Project/Assignment Decisions

- For Side Quest 5, I modified “example 5” from week 5’s lecture by using generative AI to create advanced visuals that resemble the moon.
- I decided to put the codes of example 5 into generative AI to understand how the codes work together, as the slides were a bit vague and there were parts that I still did not understand. Gen AI was able to walk me through the code step by step so I know how I can design my game. Furthermore, letting generative AI know the entire code helped prevent confusing errors that could take up a lot of time to fix if it wasn’t aware of the whole context of the code.

Process Evidence

Name: Jade Sun

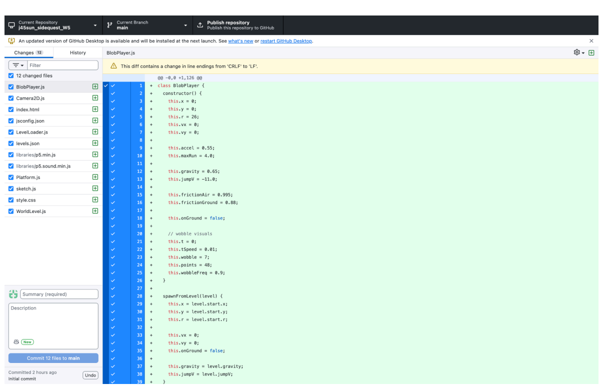
Role: Coding with Generative AI

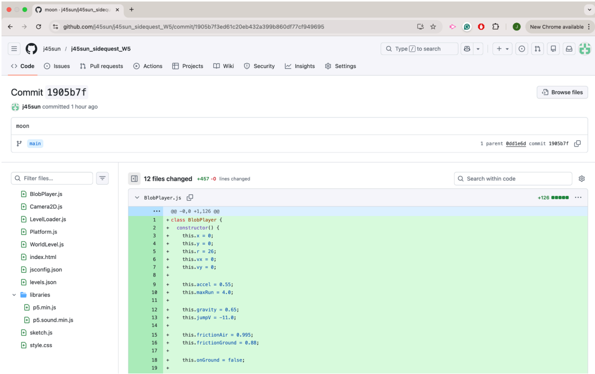
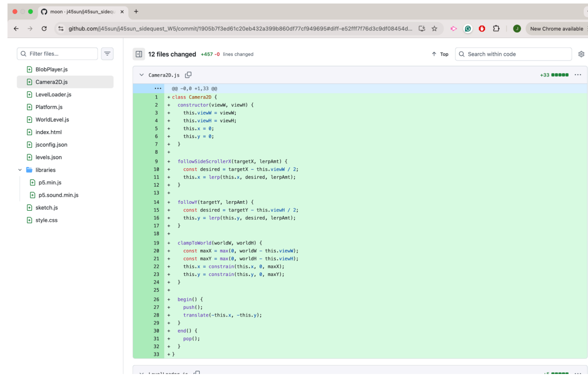
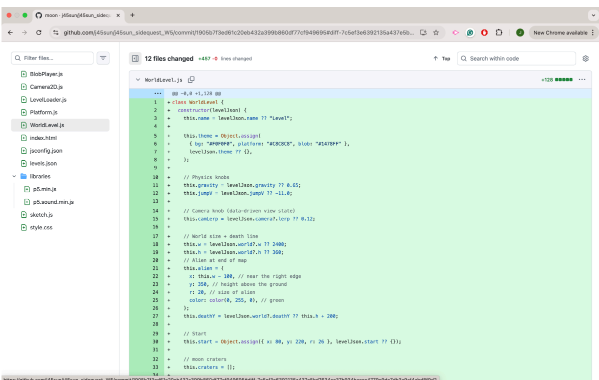
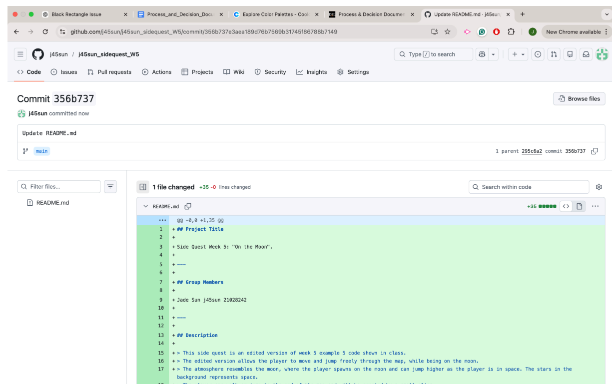
Primary Responsibility: Using generative AI to learn and write code for complex functions in Visual Studio Code

Goal of Work Session: The goal for my work session was to use generative AI to learn how I can make the atmosphere feel like being on the moon through visuals. Generative AI was able to provide me with steps on how I could make my designs possible. I also used lecture notes as reference to ensure that what I was creating aligned with class content.

Tools used: ChatGPT Plus, ChatGPT

Process Evidence:



GenAI Documentation

Date used: January Feb 23, 2026

Tool Disclosure: ChatGPT 5.2, ChatGPT 5.2 Plus

Purpose of Use: To learn the functions of camera systems learned in class, and to learn how I can use those functions in my side quest.

Summary of Interaction: ChatGPT was able to help me with learning and understanding the original code from example 5. Furthermore, I used ChatGPT to learn how I can incorporate my visual designs into the code, such as adding twinkling stars in the background. When there were errors in my code that messed up my sketch, ChatGPT was able to find the error and walk me though how to fix it.

Human Decision Points: All design choices, such as choosing colours and editing variables for visuals, were all decided without using generative AI. Furthermore, ChatGPT provided me with many different options on how to code my design choices. I ended up trying every option, and I decided which response was the best to use in my side quest based on how well it works with the rest of my code. I rejected responses that would give me errors, and when I had to choose a response that created another issue that I had to troubleshoot, I iterated my prompts with ChatGPT to ask how I could fix the errors.

Integrity & Verification Note: Before starting the side quest, I carefully went through the slides and the instructions to ensure that I would understand when ChatGPT gave responses that were too complex or did not align with course concepts. While asking ChatGPT questions, it would sometimes provide me with codes that did not end up working when I incorporated it into my side quest. One example was when I asked ChatGPT to help me add a gradient background into the side quest, the response ended up creating other issues. In order to fix these errors, I iterated my prompts so that ChatGPT could understand the context of my design or current code better and provide me with more accurate responses.

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). The brainstorming process of creating the designs of this side quest were all thought of without using generative AI.

Limitations of Misfires: While iterating my prompts, ChatGPT suddenly was unable to answer them since the free trial for ChatGPT Plus stopped working. I had to type in my prompt in a new chat, while also adding the contexts of the information from the old chat. After this incident, ChatGPT's responses were less detailed and more confusing. It was difficult to understand where to put certain codes, and many more iterations were made while using ChatGPT plus. Furthermore, ChatGPT failed to see that some responses would give errors or "break the game", which lead to many troubleshooting moments.

Appendix

Prompt 1 (ChatGPT Plus)

please explain in detail how this code works with each other to create a game: bloblplayer.js: class BlobPlayer { constructor() { this.x = 0; this.y = 0; this.r = 26; this.vx = 0; this.vy = 0; this.accel = 0.55; this.maxRun = 4.0; this.gravity = 0.65; this.jumpV = -11.0; this.frictionAir = 0.995; this.frictionGround = 0.88; this.onGround = false; // wobble visuals this.t = 0; this.tSpeed = 0.01; this.wobble = 7; this.points = 48; this.wobbleFreq = 0.9; } spawnFromLevel(level) { this.x = level.start.x; this.y = level.start.y; this.r = level.start.r; this.vx = 0; this.vy = 0; this.onGround = false; this.gravity = level.gravity; this.jumpV = level.jumpV; } tryJump() { if (this.onGround) { this.vy = this.jumpV; this.onGround = false; } } update(level) { // input let move = 0; if (keyIsDown(65) || keyIsDown(LEFT_ARROW)) move -= 1; if (keyIsDown(68) || keyIsDown(RIGHT_ARROW)) move += 1; this.vx += this.accel * move; this.vx *= this.onGround ? this.frictionGround : this.frictionAir; this.vx = constrain(this.vx, -this.maxRun, this.maxRun); this.vy += this.gravity; // collider box let box = { x: this.x - this.r, y: this.y - this.r, w: this.r * 2, h: this.r * 2, }; // move X box.x += this.vx; for (const s of level.platforms) { if (BlobPlayer.overlap(box, s)) { if (this.vx > 0) box.x = s.x - box.w; else if (this.vx < 0) box.x = s.x + s.w; this.vx = 0; } } // move Y box.y += this.vy; this.onGround = false; for (const s of level.platforms) { if (BlobPlayer.overlap(box, s)) { if (this.vy > 0) { box.y = s.y - box.h; this.vy = 0; this.onGround = true; } else if (this.vy < 0) { box.y = s.y + s.h; this.vy = 0; } } } // write back this.x = box.x + box.w / 2; this.y = box.y + box.h / 2; // keep inside world horizontally, allow falling below world this.x = constrain(this.x, this.r, level.w - this.r); this.t += this.tSpeed; } draw(colHex) { fill(color(colHex)); noStroke(); beginShape(); for (let i = 0; i < this.points; i++) { const a = (i / this.points) * TAU; const n = noise(cos(a) * this.wobbleFreq + 100, sin(a) * this.wobbleFreq + 100, this.t,); const rr = this.r + map(n, 0, 1, -this.wobble, this.wobble); vertex(this.x + cos(a) * rr, this.y + sin(a) * rr); } endShape(CLOSE); } static overlap(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); } } levels.json: { "schemaVersion": 1, "levels": [{ "name": "Intro Steps", "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" }, "gravity": 0.65, "jumpV": -11.0, "camera": { "lerp": 0.12 }, "world": { "w": 2400, "h": 360, "deathY": 560 }, "start": { "x": 80, "y": 220, "r": 26 }, "platforms": [{ "x": 0, "y": 424, "w": 640, "h": 36 }, { "x": 120, "y": 354, "w": 120, "h": 12 }, { "x": 300, "y": 304, "w": 90, "h": 12 }, { "x": 440, "y": 244, "w": 130, "h": 12 }, { "x": 520, "y": 354, "w": 90, "h": 12 }, { "x": 570, "y": 164, "w": 30, "h": 12 }, { "x": 720, "y": 424, "w": 420, "h": 36 }, { "x": 760, "y": 364, "w": 110, "h": 12 }, { "x": 930, "y": 324, "w": 110, "h": 12 }, { "x": 1090, "y": 284, "w": 110, "h": 12 }, { "x": 1200, "y": 424, "w": 520, "h": 36 }, { "x": 1400, "y": 344, "w": 140, "h": 12 }] }] } platforms.json: class Platform { constructor(x, y, w, h) { this.x = x; this.y = y; this.w = w; this.h = h; } } sketch.js: const VIEW_W = 800; const VIEW_H = 480; let allLevelsData; let levelIndex = 0; let level; let player; let cam; function preload() { allLevelsData = loadJSON("levels.json"); // levels.json beside index.html [web:122] } function setup() { createCanvas(VIEW_W, VIEW_H); textFont("sans-serif"); textSize(14); cam = new Camera2D(width, height); loadLevel(levelIndex); } function loadLevel(i) { level = LevelLoader.fromLevelsJson(allLevelsData, i); player = new BlobPlayer(); player.spawnFromLevel(level); cam.x = player.x - width / 2; cam.y = 0; cam.clampToWorld(level.w, level.h); } function draw() { // --- game state --- player.update(level); // Fall death → respawn if (player.y - player.r > level.deathY) { loadLevel(levelIndex); return; } // --- view state (data-driven smoothing) --- cam.followSideScrollerX(player.x, level.camLerp); cam.y = 0; cam.clampToWorld(level.w, level.h); // --- draw --- cam.begin(); level.drawWorld(); player.draw(level.theme.blob); cam.end(); // HUD fill(0); noStroke(); text(level.name + " (Example 5)", 10, 18); text("A/D or ←/→ move • Space/W/↑ jump • Fall = respawn", 10, 36); text("camLerp(JSON): " + level.camLerp + " world.w: " + level.w, 10, 54); text("cam: " + cam.x + ", " + cam.y, 10, 90); const p0 = level.platforms[0]; text(`p0: x=${p0.x} y=${p0.y} w=${p0.w} h=${p0.h}`, 10, 108); text("platforms: " + level.platforms.length + " start: " + level.start.x + "," + level.start.y, 10, 72,); } function keyPressed() { if (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) { player.tryJump(); } if (key === "r" || key === "R") loadLevel(levelIndex); } style.css: html, body { margin: 0; padding: 0; } canvas { display: block; } wordllevel.js: class WorldLevel { constructor(levelJson) { this.name = levelJson.name ?? "Level"; this.theme = Object.assign({ bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" }, levelJson.theme ?? {},); // Physics knobs this.gravity = levelJson.gravity ?? 0.65; this.jumpV = levelJson.jumpV ?? -11.0; // Camera knob (data-driven view state) this.camLerp = levelJson.camera?.lerp ?? 0.12; // World size + death line this.w = levelJson.world?.w ?? 2400; this.h = levelJson.world?.h ?? 360; this.deathY = levelJson.world?.deathY ?? this.h + 200; // Start this.start = Object.assign({ x: 80, y: 220, r: 26 }, levelJson.start ?? {}); // Platforms this.platforms = (levelJson.platforms ?? []).map((p) => new Platform(p.x, p.y, p.w, p.h),); } drawWorld() { background(this.theme.bg); push(); rectMode(CORNER); // critical: undo any global rectMode(CENTER) [web:230] noStroke(); fill(this.theme.platform); for (const p of this.platforms) rect(p.x, p.y, p.w, p.h); // x,y = top-left [web:234] pop(); } } explain how these } camera2D.js: class Camera2D { constructor(viewW, viewH) { this.viewW = viewW; this.viewH = viewH; this.x = 0; this.y = 0; } followSideScrollerX(targetX, lerpAmt) { const desired = targetX - this.viewW / 2; this.x = lerp(this.x, desired, lerpAmt); } clampToWorld(worldW, worldH) { const maxX = max(0, worldW - this.viewW); const maxY = max(0, worldH - this.viewH); this.x = constrain(this.x, 0, maxX); this.y = constrain(this.y, 0, maxY); } begin() { push(); translate(-this.x, -this.y); } end() { pop(); } } index.html: <!doctype html> <html lang="en"> <head> <meta charset="utf-8" /> <meta name="viewport" content="width=device-width, initial-scale=1" /> <title>Week 5 — Example 5 (Side-scroller + Camera + JSON)</title> <link rel="stylesheet" href="style.css" /> <script src="libraries/p5.min.js"></script> <!-- classes --> <script src="Platform.js"></script> <script src="WorldLevel.js"></script> <script src="BlobPlayer.js"></script> <script src="Camera2D.js"></script> <script src="LevelLoader.js"></script> <!-- sketch last --> <script src="sketch.js"></script> </head> <body> <main></main> </body> </html> jsonconfig.json: { "include": ["*.js", "libraries/*.js", "c:\\Users\\david\\.vscode\\extensions\\wmcicompsci.cs30-p5-1.9.3\\p5types\\global.d.ts"] } levelloader.js: class LevelLoader { static fromLevelsJson(allData, index) { return new WorldLevel(allData.levels[index]); } }

Response 1

Good — this is the point where everything "clicks."
I'm going to explain this like we're zooming out and watching how every file cooperates to create one running game.

This is a **modular side-scroller architecture**. Each file has one responsibility.

🧠 Big Picture: How the Whole Game Runs

When the page loads:

1. **index.html loads p5**
2. It loads all class files

3. It loads sketch.js last
4. p5 automatically calls:
   - preload()
   - setup()
   - then repeatedly calls draw() 60 times per second

Every frame, this happens:

update game state → update camera → draw world → draw player → draw HUD

Now let's break each piece down.

📁 index.html — The Entry Point

This file loads everything in order:

```
<script src="Platform.js"></script>
<script src="WorldLevel.js"></script>
<script src="BlobPlayer.js"></script>
<script src="Camera2D.js"></script>
<script src="LevelLoader.js"></script>
<script src="sketch.js"></script>
```

Why this order matters:

- Classes must exist before sketch.js uses them.
- sketch.js is the "conductor."

🎮 sketch.js — The Conductor

This is the control center.

```
preload()
allLevelsData = loadJSON("levels.json");
```

Loads your level data BEFORE setup runs.

---

setup()
createCanvas(VIEW_W, VIEW_H);
cam = new Camera2D(width, height);
loadLevel(levelIndex);

- Creates the 800x480 view window.
- Creates the camera.
- Loads the first level.

---

loadLevel()

This is critical.

level = LevelLoader.fromLevelsJson(allLevelsData, i);
player = new BlobPlayer();
player.spawnFromLevel(level);

This does:

1. Convert JSON → WorldLevel object
2. Create a player
3. Spawn player at level start position
4. Set camera initial position

So:
JSON → Level object → Player + Camera use it.

📄 levels.json — The Data

This file contains:

- World width & height
- Gravity
- Jump strength
- Camera lerp amount
- Platform positions
- Theme colors

This is **data-driven design**.

Instead of hardcoding:

this.gravity = 0.65;

It comes from JSON:

"gravity": 0.65

That means:
You can change physics without touching JS.

🌍 WorldLevel.js — The World Object

This class turns raw JSON into a usable game level.

Constructor does:

- Store name
- Store theme colors
- Store gravity & jump settings
- Store camera lerp
- Store world size

- Create platform objects

This line is important:

```
this.platforms = levelJson.platforms.map(
 (p) => new Platform(p.x, p.y, p.w, p.h)
);
```

It converts plain JSON objects into real Platform class instances.

```
drawWorld()
background(this.theme.bg);
for (const p of this.platforms)
 rect(p.x, p.y, p.w, p.h);
```

Draws:

- Background
- Every platform rectangle

This is world rendering.

🟦 Platform.js

Very simple:

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

Just stores rectangle data.

It doesn't draw itself.
 WorldLevel draws it.

🔵 BlobPlayer.js — Physics + Input + Collision

This is the most complex piece.

It handles:

- Movement
- Gravity
- Jumping
- Friction
- Collision detection
- Visual wobble effect

Step 1: spawnFromLevel()

When level loads:

```
this.x = level.start.x;
this.gravity = level.gravity;
this.jumpV = level.jumpV;
```

Player inherits physics from level.

Step 2: update(level)

This runs every frame.

A) Input
```
if (keyIsDown(LEFT_ARROW)) move -= 1;
if (keyIsDown(RIGHT_ARROW)) move += 1;
```

Updates velocity:

```
this.vx += this.accel * move;
```

Adds friction:

this.vx *= this.onGround ? this.frictionGround : this.frictionAir;

B) Gravity
this.vy += this.gravity;

Every frame gravity pulls down.

C) Collision Detection

This is very important.

Instead of moving directly:

1. Create a box collider
2. Move X
3. Check collision
4. Fix overlap
5. Move Y
6. Check collision
7. Fix overlap

Why separate X and Y?

Because it prevents getting stuck in corners.

```
overlap()
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 AABB (Axis-Aligned Bounding Box) collision.

Rectangles overlapping = collision.

D) Final Position Write Back

After collision corrections:

this.x = box.x + box.w / 2;
this.y = box.y + box.h / 2;

Sync visual circle position to collider.

E) Visual Wobble

In draw():

Uses Perlin noise:

noise(...)

Each frame:

- Slightly alters radius
- Creates squishy blob effect

This is purely visual.

Physics uses invisible box.
Drawing uses organic blob.

🎥 Camera2D.js — The View System

This controls what part of the world we see.

---

followSideScrollerX()

const desired = targetX - this.viewW / 2;
this.x = lerp(this.x, desired, lerpAmt);

This:

1. Centers camera on player
2. Uses lerp() for smooth motion

If lerp = 0.12:
Camera moves 12% toward target per frame.

Lower = floaty
Higher = tight

---

clampToWorld()

this.x = constrain(this.x, 0, worldW - this.viewW);

Prevents camera from:

- Showing empty space outside world.

---

begin() / end()

push();
translate(-this.x, -this.y);
...
pop();

This shifts the entire world drawing.

It converts:

World Space → Screen Space

This is the bridge.

🔁 What Happens Each Frame (draw loop)

Inside sketch.js:

1️⃣ Update Player

player.update(level);

Handles:

- Input
- Physics
- Collision

2️⃣ Death Check

If player falls below:

if (player.y - player.r > level.deathY)

Reload level.

3️⃣ Update Camera
cam.followSideScrollerX(player.x, level.camLerp);
cam.clampToWorld(level.w, level.h);

Camera smoothly follows player.
Camera stays inside world.

4️⃣ Draw World
cam.begin();
level.drawWorld();
player.draw(...);
cam.end();

Important:

- begin() shifts drawing
- draw world at world coordinates
- draw player at world coordinates

- end() resets translation

5️⃣ Draw HUD

HUD is drawn AFTER camera ends.
So it stays fixed on screen.

🧩 How Everything Connects

Here's the relationship map:

```
levels.json
    ↓
LevelLoader
    ↓
WorldLevel
    ↓
BlobPlayer  ← uses gravity, jumpV, platforms
    ↓
Camera2D  ← follows player.x
    ↓
sketch.js draw loop
```

Each class has a single job:

Class	Responsibility
Platform	Store rectangle data
WorldLevel	Store world + draw platforms
BlobPlayer	Movement + physics
Camera2D	View transformation
LevelLoader	JSON → WorldLevel
sketch.js	Game loop controller

This is clean architecture.

🎯 Why This Is Good Design

- Data separated from logic
- Physics separated from rendering
- Camera separated from player
- Modular and scalable

You could:

- Add enemies
- Add multiple levels
- Change camera style
- Modify physics

Without breaking everything.

🔥 The Core Concept

The illusion of a scrolling world is created by:

translate(-cam.x, -cam.y)

The world moves.
The camera doesn't.

That single idea is the heart of the system.

If you'd like, I can next:

- Draw a visual diagram of how coordinate spaces shift
- Or explain collision step-by-step visually
- Or explain exactly how to modify this into your meditative camera assignment

Prompt 2

this is the updated levels code: { "schemaVersion": 1, "levels": [{ "name": "Intro Steps", "theme": { "bg": "#0d1b2a", "platform": "#e0e1dd", "blob": "#1478FF" }, "gravity": 0.65, "jumpV": -11.0, "camera": { "lerp": 0.12 }, "world": { "w": 2400, "h": 360, "deathY": 560 }, "start": { "x": 80, "y": 220, "r": 26 }, "platforms": [{ "x": 0, "y": 440, "w": 2400, "h": 50 }] }] } how can i make the bottom platform resemble the moon? (add shapes on the bottom platform that resembles crevices)

Response 2:

Good — this is now a design question, not a physics one.

Right now your moon "ground" is just one flat rectangle:

{ "x": 0, "y": 440, "w": 2400, "h": 50 }

Since platforms are drawn in `WorldLevel.drawWorld()` using `rect()`, if you want it to look like the moon, you need to edit:

WorldLevel.js

You are not changing collision.
You are only changing how it is drawn.

Keep the rectangle for physics.
Layer visuals on top of it.

Step 1 — Keep the Solid Floor

Do NOT change the platform data.
Collision should stay simple.

Instead, modify how it is rendered.

Step 2 — Add "Moon Surface" Drawing in WorldLevel.js

Open:

WorldLevel.js

Find:

```
drawWorld() {
 background(this.theme.bg);
 push();
 rectMode(CORNER);
 noStroke();
 fill(this.theme.platform);

 for (const p of this.platforms)
   rect(p.x, p.y, p.w, p.h);

 pop();
}
```

We're going to enhance this.

Option 1: Add Craters (Best and Cleanest)

Inside the `for (const p of this.platforms)` loop, add ellipses.

Example:

```
for (const p of this.platforms) {
 rect(p.x, p.y, p.w, p.h);

 // draw craters
 fill(180, 185, 190);  // slightly darker than surface
 noStroke();

 for (let i = 0; i < 20; i++) {
   const cx = p.x + random(p.w);
   const cy = p.y + random(p.h / 2);
   const r = random(20, 60);
   ellipse(cx, cy, r);
 }
}
```

This creates circular craters.

However — random inside draw() will change every frame.

So better version:

Option 2 (Better): Pre-Generate Craters Once

In the constructor of `WorldLevel`, add:

```
this.craters = [];
```

Then after platforms are created:

```
for (let i = 0; i < 40; i++) {
 this.craters.push({
   x: random(this.w),
   y: random(440, 480),
   r: random(20, 60)
 });
}
```

Now in `drawWorld()`:

```
for (const p of this.platforms)
 rect(p.x, p.y, p.w, p.h);

// draw craters
fill(160, 165, 170);
noStroke();
for (const c of this.craters) {
 ellipse(c.x, c.y, c.r);
}
```

Now the moon texture is stable.

Option 3 — Jagged Surface (Moon Crevices)

Instead of flat top edge, draw a shape.

Replace the rectangle drawing with:

```
beginShape();
```

```
vertex(0, 440);

for (let x = 0; x <= this.w; x += 40) {
 let yOffset = noise(x * 0.01) * 20;
 vertex(x, 440 + yOffset);
}

vertex(this.w, 490);
vertex(0, 490);
endShape(CLOSE);
```

This creates a bumpy lunar surface using Perlin noise.

But — this is visual only.
Collision still uses the rectangle.

Option 4 — Combine Both (Best Visual Result)

Keep:

- Flat invisible rectangle for physics
- Draw noisy edge
- Draw darker crater circles

This creates depth.

Important Concept

You are separating:

Physics shape (simple rectangle)
from
Visual shape (moon texture)

That is professional design.

Games rarely use their physics shape for visuals.

If You Want It to Feel More "Moon"

You can also:

- Darken background to deep space (#0d1b2a is good)
- Add small white star dots in drawWorld()
- Lower gravity in JSON (moon gravity ≈ 0.3–0.4)
- Reduce jump velocity slightly

Example:

"gravity": 0.35

Now movement will feel lunar.

If you tell me which vibe you want:

- Realistic NASA moon
- Dreamy soft moon
- Cartoon crater moon
- Dark lonely space

I can give you the exact visual code setup for that style.

Prompt 3:

how can i add twinkling stars (represented in small circles) in the background

Response 3:

Good — this is exactly the right kind of visual layer to add.

You want **stars that:**

- stay in the background
- don't move randomly every frame
- softly twinkle

This means we must:

1. Generate stars once
2. Store them
3. Animate brightness slightly

You will edit:

WorldLevel.js

Not the player. Not the camera. Not the JSON.

Step 1 — Store Stars in the Level

Open `WorldLevel.js`.

Inside the constructor, after platforms are created, add:

```
// generate stars once
this.stars = [];

for (let i = 0; i < 120; i++) {
 this.stars.push({
  x: random(this.w),
  y: random(0, 400),     // only in sky area
  size: random(1, 3),
  phase: random(TAU)     // for twinkle timing
 });
}
```

What this does:

- Random x across entire world width
- Random y only in upper sky
- Small sizes
- Each star has its own animation phase

Now stars exist permanently.

Step 2 — Animate Them in drawWorld()

Go to `drawWorld()`.

You'll see:

background(this.theme.bg);

Immediately AFTER that, add:

```
// draw stars
noStroke();

for (const s of this.stars) {
 // twinkle effect
 const glow = 180 + 75 * sin(frameCount * 0.05 + s.phase);

 fill(glow);
 ellipse(s.x, s.y, s.size);
}
```

Why This Works

`sin(frameCount * 0.05 + s.phase)`

- `frameCount` keeps increasing

- `sin()` oscillates between -1 and 1
- Adding `s.phase` means stars don't twinkle at same time
- Glow changes brightness smoothly

Result:
Each star fades in and out independently.

Important Detail: Why This Moves With Camera

Because stars are drawn inside:

cam.begin();
level.drawWorld();
...
cam.end();

They exist in world space.

So as camera scrolls, stars scroll naturally too.

If you wanted fixed background stars (not scrolling), you'd draw them outside camera begin/end — but that's not what we want here.

If You Want Softer Twinkle

Change speed:

frameCount * 0.02

Slower = more meditative.

If You Want More Subtle Glow

Change:

```
180 + 75 * sin(...)
```

To something like:

```
200 + 30 * sin(...)
```

Less contrast.

If You Want a Slight Glow Effect

Draw twice:

```
fill(glow, glow, glow, 100);
ellipse(s.x, s.y, s.size * 2);

fill(255);
ellipse(s.x, s.y, s.size);
```

This creates a faint halo.

What You Just Added Architecturally

You kept:

- Physics separate
- Camera separate
- Player separate

And added:

- A pure visual background system inside the world renderer

That is exactly how professional layering works.

If you want, I can next help you:

- Make stars parallax slower than ground
- Add shooting stars
- Add subtle nebula glow
- Make stars react to camera movement for a dreamy effect

Prompt 4:

how can i make the gravity smaller (like on the moon), and when the player jumps, the camera moves with the player (the ground should disappear when player jumps)? how can i ensure that the stars still appear when the player jumps?

Response 4:

Good — now you're thinking about **physics + camera behavior together**, which is exactly how games feel cohesive.

We'll handle this in three clean steps:

1. Reduce gravity (moon feel)
2. Make the camera follow vertically
3. Keep stars visible while jumping

All of this is controlled by **data + camera**, not collision.

1️⃣Make Gravity Smaller (Moon Physics)

You already designed this system to be data-driven.

Open:

levels.json

Change:

"gravity": 0.65,
"jumpV": -11.0,

To something like:

"gravity": 0.25,
"jumpV": -8.0,

Why also reduce jumpV?

Because if you lower gravity but keep -11 jump velocity, the player will launch into orbit.

Start with:

- gravity: `0.25`
- jumpV: `-8` or `-7`

Adjust until it *feels* lunar.

That's it for physics. No JS editing needed.

2️⃣ Make the Camera Follow Vertically

Right now your camera only follows X.

In `sketch.js` you have:

cam.followSideScrollerX(player.x, level.camLerp);
cam.y = 0;
cam.clampToWorld(level.w, level.h);

That line:

cam.y = 0;

Is forcing the camera to stay locked vertically.

Delete that line.

Now we need to actually follow Y.

Add Vertical Follow in Camera2D.js

Open:

Camera2D.js

Add this method:

```
followY(targetY, lerpAmt) {
 const desired = targetY - this.viewH / 2;
 this.y = lerp(this.y, desired, lerpAmt);
}
```

Now Update sketch.js

Replace your camera update section with:

```
cam.followSideScrollerX(player.x, level.camLerp);
cam.followY(player.y, level.camLerp);
cam.clampToWorld(level.w, level.h);
```

Now:

- When player jumps
- Camera follows upward
- Ground scrolls off screen

Exactly what you want.

3️⃣ Ensure Stars Still Appear When Player Jumps

Stars already scroll because they are drawn inside:

cam.begin();
level.drawWorld();
player.draw(...);
cam.end();

However...

If your stars are only generated between:

y: random(0, 400)

But your world height is:

"h": 360

Then when the camera moves upward, you might scroll into empty space with no stars.

Fix: Fill Entire World Height With Stars

In `WorldLevel.js`, update your star generation:

Instead of:

y: random(0, 400),

Use:

y: random(0, this.h),

Now stars exist across the entire world vertically.

So when the camera moves up:
Stars are still there.

Optional: Make It Feel More Lunar

If you want dramatic float:

Lower lerp in JSON:

"camera": { "lerp": 0.06 }

Lower lerp = floaty drifting camera.

Now:

- Low gravity
- Slow vertical camera glide
- Ground disappears on jump
- Stars remain visible

That creates a calm, moon-like floating experience.

Summary of Exactly What You Edit

Edit 1: `levels.json`

- Lower gravity
- Adjust jumpV

Edit 2: `Camera2D.js`

- Add followY()

Edit 3: `sketch.js`

- Remove `cam.y = 0`
- Call `cam.followY()`

Edit 4: `WorldLevel.js`

- Generate stars across `this.h`

That's the clean architectural way.

If you want, I can help you make the camera feel even more cinematic — like slow upward drift before settling back down.

Prompt 5:

how can i make the background gradient

Response 5:

Good — gradients are a **rendering layer**, so you'll edit:
WorldLevel.js

Specifically inside:

drawWorld()

Right now you probably have:

background(this.theme.bg);

That fills a solid color.

We're going to replace that with a vertical gradient.

Option 1: Smooth Vertical Gradient (Clean + Professional)

Inside `drawWorld()`, replace:

background(this.theme.bg);

With:

```
// gradient background
for (let y = 0; y < this.h; y++) {
 let inter = map(y, 0, this.h, 0, 1);
 let c = lerpColor(
   color("#0d1b2a"),   // top color (deep space)
   color("#1b263b"),   // bottom color (horizon glow)
   inter
 );
 stroke(c);
```

```
 line(0, y, this.w, y);
}
```

What this does:

- Loops from top to bottom of world
- Gradually blends two colors
- Draws horizontal lines across full world width

Now your sky fades smoothly.

Important: Why Use `this.w` and `this.h`

Because you are drawing in **world space**, not screen space.

The camera will scroll across this gradient naturally.

Option 2: Horizontal Space Glow

If you want subtle space glow near horizon (like moonlight):

Change the interpolation to only affect upper area:

```
for (let y = 0; y < this.h; y++) {
 let inter = map(y, 0, this.h * 0.7, 0, 1);
 inter = constrain(inter, 0, 1);

 let c = lerpColor(
  color("#020617"),   // almost black
  color("#0d1b2a"),   // dark blue
  inter
 );

 stroke(c);
 line(0, y, this.w, y);
}
```

Now the top is darker, bottom slightly lighter.

More cinematic.

Option 3: Radial Moon Glow (More Atmospheric)

If you want it to feel dreamy:

After drawing gradient, add:

```
noStroke();
for (let r = 400; r > 0; r -= 4) {
 fill(255, 255, 255, 3);
 ellipse(this.w / 2, 200, r);
}
```

That creates faint glowing halo in sky.

Order Matters

Your `drawWorld()` should now look like:

1. Draw gradient
2. Draw stars
3. Draw platforms
4. Draw craters (if you added them)

Background → mid layer → foreground.

If You Want It To Feel More “Moon”

Try this color combo:

Top: `#020617`
Mid: `#0d1b2a`
Bottom: `#1b263b`

Dark navy space gradient feels lunar.

If you'd like, I can give you a polished full `drawWorld()` version combining:

- Gradient
- Twinkling stars
- Moon ground with craters
- Subtle glow

All layered correctly.

Prompt 6 (ChatGPT)

this is the code: blobplayer.js: class BlobPlayer { constructor() { this.x = 0; this.y = 0; this.r = 26; this.vx = 0; this.vy = 0; this.accel = 0.55; this.maxRun = 4.0; this.gravity = 0.65; this.jumpV = -11.0; this.frictionAir = 0.995; this.frictionGround = 0.88; this.onGround = false; // wobble visuals this.t = 0; this.tSpeed = 0.01; this.wobble = 7; this.points = 48; this.wobbleFreq = 0.9; } spawnFromLevel(level) { this.x = level.start.x; this.y = level.start.y; this.r = level.start.r; this.vx = 0; this.vy = 0; this.onGround = false; this.gravity = level.gravity; this.jumpV = level.jumpV; } tryJump() { if (this.onGround) { this.vy = this.jumpV; this.onGround = false; } } update(level) { // input let move = 0; if (keyIsDown(65) || keyIsDown(LEFT_ARROW)) move -= 1; if (keyIsDown(68) || keyIsDown(RIGHT_ARROW)) move += 1; this.vx += this.accel * move; this.vx *= this.onGround ? this.frictionGround : this.frictionAir; this.vx = constrain(this.vx, -this.maxRun, this.maxRun); this.vy += this.gravity; // collider box let box = { x: this.x - this.r, y: this.y - this.r, w: this.r * 2, h: this.r * 2, }; // move X box.x += this.vx; for (const s of level.platforms) { if (BlobPlayer.overlap(box, s)) { if (this.vx > 0) box.x = s.x - box.w; else if (this.vx < 0) box.x = s.x + s.w; this.vx = 0; } } // move Y box.y += this.vy; this.onGround = false; for (const s of level.platforms) { if (BlobPlayer.overlap(box, s)) { if (this.vy > 0) { box.y = s.y - box.h; this.vy = 0; this.onGround = true; } else if (this.vy < 0) { box.y = s.y + s.h; this.vy = 0; } } } // write back this.x = box.x + box.w / 2; this.y = box.y + box.h / 2; // keep inside world horizontally, allow falling below world this.x = constrain(this.x, this.r, level.w - this.r); this.t += this.tSpeed; } draw(colHex) { fill(color(colHex)); noStroke(); beginShape(); for (let i = 0; i < this.points; i++) { const a

= (i / this.points) * TAU; const n = noise(cos(a) * this.wobbleFreq + 100, sin(a) * this.wobbleFreq + 100, this.t,); const rr = this.r + map(n, 0, 1, -this.wobble, this.wobble); vertex(this.x + cos(a) * rr, this.y + sin(a) * rr); } endShape(CLOSE); } static overlap(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); } } camera2D.js: class Camera2D { constructor(viewW, viewH) { this.viewW = viewW; this.viewH = viewH; this.x = 0; this.y = 0; } followSideScrollerX(targetX, lerpAmt) { const desired = targetX - this.viewW / 2; this.x = lerp(this.x, desired, lerpAmt); } followY(targetY, lerpAmt) { const desired = targetY - this.viewH / 2; this.y = lerp(this.y, desired, lerpAmt); } clampToWorld(worldW, worldH) { const maxX = max(0, worldW - this.viewW); const maxY = max(0, worldH - this.viewH); this.x = constrain(this.x, 0, maxX); this.y = constrain(this.y, 0, maxY); } begin() { push(); translate(-this.x, -this.y); } end() { pop(); } } index.html: <!doctype html> <html lang="en"> <head> <meta charset="utf-8" /> <meta name="viewport" content="width=device-width, initial-scale=1" /> <title>Week 5 — Example 5 (Side-scroller + Camera + JSON)</title> <link rel="stylesheet" href="style.css" /> <script src="libraries/p5.min.js"></script> <!-- classes --> <script src="Platform.js"></script> <script src="WorldLevel.js"></script> <script src="BlobPlayer.js"></script> <script src="Camera2D.js"></script> <script src="LevelLoader.js"></script> <!-- sketch last --> <script src="sketch.js"></script> </head> <body> <main></main> </body> </html> levelloader.js: class LevelLoader { static fromLevelsJson(allData, index) { return new WorldLevel(allData.levels[index]); } } levels.json: { "schemaVersion": 1, "levels": [{ "name": "Intro Steps", "theme": { "bg": "#0d1b2a", "platform": "#e0e1dd", "blob": "#1478FF" }, "gravity": 0.25, "jumpV": -11.0, "camera": { "lerp": 0.12 }, "world": { "w": 2400, "h": 360, "deathY": 560 }, "start": { "x": 80, "y": 220, "r": 26 }, "platforms": [{ "x": 0, "y": 440, "w": 2400, "h": 65 }] }] } platforms.js: class Platform { constructor(x, y, w, h) { this.x = x; this.y = y; this.w = w; this.h = h; } } sketch.js: /* Week 5 — Example 5: Side-Scroller Platformer with JSON Levels + Modular Camera Course: GBDA302 | Instructors: Dr. Karen Cochrane & David Han Date: Feb. 12, 2026 Move: WASD/Arrows | Jump: Space Learning goals: - Build a side-scrolling platformer using modular game systems - Load complete level definitions from external JSON (LevelLoader + levels.json) - Separate responsibilities across classes (Player, Platform, Camera, World) - Implement gravity, jumping, and collision with platforms - Use a dedicated Camera2D class for smooth horizontal tracking - Support multiple levels and easy tuning through data files - Explore scalable project architecture for larger games */ const VIEW_W = 800; const VIEW_H = 480; let allLevelsData; let levelIndex = 0; let level; let player; let cam; function preload() { allLevelsData = loadJSON("levels.json"); // levels.json beside index.html [web:122] } function setup() { createCanvas(VIEW_W, VIEW_H); textFont("sans-serif"); textSize(14); cam = new Camera2D(width, height); loadLevel(levelIndex); } function loadLevel(i) { level = LevelLoader.fromLevelsJson(allLevelsData, i); player = new BlobPlayer(); player.spawnFromLevel(level); cam.x = player.x - width / 2; cam.y = 0; cam.clampToWorld(level.w, level.h); } function draw() { // --- game state --- player.update(level); // Fall death → respawn if (player.y - player.r > level.deathY) { loadLevel(levelIndex); return; } // --- view state (data-driven smoothing) --- cam.followSideScrollerX(player.x, level.camLerp); cam.clampToWorld(level.w, level.h);

cam.followY(player.y, level.camLerp); // --- draw --- cam.begin(); level.drawWorld(); player.draw(level.theme.blob); cam.end(); // HUD fill(0); noStroke(); text(level.name + " (Example 5)", 10, 18); text("A/D or ←/→ move • Space/W/↑ jump • Fall = respawn", 10, 36); text("camLerp(JSON): " + level.camLerp + " world.w: " + level.w, 10, 54); text("cam: " + cam.x + ", " + cam.y, 10, 90); const p0 = level.platforms[0]; text(`p0: x=${p0.x} y=${p0.y} w=${p0.w} h=${p0.h}`, 10, 108); text("platforms: " + level.platforms.length + " start: " + level.start.x + "," + level.start.y, 10, 72,); } function keyPressed() { if (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) { player.tryJump(); } if (key === "r" || key === "R") loadLevel(levelIndex); } style.css: html, body { margin: 0; padding: 0; } canvas { display: block; } worldlevel.js: class WorldLevel { constructor(levelJson) { this.name = levelJson.name ?? "Level"; this.theme = Object.assign({ bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" }, levelJson.theme ?? {},); // Physics knobs this.gravity = levelJson.gravity ?? 0.65; this.jumpV = levelJson.jumpV ?? -11.0; // Camera knob (data-driven view state) this.camLerp = levelJson.camera?.lerp ?? 0.12; // World size + death line this.w = levelJson.world?.w ?? 2400; this.h = levelJson.world?.h ?? 360; this.deathY = levelJson.world?.deathY ?? this.h + 200; // Start this.start = Object.assign({ x: 80, y: 220, r: 26 }, levelJson.start ?? {}); // moon craters this.craters = []; // Platforms this.platforms = (levelJson.platforms ?? []).map((p) => new Platform(p.x, p.y, p.w, p.h),); for (let i = 0; i < 40; i++) { this.craters.push({ x: random(this.w), y: random(450, 480), r: random(20, 50), }); } //stars this.stars = []; for (let i = 0; i < 120; i++) { this.stars.push({ x: random(this.w), y: random(0, 400), // only in sky area size: random(1, 3), phase: random(TAU), // for twinkle timing }); } } drawWorld() { background(this.theme.bg); // draw stars noStroke(); for (const s of this.stars) { // twinkle effect const glow = 180 + 75 * sin(frameCount * 0.05 + s.phase); fill(glow); ellipse(s.x, s.y, s.size); } push(); rectMode(CORNER); // critical: undo any global rectMode(CENTER) [web:230] noStroke(); fill(this.theme.platform); for (const p of this.platforms) rect(p.x, p.y, p.w, p.h); // x,y = top-left [web:234] // moon craters for (const p of this.platforms) rect(p.x, p.y, p.w, p.h); fill(160, 165, 170); noStroke(); for (const c of this.craters) { ellipse(c.x, c.y, c.r); } pop(); } } how can i make the background gradient without touching level.json

Response 6: You can add a gradient background **directly in the `WorldLevel.drawWorld()` method** without touching `levels.json`. Essentially, instead of `background(this.theme.bg)`, you draw a vertical gradient using `lerpColor` and a loop. Here's how you can modify your `drawWorld()`:

```
drawWorld() {
 // --- Gradient background ---
 noFill();
 for (let y = 0; y < this.h; y++) {
   const inter = map(y, 0, this.h, 0, 1);
   // define start and end colors of your gradient
   const c1 = color('#0d1b2a'); // top color
   const c2 = color('#1b263b'); // bottom color
   stroke(lerpColor(c1, c2, inter));
```

```
  line(0, y, this.w, y);
 }

 // draw stars
 noStroke();
 for (const s of this.stars) {
  const glow = 180 + 75 * sin(frameCount * 0.05 + s.phase);
  fill(glow);
  ellipse(s.x, s.y, s.size);
 }

 push();
 rectMode(CORNER);
 noStroke();
 fill(this.theme.platform);

 for (const p of this.platforms) rect(p.x, p.y, p.w, p.h);

 // moon craters
 fill(160, 165, 170);
 for (const c of this.craters) {
  ellipse(c.x, c.y, c.r);
 }
 pop();
}
```

✅ Notes:

1. `c1` and `c2` are your gradient's top and bottom colors. You can adjust them to whatever looks good.
2. `map(y, 0, this.h, 0, 1)` gives a normalized value from 0–1 for `lerpColor`.
3. This method does **not require changing `levels.json`**, because the colors are hardcoded in the draw function.

If you want, I can also show a **slightly faster version** that avoids drawing 480+ lines every frame but still gives a smooth vertical gradient. This is useful for performance in bigger worlds.

Do you want me to do that?

Prompt 7:

why does the blob and the stars near the bottom of the screen leave a "trail"

Response 7:

Ah! That "trail" effect is happening because your gradient is **not clearing the background properly for the canvas size you're rendering**. Let me break it down carefully:

1️⃣ Why the trail happens

In `drawWorld()` you replaced:

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

with a gradient like:

```
for (let y = 0; y < this.h; y++) {
  stroke(lerpColor(c1, c2, y / this.h));
  line(0, y, this.w, y);
}
```

- Here, you're drawing the gradient for `y` from 0 → `this.h` (the **world height**, e.g., 360).
- But your canvas is **viewport-sized** (`VIEW_H = 480`), so the bottom 120 pixels **don't get overwritten**.
- When the camera moves down and the blob or stars enter that region, the old pixels remain → trail effect.

2️⃣ How to fix it

You need to **draw the gradient across the actual visible canvas height**, not the world height:

```
// inside WorldLevel.drawWorld()
noFill();
for (let y = 0; y < height; y++) { // <-- use viewport height
  const inter = map(y, 0, height, 0, 1);
  const c1 = color('#0d1b2a'); // top
  const c2 = color('#1b263b'); // bottom
```

```
  stroke(lerpColor(c1, c2, inter));
  line(0, y, width, y);       // <-- width of canvas
}
```

- Using `height` and `width` ensures you cover the **entire visible canvas** every frame.
- Now no matter where the camera scrolls, old pixels are cleared, and you won't get trails.

3️⃣ Optional Performance Tip

Instead of drawing 480+ lines every frame, you can **pre-render the gradient to a p5 `graphics` buffer**:

```
let bgGradient;

function setup() {
 createCanvas(VIEW_W, VIEW_H);
 bgGradient = createGraphics(width, height);
 for (let y = 0; y < height; y++) {
  const inter = map(y, 0, height, 0, 1);
  const c1 = color('#0d1b2a');
  const c2 = color('#1b263b');
  bgGradient.stroke(lerpColor(c1, c2, inter));
  bgGradient.line(0, y, width, y);
 }
}

function drawWorld() {
 image(bgGradient, 0, 0);
 // ... rest of drawWorld
}
```

- This **draws the gradient once** instead of every frame → smoother performance.
- The blob and stars won't leave trails because the background fully redraws.

If you want, I can rewrite your `WorldLevel.drawWorld()` **fully gradient + stars + platforms** so it's clean, no trails, and still works with your camera system.

Do you want me to do that?

Prompt 8:
why do the stars and the blob (when i jump) leave a trail when the camera moves

Response 8:
Ah — now I understand exactly what's happening. This isn't just about your gradient; it's about **how p5.js handles the camera and background redraws**. Let's break it carefully.

1️⃣ What's causing the "trail"

In your `draw()` loop:

```
cam.begin();
level.drawWorld();  // draws background, stars, platforms
player.draw(level.theme.blob);
cam.end();
```

- `cam.begin()` does a `push()` and `translate(-cam.x, -cam.y)`.
- Your `drawWorld()` draws the **background gradient using world coordinates** (or your previous version with `this.h`) **relative to the world**, not the canvas.

Problem:

- The camera is moving (`translate(-cam.x, -cam.y)`).
- But your gradient and stars are being drawn **inside the camera translation**.
- The `background()` equivalent (your gradient) is **also being translated**, so it doesn't "cover" the new area the camera scrolls into.

This means when the blob moves up or the camera moves, **old pixels in the previous camera position remain**, leaving trails.

2️⃣ Two main solutions

Option A: Draw the background gradient in screen coordinates (outside the camera)

- Draw the gradient **before `cam.begin()`** so it always covers the viewport.
- Then draw stars, platforms, and the blob inside the camera translation.

```
function draw() {
 // --- background gradient (screen-space) ---
 noFill();
 for (let y = 0; y < height; y++) {
  const inter = map(y, 0, height, 0, 1);
  const c1 = color('#0d1b2a'); // top
  const c2 = color('#1b263b'); // bottom
  stroke(lerpColor(c1, c2, inter));
  line(0, y, width, y);
 }

 // --- stars, platforms, blob inside camera ---
 cam.begin();
 level.drawStarsAndPlatforms();  // draw stars + platforms + craters
 player.draw(level.theme.blob);
 cam.end();
}
```

- Notice stars can also be **drawn relative to world coordinates**, but the background gradient must **not move with the camera**, otherwise it trails.

Option B: Draw a "full-world" background inside the camera

- If you want the gradient to move with the camera (parallax effect), you **must draw enough of it to cover the entire visible camera area**, and update stars properly.
- But it's trickier: you need to account for camera offset:

```
cam.begin();

// draw gradient relative to camera
for (let y = cam.y; y < cam.y + height; y++) {
 const inter = map(y, cam.y, cam.y + height, 0, 1);
 stroke(lerpColor(c1, c2, inter));
```

```
 line(cam.x, y, cam.x + width, y);
}

// then draw stars, platforms, blob
level.drawStarsAndPlatforms();
player.draw(level.theme.blob);

cam.end();
```

- This works but is less efficient than Option A.

✅ Quick fix recommendation

For your side-scroller:

1. Draw the **gradient background outside the camera** (screen-space).
2. Draw stars, platforms, blob **inside the camera**.
3. This prevents trails completely.

If you want, I can **rewrite your `WorldLevel.drawWorld()` + `draw()` loop** so your **gradient + stars + blob + platforms** all work with the camera **and don't leave trails**. It will be ready to copy-paste.

Do you want me to do that?

Prompt 9:

how can i add a little alien at the end of the map

Response 9:

Perfect! You can add a little alien as a **sprite or ellipse/shape** at the end of your level. Since your game is modular and data-driven, you can do this **without touching `levels.json`** by adding the alien in `WorldLevel.drawWorld()`.

Here's a simple step-by-step:

1️⃣ Add alien data

In `WorldLevel`'s constructor, add a property for the alien:

```
class WorldLevel {
 constructor(levelJson) {
  // ...existing code...

  // alien at end of map
  this.alien = {
   x: this.w - 100,       // near the right edge of the world
   y: 350,                // height above the ground
   r: 20,                 // size of alien
   color: color(0, 255, 0) // green
  };
 }
}
```

- `x` = horizontal position (place near the end of world: `this.w - 100`)
- `y` = vertical position (above the platforms/ground)
- `r` = radius of alien
- `color` = any color you like

2️⃣ Draw the alien

In `drawWorld()`, **after drawing platforms and craters**:

```
drawWorld() {
 // ... existing stars, platforms, craters ...

 // draw alien
 fill(this.alien.color);
 noStroke();
 ellipse(this.alien.x, this.alien.y, this.alien.r * 2); // simple round alien

 // optional: add eyes
 fill(0);
 ellipse(this.alien.x - 5, this.alien.y - 5, 5);
 ellipse(this.alien.x + 5, this.alien.y - 5, 5);
}
```

- The alien will now be part of the world and will **move with the camera**.
- You can make it more fun by adding a **wobble**, **arms**, or **animation**.

3️⃣ Optional: Trigger interaction

If you want the alien to do something when the player reaches it:

```
function checkAlienCollision() {
 const dx = player.x - level.alien.x;
 const dy = player.y - level.alien.y;
 const dist = sqrt(dx*dx + dy*dy);
 if (dist < player.r + level.alien.r) {
   console.log("You found the alien!");
   // e.g., show message, play sound, advance level, etc.
 }
}
```

- Call `checkAlienCollision()` in your main `draw()` after `player.update(level);`.

If you want, I can rewrite **`WorldLevel.drawWorld()`** **fully** so it includes:

- Gradient background
- Stars
- Platforms & craters
- Alien at the end

…all camera-safe and ready to copy-paste.

Do you want me to do that?