

Process & Decision Documentation

Side Quest Week 2

I added a feature which is a short-timed visual transformation where the blob becomes red and spiky when it hits a platform. The updated behavior improved clarity and offer a more responsive and expressive experience without affecting the core mechanics.

Role-Based Process Evidence

Name: Frank Le

Role(s): Designer / Developer

Primary responsibility for this work: Implementing and refining interactive behavior and visual feedback in a p5.js prototype.

Goal of Work Session

Refine the blob interaction mechanic by improving collision feedback so that hits are more visible and expressive during gameplay.

Tools, Resources, or Inputs Used

- **GenAI:** ChatGPT 5.2 (used for technical guidance on modifying collision logic and visual states)
- **Prior Code:** Existing p5.js platformer prototype (Example 03)
- **Testing:** Live browser testing of interaction behavior and visual feedback
- **Lecture Concepts:** Interaction feedback, iteration, state-based behavior

GenAI Documentation

Date Used: January 25, 2026

Tool Disclosure: ChatGPT 5.2

Purpose of Use: Guiding instructions and technical clarification for modifying a p5.js interactive sketch, including identifying code structure, adding state-based color feedback, implementing timed visual feedback, and extending interaction behavior with a spike-based deformation effect.

Summary of Interaction: ChatGPT was used as a technical assistant to help interpret an existing p5.js sketch, identify where visual properties were controlled, and guide incremental modifications to behavior and visual feedback. Specifically, the tool explained how the blob's color was defined, how to change color based on movement state, how to

detect and persist collision feedback using a timed mechanism (200ms), and how to extend the hit behavior to temporarily morph the blob into a spiky, wobbling form by adjusting noise parameters. The tool provided step-by-step guidance on where to modify the code and why those changes were needed, supporting iterative experimentation and debugging.

Human Decision Point(s):

- I decided that the initial red hit feedback was too brief and chose to extend the visual duration to 200ms, and then 300ms after testing, using a timer rather than a single-frame boolean state.
- I redirected the feedback design from purely color-based signaling to a more expressive form-based response (spiky deformation), prioritizing perceptual clarity and expressive interaction rather than minimal visual change.
- I determined which collision events should trigger the hit behavior (side and underside collisions, but not landing), aligning interaction logic with gameplay intention.

Integrity & Verification Note: All generated guidance was verified by directly implementing and testing changes within the p5.js environment. I validated correctness by observing runtime behavior (collision detection, color persistence timing, and spike deformation behavior) and ensuring the sketch performed as intended without breaking physics or rendering logic. Suggestions were evaluated for appropriateness relative to course concepts such as interaction feedback, state representation, iteration, and user experience clarity. I confirm that the logic matched expected programming patterns and did not introduce unintended side effects.

Scope of GenAI Use: GenAI did not author the original sketch, design the interaction concept, or make final design decisions. I retained my responsibility for selecting which features to implement, tuning parameters, testing outcomes, and integrating changes into their broader assignment context. The tool served only as a technical guide and explanatory assistant rather than an autonomous creator of the project or final submission.

Limitations or Misfires:

- The tool could not visually preview the result and relied on descriptive explanations rather than direct visual validation.
- Some parameter suggestions (e.g., wobble strength and frequency) required manual tuning to match aesthetic expectations.
- Minor iteration was required to ensure the hit effect felt perceptible.

Summary of Process (Human + Tool)

1. I first requested an explanation of what each part of the code did to build understanding of structure and control flow.
2. I then identified a need for visual feedback and implemented color changes based on movement state.
3. After observing that hit feedback was too brief, I revised the logic to use a timed persistence model.
4. I further enhanced the interaction by adding to the color-only feedback with a temporary shape deformation effect (spiky wobble), improving clarity and expressiveness.
5. Each change was tested live in the sketch, evaluated visually, and adjusted as needed.
6. The tool was used repeatedly to clarify where in the code changes should be made and how those changes affected runtime behavior.

Decision Points & Trade-offs

Decision 1: Boolean hit state vs. timed hit feedback

- **Options considered:**
 - Keep a single-frame boolean collision flag.
 - Use a timestamp-based timer to persist feedback.
- **What changed:**

The implementation moved to a 300ms timer using millis().
- **Why:**

Single-frame feedback was barely perceptible to users. A short temporal persistence improves recognition and usability without disrupting gameplay flow.

Decision 2: Color feedback vs. shape deformation feedback

- **Options considered:**
 - Continue using only color change for hit feedback.
 - Add geometric deformation (spiky wobble) to communicate impact.
- **What changed:**

Shape deformation was added during hit states while retaining color change.

- **Why:**

Shape change provides stronger visual communication, aligns with expressive interaction principles, and improves perceptual clarity beyond color alone.

Verification & Judgement

The effectiveness of changes was evaluated through:

- **Playtesting:** Observing responsiveness, readability, and timing during interaction.
- **Visual inspection:** Confirming that spikes appeared only during intended hit events and returned smoothly to normal state.
- **Comparison with course concepts:** Ensuring the design reflected principles of feedback, affordance, iteration, and user-centered interaction.
- **Constraint checking:** Confirming that changes did not break collision logic or introduce unstable motion.

Limitations, Dead Ends, or Open Questions

- Fine-tuning spike sharpness and wobble frequency required subjective judgment and could vary by display or frame rate.
- The system currently treats all hits equally; future refinement could differentiate between light and heavy impacts.
- Visual accessibility considerations (e.g., color-blind friendliness, motion sensitivity) were not fully explored.
- More advanced easing or animation blending could further improve polish.

Appendix

Prompt:

I will provide you with a p5.js, help me identify what is what and what does it do in the code:

```
// Y-position of the floor (ground level)
```

```
let floorY3;
```

```
// Player character (soft, animated blob)
```

```
let blob3 = {
```

```
// Position (centre of the blob)
x: 80,
y: 0,

// Visual properties
r: 26, // Base radius
points: 48, // Number of points used to draw the blob
wobble: 7, // Edge deformation amount
wobbleFreq: 0.9,

// Time values for breathing animation
t: 0,
tSpeed: 0.01,

// Physics: velocity
vx: 0, // Horizontal velocity
vy: 0, // Vertical velocity

// Movement tuning
accel: 0.55, // Horizontal acceleration
maxRun: 4.0, // Maximum horizontal speed
gravity: 0.65, // Downward force
jumpV: -11.0, // Initial jump impulse

// State
onGround: false, // True when standing on a platform
```

```
// Friction
frictionAir: 0.995, // Light friction in air
frictionGround: 0.88, // Stronger friction on ground
};

// List of solid platforms the blob can stand on
// Each platform is an axis-aligned rectangle (AABB)
let platforms = [];

function setup() {
  createCanvas(640, 360);

  // Define the floor height
  floorY3 = height - 36;

  noStroke();
  textAlign(CENTER);
  textSize(14);

  // Create platforms (floor + steps)
  platforms = [
    { x: 0, y: floorY3, w: width, h: height - floorY3 }, // floor
    { x: 120, y: floorY3 - 70, w: 120, h: 12 }, // low step
    { x: 300, y: floorY3 - 120, w: 90, h: 12 }, // mid step
    { x: 440, y: floorY3 - 180, w: 130, h: 12 } // high step
  ];
}
```

```

{ x: 520, y: floorY3 - 70, w: 90, h: 12 }, // return ramp
];

// Start the blob resting on the floor
blob3.y = floorY3 - blob3.r - 1;

}

function draw() {
background(240);

// --- Draw all platforms ---
fill(200);
for (const p of platforms) {
rect(p.x, p.y, p.w, p.h);
}

// --- Input: left/right movement ---
let move = 0;
if (keyIsDown(65) || keyIsDown(LEFT_ARROW)) move -= 1; // A or ←
if (keyIsDown(68) || keyIsDown(RIGHT_ARROW)) move += 1; // D or →
blob3.vx += blob3.accel * move;

// --- Apply friction and clamp speed ---
blob3.vx *= blob3.onGround ? blob3.frictionGround : blob3.frictionAir;
blob3.vx = constrain(blob3.vx, -blob3.maxRun, blob3.maxRun);
}

```

```
// --- Apply gravity ---
blob3.vy += blob3.gravity;

// --- Collision representation ---
// We collide using a rectangle (AABB),
// even though the blob is drawn as a circle
let box = {
    x: blob3.x - blob3.r,
    y: blob3.y - blob3.r,
    w: blob3.r * 2,
    h: blob3.r * 2,
};

// --- STEP 1: Move horizontally, then resolve X collisions ---
box.x += blob3.vx;
for (const s of platforms) {
    if (overlap(box, s)) {
        if (blob3.vx > 0) {
            // Moving right → hit the left side of a platform
            box.x = s.x - box.w;
        } else if (blob3.vx < 0) {
            // Moving left → hit the right side of a platform
            box.x = s.x + s.w;
        }
        blob3.vx = 0;
    }
}
```

```
}
```

```
// --- STEP 2: Move vertically, then resolve Y collisions ---
```

```
box.y += blob3.vy;
```

```
blob3.onGround = false;
```

```
for (const s of platforms) {
```

```
    if (overlap(box, s)) {
```

```
        if (blob3.vy > 0) {
```

```
            // Falling → land on top of a platform
```

```
            box.y = s.y - box.h;
```

```
            blob3.vy = 0;
```

```
            blob3.onGround = true;
```

```
        } else if (blob3.vy < 0) {
```

```
            // Rising → hit the underside of a platform
```

```
            box.y = s.y + s.h;
```

```
            blob3.vy = 0;
```

```
        }
```

```
}
```

```
}
```

```
// --- Convert collision box back to blob centre ---
```

```
blob3.x = box.x + box.w / 2;
```

```
blob3.y = box.y + box.h / 2;
```

```
// Keep blob inside the canvas horizontally
```

```
blob3.x = constrain(blob3.x, blob3.r, width - blob3.r);

// --- Draw the animated blob ---

blob3.t += blob3.tSpeed;

drawBlobCircle(blob3);

// --- HUD ---

fill(0);

text("Move: A/D or ←/→ • Jump: Space/W/↑ • Land on platforms", 10, 18);

}

// Axis-Aligned Bounding Box (AABB) overlap test

// Returns true if rectangles a and b intersect

function 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

);

}

// Draws the blob using Perlin noise for a soft, breathing effect

function drawBlobCircle(b) {

fill(20, 120, 255);

beginShape();

for (let i = 0; i < b.points; i++) {

const a = (i / b.points) * TAU;
```

```

// Noise-based radius offset

const n = noise(
    cos(a) * b.wobbleFreq + 100,
    sin(a) * b.wobbleFreq + 100,
    b.t,
);

const r = b.r + map(n, 0, 1, -b.wobble, b.wobble);

vertex(b.x + cos(a) * r, b.y + sin(a) * r);

}

endShape(CLOSE);

}

// Jump input (only allowed when grounded)

function keyPressed() {
    if (
        (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) &&
        blob3.onGround
    ){
        blob3.vy = blob3.jumpV;
        blob3.onGround = false;
    }
}

```

Result:

Absolutely — here's a “what is what / what it does” walkthrough of your sketch, grouped by purpose, and mapped to the exact parts of the code.

1) Global variables (stuff the whole sketch can access)

let floorY3;

- Stores the **Y position of the floor** (ground level).
- You set it in `setup()` as `height - 36`, meaning the floor is a rectangle at the bottom 36px of the canvas.

let blob3 = { ... }

This is your **player character**, stored as one object (so all player-related values live together).

Position

- `x, y: center` of the blob on the canvas.

Visual (how it looks)

- `r: base radius (size)` of the blob.
- `points`: how many vertices are used to draw the blob’s outline (more points = smoother).
- `wobble`: how “blobby” the edges deform.
- `wobbleFreq`: how quickly the noise pattern changes around the perimeter (spatial frequency).

Animation timing

- `t: time value` used for the breathing/wobble animation.
- `tSpeed`: how fast `t` increases each frame (higher = faster breathing).

Physics (movement)

- `vx, vy: horizontal/vertical velocity`.

Movement tuning (feel)

- accel: how quickly you speed up when holding left/right.
- maxRun: cap on horizontal speed.
- gravity: how fast you accelerate downward each frame.
- jumpV: jump impulse (negative because up is smaller Y in p5).

State

- onGround: whether the blob is standing on a platform (used to allow jumping + apply ground friction).

Friction

- frictionAir: mild slowing while in air.
 - frictionGround: stronger slowing while on ground.
-

2) Platforms / level geometry

let platforms = [];

- An array that will contain **rectangles** the player can collide with / stand on.
- Each platform is an **AABB**: Axis-Aligned Bounding Box, meaning rectangles not rotated.

In setup() you fill it like this:

{ x, y, w, h }

Example:

- { x: 0, y: floorY3, w: width, h: height - floorY3 }
This is the **floor**, a wide rectangle at the bottom.

The others are “steps” at different heights.

3) setup() (runs once at the start)

```
function setup() {
  createCanvas(640, 360);
  floorY3 = height - 36;
```

```
...
platforms = [ ... ];
blob3.y = floorY3 - blob3.r - 1;
}
```

What it does:

- Creates the canvas.
- Calculates floor height.
- Sets font settings.
- Builds the platform list.
- Places the blob so it starts **resting on the floor**:
 - floorY3 - blob3.r puts the blob's *bottom* at the floor.
 - the -1 prevents initial "stuck in the floor" overlap.

4) draw() (runs every frame, game loop)

A) Clear background

```
background(240);
```

Resets screen each frame.

B) Draw platforms

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

Just visuals — collision is handled later.

C) Read input (left/right)

```
let move = 0;

if (keyIsDown(65) || keyIsDown(LEFT_ARROW)) move -= 1;
if (keyIsDown(68) || keyIsDown(RIGHT_ARROW)) move += 1;

blob3.vx += blob3.accel * move;
```

- move becomes -1, 0, or 1

- Pressing A/← accelerates left, D/→ accelerates right
- This changes **velocity**, not position directly (more “physics-y” feel)

D) Apply friction + cap speed

```
blob3.vx *= blob3.onGround ? blob3.frictionGround : blob3.frictionAir;
```

```
blob3.vx = constrain(blob3.vx, -blob3.maxRun, blob3.maxRun);
```

- If grounded, you slow down faster.
- constrain prevents infinite speed.

E) Gravity

```
blob3.vy += blob3.gravity;
```

Each frame you gain downward velocity.

5) Collision system (the most important part)

Why a “box”?

Even though the blob is drawn as a circle, collisions use a **rectangle** for simplicity:

```
let box = {
  x: blob3.x - blob3.r,
  y: blob3.y - blob3.r,
  w: blob3.r * 2,
  h: blob3.r * 2,
};
```

This box is the blob’s “hitbox”.

STEP 1: Move X, then resolve X collisions

```
box.x += blob3.vx;
for (const s of platforms) {
  if (overlap(box, s)) {
```

```

if (blob3.vx > 0) box.x = s.x - box.w;

else if (blob3.vx < 0) box.x = s.x + s.w;

blob3.vx = 0;

}

}

```

What's happening:

- You apply horizontal movement first.
- If the moved hitbox overlaps a platform:
 - moving right → snap the box to the **left side** of the platform
 - moving left → snap to the **right side**
- Then set vx = 0 so you stop pushing into it.

This prevents “tunneling” through walls (at least at these speeds).

STEP 2: Move Y, then resolve Y collisions

```

box.y += blob3.vy;

blob3.onGround = false;

for (const s of platforms) {

  if (overlap(box, s)) {

    if (blob3.vy > 0) {

      box.y = s.y - box.h;

      blob3.vy = 0;

      blob3.onGround = true;

    } else if (blob3.vy < 0) {

      box.y = s.y + s.h;

      blob3.vy = 0;
    }
  }
}

```

```
    }  
}  
}
```

Key ideas:

- You apply vertical movement.
- Reset onGround to false, then only set it true if you land.
- If falling ($vy > 0$):
 - snap box on top of platform
 - zero vertical speed
 - mark grounded
- If jumping upward ($vy < 0$):
 - snap box to underside
 - zero vertical speed (head bump)

Convert box back to blob center

```
blob3.x = box.x + box.w / 2;
```

```
blob3.y = box.y + box.h / 2;
```

Because box stores top-left, but blob3 stores center.

Keep inside canvas horizontally

```
blob3.x = constrain(blob3.x, blob3.r, width - blob3.r);
```

Stops the player leaving the screen left/right.

6) Drawing the blob (soft breathing shape)

Update time

```
blob3.t += blob3.tSpeed;
```

This drives the noise animation.

drawBlobCircle(blob3)

This draws a **wobbly circle** using Perlin noise:

- It loops points times.
- For each point it calculates an angle α around the circle.
- Uses noise(...) to get a smooth random value.
- Converts that noise into a small radius offset (between -wobble and +wobble).
- Places a vertex at that radius.

This is why it looks like a living blob instead of a perfect circle.

7) HUD (instructions text)

```
text("Move: A/D ...", 10, 18);
```

Just info on screen.

8) Collision helper

overlap(a, b)

A standard AABB intersection test:

```
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  
)
```

Returns true if rectangles overlap at all.

9) Jump input

keyPressed()

Runs once when a key is pressed (not held):

```
if ((space/W/up) && blob3.onGround) {  
    blob3.vy = blob3.jumpV;  
    blob3.onGround = false;  
}
```

Important:

- Jump only works if onGround is true (prevents double jump).
 - Sets upward velocity instantly (jumpV is negative).
-

Mental model of the whole game loop

Every frame:

1. Read input → change vx
 2. Apply friction + gravity → update vx, vy
 3. Build hitbox
 4. Move X, fix X collisions
 5. Move Y, fix Y collisions + set onGround
 6. Update blob center from hitbox
 7. Animate + draw blob
-

If you want, paste your next version (or tell me what you want to change: double jump, slopes, camera, better circle collisions, coyote time, etc.) and I'll point to the exact places you'd edit.

Prompt:

Identify where the color of the blob can be edit in the code

Result:

The blob's color is defined inside the **drawBlobCircle()** function.

Look for this line:

```
fill(20, 120, 255);

💡 Location in your code:

function drawBlobCircle(b) {
    fill(20, 120, 255); // ← THIS sets the blob color
    beginShape();
    ...
}
```

How to change the color

That `fill(r, g, b)` uses RGB values (0–255).

Examples

Red blob

```
fill(255, 60, 60);
```

Green blob

```
fill(80, 220, 120);
```

Purple blob

```
fill(170, 80, 255);
```

Semi-transparent blue

```
fill(20, 120, 255, 180); // RGBA (last value = opacity)
```

Optional (if you want more control later)

If you'd like to control the color from the `blob3` object (cleaner for design tweaks), you can add a `color` property:

```
// inside blob3
color: [20, 120, 255],
```

Then change the `fill` line to:

```
fill(...b.color);
```

Now you can edit the blob's color directly inside the blob object.

If you'd like, I can also show you how to:

- Change color based on movement
- Animate color over time
- Change color when jumping / landing
- Make gradient or glow effects

Very useful for polish and interaction design ✨

Prompt:

Say that when the blob moves, the color is green. How can I modify it?

Result:

Great question — this is a classic **state-based visual feedback** pattern (motion → color change). Perfect for making interaction feel alive and readable.

You want:

- When the blob is moving → it turns green
 - When it's not moving → it stays its normal color
-

Option 1 — Simple & clean (recommended)

We detect movement using the blob's velocity and change the fill() dynamically.

 **Step 1: Find this function**

In your code:

```
function drawBlobCircle(b) {  
  fill(20, 120, 255); // ← current fixed color  
  beginShape();  
  ...  
}
```

Step 2: Replace the fill() line with this

```
// If blob is moving horizontally, make it green  
if (abs(b.vx) > 0.2) {  
    fill(80, 220, 120); // green  
} else {  
    fill(20, 120, 255); // blue (idle)  
}
```

Your function becomes:

```
function drawBlobCircle(b) {  
  
    if (abs(b.vx) > 0.2) {  
        fill(80, 220, 120); // moving = green  
    } else {  
        fill(20, 120, 255); // idle = blue  
    }  
  
    beginShape();  
    ...  
}
```

Why this works

- `b.vx` = horizontal velocity
 - If it's close to 0, the blob is basically standing still.
 - `abs()` handles both left and right movement.
 - 0.2 is a small threshold to avoid flickering due to tiny floating-point drift.
-

Optional upgrades (if you want polish)

1. Also turn green when jumping / falling

Change the condition to:

```
if (abs(b.vx) > 0.2 || abs(b.vy) > 0.2) {
```

2. Smooth color transition instead of instant change

If you want a softer animation effect (more premium feel), I can show you how to interpolate colors over time.

3. Different colors for different states

Example:

-  Idle = blue
-  Moving = green
-  Jumping = red

Tell me if you'd like this — it's great for UX readability and game feel.

Given your design background, this kind of micro-feedback is exactly the kind of polish that elevates interaction design 

Prompt:

I want to add another state when the blob hits, not land on, the platform, it changes its color to red. Guide me to where I can modify in the code.

Result:

Perfect — this is a great example of **state-driven feedback** (collision → color cue). You already have exactly the right collision logic in place — we just need to tap into it.

Right now your code already *detects* when the blob hits the **side or underside** of a platform — it just doesn't store that information.

We'll add a small “**hit state flag**” and trigger it inside the collision code.

I'll show you exactly **where to modify and why**.

Goal

Color states:

-  Blue → idle
 -  Green → moving
 -  Red → hits platform (side or underside, NOT landing)
-

Step 1 — Add a new state variable to the blob

 Find your blob3 object at the top of the sketch.

Add this line inside it:

```
hitPlatform: false, // True when bumping into a platform
```

Example:

```
let blob3 = {  
    ...  
    onGround: false,  
    hitPlatform: false, // ← ADD THIS  
    frictionAir: 0.995,  
    frictionGround: 0.88,  
};
```

This gives the blob a memory of whether it hit something this frame.

Step 2 — Reset the hit flag every frame

 Find this line in draw():

```
blob3.onGround = false;
```

Right after it, add:

```
blob3.hitPlatform = false;
```

So it becomes:

```
blob3.onGround = false;  
  
blob3.hitPlatform = false; // ← reset each frame
```

This ensures red only flashes when an actual hit happens.

Step 3 — Detect side collisions (horizontal hits)

 Find this block in draw():

```
// --- STEP 1: Move horizontally, then resolve X collisions ---
```

```
box.x += blob3.vx;
```

```
for (const s of platforms) {
```

```
    if (overlap(box, s)) {
```

```
        if (blob3.vx > 0) {
```

```
            box.x = s.x - box.w;
```

```
        } else if (blob3.vx < 0) {
```

```
            box.x = s.x + s.w;
```

```
        }
```

```
        blob3.vx = 0;
```

```
}
```

```
}
```

 Modify it by adding this line inside the collision block:

```
blob3.hitPlatform = true;
```

Result:

```
if (overlap(box, s)) {  
    if (blob3.vx > 0) {  
        box.x = s.x - box.w;  
    } else if (blob3.vx < 0) {
```

```
    box.x = s.x + s.w;  
}  
  
blob3.vx = 0;  
  
blob3.hitPlatform = true; // ← side hit detected  
}
```

This triggers when you bump into walls or platform edges.

Step 4 — Detect underside hits (head bumps)

 Find this vertical collision block:

```
if (overlap(box, s)) {  
  
    if (blob3.vy > 0) {  
  
        // landing  
  
        ...  
  
    } else if (blob3.vy < 0) {  
  
        // Rising → hit underside  
  
        box.y = s.y + s.h;  
  
        blob3.vy = 0;  
  
    }  
  
}
```

 Add hit detection only in the **underside case**:

```
else if (blob3.vy < 0) {  
  
    box.y = s.y + s.h;  
  
    blob3.vy = 0;  
  
    blob3.hitPlatform = true; // ← head bump hit  
  
}
```

 We intentionally do NOT mark hits when landing.

Step 5 — Change color logic in drawBlobCircle()

 Find:

```
fill(20, 120, 255);
```

Replace the color logic with:

```
if (b.hitPlatform) {  
    fill(255, 80, 80);      //  hit  
}  
else if (abs(b.vx) > 0.2) {  
    fill(80, 220, 120);      //  moving  
}  
else {  
    fill(20, 120, 255);      //  idle  
}
```

Final behavior

Now the blob:

-  Turns red when hitting walls or bumping its head
-  Turns green when moving
-  Turns blue when idle / standing

The red will flash briefly on impact — which feels responsive and readable.

Optional polish (if you want next)

If you'd like, I can also help you:

-  Add a short “red flash timer” so the hit color lingers for 100–200ms
-  Add squash/stretch when hitting
-  Add screen shake or particle feedback
-  Add sound triggers tied to collisions

These are exactly the kinds of interaction details that elevate game feel and UX — very aligned with your design instincts.

Prompt:

The "red flash" happens too briefly. Let's make the color appearance for 200ms.

Result:

Nice — this is exactly the right instinct. A **200ms hit-flash** makes the feedback perceptible without feeling sluggish. 🎉

We'll convert your boolean hitPlatform into a **short timer**.

Right now it only stays true for one frame (~16ms). We'll instead store a **timestamp** and keep the blob red for 200ms after a hit.

 **Goal**

When the blob hits a platform:

- turn red
 - stay red for **200 milliseconds**
 - then return to normal logic (green / blue)
-

 **Step 1 — Replace the boolean with a timer**

 Find this line inside blob3:

hitPlatform: false,

 Replace it with:

hitTime: 0, // Timestamp of last platform hit (ms)

Your blob now stores *when* the hit happened instead of just true/false.

 **Step 2 — Remove the reset line (important)**

Earlier you added:

blob3.hitPlatform = false;

 Find this line in draw() and **delete it completely**.

We no longer reset manually — time will handle it.

Step 3 — Trigger the hit timer when collision happens

Horizontal collision (side hit)

 Find this line inside the X collision block:

```
blob3.hitPlatform = true;
```

 Replace it with:

```
blob3.hitTime = millis();
```

Vertical collision (underside hit)

 Find the underside collision code and replace:

```
blob3.hitPlatform = true;
```

 With:

```
blob3.hitTime = millis();
```

Now every hit records the current time in milliseconds.

Step 4 — Update color logic to use the timer

 Go to drawBlobCircle(b) and replace your color logic with this:

```
const hitDuration = 200; // milliseconds
```

```
const isHit = millis() - b.hitTime < hitDuration;
```

```
if (isHit) {  
    fill(255, 80, 80);      //  hit flash  
}  
else if (abs(b.vx) > 0.2) {  
    fill(80, 220, 120);    //  moving  
}  
else {
```

```
fill(20, 120, 255); // ⚪ idle  
}
```

🧪 What this does

- millis() = time since sketch started
- When a hit occurs → hitTime = millis()
- Each frame:
 - If current time minus hitTime < 200ms → still red
 - Otherwise → revert to normal state logic

This gives you a clean, consistent **200ms flash**, regardless of frame rate.

✨ Optional polish (if you want)

Since you're design-driven, here are some nice upgrades you might enjoy:

🌈 Fade-out instead of hard switch

Make the red gradually fade back to normal.

💥 Stronger hit feedback

Add a tiny squash or shake during hit flash.

🎨 Blend color instead of replacing it

Mix red into the base color instead of snapping.

If you want any of these, I can wire them in cleanly.

Prompt:

Update the hitting behavior with the following idea: When hitting, the blob transforms to wobbling spikes. Guide me through the code to add this feature.

Result:

Yep — you can do this by **reusing the same “hit timer” idea**, but instead of only changing fill(), you temporarily change the **shape parameters** used in drawBlobCircle() so the outline becomes *spiky* for ~200ms.

Below is exactly **where** to edit, and **what** to add.

1) Add “hit spike” settings to the blob object

💡 At the top in let blob3 = { ... }, add these properties (near the visual section is best):

```
// Hit effect timing
```

```
hitTime: -9999, // last hit timestamp (ms)
```

```
// Normal look (store defaults so we can return cleanly)
```

```
baseWobble: 7,
```

```
baseWobbleFreq: 0.9,
```

```
// Spike look (hit state)
```

```
spikeWobble: 18, // bigger deformation = spikes
```

```
spikeWobbleFreq: 2.2 // higher frequency = more jagged
```

Also, since you already have wobble and wobbleFreq, you’ll want to make sure your “normal” values match them. Easiest is:

- keep wobble: 7 and wobbleFreq: 0.9 as-is
 - set baseWobble and baseWobbleFreq to the same numbers
-

2) Trigger the hit timer on *hits* (side + underside only)

You already have collision blocks where you previously set the red state. Replace/insert the timestamp assignment.

A) Horizontal hits (STEP 1 block)

💡 Find this block in draw():

```
// --- STEP 1: Move horizontally, then resolve X collisions ---
```

```
box.x += blob3.vx;
```

```
for (const s of platforms) {
```

```
if (overlap(box, s)) {  
    ...  
    blob3.vx = 0;  
}  
}  
}
```

 Inside the if (overlap(box, s)) { ... } block, add:

```
blob3.hitTime = millis();
```

Put it right after blob3.vx = 0; (or before—either works).

B) Underside hits (STEP 2 block, rising)

 Find the vertical collision section:

```
else if (blob3.vy < 0) {  
  
    box.y = s.y + s.h;  
  
    blob3.vy = 0;  
}
```

 Add:

```
blob3.hitTime = millis();
```

So it becomes:

```
else if (blob3.vy < 0) {  
  
    box.y = s.y + s.h;  
  
    blob3.vy = 0;  
  
    blob3.hitTime = millis();  
}
```

 Do **not** set hit time on landing (blob3.vy > 0 case), since you said “hits, not land on”.

3) Make the blob “spiky” for 200ms inside drawBlobCircle()

This is the main visual change.

💡 Go to:

```
function drawBlobCircle(b) {  
    fill(20, 120, 255);  
    beginShape();  
    ...  
}
```

A) Add a hit window + choose current shape parameters

At the top of the function (right after {}), add:

```
const hitDuration = 200;  
const isHit = millis() - b.hitTime < hitDuration;
```

```
// Choose wobble settings depending on state  
const wobbleNow = isHit ? b.spikeWobble : b.baseWobble;  
const freqNow = isHit ? b.spikeWobbleFreq : b.baseWobbleFreq;
```

B) Use those values in the noise radius math

💡 Find this part:

```
const n = noise(  
    cos(a) * b.wobbleFreq + 100,  
    sin(a) * b.wobbleFreq + 100,  
    b.t,  
)
```

```
const r = b.r + map(n, 0, 1, -b.wobble, b.wobble);
```

- Change b.wobbleFreq → freqNow
- Change b.wobble → wobbleNow

So it becomes:

```
const n = noise(  
    cos(a) * freqNow + 100,  
    sin(a) * freqNow + 100,  
    b.t,  
)  
  
const r = b.r + map(n, 0, 1, -wobbleNow, wobbleNow);
```

That alone will make the blob outline “spikier” during the hit window.

4) (Optional but looks way better) Make spikes “sharper” using a power curve

Right now the deformation is smooth both ways. To make spikes feel *spiky*, bias the noise so you get more extreme peaks.

Right after `const n = noise(...)`, add:

```
// Sharpen spikes during hit (push values away from 0.5)  
  
let ns = n;  
  
if (isHit) {  
  
    ns = pow(n, 0.35); // smaller exponent = sharper peaks  
  
}
```

Then use `ns` instead of `n` in the radius:

```
const r = b.r + map(ns, 0, 1, -wobbleNow, wobbleNow);
```

This gives you “teeth” rather than just a rough edge.

5) Keeping your color states (if you still want them)

If you still want red on hit *and* spikes, keep your existing color logic, but make hit override:

```
if (isHit) fill(255, 80, 80);
```

```
else if (abs(b.vx) > 0.2) fill(80, 220, 120);  
else fill(20, 120, 255);
```

This pairs perfectly: **hit = red + spikes.**

Summary of “where to touch”

1. **blob3 object** → add hitTime, plus base/spike wobble settings
 2. **draw() collision blocks** → on side hit and underside hit: blob3.hitTime = millis();
 3. **drawBlobCircle()** → compute isHit, swap wobble/freq to spiky ones, optionally sharpen spikes
-

If you paste your updated drawBlobCircle() after you try it, I can help you tune the spike feel (more “porcupine” vs more “electric shock”) without breaking the physics/collisions.