Process & Decision Documentation

Project/Assignment Decisions

During my process I had some troubles implementing a start feature. It would either crash the game, freeze it, or nothing will load at all. I had a if statement system set up and had to constantly adjust what the variable means and what would be in each if statement. Soon I managed to find the sequence that worked. Additionally, I adjusted the game to be slower and have smaller jumps to make it easier. The first iteration was too fast and the larger jumps made it extremely difficult to make it through the first pipe.

GenAI Documentation

Date Used: Monday February 9th

Tool Disclosure: ChatGPT

Purpose of Use: I used Chatgpt to assist with creating and troubleshooting a feature for the side quest

Summary of Interaction: I asked Chatgpt how to modify the blob from an example to have a flappy bird mechanic versus the original one. Additionally I asked for assistance with troubleshooting and creation of the world.

Human Decision Point(s): I would only use code ChatGPT gave me if it was what I was looking for. It would throw in unnecessary details or components that I wasn't interested in. Additionally, I modified and tested the code provided to see if it would even function well.

Integrity & Verification Note: I noticed when asking for assistance ChatGPT didn't truly understand that everything was divided in different pages. Which meant I had to test and edit code during implementation as it was coded in a way where everything is on one page

Scope of GenAI Use: Gen AI was used for the flying and pipe features as well as troubleshooting

Limitations or Misfires: Because it didn't have a full picture of the entire code it would constantly create new errors. I would go through and manually undo and edit to get a working product,

Summary of Process (Human + Tool) The process involved getting a base from chatgpt. There I edited and adjusted components to fit my vision. If I arrived at any errors I was unable to solve I would ask ChatGPT how I could fix it. I would test and edit the solutions as the majority of the time it wasn’t fixing much. It did however show me a path I should take for fixing said error.

Verification & Judgement: Without context ChatGPT has to make a lot of assumptions. It assumed a variable was undefined since it was on a different page. It would forget any previous code you showed it, causing misleading suggestions. It helped with pointing out where I should implement items or how to go about fixing something. However, it wasn’t too effective at actual solutions.

Appendix

Me: In p5.js how would I convert this blob to a flappy bird mechanic where the blob's x value is fixed in place and players must click space, the up arrow, or w to make it fly?

```
/*
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
  );
}
```

ChatGPT: Ah — this is a nice conversion to make 👍
Flappy-Bird mechanics are actually much simpler than what you have here, because:

- ❌ No horizontal movement
- ❌ No platform collisions
- ❌ No grounded jumping logic
- ✅ Constant gravity pulling down
- ✅ Player presses a key → apply **instant upward impulse**
- ✅ Blob’s **x stays fixed**

So we’ll mostly strip the platformer logic out of `update()` and change `jump()` into a “flap”.

I’ll walk you through the minimal changes.

✅ Core idea

Replace this behavior:

- Horizontal accel/friction
- Collision resolution
- `onGround` checks

With:

- Fixed `x`
- `vy += gravity`
- `y += vy`
- Key press → `vy = flapStrength`

✏️ Step-by-step modifications

1️⃣ Add flap tuning variables

In constructor:

```
// Flappy mechanics tuning
this.fixedX = 200;        // where blob stays horizontally
this.flapStrength = -10;  // upward impulse
this.maxFall = 12;        // clamp falling speed (optional)
```

2️⃣ Modify spawn

Lock x position:

```
spawnFromLevel(level) {
  this.gravity = level.gravity;

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

  this.vx = 0;
  this.vy = 0;
}
```

(We don’t need `jumpV` or `onGround` anymore.)

3️⃣ Replace update() with Flappy physics

You can completely replace your current `update()` with this:

```
update() {

  // Apply gravity
  this.vy += this.gravity;

  // Clamp fall speed (prevents ridiculous acceleration)
  this.vy = constrain(this.vy, -Infinity, this.maxFall);

  // Move vertically
  this.y += this.vy;

  // Lock horizontal position
  this.x = this.fixedX;

  // Optional: keep inside screen
  this.y = constrain(this.y, this.r, height - this.r);
  // Advance blob animation
  this.t += this.tSpeed;
}
```

No platforms needed anymore.

4️⃣ Change jump() → flap()

Flappy input should work anytime (no ground check):

```
flap() {
  this.vy = this.flapStrength;
}
```

5️⃣ Add keyboard trigger

In your main sketch:

```
function keyPressed() {
  if (
    key === ' ' ||
    keyCode === UP_ARROW ||
    key === 'w' ||
    key === 'W'
  ) {
    blob.flap();
  }
}
```

⭐ That’s it — you now have Flappy Blob

Behavior will be:

- Gravity constantly pulls down

- Press Space / ↑ / W
- Blob gets instant upward kick
- X never moves

Exactly the Flappy Bird control feel.

🧠 (Optional polish ideas — HIGHLY recommended)

These are small touches that make it feel *way better*:

Smooth impulse instead of hard reset

```
this.vy += this.flapStrength;
```

instead of

```
this.vy = this.flapStrength;
```

Gives stacked flaps.

Slight rotation based on velocity

Add in `draw()` before vertices:

```
push();
translate(this.x, this.y);
rotate(map(this.vy, -10, 10, -PI/6, PI/6));
translate(-this.x, -this.y);
```

Then `pop();` after drawing — makes the blob tilt.

Screen death conditions

```
if (this.y <= this.r || this.y >= height - this.r) {
  gameOver();
}
```

👉 If you want — next step

I can help you add:

✅ Pipe spawning
✅ Scrolling world illusion
✅ Score system
✅ Increasing difficulty
✅ Sound on flap

Just tell me 🙂

Me: has this been implemented correctly?

```
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;

// Flappy mechanics tuning
this.fixedX = 200;        // where blob stays horizontally
this.flapStrength = -10;  // upward impulse
this.maxFall = 12;        // clamp falling speed (optional)


    // ----- 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.x = this.fixedX;
  this.y = level.start.y;
  this.r = level.start.r;

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


  /*
  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;

    // 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;


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);
  }
}

flap() {
  this.vy = this.flapStrength;
}

function keyPressed() {
  if (
    key === ' ' ||
    keyCode === UP_ARROW ||
    key === 'w' ||
    key === 'W'
  ) {
    blob.flap();
  }
}

/*
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
  );
}
```

ChatGPT: You’re close — but no, this hasn’t actually been converted to Flappy mechanics yet 🙂
Right now you’ve got a **mix of platformer code + partial Flappy additions**, which will cause weird behavior.

Let me point out exactly what’s happening and how to fix it.

🚨 Issues in your current version

❌1️⃣ `update()` is still the platformer system

You still have:

- Horizontal input (`A/D`)
- Collision AABB box logic
- Platform resolution
- `this.x = constrain(...)`
- `onGround` logic

👉 Flappy Bird doesn't use any of this.

Your blob is still basically a platform character — just with gravity added.

❌2️⃣ `flap()` is outside the class

You wrote:

```
flap() {
  this.vy = this.flapStrength;
}
```

This is **not inside `class BlobPlayer`**, so JS treats it as invalid syntax.

That method must be inside the class body.

❌3️⃣ `jump()` still exists

Flappy doesn't need:

```
jump()
```

You should remove it (or it will conflict with behavior expectations).

❌4️⃣ X position isn’t actually fixed

Inside `update()` you do:

```
this.x = box.x + box.w / 2;
```

and constrain it

That completely overrides `fixedX`.

✅ What a correct Flappy conversion looks like

I’ll give you the **minimal corrected version** so you can compare.

This keeps your rendering and structure intact.

⭐ Replace ONLY these parts

✅ Put flap inside class

Add this inside `class BlobPlayer { ... }`

```
flap() {
  this.vy = this.flapStrength;
}
```

✅ Replace `update()` completely

Delete your entire current update function and use this:

```
update() {
  // Apply gravity
  this.vy += this.gravity;

  // Clamp falling speed
  this.vy = constrain(this.vy, -Infinity, this.maxFall);

  // Move vertically
  this.y += this.vy;

  // Lock horizontal position
  this.x = this.fixedX;

  // Keep inside screen
  this.y = constrain(this.y, this.r, height - this.r);

  // Animate blob
  this.t += this.tSpeed;
}
```

✅ Remove these (not needed anymore)

Delete:

```
jump()
```

Delete from constructor:

```
this.accel
this.maxRun
this.jumpV
this.onGround
```

(Delete if you want clean code — not mandatory but recommended)

✅ Keep key input outside class (correct!)

This part is good 👍

```
function keyPressed() {
  if (
    key === ' ' ||
    keyCode === UP_ARROW ||
    key === 'w' ||
    key === 'W'
  ) {
    blob.flap();
  }
}
```

🟢 After fixes — expected behavior

You should see:

- Blob stays at constant X
- Falls downward continuously
- Press key → instant upward boost
- No platform snapping
- Smooth vertical float

That’s true Flappy motion.

⭐ Quick sanity test

If implemented correctly:

Press nothing

Blob should drop to bottom.

Tap Space

Blob should hop upward.

Hold nothing again

Blob falls again.

If that happens — you nailed it.

👉 If you want next-level polish (seriously worth it)

I can show you how to add:

- Pipe spawning
- Horizontal scrolling illusion
- Hit detection
- Score counter
- Juice (squish/stretch blob on flap)

Just say the word 🙂

Me: how can I turn this json loop from platforms to the traditional flappy bird pipes

```
{
  "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 }
      ]
    }
  ]
}
```

ChatGPT: Good question — this is exactly the right moment to rethink the structure a bit 🙂

Right now your JSON is built for a **platformer**, where:

- Platforms are static rectangles
- They sit at fixed x/y
- You loop through them for collision/drawing

Flappy Bird pipes are fundamentally different:

✅ They come in **pairs (top + bottom)**
✅ They move horizontally
✅ They have a **gap** between them
✅ They are usually **generated procedurally**, not hand-placed

So we don't just rename `"platforms"` → `"pipes"` — we slightly change the data model.

🧠 Step 1 — Understand the data difference

Platform structure (current)

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

This describes one static rectangle.

Pipe structure (typical Flappy)

We only need:

- x position
- gap vertical center
- gap size
- width

Because the top/bottom rectangles are computed.

Pipe JSON entry

{

```
  "x": 500,
  "gapY": 200,
  "gapH": 140,
  "w": 70
}
```

That defines BOTH pipes.

✅ Step 2 — Convert your JSON

Here’s a Flappy-ready version of your level:

```
{
  "schemaVersion": 1,
  "levels": [
    {
      "name": "Intro Pipes",
      "gravity": 0.65,
      "theme": { "bg": "#F0F0F0", "pipe": "#C8C8C8", "blob": "#1478FF" },

      "start": { "x": 200, "y": 220, "r": 26 },

      "pipes": [
        { "x": 400, "gapY": 200, "gapH": 140, "w": 70 },
        { "x": 600, "gapY": 150, "gapH": 140, "w": 70 },
        { "x": 800, "gapY": 240, "gapH": 140, "w": 70 },
        { "x": 1000, "gapY": 180, "gapH": 140, "w": 70 }
      ]
    }
  ]
}
```

Notice:

- Increasing x spacing
- Only gap position changes
- Much cleaner data

✅ Step 3 — Drawing loop (conversion from platforms loop)

OLD platform loop

You probably had something like:

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

NEW pipe loop

Replace with:

```
for (const pipe of level.pipes) {

  let topH = pipe.gapY - pipe.gapH/2;
  let bottomY = pipe.gapY + pipe.gapH/2;
  let bottomH = height - bottomY;

  // Top pipe
  rect(pipe.x, 0, pipe.w, topH);

  // Bottom pipe
  rect(pipe.x, bottomY, pipe.w, bottomH);
}
```

That generates both pieces automatically.

✅ Step 4 — Movement (the Flappy magic)

Platforms never moved. Pipes should.

Inside `draw()` or update loop:

```
for (const pipe of level.pipes) {
  pipe.x -= 3;   // scrolling speed
}
```

Now the world scrolls toward the blob.

⭐⭐⭐ Step 5 — Collision (SUPER important)

Instead of AABB with platforms, check blob vs two rectangles:

```
function hitPipe(blob, pipe) {

  let topH = pipe.gapY - pipe.gapH/2;
  let bottomY = pipe.gapY + pipe.gapH/2;

  // top collision
  if (
    blob.x + blob.r > pipe.x &&
    blob.x - blob.r < pipe.x + pipe.w &&
    blob.y - blob.r < topH
  ) return true;

  // bottom collision
  if (
    blob.x + blob.r > pipe.x &&
    blob.x - blob.r < pipe.x + pipe.w &&
    blob.y + blob.r > bottomY
  ) return true;

  return false;
}
```

🟣 BIG DESIGN TIP (from actual Flappy architecture)

Most real implementations **don't store pipes in JSON** because:

- They're random
- Endless
- Procedural

Typical approach:

```
spawnPipeEveryXSeconds()
randomize gapY
push into array
```

Your JSON version is totally fine for coursework / level design though 👍

👉 If you want — next step

I can show you the clean next evolution:

⭐ Procedural pipe spawning system
⭐ Pipe recycling (infinite scrolling)
⭐ Score detection when passing gap
⭐ Difficulty scaling

Just tell me 🙂

Me: this is the code that draws the world and level, what should I change to implement the pipes?

```
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 === " " || keyCode === UP_ARROW || key === "w" || key === "W") {
    player.flap();
  }

  // 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);
}

/*
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));
    }
  }
}
```

ChatGPT: Ok — this is actually a great place to switch over, because your architecture is already nicely separated 🙂
You don't need to rewrite everything — just redirect a few responsibilities.

Right now the flow is:

JSON → WorldLevel → platforms[] → drawWorld() → player.update(platforms)

For Flappy, we change it to:

JSON → WorldLevel → pipes[] → drawWorld() → player.update() → pipe movement + collision

I'll show you **exactly what to change**, step-by-step.

✅1️⃣ Change WorldLevel to read pipes instead of platforms

Replace THIS line in constructor

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

With THIS

```
this.pipes = (levelJson.pipes || []).map(p => ({...p}));
```

We don’t need a Platform class anymore — pipes are dynamic.

✅2️⃣ Fix canvas sizing

Your `inferWidth/Height()` currently look at platforms.

Replace both functions

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

inferHeight(defaultH = 360) {
  return defaultH; // pipes don't define vertical size
}
```

✅3️⃣ Replace drawWorld()

This is the BIG one.

Replace entire function with:

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

  fill(color(this.theme.pipe));

  for (const pipe of this.pipes) {

    // Move pipes left (scroll world)
    pipe.x -= 3;

    let topH = pipe.gapY - pipe.gapH / 2;
    let bottomY = pipe.gapY + pipe.gapH / 2;

    // Draw top pipe
    rect(pipe.x, 0, pipe.w, topH);
    // Draw bottom pipe
    rect(pipe.x, bottomY, pipe.w, height - bottomY);
  }
}
```

That alone gives you moving pipes 🎉

✅4️⃣Update main sketch draw()

Player should no longer receive platforms.

Change THIS

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

To THIS

```
player.update();
```

(Remember — Flappy physics shouldn’t use platform collisions anymore)

✅5️⃣Add pipe collision detection

Add this helper function (outside classes):

```
function checkPipeCollision(player, pipes) {
  for (const pipe of pipes) {

    let topH = pipe.gapY - pipe.gapH/2;
    let bottomY = pipe.gapY + pipe.gapH/2;

    // Horizontal overlap
    if (
      player.x + player.r > pipe.x &&
      player.x - player.r < pipe.x + pipe.w
    ) {

      // Hit top pipe
      if (player.y - player.r < topH) return true;

      // Hit bottom pipe
      if (player.y + player.r > bottomY) return true;
    }
  }

  return false;
}
```

✅6️⃣ Call collision check in draw()

Add after player.update():

```
if (checkPipeCollision(player, world.pipes)) {
  text("GAME OVER", width/2 - 40, height/2);
  noLoop();
}
```

⭐⭐⭐⭐ VERY IMPORTANT (Architecture Insight)

You just made a structural shift:

Platform Game	Flappy Game
Static geometry	Moving obstacles
Collision resolution	Hit detection
World defines terrain	World spawns hazards

Player moves X World moves X

This is a **fundamental genre change**, and you handled it cleanly by modifying only:

✅ JSON interpretation
✅ World rendering
✅ Player update call

That’s good engine design 🙂

🟢 If you want the next real Flappy step

Right now pipes will eventually leave the screen forever.

The professional next upgrade is:

Pipe Recycling System

- Remove pipes when offscreen
- Spawn new random ones
- Infinite scrolling
- Score tracking

This is the step that makes it feel like an actual game.

If you want, I can walk you through implementing that next 🙂

Me: I am having some trouble with a feature where players click enter to restart after a lose or win, how would I implement it let data; // raw JSON data

```
let levelIndex = 0;

let world; // WorldLevel instance (current level)
let player; // BlobPlayer instance

let loss = 0;
let win = 0;

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();

  if (checkFinCollision(player, world.pipes)) {
    win += 1;
    fill("white");
    rect(width / 2 - 120, height / 2 - 25, 300, 70);
    fill("#6422e0");
    text("You Did It! ", width / 2 - 100, height / 2);
    fill("red");
    text("Win count: " + win, width / 2 - 10, height / 2 + 25);
    noLoop();
  }

  if (checkPipeCollision(player, world.pipes)) {
    loss += 1;
    fill("white");
    rect(width / 2 - 120, height / 2 - 25, 300, 70);
    fill("#6422e0");
    text("GAME OVER. Press ENTER to try again.", width / 2 - 100, height / 2);
    fill("red");
    text("Loss count: " + loss, width / 2 - 10, height / 2 + 25);
    noLoop();
  }

  player.draw(world.theme.blob);
  // 3) HUD
  fill(0);
  text(world.name, 10, 18);
  text(
    "Press enter to start. Use space bar, w, or the up arrow to fly!",
    10,
    36,
  );
}

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

/*
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);
}

function checkPipeCollision(player, pipes) {
  for (const pipe of pipes) {
    let topH = pipe.gapY - pipe.gapH / 2;
    let bottomY = pipe.gapY + pipe.gapH / 2;
    // Horizontal overlap
    if (player.x + player.r > pipe.x && player.x - player.r < pipe.x + pipe.w) {
      // Hit top pipe
      if (player.y - player.r < topH) return true;

      // Hit bottom pipe
      if (player.y + player.r > bottomY) return true;
    }
  }

  return false;
}

function checkFinCollision(player, pipes) {
  for (const pipe of pipes) {
    let topH = pipe.gapY - pipe.gapH / 2;
    let bottomY = pipe.gapY + pipe.gapH / 2;

    // Horizontal overlap
    if (player.x + player.r > pipe.x && player.x - player.r < pipe.x + pipe.w) {
      // Hit top pipe
      if (player.y - player.r < topH) return true;

      // Hit bottom pipe
      if (player.y + player.r > bottomY) return true;
    }
  }

  return false;
}
```

ChatGPT: Ah — you're actually very close. The main issue isn't Enter itself — it's how the game state is structured.

Right now:

- **When win/lose happens → `noLoop()` stops `draw()`**
- **But you never added logic to:**
  - **detect ENTER**
  - **reset the level**
  - **restart the loop (`loop()`)**

So pressing Enter does nothing 🙂

Let's fix this cleanly.

✅ Step 1 — Add a game state flag

Right now you detect win/lose every frame.
We want to track whether the game is running.

Add near the top:

```
let gameState = "start";
// "start" | "playing" | "win" | "lose"
```

✅ Step 2 — Update draw() logic

Replace your win/lose blocks with state transitions.

⭐ Change THIS section

```
player.update();

if (checkFinCollision(player, world.pipes)) {
```

👉 To THIS

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

  player.update();

  if (checkFinCollision(player, world.pipes)) {
    win++;
    gameState = "win";
    noLoop();
  }

  if (checkPipeCollision(player, world.pipes)) {
    loss++;
    gameState = "lose";
    noLoop();
  }
}
```

✅ Step 3 — Draw messages based on state

Add this AFTER player.draw():

```
if (gameState === "win") {
  fill("white");
  rect(width/2-120, height/2-25, 300, 70);

  fill("#6422e0");
  text("You Did It!", width/2-100, height/2);

  fill("red");
  text("Win count: " + win, width/2-10, height/2+25);

  text("Press ENTER to restart", width/2-100, height/2+45);
}

if (gameState === "lose") {
  fill("white");
  rect(width/2-120, height/2-25, 300, 70);

  fill("#6422e0");
  text("GAME OVER", width/2-100, height/2);

  fill("red");
  text("Loss count: " + loss, width/2-10, height/2+25);

  text("Press ENTER to restart", width/2-100, height/2+45);
}
```

✅ Step 4 — Enable ENTER restart

Modify `keyPressed()`:

```
function keyPressed() {

  // Start or restart game
  if (keyCode === ENTER) {
    loadLevel(levelIndex);  // reset world + player
    gameState = "playing";
    loop();                 // restart draw loop
  }

  // Flap only when playing
  if (
    gameState === "playing" &&
    (key === " " || keyCode === UP_ARROW || key === "w" || key === "W")
  ) {
    player.flap();
  }
}
```

⭐⭐⭐ Why this works (important concept)

You just implemented a finite state machine, which is how nearly all games manage flow:

START
 ↓ ENTER
PLAYING
 ↓ collision
WIN / LOSE
 ↓ ENTER
PLAYING

Instead of trying to control everything with `noLoop()` alone.

This scales WAY better when you add:

- menus
- pause
- animations
- multiple levels

🟣 Small Bug You Probably Didn't Notice

These two functions are identical:

checkPipeCollision()
checkFinCollision()

So "winning" and "losing" trigger on the same condition 😅

You likely want:

Win when player passes final pipe

Example:

```
function checkFinCollision(player, pipes) {
  let lastPipe = pipes[pipes.length - 1];
  return player.x > lastPipe.x + lastPipe.w;
}
```

✅ That's it

Now:

- **ENTER starts game**
- **ENTER restarts after loss/win**
- **Controls disabled when not playing**
- **Loop resumes properly**

If you want — next level polish (worth it):

⭐ Freeze pipes on death instead of loop stop
⭐ Add fade overlay
⭐ Reset pipe positions dynamically
⭐ Add countdown restart

Just tell me 🙂

Me: the blob does not show up and the game does not start, how may I fix this?

```
let data; // raw JSON data
let levelIndex = 0;

let world; // WorldLevel instance (current level)
let player; // BlobPlayer instance

let loss = 0;
let win = 0;
let gameState = "playing";

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();
  player.update();

  if (checkFinCollision(player, world.finish)) {
    win++;
    gameState = "win";
    noLoop();
  }

  if (checkPipeCollision(player, world.pipes)) {
    loss++;
    gameState = "lose";
    noLoop();
  }

  player.draw(world.theme.blob);

  // 3) HUD
  fill(0);
  text(world.name, 10, 18);
  text(
    "Press enter to start. Use space bar, w, or the up arrow to fly!",
    10,
    36,
  );

  if (gameState === "win") {
    fill("white");
    rect(width / 2 - 120, height / 2 - 25, 300, 70);

    fill("#6422e0");
    text("You Did It!", width / 2 - 100, height / 2);

    fill("red");
    text("Win count: " + win, width / 2 - 10, height / 2 + 25);

    text("Press ENTER to restart", width / 2 - 100, height / 2 + 45);
  }

  if (gameState === "lose") {
    fill("white");
    rect(width / 2 - 120, height / 2 - 25, 300, 70);

    fill("#6422e0");
    text("GAME OVER", width / 2 - 100, height / 2);

    fill("red");
    text("Loss count: " + loss, width / 2 - 10, height / 2 + 25);

    text("Press ENTER to restart", width / 2 - 100, height / 2 + 45);
  }
}

function keyPressed() {
  // Start or restart game
  if (keyCode === ENTER) {
    loadLevel(levelIndex); // reset world + player
    gameState = "playing";
    loop(); // restart draw loop
  }

  // Flap only when playing
  if (
    gameState === "playing" &&
    (key === " " || keyCode === UP_ARROW || key === "w" || key === "W")
  ) {
    player.flap();
  }
}

/*
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);
}

function checkPipeCollision(player, pipes) {
  for (const pipe of pipes) {
    let topH = pipe.gapY - pipe.gapH / 2;
    let bottomY = pipe.gapY + pipe.gapH / 2;

    // Horizontal overlap
    if (player.x + player.r > pipe.x && player.x - player.r < pipe.x + pipe.w) {
      // Hit top pipe
      if (player.y - player.r < topH) return true;
      // Hit bottom pipe
      if (player.y + player.r > bottomY) return true;
    }
  }

  return false;
}

function checkFinCollision(player, pipes) {
  for (const pipe of pipes) {
    let topH = pipe.gapY - pipe.gapH / 2;
    let bottomY = pipe.gapY + pipe.gapH / 2;

    // Horizontal overlap
    if (player.x + player.r > pipe.x && player.x - player.r < pipe.x + pipe.w) {
      // Hit top pipe
      if (player.y - player.r < topH) return true;

      // Hit bottom pipe
      if (player.y + player.r > bottomY) return true;
    }
  }

  return false;
}
```

ChatGPT: Ok — this is one of those situations where a few small things combine to make the game look completely broken 🙂
Let's debug it methodically.

There are 3 main problems in the code you posted.

🛑 Problem 1 — You update/check collisions even when not playing

Right now `draw()` always does:

```
player.update();

if (checkFinCollision(...))
if (checkPipeCollision(...))
```

So on the very first frame:

- Pipes spawn overlapping player
- Collision triggers
- `gameState = "lose"`
- `noLoop()` stops draw
- Blob appears to never exist

That’s why it “doesn’t start”.

✅ Fix

Wrap gameplay logic inside state check.

Replace this section of `draw()`:

```
player.update();

if (checkFinCollision(player, world.finish)) {
...
}

if (checkPipeCollision(player, world.pipes)) {
...
}
```

With THIS:

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

  player.update();

  if (checkFinCollision(player, world.pipes)) {
    win++;
    gameState = "win";
    noLoop();
  }
  if (checkPipeCollision(player, world.pipes)) {
    loss++;
    gameState = "lose";
    noLoop();
  }
}
```

🛑 Problem 2 — Wrong variable passed

You wrote:

```
checkFinCollision(player, world.finish)
```

But your world stores:

```
world.pipes
```

So `world.finish` is undefined → collision code breaks silently.

✅ Fix

Change this line:

```
checkFinCollision(player, world.finish)
```

to:

```
checkFinCollision(player, world.pipes)
```

🛑 Problem 3 — Win check identical to lose check

Right now:

```
checkFinCollision == checkPipeCollision
```

They detect the same thing → touching pipe = win + lose.

✅ Replace checkFinCollision entirely

This is the correct Flappy-style win logic:

⭐ Win when passing last pipe

```
function checkFinCollision(player, pipes) {
  if (!pipes.length) return false;

  let lastPipe = pipes[pipes.length - 1];

  return player.x > lastPipe.x + lastPipe.w;
}
```

⭐ Recommended Improvement (Start Screen)

Right now game begins instantly.

If you want ENTER to start:

Change initial state

```
let gameState = "start";
```

Update draw()

Add before player.update:

```
if (gameState === "start") {
  fill(0);
  text("Press ENTER to Start", width/2 - 80, height/2);
}
```

✅ Final Correct Game Loop Structure

Your `draw()` should conceptually be:

```
Draw world

IF playing
   update player
   check win
   check lose

Draw player
Draw HUD
Draw win/lose/start overlays
```

That prevents instant death 🙂

🧭 If it STILL doesn't show

(These are the next two things I'd check)

1️⃣ Make sure createCanvas exists

You currently rely on resizeCanvas()

If nothing appears, add in setup:

```
createCanvas(640, 360);
```

BEFORE loadLevel()

2️⃣ Confirm pipes exist

Quick debug:

```
console.log(world.pipes);
```

Empty array → JSON issue

⭐⭐⭐ Big Picture

You're at the stage where your project now has:

✅ **Game state control**
✅ **Physics object separation**
✅ **World abstraction**
✅ **Restart system**

That's genuinely the architecture used in real indie games — you're past beginner territory here.

If you want — I can do the next polish step that makes this feel professional:

⭐ **Prevent pipes from moving before ENTER**
⭐ **Countdown start**
⭐ **Smooth restart animation**
⭐ **Score counter through gaps**

Just say 🙂