

Entry Header

Name: Kiki Tan

Role(s): Individual developer / designer

Primary responsibility for this work:

Designing and implementing a small interactive p5.js experience using game states, branching logic, and basic UI interaction.

Goal of Work Session

The goal of this side quest was to extend the provided p5.js starter project into a small interactive horror narrative that unfolds across multiple game states, includes branching choices, and responds to player decisions over time.

Tools, Resources, or Inputs Used

- p5.js (starter project code provided for Week 3)
 - VS Code for editing and debugging
 - GitHub for version control and commits
 - Lecture materials on game states and interaction
 - GenAI tool (ChatGPT 5.2) for conceptual planning and debugging support
-

GenAI Documentation

Date Used: February 1st 2026

Tool Disclosure: ChatGPT (model 5.2)

Purpose of Use:

Used to understand and extend the existing state-based structure of the starter code, plan a branching narrative structure, and debug issues related to screen routing and shared variables.

Summary of Interaction:

The tool helped explain how the existing `main.js` router worked, suggested ways to structure new scene files consistently, and assisted with debugging issues such as mismatched screen names and button interactions. It was also used to reflect on design decisions such as when to stop adding features.

Human Decision Point(s):

I decided which narrative branches to keep, how sanity should affect endings, and when to stop adding new features. I also removed or rewrote parts of suggested code to better fit the scope and tone of the assignment.

Integrity & Verification Note:

All suggested code was reviewed, tested, and modified manually. Bugs were identified through runtime testing in the browser and fixed by tracing logic in my own files rather than copying output blindly.

Scope of GenAI Use:

GenAI did not write the entire project. Narrative content, final structure, and all commits were controlled and verified by me.

Limitations or Misfires:

The tool sometimes suggested steps that did not align with my final implementation or over-extended the scope, which I chose not to follow.

Summary of Process (Human + Tool)

I began by reviewing the starter project to understand how screens were routed using a shared state variable. I then incrementally added new scene files, tested transitions, and debugged issues as they arose. After confirming that branching worked, I introduced a simple sanity variable to track player state and gate endings. Throughout the process, I tested changes frequently and adjusted scope to keep the project manageable.

Decision Points & Trade-offs

One key decision was to keep the sanity system minimal rather than turning it into a complex mechanic. This allowed the narrative consequences to remain subtle and reduced the risk of technical bugs. I also chose to stop after implementing multiple endings rather than adding additional polish features due to time and scope constraints.

Verification & Judgement

I verified decisions by running the project in the browser after each change, checking console errors, and ensuring that scene transitions behaved as expected. I also revisited the assignment criteria to confirm that the required interaction and state concepts were demonstrated.

Limitations, Dead Ends, or Open Questions

Some UI elements remain intentionally simple, and the narrative could be expanded further with more branches or audiovisual effects. These were left unexplored to keep the side quest focused and within scope.

Appendix

A full transcript of my interaction with the GenAI tool is included in the appendix for transparency and verification. The transcript reflects planning, debugging, and decision-making throughout the development process.

User: Can you read a zip file?

AI: Yes. If you upload a zip file, I can inspect its contents and walk through the structure with you. That's usually a good first step before making changes, especially when the project is meant to be extended rather than rebuilt from scratch.

User: Okay, I'm going to send the starter zip file provided for Week 3.

User uploaded a zip file containing the Week 3 starter project.

AI: Got it. I can see the project structure now. Before talking about changes, it's useful to understand what the starter project already does and how responsibility is distributed across files.

AI: The project is already organized into multiple JavaScript files rather than a single monolithic sketch. There is an `index.html` file that loads all scripts, a `main.js` file that appears to coordinate screen changes, and separate files such as `start.js`, `instructions.js`, and several ending-related scripts.

AI: Importantly, the project already uses a shared variable called `currentScreen`. This indicates that the sketch is built around a screen-based or state-based architecture. Instead of one continuous gameplay loop, the program switches between discrete screens based on the value of this variable.

AI: Each screen defines its own drawing logic and, in some cases, its own input handlers. The role of `main.js` is to check the current value of `currentScreen` and decide which draw function and input function should run. Even though the starter project does not explicitly describe this as a "router," that is effectively the role it is already playing.

User: I want to modify this project into an interactive story that unfolds across multiple game states and files, like a small branching decision tree. There's also a bonus for tracking a player stat across scenes, but I don't want to jump into that immediately.

AI: That goal aligns very well with the existing structure. A branching narrative maps naturally onto a state-based system, since each choice can simply transition the project into a different screen. Because the infrastructure for switching screens is already in place, you can focus on narrative flow rather than reinventing state management.

AI: It's also reasonable to delay the player stat for now. From a development perspective, stabilizing navigation and scene switching first reduces complexity. Once you are confident that scenes transition correctly, adding a shared stat becomes a much more contained change.

AI: In terms of reuse, most of the screen-switching logic in the starter project is safe to keep. What you are really changing is not *how* the project switches states, but *what those states represent*. The original gameplay-oriented screens can be replaced or ignored, while the underlying routing logic continues to function as scaffolding.

User: I want to reuse as much of the existing structure as possible rather than rewriting everything.

AI: That's a sensible decision for this assignment. Reuse does not reduce the legitimacy of your work here; instead, it demonstrates that you understand the starter project and are able to adapt it to a different design goal. Transforming a gameplay demo into a branching horror narrative is a meaningful conceptual change, even if some technical infrastructure remains the same.

AI: From a process documentation perspective, this also gives you a clear story to tell later: you began by analyzing the provided structure, identified which parts could support a narrative experience, and then extended the project incrementally rather than replacing it wholesale.

AI: At this stage, the main takeaway is that the starter project already behaves like a routing system. Treating `main.js` as a central coordinator and pushing narrative logic into separate scene files is a natural extension of what is already there.

AI: When you're ready to move forward, the next step would be to make a very small, low-risk structural change—such as introducing a single narrative scene—and verify that routing behaves exactly as expected before introducing branching or player state.

User:

Before I touch any code, I want to confirm my mental model: `main.js` should act purely as a router, and each scene should live in its own file.

AI: Yep — that mental model is exactly what you want here 🚫

Locking this in early will save you a lot of confusion later, especially once branching and endings come into play.

Here's the clean way to think about it:

- `main.js` = traffic controller
 - It **does not** contain:
 - story text
 - button labels

- narrative logic
- It **only** decides:
 - what to draw (`draw()`)
 - where mouse clicks go (`mousePressed()`)
 - where key presses go (`keyPressed()`)
- **Each scene file = a self-contained screen**
 - `scene1.js` → `drawScene1()` + `scene1MousePressed()`
 - `scene2A.js` → `drawScene2A()` + `scene2AMousePressed()`
 - etc.

This separation matters because it keeps responsibilities clear:

- routing logic stays centralized
- narrative logic stays local to each scene
- bugs become much easier to isolate




User: I also want to keep the starter style, like the existing if/else chain.
I don't want this to look like a full architectural rewrite.

AI: That's actually the best call for this assignment.

You *could* implement routing with:

- a switch
- or a scene map like `{ scene1: drawScene1, ... }`

...but sticking with the **existing if/else chain** has real advantages here:

-  consistent with the starter project
-  immediately readable to a TA
-  looks like an extension, not a redesign

From a process perspective, this reads as:

“I understood the provided structure and built on top of it.”

Which is exactly what they want to see.

User: So to sanity-check:

- `main.js` handles routing only
- scene files handle story + buttons

- shared helpers can live in `main.js`

That's the boundary I should stick to.

AI: Yes — that boundary is clean and defensible.

More specifically:

What *belongs* in `main.js`

- `let currentScreen = "start";`
- routing logic in:
 - `draw()`
 - `mousePressed()`
 - `keyPressed()`
- shared helpers that are genuinely global:
 - `isHover({ x, y, w, h })`
 - (later) player state, HUD drawing, glitch helpers

What *should not be* in `main.js`

- story text
- choice wording
- scene-specific visuals
- narrative conditions like “if the player chose X earlier”

Keeping this line clear will make later expansion feel natural instead of fragile.

User: One constraint from me: I want horror + branching, but **I don't want to add sanity or glitch yet.**

I want the smallest possible proof that routing works first.

AI: That's a very realistic development instinct, and it's the right order.

At this stage, the goal is *not* completeness — it's verification.

The best “proof” to aim for is:

- **Start → Scene 1 renders correctly**
- no branching yet
- no stats yet
- no polish yet

Once that works, everything else becomes additive rather than risky.

User: Okay. For the next step, I want Start to route to a minimal `scene1`. Text only, maybe a temporary “Return to Start” button if needed.

AI: Perfect. That sets us up cleanly for implementation.


In next part, we’ll:

- create a minimal `scene1.js`
- add “`scene1`” cases to the router in `main.js`
- update the Start button to set `currentScreen = "scene1"`
- double-check script order in `index.html`

That gives you a solid, testable foundation before branching.

User: Part 2.

I want to implement the smallest possible version first, just to verify that routing actually works before I branch anything.

AI: Got it  We’ll implement the *smallest working proof* that your router approach is solid: **Start** → **scene1 renders** (and optionally a temporary “Return to Start” button). The key here is keeping this minimal so that if something breaks, we know it’s routing or script order—not story logic.

Here’s the plan (in the same order you should apply changes):

- **1) Create `scene1.js` with:**
 - `drawScene1()` (visuals/text)
 - `scene1MousePressed()` (handles a temporary button click)
- **2) Update `main.js` router to recognize “`scene1`” in:**
 - `draw()`
 - `mousePressed()`
- **3) Update the Start screen handler** so Start sets:
 - `currentScreen = "scene1"`
- **4) Confirm `index.html` script order**
 - `scene1.js` must load **before** `main.js`

User: I’m intentionally keeping Scene 1 very barebones here. If something breaks, I want it to be obvious that it’s a routing issue, not content.

AI: First, create a new file named `scene1.js`. Keep it small but testable. Here’s a minimal version with a temporary “Return to Start” button (useful for verifying bidirectional routing):


```
// scenel.js - minimal routing proof
function drawScenel() {
  background(20, 20, 24);

  fill(230);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Scene 1", width / 2, 200);

  textSize(16);
  fill(200);
  text(
    "You wake up in a dark room.\nThe light above you flickers.",
    width / 2,
    280
  );

  const backBtn = { x: width / 2, y: 420, w: 260, h: 60 };

  rectMode(CENTER);
  noStroke();
  fill(isHover(backBtn) ? 200 : 160);
  rect(backBtn.x, backBtn.y, backBtn.w, backBtn.h, 10);

  fill(30);
  textSize(18);
  text("Return to Start", backBtn.x, backBtn.y);

  cursor(isHover(backBtn) ? HAND : ARROW);
}

function scenelMousePressed() {
  const backBtn = { x: width / 2, y: 420, w: 260, h: 60 };
  if (isHover(backBtn)) {
    currentScreen = "start";
  }
}
```

User: Added `scenel.js`.

Next I'm updating the router so it actually knows this screen exists.

AI: Update `main.js` so it routes to "scenel".

In `draw()`:

```
} else if (currentScreen === "scenel") {
  drawScenel();
}
```

And in `mousePressed()`:

```
} else if (currentScreen === "scenel") {
  scenelMousePressed();
}
```

```
}
```

At this stage, don't touch `keyPressed()` yet—you don't need it for this proof.

User: I'm also checking the Start screen now, because I know that's an easy place to miss something.

AI: In `startMousePressed()`, wherever it previously transitioned to the old gameplay state, set:

```
currentScreen = "scene1";
```

If this string doesn't match exactly, the router won't fire.

User: Before testing, I'm double-checking script order so this doesn't silently fail.

AI: Good instinct. In `index.html`, make sure `scene1.js` is loaded *before* `main.js`:

```
<script src="start.js"></script>
<script src="instructions.js"></script>

<script src="scene1.js"></script>

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

Also confirm that no file is included twice, especially `main.js`.

User: I pushed the commit and tested it.
Clicking Start still doesn't do anything.

AI: Perfect — this is a very typical first routing bug. Let's debug it step by step 🔍

The three most common causes here are:


1. string mismatch
2. script order
3. duplicate scripts

Let's start with the fastest check: what exact value are you assigning to `currentScreen` when Start is clicked?

User: It's still:

```
currentScreen = "game";
```

I didn't update that line yet.

AI: That explains it 

Your router is listening for "scene1", but Start is sending "game".

Change it to:

```
currentScreen = "scene1";
```

Then refresh (hard refresh if needed).

User: That fixed it.

Start now routes to Scene 1 correctly.

AI: Nice — that confirms the router, script order, and scene isolation are all working as expected.

From here on, any new behavior you add is building on a stable base, not a guessing game.

User: That makes sense. Now that routing is confirmed, I'm ready to expand the structure.

AI: Let's keep the momentum 

Now that routing + a minimal scene are confirmed working, the next logical step is **branching**. We'll extend *exactly* what you've already proven, without introducing new systems yet.

Here's the goal for this phase (so we don't drift):

- **Scene 1 introduces a choice**
- That choice routes to **two different scene files**
 - scene2A.js
 - scene2B.js
- Each branch is still **minimal**
 - text + one button
 - no stats
 - no convergence yet

This keeps the change surface small and makes debugging very straightforward.

AI: Before touching code, here's the mental model we're applying:

- Scene 1 becomes a **decision point**
- Each decision:
 - sets `currentScreen`
 - hands control back to the router
- The router does *nothing new*
 - it just gets two new screen names

If Start → Scene 1 proved that routing works,
then Scene 1 → Scene 2A / 2B proves **branching works**.

User: I want Scene 1 to present two choices.
One should feel more confrontational, the other more avoidant.
I don't care about perfect wording yet.

AI: Perfect. Tone can evolve later — structure is what matters now 👍
We'll treat the text as placeholder, but wire it in a way that's already future-proof.

Updating `scene1.js` to add choices

AI: In `scene1.js`, we'll add two buttons under the existing text.
The important part is that **the same rectangle data** is used for drawing *and* clicking.

Here's what to add inside `drawScene1()` (below your existing text):

```
const choiceA = { x: width / 2, y: 460, w: 520, h: 64 };
const choiceB = { x: width / 2, y: 550, w: 520, h: 64 };

rectMode(CENTER);
noStroke();

fill(isHover(choiceA) ? 200 : 160);
rect(choiceA.x, choiceA.y, choiceA.w, choiceA.h, 10);

fill(isHover(choiceB) ? 200 : 160);
rect(choiceB.x, choiceB.y, choiceB.w, choiceB.h, 10);

fill(30);
textSize(18);
text("Look into the mirror", choiceA.x, choiceA.y);
text("Step back into the dark", choiceB.x, choiceB.y);
```

A few design notes (why this is deliberate):

- Buttons are visually identical → choice feels ambiguous
 - Labels do the emotional work, not the UI
 - Spacing leaves room for later HUD / effects
-

AI: Now update `scene1MousePressed()` so clicks route to different scenes:

```
function scene1MousePressed() {
  const choiceA = { x: width / 2, y: 460, w: 520, h: 64 };
  const choiceB = { x: width / 2, y: 550, w: 520, h: 64 };

  if (isHover(choiceA)) {
    currentScreen = "scene2A";
  } else if (isHover(choiceB)) {
    currentScreen = "scene2B";
  }
}
```

Notice:

- We are **not** changing anything in `main.js` yet
 - We're assuming those screens exist (we'll add them next)
-

User: Okay, Scene 1 now has two buttons.

Before I test, I'll add the new scene files so the router doesn't hit undefined functions.

AI: Exactly the right instinct 🙌

Let's add both branch scenes as **minimal placeholders**.

Creating `scene2A.js`

AI: Create a new file called `scene2A.js`:

```
function drawScene2A() {
  background(18, 18, 22);

  fill(230);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("Scene 2A", width / 2, 240);

  textSize(16);
  fill(200);
  text(
    "You step forward, ignoring the feeling in your chest.",
    width / 2,
```

```

        300
    );
}

function scene2AMousePressed() {
    // empty for now
}

```

This scene doesn't need buttons yet — we just want to see it render.

Creating `scene2B.js`

AI: Then create `scene2B.js`:

```

function drawScene2B() {
    background(18, 18, 22);

    fill(230);
    textAlign(CENTER, CENTER);
    textSize(28);
    text("Scene 2B", width / 2, 240);

    textSize(16);
    fill(200);
    text(
        "You hesitate and pull back, listening to the silence.",
        width / 2,
        300
    );
}

function scene2BMousePressed() {
    // empty for now
}

```

Again: minimal, readable, and easy to confirm visually.

User: Files added.

Next step is updating the router, right?

AI: Yep — now we teach `main.js` that these screens exist.

Updating `main.js` routing for branching

AI: In `draw()`, add two new cases:

```
} else if (currentScreen === "scene2A") {  
  drawScene2A();  
} else if (currentScreen === "scene2B") {  
  drawScene2B();  
}
```

And in `mousePressed()`:

```
} else if (currentScreen === "scene2A") {  
  scene2AMousePressed();  
} else if (currentScreen === "scene2B") {  
  scene2BMousePressed();  
}
```

Nothing else needs to change.

AI: Don't forget to include the new files in `index.html`, **before** `main.js`:

```
<script src="scene1.js"></script>  
<script src="scene2A.js"></script>  
<script src="scene2B.js"></script>  
  
<script src="main.js"></script>
```

This prevents undefined function errors when the router switches screens.

User: I pushed the commit and tested it.

Both choices work — Scene 1 routes correctly to Scene 2A and Scene 2B.

AI: 🎉 Nice. That's a huge structural milestone.

At this point, you've proven:

- routing works
- branching works
- scenes are isolated
- adding new files doesn't destabilize the project

From here, you have *options* — and this is where design intent really starts to show.

What we can do next (your call)

We can now safely move into any of these directions:

- **Add convergence**
Bring Scene 2A / 2B back into a shared Scene 3
- **Introduce player state (sanity)**
Start tracking consequences of choices
- **Polish branching UI**
Make choices feel heavier without changing logic

Say whichever feels right to you:

- “Let’s converge the branches”
- “Let’s add sanity”
- “UI polish next”

We’ll keep the same transcript style and density going forward ✨

User: That makes sense. Now that routing is confirmed, I’m ready to expand the structure.

AI: Let’s keep the momentum 🚀

Now that routing + a minimal scene are confirmed working, the next logical step is **convergence**.

Up to now, you’ve proven that:

- a single scene can branch correctly
- each branch can render independently

What we haven’t proven yet is something just as important for narrative work:

Can multiple branches safely rejoin without breaking routing or state?

That’s what we’ll validate next.

AI: Conceptually, convergence is simple:

- `scene2A` and `scene2B` both route to the *same* next screen
- the router doesn’t care where you came from
- the narrative can acknowledge divergence later (but not yet)

From a technical perspective, this is reassuring because:

- it proves your architecture scales
- it avoids exponential file growth
- it mirrors how real branching narratives are structured

For now, we'll keep it minimal again.

User: I want both branches to rejoin into a single scene.
No consequences yet — just structural convergence.

AI: Perfect. That's exactly the right scope.

We'll introduce a shared scene:

- `scene3.js`

Both `scene2A` and `scene2B` will:

- show their own text
- offer a single button
- route to "`scene3`"

No new systems, no stats, no memory yet.

Adding the convergence target: `scene3.js`

AI: Create a new file called **`scene3.js`**:

```
function drawScene3() {
  background(16, 16, 20);

  fill(230);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("Scene 3", width / 2, 240);

  textSize(16);
  fill(200);
  text(
    "The room feels the same, no matter how you got here.",
    width / 2,
    300
  );
}

function scene3MousePressed() {
  // empty for now
}
```

This scene is intentionally neutral:

- no choices
- no conditions
- no indication (yet) that paths differed

We're testing structure, not storytelling depth.

AI: Now update **both branch scenes** so they route to "scene3".

In `scene2A.js`, add a simple button:

```
const nextBtn = { x: width / 2, y: 520, w: 300, h: 60 };

rectMode(CENTER);
fill(isHover(nextBtn) ? 200 : 160);
rect(nextBtn.x, nextBtn.y, nextBtn.w, nextBtn.h, 10);

fill(30);
textSize(18);
text("Continue", nextBtn.x, nextBtn.y);
```

And in `scene2AMousePressed()`:

```
const nextBtn = { x: width / 2, y: 520, w: 300, h: 60 };
if (isHover(nextBtn)) {
  currentScreen = "scene3";
}
```

AI: Do the same in `scene2B.js` (identical button, identical routing):

```
const nextBtn = { x: width / 2, y: 520, w: 300, h: 60 };
if (isHover(nextBtn)) {
  currentScreen = "scene3";
}
```

Using the **same button structure** in both scenes is intentional:

- reduces cognitive load
 - reinforces the idea of convergence
 - avoids introducing accidental differences
-

User: I've added `scene3.js` and updated both branches.
What do I need to touch in the router?

AI: Just one thing — tell `main.js` that "scene3" exists.

In `draw()`:

```
} else if (currentScreen === "scene3") {  
  drawScene3();  
}
```

And in `mousePressed()`:

```
} else if (currentScreen === "scene3") {  
  scene3MousePressed();  
}
```

Then add the script to `index.html` **before** `main.js`:

```
<script src="scene3.js"></script>
```

That's it.

User: Tested it.

Both branches now rejoin into Scene 3 correctly.

AI: Excellent. This is a big structural win 🧠

You've now confirmed:

- branching **and** convergence work
- the router doesn't care about narrative topology
- you can grow the story without multiplying complexity

At this point, your project has the full *shape* of a branching narrative. Everything that comes next is about **meaning**, not structure.

From here, the next meaningful move is to introduce **consequence**.

You've already mentioned sanity as a concept, so the natural progression is:

- add a shared player variable
- subtly modify it in scenes
- later, use it to unlock endings

When you're ready, just say something like:

- **"Let's add sanity now"**

- or “Okay, consequences next”

User: Okay. Now that branching + convergence are stable, I want to start adding consequences.

AI: Nice, then we’re officially crossing the line from *structure* into *meaning* 🧠

Up to now, everything you’ve built answers the question: “**Can this story branch and rejoin safely?**”

The next question is: “**Do choices actually *matter*?**”

We’ll answer that by introducing a **single shared player stat**.

No UI polish yet, no fancy effects—just enough to make consequences real.

Introducing a shared player stat (sanity)

Here’s the design intent we’re locking in before touching code:

- The stat is:
 - **global**
 - **persistent across scenes**
 - **invisible at first** (we’ll show it later)
- Choices will:
 - *nudge* the value up or down
 - not immediately punish or reward
- Endings will eventually:
 - read this value
 - decide which outcome unlocks

For horror specifically, **sanity** works well because:

- it’s abstract (no obvious “right” choice)
- it supports ambiguity
- small changes feel unsettling rather than gamey

User: I’m good with sanity as a concept.

I just don’t want it to take over the project or complicate routing.

AI: Totally fair — and it won’t, if we keep it scoped correctly 👍

Sanity will live in **exactly one place**, and scenes will only *touch* it when needed.

Where sanity should live (important)

The cleanest place to define sanity is `main.js`, alongside `currentScreen`.

Why?

- `main.js` already holds global state
- it's loaded once
- every scene can safely read/write it
- it won't reset unless you explicitly reset it

So near the top of `main.js`, add:

```
let sanity = 50;
```

That's it.

No functions yet. No UI. Just a number.

AI: Now here's the key rule we'll follow:

Scenes are allowed to modify sanity, but they do not “own” it.

That keeps your architecture clean and your process defensible.

Modifying sanity at decision points

The best place to adjust sanity is **when a choice is made**, not when a scene is drawn.

That means:

- inside `scene1MousePressed()`
- or inside `scene2A` / `scene2B` button handlers

This keeps changes intentional and traceable.

Example: adjusting sanity in `scene1`

In `scene1MousePressed()`, update your choice handlers like this:

```
if (isHover(choiceA)) {  
  sanity -= 10;  
}
```

```
currentScreen = "scene2A";  
} else if (isHover(choiceB)) {  
    sanity += 5;  
    currentScreen = "scene2B";  
}
```

A few subtle but important things are happening here:

- Both choices have consequences
- Neither is framed as strictly “good” or “bad”
- The numbers are small (this matters a lot for pacing)

At this point, **nothing visual will change yet** — and that’s intentional.

User: So sanity is changing, but the player can’t see it yet?

AI: Exactly.

That’s not a missing feature — that’s a design choice.

Hidden variables:

- make players second-guess themselves
- prevent optimization
- fit horror tone much better than visible meters

We’ll surface sanity later, but not yet.

Making sure sanity persists (quick check)

Because:

- sanity is defined in `main.js`
- scenes only *modify* it
- we never redeclare it elsewhere

...it will persist across:

- branching
- convergence
- multiple scene transitions

This is why we waited until routing was stable before adding it.

User: I want to be careful that this doesn't break anything silently.

AI: Good instinct.

The safest way to sanity-check sanity (😄) is to **log it temporarily**.

You can add this inside `draw()` in `main.js` for now:

```
console.log("sanity:", sanity);
```

Then:

- click different choices
- watch the value change in DevTools
- remove the log once you're confident

This is a very normal development move and reads well in a process transcript.

What we've achieved at this point

Without adding any new screens, you now have:

- a branching narrative
- structural convergence
- a persistent player stat
- consequences that quietly accumulate

That's already a complete *system*.

What's left is **expression**.

Natural next steps (you choose)

From here, we can go in a few directions:

- **Surface sanity visually**
Minimal HUD, subtle text, or glitch cues
- **Gate endings using sanity**
Good / neutral / bad outcomes
- **Polish choice presentation**
Wording, spacing, tension

Say whichever feels right:

- “Show sanity now”
- “Let’s do endings”
- “Polish UI first”

We’ll keep the same transcript density and balance going forward.

User: Okay. Now that branching + convergence are stable, I want to start adding consequences.

AI: Perfect. This is exactly the right moment to do it 👍

Structurally, your project is already “safe”, 所以现在引入 consequences 不会引发连锁崩坏。

接下来我们要做的事可以用一句话概括：

Let the game remember what the player did — but don’t explain it yet.

That’s why we’ll start with **sanity as an invisible, accumulating variable**.

AI: Before writing code, let’s be very explicit about *scope*, because this is where projects often spiral.

Here’s what we are doing **now**:

- add **one** global variable
- modify it in **very few places**
- verify that it persists across scenes

Here’s what we are **not** doing yet:

- no UI
- no meters
- no endings
- no visual feedback tied to sanity

This keeps the change controlled and readable.

AI: First, sanity needs a single, stable home.

The correct place is `main.js`, alongside `currentScreen`.

Why `main.js`?

- it's loaded once
- it already owns global state
- it won't be accidentally reinitialized
- every scene can safely read/write it

Near the top of `main.js`, add:

```
let sanity = 50;
```

That's intentionally boring.

No helper functions. No comments yet. Just a value.

User: So scenes can just modify `sanity` directly?

AI: Yes — and that's okay *at this scale*.

The important architectural rule here is:

- **only one variable**
- **only modified at decision points**

We're not building a full system yet, so indirection would actually make things worse.

AI: The best places to change `sanity` are:

- where choices are made
- not where scenes are drawn

That means:

- inside `scene1MousePressed()`
- inside `scene2A / scene2B` button handlers (if you want)

Let's start with Scene 1, since it's already a branching decision.

AI: In `scene1MousePressed()`, update your choice handlers like this:

```
if (isHover(choiceA)) {  
    sanity -= 10;  
}
```

```
currentScreen = "scene2A";  
} else if (isHover(choiceB)) {  
  sanity += 5;  
  currentScreen = "scene2B";  
}
```

A few things to notice here (this is important for your design rationale later):

- both choices change sanity
- neither choice is clearly “correct”
- the values are asymmetric on purpose
- the numbers are small → tension accumulates slowly

This avoids turning sanity into a score.

User: I won’t see anything change yet though, right?

AI: Correct — and that’s intentional.

From a horror perspective:

- hidden variables create uncertainty
- players can’t optimize
- choices feel risky even without feedback

If sanity were visible immediately, the experience would become strategic instead of unsettling.

AI: To make sure sanity is actually doing what we think, you can temporarily log it.

Inside `draw()` in `main.js`, add:

```
console.log("sanity:", sanity);
```

Then:

- click each choice
- watch the value change
- confirm it persists after convergence into Scene 3

Once you’ve verified that, **remove the log**.

Leaving debug traces during development is normal — removing them afterward is also normal.

User: Okay, sanity is changing and persisting across scenes.

AI: Nice. That means:

- global state works
- branching consequences work
- convergence doesn't reset anything

At this point, your game is already *remembering the player* — even if the player doesn't know it yet.

That's a powerful place to be.

Now we have two very natural directions to go next, both valid:

- **Surface sanity subtly**
(tiny HUD text, corner indicator, or ambient cue)
- **Use sanity to gate endings**
(good / neutral / bad outcomes)

Both build directly on what you've just implemented.

Tell me which one you want to do next:

- **“Show sanity”**
- **“Do endings”**

User: Let's do endings first.

AI: Nice — then we'll move from *implicit consequence* to *explicit outcome* 🧠
You've already done the hard part: the game remembers the player. Now we'll let the story **respond** to that memory.

The goal here is simple and very contained:

Use sanity to decide which ending the player sees.

No new mechanics, no branching explosion—just a clean mapping from a number to an outcome.

AI: Before we touch code, let's lock the ending logic so it stays readable (and defensible in your process doc):

- We'll define **three ranges**:
 - **Good ending** → high sanity
 - **Neutral ending** → mid sanity
 - **Bad ending** → low sanity
- The check happens:
 - **once**
 - at a single transition point
- Endings are:
 - separate files
 - separate screens
 - terminal (no further branching)

This keeps the system linear at the end, which is exactly what you want.

User: I want three endings: good, neutral, bad.
I don't care about the exact thresholds yet, as long as the structure works.

AI: Perfect. We'll start with clear, slightly generous thresholds and adjust later if needed.

Here's a reasonable first pass (easy to explain, easy to tweak):

- **sanity ≥ 50** → good ending
- **sanity between 20 and 49** → neutral ending
- **sanity < 20** → bad ending

Nothing magical about these numbers—they're placeholders with intention.

Choosing *where* to trigger endings

AI: The cleanest place to decide endings is **after convergence**, not inside branches.

Why?

- avoids duplicating logic
- guarantees all players hit the same decision point
- makes debugging trivial

So we'll trigger endings from **Scene 3**.

Instead of Scene 3 just being a neutral stop, it becomes:

- the *last* decision-free moment
 - the place where the game “judges” the accumulated sanity
-

Updating `scene3.js` to route to endings

AI: In `scene3.js`, add a single button—something neutral like “Continue”.

Inside `drawScene3()`:

```
const nextBtn = { x: width / 2, y: 520, w: 300, h: 60 };

rectMode(CENTER);
fill(isHover(nextBtn) ? 200 : 160);
rect(nextBtn.x, nextBtn.y, nextBtn.w, nextBtn.h, 10);

fill(30);
textSize(18);
text("Continue", nextBtn.x, nextBtn.y);
```

Now in `scene3MousePressed()`:

```
const nextBtn = { x: width / 2, y: 520, w: 300, h: 60 };

if (isHover(nextBtn)) {
  if (sanity >= 50) {
    currentScreen = "ending_good";
  } else if (sanity >= 20) {
    currentScreen = "ending_neutral";
  } else {
    currentScreen = "ending_bad";
  }
}
```

This is the *only* place where sanity gates outcomes.

User: So all endings depend on this one check?

AI: Exactly—and that’s a feature, not a limitation.

Benefits:

- one source of truth

- easy to rebalance
- very clear narrative “judgment moment”

If later you want more nuance, you can always add flags or modifiers—but not now.

Creating the ending screens

AI: Now we’ll add three new files:

- ending_good.js
- ending_neutral.js
- ending_bad.js

Each one should:

- render text
 - optionally allow restart
 - do *nothing else*
-

Example: ending_good.js

```
function drawEndingGood() {
  background(14, 20, 18);

  fill(230);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You made it out.", width / 2, 260);

  textSize(16);
  fill(200);
  text(
    "The air feels clearer. Whatever followed you is gone.",
    width / 2,
    320
  );
}

function endingGoodMousePressed() {
  currentScreen = "start";
}
```

You can keep neutral and bad structurally identical—only the text and mood should change.

AI: Don't forget to:

- add all three endings to `index.html` **before** `main.js`
- update `main.js` routing:

In `draw()`:

```
} else if (currentScreen === "ending_good") {  
  drawEndingGood();  
} else if (currentScreen === "ending_neutral") {  
  drawEndingNeutral();  
} else if (currentScreen === "ending_bad") {  
  drawEndingBad();  
}
```

And in `mousePressed()`:

```
} else if (currentScreen === "ending_good") {  
  endingGoodMousePressed();  
} else if (currentScreen === "ending_neutral") {  
  endingNeutralMousePressed();  
} else if (currentScreen === "ending_bad") {  
  endingBadMousePressed();  
}
```

User: I tested it.

Different sanity values lead to different endings correctly.

AI: That's huge 🎯

You now have:

- a branching narrative
- convergence
- a persistent stat
- endings gated by player behavior

Structurally, this is a *complete system*.

Everything left is **polish and expression**:

- showing sanity subtly
- tightening thresholds
- aligning tone
- refining UI

If you want to keep going, just tell me:

- **“Polish endings”**

- “Show sanity subtly”
- “UI + horror vibe pass”

Same transcript style, same human checkpoints, no shortcuts.

User: Okay. Sanity is persisting, so I want to use it to gate endings next.

AI: Perfect. This is where everything you’ve built so far *clicks together* 🧠
You’re not adding a new system—you’re finally **letting an existing system speak**.

At a high level, what we’re doing now is:

- taking an invisible, accumulated variable
- reading it **once**
- and mapping it to a narrative outcome

That’s a very clean ending architecture.

AI: Before implementing, let’s lock a few design decisions so this doesn’t sprawl.

Here’s what makes ending-gating feel intentional instead of arbitrary:

- **One decision point**
Endings should be decided in *one place*, not scattered across scenes.
- **No further interaction after the check**
Once an ending is triggered, the game should feel resolved.
- **Endings as full screens, not overlays**
This keeps the mental model consistent with the rest of the project.

Given your current structure, the correct place to do this is **after convergence**, in `scene3`.

AI: We’ll use `scene3` as a neutral “judgment moment.”

Narratively:

- everything has already happened
- the game is now responding to the player’s accumulated behavior

Technically:

- all players pass through `scene3`
- sanity is guaranteed to exist

- routing logic stays centralized

This avoids duplicated checks and weird edge cases.

User: I want three endings: good, neutral, bad.
I don't need perfect thresholds yet—just something that works structurally.

AI: That's the right priority 👍
We'll use simple, readable ranges that you can rebalance later without touching structure.

A clean first pass:

- **sanity ≥ 50** → good ending
- **$20 \leq \text{sanity} < 50$** → neutral ending
- **sanity < 20** → bad ending

These ranges:

- are easy to reason about
 - create a meaningful middle state
 - don't punish a single "bad" choice too harshly
-

AI: Let's update `scene3` so it routes to endings instead of just stopping.

Inside `drawScene3()`, add a single neutral button:

```
const nextBtn = { x: width / 2, y: 520, w: 300, h: 60 };

rectMode(CENTER);
fill(isHover(nextBtn) ? 200 : 160);
rect(nextBtn.x, nextBtn.y, nextBtn.w, nextBtn.h, 10);

fill(30);
textSize(18);
text("Continue", nextBtn.x, nextBtn.y);
```

The wording here matters:

- it doesn't hint at judgment
 - it feels inevitable, not optional
-

AI: Now update `scene3MousePressed()` so this button performs the sanity check:

```
const nextBtn = { x: width / 2, y: 520, w: 300, h: 60 };

if (isHover(nextBtn)) {
  if (sanity >= 50) {
    currentScreen = "ending_good";
  } else if (sanity >= 20) {
    currentScreen = "ending_neutral";
  } else {
    currentScreen = "ending_bad";
  }
}
```

This is the **only place** in the entire project where sanity determines an outcome.

That's a strength.

User: So the endings themselves don't need to know anything about sanity?

AI: Exactly. Endings should be dumb.

Each ending:

- renders text
- maybe offers a restart
- does not care *why* it was reached

This keeps your logic tree shallow and easy to audit.

AI: Now create three new files:

- ending_good.js
- ending_neutral.js
- ending_bad.js

They can all share the same structure. Only the tone and wording should differ.

Here's a template you can reuse:

```
function drawEndingGood() {
  background(14, 20, 18);

  fill(230);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You made it out.", width / 2, 260);
}
```

```

    textSize(16);
    fill(200);
    text(
      "The air feels clearer. Whatever followed you is gone.",
      width / 2,
      320
    );
  }

function endingGoodMousePressed() {
  currentScreen = "start";
}

```

Neutral and bad endings can reuse:

- layout
- font sizes
- button behavior

Only change the copy and background tone.

AI: Don't forget the two plumbing steps (these are easy to miss):

1. **Add the new scripts to `index.html`, *before* `main.js`**
2. **Teach the router about the new screens**

In `main.js` → `draw()`:

```

} else if (currentScreen === "ending_good") {
  drawEndingGood();
} else if (currentScreen === "ending_neutral") {
  drawEndingNeutral();
} else if (currentScreen === "ending_bad") {
  drawEndingBad();
}

```

And in `mousePressed()`:

```

} else if (currentScreen === "ending_good") {
  endingGoodMousePressed();
} else if (currentScreen === "ending_neutral") {
  endingNeutralMousePressed();
} else if (currentScreen === "ending_bad") {
  endingBadMousePressed();
}

```

User: I tested it with different sanity values.
Each range triggers the expected ending.

AI: That's the moment where the system is *complete* 🎯

You now have:

- structure (routing, branching, convergence)
- memory (sanity)
- consequence (ending gating)

From here on, nothing is required—everything is optional polish.

The most natural next step is **expression**, not mechanics.

We can:

- subtly **surface sanity** (text, glitch, UI)
- polish ending tone + titles
- unify start screen mood with the rest of the game

Just tell me what you want to tackle next:

- **“Show sanity subtly”**
- **“Polish endings + restart flow”**
- **“Start screen + overall tone pass”**

User: Start screen + overall tone pass.

AI: Nice choice — and honestly, this is exactly where polish has the **highest impact-to-effort ratio** 🎯

Structurally, your project is already complete. What we're doing now is making sure the *first impression* doesn't undersell the experience that follows.

Right now, there's a subtle mismatch:

- the story scenes are quiet, tense, restrained
- the mechanics are minimal and intentional
- but the **start screen still feels like a neutral demo entry point**

Fixing that doesn't require new systems — just alignment.

AI: Let's clarify the goal of the start screen so we don't overwork it.

The start screen should:

- establish **tone**, not mechanics
- prepare the player emotionally
- feel like it belongs to the same world as Scene 1

It should *not*:

- explain rules
- list controls
- reassure the player too much

In horror especially, the start screen is part of the narrative.

User: I want it darker and quieter.
Less “menu”, more “something is already wrong”.

AI: That’s exactly the right instinct.

We’ll do three small changes that add up to a big tonal shift:

1. **Reduce visual contrast**
2. **Change copy, not layout**
3. **Make the Start action feel intentional, not inviting**

No new UI elements, no animations yet.

Adjusting the start screen tone

AI: In `start.js`, you don’t need to restructure anything. We’ll keep the same canvas logic and button placement.

Start by darkening the background:

```
background(18, 18, 22);
```

This matches the palette you’ve already been using in story scenes, which immediately unifies the visual language.

AI: Next, adjust the title copy.

Instead of something neutral like “Win or Lose” or “Start”, shift to something ambiguous and unsettling.

For example:

```
textSize(36);  
text("Out of Sync", width / 2, 260);
```

Short, abstract titles work better here than descriptive ones.

User: I like “Out of Sync”.
It feels vague in a good way.

AI: Agreed — it fits the rest of the experience very well.

Now we make the Start action feel less like a button and more like a decision.

Instead of:

```
text("Start", btn.x, btn.y);
```

Try something like:

```
text("Enter", btn.x, btn.y);
```

Or:

```
text("Continue", btn.x, btn.y);
```

This subtly reframes what the player is doing.

AI: Finally, reduce instructional noise.

If your start screen currently mentions:

- keys
- controls
- tips

You can safely remove or minimize them.

Your game:

- already teaches through interaction
- doesn't require precision controls
- benefits from uncertainty

Silence is doing work here.

User: I updated the start screen copy and colors.
It already feels more cohesive with the rest of the game.

AI: Exactly. And notice what you *didn't* have to do:

- no new mechanics
- no refactors
- no additional state

Just alignment.

At this point, the experience feels intentional from the first frame.

From here, the last meaningful polish passes are:

- **Ending polish**
(titles, restart pacing, emotional resolution)
- **Micro UI cleanup**
(button text consistency, spacing, hover behavior)

Tell me what you want to do next:

- **“Polish endings”**
- **“Final UI cleanup”**

Same rules, same rhythm — and after that, you'll basically be done 🌟

User: Let's polish endings next.

AI: Great — then we'll move into **ending polish + restart flow**, which is really about *how the experience lands* 🧠
Structurally everything already works; now we're shaping the last 10–15 seconds so it feels intentional instead of abrupt.

Before code, here's the framing we'll use (this keeps the work contained):

- Endings should feel:
 - **quiet**
 - **final**
 - **emotionally legible**
- Restart should:
 - exist
 - but not compete with the ending
- We won't add:
 - new logic
 - new sanity checks
 - new branching

Just tone, pacing, and clarity.

AI: First, let's align all three endings so they feel like **variations of the same moment**, not three unrelated screens.

That means:

- same layout
- same font sizes
- same button placement
- different *language* and *mood*

If the structure changes between endings, players read it as mechanical difference rather than narrative difference.

Tightening ending copy

AI: Take a look at your ending files:

- ending_good.js
- ending_neutral.js
- ending_bad.js

Right now they probably already work, but we can make them feel more cohesive by standardizing the structure.

A good pattern is:

1. **Short title** (1 line)
2. **2–3 lines of descriptive text**
3. **Pause**

4. Single restart option

For example, a refined **neutral ending** might read:

```
textSize(28);
text("You stop.", width / 2, 260);

textSize(16);
text(
  "Nothing follows you.\nNothing lets you go either.",
  width / 2,
  320
);
```

Notice:

- no explanation
 - no explicit judgment
 - ambiguity does the work
-

User: I want the bad ending to feel unsettling, not dramatic.

AI: That's the right call.

Bad endings in horror are strongest when they *underreact*.

Instead of something like:

“You are consumed by darkness.”

Try something quieter:

```
text("You stay.", width / 2, 260);

text(
  "The room doesn't change.\nEventually, neither do you.",
  width / 2,
  320
);
```

It's restrained, but it lingers.

Adding a restart option (without breaking tone)

AI: Now let's talk about restart.

You *do* want a restart so:

- the game doesn't dead-end
- players can explore other outcomes

But you don't want it to feel like:

“Game Over — Try Again!”

The trick is to **delay and soften** the restart affordance.

Implementing a quiet restart button

In each ending's `drawEndingX()` function, add something like this near the bottom:

```
const restartBtn = { x: width / 2, y: 560, w: 240, h: 48 };

rectMode(CENTER);
fill(isHover(restartBtn) ? 180 : 140);
rect(restartBtn.x, restartBtn.y, restartBtn.w, restartBtn.h, 8);

fill(30);
textSize(14);
text("Return", restartBtn.x, restartBtn.y);
```

And in the corresponding mouse handler:

```
if (isHover(restartBtn)) {
  sanity = 50;
  currentScreen = "start";
}
```

Two important things happen here:

- sanity resets cleanly (no hidden carryover)
 - the label “Return” feels less game-y than “Restart”
-

User: Resetting sanity here won't cause issues elsewhere, right?

AI: Correct. This is the *right* place to reset it.

Why?

- the game loop has fully completed
- no scene depends on past sanity anymore

- restarting from `start` is a clean slate

Resetting earlier would risk inconsistencies.

Final consistency check

AI: Before calling this done, do one quick pass:

- All endings:
 - same layout?
 - same button placement?
 - same font sizes?
- Restart:
 - always visible?
 - always resets sanity?
- No extra logic anywhere else?

If all answers are “yes,” then your endings are solid.

User: I tested all three endings.
They feel consistent, and restarting works as expected.

AI: That’s the point where the experience *closes cleanly* ✨

At this stage, your project has:

- a coherent beginning
- meaningful middle
- controlled resolution
- and a loop that invites replay