Project/Assignment Decisions

- For Side Quest 4, I took the code from “example 5” and modified it using generative AI to understand the concepts and structure of the code, and also used AI to generate complex codes to create levels and other features in my game.
- There were moments where generative AI created codes that were actually a lot more complicated to understand (for example, it added many more functions to change the colours of obstacles in each level, when I could have just changed the colours in the json file). Since the AI would also generate easier solutions that I could understand, I chose to use the simpler solutions so that the structure code would not become too confusing.
- While I wanted to add many more complex features, I realized that it would make the code a lot more complicated to understand, which would make troubleshooting a lot more complex in the long run. After trying a few complex codes, I scraped the ideas and decided to stick with simpler designs for a clear learning experience.

Process Evidence

Name: Jade Sun

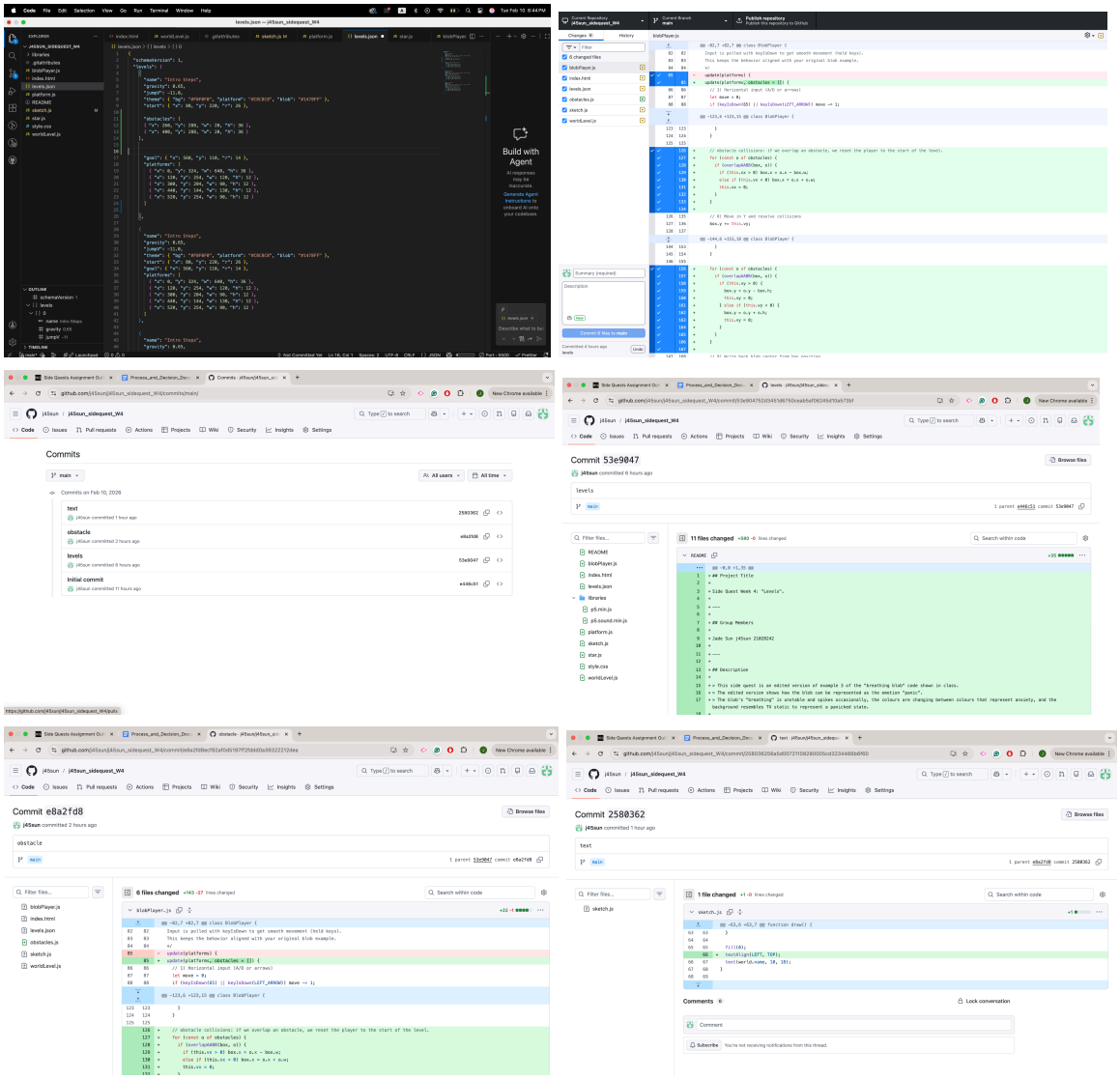
Role: Coding with Generative AI

Primary Responsibility: Using generative AI to learn and write code for complex functions in Visual Studio Code

Goal of Work Session: The goal for my work session was to use generative AI to learn how I can use loops to create obstacles, and how I can automatically load a new level after completing a previous level.

Tools used: ChatGPT, Claude

Process Evidence:



GenAI Documentation

Date used: January 25, 2026

Tool Disclosure: ChatGPT 5.2, Sonnet 4.5 (Claude)

Purpose of Use: To learn the functions of json and class files, learn the structure of the given examples from class, and generate code for complex design ideas.

Summary of Interaction: Claude was able to help with the initial learning process of the side quest, such as debugging (explaining why nothing showed up when I opened the original code) and further explaining how json files and classes worked. ChatGPT was able to help with troubleshooting and generating my ideas into codes, and helped me learn where I should place certain sections of codes so that I could fully understand its purpose without simply copy and pasting. Both of the AI tools were able to help clearly

explain how the structure of the code from example 5 worked, as I had difficulty understanding the concepts solely from the slides.

Human Decision Points: ChatGPT provided me with many solutions for design ideas that I had. I chose the solutions that were easier to implement into the code, and rejected long and complex codes that were not talked about in class to avoid confusions while troubleshooting in the long run. The design and aesthetic choices that were originally generated by AI, such as colours and obstacle dimensions, were all refined using human decisions.

Integrity & Verification Note: ChatGPT made assumptions that the levels did not need to be made with json files, and generated levels using other functions. Furthermore, some codes that were generated by ChatGPT were too complex, and I did not think it was appropriate for using as it did not align with course concepts.

Scope of GenAI Use: No generative AI was used to generate design ideas or text for written portions of this side quest (README.md and process/design document).

Limitations of Misfires: ChatGPT would explain how the code for my design choices works, but it was difficult to understand where it would place the code. I had to constantly ask ChatGPT to explain where I should put the code that it generated for me, and why I had to put it there. Because of this limitation, I ran into many issues that would stop my code from working.

Appendix

Prompt 1 (Claude):

why is there a white blank screen when i try to open in live serve in visual studio code? i imported different js files and json files from an example from class to try and load into live server

Answer 1:

A blank white screen in Live Server usually means JavaScript is failing silently. Here are the most common culprits:

Two quick questions:

1. Can you check your browser's console? (Right-click → Inspect → Console tab) - Are there any error messages in red?
2. What file are you opening in Live Server? Is it an HTML file, or are you trying to open a .js file directly?

Common issues when importing files from class examples:

- File paths are broken - If the example used relative paths like ./data.json or ../scripts/app.js, they might not match your folder structure

- Missing CORS headers - JSON files often can't be loaded locally without a server (which Live Server should handle, but depends on how you're importing)
- Module vs regular script - If using import/export, you need <script type="module"> in your HTML

If you can share:

- The error from the console, or
- Your HTML file and how you're importing the JS/JSON

I can pinpoint exactly what's wrong!

Prompt 2 (ChatGPT):

without using emojis, please explain how the different code structure works, while elaborating on json files and class files:

<!doctype html> <html lang="en"> <head> <meta charset="utf-8" /> <meta name="viewport" content="width=device-width, initial-scale=1" /> <title>j45sun Sidequest Week 4</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="blobPlayer.js"></script> <script src="worldLevel.js"></script> <!-- Main sketch --> <script src="sketch.js"></script> </body> </html> this is style.css: html, body { margin: 0; padding: 0; } canvas { display: block; } this is blobPLayer.js (player class): /* 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); } this is platforms.js (platform class): /* 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); } } this is 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)); } } } this is sketch.js: /* 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 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); 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); } this is levels.json: { "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 }] }] }

Answer 2:

Great question — this project is actually a very clean example of how to structure a small game in a more "real-world" way instead of putting everything in one giant sketch. I'll walk through how the pieces fit together, then zoom in on JSON files and class files, and finally explain why this structure is useful.

---

1. Big-picture structure (what talks to what)
Think of the project as having three layers:

1. Data (levels.json)
   → Pure data: numbers, colors, positions. No logic.

2. Logic / Objects (class files)
   → Rules and behavior: how platforms work, how the player moves, how a level is built.

3. Orchestration (sketch.js)
   → The conductor: loads data, creates objects, updates and draws everything every frame.

index.html just loads everything in the correct order so p5 can run.

---

2. index.html: why load order matters

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

This order is important because:

- WorldLevel uses Platform
- sketch.js uses WorldLevel and BlobPlayer
- JavaScript must know what a class is before you use it

So the rule is:
dependencies first, main sketch last
index.html itself does not contain game logic — it's just the loader.

---

3. JSON: what levels.json actually does

What JSON is for
levels.json is configuration, not code.
It answers questions like:

- Where are platforms?
- What color is the level?
- How strong is gravity?
- Where does the player spawn?

JSON cannot:

- Move the player

- Detect collisions
- Draw shapes

It only describes the level.

Example from your file

```
{
  "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 }
  ]
}
```

This says:

- Gravity should be 0.65
- The blob jumps with velocity -11
- The background color is light gray
- There is a platform at (0, 324) that is 640 wide

Nothing here explains how gravity works — that's the job of the player class.

How JSON enters the game

In sketch.js:

```
function preload() {
  data = loadJSON("levels.json");
}
```

Now data.levels[0] is a normal JavaScript object containing that JSON.

---

4. WorldLevel.js: turning data into a playable level
WorldLevel is a translator between JSON and gameplay.
What it owns

- Theme colors
- Gravity and jump settings
- Player spawn position
- Platform objects

Key idea
JSON platforms look like this:
{ "x": 120, "y": 254, "w": 120, "h": 12 }

But your game doesn't want raw objects — it wants Platform instances.
So this line is crucial:

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

That means:

- Take each platform object from JSON
- Convert it into a Platform class
- Store them in an array

Now every platform has:

- Data (x, y, w, h)
- Behavior (draw())

Canvas sizing logic
These methods:
inferWidth()

inferHeight()

Scan all platforms and find how big the world needs to be.
This allows:

- Different levels to have different sizes
- No hard-coded canvas dimensions in sketch.js

---

5. Platform.js: small, focused class
The Platform class is intentionally simple.
It does only two things:

1. Store position and size
2. Draw itself

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

Collision logic is not here — that belongs to the player.
This separation is important:

- Platforms are passive
- The player is the thing that moves and collides

---

6. BlobPlayer.js: the most complex class
This class owns everything that changes every frame.
What the player controls

- Position and velocity
- Gravity and jump force
- Grounded state
- Input handling

- Collision resolution
- Drawing the blob shape

Update flow (high level)
Every frame, update() does this in order:

1. Read keyboard input
2. Apply acceleration
3. Apply friction
4. Clamp speed
5. Apply gravity
6. Build a collision box
7. Move in X and resolve collisions
8. Move in Y and resolve collisions
9. Update position
10. Update animation time

This order is very deliberate and mirrors real platformer physics.

Why collision is AABB instead of circles
Even though the blob looks round, collision is done with a rectangle:

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

This makes:

- Collision simpler
- Platform snapping easier
- Movement more predictable

The blob is “visually round, physically square.”

---

7. sketch.js: the conductor
sketch.js does not know details of physics or drawing shapes.
Its job is to:

- Load data
- Create objects
- Call methods in the right order

---

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

This reads almost like English:

1. Draw the level
2. Update the player using the level’s platforms
3. Draw the player using the level’s blob color

That clarity is the payoff of good structure.

---

loadLevel()
This function connects everything:
world = new WorldLevel(data.levels[levelIndex]);
resizeCanvas(world.inferWidth(), world.inferHeight());
player.spawnFromLevel(world);

It:

- Builds a level from JSON
- Resizes the canvas
- Applies physics + spawn info to the player

No duplication, no hard-coded values.

---

8. Why this structure matters
Compared to a single-file sketch, this setup:

- Makes adding new levels trivial (just edit JSON)
- Keeps logic readable and testable
- Mirrors how real games and apps are structured
- Separates what the game is from how it runs

If you wanted to:

- Add enemies → new class
- Add moving platforms → extend Platform
- Add more levels → edit JSON only

Nothing else has to be rewritten.

---

9. One-sentence summary
Your project separates data (JSON), behavior (classes), and control flow (sketch.js) so the game is easier to extend, debug, and reason about — exactly how larger interactive projects are built.

Prompt 3:
how can i make it so that when the blob collects a spinning star, it moves on to the next level?

Answer 3:

Great question — this is a classic and very "correct" next step for a platformer.

Right now, your game only advances levels when you press **N**. To make the game advance **when the blob collects a spinning star**, you need four things:

1. A star defined in levels.json
2. A Star class (draw + spin)
3. Collision detection between blob and star
4. Logic to load the next level when collected

I'll walk through this cleanly, step by step, and keep it beginner-friendly.

---

1. Add a star to each level in levels.json

Add a new property to each level, for example "goal".

Example (add this inside each level object):

```
"goal": { "x": 560, "y": 110, "r": 14 }
```

So one full level becomes:

```
{
 "name": "Level 1",
 "gravity": 0.65,
 "jumpV": -11.0,
 "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
 "start": { "x": 80, "y": 220, "r": 26 },
 "goal": { "x": 560, "y": 110, "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 }
  ]
}
```

Do this for each level.

This star is your “exit”.

---

2. Create a Star class (new file: star.js)

Create a new file called star.js.

```
class Star {
  constructor({ x, y, r }) {
    this.x = x;
    this.y = y;
    this.r = r;
    this.angle = 0;
  }

  update() {
    this.angle += 0.05;
  }

  draw() {
    push();
    translate(this.x, this.y);
    rotate(this.angle);
    fill(255, 200, 0);
    beginShape();
    for (let i = 0; i < 5; i++) {
      let a = (i * TWO_PI) / 5;
      vertex(cos(a) * this.r, sin(a) * this.r);
      vertex(
        cos(a + PI / 5) * (this.r * 0.5),
        sin(a + PI / 5) * (this.r * 0.5)
      );
```

```
    }
    endShape(CLOSE);
    pop();
  }

  isCollectedBy(player) {
    let d = dist(this.x, this.y, player.x, player.y);
    return d < this.r + player.r;
  }
}
```

This gives you:

- A spinning star
- A simple circle-based collision check

---

3. Load the star in WorldLevel

Modify worldLevel.js.

Add this in the constructor:

```
this.goal = levelJson.goal
  ? new Star(levelJson.goal)
  : null;
```

Add this to drawWorld() (after platforms):

```
if (this.goal) {
  this.goal.update();
  this.goal.draw();
}
```

Now each level knows whether it has a goal star.

---

4. Tell the game to advance when collected

Modify draw() in sketch.js.

After updating the player, add this:

```
if (world.goal && world.goal.isCollectedBy(player)) {
  const next = (levelIndex + 1) % data.levels.length;
  loadLevel(next);
}
```

So your draw() function becomes (simplified):

```
function draw() {
  world.drawWorld();

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

  if (world.goal && world.goal.isCollectedBy(player)) {
    const next = (levelIndex + 1) % data.levels.length;
    loadLevel(next);
  }

  fill(0);
  text(world.name, 10, 18);
}
```

---

5. Load star.js in index.html

Add this line before worldLevel.js:

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

Order matters, because WorldLevel uses Star.

Mental model (important)

- levels.json defines **what exists**
- WorldLevel builds objects from that data
- Star knows how to draw and detect collision
- sketch.js decides **what happens when the star is collected**

You now have a proper "goal-based" level system instead of a debug key.

Optional extensions (if you want to go further)

- Play a sound when collected
- Add multiple stars (collect all before advancing)
- Animate the blob when touching the star
- Lock the next level until star is collected

If you want, I can help you implement any of those next.

Prompt 4:
where do i put the code in step 3?

Answer 4:
Good catch — this is exactly the right thing to ask.
Step 3 goes **inside** `worldLevel.js`, in two very specific places.

I'll point to *exactly where* to put each piece.

---

1. Open `worldLevel.js`

You are editing **this file only** for step 3.

---

2. Where to put the constructor code

Find the **constructor** in `WorldLevel`:

```
class WorldLevel {
  constructor(levelJson) {
```

Inside that constructor, scroll down until you see this part:

```
// Convert raw platform objects into Platform instances.
this.platforms = (levelJson.platforms || []).map((p) => new
Platform(p));
```

Put the goal code directly after that

Add this **right below it**:

```
this.goal = levelJson.goal
  ? new Star(levelJson.goal)
  : null;
```

So the bottom of your constructor should look like this:

```
// Convert raw platform objects into Platform instances.
this.platforms = (levelJson.platforms || []).map((p) => new
Platform(p));

// Create the goal star (if this level has one)
```

```
this.goal = levelJson.goal
  ? new Star(levelJson.goal)
  : null;
```

That's it for the constructor.

---

3. Where to put the draw code

Still in `worldLevel.js`, find this method:

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

Add the goal drawing inside this function, after platforms

Change it to:

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

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

  if (this.goal) {
    this.goal.update();
    this.goal.draw();
  }
}
```

---

4. Quick checklist (to avoid errors)

Make sure all of these are true:

- `star.js` exists
- `star.js` is loaded **before** `worldLevel.js` in `index.html`
- You did not put this code in `sketch.js`
- You did not put this code outside the class

Correct order in `index.html`:

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

Why this works

- `WorldLevel` owns level-specific things (platforms, theme, goal)
- `Star` is just a visual + collision object
- `sketch.js` only checks **if** the goal was collected

You're keeping responsibilities clean, which is exactly what your instructors want to see.

If you want, next I can:

- Add a visual “level complete” delay
- Prevent instant re-triggering
- Support multiple stars per level
- Or help debug if nothing appears on screen

Prompt 5:
how can i add obstacles that block the blob with loops in each level

Answer 5:

Perfect — now you’re moving from simple platforms to **dynamic obstacles**. I’ll explain a clean way to add them using **loops in `levels.json`**, **classes**, and your existing `WorldLevel` system.

---

1. Decide how obstacles differ from platforms

- Platforms: you can **stand on** them (like ground).
- Obstacles: you **cannot pass through** them (like spikes, walls).

You can represent them as **rectangles** just like platforms, but your player will **collide differently** (maybe instantly stops, or takes damage).

For now, let’s just **block the player** like a wall.

---

2. Update `levels.json` to include obstacles

Add a new array called `"obstacles"` in each level. Example:

```
"obstacles": [
```

```
  { "x": 200, "y": 280, "w": 20, "h": 44 },
  { "x": 350, "y": 180, "w": 20, "h": 24 }
]
```

Using a loop for repeated obstacles

If you want **a line of spikes** or walls, don't copy/paste manually — just generate them with a small JS snippet **before saving the JSON**:

Example in JS (not part of game code, just for generating JSON):

```
const obstacles = [];
for (let i = 0; i < 5; i++) {
  obstacles.push({ x: 100 + i * 50, y: 300, w: 20, h: 24 });
}
console.log(JSON.stringify(obstacles, null, 2));
```

Copy the result into `"obstacles"`.

---

3. Create an `Obstacle` class

Add a new file `obstacle.js`:

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

```
    this.w = w;
    this.h = h;
  }

  draw() {
    fill(200, 50, 50); // red for danger
    rect(this.x, this.y, this.w, this.h);
  }
}
```

---

4. Update `WorldLevel` to load obstacles

Inside the constructor of `WorldLevel`, after platforms:

```
this.obstacles = (levelJson.obstacles || []).map(o => new
Obstacle(o));
```

---

5. Draw obstacles in `drawWorld()`

Inside `drawWorld()` in `WorldLevel`:

```
for (const o of this.obstacles) {
  o.draw();
```

```
}
```

Place this **after platforms** so they appear above platforms visually.

---

6. Block the blob in `BlobPlayer.update()`

Right now, the player only collides with `platforms`.
Modify `update(platforms)` to also accept obstacles:

```
update(platforms, obstacles = []) {
  // existing horizontal movement code...

  // horizontal collision with platforms
  for (const s of platforms) { ... }

  // horizontal collision with obstacles
  for (const o of obstacles) {
    if (overlapAABB(box, o)) {
      if (this.vx > 0) box.x = o.x - box.w;
      else if (this.vx < 0) box.x = o.x + o.w;
      this.vx = 0;
    }
  }
```

```
  // vertical collision with platforms
  for (const s of platforms) { ... }

  // vertical collision with obstacles
  for (const o of obstacles) {
    if (overlapAABB(box, o)) {
      if (this.vy > 0) box.y = o.y - box.h;
      else if (this.vy < 0) box.y = o.y + o.h;
      this.vy = 0;
    }
  }

  // write back x, y ...
}
```

---

7. Pass obstacles from `sketch.js`

In `draw()`:

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

✅ Result

- You can now add **any number of obstacles** per level via JSON.
- You can use loops **outside the game** to generate many obstacles in a line.
- The player will **collide and stop** when touching them.

If you want, I can next show a **shortcut to generate complex patterns in JSON** so each level can have **spinning stars + lines of obstacles + platforms** automatically, without writing every obstacle by hand.

Do you want me to show that?

Prompt 6:
why am i getting a "error: comma required" when adding obstacles into levels.json

Answer 6:
his is a classic JSON gotcha, and the error message is actually pointing to something very small and specific. The key thing to remember is:

JSON is extremely strict. One missing or extra comma will break the whole file.

I'll explain *why* this happens, then show you exactly what to check and how to fix it.

---

1. What "comma required" actually means

When the JSON parser says **"comma required"**, it means:

> "I just finished reading one value, and I expected either a comma `,` or a closing brace `}` — but I got something else."

This almost always happens when:

- You forgot a comma **between properties**

- You forgot a comma **between objects in an array**
- You added a new property (like `obstacles`) but didn't separate it correctly

---

2. The most common mistake when adding `obstacles`

❌ This will cause "comma required"

```
"platforms": [
  { "x": 0, "y": 324, "w": 640, "h": 36 }
]
"obstacles": [
  { "x": 200, "y": 280, "w": 20, "h": 44 }
]
```

Problem:
There is **no comma after the `platforms` array**.

✅ Correct version

```
"platforms": [
  { "x": 0, "y": 324, "w": 640, "h": 36 }
],
"obstacles": [
  { "x": 200, "y": 280, "w": 20, "h": 44 }
```

```
]
```

That comma after `]` is mandatory.

---

3. Another very common mistake: trailing commas

JSON does **not** allow trailing commas.

❌ This will break JSON

```
"obstacles": [
  { "x": 200, "y": 280, "w": 20, "h": 44 },
]
```

That last comma after the object is illegal.

✅ Correct version

```
"obstacles": [
  { "x": 200, "y": 280, "w": 20, "h": 44 }
]
```

---

4. Check this exact structure (safe template)

Here is a **known-good template** you can compare against:

```
{
  "name": "Level 1",
  "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 }
  ],

  "obstacles": [
    { "x": 200, "y": 280, "w": 20, "h": 44 }
  ]
}
```

Notice:

- Comma after `platforms` block
- No comma after the last item in arrays
- No comma after the last property in the object

---

5. Easiest way to catch this instantly

Before reloading the game:

- Paste `levels.json` into **jsonlint.com**
- Or use VS Code's red squiggly underline
- Or collapse sections in your editor to visually check commas

---

Mental rule to remember

> **In JSON:**
>
> - Items in arrays → separated by commas
> - Properties in objects → separated by commas
> - Last item → no comma

Once you internalize that, this error disappears forever.

If you want, paste **your exact `levels.json` with obstacles** here and I'll point to the exact character that's breaking it.

Prompt 7:
how can i create an end screen after player collects the star of the third level with a button that says "replay"

Answer 7:
Great capstone feature. You can do this cleanly without breaking your level system by adding **one game state** and a **simple button**.

Below is a clear, minimal approach that fits your current structure.

High-level idea

Right now your game only has one implicit state: “playing a level”.

You will add:

- a **gameState variable**
- an **end screen** that draws instead of the world
- a **Replay button** that resets everything

States:

- `"play"` → normal gameplay
- `"end"` → final screen after level 3 star is collected

---

1. Add a game state variable

At the **top of `sketch.js`**, add:

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

---

2. Detect “final level star collected”

In your `draw()` function, you already have logic like this:

```
if (world.goal && world.goal.isCollectedBy(player)) {
  const next = (levelIndex + 1) % data.levels.length;
  loadLevel(next);
}
```

Replace it with this:

```
if (world.goal && world.goal.isCollectedBy(player)) {
  if (levelIndex === data.levels.length - 1) {
    gameState = "end";
  } else {
    loadLevel(levelIndex + 1);
  }
}
```

This means:

- If the player collects the star on **level 3**, go to the end screen
- Otherwise, go to the next level as usual

---

3. Stop drawing the world when the game ends

At the **top of `draw()`**, add this guard:

```
if (gameState === "end") {
```

```
  drawEndScreen();
  return;
}
```

Your `draw()` now looks conceptually like:

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

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

  // star check happens here
}
```

---

4. Create the end screen function

Still in `sketch.js`, add this **below** `draw()`:

```
function drawEndScreen() {
  background(240);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("You Win!", width / 2, height / 2 - 60);

  drawReplayButton();
}
```

---

5. Draw the Replay button

Add this helper function:

```
function drawReplayButton() {
  const bw = 160;
  const bh = 50;
  const bx = width / 2 - bw / 2;
  const by = height / 2;

  // button background
```

```
  fill(200);
  rect(bx, by, bw, bh, 8);

  // button text
  fill(0);
  textSize(20);
  text("Replay", width / 2, by + bh / 2);
}
```

---

6. Make the Replay button clickable

Add this to `sketch.js`:

```
function mousePressed() {
  if (gameState !== "end") return;

  const bw = 160;
  const bh = 50;
  const bx = width / 2 - bw / 2;
  const by = height / 2;

  if (
```

```
    mouseX > bx &&
    mouseX < bx + bw &&
    mouseY > by &&
    mouseY < by + bh
  ) {
    restartGame();
  }
}
```

---

7. Reset everything on replay

Add this function:

```
function restartGame() {
  gameState = "play";
  levelIndex = 0;
  loadLevel(0);
}
```

This:

- resets the level index
- reloads level 1

- puts the player back at the start

Final mental model

- `levels.json` still controls progression
- collecting the last star switches the game to `"end"`
- the end screen is just another draw mode
- the replay button resets the state

This is exactly how real games handle menus and endings, just simplified.

Optional polish (easy upgrades)

If you want to go further later:

- Fade-in end screen
- Play a victory sound
- Animate the star exploding
- Disable physics when ending
- Add a “Thanks for playing” message

If you want help polishing the end screen visually or structuring this for grading clarity, I can help with that next.

Prompt 8:
why is the button not working? i put the functions into draw()

Answer 8:
This explains it exactly.
The button isn't working because you put the functions *inside* `draw()`.

In JavaScript (and p5.js), **functions must be defined at the top level**, not inside `draw()`.

When you put them inside `draw()`:

- They get redefined every frame
- `mousePressed()` is never registered properly
- p5 cannot call it as an event handler

So clicks are ignored.

What MUST be outside `draw()`

These functions **must NOT be inside `draw()`**:

```
drawEndScreen()

drawReplayButton()

mousePressed()

restartGame()
```

They should sit **below `draw()`**, like this:

```
function draw() {
  if (gameState === "end") {
    drawEndScreen();
    return;
```

```
  }

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

Then later in the file:

```
function drawEndScreen() {
  ...
}

function drawReplayButton() {
  ...
}

function mousePressed() {
  ...
}

function restartGame() {
  ...
```

```
}
```

Why `mousePressed()` specifically breaks

p5.js looks for **global functions** named exactly:

```
function mousePressed() {}
```

If it's inside `draw()`:

```
function draw() {
  function mousePressed() { ... } // ❌
}
```

p5 **cannot see it**, so clicks never fire.

Quick sanity check (do this)

Add this temporarily:

```
function mousePressed() {
  console.log("clicked");
}
```

If you don't see `"clicked"` in the console when you click:

- The function is still inside `draw()`
- Or there's a syntax error above it

One more common mistake

Make sure you didn't accidentally do this:

```
if (gameState === "end") {
  function drawReplayButton() { ... } // ❌
}
```

Functions cannot live inside conditionals either.

Mental rule to remember

> `draw()` is for **calling** functions,
> not **defining** them.

Once you move those functions out, the button will immediately start working.

If it still doesn't after moving them, paste your `sketch.js` and I'll point to the exact line causing the issue.

Prompt 9:
how to change the colour of obstacles in each level?

Answer 9:

You're very close already. The clean way to do this is:

Put colour in the level data → pass it into obstacles → draw using that colour.

Right now your obstacles are always red because the colour is hard-coded in `Obstacle.draw()`.

Option A (recommended): one obstacle colour per level

This keeps things simple and matches how you already handle themes.

1. Add an obstacle colour to the level theme (JSON)

Update your level like this:

```
"theme": {
  "bg": "#bde0fe",
  "platform": "#023e8a",
  "blob": "#1478FF",
  "obstacle": "#9d0208"
}
```

You don't need to touch the obstacle objects themselves.

---

2. Store the obstacle colour in `WorldLevel`

In `worldLevel.js`, inside the constructor, your theme already exists:

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

Nothing else needed here — `theme.obstacle` will now exist automatically.

---

3. Pass the colour when drawing obstacles

In `drawWorld()` (still in `worldLevel.js`), update the obstacle drawing loop:

```
for (const o of this.obstacles) {
  o.draw(color(this.theme.obstacle));
}
```

---

4. Update `Obstacle.draw()`

In `obstacle.js`, change `draw()` to accept a colour:

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

Now:

- Each level controls obstacle colour
- No hard-coded colours
- Matches your existing theme system

Option B: different colours per obstacle (more advanced)

If you want **individual obstacles** to have different colours:

JSON

```
"obstacles": [
  { "x": 260, "y": 188, "w": 20, "h": 136, "color": "#d00000" },
  { "x": 400, "y": 120, "w": 20, "h": 100, "color": "#ff8800" }
]
```

`Obstacle` class

```
constructor({ x, y, w, h, color }) {
  this.x = x;
  this.y = y;
  this.w = w;
  this.h = h;
  this.color = color || "#ff0000";
}

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

This is useful for spikes vs walls vs lava, but it's more data-heavy.

Which should you use?

For a course project, **Option A is ideal**:

- Cleaner JSON
- Consistent visual language
- Easier to mark and extend

Option B is better if obstacles have **different meanings**.

Mental model to remember

- `levels.json` decides *what things look like*

- Classes decide *how things behave*
- Never hard-code visuals if the level might change them

If you want next:

- Animated obstacles
- Flashing danger colours
- Obstacles that change colour per difficulty
- Lava vs wall vs spike logic

Just say which one.