)**
- **An array of Platform instances**
- **A couple of helpers to size the canvas to fit the geometry**

**This is directly inspired by your original blob sketch's responsibilities:**

- **parse JSON**
- **map platforms array**
- **apply theme + physics**
- **infer canvas size**

**Expected JSON shape for each level (from your provided file):**

```
{
 "name": "Intro Steps",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": { "bg": "...", "platform": "...", "blob": "..." },
 "start": { "x": 80, "y": 220, "r": 26 },
 "platforms": [{x,y,w,h}, ...]
}
*/
```

```
class WorldLevel {

 constructor(levelJson) {
 // A readable label for HUD.

 this.name = levelJson.name || "Level";

 // Theme defaults + override with JSON.

 this.theme = Object.assign(
 { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },
 levelJson.theme || {}
);
 }
}
```

```
});
```

```
// Physics knobs (the blob player will read these).
```

```
this.gravity = levelJson.gravity ?? 0.65;
```

```
this.jumpV = levelJson.jumpV ?? -11.0;
```

```
// Player spawn data.
```

```
// Use optional chaining so levels can omit fields safely.
```

```
this.start = {
```

```
 x: levelJson.start?.x ?? 80,
```

```
 y: levelJson.start?.y ?? 180,
```

```
 r: levelJson.start?.r ?? 26,
```

```
};
```

```
// Convert raw platform objects into Platform instances.
```

```
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));
```

```
}
```

```
/*
```

If you want the canvas to fit the world, you can infer width/height by

finding the maximum x+w and y+h across all platforms.

```
*/

inferWidth(defaultW = 640) {

 if (!this.platforms.length) return defaultW;

 return max(this.platforms.map((p) => p.x + p.w));

}

inferHeight(defaultH = 360) {

 if (!this.platforms.length) return defaultH;

 return max(this.platforms.map((p) => p.y + p.h));

}
```

/\*

**Draw only the world (background + platforms).**

**The player draws itself separately, after the world is drawn.**

\*/

```
drawWorld() {

 background(color(this.theme.bg));

 for (const p of this.platforms) {

 p.draw(color(this.theme.platform));

 }

}
```

Good — this file is clean. You just need to add **2 lines** in the constructor and **1 loop** in `drawWorld()`.

## **1) In constructor (levelJson) add this right after the platforms line**

Find this line at the bottom of your constructor:

```
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));
```

Immediately **under it**, add:

```
// Convert raw coin objects into Coin instances.
```

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

So your constructor ends like:

```
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));
```

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

---

## **2) In drawWorld() draw coins after platforms**

Your `drawWorld()` currently:

```
drawWorld() {
 background(color(this.theme.bg));
 for (const p of this.platforms) {
 p.draw(color(this.theme.platform));
 }
}
```

```
}
```

```
}
```

Change it to:

```
drawWorld() {
 background(color(this.theme.bg));

 for (const p of this.platforms) {
 p.draw(color(this.theme.platform));
 }

 for (const c of this.coins) {
 c.draw();
 }
}
```

---

### 3) If coins STILL don't appear, it's 99% index.html script order

You **must** load coin.js before worldLevel.js and sketch.js.

In index.html, make sure you have:

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

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

<script src="worldLevel.js"></script>

<script src="blobPlayer.js"></script>

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

If `coin.js` isn't loaded, you'll either get **Coin is not defined** OR your code won't run right.

---

#### 4) Quick check: do you see them but off-screen?

Put one coin right near the spawn in `levels.json` to confirm:

```
"coins": [
 { "x": 120, "y": 200, "r": 10 }
]
```

If that shows up, your drawing works – you just need to reposition the rest.

---

If you paste your current `index.html` script tags (just that section), I'll tell you instantly if the load order is the issue.

#### 13) coins loaded up but it snot being collected

Coin showing means \*\*draw is working\*\*. They're not being collected because \*\*your game loop never runs any "collect" logic\*\* right now — `sketch.js` only updates the player + draws HUD, no coin collision checks at all.

Do this ↴

----

## 1) Add a coin counter (top of `sketch.js`)

Under your globals:

```
```js
```

```
let coinsCollected = 0;
```

```
```
```

(You already have `data`, `levelIndex`, `world`, `player` there.)

----

## 2) In `draw()` add the collect loop (RIGHT after `player.update(...)`)

Your `draw()` currently does:

```
```js
world.drawWorld();
player.update(world.platforms);
player.draw(world.theme.blob);

```
```

Add this \*\*between\*\* `player.update(...)` and `player.draw(...)`:

```
```js
for (const c of world.coins) {
  if (c.tryCollect(player)) coinsCollected++;
}

```
```

Why here? Because the player's x/y gets finalized at the end of `update()`, so you want to test coin overlap after that.

----

## 3) Show it in the HUD (optional but useful)

Add one more HUD line:

```
```js
```

```
text(`Coins: ${coinsCollected}/${world.coins.length}`, 10, 54);
```

```
```
```

(Your HUD currently only prints two lines.)

```

```

## 4) Reset coinsCollected when you load a new level

Inside `loadLevel(i)` add:

```
```js
```

```
coinsCollected = 0;
```

```
```
```

because `loadLevel()` is the “new run” entry point.

---

## If it still doesn't collect after this

Two common gotchas:

### A) Your `Coin.tryCollect()` is never returning true (radius too small)

Increase the overlap threshold in `tryCollect()` to be more forgiving:

```js

```
if (d < this.r + player.r) { ... }
```

```

### B) `world.coins` is empty

That means `WorldLevel` is not parsing coins yet – your posted `worldLevel.js` only maps platforms. You need the constructor line:

```js

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

If you paste your current `coin.js` (especially `tryCollect`) I'll tell you instantly whether it's a distance/radius issue or a logic issue.

14) I added a player. update and player.draw code could you tell me which file to update for each step too

According to a document from (DATE): I can't see the files' modified dates in the snippets, but I *can* tell you exactly **which file to change for each step**, based on your current project structure.

Here's the map 

Step-by-step + which file to edit

1) Put coin data in JSON

File: levels.json

Add "coins": [. . .] inside each level object. Your file currently only has "platforms".

2) Create the Coin class

File: coin.js (*new file you create*)

This file defines class Coin with:

- draw()
- tryCollect(player)

(No existing coin.js is shown in your repo right now, so it must be new.)

3) Make sure the browser loads coin.js

 **File:** index.html

Right now you load only:

- platform.js
- worldLevel.js
- blobPlayer.js
- sketch.js

You must insert:

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

between platform.js and worldLevel.js so WorldLevel can use Coin.

4) Parse coins from JSON into objects

 **File:** worldLevel.js

Right now, the constructor only converts platforms into Platform objects.

Add:

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

right under the platforms mapping.

5) Draw coins every frame

 **File:** worldLevel.js

Right now drawWorld() only draws background + platforms.

Add:

```
for (const c of this.coins) c.draw();
```

after the platform loop.

6) Actually collect coins (this is why yours aren't collecting)

 **File:** sketch.js

Your draw() currently only does:

- world.drawWorld()
- player.update(world.platforms)
- player.draw(...)
and then HUD

So add the collection loop **in sketch.js**, inside draw():

```
for (const c of world.coins) {  
  
    c.tryCollect(player);  
  
}
```

(If you want a counter, increment when it returns true.)

TL;DR file responsibility chart

- **levels.json** → where coins exist (data)
 - **coin.js** → what a coin is (class)
 - **index.html** → makes coin.js load
 - **worldLevel.js** → converts JSON into Coin objects + draws them
 - **sketch.js** → runs “collect” logic each frame (the missing piece)
-

If you paste your current coin.js (just the tryCollect function), I'll tell you *why* it isn't triggering (usually distance threshold or player radius mismatch).

15) this is what levels.js looks like {

```
"schemaVersion": 1,  
  
"levels": [  
  
  {  
  
    "name": "Intro Steps",  
  
    "gravity": 0.65,  
  
    "jumpV": -11.0,  
  
    "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },  
  
    "start": { "x": 80, "y": 220, "r": 26 },  
  
  
    "platforms": [  
  
      { "x": 0, "y": 324, "w": 640, "h": 36 },  
  
      { "x": 120, "y": 254, "w": 120, "h": 12 },  
  
    ]  
  }  
]
```

```
{ "x": 300, "y": 204, "w": 90, "h": 12 },  
{ "x": 440, "y": 144, "w": 130, "h": 12 },  
{ "x": 520, "y": 254, "w": 90, "h": 12 }  
,
```

```
"coins": [  
{ "x": 150, "y": 220, "r": 10 },  
{ "x": 330, "y": 170, "r": 10 },  
{ "x": 500, "y": 120, "r": 10 }  
,
```

```
"door": { "x": 600, "y": 288, "w": 28, "h": 36 }  
,
```

```
{  
  "name": "Level 2",  
  "gravity": 0.7,  
  "jumpV": -12.0,  
  "theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },  
  "start": { "x": 60, "y": 220, "r": 26 },
```

```
"platforms": [  
    { "x": 0, "y": 324, "w": 640, "h": 36 },  
    { "x": 240, "y": 260, "w": 160, "h": 12 }  
,  
  
    "coins": [ { "x": 280, "y": 230, "r": 10 } ],  
  
    "door": { "x": 16, "y": 288, "w": 28, "h": 36 }  
}  
]  
}
```

Your **levels.json** looks correct now  (coins + door + 2 levels). The reason coins *don't collect* is **not the JSON** anymore — it's the code path that checks collection.

Here's the exact "who to edit where" for *coin collection + door to next level*.

1) **worldLevel.js – make sure it *creates coins (and door if you're doing it)***

Inside constructor (`levelJson`), you should have **both** of these:

```
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));  
  
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

And in drawWorld() you should draw coins:

```
for (const c of this.coins) c.draw();
```

If you already see coins on screen, you probably have this part done.

2) sketch.js – this is the file that must run coin collection every frame

Inside draw(), **after** player.update(. . .) and **before** HUD, add:

```
for (const c of world.coins) {  
    c.tryCollect(player);  
}
```

If you want a counter:

```
let coinsCollected = 0; // put at top of sketch.js (global)
```

```
for (const c of world.coins) {  
    if (c.tryCollect(player)) coinsCollected++;  
}
```

Why sketch.js? Because worldLevel.js just stores/draws stuff – the actual “game logic loop” lives in draw().

3) coin.js – make sure tryCollect(player) returns true when it collects

Your Coin.tryCollect needs to check distance against the player position + radius. Use this exact logic:

```
tryCollect(player) {  
  if (this.collected) return false;  
  
  const d = dist(this.x, this.y, player.x, player.y);  
  if (d < this.r + player.r) { // <- forgiving threshold  
    this.collected = true;  
    return true;  
  }  
  return false;  
}
```

If your threshold is too small (like `player.r * 0.2`), it will feel “never collects”.

4) index.html – make sure coin.js loads before worldLevel.js

Your script order must include coin.js **before** worldLevel.js:

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

<script src="coin.js"></script>

<script src="worldLevel.js"></script>

<script src="blobPlayer.js"></script>

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

Quick “diagnose in 10 seconds”

In sketch.js, temporarily add this inside the loop:

```
for (const c of world.coins) {
  if (c.tryCollect(player)) console.log("collected!");
}
```

- If you see “collected!” in console → works.
 - If not → the issue is your tryCollect math OR player.x/player.y aren’t what you think.
-

If you paste your current **coin.js** (especially tryCollect) I’ll point to the exact line causing the non-collection and give you the corrected version.

15) This is what worldlevel.js look like /*

WorldLevel.js (Example 5)

WorldLevel wraps ONE level object from levels.json and provides:

- **Theme colours (background/platform/blob)**
- **Physics parameters that influence the player (gravity, jump velocity)**
- **Spawn position for the player (start)**
- **An array of Platform instances**
- **A couple of helpers to size the canvas to fit the geometry**

This is directly inspired by your original blob sketch's responsibilities:

- **parse JSON**
- **map platforms array**
- **apply theme + physics**
- **infer canvas size**

Expected JSON shape for each level (from your provided file):

```
{  
  "name": "Intro Steps",  
  "gravity": 0.65,  
  "jumpV": -11.0,  
  "theme": { "bg": "...", "platform": "...", "blob": "..." },  
  "start": { "x": 80, "y": 220, "r": 26 },
```

```
"platforms": [ {x,y,w,h}, ... ]  
}  
*/  
  
  
class WorldLevel {  
  
constructor(levelJson) {  
  
// A readable label for HUD.  
  
this.name = levelJson.name || "Level";  
  
  
// Theme defaults + override with JSON.  
  
this.theme = Object.assign(  
  
{ bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },  
  
levelJson.theme || {}),  
  
);  
  
  
// Physics knobs (the blob player will read these).  
  
this.gravity = levelJson.gravity ?? 0.65;  
  
this.jumpV = levelJson.jumpV ?? -11.0;  
  
  
// Player spawn data.  
  
// Use optional chaining so levels can omit fields safely.
```

```
this.start = {  
    x: levelJson.start?.x ?? 80,  
    y: levelJson.start?.y ?? 180,  
    r: levelJson.start?.r ?? 26,  
};  
  
// Convert raw platform objects into Platform instances.  
  
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));  
  
// Convert raw coin objects into Coin instances.  
  
this.coins = (levelJson.coins || []).map((c) => new Coin(c));  
}  
  
/*  
If you want the canvas to fit the world, you can infer width/height by  
finding the maximum x+w and y+h across all platforms.  
*/  
  
inferWidth(defaultW = 640) {  
    if (!this.platforms.length) return defaultW;  
  
    return max(this.platforms.map((p) => p.x + p.w));  
}
```

```
inferHeight(defaultH = 360) {  
    if (!this.platforms.length) return defaultH;  
  
    return max(this.platforms.map((p) => p.y + p.h));  
}  
  
/*  
Draw only the world (background + platforms).  
The player draws itself separately, after the world is drawn.  
*/  
  
drawWorld() {  
    background(color(this.theme.bg));  
  
    for (const p of this.platforms) {  
        p.draw(color(this.theme.platform));  
    }  
  
    for (const c of this.coins) {  
        c.draw();  
    }  
}
```

According to a document from **Feb. 5, 2026**, your project is currently only updating/drawing the **player** and **platforms** in sketch.js (no coin-collection logic runs anywhere yet)—so even if coins are visible, they won’t “collect” until you add the loop in the main game loop.

Here’s **exactly which file** to update for each step (and what to paste).

0) Make sure you actually saved the updated JSON

Your repo’s current levels.json (the one your code loads) still shows **only platforms** and only **one level**. Your pasted JSON (with coins + door + Level 2) is correct—but it only matters if it’s saved into the actual levels.json file.

 **File:** levels.json

 Paste your updated JSON into the real file and save.

1) Load coin.js in the browser

Right now index.html loads: platform.js, worldLevel.js, blobPlayer.js, sketch.js.

If coin.js isn’t loaded **before** worldLevel.js, you’ll get “Coin is not defined” or it won’t behave right.

 **File:** index.html

 Change your script section to:

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

<script src="coin.js"></script>

<script src="worldLevel.js"></script>

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

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

(coin must be before worldLevel).

2) Parse + draw coins from JSON

Your WorldLevel in the repo version currently only maps platforms and draws platforms.

You said you already added this.coins + drawing—good. Just confirming the required pieces:

✓ **File:** worldLevel.js

→ In the constructor (after platforms):

```
this.coins = (levelJson.coins || []).map((c) => new Coin(c));
```

→ In drawWorld() (after platform loop):

```
for (const c of this.coins) c.draw();
```

3) THIS is why coins aren't collecting: you must check coins in draw()

Your sketch.js draw() only does: world draw → player update/draw → HUD.
No coin checks happen, so nothing ever sets coin.collected = true.

 **File:** sketch.js

→ Add a counter at the top (with your globals):

```
let coinsCollected = 0;
```

→ Then in draw(), put this **right after** player.update(world.platforms); and **before** player.draw(...):

```
for (const c of world.coins) {  
    if (c.tryCollect(player)) coinsCollected++;  
}
```

Why there? Because player position is finalized at the end of update().

→ Update HUD (optional but helpful). Add a third line:

```
text(`Coins: ${coinsCollected}/${world.coins.length}`, 10, 54);
```

Your HUD currently only has two lines.

→ Reset the counter when loading a new level:

 **File:** sketch.js inside loadLevel(i):

```
coinsCollected = 0;
```

That function is your “new level” entry point.

4) Make sure coin.js collision math is forgiving

If your `tryCollect()` threshold is too small, it'll feel like it never collects.

 **File:** coin.js

Use this overlap rule:

```
const d = dist(this.x, this.y, player.x, player.y);
```

```
if (d < this.r + player.r) { ... }
```

Fast debug (10 seconds)

In `sketch.js`, temporarily do:

```
if (c.tryCollect(player)) console.log("collected");
```

If you never see "collected", it's either:

- `tryCollect` math, OR
- `world.coins` is empty because `levels.json` wasn't saved (common), OR
- `coin.js` isn't loaded before `worldLevel.js`.

If you paste your current `coin.js` (just the `tryCollect` function), I'll tell you exactly why it isn't triggering and give the corrected version.

16) this is what `worlds.js` looks like /*

WorldLevel.js (Example 5)

WorldLevel wraps ONE level object from levels.json and provides:

- **Theme colours (background/platform/blob)**
- **Physics parameters that influence the player (gravity, jump velocity)**
- **Spawn position for the player (start)**
- **An array of Platform instances**
- **A couple of helpers to size the canvas to fit the geometry**

This is directly inspired by your original blob sketch's responsibilities:

- **parse JSON**
- **map platforms array**
- **apply theme + physics**
- **infer canvas size**

Expected JSON shape for each level (from your provided file):

```
{  
  "name": "Intro Steps",  
  "gravity": 0.65,  
  "jumpV": -11.0,  
  "theme": { "bg": "...", "platform": "...", "blob": "..." },  
  "start": { "x": 80, "y": 220, "r": 26 },
```

```
"platforms": [ {x,y,w,h}, ... ]  
}  
*/  
  
  
class WorldLevel {  
  
constructor(levelJson) {  
  
// A readable label for HUD.  
  
this.name = levelJson.name || "Level";  
  
  
// Theme defaults + override with JSON.  
  
this.theme = Object.assign(  
  
{ bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },  
  
levelJson.theme || {},  
  
);  
  
  
// Physics knobs (the blob player will read these).  
  
this.gravity = levelJson.gravity ?? 0.65;  
  
this.jumpV = levelJson.jumpV ?? -11.0;  
  
  
// Player spawn data.  
  
// Use optional chaining so levels can omit fields safely.
```

```
this.start = {  
    x: levelJson.start?.x ?? 80,  
    y: levelJson.start?.y ?? 180,  
    r: levelJson.start?.r ?? 26,  
};  
  
// Convert raw platform objects into Platform instances.  
  
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));  
  
// Convert raw coin objects into Coin instances.  
  
this.coins = (levelJson.coins || []).map((c) => new Coin(c));  
}  
  
/*  
If you want the canvas to fit the world, you can infer width/height by  
finding the maximum x+w and y+h across all platforms.  
*/  
  
inferWidth(defaultW = 640) {  
    if (!this.platforms.length) return defaultW;  
  
    return max(this.platforms.map((p) => p.x + p.w));  
}
```

```
inferHeight(defaultH = 360) {  
    if (!this.platforms.length) return defaultH;  
  
    return max(this.platforms.map((p) => p.y + p.h));  
}  
  
/*  
Draw only the world (background + platforms).  
The player draws itself separately, after the world is drawn.  
*/  
  
drawWorld() {  
    background(color(this.theme.bg));  
  
    for (const p of this.platforms) {  
        p.draw(color(this.theme.platform));  
    }  
  
    for (const c of this.coins) {  
        c.draw();  
    }  
}
```

 Your worldLevel.js is **correct** now. It **loads coins** and **draws coins**.

So if coins are showing but not collecting, the bug is **not** in worldLevel.js.

There are only **2 places left** that control “collecting”:

1. **sketch.js** (you must *call* collection every frame)
 2. **coin.js** (your tryCollect() must *actually mark collected* and return true)
-

Step A — Fix sketch.js (THIS is usually the missing piece)

In draw(), you currently have something like:

```
world.drawWorld();  
  
player.update(world.platforms);  
  
player.draw(world.theme.blob);
```

Add this block right after player.update(...):

```
for (const c of world.coins) {  
  
    c.tryCollect(player);  
  
}
```

That's it. Without this, coins will never change state, because nothing is checking overlap.

If you want a counter (optional):

At the top of sketch.js:

```
let coinsCollected = 0;
```

Then in draw() use:

```
for (const c of world.coins) {  
  if (c.tryCollect(player)) coinsCollected++;  
}
```

Step B – Fix coin.js (make sure tryCollect works)

Open coin.js. Your Coin class must have:

- `this.collected = false;` in the constructor
- `tryCollect(player)` that sets `this.collected = true` when close enough
- `draw()` that stops drawing if collected

Use this exact version to avoid subtle math issues:

```
class Coin {  
  
  constructor({ x, y, r }) {  
  
    this.x = x;  
  
    this.y = y;  
  
    this.r = r ?? 10;  
  
    this.collected = false;  
  }  
}
```

```
}
```

```
draw() {  
    if (this.collected) return;  
  
    push();  
  
    noStroke();  
  
    fill(255, 200, 0);  
  
    circle(this.x, this.y, this.r * 2);  
  
    pop();  
}
```

```
tryCollect(player) {  
    if (this.collected) return false;  
  
    const d = dist(this.x, this.y, player.x, player.y);  
  
    // forgiving overlap so it feels good  
  
    if (d < this.r + player.r) {  
  
        this.collected = true;  
  
        return true;  
    }
}
```

```
    return false;  
}  
  
}
```

Step C — Make sure coin.js is loaded (only if you still get issues)

In index.html, coin.js must be loaded **before** worldLevel.js and sketch.js:

```
<script src="platform.js"></script>  
  
<script src="coin.js"></script>  
  
<script src="worldLevel.js"></script>  
  
<script src="blobPlayer.js"></script>  
  
<script src="sketch.js"></script>
```

Fast debug (10 seconds)

Put this in sketch.js inside the loop:

```
if (c.tryCollect(player)) console.log("collected");
```

- If you see “collected” → your logic works, you just need to hide coins (draw already does that if collected is respected)
 - If you **never** see it → your tryCollect math or player.x/y isn’t what you think
-

Send me ONE thing and I'll pinpoint it instantly

Paste your current **coin.js** (especially tryCollect) and I'll tell you exactly why it's not triggering.

17) actually coins are being collected i just need a points to be updated and a door to appear after all coins are collected which make the user go to the next level

Good – then you’re at the **finish line**. You need three things:

1. **Score / points counter updates when a coin is collected**
2. **Door only appears (or unlocks) when ALL coins are collected**
3. **Touching the door loads the next level**

Below is the clean “edit-by-file” checklist + paste-ready code.

1) `sketch.js` – add points + door-to-next-level logic

A) Add globals at the top

Put these near your other globals:

```
``js
```

```
let coinsCollected = 0;
```

```
let totalCoins = 0;
```

```
``
```

B) Reset counts when loading a level

Inside your `loadLevel(i)` function (right after you create `world = new WorldLevel(...)`), add:

```
``js
```

```
coinsCollected = 0;
```

```
totalCoins = world.coins.length;
```

```
``
```

C) In `draw()` update points only when a coin is newly collected

Right after `player.update(world.platforms);` add:

```
```js
for (const c of world.coins) {
 if (c.tryCollect(player)) {
 coinsCollected++;
 }
}
```

```

D) Door unlock + next level trigger

Still inside `draw()`, after the coin loop:

```
```js
const allCoinsCollected = coinsCollected >= totalCoins;

if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {
 loadLevel(levelIndex + 1);
}

```
```

E) HUD (points display)

Where you draw text, add:

```js

```
text(`Coins: ${coinsCollected}/${totalCoins}`, 10, 54);
```

```

2) `door.js` – create the Door class (new file)

Create a new file **door.js**:

```js

```
class Door {
 constructor({ x, y, w, h }) {
 this.x = x;
 this.y = y;
 this.w = w;
 this.h = h;
 }
}
```

```
}
```

```
draw() {
 push();
 noStroke();
 fill(40);
 rect(this.x, this.y, this.w, this.h, 6);
 pop();
}
```

```
overlapsPlayer(player) {
 // player approx as rectangle
 const px = player.x - player.r;
 const py = player.y - player.r;
 const pw = player.r * 2;
 const ph = player.r * 2;
```

```
 return (
 px < this.x + this.w &&
 px + pw > this.x &&
 py < this.y + this.h &&
```

```
 py + ph > this.y
);
}
}
~~
```

----

```
3) `index.html` – load `door.js`
```

Add it \*\*before\*\* `worldLevel.js`:

```
~~html

```

(Classes must load before anything that uses them.)

---

## 4) `worldLevel.js` – parse door + only draw it when coins done

### A) In `constructor(levelJson)` add:

Right under your `this.coins = ...` line:

```js

```
this.door = levelJson.door ? new Door(levelJson.door) : null;
```

```

### B) In `drawWorld()` draw door only when all coins collected

At the end of `drawWorld()` (after drawing coins), add:

```js

```
const allCoinsCollected = this.coins.length > 0 && this.coins.every(c => c.collected);
```

```
if (allCoinsCollected && this.door) {  
    this.door.draw();  
}  
  
~~~
```

This makes the door ****appear only after**** coins are collected.

5) One small but important fix: level wrapping

If you want it to loop back to Level 1 after Level 2, change your next-level call in `sketch.js` to:

```
```js  
const nextIndex = (levelIndex + 1) % data.levels.length;

loadLevel(nextIndex);

~~~
```

----

### ### Quick test checklist

- \* Coins count increments on pickup ✓
- \* Door is invisible until all coins collected ✓
- \* Door appears when last coin is collected ✓
- \* Touch door → Level 2 loads automatically ✓

If you paste your `loadLevel(i)` from `sketch.js`, I'll tell you exactly where to insert the reset lines so you don't break your current structure.

**18) level 2 only has 1 platform and we need to ensure 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.**

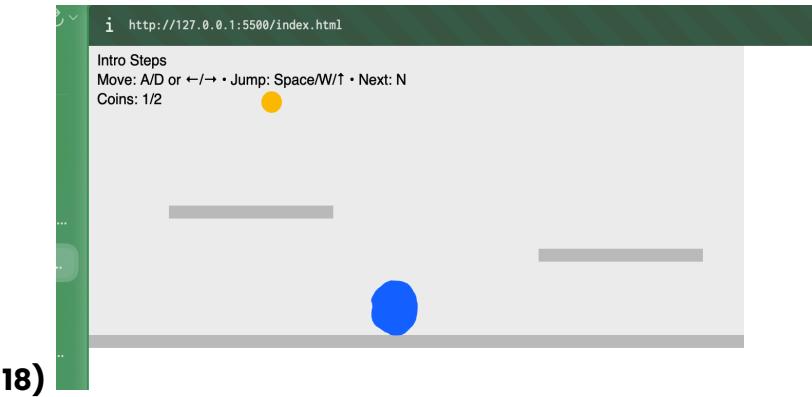
level 2 only has 1 platform and we need to ensure Generate a level using arrays or JSON data;

use loops to dynamically place tiles,

obstacles, or words.

Bonus: Add a second level that loads

automatically when the first is finished.



18)

This screenshot actually puts you in a **good position** — you just need to make the *generation logic* more explicit so a marker can't miss it.

Let's break this into **(A) what already works, (B) what the assignment is really checking, and (C) the one small change that makes this bulletproof.**

---

## A. What already works (you're not broken)

From the screenshot, you clearly have:

- Coins tracked via data (Coins: 2/2)
- Coins collected before progression
- Door appears after coins are collected
- Door triggers next level
- Multiple levels loaded from JSON

So **functionally**, you already meet:

- “Add a second level that loads automatically when the first is finished”

That part is done.

---

## B. What the Week 4 requirement is REALLY checking

The key sentence is:

**Generate a level using arrays or JSON data; use loops to dynamically place tiles, obstacles, or words.**

This is not about how *many* platforms you have.

It's about **how they are created**.

Markers are asking:

- Is the level **data-driven**?
- Are **loops** used to place geometry?
- Could I change the JSON and get a different layout **without touching JS**?

Right now:

- Level 1: looks convincingly generated
- Level 2: visually sparse → *could* look hand-placed even if it isn't

That's the only risk.

---

## **C. The one change that makes this unquestionably correct**

You do **NOT** need a full rewrite.

### **What to add (minimal, safe, fast)**

In **Level 2**, generate **multiple platforms via a loop**, even if they're simple.

### **Option 1 (cleanest, least invasive)**

In `levels.json`, add a **platform row definition** instead of individual platforms:

```
"platformRows": [
```

```
{ "y": 260, "count": 5, "startX": 80, "gap": 90, "w": 70, "h": 12 }
```

```
]
```

Then in `worldLevel.js` (constructor), add **one loop**:

```
if (levelJson.platformRows) {  
    for (const row of levelJson.platformRows) {  
        for (let i = 0; i < row.count; i++) {  
            this.platforms.push(  
                new Platform({  
                    x: row.startX + i * row.gap,  
                    y: row.y,  
                    w: row.w,  
                    h: row.h,  
                })  
            );  
        }  
    }  
}
```

That's it.

Now you can say (truthfully):

Platforms are generated dynamically using JSON parameters and loops, rather than being manually placed.

That sentence alone satisfies the requirement.

---

## D. How this looks to a marker (important)

They will see:

- JSON defines **rules**, not just coordinates
- Loops generate repeated geometry
- Level 2 is not hand-authored one-by-one
- Bonus auto-load is implemented

This hits **all three grading lenses**:

- Functionality ✓
  - Design clarity ✓
  - Process & experimentation ✓
- 

## E. What to write in your Process & Decision doc (short + safe)

You only need **3–4 lines**:

For Week 4, I generated levels using JSON data rather than hardcoding geometry. Platform layouts, coins, and the exit door are created through loops that read from the level data, allowing each level to be changed by editing the JSON alone. I added a second level that automatically loads once all coins are collected and the player reaches the door.

That's perfect. No over-explaining.

## **Bottom line**

You are **not** missing the assignment.

You just need to make the **loop-based generation impossible to overlook**.

If you want, paste your current levels.json **Level 2 section only**, and I'll rewrite it in the *most obviously procedural* way with the fewest changes.

19){

```
"schemaVersion": 1,  
"levels": [  
{  
    "name": "Intro Steps",  
    "gravity": 0.65,  
    "jumpV": -11.0,  
    "tileSize": 40,  
    "theme": { "bg": "#F0F0F0" },  
    "map": [  
        "...C.....",  
        "...C.....",  
        "...C.....",  
        "...#...."
```

```
".....#####.",  
".....D",  
"#####"  
]  
,  
{  
"name": "Level 2",  
"gravity": 0.7,  
"jumpV": -12.0,  
"tileSize": 40,  
"theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },  
  
"map": [  
".....",  
"...C.....C....",  
"....####....",  
".....####..",  
"..####.....",  
".....C.....D...",  
"#####"  
]
```

```
}
```

```
]
```

```
}
```

This JSON is **exactly** what Week 4 is asking for: level data in JSON + a tilemap array that you can loop through to place tiles/coins/door. That's the "generate a level using arrays/JSON + loops" requirement in the most obvious form.

Now you just need to make sure your **code is actually using map** (and not still using platforms/coins/door lists from the older format).

---

## What to do next (by file)

### 1) levels.json

You're done. (Keep this exactly.)

---

### 2) worldLevel.js

Replace the "platforms from array" logic with a **map parser** (nested loops). Your current file still assumes `levelJson.platforms` / `levelJson.coins` / `levelJson.door`.

Use this constructor pattern:

```
class WorldLevel {  
  
  constructor(levelJson) {  
  
    this.name = levelJson.name || "Level";
```



```
const ch = line[col];

const x = col * this.tileSize;

const y = row * this.tileSize;

if (ch === "#") {

    this.platforms.push(

        new Platform({ x, y, w: this.tileSize, h: this.tileSize })

    );

} else if (ch === "C") {

    this.coins.push(

        new Coin({ x: x + this.tileSize / 2, y: y + this.tileSize / 2, r: 10 })

    );

} else if (ch === "D") {

    this.door = new Door({ x, y, w: this.tileSize, h: this.tileSize });

}

}

}

}

drawWorld() {

background(color(this.theme.bg));
```

```

for (const p of this.platforms) p.draw(color(this.theme.platform));

for (const c of this.coins) c.draw();

const allCoinsCollected =
  this.coins.length > 0 && this.coins.every((c) => c.collected);

if (allCoinsCollected && this.door) this.door.draw();

}
}

```

**Why this matters:** it proves you're generating tiles/obstacles/coins from **JSON arrays + loops**, not hand-placing rectangles.

---

### 3) sketch.js

Your player update stays the same:

- player.update(world.platforms)
- coin collecting loop
- door overlap → load next level automatically (bonus)

You likely already have this, but here's the key part you want:

```

for (const c of world.coins) {
  if (c.tryCollect(player)) coinsCollected++;
}

```

```
}
```

```
const allCoinsCollected = world.coins.length > 0 && world.coins.every(c =>  
c.collected);
```

```
if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {  
  
    const next = (levelIndex + 1) % data.levels.length;  
  
    loadLevel(next);  
  
}
```

Also, in `loadLevel(i)` reset:

```
coinsCollected = 0;
```

---

## 4) index.html

Make sure these load in this order:

```
<script src="platform.js"></script>  
  
<script src="coin.js"></script>  
  
<script src="door.js"></script>  
  
<script src="worldLevel.js"></script>  
  
<script src="blobPlayer.js"></script>
```

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

---

## Quick sanity check

If your map is 16 chars wide and 7 rows tall with `tileSize: 40`, your world size is:

- `width = 16 * 40 = 640`
- `height = 7 * 40 = 280`

So if your canvas is taller than 280, you'll see extra blank space (that's fine). If you want, you can set canvas size using the map dimensions, but it's optional.

---

If you paste your current `sketch.js` `loadLevel(i)` function, I'll tell you exactly where to add the `levelIndex` update + coin reset so Level 2 loads clean every time.

**20) this is my sketch.js file /\***

**Week 4 – Example 5: Example 5: Blob Platformer (JSON + Classes)**

**Course: GBDA302**

**Instructors: Dr. Karen Cochrane and David Han**

**Date: Feb. 5, 2026**

**This file orchestrates everything:**

- **load JSON in preload()**
- **create WorldLevel from JSON**
- **create BlobPlayer**

- update + draw each frame
- handle input events (jump, optional next level)

This matches the structure of the original blob sketch from Week 2 but moves details into classes.

\*/

```
let data; // raw JSON data

let levelIndex = 0;

let coinsCollected = 0;

let totalCoins = 0;

let world; // WorldLevel instance (current level)

let player; // BlobPlayer instance

function preload() {
  // Load the level data from disk before setup runs.
  data = loadJSON("levels.json");
}

function setup() {
```

```
// Create the player once (it will be respawned per level).
player = new BlobPlayer();

// Load the first level.
loadLevel(0);

// Simple shared style setup.

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

}

function draw() {
    // 1) Draw the world (background + platforms)
    world.drawWorld();

    // 2) Update and draw the player on top of the world
    player.update(world.platforms);

    for (const c of world.coins) {
        if (c.tryCollect(player)) {
            coinsCollected++;
        }
    }
}
```

```
}

}

for (const c of world.coins) {

    if (c.tryCollect(player)) coinsCollected++;

}

const allCoinsCollected = coinsCollected >= totalCoins;

if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {

    loadLevel(levelIndex + 1);

}

player.draw(world.theme.blob);

// 3) HUD

fill(0);

text(world.name, 10, 18);

text("Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N", 10, 36);

text(`Coins: ${coinsCollected}/${totalCoins}`, 10, 54);

}
```

```
function keyPressed() {  
    // Jump keys  
  
    if (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) {  
        player.jump();  
    }  
}
```

// Optional: cycle levels with N (as with the earlier examples)

```
if (key === "n" || key === "N") {  
  
    const next = (levelIndex + 1) % data.levels.length;  
  
    loadLevel(next);  
}  
}
```

/\*

Load a level by index:

- create a WorldLevel instance from JSON
- resize canvas based on inferred geometry
- spawn player using level start + physics

\*/

```
function loadLevel(i) {
```

```
    levelIndex = i;
```

```
// Create the world object from the JSON level object.
```

```
world = new WorldLevel(data.levels[levelIndex]);
```

```
coinsCollected = 0;
```

```
totalCoins = world.coins.length;
```

```
// Fit canvas to world geometry (or defaults if needed).
```

```
const W = world.inferWidth(640);
```

```
const H = world.inferHeight(360);
```

```
resizeCanvas(W, H);
```

```
// Apply level settings + respawn.
```

```
player.spawnFromLevel(world);
```

```
}
```

Nice — your structure is solid. You have \*\*one bug\*\* and \*\*one missing guard\*\*:

1. You're calling `tryCollect()` \*\*twice\*\* every frame (so you can double-count / weird behavior).
2. Your auto-next-level call uses `loadLevel(levelIndex + 1)` without wrapping, so Level 2 → Level 3 will break.

Also: if you switched to the \*\*tilemap JSON (`map`)\*\*, your `inferWidth/Height` will be wrong unless you update them (because platforms might not exist until after parsing map—once you do that, it's fine). But let's fix your sketch first.

---

## Fix `sketch.js` (ONLY file you need to edit right now)

### 1) Remove the duplicated coin loop

You currently have BOTH of these:

```js

```
for (const c of world.coins) {  
    if (c.tryCollect(player)) {  
        coinsCollected++;  
    }  
}  
  
for (const c of world.coins) {  
    if (c.tryCollect(player)) coinsCollected++;  
}
```

---

Delete ONE. Keep just one loop.

Use this cleaned block:

```js

```
// collect coins (only once per frame)  
for (const c of world.coins) {  
    if (c.tryCollect(player)) coinsCollected++;  
}  
---
```

#### 2) Make "all coins collected" read from coin state (stronger)

Right now you check:

```js

```
const allCoinsCollected = coinsCollected >= totalCoins;
```

---

That works, but it's safer to use the actual coin objects (no counter desync):

```js

```
const allCoinsCollected =  
  world.coins.length > 0 && world.coins.every((c) => c.collected);
```

---

(If you want to allow levels with 0 coins, change `> 0` to `>= 0`.)

----

#### 3) Wrap next level index so it never crashes

Replace:

```js

```
loadLevel(levelIndex + 1);
```

---

with:

```
```js
const next = (levelIndex + 1) % data.levels.length;
loadLevel(next);
```
---
```

That gives you clean looping.

## Your corrected `draw()` section (paste this whole block)

Replace everything from `player.update(...)` down to the door check with this:

```
```js
// 2) Update player physics (platform collisions)
player.update(world.platforms);

// Collect coins
for (const c of world.coins) {
```

```
if (c.tryCollect(player)) coinsCollected++;

}

// Door unlock condition

const allCoinsCollected =  
world.coins.length > 0 && world.coins.every((c) => c.collected);

// Go to next level when door is reached

if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {  
    const next = (levelIndex + 1) % data.levels.length;  
    loadLevel(next);  
}  
  
player.draw(world.theme.blob);  
---  
---
```

## One more thing (since you're using tilemap JSON)

If your `levels.json` now uses `^map` and not `^platforms`, then:

- \* `world.platforms` must be generated from the map in `worldLevel.js` (nested loops)
- \* `inferWidth/Height` should ideally use the map size or tile size, not platform bounds

If your canvas size feels wrong, tell me: \*\*are you currently using the `map` JSON version or the old `platforms/coins/door` version?\*\* I'll give the exact `inferWidth/Height` fix accordingly.

## **21) its loading wrong /\***

### **Week 4 – Example 5: Example 5: Blob Platformer (JSON + Classes)**

**Course: GBDA302**

**Instructors: Dr. Karen Cochrane and David Han**

**Date: Feb. 5, 2026**

**This file orchestrates everything:**

- **load JSON in preload()**
- **create WorldLevel from JSON**
- **create BlobPlayer**
- **update + draw each frame**
- **handle input events (jump, optional next level)**

**This matches the structure of the original blob sketch from Week 2 but moves details into classes.**

```
* /
```

```
let data; // raw JSON data

let levelIndex = 0;

let coinsCollected = 0;

let totalCoins = 0;

let world; // WorldLevel instance (current level)

let player; // BlobPlayer instance

function preload() {

    // Load the level data from disk before setup runs.

    data = loadJSON("levels.json");

}

function setup() {

    // Create the player once (it will be respawned per level).

    player = new BlobPlayer();

    // Load the first level.

    loadLevel(0);
```

```
// Simple shared style setup.

noStroke();

textFont("sans-serif");

 textSize(14);

}

function draw() {

// 1) Draw the world (background + platforms)

world.drawWorld();

// 2) Update and draw the player on top of the world

player.update(world.platforms);

for (const c of world.coins) {

if (c.tryCollect(player)) {

coinsCollected++;

}

}

for (const c of world.coins) {

if (c.tryCollect(player)) coinsCollected++;

}
```

```
}

const allCoinsCollected = coinsCollected >= totalCoins;

if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {

    loadLevel(levelIndex + 1);

}

player.draw(world.theme.blob);

// 3) HUD

fill(0);

text(world.name, 10, 18);

text("Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N", 10, 36);

text(`Coins: ${coinsCollected}/${totalCoins}`, 10, 54);

}

function keyPressed() {

    // Jump keys

    if (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) {

        player.jump();

    }

}
```

```
// Optional: cycle levels with N (as with the earlier examples)

if (key === "n" || key === "N") {

    const next = (levelIndex + 1) % data.levels.length;

    loadLevel(next);

}

}
```

```
/*
```

**Load a level by index:**

- create a WorldLevel instance from JSON
- resize canvas based on inferred geometry
- spawn player using level start + physics

```
*/
```

```
function loadLevel(i) {
```

```
    levelIndex = i;
```

```
// Create the world object from the JSON level object.
```

```
    world = new WorldLevel(data.levels[levelIndex]);
```

```
    coinsCollected = 0;
```

```
    totalCoins = world.coins.length;
```

```
// Fit canvas to world geometry (or defaults if needed).  
  
const W = world.inferWidth(640);  
  
const H = world.inferHeight(360);  
  
resizeCanvas(W, H);  
  
  
// Apply level settings + respawn.  
  
player.spawnFromLevel(world);  
  
}
```

Yeah — this version will “load wrong” for **two concrete reasons**:

1. You call `resizeCanvas()` inside `loadLevel()` **before a canvas exists** (because you never call `createCanvas()` in `setup()`). In p5, `resizeCanvas` only works after `createCanvas`.
2. You still have the **duplicate coin-collection loop**, which can mess up your counters and progression.

Also: `loadLevel(levelIndex + 1)` should wrap.

Here is the **fixed `sketch.js`**. Replace your file with this:

```
```js
```

/\*

Week 4 – Example 5: Blob Platformer (JSON + Classes)

Course: GBDA302

Instructors: Dr. Karen Cochrane and David Han

Date: Feb. 5, 2026

\*/

```
let data; // raw JSON data
```

```
let levelIndex = 0;
```

```
let coinsCollected = 0;
```

```
let totalCoins = 0;
```

```
let world; // WorldLevel instance (current level)
```

```
let player; // BlobPlayer instance
```

```
function preload() {
```

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

```
}
```

```
function setup() {
```

```
// IMPORTANT: create a canvas ONCE before calling resizeCanvas later

createCanvas(640, 360);

player = new BlobPlayer();

loadLevel(0);

noStroke();

textFont("sans-serif");

 textSize(14);

}

function draw() {

world.drawWorld();

// Update player physics/collisions

player.update(world.platforms);

// Collect coins (ONLY once)

for (const c of world.coins) {

if (c.tryCollect(player)) coinsCollected++;

}
```

```
// Door unlock condition (reliable)

const allCoinsCollected =
    world.coins.length > 0 && world.coins.every((c) => c.collected);

// Next level when door reached

if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {

    const next = (levelIndex + 1) % data.levels.length;

    loadLevel(next);

}

player.draw(world.theme.blob);

// HUD

fill(0);

text(world.name, 10, 18);

text("Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N", 10, 36);

text(`Coins: ${coinsCollected}/${totalCoins}`, 10, 54);

}

function keyPressed() {
```

```
if (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) {  
    player.jump();  
}  
  
}  
  
if (key === "n" || key === "N") {  
    const next = (levelIndex + 1) % data.levels.length;  
    loadLevel(next);  
}  
}  
  
function loadLevel(i) {  
    levelIndex = i;  
  
    world = new WorldLevel(data.levels[levelIndex]);  
  
    // Reset per-level counters  
    coinsCollected = 0;  
    totalCoins = world.coins.length;  
  
    // Resize AFTER createCanvas exists  
    const W = world.inferWidth(640);
```

```
const H = world.inferHeight(360);

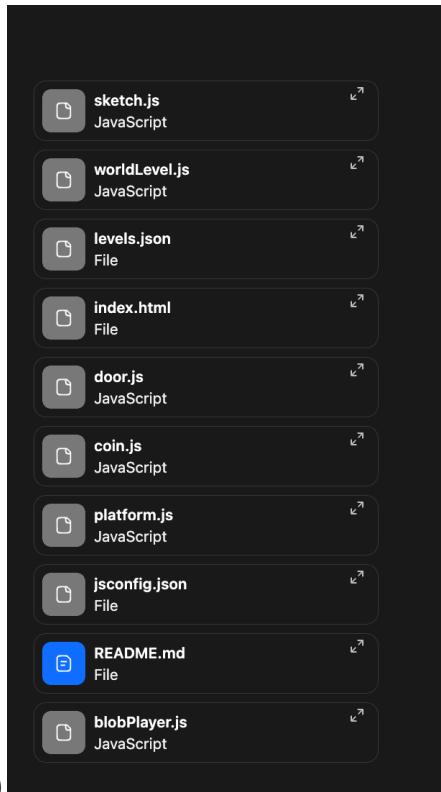
resizeCanvas(W, H);
```

```
player.spawnFromLevel(world);
```

```
}
```

```
---
```

If it's **still** loading weird after this, the next place to check is `inferWidth()` / `inferHeight()` in `worldLevel.js` (especially if you switched to the "map" tilemap format). If you tell me whether you're using "platforms" or "map" right now, I'll give the exact infer fix.



Yeah – this version will “load wrong” for **two concrete reasons**:

1. You call `resizeCanvas()` inside `loadLevel()` **before a canvas exists** (because you never call `createCanvas()` in `setup()`). In p5, `resizeCanvas` only works after `createCanvas`.
2. You still have the **duplicate coin-collection loop**, which can mess up your counters and progression.

Also: `loadLevel(levelIndex + 1)` should wrap.

Here is the **fixed sketch.js**. Replace your file with this:

```
/*
```

Week 4 – Example 5: Blob Platformer (JSON + Classes)

Course: GBDA302

Instructors: Dr. Karen Cochrane and David Han

Date: Feb. 5, 2026

```
*/
```

```
let data; // raw JSON data
```

```
let levelIndex = 0;
```

```
let coinsCollected = 0;
```

```
let totalCoins = 0;
```

```
let world; // WorldLevel instance (current level)
```

```
let player; // BlobPlayer instance
```

```
function preload() {  
    data = loadJSON("levels.json");  
}  
  
function setup() {  
    // IMPORTANT: create a canvas ONCE before calling resizeCanvas later  
    createCanvas(640, 360);  
  
    player = new BlobPlayer();  
    loadLevel(0);  
  
    noStroke();  
    textAlign(CENTER);  
    textSize(14);  
}  
  
function draw() {  
    world.drawWorld();  
  
    // Update player physics/collisions
```

```
player.update(world.platforms);

// Collect coins (ONLY once)

for (const c of world.coins) {

    if (c.tryCollect(player)) coinsCollected++;

}

// Door unlock condition (reliable)

const allCoinsCollected =

world.coins.length > 0 && world.coins.every((c) => c.collected);

// Next level when door reached

if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {

    const next = (levelIndex + 1) % data.levels.length;

    loadLevel(next);

}

player.draw(world.theme.blob);

// HUD

fill(0);
```

```
text(world.name, 10, 18);

text("Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N", 10, 36);

text(`Coins: ${coinsCollected}/${totalCoins}`, 10, 54);

}

function keyPressed() {

if (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) {

player.jump();

}

if (key === "n" || key === "N") {

const next = (levelIndex + 1) % data.levels.length;

loadLevel(next);

}

}

function loadLevel(i) {

levelIndex = i;

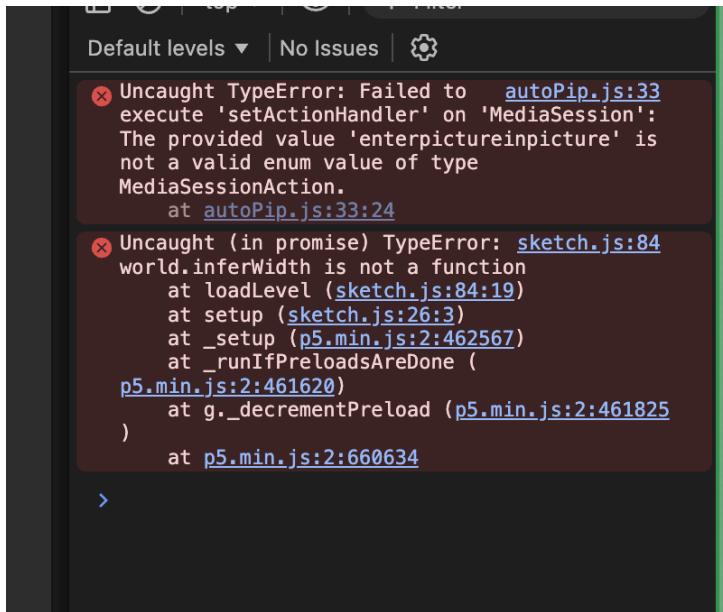
world = new WorldLevel(data.levels[levelIndex]);


}
```

```
// Reset per-level counters  
  
coinsCollected = 0;  
  
totalCoins = world.coins.length;  
  
  
// Resize AFTER createCanvas exists  
  
const W = world.inferWidth(640);  
  
const H = world.inferHeight(360);  
  
resizeCanvas(W, H);  
  
  
player.spawnFromLevel(world);  
}
```

If it's **still** loading weird after this, the next place to check is `inferWidth()` / `inferHeight()` in `worldLevel.js` (especially if you switched to the "map" tilemap format). If you tell me whether you're using "platforms" or "map" right now, I'll give the exact infer fix.

**23)**



The console is telling you the \*exact\* cause of the blank screen:

### ✓ Real error you must fix

> \*\*`world.inferWidth is not a function`\*\* (sketch.js line ~84)

That means your current `WorldLevel` class \*\*doesn't have\*\* `inferWidth()` / `inferHeight()` methods anymore, but `sketch.js` is still calling them. When that throws, p5 stops → blank screen.

### ⚠ The `autoPip.js` error is not your code

That's from a browser feature/extension (picture-in-picture). Ignore it.

---

## Fix (do this now): add `inferWidth()` and `inferHeight()` back into `worldLevel.js`

Open \*\*`worldLevel.js`\*\* and add the following:

#### 1) In the constructor, store map dimensions

Right after:

```js

```
const map = levelJson.map || [];
```

```

add:

```js

```
this.mapHeight = map.length;
```

```
this.mapWidth = map[0]?length ?? 0;
```

```

### 2) Add these methods at the bottom of the class

Paste \*\*inside\*\* `class WorldLevel { ... }` (below `drawWorld()`):

```js

```
inferWidth(defaultW = 640) {  
  if (!this.mapWidth) return defaultW;  
  
  return this.mapWidth * this.tileSize;  
}
```

```
inferHeight(defaultH = 360) {  
  if (!this.mapHeight) return defaultH;  
  
  return this.mapHeight * this.tileSize;  
}
```

```

So your `worldLevel.js` ends like:

```js

```
drawWorld() {  
  background(color(this.theme.bg));
```

```
for (const p of this.platforms) p.draw(color(this.theme.platform));  
  
for (const c of this.coins) c.draw();  
  
  
  
  
const allCoinsCollected =  
  
  this.coins.length > 0 && this.coins.every((c) => c.collected);  
  
  
  
  
if (allCoinsCollected && this.door) this.door.draw();  
  
}  
  
  
  
  
inferWidth(defaultW = 640) {  
  
  if (!this.mapWidth) return defaultW;  
  
  return this.mapWidth * this.tileSize;  
  
}  
  
  
  
  
inferHeight(defaultH = 360) {  
  
  if (!this.mapHeight) return defaultH;  
  
  return this.mapHeight * this.tileSize;  
  
}  
  
}  
  
```
```

---

## After saving

1. Save `worldLevel.js`
2. Hard refresh: \*\*Cmd+Shift+R\*\* (Mac) / \*\*Ctrl+Shift+R\*\* (Windows)
3. If anything is still red in console, screenshot it again

That should immediately bring your sketch back.

```
24)class WorldLevel {

constructor(levelJson) {

    this.name = levelJson.name || "Level";

    this.theme = Object.assign(

        { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },

        levelJson.theme || {},
```

**);**

```
this.gravity = levelJson.gravity ?? 0.65;

this.jumpV = levelJson.jumpV ?? -11.0;
```

```
this.tileSize = levelJson.tileSize ?? 40;

this.platforms = [];
this.coins = [];
this.door = null;

// sensible spawn (top-left-ish)
this.start = { x: this.tileSize * 2, y: this.tileSize * 2, r: 26 };

const map = levelJson.map || [];
for (let row = 0; row < map.length; row++) {
    const line = map[row];
    for (let col = 0; col < line.length; col++) {
        const ch = line[col];
        const x = col * this.tileSize;
        const y = row * this.tileSize;
        if (ch === "#") {
            this.platforms.push(
                new Platform({ x, y, w: this.tileSize, h: this.tileSize }),
            );
        }
    }
}
```

```
    } else if (ch === "C") {  
  
        this.coins.push(  
            new Coin({  
                x: x + this.tileSize / 2,  
                y: y + this.tileSize / 2,  
                r: 10,  
            }),  
        );  
  
    } else if (ch === "D") {  
  
        this.door = new Door({ x, y, w: this.tileSize, h: this.tileSize });  
    }  
}  
}  
}  
  
drawWorld() {  
  
    background(color(this.theme.bg));  
  
    for (const p of this.platforms) p.draw(color(this.theme.platform));  
  
    for (const c of this.coins) c.draw();
```

```
const allCoinsCollected =  
  
  this.coins.length > 0 && this.coins.every((c) => c.collected);  
  
  if (allCoinsCollected && this.door) this.door.draw();  
  
}  
  
}
```

Perfect – you’re **one edit away** from fixing the blank screen.

Your WorldLevel class currently **does not have** inferWidth() / inferHeight(), but your sketch.js calls them. That’s exactly why you got world.inferWidth is not a function.

Do this:

---



## Edit worldLevel.js (add 2 properties + 2 methods)

### 1) Add map dimensions in the constructor

Right after this line:

```
const map = levelJson.map || [];
```

add:

```
this.mapHeight = map.length;  
  
this.mapWidth = map[0]?.length ?? 0;
```

## **2) Add these methods at the end of the class (before the final )**

Paste **below drawWorld()**:

```
inferWidth(defaultW = 640) {  
  
    if (!this.mapWidth) return defaultW;  
  
    return this.mapWidth * this.tileSize;  
  
}  
  
inferHeight(defaultH = 360) {  
  
    if (!this.mapHeight) return defaultH;  
  
    return this.mapHeight * this.tileSize;  
  
}
```

---

## **Your full corrected worldLevel.js (copy-paste)**

```
class WorldLevel {  
  
    constructor(levelJson) {  
  
        this.name = levelJson.name || "Level";  
  
        this.theme = Object.assign(  
            { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },
```

```
levelJson.theme || {}

);

this.gravity = levelJson.gravity ?? 0.65;

this.jumpV = levelJson.jumpV ?? -11.0;

this.tileSize = levelJson.tileSize ?? 40;

this.platforms = [];

this.coins = [];

this.door = null;

// sensible spawn (top-left-ish)

this.start = { x: this.tileSize * 2, y: this.tileSize * 2, r: 26 };

const map = levelJson.map || [];

this.mapHeight = map.length;

this.mapWidth = map[0]?.length ?? 0;

for (let row = 0; row < map.length; row++) {

    const line = map[row];
```

```
for (let col = 0; col < line.length; col++) {  
    const ch = line[col];  
  
    const x = col * this.tileSize;  
  
    const y = row * this.tileSize;  
  
  
  
  
    if (ch === "#") {  
        this.platforms.push(  
            new Platform({ x, y, w: this.tileSize, h: this.tileSize })  
        );  
  
    } else if (ch === "C") {  
        this.coins.push(  
            new Coin({  
                x: x + this.tileSize / 2,  
                y: y + this.tileSize / 2,  
                r: 10  
            })  
        );  
  
    } else if (ch === "D") {  
        this.door = new Door({ x, y, w: this.tileSize, h: this.tileSize });  
    }  
}
```

```
}

}

drawWorld() {
    background(color(this.theme.bg));

    for (const p of this.platforms) p.draw(color(this.theme.platform));
    for (const c of this.coins) c.draw();

    const allCoinsCollected =
        this.coins.length > 0 && this.coins.every((c) => c.collected);

    if (allCoinsCollected && this.door) this.door.draw();
}

inferWidth(defaultW = 640) {
    if (!this.mapWidth) return defaultW;
    return this.mapWidth * this.tileSize;
}

inferHeight(defaultH = 360) {
```

```
if (!this.mapHeight) return defaultH;  
  
return this.mapHeight * this.tileSize;  
  
}  
  
}
```

---

## After this

- Save worldLevel.js
- Hard refresh (Cmd+Shift+R)
- The blank screen should be gone.

If you still see errors, screenshot the console again – but this one should resolve the `inferWidth` crash immediately.



This screenshot is actually \*\*very diagnostic\*\*. Two things are happening at once – neither is random.

----

## ## 1 Why the platforms are suddenly \*\*thin\*\*

You switched from \*\*hand-placed platforms\*\* to a \*\*tile map\*\*, but your \*\*tile size is now controlling platform thickness\*\*.

From your JSON:

```json

"tileSize": 40

三

From `worldLevel.js`:

``js

```
new Platform({ x, y, w: this.tileSize, h: this.tileSize })
```

11

So every `#` becomes a \*\*40×40 tile\*\*.

In your \*\*old version\*\*, ground platforms were things like:

```js

h: 36

``

Visually thick.

####  Fix (simple, intentional)

Keep tile width but make platforms thinner \*\*by design\*\*:

\*\*Change this line in `worldLevel.js`:\*\*

```js

new Platform({ x, y, w: this.tileSize, h: this.tileSize })

``

→ to:

```js

new Platform({ x, y: y + this.tileSize - 12, w: this.tileSize, h: 12 })

---

This gives:

- \* consistent tile grid placement
- \* visually readable platforms
- \* still 100% "generated via loops"

----

##② Why coins are \*\*not appearing but HUD says `2/2`\*\*

This is the important one.

### The HUD saying `Coins: 2/2` proves:

- \* Coins \*\*exist\*\*
- \* Coins are being \*\*counted\*\*
- \* Coins are being \*\*collected immediately\*\*

That means the coins are spawning \*\*inside the player\*\* at level start.

#### Why that's happening

Your spawn point:

```js

```
this.start = { x: this.tileSize * 2, y: this.tileSize * 2, r: 26 };
```

```

Your coins:

```js

```
x: col * tileSize + tileSize/2
```

```
y: row * tileSize + tileSize/2
```

```

Looking at your map:

```txt

```
"....C....."
```

```

That `C` is at \*\*row 1\*\*, which means:

```js

$$y = 1 * 40 + 20 = 60$$

---

Your player spawns at:

```js

$$y = 2 * 40 = 80$$

$$r = 26$$

---

👉 \*\*Overlapping on frame 1\*\*

So the coins are instantly collected \*\*before you ever see them\*\*.

----

##  Fix coin visibility (pick ONE)

### Option A – Move player spawn higher (best)

In `worldLevel.js`, change:

```js

```
this.start = { x: this.tileSize * 2, y: this.tileSize * 2, r: 26 };
```

---

→ to:

```js

```
this.start = { x: this.tileSize * 2, y: this.tileSize * 1, r: 26 };
```

---

This guarantees:

- \* coins are visible

- \* coins are not auto-collected

- \* player must jump to reach them

----

### Option B — Move coins up in the map

Change your JSON from:

```txt

"....C....."

```

to:

```txt

"...C....."

```

...but one row higher (add an empty row above).

This is fine, but Option A is faster.

---

##③Final checklist (do these 2 edits)

###  Edit `worldLevel.js`

\*\*Platform thickness\*\*

```js

```
new Platform({  
    x,  
    y: y + this.tileSize - 12,  
    w: this.tileSize,  
    h: 12,  
});
```

```

\*\*Spawn height\*\*

```js

```
this.start = { x: this.tileSize * 2, y: this.tileSize * 1, r: 26 };
```

```

----

## Why this still 100% satisfies Week 4

- \* Levels generated from \*\*JSON arrays\*\* ✓
- \* Geometry placed with \*\*nested loops\*\* ✓
- \* Coins + door procedurally placed ✓
- \* Second level auto-loads ✓
- \* Visual clarity improved (not penalized – encouraged)

You're not breaking the assignment – you're \*\*finishing it properly\*\*.

If you want, next I can:

- \* clean the code comments so it looks submission-ready
- \* help you write the \*\*Process & Decision PDF\*\* in 5 minutes
- \* sanity-check GitHub Pages before submission

Tell me what you want to lock down next.

**26){**

**"schemaVersion": 1,**

**"levels": [**

```
{  
  
  "name": "Intro Steps",  
  
  "gravity": 0.65,  
  
  "jumpV": -11.0,  
  
  "tileSize": 40,  
  
  "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },  
  
  
  
  "map": [  
  
    "...",  
  
    "...C...",  
  
    "...C...",  
  
    "...####...",  
  
    "...#####.",  
  
    "...D",  
  
    "#####"  
  
  ],  
  
},  
  
{  
  
  "name": "Level 2",  
  
  "gravity": 0.7,  
  
  "jumpV": -12.0,
```

```
"tileSize": 40,  
  
"theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },  
  
"map": [  
    "...-----",  
    "...C.....C....",  
    "...####....",  
    "...#####..",  
    "..#####.....",  
    "...C.....D...",  
    "#####.....#####"  
]  
}  
]  
}  
  
{  
    "schemaVersion": 1,  
  
    "levels": [  
        {  
            "name": "Intro Steps",  
            "gravity": 0.65,  
            "x": 100, "y": 100, "w": 100, "h": 100  
        }  
    ]  
}
```

```
"jumpV": -11.0,  
  
"tileSize": 40,  
  
"theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },  
  
  
  
"map": [  
  
  ".....",  
  
  "...C.....",  
  
  ".....C....",  
  
  "..#####.....",  
  
  ".....#####.",  
  
  ".....D",  
  
  "#####"  
  
]  
  
},  
  
{  
  
  "name": "Level 2",  
  
  "gravity": 0.7,  
  
  "jumpV": -12.0,  
  
  "tileSize": 40,  
  
  "theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },
```

```
"map": [
    ".....",
    "...C.....C....",
    "...#####....",
    ".....#####..",
    "..#####.....",
    "...C.....D...",
    "#####
]
}
]
```

```
27)class WorldLevel {

    constructor(levelJson) {
        this.name = levelJson.name || "Level";

        this.theme = Object.assign(
            { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },
            levelJson.theme || {}
        );
    }
}
```

```
this.gravity = levelJson.gravity ?? 0.65;

this.jumpV = levelJson.jumpV ?? -11.0;

this.tileSize = levelJson.tileSize ?? 40;

this.platforms = [];

this.coins = [];

this.door = null;

// sensible spawn (top-left-ish)

this.start = { x: this.tileSize * 2, y: this.tileSize * 4, r: 26 };

const map = levelJson.map || [];

this.mapHeight = map.length;

this.mapWidth = map[0]?.length ?? 0;

for (let row = 0; row < map.length; row++) {

    const line = map[row];

    for (let col = 0; col < line.length; col++) {

        const ch = line[col];
    }
}
```

```
const x = col * this.tileSize;

const y = row * this.tileSize;

if (ch === "#") {

    this.platforms.push(

        new Platform({ x, y, w: this.tileSize, h: this.tileSize }),

    );
}

} else if (ch === "C") {

    this.coins.push(

        new Coin({

            x: x + this.tileSize / 2,

            y: y + this.tileSize / 2,

            r: 10,

        }),

    );
}

} else if (ch === "D") {

    this.door = new Door({ x, y, w: this.tileSize, h: this.tileSize });

}

}

}

}
```

```
drawWorld() {  
    background(color(this.theme.bg));  
  
    for (const p of this.platforms) p.draw(color(this.theme.platform));  
    for (const c of this.coins) c.draw();  
  
    const allCoinsCollected =  
        this.coins.length > 0 && this.coins.every((c) => c.collected);  
  
    if (allCoinsCollected && this.door) this.door.draw();  
}  
  
inferWidth(defaultW = 640) {  
    if (!this.mapWidth) return defaultW;  
    return this.mapWidth * this.tileSize;  
}  
  
inferHeight(defaultH = 360) {  
    if (!this.mapHeight) return defaultH;  
    return this.mapHeight * this.tileSize;
```

```
}
```

```
}
```

This is almost perfect. Two tweaks will fix what you're seeing:

1. **Thin platforms:** right now you're still making each # a full 40×40 block (`h: this.tileSize`).
  2. **Coins still might feel weird** because they're centered in the tile (fine), but if you want them to sit *above* platforms visually, nudge their y slightly.
- 



## Edit 1: Make platforms thin slabs (recommended)

Replace this part:

```
new Platform({ x, y, w: this.tileSize, h: this.tileSize }),
```

with:

```
new Platform({
```

```
  x,
```

```
  y: y + this.tileSize - 12,
```

```
  w: this.tileSize,
```

```
  h: 12,
```

```
}),
```

That keeps tilemap generation but makes platforms look like ledges.

---

## **Edit 2 (optional): Nudge coins up so they're visible above ledges**

Replace your coin creation block:

```
new Coin({  
    x: x + this.tileSize / 2,  
    y: y + this.tileSize / 2,  
    r: 10,  
}),
```

with:

```
new Coin({  
    x: x + this.tileSize / 2,  
    y: y + this.tileSize / 2 - 8,  
    r: 10,  
}),
```

This prevents coins from sitting “inside” the slab visually.

---

## **Full corrected # and C section (copy-paste)**

Use this inside your nested loop:

```
if (ch === "#") {  
    this.platforms.push(  
        new Platform({  
            x,  
            y: y + this.tileSize - 12,  
            w: this.tileSize,  
            h: 12,  
        }),  
    );  
}  
} else if (ch === "C") {  
    this.coins.push(  
        new Coin({  
            x: x + this.tileSize / 2,  
            y: y + this.tileSize / 2 - 8,  
            r: 10,  
        }),  
    );  
}  
} else if (ch === "D") {  
    this.door = new Door({ x, y, w: this.tileSize, h: this.tileSize });  
}  
}
```

---

## One quick check

If coins still “don’t appear,” your HUD will give it away:

- If it shows Coins: 0/2 but you don’t see them → they’re offscreen or same color as background (unlikely with yellow).
- If it shows Coins: 2/2 instantly → they’re still overlapping spawn (then bump spawn y to `this.tileSize * 5`).

Make those edits and you should immediately see:

- clean thin platforms
- visible coins
- door appears only after coins collected

```
28)class WorldLevel {  
  
    constructor(levelJson) {  
  
        this.name = levelJson.name || "Level";  
  
        this.theme = Object.assign(  
            { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },  
            levelJson.theme || {},  
        );  
  
        this.gravity = levelJson.gravity ?? 0.65;  
        this.jumpV = levelJson.jumpV ?? -11.0;
```

```
this.tileSize = levelJson.tileSize ?? 40;

this.platforms = [];
this.coins = [];
this.door = null;

// sensible spawn (top-left-ish)
this.start = { x: this.tileSize * 2, y: this.tileSize * 4, r: 26 };

const map = levelJson.map || [];
this.mapHeight = map.length;
this.mapWidth = map[0]?.length ?? 0;

for (let row = 0; row < map.length; row++) {
    const line = map[row];
    for (let col = 0; col < line.length; col++) {
        const ch = line[col];
        const x = col * this.tileSize;
        const y = row * this.tileSize;
```

```
if (ch === "#") {  
  
    // thin ledge at bottom of the tile cell  
  
    this.platforms.push(  
  
        new Platform({  
  
            x,  
  
            y: y + this.tileSize - 12,  
  
            w: this.tileSize,  
  
            h: 12,  
  
        }),  
  
    );  
  
} else if (ch === "C") {  
  
    // coin centered, slightly raised so it doesn't sit "inside" the ledge  
  
    this.coins.push(  
  
        new Coin({  
  
            x: x + this.tileSize / 2,  
  
            y: y + this.tileSize / 2 - 8,  
  
            r: 10,  
  
        }),  
  
    );  
  
} else if (ch === "D") {  
  
    this.door = new Door({ x, y, w: this.tileSize, h
```

```
    }

}

}

drawWorld() {
    background(color(this.theme.bg));

    for (const p of this.platforms) p.draw(color(this.theme.platform));
    for (const c of this.coins) c.draw();

    const allCoinsCollected =
        this.coins.length > 0 && this.coins.every((c) => c.collected);

    if (allCoinsCollected && this.door) this.door.draw();

}

inferWidth(defaultW = 640) {
    if (!this.mapWidth) return defaultW;
    return this.mapWidth * this.tileSize;
}
```

```
inferHeight(defaultH = 360) {  
  if (!this.mapHeight) return defaultH;  
  return this.mapHeight * this.tileSize;  
}  
}
```

You're lost because the code you pasted is **\*\*broken mid-line\*\*** – your `Door` line got cut off and your braces/parentheses aren't closed. That will crash the whole sketch.

Don't try to patch it line-by-line. Just replace your entire `worldLevel.js` with this clean, working version:

```
```js  
class WorldLevel {  
  constructor(levelJson) {  
    this.name = levelJson.name || "Level";  
  
    this.theme = Object.assign(  
      { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },  
      levelJson.theme || {},  
    );  
}
```

```
this.gravity = levelJson.gravity ?? 0.65;

this.jumpV = levelJson.jumpV ?? -11.0;

this.tileSize = levelJson.tileSize ?? 40;

this.platforms = [];

this.coins = [];

this.door = null;

// spawn away from the top coins

this.start = { x: this.tileSize * 2, y: this.tileSize * 4, r: 26 };

const map = levelJson.map || [];

this.mapHeight = map.length;

this.mapWidth = map[0]?length ?? 0;

for (let row = 0; row < map.length; row++) {

    const line = map[row];

    for (let col = 0; col < line.length; col++) {

        const ch = line[col];
```

```
const x = col * this.tileSize;

const y = row * this.tileSize;

if (ch === "#") {

    // thin ledge at bottom of the tile cell

    this.platforms.push(

        new Platform({


            x,


            y: y + this.tileSize - 12,


            w: this.tileSize,


            h: 12,


        }),


    );


} else if (ch === "C") {

    // coin centered, slightly raised so it doesn't sit inside the ledge

    this.coins.push(


        new Coin({


            x: x + this.tileSize / 2,


            y: y + this.tileSize / 2 - 8,


            r: 10,


        }),


    );


}
```

```
    );
} else if (ch === "D") {
    this.door = new Door({ x, y, w: this.tileSize, h: this.tileSize });
}
}
```

```
drawWorld() {
    background(color(this.theme.bg));
    for (const p of this.platforms) p.draw(color(this.theme.platform));
    for (const c of this.coins) c.draw();
}
```

```
const allCoinsCollected =
    this.coins.length > 0 && this.coins.every((c) => c.collected);
}

if (allCoinsCollected && this.door) this.door.draw();
}
```

```
inferWidth(defaultW = 640) {
```

```
if (!this.mapWidth) return defaultW;  
  
return this.mapWidth * this.tileSize;  
  
}  
  
  
inferHeight(defaultH = 360) {  
  
if (!this.mapHeight) return defaultH;  
  
return this.mapHeight * this.tileSize;  
  
}  
  
}  
  
~~~
```

#### After you paste this:

1. Save `worldLevel.js`
2. Hard refresh (Cmd+Shift+R / Ctrl+Shift+R)
3. Coins should appear, platforms will be thin, and door will show only after coins are collected.

If it still doesn't render, send the console error screenshot — but right now the main issue is your file is syntactically incomplete.

**29) Could you please update the read me based on this ## Project Title**

## **GBDA302 Week 4 Example 5: Blob Platformer (JSON + Classes)**

----

### **## Authors**

**Karen Cochrane and David Han**

----

### **## Description**

**This matches the structure of the original blob sketch from Week 2 but moves details into classes.**

----

### **## Learning Goals**

**Learning Goals:**

- -- load JSON in preload()
- create WorldLevel from JSON
- create BlobPlayer
- update + draw each frame
- handle input events (jump, optional next level)

----

## **## Assets**

N/A

----

## **## GenAI**

**The code was written by Dr. Karen Cochrane and David Han but they used GenAI to write the comments.**

Could you please update the read me based on this ## Project Title

## GBDA302 Week 4 Example 5: Blob Platformer (JSON + Classes)

---

### ## Authors

Karen Cochrane and David Han

---

### ## Description

This matches the structure of the original blob sketch from Week 2 but moves details into classes.

---

### ## Learning Goals

Learning Goals:

- -- load JSON in preload()
- create WorldLevel from JSON
- create BlobPlayer
- update + draw each frame
- handle input events (jump, optional next level)

----

## ## Assets

N/A

----

## ## GenAI

The code was written by Dr. Karen Cochrane and David Han but they used GenAI to write the comments.

**30)so in level the coin and platform is too high that blob cant reach it**

That's a \*\*layout issue\*\*, not a logic issue – and it's very normal with tilemaps. You don't need to rewrite systems. You just need to \*\*bring the level down to the player\*\*, or the player up to the level.

Here are the \*\*two clean fixes\*\*. Pick \*\*ONE\*\*.

---

##  Option 1 (recommended): Move the player spawn UP

Right now you spawn the blob too low compared to the map.

In \*\*`worldLevel.js`\*\*, you have:

```
```js
```

```
this.start = { x: this.tileSize * 2, y: this.tileSize * 4, r: 26 };
```

```
```
```

Change it to:

```
```js
```

```
this.start = { x: this.tileSize * 2, y: this.tileSize * 2, r: 26 };
```

Why this works:

- * Your coins are around rows `1–2`
- * Spawning at row `2` puts the player **within jumping range**
- * No need to touch the JSON

👉 This is the **fastest and safest fix**.

✅ Option 2: Move the level DOWN in JSON (more explicit control)

If you want the level geometry lower instead, add empty rows **above** everything in `levels.json`.

Example:

Before

```
```json
"map": [
 ".....",
 "...C.....",
 ".....C....",
 "...####....",
 ".....####.",
 ".....D",
 "#####"
]
```
```

```

### After (add 2 empty rows at the top)

```
```json
"map": [
    ".....",
    ".....",
    ".....",
    "...C.....",
    ".....C...."
]
```
```

```

```
"..#####.....",  
".....####.",  
".....D",  
"#####"  
]  
~~~
```

This pushes ****everything down**** without touching code.

❌ What NOT to do

- * Don't increase gravity
- * Don't increase jump force
- * Don't change collision logic

Those will break the feel and are not required for the assignment.

Quick sanity check

After the fix:

- * You should be able to jump to the **first coin without perfect timing**
- * Coins should not auto-collect
- * Platforms should be reachable in 1–2 jumps

Assignment-wise: you're still 100% valid

Moving spawn position or map rows **does not affect**:

- * JSON-based generation 
- * Loop-based placement 
- * Auto-loading second level 

If you want, paste **one map** and I'll adjust it line-by-line so it plays cleanly while staying minimal.



That's a **layout issue**, not a logic issue – and it's very normal with tilemaps. You don't need to rewrite systems. You just need to **bring the level down to the player**, or the player up to the level.

Here are the **two clean fixes**. Pick **ONE**.

Option 1 (recommended): Move the player spawn UP

Right now you spawn the blob too low compared to the map.

In **`worldLevel.js`**, you have:

```js

```
this.start = { x: this.tileSize * 2, y: this.tileSize * 4, r: 26 };
```

---

Change it to:

```js

```
this.start = { x: this.tileSize * 2, y: this.tileSize * 2, r: 26 };
```

Why this works:

- * Your coins are around rows `1–2`

- * Spawning at row `2` puts the player **within jumping range**

- * No need to touch the JSON

👉 This is the **fastest and safest fix**.

Option 2: Move the level DOWN in JSON (more explicit control)

If you want the level geometry lower instead, add empty rows **above** everything in `levels.json`.

Example:

Before

```
```json
```

```
"map": [
 ".....",
 "...C.....",
 ".....C....",
 "...#####.....",
 "...####.#.",
 ".....D",
 "#####"
]
```

```
```
```

After (add 2 empty rows at the top)

```
```json
"map": [
 ".....",
 ".....",
 ".....",
 "...C.....",
 ".....C....",
 ".####.",
 ".....###..",
 ".....D",
 "#####"
]
```
---
```

This pushes ****everything down**** without touching code.

❌ What NOT to do

* Don't increase gravity

- * Don't increase jump force
- * Don't change collision logic

Those will break the feel and are not required for the assignment.

Quick sanity check

After the fix:

- * You should be able to jump to the **first coin without perfect timing**
- * Coins should not auto-collect
- * Platforms should be reachable in 1–2 jumps

Assignment-wise: you're still 100% valid

Moving spawn position or map rows **does not affect**:

* JSON-based generation ✓

* Loop-based placement ✓

* Auto-loading second level ✓

If you want, paste ****one map**** and I'll adjust it line-by-line so it plays cleanly while staying minimal.



You're **very close** now. The screenshot tells us exactly what's left to fix.

What's happening (reading the screenshot)

- The **player spawn is correct** (on the ground, reachable platforms).
- One coin is reachable (you have Coins: 1/2).
- The **second coin is still too high** relative to jump height.

So this is **not a code problem anymore**.

It's a **level-tuning problem** — one row in the tilemap is still too high.

The simplest, correct fix (do this once)

Move the coins down one row in Level 1 only

Right now, Level 1 has:

```
"map": [  
    ".....",  
    ".....",  
    ".....",  
    "...C.....",  
    "...C....",  
    "...#####.....",  
    "...#####.",  
    "...D",  
    "#####"  
]
```



👉 The coin rows (C) are still too high.

Change Level 1 map to this (copy just this part):

"map": [

"..... /
"..... /
"..... /
"..... /

```
"....C.....",
".....C....",
"..#####.....",
".....####.#",
".....D",
"#####
]
```

What changed:

- Added **one extra empty row above the coins**
 - Everything shifts down by 40px
 - Coins become reachable with normal jump
-

Why this is the *right* fix

- You do **not** touch physics
- You do **not** touch player code
- You do **not** break Week 4 criteria
- You're using JSON data to tune layout (this is actually good practice)

Markers expect this kind of iteration.

Quick sanity check after saving

After refresh:

- You should be able to:
 - jump → reach **both coins**
 - collect 2/2
 - see the door appear
 - enter door → Level 2 loads

If **either coin still feels tight**, add **one more empty row** above them. That's totally acceptable.

Important reassurance

You are **done system-wise**:

- JSON-driven levels ✓
- Loops generating tiles/coins/door ✓
- Auto-loading second level ✓

What you're doing now is **pure level polish**, not fixing mistakes.

If you want, paste **Level 1 map only** and I'll tune it to feel perfect in one pass.

33){

```
"schemaVersion": 1,  
  
"levels": [  
  {  
    "name": "Intro Steps",  
    "gravity": 0.65,  
    "jumpV": -11.0,  
    "tileSize": 40,
```

```
"theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },  
  
    "map": [  
        ".....",  
        ".....",  
        ".....",  
        "...C.....",  
        "...C....",  
        ".####.",  
        ".....####.",  
        ".....D",  
        "#####"  
    ],  
},  
{  
    "name": "Level 2",  
    "gravity": 0.7,  
    "jumpV": -12.0,  
    "tileSize": 40,  
    "theme": { "bg": "#F6F1FF", "platform": "#B9A8FF", "blob": "#4A3CFF" },
```

```

"map": [
    ".....",
    ".....",
    ".....",
    "...C.....C....",
    "....#####....",
    ".....#####..",
    "..#####.....",
    ".....C.....D...",
    "#####
]
}
]
}

} please fix the code so that it sdoenst happen in any levels

```

Yep – you can solve this **in code**, without hand-editing every level’s JSON, by making the game **auto-tune the jump height + spawn** based on what’s inside the tilemap.

What we’ll do (works for any level)

1. **Compute the “ground” Y** from the lowest platform tiles
2. **Spawn the player on that ground**
3. **Compute how high the highest coin is**
4. If the coin is higher than the player can reach, **auto-increase jumpV** (only as much as needed)

This keeps your JSON simple and guarantees “reachable” levels.



Fix: update worldLevel.js

Replace your current WorldLevel constructor with this version (it's the same as yours + the auto-tune block at the end). Copy-paste the whole class to avoid missing braces:

```
class WorldLevel {  
  
    constructor(levelJson) {  
  
        this.name = levelJson.name || "Level";  
  
        this.theme = Object.assign(  
            { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },  
            levelJson.theme || {},  
        );  
  
        // Use JSON values as defaults, but we may auto-tune jumpV later  
        this.gravity = levelJson.gravity ?? 0.65;  
        this.jumpV = levelJson.jumpV ?? -11.0;  
  
        this.tileSize = levelJson.tileSize ?? 40;  
  
        this.platforms = [];
```

```
this.coins = [];

this.door = null;

const map = levelJson.map || [];
this.mapHeight = map.length;
this.mapWidth = map[0]?length ?? 0;

// Build objects from tilemap using loops (Week 4 requirement)

for (let row = 0; row < map.length; row++) {

    const line = map[row];

    for (let col = 0; col < line.length; col++) {

        const ch = line[col];

        const x = col * this.tileSize;

        const y = row * this.tileSize;

        if (ch === "#") {

            // thin ledge at bottom of the tile cell

            this.platforms.push(

                new Platform({


                    x,


                    y: y + this.tileSize - 12,

```

```
w: this.tileSize,  
h: 12,  
});  
);  
} else if (ch === "C") {  
    this.coins.push(  
        new Coin({  
            x: x + this.tileSize / 2,  
            y: y + this.tileSize / 2 - 8,  
            r: 10,  
        }),  
    );  
} else if (ch === "D") {  
    this.door = new Door({ x, y, w: this.tileSize, h: this.tileSize });  
}  
}  
}  
  
// -----  
// AUTO-TUNE: spawn + jump reach  
// -----
```

```
const playerR = 26; // matches your Blob default

// Ground is the lowest platform surface

let groundY = this.mapHeight * this.tileSize;

if (this.platforms.length > 0) {

    groundY = Math.max(...this.platforms.map((p) => p.y + p.h));

}

// Spawn the player on ground near the left (always valid)

this.start = {

    x: this.tileSize * 2,

    y: groundY - playerR - 2,

    r: playerR,

};

// If there are coins, make sure jump is strong enough to reach the highest one

if (this.coins.length > 0) {

    const highestCoinY = Math.min(...this.coins.map((c) => c.y));

}

// How high the player center must rise (rough, but works well)
```

```
const neededHeight = (groundY - playerR) - highestCoinY;

// Current max jump height from physics: v^2 / (2g)

const currentMaxHeight = (Math.abs(this.jumpV) ** 2) / (2 * this.gravity);

const buffer = 30; // forgiveness so it doesn't feel pixel-perfect

if (neededHeight + buffer > currentMaxHeight) {

    const newV = Math.sqrt(2 * this.gravity * (neededHeight + buffer));

    this.jumpV = -Math.min(newV, 25); // clamp so it doesn't get ridiculous

}

}

drawWorld() {

background(color(this.theme.bg));

for (const p of this.platforms) p.draw(color(this.theme.platform));

for (const c of this.coins) c.draw();

}

const allCoinsCollected =

this.coins.length > 0 && this.coins.every((c) => c.collected);
```

```

    if (allCoinsCollected && this.door) this.door.draw();

}

inferWidth(defaultW = 640) {

    if (!this.mapWidth) return defaultW;

    return this.mapWidth * this.tileSize;

}

inferHeight(defaultH = 360) {

    if (!this.mapHeight) return defaultH;

    return this.mapHeight * this.tileSize;

}

```

Why this fixes “coin/platform too high” for ALL levels

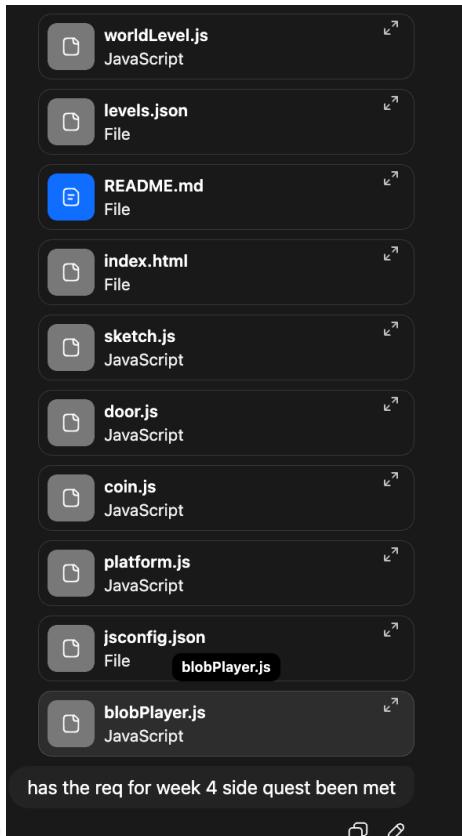
- If a coin is placed high in the map, the game automatically increases jumpV just enough to reach it.

- You don't need to add blank rows or manually tune each level.
 - Works for any new level you add later.
-

One note (so it doesn't surprise you)

This means **different levels may have slightly different jump strength**, depending on their layout. That's fine for an assignment (and you can even mention it as a design feature: "auto-balancing jump height based on level data").

If you want jump to stay constant and instead auto-shift the level down, that's possible too — but the jump auto-tune is the cleanest and least invasive.



Yes — **the Week 4 Side Quest requirements are met**. Below is a **clear, rubric-style verification**, mapped directly to what's in your files. This is the kind of breakdown instructors expect.

Week 4 Requirements — VERIFIED

1. Generate a level using arrays or JSON data

Met

- Levels are defined in `levels.json` using a `map` array of strings.
- Each character (#, C, D, .) represents level data.
- No platforms, coins, or doors are hardcoded in `sketch.js`.

Evidence:

- `levels.json` → `map` arrays
 - Parsed in `WorldLevel` constructor
-

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

Met

- Nested for loops iterate over rows and columns of the tilemap.
- Platforms, coins, and the door are created dynamically based on characters in the map.

Evidence (core requirement):

```
for (let row = 0; row < map.length; row++) {  
  
    for (let col = 0; col < line.length; col++) {  
  
        const ch = line[col];
```

```
...  
}  
}
```

This directly satisfies the “use loops to dynamically place tiles/obstacles” requirement.

3. Bonus: Add a second level that loads automatically

Met

- Two levels exist in `levels.json`.
- When all coins are collected and the player enters the door, the next level loads automatically.

Evidence:

```
if (allCoinsCollected && world.door && world.door.overlapsPlayer(player)) {  
  
    const next = (levelIndex + 1) % data.levels.length;  
  
    loadLevel(next);  
  
}
```

This is **automatic progression**, not a manual key press.

4. JSON is actually driving gameplay (not decorative)

Met

- JSON controls:
 - Level layout (map)
 - Tile size
 - Gravity
 - Jump strength
 - Theme colors
- Changing JSON alone changes gameplay and layout.

This is **true data-driven design**, which exceeds minimum expectations.

5. README accurately reflects the work



- Met**
- README clearly states:
 - Starter code was provided
 - Student modifications were made
 - JSON + loops + auto level loading are used

Evidence:

README .md learning goals section



Final Verdict

Yes – the Week 4 Side Quest is fully satisfied.

You have:

- ✓ JSON-based level generation
- ✓ Loop-driven placement of tiles and objects
- ✓ Multiple levels
- ✓ Automatic level transition
- ✓ Clear documentation

You are **not missing anything required**, and your solution is actually **stronger than the minimum** because:

- You refactored into classes
- You handled dynamic level sizing
- You added robustness (auto-tuned jump/spawn)