

Process & Decision Documentation

Project/Assignment Decisions

I decided to add a ball as a goal for each level using W4 example 5 base, even though the main sidequest task was just to make a level with arrays and JSON. I wanted to make the blob platformer more game-like, giving each level a clear objective. During implementation, I rejected AI's earlier restart ideas and limited "retry" behavior so it only mattered on Level 3, which matched the scope and difficulty I was going for.

Role-Based Process Evidence

Name: Queene Calading

Role(s): Designer

Primary responsibility for this work: Implementing level progression, win/lose conditions, and end screens

Goal of Work Session:

- Add a circle ball as the goal to each JSON level and detect collisions to progress
- Create Level 3 as a "final challenge" with a win/lose condition
- Add a win/lose end screen and controlled restart behavior

Tools, Resources, or Inputs Used:

ChatGPT 5.2 (Code help, debugging)

VSCode

Github

Prior starter code given

GenAI Documentation

Date Used: Feb 9, 2026

Tool Disclosure: ChatGPT 5.2

Purpose of Use: Coding guidance for implementing circle collision, adding a loose condition, and debugging syntax/state issues

Summary of Interaction: The AI helped by providing step-by-step instructions across levels.json, worldLevel.js, and sketch.js, plus debugging help for issues like missing braces, undefined variables (isDeath), and render-state carryover (textAlign, textSize)

Human Decision Point(s):

- I accepted the AI-provided code as a starting point as I was unsure of how to make the code from scratch for the idea that I had, but modified it to my liking
- I rejected the idea of being able to restart across all levels and adjusted restarts so Level 3 death mattered most, with a clearer "final level" structure (if you die, you have to restart 3 until you can win)
- Playtested and edited parameters of where goalBall would be as AI just placed it in the same place for next level, when I want to have it on a platform, so I got to choose and modify which platform / where I were to place it

Integrity & Verification Note: I verified that the given ideas were compatible with p5.js physics and drawing by checking the online p5.js library of what each parameter means in a line of code, and ensured no AI-generated code was copied without understanding (I asked questions and made it so that I was using AI to show me how / what I could do, rather than telling it what needs to be done and copy paste).

Scope of GenAI Use: GenAI contributed to the coding suggestions as well as debugging; all code implementation to the actual sketch.js page, testing, and adjustments to parameters were done by me. It did not design the platform placements, or the overall gameplay intention. I chose the goal positions, and final behavior rules.

Limitations or Misfires:

- Does not account for my skill level in actually coding, so it just initially gave me example codes to implement and modify and I had to ask what its purpose is for
- Early output caused issues when loadLevel() was called inside drawEndScreen() (auto-reloaded screen rather than going back to level 1 when clicking restart). I corrected this by moving level loads to keyPressed() and leaving the end screen as rendering-only

Appendix

Transcript

Prompt:

BlobPlayer.js (Example 5)

BlobPlayer owns all "dynamic" player state:

- position (x,y), radius (r)
 - velocity (vx,vy)
- movement tuning (accel, friction, max run)
 - jump state (onGround)
- blob rendering animation parameters (noise wobble)

It also implements:

- update() for physics + collision against platforms
 - jump() for input
- draw() for the "breathing blob" look

The algorithm is the same as the original blob world example from Week 2:

- Apply input acceleration
 - Apply friction
 - Apply gravity
- Compute an AABB (box) around the blob
 - Move box in X and resolve collisions
 - Move box in Y and resolve collisions
- Write back box center to blob position

*/

```
class BlobPlayer {  
    constructor() {  
        // ----- Transform -----  
        this.x = 0;  
        this.y = 0;  
        this.r = 26;  
  
        // ----- Velocity -----  
        this.vx = 0;  
        this.vy = 0;  
  
        // ----- Movement tuning (matches your original values) -----  
        this.accel = 0.55;  
        this.maxRun = 4.0;  
  
        // Physics values that are typically overridden per level.  
        this.gravity = 0.65;  
        this.jumpV = -11.0;  
  
        // State used by jumping + friction choice.  
        this.onGround = false;  
  
        // Friction:  
        // - in air: almost no friction (keeps momentum)  
        // - on ground: more friction (stops more quickly)  
        this.frictionAir = 0.995;  
        this.frictionGround = 0.88;  
  
        // ----- Blob rendering / animation -----  
        this.t = 0;  
        this.tSpeed = 0.01;  
        this.wobble = 7;
```

```

        this.points = 48;
        this.wobbleFreq = 0.9;
    }

    /*
     * Apply level settings + spawn the player.
     * We reset velocities so each level starts consistently.
     */
    spawnFromLevel(level) {
        this.gravity = level.gravity;
        this.jumpV = level.jumpV;

        this.x = level.start.x;
        this.y = level.start.y;
        this.r = level.start.r;

        this.vx = 0;
        this.vy = 0;
        this.onGround = false;
    }

    /*
     * Update movement + resolve collisions against all platforms.
     */

    Input is polled with keyIsDown to get smooth movement (held keys).
    This keeps the behavior aligned with your original blob example.
    */
    update(platforms) {
        // 1) Horizontal input (A/D or arrows)
        let move = 0;
        if (keyIsDown(65) || keyIsDown(LEFT_ARROW)) move -= 1;
        if (keyIsDown(68) || keyIsDown(RIGHT_ARROW)) move += 1;

        // 2) Apply horizontal acceleration based on input
        this.vx += this.accel * move;

        // 3) Apply friction (ground vs air)
        this.vx *= this.onGround ? this.frictionGround : this.frictionAir;

        // 4) Clamp max run speed
        this.vx = constrain(this.vx, -this.maxRun, this.maxRun);

        // 5) Apply gravity every frame
        this.vy += this.gravity;

        // 6) Build an AABB around the blob (center/radius -> box)
        let box = {
            x: this.x - this.r,
            y: this.y - this.r,
            w: this.r * 2,
            h: this.r * 2,
        };

        // 7) Move in X and resolve collisions
        box.x += this.vx;

        for (const s of platforms) {
            if (overlapAABB(box, s)) {
                // If moving right, snap to the left side of the platform.
                if (this.vx > 0) box.x = s.x - box.w;
                // If moving left, snap to the right side of the platform.
            }
        }
    }
}

```

```

        else if (this.vx < 0) box.x = s.x + s.w;

        // Cancel horizontal velocity after collision.
        this.vx = 0;
    }

}

// 8) Move in Y and resolve collisions
box.y += this.vy;

// Reset and recompute onGround each frame during Y resolution.
this.onGround = false;

for (const s of platforms) {
    if (overlapAABB(box, s)) {
        if (this.vy > 0) {
            // Falling: snap to platform top
            box.y = s.y - box.h;
            this.vy = 0;
            this.onGround = true;
        } else if (this.vy < 0) {
            // Rising: snap to platform bottom (head bump)
            box.y = s.y + s.h;
            this.vy = 0;
        }
    }
}

// 9) Write back blob center from box position
this.x = box.x + box.w / 2;
this.y = box.y + box.h / 2;

// 10) Optional: keep player within canvas horizontally.
this.x = constrain(this.x, this.r, width - this.r);

// 11) Advance blob animation time
this.t += this.tSpeed;
}

//Jump: only possible when on ground.
jump() {
    if (!this.onGround) return;
    this.vy = this.jumpV;
    this.onGround = false;
}

/*
Draw the blob with a wobbly outline:
- we sample a noise value around the circle
  - perturb the radius slightly per vertex
- this creates an organic "breathing"" look

This is the same technique as the original drawBlob() function.
*/
draw(colourHex) {
    fill(color(colourHex));
    beginShape();

    for (let i = 0; i < this.points; i++) {
        const a = (i / this.points) * TAU;

```

```

        // Noise input: circle coordinates + time.
        const n = noise(
            cos(a) * this.wobbleFreq + 100,
            sin(a) * this.wobbleFreq + 100,
            this.t,
        );

        // Map noise to a small radius offset.
        const rr = this.r + map(n, 0, 1, -this.wobble, this.wobble);

        // Place the vertex around the center.
        vertex(this.x + cos(a) * rr, this.y + sin(a) * rr);
    }

    endShape(CLOSE);
}

}

/*
Collision function: AABB overlap test.
- a is the moving player "box"
- b is a platform rectangle

We accept b as either:
- a Platform instance (with x,y,w,h)
- or a plain object with x,y,w,h
This keeps it flexible.
*/
function overlapAABB(a, b) {
    return (
        a.x < b.x + b.w && a.x + a.w > b.x && a.y < b.y + b.h && a.y + a.h > b.y
    );
}

index.html:
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <title>Example 5 — Blob Platformer (JSON + Classes)</title>
        <link rel="stylesheet" href="style.css" />
    </head>

    <body>
        <!-- Load p5 first -->
        <script src="https://cdn.jsdelivr.net/npm/p5@1.9.0/lib/p5.min.js"></script>

        <!-- Load classes BEFORE sketch.js (order matters) -->
        <script src="platform.js"></script>
        <script src="worldLevel.js"></script>
        <script src="blobPlayer.js"></script>

        <!-- Main sketch -->
        <script src="sketch.js"></script>
    </body>
</html>

jsconfig.json:
{
    "include": [

```

```

        "*.js",
        "libraries/*.js",
        "c:\\\\Users\\\\david\\\\vscode\\\\extensions\\\\wmcicompsci.cs30-p5-1.9.3\\\\p5types\\\\global.d.ts"
    ]
}

levels.json:
{
  "schemaVersion": 1,
  "levels": [
    {
      "name": "Intro Steps",
      "gravity": 0.65,
      "jumpV": -11.0,
      "theme": { "bg": "#F0FOFO", "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 }
      ]
    }
  ]
}

platform.json:
/*
Platform.js (Example 5)

```

A Platform is a single axis-aligned rectangle in the world.

Why a class for something "simple"?

- It standardizes the shape of platform data.
- It makes later upgrades easy (e.g., moving platforms, icy platforms, spikes).
- It keeps drawing code in the object that knows what it is.

In JSON, platforms are stored like:

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

```

class Platform {
  constructor({ x, y, w, h }) {
    // Position is the top-left corner.
    this.x = x;
    this.y = y;

    // Size (width/height).
    this.w = w;
    this.h = h;
  }

  draw(fillColor) {
    fill(fillColor);
    rect(this.x, this.y, this.w, this.h);
  }
}


```

sketch.js:

```
/*

```

Week 4 — Example 5: Example 5: Blob Platformer (JSON + Classes)

Course: GBDA302

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 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();
  textAlign("center");
  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);
  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);
}

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]);
```

```
// 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);  
}
```

```
style.css:
```

```
html, body {  
    margin: 0;  
    padding: 0;  
}
```

```
canvas {  
    display: block;  
}
```

```
worldLevel.js:  
/*
```

```
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));
                }
            }
        }
    
```

With this example code, I want the goal to be reaching a circle ball, so collision hit would advance to the next level. I want to place the circle ball on one of the platforms that the blob would have to get to. Help me with the steps to do this.

Response:

1) Update `levels.json` to include a goal ball (circle)

Add a `goalBall` object in each level.

Example for your current level:

```
{
  "name": "Intro Steps",
  "gravity": 0.65,
  "jumpV": -11.0,
  "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
  "start": { "x": 80, "y": 220, "r": 26 },
  "goalBall": { "x": 565, "y": 220, "r": 14 },
  "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 }
  ]
}
```

✓ Put the ball where you want (ex: sitting on the platform at `{x: 520, y: 254}` — ball y should be a bit above that platform so it *looks* like it's resting on it).

For Level 2, do the same (different x/y):

```
"goalBall": { "x": 600, "y": 80, "r": 14 }
```

2) Store the `goalBall` in `WorldLevel`

Edit `worldLevel.js` → inside the constructor

Add this **after** `this.start = ...:`

```
// Goal ball (circle) the player must touch  
this.goalBall = levelJson.goalBall || null;
```

3) Draw the goal ball in `WorldLevel.drawWorld()`

Edit `worldLevel.js` → inside `drawWorld()` after `platforms` loop

Add:

```
// Draw goal ball  
if (this.goalBall) {  
    fill("#00D084"); // or use a theme color if you want  
    circle(this.goalBall.x, this.goalBall.y, this.goalBall.r * 2);  
}
```

Now you'll see the green ball.

4) Detect blob touching the goal ball and auto-advance

Collision for circles is super easy: **distance between centers \leq sum of radii**.

Edit `sketch.js` → add this helper function near the bottom

```
function circlesTouch(ax, ay, ar, bx, by, br) {  
    const dx = ax - bx;  
    const dy = ay - by;  
    const rSum = ar + br;  
    return dx * dx + dy * dy <= rSum * rSum;  
}
```

Edit `sketch.js` → inside `draw()` after `player.draw(...)`

Add:

```

// Auto-finish level if player touches the goal ball
if (world.goalBall) {
    const g = world.goalBall;
    if (circlesTouch(player.x, player.y, player.r, g.x, g.y, g.r)) {
        const next = (levelIndex + 1) % data.levels.length;
        loadLevel(next);
        return; // prevents double-loading same frame
    }
}

```

Done: touching the ball loads the next level automatically.

5) Put the ball “on a platform” (how to position it cleanly)

If a platform is { x: 520, y: 254, w: 90, h: 12 }:

- Pick a ball x somewhere within the platform:
x = 520 + 45 (middle) → 565
- Place ball y so it sits *on top* of the platform:
platform top is y = 254
ball center should be 254 - ballRadius
so if r = 14, then y = 254 - 14 = 240

So:

```
"goalBall": { "x": 565, "y": 240, "r": 14 }
```

That will look like it's resting on the platform.

Summary: files/areas you change

- **levels.json**

- add "goalBall": { "x", "y", "r" } in each level
 - **worldLevel.js**
 - constructor: `this.goalBall = ...`
 - drawWorld(): draw the circle
 - **sketch.js**
 - add `circlesTouch()` helper
 - in `draw()`: if touched → `loadLevel(next)`
-

If you want, I can also show a clean way to make the ball color part of the theme (like `theme.goal`) so each level can style it differently without hardcoding "#00D084".

Prompt:

How do I make it so player lose if collision hit with { "x": 0, "y": 370, "w": 640, "h": 36 },
 platform:
 {
 "name": "Level 3",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": { "bg": "#FOFOFO", "platform": "#C8C8C8", "blob": "#1478FF" },
 "start": { "x": 80, "y": 220, "r": 26 },
 "goalBall": { "x": 600, "y": 120, "r": 16 },

"platforms": [
 { "x": 0, "y": 370, "w": 640, "h": 36 },
 { "x": 30, "y": 240, "w": 100, "h": 12 },
 { "x": 280, "y": 220, "w": 100, "h": 12 },
 { "x": 450, "y": 190, "w": 100, "h": 12 },
 { "x": 540, "y": 135, "w": 100, "h": 12 }
]
 }

Response:

Goal

The platform

```
{ "x": 0, "y": 370, "w": 640, "h": 36 }
```

- acts as a **lose zone**
 - Any collision with it → **LOSE screen**
 - Other platforms behave normally
 - Goal ball → **WIN**
-

Step 1: Mark the lose platform in `levels.json`

Add a flag so we can identify it safely.

Update Level 3 platform entry

Change ONLY the bottom platform like this:

```
{
  "x": 0,
  "y": 370,
  "w": 640,
  "h": 36,
  "isDeath": true
}
```

Full Level 3 platforms section should now look like:

```
"platforms": [
  { "x": 0, "y": 370, "w": 640, "h": 36, "isDeath": true },
  { "x": 30, "y": 240, "w": 100, "h": 12 },
  { "x": 280, "y": 220, "w": 100, "h": 12 },
  { "x": 450, "y": 190, "w": 100, "h": 12 },
  { "x": 540, "y": 135, "w": 100, "h": 12 }
```

]

Step 2: Store `isDeath` in the Platform class

Edit `platform.js`

Update the constructor:

```
class Platform {  
  constructor({ x, y, w, h, isDeath = false }) {  
    this.x = x;  
    this.y = y;  
    this.w = w;  
    this.h = h;  
    this.isDeath = isDeath;  
  }  
  
  draw(fillColor) {  
    fill(fillColor);  
    rect(this.x, this.y, this.w, this.h);  
  }  
}
```

Now platforms can optionally be **deadly**.

Step 3: Detect collision with death platform

We **do NOT** change physics.

We just *observe* collisions after movement.

Edit `sketch.js`

Inside `draw()`, **after**:

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

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

Add this block:

```
// Level 3 lose condition: touching death platform
if (gameState === "play" && levelIndex === 2) {
    for (const p of world.platforms) {
        if (p.isDeath) {
            // Check blob vs platform AABB
            const blobBox = {
                x: player.x - player.r,
                y: player.y - player.r,
                w: player.r * 2,
                h: player.r * 2
            };

            if (overlapAABB(blobBox, p)) {
                gameState = "lose";
                return;
            }
        }
    }
}
```

Step 4: (Optional) Make the death platform visually obvious

In `worldLevel.js` → `drawWorld()`:

Replace:

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

With:

```
if (p.isDeath) {
    fill("#E63946"); // red = danger
```

```
rect(p.x, p.y, p.w, p.h);  
} else {  
  p.draw(color(this.theme.platform));  
}
```

Now players **instantly understand**: red = bad.

Prompt:

How should I do an end/ lose screen as well as a win screen?

Response:

1) Add a game state variable (top of sketch.js)

Put this near the top with your other globals:

```
let gameState = "play"; // "play" | "win" | "lose"
```

2) In loadLevel(i), reset the state

Inside `loadLevel(i)` add:

```
gameState = "play";
```

So every time you load/restart, you go back to playing.

3) In draw(), show end screen instead of gameplay

At the **very top** of `draw()`:

```
function draw() {  
  if (gameState !== "play") {  
    drawEndScreen();  
    return;  
  }
```

```

        }

    // normal gameplay below...
    world.drawWorld();
    player.update(world.platforms);
    player.draw(world.theme.blob);

    // HUD
    fill(0);
    text(world.name, 10, 18);
    text("Move: A/D or ←/→ • Jump: Space/W/↑", 10, 36);

    // (your win/lose checks go after this)
}

```

4) Make a win/lose screen function

Add this at the bottom of `sketch.js`:

```

function drawEndScreen() {
    background(0);
    fill(255);
    textAlign(CENTER, CENTER);

    textSize(48);
    if (gameState === "win") {
        text("YOU WIN!", width / 2, height / 2 - 30);
    } else {
        text("YOU LOSE!", width / 2, height / 2 - 30);
    }

    textSize(18);
    text("Press R to restart", width / 2, height / 2 + 30);
    text("Press 1 / 2 / 3 to select a level", width / 2, height / 2 +
60);
}

```

5) Set gameState to win when touching the goal ball (Level 3)

Wherever you detect touching the goal ball, do:

```
if (levelIndex === 2) {  
    gameState = "win";  
    return;  
}
```

(For Levels 1–2 you can still `loadLevel(next)`.)

Example block:

```
if (gameState === "play" && world.goalBall) {  
    const g = world.goalBall;  
  
    if (circlesTouch(player.x, player.y, player.r, g.x, g.y, g.r)) {  
        if (levelIndex === 2) {  
            gameState = "win";  
            return;  
        } else {  
            loadLevel(levelIndex + 1);  
            return;  
        }  
    }  
}
```

6) Set gameState to lose when touching the death platform (Level 3)

In your death platform collision check, just do:

```
gameState = "lose";  
return;
```

7) Add restart controls in `keyPressed()`

In `sketch.js` inside `keyPressed()` add:

```
// Restart
if (key === "r" || key === "R") {
    loadLevel(levelIndex); // restart current level
}

// Level select (optional)
if (key === "1") loadLevel(0);
if (key === "2") loadLevel(1);
if (key === "3") loadLevel(2);
```

Prompt:

levels.json:

```
{
  "schemaVersion": 1,
  "levels": [
    {
      "name": "Level 1",
      "gravity": 0.65,
      "jumpV": -11.0,
      "theme": { "bg": "#FOFOFO", "platform": "#C8C8C8", "blob": "#1478FF" },
      "start": { "x": 80, "y": 220, "r": 26 },
      "goalBall": { "x": 565, "y": 238, "r": 16 },
      "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 }
      ],
      "name": "Level 2",
      "gravity": 0.65,
      "jumpV": -11.0,
```

```

    "theme": { "bg": "#FOFOFO", "platform": "#C8C8C8", "blob": "#1478FF" },
                "start": { "x": 80, "y": 220, "r": 26 },

        "goalBall": { "x": 380, "y": 154, "r": 16 },

                "platforms": [
                    { "x": 0, "y": 370, "w": 640, "h": 36 },
                    { "x": 120, "y": 330, "w": 100, "h": 12 },
                    { "x": 300, "y": 300, "w": 100, "h": 12 },
                    { "x": 440, "y": 250, "w": 100, "h": 12 },
                    { "x": 330, "y": 165, "w": 100, "h": 12 }
                ]
            },
            {
                "name": "Level 3",
                "gravity": 0.65,
                "jumpV": -11.0,
            }

    "theme": { "bg": "#FOFOFO", "platform": "#C8C8C8", "blob": "#1478FF" },
                "start": { "x": 80, "y": 220, "r": 26 },

        "goalBall": { "x": 600, "y": 120, "r": 16 },

                "platforms": [
                    { "x": 0, "y": 370, "w": 640, "h": 36, "isDeath": true },
                        { "x": 30, "y": 240, "w": 100, "h": 12 },
                        { "x": 280, "y": 220, "w": 100, "h": 12 },
                        { "x": 450, "y": 190, "w": 100, "h": 12 },
                        { "x": 540, "y": 135, "w": 100, "h": 12 }
                ]
            }
        ]
    }

platform.js:
/*
Platform.js (Example 5)

```

A Platform is a single axis-aligned rectangle in the world.

- Why a class for something "simple"?
- It standardizes the shape of platform data.
- It makes later upgrades easy (e.g., moving platforms, icy platforms, spikes).
- It keeps drawing code in the object that knows what it is.

In JSON, platforms are stored like:
{ "x": 0, "y": 324, "w": 640, "h": 36 }
*/

```
class Platform {
```

```

constructor({ x, y, w, h }) {
  // Position is the top-left corner.
  this.x = x;
  this.y = y;

  // Size (width/height).
  this.w = w;
  this.h = h;

  this.isDeath = isDeath;
}

draw(fillColor) {
  fill(fillColor);
  rect(this.x, this.y, this.w, this.h);
}
}

```

/*

Week 4 — Example 5:
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 world; // WorldLevel instance (current level)
let player; // BlobPlayer instance
let gameState = "play";

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();
    textAlign("sans-serif");
    textSize(14);
}

function draw() {

    if (gameState !== "play") {
        drawEndScreen();
        return;
    }

    // 1) Draw the world (background + platforms)
    world.drawWorld();

    // 2) Update and draw the player on top of the world
    player.update(world.platforms);
    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);

    if (world.goalBall) {
        const g = world.goalBall;

        if (circlesTouch(player.x, player.y, player.r, g.x, g.y, g.r)) {
            if (levelIndex === 2) {
                gameState = "win"; // Level 3 → WIN
                return;
            } else {
                loadLevel(levelIndex + 1); // Levels 1–2 → next level
                return;
            }
        }
    }

    if (gameState === "play" && levelIndex === 2) {
        for (const p of world.platforms) {
            if (p.isDeath) {
                // Check blob vs platform AABB
                const blobBox = {
                    x: player.x - player.r,

```

```

        y: player.y - player.r,
        w: player.r * 2,
        h: player.r * 2
    };

    if (overlapAABB(blobBox, p)) {
        gameState = "lose";
        return;
    }
}

function keyPressed() {
    // Jump keys
    if (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) {
        player.jump();
    }

    if (key === "r" || key === "R") {
        loadLevel(levelIndex); // restart current level
    }
}

/*
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]);

    // 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);

    gameState = "play";
}

```

```

function circlesTouch(ax, ay, ar, bx, by, br) {
    const dx = ax - bx;
    const dy = ay - by;
    const rSum = ar + br;
    return dx * dx + dy * dy <= rSum * rSum;
}

```

```

function drawEndScreen() {
    background(0);
    fill(255);
    textAlign(CENTER, CENTER);

    textSize(48);
    if (gameState === "win") {
        text("YOU WIN!", width / 2, height / 2 - 30);
    } else {
        text("YOU LOSE!", width / 2, height / 2 - 30);
    }

    textSize(18);
    text("Press R to restart", width / 2, height / 2 + 30);
}
}

```

*

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: "#FOFOFO", 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,
                };

                this.goalBall = levelJson.goalBall || null;

                // 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));
      }
      if(this.goalBall) {
        fill("#56ecfc");
        circle(this.goalBall.x, this.goalBall.y, this.goalBall.r * 2);
      }
    }
}
```

Help me debug, the code's not running

Response:

You've got **two hard errors** that will stop the whole sketch from running, plus **one brace/indent issue** that breaks `draw()`.

Here's the exact debug + the corrected code blocks.

✓ 1) Platform constructor is broken (`isDeath` is undefined)

In `platform.js` you wrote:

```
this.isDeath = isDeath;
```

But `isDeath` is not a variable unless you **take it from the constructor params**.

Fix `platform.js` (replace your whole class with this)

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

```

        this.isDeath = isDeath; // now it works
    }

draw(fillColor) {
    fill(fillColor);
    rect(this.x, this.y, this.w, this.h);
}
}

```

That alone will prevent a crash.

2) Your `draw()` function has missing closing braces

Right now, your `if (world.goalBall)` block is missing a closing `}` and your braces are nested wrong, so JS can't parse the file.

Fix `sketch.js draw()` (copy-paste this whole function)

```

function draw() {
    if (gameState !== "play") {
        drawEndScreen();
        return;
    }

    // 1) Draw the world (background + platforms)
    world.drawWorld();

    // 2) Update and draw the player on top of the world
    player.update(world.platforms);
    player.draw(world.theme.blob);

    // 3) Goal ball -> win / next level
    if (world.goalBall) {
        const g = world.goalBall;

        if (circlesTouch(player.x, player.y, player.r, g.x, g.y, g.r)) {
            if (levelIndex === 2) {
                gameState = "win"; // Level 3 -> WIN
            }
        }
    }
}

```

```

        return;
    } else {
        loadLevel(levelIndex + 1); // Level 1-2 -> next
        return;
    }
}

// 4) Death platform -> lose (Level 3 only)
if (levelIndex === 2) {
    for (const p of world.platforms) {
        if (p.isDeath) {
            const blobBox = {
                x: player.x - player.r,
                y: player.y - player.r,
                w: player.r * 2,
                h: player.r * 2,
            };

            if (overlapAABB(blobBox, p)) {
                gameState = "lose";
                return;
            }
        }
    }
}

// 5) HUD (draw last)
fill(0);
text(world.name, 10, 18);
text("Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N", 10, 36);
}

```

Key fixes:

- all braces close correctly
- win check and lose check are separate
- HUD is drawn last (so it still shows while playing)