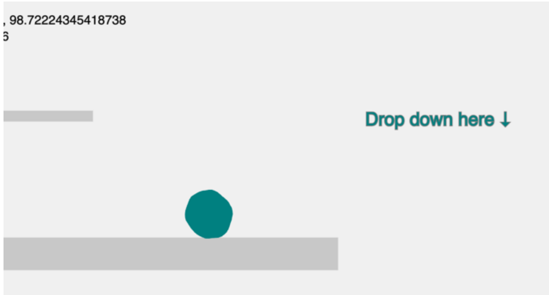
Process & Decision Documentation

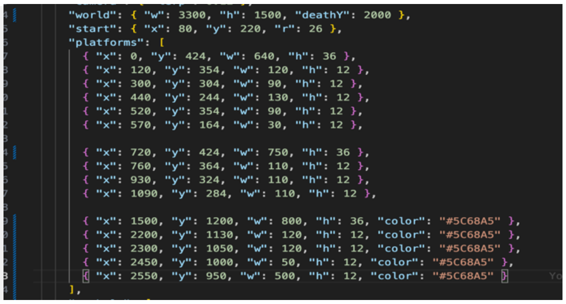
Project/Assignment Decisions

- I had an idea of making the player drop down into another platform which required me to expand the entire map boundaries for width, height, and the death drop constraints.
- Since the blob is supposed to drop down, I decide to transition the background from grey to blue to express an underwater environment. The direction of the game turned into an environmental experience. I also made sure to add a "drop down here" to show the player that they must drop down to progress the experience.
- To create more dynamic movement, I increased the how far the camera moves and tracked the "y" camera to make the camera follow the player's jumps

Role-Based Process Evidence



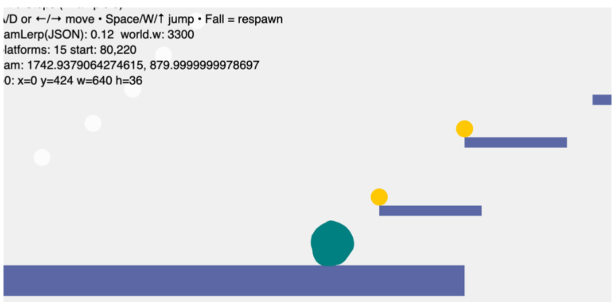
```
, 98.72224345418738
6
```



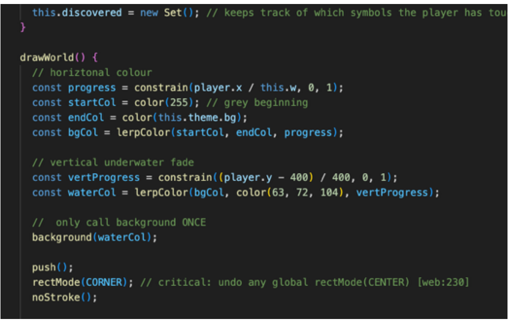
```
"world": { "w": 3300, "h": 1500, "deathY": 2000 },
"start": { "x": 80, "y": 220, "r": 26 },
"platforms": [
  { "x": 0, "y": 424, "w": 640, "h": 36 },
  { "x": 120, "y": 354, "w": 120, "h": 12 },
  { "x": 300, "y": 304, "w": 90, "h": 12 },
  { "x": 440, "y": 244, "w": 130, "h": 12 },
  { "x": 520, "y": 354, "w": 90, "h": 12 },
  { "x": 570, "y": 164, "w": 30, "h": 12 },

  { "x": 720, "y": 424, "w": 750, "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": 1500, "y": 1200, "w": 800, "h": 36, "color": "#5C68A5" },
  { "x": 2200, "y": 1130, "w": 120, "h": 12, "color": "#5C68A5" },
  { "x": 2300, "y": 1050, "w": 120, "h": 12, "color": "#5C68A5" },
  { "x": 2450, "y": 1000, "w": 50, "h": 12, "color": "#5C68A5" },
  { "x": 2550, "y": 950, "w": 500, "h": 12, "color": "#5C68A5" }
],
```



```
/D or ←/→ move • Space/W/↑ jump • Fall = respawn
amLerp(JSON): 0.12  world.w: 3300
latforms: 15 start: 80,220
am: 1742.9379064274615, 879.9999999978697
0: x=0 y=424 w=640 h=36
```



```
    this.discovered = new Set(); // keeps track of which symbols the player has tou
  }

  drawWorld() {
    // horiztonal colour
    const progress = constrain(player.x / this.w, 0, 1);
    const startCol = color(255); // grey beginning
    const endCol = color(this.theme.bg);
    const bgCol = lerpColor(startCol, endCol, progress);

    // vertical underwater fade
    const vertProgress = constrain((player.y - 400) / 400, 0, 1);
    const waterCol = lerpColor(bgCol, color(63, 72, 104), vertProgress);

    //  only call background ONCE
    background(waterCol);

    push();
    rectMode(CORNER); // critical: undo any global rectMode(CENTER) [web:230]
    noStroke();
```

Side Quests and A4 (Individual Work)

Keep this section brief, typically 2 to 4 sentences.

- Initially, I wanted the orbs to be collected by the player and it would be saved into a counter. But to save time, I wanted to shift the focus from adding complex mechanics to refining atmosphere and interaction, while keeping the experience calm and cohesive. Thus, I made it so the when the player collides with an orb it changes their colour, hinting for players to change the rest of the orb's colour.

GenAI Documentation

Gen AI was used to debug code and guide how to code loop functions.

If GenAI was used (keep each response as brief as possible):

Date Used: February 20 2026

Tool Disclosure: ChatGPT 5.2

Purpose of Use: I used AI for both brainstorming, coding help,(suggesting me methods on how to approach the bubbles floating up the water) and debugging (helped fixed errors that were not working).

Summary of Interaction: I interacted with GenAI through iterative prompts, sharing code snippets of my own attempts, described my issues, and objectives so that AI can help debug my code, such as trying to get the bubble movement and the camera movement the way I want by simplifying the logic. I tested suggestions, and adjusted them to fit my vision while using ongoing feedback to guide my work.

Human Decision Point(s): I rejected and repeatedly asked for more cleaner and easier methods for my own understanding and readability. For example, the bubble movement I moved away from math-heavy and layered formula coding to a more cleaner approach with loop functions. I was able to experiment with the "%" function and understand what it does. Additionally, I steered the project towards a mediative, and reflective tone through variables changes like background colours, placement of platform, and camera movement.

Integrity & Verification Note: Explain how you checked GenAI output for accuracy, bias, appropriateness, or fit with course concepts.

I checked GenAI output for accuracy by cross-checking with course concepts and the actual instructions of this week's side quest prompt to ensure that my code met the requirements of creating a reflective camera experience. I ensured that I understood the code and play tested all the code to ensure it functioned correctly.

Scope of GenAI Use:

- Used GenAI to explore ways to make the camera more mediative and less rigid
- Adjusted camera lookahead length myself to match the tone of the game
- Majority of GenAI was to help debug and code the floating bubbles which used sin, frame ount, and the remainder option to help create the mood
- Made the final decisions independently to ensure the camera supported the calm experience, and the blob is able to navigate the scrollable world

Limitations or Misfires: Certain camera adjustments initially caused the movement to be overly floaty or unresponsive, so I had to manually change it. Some suggestions did not align with my existing code structure, or it expanded the scope to beyond what was necessary, which I had to simplify.

Summary of Process (Human + Tool)

Describe what you did, focusing on process rather than outcome. This may include:

- Asked AI to give suggestions on how I should make the camera movement even more dynamic while remaining smooth and not abrupt
- AI suggested to make new bounce property and introduced new functions, but I just used this.vy *= 0.95 for a simple float effect when falling
- Manually adjusted lerp values, strength of lookahead, and placed orbs on platform.
- Asked AI to help debug errors of my code or why it isn't giving the code I want, for example: "Why are the bubbles not resetting even after it reaches above the 500 height condition I set?"
- I asked for explanations on how these functions work and the logic of getting the bubbles to move upward continuously by using modulo
- Through testing, I realize I can breakdown the bubble code into more steps and simplfy the upward motion to be more controlled

Verification & Judgement

Explain how you evaluated whether your decision or change was appropriate:

I evaluated each change by directly testing it in my environment and observing how it affected my gameplay, for instance:

- When the bubble resets or glitches, or it didn't respawn after going over a threshold like I wanted, I reduced the complexity, manually altered areas of code and iteratively prompted AI for further feedback. Final decisions were based on whether the mechanics felt smooth, intentional, and kept my ideas.

Limitations, Dead Ends, or Open Questions

- Early floating bubble movements using %, and frame counts were harder to debug and did not produce the flow I want, leading to rework.
- Furthermore, I wanted to make the camera movement even more smooth like in some of those games where the camera is extremely dynamic yet smooth despite changing the lerp I couldn't replicate that feeling.

Appendix

Me: How could I gradually shift the background from grey to full colour as the player moves across the level? I'm not sure how to structure that

Prompt: To gradually restore colour as the player moves across the level, you can calculate how far they've progressed horizontally compared to the total world width. That gives you a progress value between 0 and 1. You then use that value to blend between a starting grey colour and your final background colour each frame before drawing the background.

Code Snippet (inside `drawWorld()`)

```
// 1. Calculate horizontal progress (0 → 1)
const progress = constrain(player.x / this.w, 0, 1);

// 2. Define start and end colours
const startCol = color(220);           // grey (desaturated)
const endCol = color(this.theme.bg);   // full theme colour

// 3. Blend between them based on progress
const bgCol = lerpColor(startCol, endCol, progress);

// 4. Apply background
background(bgCol);
```

This keeps the structure simple:

- Player movement → progress value
- Progress value → colour interpolation
- Colour interpolation → emotional visual shift

Me: I need the camera the look forward more, which functions should I change so that when the blob moves the camera would be more dynamic?

Prompt: Nice — this is exactly where the camera starts to feel *alive* instead of attached to the player.

Right now your camera does:

player moves → camera immediately follows → feels tight / platformer

You want:

player moves → camera hesitates → then glides → looks ahead further

That's called **temporal lag + predictive lookahead**.

We'll build it in 3 simple layers:

🧠 The Idea

We separate the camera into 3 positions:

1. **Player position** (real)
2. **Target position** (delayed memory of player)
3. **Camera position** (slowly follows target)

So the camera reacts late — like a human turning their head.

1️⃣ Add a delayed target (camera memory)

Open `camera.js`

Add variables in constructor:

```
this.targetX = 0; // remembered player pos
this.lookAhead = 0;
```

Replace `followSideScrollerX` completely

```
followSideScrollerX(playerX, playerVX, lerpAmt) {

  // --- 1. delay: camera remembers where player WAS ---
  this.targetX = lerp(this.targetX, playerX, 0.06);
  // smaller = more delay (0.03 very floaty, 0.15 snappy)

  // --- 2. predictive look ahead ---
  const desiredLook = playerVX * 90;
  this.lookAhead = lerp(this.lookAhead, desiredLook, 0.08);

  // --- 3. camera follows the delayed target ---
  const desired = this.targetX + this.lookAhead - this.viewW / 2;
  this.x = lerp(this.x, desired, lerpAmt * 0.7);
}
```

And keep your sketch call:

```
cam.followSideScrollerX(player.x, player.vx, level.camLerp);
```

What this changes (very important)

Before:

camera glued to player

Me: followSideScrollerX(targetX, targetvx, lerpAmt) { const lookAhead = targetvx * 20; // how far camera anticipates const desired = targetX - this.viewW / 2 + lookAhead; this.x = lerp(this.x, desired, lerpAmt); } followSoftY(targetY, lerpAmt) { const desired = targetY - this.viewH / 2; this.y = lerp(this.y, desired, lerpAmt * 0.6); } 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();, I did the same structure for the "Y" Camera, but this isnt mkaing the camera rise when I jump, I feel no difference

Prompt: You actually broke the Y camera for **two reasons at once** — both subtle:

🧨 Problem — You used the wrong velocity

```
followSoftY(targetY, targetxv, lerpAmt) {
  const lookAheady = targetxv * 20;
```

You passed **X velocity** into the Y camera.

So when you jump:

- `player.vx ≈ 0`
- lookAheady = 0
- desired ≈ constant
- camera barely changes

👉 The Y camera must use **vertical velocity (`vy`)**, not `vx`.

Me: In this game file, which functions give the blob a float effect? I tried changing the gravity to be lower and the jump height to be higher but I don't think that did anything.

Prompt: Floaty = slower fall + softer gravity.

Reduce gravity

```
this.gravity = 0.4;    // was 0.65
```

Increase jump height slightly

```
this.jumpV = -13.5;    // was -11.0
```

Lower gravity + stronger jump = longer airtime.

✅ 2. Add Bounce on Landing

Right now you stop completely when hitting ground:

```
this.vy = 0;
```

Instead, multiply by a bounce factor.

Add a bounce property in constructor:

```
this.bounce = 0.4; // 0 = no bounce, 1 = super ball
```

Change your vertical collision:

Replace this:

```
this.vy = 0;
this.onGround = true;
```

With this:

```
this.vy *= -this.bounce;

if (abs(this.vy) < 1) {
  this.vy = 0;
  this.onGround = true;
}
```

Now it'll bounce slightly before settling.

Try:

- `0.3` = soft jelly bounce
- `0.6` = playful blob
- `0.8` = chaotic rubber ball

✅ 3. Make Air Movement More Floaty

Reduce air friction slightly:

```
this.frictionAir = 0.998;  // was 0.995
```

This makes horizontal movement feel smoother mid-air.

You can also increase air control slightly:

```
this.accel = 0.5;   // was 0.4
```

✅ 4. Optional: Add Squash & Stretch (Makes It FEEL Bouncy)

This dramatically improves the perception of bounce.

Inside `draw()` before `beginShape()`:

```
push();
translate(this.x, this.y);

let stretchX = 1;
let stretchY = 1;

// squash when landing
if (this.onGround && abs(this.vy) < 0.5) {
  stretchX = 1.2;
  stretchY = 0.8;
}

// stretch when jumping
if (!this.onGround) {
  stretchX = 0.9;
  stretchY = 1.1;
}

scale(stretchX, stretchY);
```

Then change your vertices to:

```
vertex(cos(a) * rr, sin(a) * rr);
```

After `endShape(CLOSE);` add:

```
pop();
```

That alone will make it feel WAY more alive.

🎮 Recommended "Cute Floaty Blob" Settings

```
this.gravity = 0.4;
this.jumpV = -13.5;
this.frictionAir = 0.998;
this.frictionGround = 0.9;
this.bounce = 0.45;
```

Me: when player goes below 400 pixels, fade to blue let vertProgress = constrain((player.y - 400) / 400, 0, 1); // 0→400 = fade const waterCol = lerpColor(bgCol, color(100, 150, 255), vertProgress); // bluish background(waterCol); where does this specifically go?

Prompt: Good question — placement matters a LOT here 👇

You already have this in `WorldLevel.drawWorld()`:

```
const progress = constrain(player.x / this.w, 0, 1);

const startCol = color(220);
const endCol = color(this.theme.bg);

const bgCol = lerpColor(startCol, endCol, progress);

background(bgCol);
```

Right now the background only reacts to **horizontal movement**.

Where the vertical water fade goes

You **replace the `background(bgCol);` line**, not add a second background later.

Because `background()` clears the screen — if you call it twice, the second one erases the first fade.

So it should become:

```
const progress = constrain(player.x / this.w, 0, 1);

// horizontal color restoration (Gris-style)
const startCol = color(220);
const endCol = color(this.theme.bg);
const bgCol = lerpColor(startCol, endCol, progress);

// vertical underwater fade
let vertProgress = constrain((player.y - 400) / 400, 0, 1);
const waterCol = lerpColor(bgCol, color(100, 150, 255), vertProgress);

// FINAL background (only call background once)
background(waterCol);
```

Why this works

You now have **two layered emotional effects combined**:

1. Move right → world gains color
2. Fall down → world turns blue

So the color pipeline becomes:

```
gray → themed color → underwater blue
```

Important

Make sure this is at the **very top of `drawWorld()`**, before:

- platforms
- symbols
- bubbles
- player rendering

Because background must be drawn first every frame.

Me: How can I bubbles drift upward but only when blob passed a certain point like its coming from this platform { "x": 1500, "y": 1200, "w": 800, "h": 36, "color": "#5C68A5" },, starting from here and drift upwards, would this use loop function to create the bubbles?

Prompt:Put this **after you draw platforms**, but before `pop();`

```
// -------- UNDERWATER BUBBLES --------
if (player.x > 1500) {

  for (let i = 0; i < 25; i++) {

    const baseX = 1500 + (i * 70) % 800;  // spread across 800 width
    const rise = (frameCount * 0.8 + i * 60) % 600; // upward motion loop

    const bubbleX = baseX + sin(frameCount * 0.02 + i) * 15;
    const bubbleY = 1200 - rise;

    fill(255, 255, 255, 40);
    noStroke();
    ellipse(bubbleX, bubbleY, 8, 8);
  }

}
```

What This Does

When blob.x > 1500:

- Bubbles spawn around y = 1200
- They rise upward
- They gently sway side to side
- They loop continuously
- They feel soft and ambient

Me: can you break down the code to be easier to understand for me, since this is a very condensed and logical code?

Prompt: Step 1 — Add a bubble list to the level

Inside `WorldLevel` constructor:

```
this.bubbles = [];
```

Step 2 — Spawn bubbles (simple!)

Inside `drawWorld()` (near the top or bottom — doesn't matter):

```
// spawn bubbles once player reaches underwater area
if (player.x > 1500 && frameCount % 20 === 0) {
  this.bubbles.push({
    x: random(1500, 2300),
    y: 1200,
    size: random(4, 10),
    speed: random(0.5, 1.2)
  });
}
```

That's it — every ~20 frames a bubble appears.

Step 3 — Move & draw them

Still inside `drawWorld()`:

```
for (let b of this.bubbles) {
  b.y -= b.speed; // move upward

  fill(255, 255, 255, 40);
  noStroke();
  ellipse(b.x, b.y, b.size);

  // remove when off screen
  if (b.y < 400) b.dead = true;
}
```

```
// cleanup
this.bubbles = this.bubbles.filter(b => !b.dead);
```

What you get

- bubbles appear only after the drop
- they gently rise
- they disappear near surface
- code is readable
- feels calmer (less mechanical looping)

Tiny polish (optional but nice)

Add a slight drift:

```
b.x += sin(frameCount * 0.02 + b.y) * 0.2;
```

Now they wobble softly like real water.

Me: I don't think the reset is working even when I sent the condition still. // reset when too high if (bubbleY < 1200) { bubbleY += 800

Prompt: Right now it keeps snapping because you reset it when it reaches the **surface (1200)** — so every bubble teleports at the same visible place.

Instead, reset it **before the player can see the top**, so it feels continuous.

Think of it like this:

top of water = ~500
bottom spawn = 1200
reset slightly above the camera view

Just change your reset line

Replace:

```
if (bubbleY < 1200) {
  bubbleY += 800;
}
```

with:

```
// reset BEFORE reaching visible surface
if (bubbleY < 500) {
  bubbleY = 1200 + i * 30;
}
```

Why this works

- 500 = above the visible underwater area
- 1200 = deep water spawn
- `i * 30` staggers them so they don't all pop at once

Now bubbles:

- rise
- disappear naturally
- re-enter from bottom
- never visibly teleport

Me: // reset if gone off top if (b.y < 500) { b.y = 1200 + random(0, 100); // send back to bottom } this makes it glitch , how do I fix?

Prompt: Ah! Yep — that "glitch" happens because the reset jumps the bubble **way below the camera suddenly**, so visually it looks like it pops instead of smoothly looping.

To make it **truly smooth**, we don't "teleport" the bubble; instead we **wrap it gradually** so it loops continuously. The trick is to use a **modulo-style offset** from the starting bottom position.

Here's a simpler, smooth approach:

```
// define at top of WorldLevel
this.bubbles = [];
for (let i = 0; i < 25; i++) {
  this.bubbles.push({
    x: random(1500, 2300),  // horizontal spread
    y: random(500, 1200),   // initial vertical position
    size: random(6, 12),
    speed: random(0.5, 1.2)
  });
}

// inside drawWorld()
if (player.x > 1500) {
  for (let b of this.bubbles) {
```

```
  b.y -= b.speed; // move upward
  b.x += sin(frameCount * 0.02 + b.y) * 0.2; // gentle drift

  // smooth looping: wrap around without popping
  if (b.y < 500) {
   b.y = 1200 - (500 - b.y); // reflect back down smoothly
  }

  fill(255, 255, 255, 40);
  noStroke();
  ellipse(b.x, b.y, b.size);
 }
}
```

✅ Why this is smooth

- Instead of `b.y = 1200 + random()`, we calculate how far above the top it went: `(500 - b.y)`
- Then subtract that from 1200 → bubble continues moving naturally
- No visible teleporting or popping

Me: I added some symbols in the water and also added it to the constructor, the same which I've done with the new bubbles, here's what I have:

"symbols": [

{ "x": 1600, "y": 1200, "r": 12 },

{ "x": 2200, "y": 1130, "r": 10 },

{ "x": 2300, "y": 1050, "r": 10 },

{ "x": 2500, "y": 950, "r": 14 }

]

```
this.bubbles = [];
 this.symbols = levelJson.symbols ?? [];
```

How can I make it change colour when colliding? Can you give me a guiding structure?

Prompt: Instead of thinking "how do I change the colour?", think in terms of:

How do I know a symbol has been touched, and how do I remember that?

Step 1: Detect Collision (Structure)

Somewhere in your main update loop (after the player moves), you'll want to:

1. Loop through your `level.symbols`
2. Check the distance between the player and each symbol
3. If they overlap, update something about that symbol

Structurally it looks like:

```
for (let i = 0; i < level.symbols.length; i++) {
  const s = level.symbols[i];

  // calculate distance between player and symbol

  // if overlapping:
  //    mark this symbol as "activated"
}
```

The key idea here is that collision detection is just comparing:

- Player position + radius
- Symbol position + radius

Step 2: Give Each Symbol a State

Right now your symbols only have:

```
{ "x": ..., "y": ..., "r": ... }
```

To allow colour change, each symbol needs a *property that can change*. For example:

- `activated`
- `collected`
- `touched`

You don't need to change your JSON — you can add this property dynamically when a collision happens.

Conceptually:

```
s.activated = true;
```

Now the symbol remembers it has been touched.

Step 3: Change Colour When Drawing

Inside `drawWorld()`, where you draw the symbols, you can decide the colour based on that state.

Structure:

```
for (const s of this.symbols) {

  if (s.activated) {
    // draw changed colour
  } else {
    // draw default colour
  }

  ellipse(s.x, s.y, s.r * 2);
}
```

So instead of removing the symbol, you’re just changing how it renders.