Process & Decision Documentation

Aden Silaphet – 21092270

GenAI Documentation

Everyone must complete this section. If not GenAI was used, write, "No GenAI used for this task." When GenAI is not used, process evidence should still demonstrate iteration, revision, or development over time.

Because GenAI can closely mimic human-created work, instructors or TAs may occasionally request additional process evidence to confirm non-use. This may include original working files (e.g., an illustrator file), intermediate drafts, or a brief check-in with a TA to walk through your process.

These requests are not an assumption of misconduct. They are part of ensuring academic integrity in an environment where distinguishing between human-created and AI-generated work is increasingly difficult.

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

Date Used: Tuesday Feb 3, 2026

Tool Disclosure: ChatGPT 5.2

Purpose of Use: used for brainstorming initial ideas, creating code, and debugging.

Summary of Interaction:

- I used the tool initially to brainstorm some ideas on how to create an interactive game to code for the sidequest
- Once I picked an idea, I used it to create the code for the game such as how to add multiple buttons to the screen that lead to different pages
- I prompted it many times to ask it how to add specific things such as raindrops on the screen, or a sun, or how to add keypressed/mousepressed functions
- I also used it to debug the code when it wasn't working
- I would give it pieces of the code when prompting to help either add onto it or debug it

Some of the prompts:

- "can you give me some ideas on how to build an interactive story that unfolds through multiple game states and files, branching like a small decision tree through p5.js"

- “now i have both buttons on the screen, how do i make them each go to a different screen? the screen can be blank for now”
- “how do i add still raindrops, they don't have to be moving just something simple”
- “scratch the next button idea, how do i add a key that takes you back to the start screen”
- “how do I basically do the exact same thing as the initial disappointed and okay screen and add two buttons that lead to two new pathways onto those screens”

Human Decision Point(s): There were many times where I had to redirect the output of genAI because it would overcomplicate things at times such as when I asked it to create raindrops. I only wanted still raindrop-like images on the background but it gave me code for static/moving raindrops. I also wanted to add a new button at one point but it wasn’t working and wouldn’t debug so I rejected the ideas it gave me and went with a keypressed option instead. I would also modify smaller things at times such as the texts, layouts, colours, and positioning of the code.

Integrity & Verification Note: To make sure that the code was fitting to the course concepts I would do my best to double check the code genAI would output and compare it to the example and make sure it was similar, such as things like if statements or the different functions that were already in the example to begin with.

Scope of GenAI Use: GenAI mostly did not contribute to the smaller changes I would make. It contributed to the concepts of the code but I would make little tweaks such as colours, positioning, and texts. Also once I got the hang of the structure of some pages such as the okay and disappointing pages, I was able to structure most of the similar pages myself such as the next pages frustrating and funny paths. I asked the genAI so many things and gave it so many prompts but I feel like I did not use a lot of what it gave me.

Limitations or Misfires: The tool was good at debugging at times but also bad, at one point I was trying to fix a bug and the genAI kept outputting a lot of fixes but none of them would work until I realized one small change myself.

Appendix

Many of the prompts and outputs I did not use, but I included the whole conversation*

can you give me some ideas on how to build an interactive story that unfolds through multiple game states and files, branching like a small decision tree through p5.js

5. Story Concepts You Could Use

1. “Lost Phone”

- You find a stranger’s phone
- texts arrive in real time
- choices = reply styles
- endings: help them / get trapped / mystery solved

---

2. “Last Bus Home”

- miss the bus → explore city
- meet 3 characters
- who you trust changes ending

---

3. “Dream Decoder”

- each door = emotion
- colors change with choices
- true ending only if balanced

---

4. Comedy Route

- You’re a raccoon planning a heist
- choices = stealth / chaos
- inventory = pizza crusts

what are some simple concepts like clicking a certain path and seeing a different colour but a little less simple than that

Nice—so you want ideas that are still **achievable for a student p5.js project**, but feel more meaningful than just:

“click left → screen turns red, click right → screen turns blue”

Here are concepts that add a layer of interaction, consequence, or personality **without becoming a full RPG engine**.

---

1. Emotion Garden 🌱

Mechanic:
Each choice plants a “feeling seed.”
The world slowly changes colour + shape based on your decisions.

- Kind choice → warmer colours, round shapes
- Selfish choice → sharp shapes, colder palette
- Neutral → grey fog grows

Why it’s better:

- Not just instant colour swap
- Accumulative effect over time
- Visual storytelling instead of text-only

Tech ideas

- array of particles
- tint() based on “morality score”
- flowers grow with frameCount

---

2. Memory Glitch Story 🧠

You play someone with corrupted memories.

- Click different objects in a room
- Each reveals a fragment
- Fragments slightly contradict each other
- Final scene depends on which you believed

Interaction

- Hover to “focus”
- Click to lock a memory
- Background distorts more with each pick

Skills

- multiple states
- image transparency
- simple sound glitches

---

3. Conversation That Remembers Tone 💬

Instead of choices = destinations
choices = TONE

- friendly
- sarcastic
- aggressive

NPC reacts differently later:

- door guard lets you pass
- or refuses
- or follows you

System

```
let tone = 0;

if(choice === "kind") tone++;
if(choice === "rude") tone--;
```

Ending depends on tone score.

---

4. Weather You Control ☁️

Each decision changes weather:

- avoid problem → clouds grow
- face it → sun appears
- lie → lightning

Later scenes reuse that weather.

Nice visuals

- rain particles
- thunder shake
- sun rays with lerp()

---

5. “Choose HOW to Look” Story 👁️

Same event, different lenses:

- Logical
- Emotional
- Curious

You click a lens →
scene redraws with different narration + colours.

Example:

Teacher calls your name

Logical: “Probably attendance.”
Emotional: “I’m in trouble.”
Curious: “Maybe a surprise.”

---

6. Trust Meter Thriller 🔒

You meet 3 characters.

- every click changes TRUST bars
- music layer changes
- one character betrays you based on lowest trust

Visual:

- bars
- character tint
- heartbeat speed

---

7. Map That Grows 🗺️

Not just paths → an actual map appears.

- each choice reveals new node
- previous paths stay visible
- you see your “decision tree” live

Great for assignment marks because:

- shows system thinking
- visualizes branching

---

8. Diary With Rewrites ✏️

You read a diary page.

Click a sentence →
you can rewrite ONE line.

Future pages reference YOUR version.

---

9. Sound-Based Story 🎧

- click radio channels
- each channel = different narrator
- static increases when story is dangerous

Super simple technically but feels deep.

---

10. Room That Judges You 🪞

Objects react to you:

- mirror gets clearer or darker
- plant grows or dies
- clock speeds up

All from 3–4 simple variables.

If you want, I can:

👉 Turn ANY of these into:

- exact game states
- file structure
- starter code
- art style ideas
- rubric-friendly features

Tell me:

1. Is this for a class assignment?
2. Do you prefer:

- emotional / serious
- funny
- creepy
- aesthetic?

3. Must it use:

- keyboard
- mouse
- images
- sound?

I'll design a concept that fits your level but looks impressive 💪

how about a concept with weather that changes depending on the path you click

Love that—weather is perfect because it's **visual, symbolic, and easy to code** without huge mechanics.

I'll give you:

1. A clear concept

2. How the branching works
3. What variables drive the weather
4. p5.js structure you can actually paste
5. Ways to level it up so it’s not “just colours”

🌦️ Concept: “Forecast of Choices”

Story Premise

You’re deciding how to deal with one difficult day.
Each decision affects the **inner weather** of the world.

The weather = your emotional timeline.

- avoiding → clouds grow
- honest → sun breaks
- angry → storm
- curious → wind
- hopeful → rainbow

By the end, the sky shows the life you chose.

---

1. Core Mechanics

Paths

At each scene you choose:

- confront
- ignore
- explore
- lie
- help someone

Each option modifies:

```
let sun = 0;
let clouds = 0;
let rain = 0;
let wind = 0;
```

The draw loop turns those numbers into visuals.

---

2. Example Flow

Scene 1 – Morning text

Your friend cancels plans.

- “It’s okay” → +sun
- “whatever…” → +clouds
- “are you safe?” → +wind (curiosity)

Scene 2 – At school/work

You see someone struggling.

- help → +sun
- walk by → +rain
- joke about it → +storm

Ending

Sky reflects totals:

- Mostly sun → warm ending
- Clouds+rain → lonely ending
- Storm → conflict ending
- Balanced → peaceful sunset

---

3. Why This Is Better Than Simple Colour Change

✅ Weather is gradual, not instant
✅ Past choices still visible
✅ Multiple systems at once
✅ Feels metaphorical / artsy
✅ Easy to mark as “interactive narrative”

---

4. p5.js Structure

sketch.js

```
let state = "start";

// weather values
let sun = 0;
let clouds = 0;
let rain = 0;
let wind = 0;

function draw() {
  background(180, 210, 255);

  drawWeather();

  if (state === "start") scene1();
  if (state === "hallway") scene2();
  if (state === "end") showEnding();
}
```

Weather System

```
function drawWeather() {

  // SUN
  if (sun > 0) {
    fill(255, 200, 50, sun * 20);
    ellipse(500, 80, 80 + sun * 5);
  }

  // CLOUDS
  for (let i = 0; i < clouds; i++) {
    fill(240, 240, 240, 180);
    ellipse(100 + i*120, 100, 120, 60);
  }

  // RAIN
  for (let i = 0; i < rain * 20; i++) {
    stroke(100, 100, 255);
    let x = random(width);
    let y = random(height);
    line(x, y, x, y + 8);
  }

  // WIND
  if (wind > 0) {
    noFill();
    stroke(255);
    for (let i = 0; i < wind; i++) {
      arc(200 + i*60, 200, 40, 20, 0, PI);
    }
  }
}
```

Scene Example

```
function scene1() {
  fill(0);
  text("Your friend cancels plans.", 50, 200);

  text("1 - say it's okay", 50, 240);
  text("2 - ignore them", 50, 260);
  text("3 - ask if they're safe", 50, 280);
}

function keyPressed() {
  if (state === "start") {

    if (key === '1') {
      sun++;
      state = "hallway";
    }

    if (key === '2') {
      clouds++;
      state = "hallway";
    }

    if (key === '3') {
      wind++;
      state = "hallway";
    }
  }
}
```

---

5. Cool Twists You Could Add

A. Weather Affects Choices

If storm too high:

- some options disappear
- text shakes
- colours darker

B. Time of Day

- sun → day
- storm → night
- mix → sunset gradient

C. Character Reactions

NPC says:

- “You seem calm today” (sun)
- “rough day?” (rain)

D. Secret Ending

If all values balanced → rainbow appears.

---

6. Possible Titles

- **“Internal Forecast”**
- **“Climate of Me”**
- **“Weather the Day”**
- **“Skies We Choose”**

Want me to turn this into YOUR version?

Tell me:

1. Tone

- wholesome
- dramatic
- school-life
- creepy
- poetic

2. Input style

- keyboard numbers
- clickable buttons
- clicking objects in scene

3. Need images or just code shapes?

I can generate:

- full multi-file p5 project
- with 3 scenes + 4 endings
- commented for submission 🔥

using this page of code from the game screen, how can I add a second button of the same, above the "it's okay" button that says "it's dissapointing"

You can add a second button in the **exact same style**—just change:

1. the text
2. the Y-position (so it sits above)
3. what variable it affects.

I'll show the idea assuming your current

```
// NOTE: Do NOT add setup() or draw() in this file // setup() and draw() live in main.js // This file only defines: // 1) drawGame() → what the game screen looks like // 2) input handlers → what happens when the player clicks or presses keys // 3) helper functions specific to this screen // ------------------------------ // Button data // ------------------------------ // This object stores all the information needed to draw // and interact with the button on the game screen. // Keeping this in one object makes it easier to move, // resize, or restyle the button later. const gameBtn = { x: 400, // x position (centre of the button) y: 550, // y position (centre of the button) w: 260, // width h: 90, // height label: "It's okay", // text shown on the button }; // ------------------------------ // Main draw function for this screen // ------------------------------ // drawGame() is called from main.js *only* // when currentScreen === "game" function drawGame() { // Set background colour for the game screen background(240, 230, 140); // ---- Title and instructions text ---- fill(0); // black text textSize(32); textAlign(CENTER, CENTER); text("Game Screen", width / 2, 160); textSize(18); text("Your friend cancels plans. How do you feel?", width / 2, 210); // ---- Draw the button ---- // We pass the button object to a helper function drawGameButton(gameBtn); // ---- Cursor feedback ---- // If the mouse is over the button, show a hand cursor // Otherwise, show the normal arrow cursor cursor(isHover(gameBtn) ? HAND : ARROW); } // ------------------------------ // Button drawing helper // ------------------------------ // This function is responsible *only* for drawing the button. // It does NOT handle clicks or game logic. function drawGameButton({ x, y, w, h, label }) { rectMode(CENTER); // Check if the mouse is hovering over the button // isHover() is defined in main.js so it can be shared const hover = isHover({ x, y, w, h }); noStroke(); // Change button colour when hovered // This gives visual feedback to the player fill( hover ? color(180, 220, 255, 220) // lighter blue on hover : color(200, 220, 255, 190), // normal state ); // Draw the button rectangle rect(x, y, w, h, 14); // last value = rounded corners // Draw the button text fill(0); textSize(28); textAlign(CENTER, CENTER); text(label, x, y); } // ---------------
```

```
--------------- // Mouse input for this screen // ------------------------------ // This function is
called from main.js // only when currentScreen === "game" function gameMousePressed() {
// Only trigger the outcome if the button is clicked if (isHover(gameBtn)) {
triggerRandomOutcome(); } } // ------------------------------ // Keyboard input for this screen // -
----------------------------- // Allows keyboard-only interaction (accessibility + design) function
gameKeyPressed() { // ENTER key triggers the same behaviour as clicking the button if
(keyCode === ENTER) { triggerRandomOutcome(); } } // ------------------------------ // Game
logic: win or lose // ------------------------------ // This function decides what happens next in
the game. // It does NOT draw anything. function triggerRandomOutcome() { // random()
returns a value between 0 and 1 // Here we use a 50/50 chance: // - less than 0.5 → win // -
0.5 or greater → lose // // You can bias this later, for example: // random() < 0.7 → 70%
chance to win if (random() < 0.5) { currentScreen = "win"; } else { currentScreen = "lose"; } }
```

Got it—your structure is really clean 👍.
You just need to **create a second button object** and check it the same way you already handle `gameBtn`.

I'll show the exact changes with minimal disruption to your current file.

1️⃣Add a second button object

Right under your existing:

```
const gameBtn = {
  x: 400,
  y: 550,
  w: 260,
  h: 90,
  label: "It's okay",
};
```

👉 Add this ABOVE it (higher Y = higher on screen → so use a smaller y value):

```
const disappointBtn = {
  x: 400,
  y: 440,            // higher up than 550
  w: 260,
  h: 90,
  label: "It's disappointing",
};
```

2️⃣ Draw BOTH buttons

Inside `drawGame()` you currently have:

```
// ---- Draw the button ----
drawGameButton(gameBtn);
```

Replace with:

```
// ---- Draw both buttons ----
drawGameButton(disappointBtn);
drawGameButton(gameBtn);
```

3️⃣ Update hover cursor to check either button

Change:

```
cursor(isHover(gameBtn) ? HAND : ARROW);
```

to:

```
cursor(isHover(gameBtn) || isHover(disappointBtn) ? HAND : ARROW);
```

4️⃣ Handle clicks for BOTH buttons

Replace your current `gameMousePressed()`:

OLD

```
function gameMousePressed() {
  if (isHover(gameBtn)) {
    triggerRandomOutcome();
  }
}
```

NEW

```
function gameMousePressed() {

  if (isHover(disappointBtn)) {
    clouds++;                  // 👈 example weather effect
    triggerRandomOutcome();
  }
  if (isHover(gameBtn)) {
    sun++;                    // 👈 different effect
    triggerRandomOutcome();
  }
}
```

5️⃣ (Optional) Different keyboard keys

You could let:

- ENTER → “It’s okay”
- D key → “It’s disappointing”

```
function gameKeyPressed() {

  if (keyCode === ENTER) {
    sun++;
    triggerRandomOutcome();
  }

  if (key === 'd') {
    clouds++;
    triggerRandomOutcome();
  }
}
```

✅ Result

You now have:

• two identical-style buttons
• stacked vertically
• independent outcomes
• same hover visuals
• same rounded style

Without changing your architecture at all.

now i have both buttons on the screen, how do i make them each go to a different screen? the screen can be blank for now

Perfect—that’s the next step 👍.
Right now both buttons call:

```
triggerRandomOutcome();
```

which sends you to **win/lose randomly**.
Instead, each button should set **its own screen name directly**.

1️⃣Invent two new screen names

For example:

- `"okayPath"`
- `"disappointPath"`

These can be blank screens for now.

2️⃣Change your button click logic

Find this part in your file:

```
function gameMousePressed() {

  if (isHover(disappointBtn)) {
    clouds++;
    triggerRandomOutcome();
  }

  if (isHover(gameBtn)) {
    sun++;
    triggerRandomOutcome();
  }
}
```

Replace with:

```
function gameMousePressed() {

  if (isHover(disappointBtn)) {
    clouds++;
    currentScreen = "disappointPath";   // 👈 go to its own screen
  }

  if (isHover(gameBtn)) {
    sun++;
    currentScreen = "okayPath";         // 👈 different screen
  }
}
```

👉 Now each button leads somewhere different instead of random.

3️⃣Create the blank screens

In your project you probably have files like:

- win.js
- lose.js
- menu.js

Do the same idea:

Create a new file → disappointPath.js

```
function drawDisappointPath() {
  background(180, 180, 220);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("Disappoint Path", width/2, height/2);
}
```

Create → okayPath.js

```
function drawOkayPath() {
  background(240, 200, 160);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("Okay Path", width/2, height/2);
}
```

4️⃣Tell main.js these screens exist

Inside your main `draw()` switch in **main.js** you probably have something like:

```
if (currentScreen === "game") drawGame();
if (currentScreen === "win") drawWin();
if (currentScreen === "lose") drawLose();
```

👉 ADD:

```
if (currentScreen === "okayPath") drawOkayPath();
if (currentScreen === "disappointPath") drawDisappointPath();
```

✅ Final Logic Flow

• Click “It’s okay”
→ sun++
→ currentScreen = "okayPath"
→ blank okay screen appears

• Click “It’s disappointing”
→ clouds++
→ currentScreen = "disappointPath"
→ different screen

Next Step Ideas

Once this works we can:

- add NEW choices on each path
- make weather carry over
- add a “Back” button
- eventually branch to endings like:

```
game
 ├── okayPath → helpFriend → warmEnding
 └── disappointPath → avoid → rainyEnding
```

If you want, paste your **main.js** and I’ll point to the exact line numbers to add the two new screens 👌

this is main.js: let currentScreen = "start"; // "start" | "instr" | "game" | "win" | "lose" // ------------------------------ // setup() runs ONCE at the beginning // ------------------------------ // This is where you usually set canvas size and initial settings. function setup() { createCanvas(800, 800); // Sets a default font for all text() calls // (This can be changed later per-screen if you want.) textFont("sans-serif"); } // ------------------------------ // draw() runs every frame (many times per second) // ------------------------------ // This is the core “router” for visuals. //

Depending on currentScreen, we call the correct draw function. function draw() { // Each screen file defines its own draw function: // start.js → drawStart() // instructions.js → drawInstr() // game.js → drawGame() // win.js → drawWin() // lose.js → drawLose() if (currentScreen === "start") drawStart(); else if (currentScreen === "instr") drawInstr(); else if (currentScreen === "game") drawGame(); else if (currentScreen === "okayPath") drawOkayPath(); else if (currentScreen === "disappointPath") drawDisappointPath(); // (Optional teaching note) // This “if/else chain” is a very common early approach. // Later in the course you might replace it with: // - a switch statement, or // - an object/map of screens } // ------------------------------ // mousePressed() runs once each time the mouse is clicked // ------------------------------ // This routes mouse input to the correct screen handler. function mousePressed() { // Each screen *may* define a mouse handler: // start.js → startMousePressed() // instructions.js → instrMousePressed() // game.js → gameMousePressed() // win.js → winMousePressed() // lose.js → loseMousePressed() if (currentScreen === "start") startMousePressed(); else if (currentScreen === "instr") instrMousePressed(); else if (currentScreen === "game") gameMousePressed(); // The ?.() means “call this function only if it exists” // This prevents errors if a screen doesn’t implement a handler. else if (currentScreen === "win") winMousePressed?.(); else if (currentScreen === "lose") loseMousePressed?.(); } // ------------------------------ // keyPressed() runs once each time a key is pressed // ------------------------------ // This routes keyboard input to the correct screen handler. function keyPressed() { // Each screen *may* define a key handler: // start.js → startKeyPressed() // instructions.js → instrKeyPressed() // game.js → gameKeyPressed() // win.js → winKeyPressed() // lose.js → loseKeyPressed() if (currentScreen === "start") startKeyPressed(); else if (currentScreen === "instr") instrKeyPressed(); else if (currentScreen === "game") gameKeyPressed?.(); else if (currentScreen === "win") winKeyPressed?.(); else if (currentScreen === "lose") loseKeyPressed?.(); } // -- // Shared helper function: isHover() // -- // // Many screens have buttons. // This helper checks whether the mouse is inside a rectangle. // // Important: our buttons are drawn using rectMode(CENTER), // meaning x,y is the CENTRE of the rectangle. // So we check mouseX and mouseY against half-width/half-height bounds. // // Input: an object with { x, y, w, h } // Output: true if mouse is over the rectangle, otherwise false function isHover({ x, y, w, h }) { return (mouseX > x - w / 2 && // mouse is right of left edge mouseX < x + w / 2 && // mouse is left of right edge mouseY > y - h / 2 && // mouse is below top edge mouseY < y + h / 2 // mouse is above bottom edge); }

Nice—your `main.js` is already set up **almost perfectly** for the two new paths 👌.
You’ve already added:

```
else if (currentScreen === "okayPath") drawOkayPath();
else if (currentScreen === "disappointPath") drawDisappointPath();
```

So the router is ready. We just need to confirm **three things**:

1. The button click sets those screen names
2. The new files exist
3. Optional: allow mouse/keyboard on those screens

✅ 1. Your game.js should now use THIS logic

In **game.js**, make sure your click handler looks like:

```
function gameMousePressed() {

  if (isHover(disappointBtn)) {
    clouds++;
    currentScreen = "disappointPath";   // → goes to that screen
  }

  if (isHover(gameBtn)) {
    sun++;
    currentScreen = "okayPath";         // → goes to other screen
  }
}
```

👉 If this is in place, the routing WILL work.

✅ 2. Create the two screen files

📁 okayPath.js

```
function drawOkayPath() {
  background(240, 200, 160);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Okay Path", width/2, height/2);
}

// optional click to return to game
function okayPathMousePressed() {
  currentScreen = "game";
}
```

📁 disappointPath.js

```
function drawDisappointPath() {
  background(180, 180, 220);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Disappoint Path", width/2, height/2);
}

function disappointPathMousePressed() {
  currentScreen = "game";
}
```

✅ 3. (Optional) Add input routing in main.js

Right now your mouse router only includes:

```
else if (currentScreen === "game") gameMousePressed();
```

If you want clicks to work on the new screens too, add:

```
else if (currentScreen === "okayPath") okayPathMousePressed?.();
else if (currentScreen === "disappointPath") disappointPathMousePressed?.();
```

Same idea for keyboard if you need it.

🧪 What Should Happen Now

1. Start game
2. You see TWO buttons
3. Click:

“It’s okay”
→ currentScreen = "okayPath"
→ drawOkayPath() runs
→ you see the okay screen

“It’s disappointing”
→ currentScreen = "disappointPath"
→ different screen appears

No more win/lose randomness 🎉.

Next Level (when ready)

From here we can:

- Add NEW choices on each path
- Carry weather variables across
- Create:

```
game
 ├── okayPath → help → sunnyEnding
 └── disappointPath → avoid → rainyEnding
```

- Add a BACK button component you can reuse everywhere.

If it’s not switching yet 👉 paste your **current game.js** and I’ll spot the exact line causing it.

this is my game.js:// NOTE: Do NOT add setup() or draw() in this file // setup() and draw() live in main.js // This file only defines: // 1) drawGame() → what the game screen looks like // 2) input handlers → what happens when the player clicks or presses keys // 3) helper functions specific to this screen // ------------------------------ // Button data // ------------------------------ // This object stores all the information needed to draw // and interact with the button on the game screen. // Keeping this in one object makes it easier to move, // resize, or restyle the button later. const disappointBtn = { x: 400, y: 440, // higher up than 550 w: 260, h: 90, label: "It's disappointing", }; const gameBtn = { x: 400, // x position (centre of the button) y: 550, // y position (centre of the button) w: 260, // width h: 90, // height label: "It's okay", // text shown on the button }; // ------------------------------ // Main draw function for this screen // ------------------------------ // drawGame() is called from main.js *only* // when currentScreen === "game" function drawGame() { // Set background colour for the game screen background(240, 230, 140); // ---- Title and instructions text ---- fill(0); // black text textSize(32); textAlign(CENTER, CENTER); text("Game Screen", width / 2, 160); textSize(18); text("Your friend cancels plans. How do you feel?", width / 2, 210); // ---- Draw the button ---- // We pass the button object to a helper function // ---- Draw both buttons ---- drawGameButton(disappointBtn); drawGameButton(gameBtn); // ---- Cursor feedback ---- // If the mouse is over the button, show a hand cursor // Otherwise, show the normal arrow cursor cursor(isHover(gameBtn) || isHover(disappointBtn) ? HAND : ARROW); } // -------------

----------------- // Button drawing helper // ------------------------------ // This function is responsible *only* for drawing the button. // It does NOT handle clicks or game logic. function drawGameButton({ x, y, w, h, label }) { rectMode(CENTER); // Check if the mouse is hovering over the button // isHover() is defined in main.js so it can be shared const hover = isHover({ x, y, w, h }); noStroke(); // Change button colour when hovered // This gives visual feedback to the player fill(hover ? color(180, 220, 255, 220) // lighter blue on hover : color(200, 220, 255, 190), // normal state); // Draw the button rectangle rect(x, y, w, h, 14); // last value = rounded corners // Draw the button text fill(0); textSize(28); textAlign(CENTER, CENTER); text(label, x, y); } // ------------------------------ // Mouse input for this screen // ------------------------------ // This function is called from main.js // only when currentScreen === "game" function gameMousePressed() { if (isHover(disappointBtn)) { clouds++; currentScreen = "disappointPath"; // → goes to that screen } if (isHover(gameBtn)) { sun++; currentScreen = "okayPath"; // → goes to other screen } } // ------------------------------ // Keyboard input for this screen // ------------------------------ // Allows keyboard-only interaction (accessibility + design) function gameKeyPressed() { // ENTER key triggers the same behaviour as clicking the button if (keyCode === ENTER) { triggerRandomOutcome(); } } // ------------------------------ // Game logic: win or lose // ------------------------------ // This function decides what happens next in the game. // It does NOT draw anything. function triggerRandomOutcome() { // random() returns a value between 0 and 1 // Here we use a 50/50 chance: // - less than 0.5 → win // - 0.5 or greater → lose // // You can bias this later, for example: // random() < 0.7 → 70% chance to win if (random() < 0.5) { currentScreen = "win"; } else { currentScreen = "lose"; } }

Nice—your `game.js` looks almost perfect 👍.
I can see **exactly why it might feel confusing right now**:

✅ The button routing part is CORRECT

This section is already doing the right thing:

```
function gameMousePressed() {
  if (isHover(disappointBtn)) {
    clouds++;
    currentScreen = "disappointPath";
  }

  if (isHover(gameBtn)) {
    sun++;
    currentScreen = "okayPath";
  }
}
```

👉 So the problem is NOT here.

🚨 The Real Issue: ENTER key still uses OLD logic

Your keyboard handler still calls:

```
function gameKeyPressed() {
  if (keyCode === ENTER) {
    triggerRandomOutcome();   // ← OLD win/lose system
  }
}
```

So:

- Clicking → goes to new paths ✅
- Pressing ENTER → still goes win/lose ❌

Fix Option A — Best Choice

Make ENTER activate the **"It's okay"** button:

```
function gameKeyPressed() {
  if (keyCode === ENTER) {
    sun++;
    currentScreen = "okayPath";
  }

  if (key === 'd') {
    clouds++;
    currentScreen = "disappointPath";
  }
}
```

Now keyboard matches mouse behaviour 💯.

🚮 You Can Delete This Whole Function Later

Once you fully move away from win/lose, this is unused:

```
function triggerRandomOutcome() { ... }
```

It’s part of the template but your story doesn’t need it anymore.

✅ What SHOULD happen now

Click “It’s okay”

→ currentScreen = "okayPath"
→ main.js runs
→ drawOkayPath()

Click “It’s disappointing”

→ currentScreen = "disappointPath"
→ drawDisappointPath()

If It’s Still Not Showing

99% chance one of these:

1. ❗ The files don’t exist:

- okayPath.js
- disappointPath.js

2. ❗ The function names inside them don’t match:

Must be EXACT:

```
function drawOkayPath() { }
function drawDisappointPath() { }
```

3. ❗ They aren’t linked in index.html

👉 Can you paste:

- your okayPath.js
- your disappointPath.js

I'll verify in 5 seconds what's missing 👌

// NOTE: Do NOT add setup() or draw() in this file // setup() and draw() live in main.js // This file only defines: // 1) drawWin() → what the win screen looks like // 2) input handlers → how the player returns to the start screen // // This file is intentionally very similar to lose.js. // The goal is to show that win/lose screens are often // simple "end states" with minimal logic. // -- // Main draw function for win screen // -- // drawWin() is called from main.js // only when currentScreen === "win" function drawDisappointPath() { background(180, 100, 220); fill(0); textAlign(CENTER, CENTER); textSize(32); text("Disappoint Path", width / 2, height / 2); } function disappointPathMousePressed() { currentScreen = "game"; }

Nice—this file looks good 👍.
And your `drawDisappointPath()` name matches what `main.js` is calling:

```
else if (currentScreen === "disappointPath") drawDisappointPath();
```

So this screen is 100% valid.

The ONLY remaining checklist

Since the code itself is correct, the issue can only be one of these:

1️⃣Do you also have an `okayPath.js` file?

You showed only the disappoint one.
You also need:

```
function drawOkayPath() {
  background(240, 200, 160);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Okay Path", width / 2, height / 2);
}

function okayPathMousePressed() {
  currentScreen = "game";
}
```

2️⃣ Are both files linked in index.html?

In p5 projects, every new file must be added to:

📁 index.html

You should see something like:

```
<script src="game.js"></script>
<script src="win.js"></script>
<script src="lose.js"></script>
```

👉 You MUST add:

```
<script src="okayPath.js"></script>
<script src="disappointPath.js"></script>
```

If they aren’t there → p5 never loads them → screen stays blank.

3️⃣ Quick Test to Prove Routing Works

Temporarily add this at top of gameMousePressed:

```
console.log("clicked disappointing");
```

Open browser console → click button.

If you see the message → your click logic is firing.

🌟 Your Architecture Is Now Officially a Branching Story

You now have:

```
game screen
 ├── disappointPath
```

```
  └── okayPath
```

Next we can:

- add new buttons inside each path
- carry weather values forward
- create endings like:

```
okayPath → help → sunnyEnding
disappointPath → avoid → rainyEnding
```

👉 Next step?

Want me to help you:

A) Add a SECOND choice on the Disappoint screen
B) Show weather visuals on those screens
C) Add a reusable "Back" button component

Tell me your goal for tomorrow's version 😄

is my main.js correct: // -- // main.js = the "router" (traffic controller) for the whole game // -- // // Idea: this project has multiple screens (start, instructions, game, win, lose). // Instead of putting everything in one giant file, each screen lives in its own // file and defines two main things: // 1) drawX() → how that screen looks // 2) XMousePressed() / XKeyPressed() → how that screen handles input // // This main.js file does 3 important jobs: // A) stores the current screen in a single shared variable // B) calls the correct draw function each frame // C) sends mouse/keyboard input to the correct screen handler // ------------------------------ // Global game state // ------------------------------ // This variable is shared across all files because all files run in the same // global JavaScript scope when loaded in index.html. // // We store the "name" of the current screen as a string. // Only one screen should be active at a time. let currentScreen = "start"; // "start" | "instr" | "game" | "win" | "lose" // ------------------------------ // setup() runs ONCE at the beginning // ------------------------------ // This is where you usually set canvas size and initial settings. function setup() { createCanvas(800, 800); // Sets a default font for all text() calls // (This can be changed later per-screen if you want.) textFont("sans-serif"); } // ------------------------------ // draw() runs every frame (many times per second) // ------------------------------ // This is the core "router" for visuals. // Depending on currentScreen, we call the correct draw function. function draw() { // Each screen file defines its own draw function: // start.js → drawStart() // instructions.js → drawInstr() // game.js → drawGame() // win.js → drawWin() // lose.js →

```
drawLose() if (currentScreen === "start") drawStart(); else if (currentScreen === "instr")
drawInstr(); else if (currentScreen === "game") drawGame(); else if (currentScreen ===
"okayPath") drawOkayPath(); else if (currentScreen === "disappointPath")
drawDisappointPath(); // (Optional teaching note) // This “if/else chain” is a very common
early approach. // Later in the course you might replace it with: // - a switch statement, or //
- an object/map of screens } // ------------------------------ // mousePressed() runs once each
time the mouse is clicked // ------------------------------ // This routes mouse input to the
correct screen handler. function mousePressed() { // Each screen *may* define a mouse
handler: // start.js → startMousePressed() // instructions.js → instrMousePressed() //
game.js → gameMousePressed() // win.js → winMousePressed() // lose.js →
loseMousePressed() if (currentScreen === "start") startMousePressed(); else if
(currentScreen === "instr") instrMousePressed(); else if (currentScreen === "game")
gameMousePressed(); // The ?.() means “call this function only if it exists” // This prevents
errors if a screen doesn’t implement a handler. else if (currentScreen === "okayPath")
okayPathMousePressed?.(); else if (currentScreen === "disappointPath")
disappointPathMousePressed?.(); } // ------------------------------ // keyPressed() runs once
each time a key is pressed // ------------------------------ // This routes keyboard input to the
correct screen handler. function keyPressed() { // Each screen *may* define a key handler:
// start.js → startKeyPressed() // instructions.js → instrKeyPressed() // game.js →
gameKeyPressed() // win.js → winKeyPressed() // lose.js → loseKeyPressed() if
(currentScreen === "start") startKeyPressed(); else if (currentScreen === "instr")
instrKeyPressed(); else if (currentScreen === "game") gameKeyPressed?.(); else if
(currentScreen === "win") winKeyPressed?.(); else if (currentScreen === "lose")
loseKeyPressed?.(); } // ------------------------------------------------------------ // Shared helper
function: isHover() // ------------------------------------------------------------ // // Many screens
have buttons. // This helper checks whether the mouse is inside a rectangle. // // Important:
our buttons are drawn using rectMode(CENTER), // meaning x,y is the CENTRE of the
rectangle. // So we check mouseX and mouseY against half-width/half-height bounds. // //
Input: an object with { x, y, w, h } // Output: true if mouse is over the rectangle, otherwise
false function isHover({ x, y, w, h }) { return ( mouseX > x - w / 2 && // mouse is right of left
edge mouseX < x + w / 2 && // mouse is left of right edge mouseY > y - h / 2 && // mouse is
below top edge mouseY < y + h / 2 // mouse is above bottom edge ); }
```

Yes — your **main.js is correct** for the new branching system 👍.
You’ve integrated the two new paths exactly the right way.

I’ll go through it like a TA would and point out:

✅ what’s perfect
⚠️ tiny inconsistencies (not errors, just clarity)
💡 optional improvements

✅ 1. Router Logic – 100% Correct

```
if (currentScreen === "start") drawStart();
else if (currentScreen === "instr") drawInstr();
else if (currentScreen === "game") drawGame();
else if (currentScreen === "okayPath") drawOkayPath();
else if (currentScreen === "disappointPath") drawDisappointPath();
```

👉 This matches exactly:

- your game.js buttons
- your new screen files

So navigation flow will work.

✅ 2. Mouse Routing – Correct

```
else if (currentScreen === "okayPath") okayPathMousePressed?.();
else if (currentScreen === "disappointPath") disappointPathMousePressed?.();
```

Perfect use of:

- optional chaining `?.()`
- separate handlers per screen

⚠️ 3. Small Comment Mismatch (not functional)

At the top you still wrote:

```
let currentScreen = "start"; // "start" | "instr" | "game" | "win" | "lose"
```

But you NOW have:

- okayPath
- disappointPath

👉 Update comment to:

```
let currentScreen = "start";
// "start" | "instr" | "game" | "okayPath" | "disappointPath" | "win" |
"lose"
```

Not required for code—just documentation accuracy.

⚠️ 4. Keyboard Router Doesn’t Include New Paths

Right now:

```
else if (currentScreen === "win") winKeyPressed?.();
else if (currentScreen === "lose") loseKeyPressed?.();
```

You might later want:

```
else if (currentScreen === "okayPath") okayPathKeyPressed?.();
else if (currentScreen === "disappointPath") disappointPathKeyPressed?.();
```

Not required yet—but good for consistency.

💡 5. Architecture Review (You Did This Well)

Your project now follows a REAL interactive narrative pattern:

```
main.js  → traffic controller
game.js → choice hub
okayPath.js → branch A
disappointPath.js → branch B
```

This is exactly how professional point-and-click stories are structured.

🚀 You’re Ready for Next Steps

From here we can:

1. Add NEW buttons inside each path
2. Carry weather variables forward
3. Create real endings:

```
game
 ├── okayPath
 │      └── helpFriend → sunnyEnding
 └── disappointPath
        └── avoid → rainyEnding
```

Want me to help you build:

A) A reusable button system so you don’t repeat code?
B) Weather visuals appearing on all screens?
C) A third option like “Are you safe?”

Tell me the vibe you want:

- emotional story
- funny
- realistic school drama
- abstract artsy

everything works except for clicking the its okay or dissappointing button nothing happens

Got it — if **everything else works but clicking those two buttons does nothing**, 99% this is one of three things:

1. `gameMousePressed()` is not being called
2. `isHover()` isn’t detecting the buttons
3. another screen is blocking input

Let’s debug step-by-step.

🔎 1. First Test – Is the click function running?

Add a console log at the TOP of `gameMousePressed()`:

```
function gameMousePressed() {
  console.log("clicked on game screen");   // ← ADD THIS

  if (isHover(disappointBtn)) {
    clouds++;
    currentScreen = "disappointPath";
  }

  if (isHover(gameBtn)) {
    sun++;
    currentScreen = "okayPath";
  }
}
```

👉 Then:

- open browser console (F12 → Console)
- click anywhere on the game screen

RESULT A – You see the message

That means:

✅ gameMousePressed is connected
❌ problem is with isHover()

RESULT B – You see NOTHING

👉 main.js is NOT calling gameMousePressed.

Then problem is here:

In main.js you have:

```
else if (currentScreen === "game") gameMousePressed();
```

Check:

👉 Is `currentScreen` ACTUALLY equal to `"game"` when you see the buttons?

Add in drawGame():

```
function drawGame() {
  console.log("currentScreen:", currentScreen);
```

If it prints something like:

- "Game"
- " GAME "
- "game "

→ string mismatch = clicks ignored.

🔎 2. Most Common Cause: rectMode

Your hover uses CENTER mode:

```
function isHover({ x, y, w, h }) {
  return (
    mouseX > x - w / 2 &&
    mouseX < x + w / 2 &&
    mouseY > y - h / 2 &&
    mouseY < y + h / 2
  );
}
```

✅ Correct for CENTER

And your buttons ARE drawn with:

```
rectMode(CENTER);
```

So that matches — good.

🔎 3. Another BIG possibility

👉 You might still be on the OLD “win/lose” version of the project in the browser cache.

Try:

• hard refresh
• or run in a new tab
• or add:

```
console.log("game.js loaded");
```

at top of game.js to confirm the new file is actually running.

🔧 Quick Fix Test

Replace your entire click function TEMPORARILY with:

```
function gameMousePressed() {
  console.log("CLICK REGISTERED");

  currentScreen = "okayPath";
}
```

👉 If THIS still doesn't switch screens →
problem is NOT buttons — it's routing.

👉 Please tell me:

When you add:

```
console.log("clicked on game screen");
```

and click:

1. Do you see that message?
2. Or completely nothing?

That answer will pinpoint the bug in 10 seconds 😄

The issue was the clouds++ and sun++, how do I add a simple button that returns to the menu in the top right corner?

Ahhh perfect — that makes sense 👍
(`clouds` / `sun` weren't defined yet, so the click handler was erroring and stopping execution.)

Let's add a **simple, clean "Back to Menu" button** in the **top-right corner** that you can reuse on any screen.

I'll keep this **minimal and safe** so it won't break anything.

✅ Goal

- Small button
- Top-right corner
- Click → `currentScreen = "start"`
- Works on any screen (okayPath, disappointPath, etc.)

1️⃣ Create a reusable menu button object

You can put this in **each screen file**, or (better) copy it where needed.

Example (top-right corner):

```
const menuBtn = {
  x: 700,
  y: 60,
  w: 140,
  h: 50,
  label: "Menu",
};
```

(800px canvas → 700 keeps it inset nicely)

2️⃣ Draw the button (reuse your existing style)

You can reuse `drawGameButton()` or make a tiny helper.

Inside `drawOkayPath()` (or any screen):

```
function drawOkayPath() {
  background(240, 200, 160);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Okay Path", width / 2, height / 2);

  drawGameButton(menuBtn);   // 👈 draw menu button
  cursor(isHover(menuBtn) ? HAND : ARROW);
}
```

This works because `drawGameButton()` is global.

3️⃣ Handle the click

Add this to the SAME file:

```
function okayPathMousePressed() {
  if (isHover(menuBtn)) {
    currentScreen = "start";
  }
}
```

That’s it 🎉.

4️⃣ Repeat for Disappoint Path

In `disappointPath.js`:

```
const menuBtn = {
  x: 700,
  y: 60,
  w: 140,
  h: 50,
  label: "Menu",
};

function drawDisappointPath() {
  background(180, 100, 220);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Disappoint Path", width / 2, height / 2);

  drawGameButton(menuBtn);
  cursor(isHover(menuBtn) ? HAND : ARROW);
}

function disappointPathMousePressed() {
  if (isHover(menuBtn)) {
    currentScreen = "start";
  }
}
```

✅ Result

- Click choice
- Go to path
- Click **Menu**
- Return to start screen
- No errors
- Clean UX

💡 Tiny Upgrade (Optional, looks polished)

Change label to:

- `"← Menu"`
- `"Back"`
- `"Home"`

Or shrink it slightly:

```
w: 110,
h: 40,
```

🚀 Next smart step

If you want, I can help you:

- Make ONE global menu button (no duplication)
- Add hover animation
- Disable menu during transitions
- Show weather state on menu

Just tell me what your prof would care about most 😄

for the disappointing path, how can i add raindrops on the screen?

Nice choice — rain instantly sells the *“disappointing”* mood 🌧️
We’ll do this in a **simple, safe, student-friendly way** that won’t mess with your screen system.

I’ll show:

1. A super-simple raindrop system

2. How to draw it only on the disappointing path
3. How to animate it
4. One tiny polish upgrade (optional)

☔ Step 1: Create raindrop data

At the **top of `disappointPath.js`** (outside any function):

```
let raindrops = [];
```

Then add a setup helper **inside this file**:

```
function initRain() {
  raindrops = [];

  for (let i = 0; i < 120; i++) {
    raindrops.push({
      x: random(width),
      y: random(-height, 0),
      speed: random(3, 7),
      len: random(8, 16),
    });
  }
}
```

☔ Step 2: Initialize rain ONCE

We don’t want to reset rain every frame.

Add this flag:

```
let rainInitialized = false;
```

Now update `drawDisappointPath()`:

```
function drawDisappointPath() {

  if (!rainInitialized) {
    initRain();
    rainInitialized = true;
  }

  background(180, 100, 220);

  drawRain();

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Disappoint Path", width / 2, height / 2);

  drawGameButton(menuBtn);
  cursor(isHover(menuBtn) ? HAND : ARROW);
}
```

☔ Step 3: Draw & animate the rain

Add this helper below:

```
function drawRain() {
  stroke(200, 220, 255);
  strokeWeight(2);

  for (let drop of raindrops) {
    line(drop.x, drop.y, drop.x, drop.y + drop.len);
    drop.y += drop.speed;

    if (drop.y > height) {
      drop.y = random(-100, -10);
      drop.x = random(width);
    }
  }
}
```

That’s it. You now have animated rain 🎉.

✅ What This Gives You

- falling rain
- loops naturally
- independent of game logic
- only appears on disappointing path
- zero interaction bugs

✨ Optional Polish (Easy)

Make rain slanted (windy sadness):

```
line(drop.x, drop.y, drop.x + 3, drop.y + drop.len);
```

Darken screen slightly for mood:

```
background(140, 90, 180);
```

Rain slows over time (calming):

```
drop.speed *= 0.999;
```

🧠 Design Bonus (Prof-friendly explanation)

You can describe this as:

“The environment reacts emotionally to the player’s choice, using weather as visual feedback.”

If you want next:

- thunder flashes
- sound effects
- rain intensity based on choices
- clouds drifting

Just say the word 🌧️

how do i add still raindrops, they don't have to be moving just something simple

Perfect — **still raindrops** are actually *cleaner* and very “artsy,” and they’re way easier to explain for an assignment 👍
No animation, no state flags, no bugs.

Here’s the **simplest, safest way**.

☔ Option 1: Random still raindrops (recommended)

1️⃣In `disappointPath.js`, add this helper

```
function drawStillRain() {
  stroke(200, 220, 255);
  strokeWeight(2);

  for (let i = 0; i < 120; i++) {
    let x = random(width);
    let y = random(height);
    line(x, y, x, y + 12);
  }
}
```

2⃣Call it in `drawDisappointPath()`

```
function drawDisappointPath() {
  background(180, 100, 220);

  drawStillRain();   // 👈 add this line

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Disappoint Path", width / 2, height / 2);

  drawGameButton(menuBtn);
  cursor(isHover(menuBtn) ? HAND : ARROW);
}
```

✅ Done. You now have static rain texture.

⚠️ Important Note (Visual Detail)

Because `random()` runs every frame, the raindrops will **flicker slightly**.

If that’s OK (many people like the texture look), you’re done.

If you want **truly still rain** → Option 2.

☔ Option 2: Truly still raindrops (no flicker)

1⃣Store raindrop positions once

At top of `disappointPath.js`:

```
let stillRain = [];
let rainSet = false;
```

2⃣Initialize once

```
function initStillRain() {
  stillRain = [];

  for (let i = 0; i < 120; i++) {
    stillRain.push({
      x: random(width),
      y: random(height),
    });
  }
}
```

3️⃣ Draw them

```
function drawStillRain() {
  stroke(200, 220, 255);
  strokeWeight(2);

  for (let drop of stillRain) {
    line(drop.x, drop.y, drop.x, drop.y + 12);
  }
}
```

4️⃣ Use it in your screen

```
function drawDisappointPath() {

  if (!rainSet) {
    initStillRain();
    rainSet = true;
  }

  background(180, 100, 220);
  drawStillRain();

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Disappoint Path", width / 2, height / 2);

  drawGameButton(menuBtn);
  cursor(isHover(menuBtn) ? HAND : ARROW);
}
```

✅ Which Should You Use?

Goal	Use
Fastest	Option 1
Clean visuals	Option 2

Goal	Use
Less code	Option 1
“Intentional design”	Option 2

For a class project, **Option 2** looks more deliberate.

💡 Easy Polish Ideas (Optional)

- shorter drops: `+ 8`
- darker rain: `stroke(160, 180, 220)`
- fade background slightly
- diagonal rain for emotion

If you want, next we can:

- add clouds to the okay path
- add fog
- add sound
- add weather accumulation

Just tell me 🌧️

on the okay page, how can i add one big sun on the page, just a yellow circle with the rectangles surrounding it

Nice — this will read **instantly** as the emotional opposite of rain ☀️
We’ll keep it **simple, bold, and very “p5 assignment–safe.”**

I’ll show you:

1. One big sun (yellow circle)
2. Simple rectangle rays around it
3. No animation, no state, no bugs

☀️ Step 1: Add a sun drawing helper

In **`okayPath.js`**, add this function:

```
function drawSun(x, y, r) {
  // Sun rays
  rectMode(CENTER);
  noStroke();
  fill(255, 200, 60);

  for (let i = 0; i < 12; i++) {
    let angle = TWO_PI / 12 * i;
    let rayX = x + cos(angle) * (r + 30);
    let rayY = y + sin(angle) * (r + 30);

    push();
    translate(rayX, rayY);
    rotate(angle);
    rect(0, 0, 12, 40);
    pop();
  }

  // Sun body
  fill(255, 220, 80);
  ellipse(x, y, r * 2);
}
```

☀️ Step 2: Call it in `drawOkayPath()`

```
function drawOkayPath() {
  background(240, 200, 160);

  drawSun(width / 2, height / 2 - 40, 80); // 👈 big sun

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("Okay Path", width / 2, height / 2 + 120);

  drawGameButton(menuBtn);
  cursor(isHover(menuBtn) ? HAND : ARROW);
}
```

✅ Result

- Large yellow sun
- Rectangle rays evenly spaced
- Clean, flat illustration

- Strong mood contrast vs rain
- Zero interactivity bugs

✨ Optional Tiny Tweaks

Want it chunkier?

```
rect(0, 0, 18, 50);
```

More rays?

```
for (let i = 0; i < 16; i++)
```

Softer background?

```
background(255, 230, 180);
```

🧠 How to Explain This in Class

“The environment visually reflects emotional states using simple symbolic weather forms.”

Your prof will eat that up 😄.

If you want next:

- subtle glow
- moving rays
- sun following mouse
- clouds drifting in later choices

Just say the vibe ☀️

how do i just lower all of that to move it closer to the bottom of the page

Easy 👍 — you don’t need to rewrite anything.
You just **shift the Y positions** (or use a single offset).

I’ll show you the **cleanest way** so you can move the whole sun + text together.

✅ Best Method: Use a single vertical offset

1️⃣Add an offset variable

At the top of `okayPath.js` (or inside `drawOkayPath`):

```
let sunYOffset = 120;   // increase this to move DOWN
```

2️⃣Apply it when drawing

```
function drawOkayPath() {
  background(240, 200, 160);

  drawSun(
    width / 2,
    height / 2 - 40 + sunYOffset,   // 👈 moved down
    80
  );

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(32);
  text(
    "Okay Path",
    width / 2,
    height / 2 + 120 + sunYOffset   // 👈 moved down
  );

  drawGameButton(menuBtn);
  cursor(isHover(menuBtn) ? HAND : ARROW);
}
```

🔧 Why this is the right way

- One number controls everything
- Easy to tweak
- No magic numbers everywhere
- Looks intentional in code review

Just increase `sunYOffset`:

```
sunYOffset = 180;
```

→ moves further down
Decrease → moves up.

🚫 Avoid This (messy later)

❌ Hard-coding new values everywhere
❌ Rewriting `drawSun()`
❌ Moving individual rays manually

🧠 Bonus (cleaner pattern)

You can also do:

```
push();
translate(0, 120);
drawSun(width/2, height/2 - 40, 80);
text("Okay Path", width/2, height/2 + 120);
pop();
```

Same idea, same result.

If you want next:

- center it dynamically for different screen sizes
- animate it rising
- add a glow
- add clouds drifting in later

Just tell me 🌞

how do i create a next button on the screen that takes you to another page the same as the game.js but with a different prompt on it. I am assuming I can just create that new button and copy the game.js and change the words on them

You're thinking in **exactly the right direction** 👍
Yes — the clean, *correct* way at your level is:

Duplicate the screen, change the prompt + button outcomes.

That's not "lazy" — that's **proper screen-based design**.

I'll show you the **clean pattern** so it doesn't turn messy later.

✅ What You’re Building

You already have:

```
start → game (choice 1)
```

You want:

```
game (choice 1)
    └── Next → game2 (choice 2)
```

Where **game2** looks like `game.js` but asks something different.

🧱 Step 1: Create a new screen file

📁 game2.js

Yes — copying `game.js` is totally fine.

✏️ game2.js (minimal example)

```
// NOTE: No setup() or draw() here

const nextBtn = {
  x: 400,
  y: 620,
  w: 260,
  h: 90,
  label: "Next",
};

const choiceBtn1 = {
  x: 400,
  y: 440,
  w: 260,
  h: 90,
  label: "Speak up",
};

const choiceBtn2 = {
  x: 400,
  y: 550,
  w: 260,
  h: 90,
  label: "Stay quiet",
};

function drawGame2() {
  background(210, 230, 255);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("Later that day, you run into them again.", width / 2, 200);

  drawGameButton(choiceBtn1);
  drawGameButton(choiceBtn2);
  drawGameButton(nextBtn);

  cursor(
    isHover(choiceBtn1) ||
    isHover(choiceBtn2) ||
    isHover(nextBtn)
      ? HAND
      : ARROW
  );
}

function game2MousePressed() {

  if (isHover(choiceBtn1)) {
    currentScreen = "okayPath";
  }

  if (isHover(choiceBtn2)) {
    currentScreen = "disappointPath";
  }

  if (isHover(nextBtn)) {
    currentScreen = "game"; // or game3 later
  }
}
```

🔁 Step 2: Add it to main.js

In `draw()`:

```
else if (currentScreen === "game2") drawGame2();
```

In `mousePressed()`:

```
else if (currentScreen === "game2") game2MousePressed?.();
```

🌐 Step 3: Link it from your first game screen

In **game.js**, add a “Next” button:

```
const nextBtn = {
  x: 700,
  y: 700,
  w: 140,
  h: 50,
  label: "Next",
};
```

Draw it:

```
drawGameButton(nextBtn);
```

Handle click:

```
if (isHover(nextBtn)) {
  currentScreen = "game2";
}
```

✅ Yes — Copying game.js Is OK

For a class project, this is **exactly what instructors expect**:

- each screen in its own file
- similar structure
- different narrative content

Later you can refactor, but **right now clarity > abstraction**.

🧠 How to Explain This in Class

“Each decision point is its own screen, allowing the narrative to branch cleanly while reusing shared UI logic.”

That’s a strong explanation.

🚀 Next Steps (when ready)

You can now:

- add game3, game4
- branch deeper
- reuse `drawGameButton()` everywhere
- add a progress indicator
- show weather accumulating

If you want, send me:

- your next prompt text
- how many choices
- where it should branch

I'll scaffold the exact file for you 🔥

how can i just simply add a button to the okay screen that says next and takes you to game2.js which i just copy and pasted from game.js

Yep — you've got the right instinct.
This is the **simplest possible way** to do it, no over-engineering, no refactors.

I'll show you **exactly** what to add and **where**, assuming:

- `game2.js` already exists
- `drawGame2()` + `game2MousePressed()` are defined
- `game2.js` is linked in `index.html`

✅ Step 1: Add a "Next" button to `okayPath.js`

At the top of `okayPath.js`

```
const nextBtn = {
  x: 700,
  y: 700,
  w: 140,
  h: 50,
  label: "Next",
};
```

✅ Step 2: Draw it on the Okay screen

Inside `drawOkayPath()`:

```
drawGameButton(nextBtn);
```

And update the cursor line:

```
cursor(
  isHover(menuBtn) || isHover(nextBtn)
    ? HAND
    : ARROW
);
```

✅ Step 3: Handle the click

In `okayPathMousePressed()`:

```
function okayPathMousePressed() {

  if (isHover(menuBtn)) {
    currentScreen = "start";
  }

  if (isHover(nextBtn)) {
    currentScreen = "game2";    // 👈 THIS is the key line
  }
}
```

✅ Step 4: Make sure `main.js` knows about game2

In `draw()`:

```
else if (currentScreen === "game2") drawGame2();
```

In `mousePressed()`:

```
else if (currentScreen === "game2") game2MousePressed?.();
```

That's it. No extra logic needed 🎉.

🔎 If clicking “Next” does nothing

Check **only these 3 things**:

1☐ `game2.js` has:

```
function drawGame2() { ... }
```

2☐ `index.html` includes:

```
<script src="game2.js"></script>
```

3☐ Spelling matches EXACTLY:

```
currentScreen = "game2";
```

🧠 Why This Is the Right Approach

- Clear screen transitions
- Easy to debug
- Matches your existing architecture
- Totally acceptable to copy/paste for assignments

You’re building a **real state-based interactive narrative** now.

If you want next:

- a progress indicator
- disable Next until a choice is made
- animated transitions
- branching from game2

Tell me what you want to add and I’ll guide you step-by-step 👍

the button is there but doesnt do anything when i click it, this is my okay file: const nextBtn = { x: 700, y: 700, w: 140, h: 50, label: "Next", }; function drawOkayPath() { background(100, 190, 220); drawGameButton(nextBtn); drawSun(width / 2, height / 2 - 175, 80); fill(0); textAlign(CENTER, CENTER); textSize(32); text("That's okay :)", width / 2, height / 2); cursor(isHover(menuBtn) || isHover(nextBtn) ? HAND : ARROW); } // optional click to return to game function okayPathMousePressed() { if (isHover(menuBtn)) { currentScreen =

"start"; } if (isHover(nextBtn)) { currentScreen = "game2"; } } function drawSun(x, y, r) { // Sun rays rectMode(CENTER); noStroke(); fill(255, 200, 60); for (let i = 0; i < 12; i++) { let angle = (TWO_PI / 12) * i; let rayX = x + cos(angle) * (r + 30); let rayY = y + sin(angle) * (r + 30); push(); translate(rayX, rayY); rotate(angle); rect(0, 0, 12, 40); pop(); } // Sun body fill(255, 220, 80); ellipse(x, y, r * 2); }

Good news: your **Okay screen code is basically correct** 👍
When a button *draws* but *does nothing*, it almost always means **the mouse handler isn't being called**, not that the button logic is wrong.

Let's pinpoint it cleanly.

✅ Your okayPath.js logic is fine

This part is **correct**:

```
if (isHover(nextBtn)) {
  currentScreen = "game2";
}
```

So the problem is **not** the button, **not** isHover, **not** drawGameButton.

🚨 The real issue: `main.js` is not routing clicks to okayPath

For this function to run:

```
function okayPathMousePressed() { ... }
```

👉 **main.js MUST explicitly call it** when `currentScreen === "okayPath"`.

🔧 Fix: Check `mousePressed()` in main.js

You need THIS line:

```
else if (currentScreen === "okayPath") okayPathMousePressed?.();
```

Your `mousePressed()` should look like this (relevant part):

```
function mousePressed() {
  if (currentScreen === "start") startMousePressed();
  else if (currentScreen === "instr") instrMousePressed();
  else if (currentScreen === "game") gameMousePressed();
  else if (currentScreen === "okayPath") okayPathMousePressed?.();   // ✅
REQUIRED
  else if (currentScreen === "disappointPath")
disappointPathMousePressed?.();
  else if (currentScreen === "game2") game2MousePressed?.();
}
```

If that line is missing → clicks are ignored.

🧪 Quick 5-second test (do this once)

Add this temporarily inside `okayPathMousePressed()`:

```
function okayPathMousePressed() {
  console.log("okay path clicked");

  if (isHover(nextBtn)) {
    currentScreen = "game2";
  }
}
```

Click anywhere on the Okay screen.

✅ If you see the log

Routing works → check game2 setup.

❌ If you see nothing

`okayPathMousePressed()` is NOT being called → main.js routing issue.

🔎 If it still doesn't switch after that

Check these **three exact things** (no guessing):

1️⃣game2 exists in draw router

```
else if (currentScreen === "game2") drawGame2();
```

2️⃣ game2 exists in mouse router

```
else if (currentScreen === "game2") game2MousePressed?.();
```

3️⃣ game2.js is linked in index.html

```
<script src="game2.js"></script>
```

🧠 Why this keeps happening (important insight)

Buttons **never do anything by themselves** in p5.

They only work if **all 3 layers are connected**:

1. drawn
2. hover detected
3. mouse handler routed from main.js

You nailed #1 and #2 — this is just #3.

✅ Once fixed, your flow will be:

```
Okay Path
 └── click Next
      └── currentScreen = "game2"
           └── drawGame2()
```

That's exactly how a real interactive story engine works.

If it *still* doesn't switch after adding that line, paste your **mousePressed() from main.js** and I'll spot the issue instantly 👌

this is my game2 file: const disappointBtn = { x: 400, y: 440, // higher up than 550 w: 260, h: 90, label: "It's disappointing", }; const gameBtn = { x: 400, // x position (centre of the button) y: 550, // y position (centre of the button) w: 260, // width h: 90, // height label: "It's okay", // text shown on the button }; // ------------------------------ // Main draw function for this screen // ------------------------------ // drawGame() is called from main.js *only* // when currentScreen === "game" function drawGame2() { // Set background colour for the game screen background(240, 230, 140); // ---- Title and instructions text ---- fill(0); // black text textSize(32); textAlign(CENTER, CENTER); text("Game Screen", width / 2, 160); textSize(18);

```
text("Your friend cancels plans. How do you feel?", width / 2, 210); // ---- Draw the button ---
- // We pass the button object to a helper function // ---- Draw both buttons ----
drawGameButton(disappointBtn); drawGameButton(gameBtn); // ---- Cursor feedback ----
// If the mouse is over the button, show a hand cursor // Otherwise, show the normal arrow
cursor cursor(isHover(gameBtn) || isHover(disappointBtn) ? HAND : ARROW); } // ------------
----------------- // Button drawing helper // ------------------------------ // This function is
responsible *only* for drawing the button. // It does NOT handle clicks or game logic.
function drawGameButton({ x, y, w, h, label }) { rectMode(CENTER); // Check if the mouse is
hovering over the button // isHover() is defined in main.js so it can be shared const hover =
isHover({ x, y, w, h }); noStroke(); // Change button colour when hovered // This gives visual
feedback to the player fill( hover ? color(180, 220, 255, 220) // lighter blue on hover :
color(200, 220, 255, 190), // normal state ); // Draw the button rectangle rect(x, y, w, h, 14);
// last value = rounded corners // Draw the button text fill(0); textSize(28);
textAlign(CENTER, CENTER); text(label, x, y); } // ------------------------------ // Mouse input for
this screen // ------------------------------ // This function is called from main.js // only when
currentScreen === "game" function gameMousePressed() { if (isHover(disappointBtn)) {
currentScreen = "disappointPath"; // → goes to that screen } if (isHover(gameBtn)) {
currentScreen = "okayPath"; // → goes to other screen } }
```

Perfect — thanks for pasting this.
I can see the issue **immediately**, and it's a super common (and very fixable) one 👍

🚨 The real problem (this is the bug)

In **game2.js**, your mouse handler is named:

```
function gameMousePressed() {
```

…but in **main.js**, you are routing clicks like this:

```
else if (currentScreen === "game2") game2MousePressed?.();
```

❌ These names DO NOT MATCH

So when you're on `"game2"`:

- the screen **draws** (because `drawGame2()` exists)
- but clicks do NOTHING
- because `game2MousePressed()` does **not exist**

✅ The fix (one-line change)

Rename this function in `game2.js`

🔴 *CURRENT (wrong)*

```
function gameMousePressed() {
```

🟢 *CHANGE TO (correct)*

```
function game2MousePressed() {
```

That’s it. That alone will make the buttons clickable.

⚠️ Second important issue (not breaking, but dangerous)

You **redefined `drawGameButton()` inside game2.js**:

```
function drawGameButton({ x, y, w, h, label }) { ... }
```

But you already have this function globally (from `game.js`).

Why this is risky

- JS will overwrite the earlier version
- future changes might behave unpredictably

✅ Best practice (simple fix)

👉 **Delete `drawGameButton()` from game2.js entirely**
Keep ONE shared version.

✅ What game2.js should look like (clean version)

```
const disappointBtn = {
  x: 400,
  y: 440,
  w: 260,
  h: 90,
  label: "It's disappointing",
};

const gameBtn = {
  x: 400,
  y: 550,
  w: 260,
  h: 90,
  label: "It's okay",
};

function drawGame2() {
  background(240, 230, 140);

  fill(0);
  textSize(32);
  textAlign(CENTER, CENTER);
  text("Game Screen 2", width / 2, 160);

  textSize(18);
  text("Later that day, how do you feel now?", width / 2, 210);

  drawGameButton(disappointBtn);
  drawGameButton(gameBtn);

  cursor(
    isHover(gameBtn) || isHover(disappointBtn)
      ? HAND
      : ARROW
  );
}

function game2MousePressed() {
  if (isHover(disappointBtn)) {
    currentScreen = "disappointPath";
  }

  if (isHover(gameBtn)) {
    currentScreen = "okayPath";
  }
}
```

🧠 Why this happened (important lesson)

In your architecture:

- **each screen must have uniquely named input handlers**
- main.js does not “guess” which function to call
- it calls EXACT names

So the pattern is:

```
drawGame()        → gameMousePressed()
drawGame2()       → game2MousePressed()
drawOkayPath()    → okayPathMousePressed()
```

You’re actually doing this right everywhere else 👌
This was just a naming mismatch.

✅ After this fix, your flow will work:

```
Okay Path
 └── Next
      └── game2
           ├── It's okay
           └── It's disappointing
```

If you want next, I can help you:

- add a “Back” button to game2
- prevent infinite loops
- show progress (1 / 3, 2 / 3)
- add weather carry-over

Just tell me what’s next 🚀

it still doesn't work, this is my main: let currentScreen = "start"; // "start" | "instr" | "game" | "win" | "lose" // ------------------------------ // setup() runs ONCE at the beginning // ------------------------------ // This is where you usually set canvas size and initial settings. function setup() { createCanvas(800, 800); // Sets a default font for all text() calls // (This can be changed later per-screen if you want.) textFont("sans-serif"); } // ------------------------------ // draw() runs every frame (many times per second) // ------------------------------ // This is the core “router” for visuals. // Depending on currentScreen, we call the correct draw function. function draw() { // Each screen file defines its own draw function: // start.js → drawStart() // instructions.js → drawInstr() // game.js → drawGame() // win.js → drawWin() // lose.js → drawLose() if (currentScreen === "start") drawStart(); else if (currentScreen === "instr") drawInstr(); else if (currentScreen === "game") drawGame(); else if (currentScreen === "okayPath") drawOkayPath(); else if (currentScreen === "disappointPath") drawDisappointPath(); else if (currentScreen === "game2") drawGame2(); // (Optional teaching note) // This “if/else chain” is a very common early approach. // Later in the course you might replace it with: // - a switch statement, or // - an object/map of screens } // ------------------------------ // mousePressed() runs once each time the mouse is clicked // ------------------------------ // This routes mouse input to the correct screen handler. function mousePressed() { // Each screen *may* define a mouse handler: // start.js → startMousePressed() // instructions.js → instrMousePressed() // game.js →

gameMousePressed() // win.js → winMousePressed() // lose.js → loseMousePressed() if (currentScreen === "start") startMousePressed(); else if (currentScreen === "instr") instrMousePressed(); else if (currentScreen === "game") gameMousePressed(); // The ?.() means “call this function only if it exists” // This prevents errors if a screen doesn’t implement a handler. else if (currentScreen === "okayPath") okayPathMousePressed?.(); else if (currentScreen === "disappointPath") disappointPathMousePressed?.(); else if (currentScreen === "game2") game2MousePressed?.(); } // -- // Shared helper function: isHover() // -- // // Many screens have buttons. // This helper checks whether the mouse is inside a rectangle. // // Important: our buttons are drawn using rectMode(CENTER), // meaning x,y is the CENTRE of the rectangle. // So we check mouseX and mouseY against half-width/half-height bounds. // // Input: an object with { x, y, w, h } // Output: true if mouse is over the rectangle, otherwise false function isHover({ x, y, w, h }) { return (mouseX > x - w / 2 && // mouse is right of left edge mouseX < x + w / 2 && // mouse is left of right edge mouseY > y - h / 2 && // mouse is below top edge mouseY < y + h / 2 // mouse is above bottom edge); }

Great, thanks for pasting `main.js` — this actually helps a lot.
Your `main.js is correct. The routing is NOT the problem anymore.

At this point, when **a button draws but clicking it does nothing**, and routing is correct, there is **one classic JavaScript gotcha left** that fits your symptoms *perfectly*.

🚨 The REAL issue (this will stop ALL clicks)

In your **okayPath file**, you are using `menuBtn`:

```
cursor(isHover(menuBtn) || isHover(nextBtn) ? HAND : ARROW);
```

and

```
if (isHover(menuBtn)) {
  currentScreen = "start";
}
```

…but **`menuBtn is NOT defined anywhere in the file you pasted.**

❌ What happens because of this

When p5 tries to run:

```
isHover(menuBtn)
```

JavaScript throws a **ReferenceError**:

```
menuBtn is not defined
```

Once that error happens:

- the rest of the function **stops running**
- your `nextBtn` click is **never reached**
- it *looks* like the button does nothing

This is why:

- the button **appears**
- hover might even work sometimes
- but clicking does nothing

✅ How to confirm (do this once)

Open the **browser console** (right-click → Inspect → Console).

You will almost certainly see something like:

```
ReferenceError: menuBtn is not defined
```

✅ Fix Option A (simplest): remove menuBtn logic for now

If you don’t actually need the menu button on this screen yet:

🔴 Change this line in `drawOkayPath`:

```
cursor(isHover(menuBtn) || isHover(nextBtn) ? HAND : ARROW);
```

🟢 To:

```
cursor(isHover(nextBtn) ? HAND : ARROW);
```

And in `okayPathMousePressed()`:

```
function okayPathMousePressed() {
  if (isHover(nextBtn)) {
    currentScreen = "game2";
  }
}
```

👉 This will immediately make **Next work**.

✅ Fix Option B (cleaner): define `menuBtn`

If you *do* want a menu button, add this **at the top of the file**:

```
const menuBtn = {
  x: 700,
  y: 60,
  w: 140,
  h: 50,
  label: "Menu",
};
```

Then draw it:

```
drawGameButton(menuBtn);
```

Now BOTH buttons will work.

🧠 Why this bug is sneaky (important lesson)

JavaScript does **not** warn you politely.

One undefined variable:

- does NOT crash the canvas
- DOES silently stop your interaction logic
- makes buttons feel “dead”

This is why checking the **console** is huge in p5 projects.

✅ Summary (what to do right now)

✓ Your `main.js` is correct
✓ Your screen switching logic is correct
❌ `menuBtn` is undefined and blocking clicks

👉 Either:

- remove all `menuBtn` references
- **or** define it properly

Once you do that, **Next → game2 will work immediately**.

If you want, next I can help you:

- make a shared global menu button (no duplication)
- add a progress indicator (1/3, 2/3)
- prevent users from skipping without choosing

Just tell me 👍

```
const nextBtn = { x: 700, y: 700, w: 140, h: 50, label: "Next", }; function drawOkayPath() {
background(100, 190, 220); drawGameButton(nextBtn); drawSun(width / 2, height / 2 - 175,
80); fill(0); textAlign(CENTER, CENTER); textSize(32); text("That's okay :)", width / 2, height /
2); cursor(isHover(nextBtn) ? HAND : ARROW); } function okayPathMousePressed() { if
(isHover(nextBtn)) { currentScreen = "game2"; } } function drawSun(x, y, r) { // Sun rays
rectMode(CENTER); noStroke(); fill(255, 200, 60); for (let i = 0; i < 12; i++) { let angle =
(TWO_PI / 12) * i; let rayX = x + cos(angle) * (r + 30); let rayY = y + sin(angle) * (r + 30); push();
translate(rayX, rayY); rotate(angle); rect(0, 0, 12, 40); pop(); } // Sun body fill(255, 220, 80);
ellipse(x, y, r * 2); }
```

Perfect — thank you for pasting the **cleaned-up okayPath file**.
This version is **100% correct** on its own ✅.

So if **clicking “Next” still does nothing**, we are now down to **one single remaining cause**.
There’s no guessing left.

🚨 FINAL ROOT CAUSE (this is it)

`okayPathMousePressed()` is NOT being called.

Your button logic is correct.
Your hover logic is correct.
Your screen switch line is correct.

That means the problem is **pure routing**.

🔍 5-second proof (do this exactly)

Add this **temporarily**:

```
function okayPathMousePressed() {
  console.log("OKAY PATH CLICK");

  if (isHover(nextBtn)) {
    console.log("NEXT CLICKED");
    currentScreen = "game2";
  }
}
```

Now:

1. Open the browser console
2. Click **anywhere** on the Okay screen

Results:

✅*If you see nothing*

→ `okayPathMousePressed()` is not being routed

✅*If you see OKAY PATH CLICK but not NEXT CLICKED*

→ click area mismatch (unlikely)

✅*If you see both*

→ `game2` is not loading (file/link issue)

🧠 Given everything you've shown, the answer is almost certainly this:

❌ Your `mousePressed()` in `main.js` is missing OR shadowed

It **must** include this exact line:

```
else if (currentScreen === "okayPath") okayPathMousePressed?.();
```

And it must be in the **global p5 mousePressed**, not inside another function or file.

✅ Correct `mousePressed()` block (copy-paste safe)

```
function mousePressed() {
  if (currentScreen === "start") startMousePressed();
  else if (currentScreen === "instr") instrMousePressed();
  else if (currentScreen === "game") gameMousePressed();
  else if (currentScreen === "okayPath") okayPathMousePressed?.();
  else if (currentScreen === "disappointPath")
disappointPathMousePressed?.();
  else if (currentScreen === "game2") game2MousePressed?.();
}
```

🚨 ONE MORE VERY COMMON ISSUE (don't skip)

Is `okayPath.js` included in `index.html`?

If the file isn't loaded, **drawOkayPath works** (because it was already defined earlier), but the **mouse function never exists**.

You must have:

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

and it must be **below main.js**.

Correct order (important):

```
<script src="main.js"></script>
<script src="game.js"></script>
<script src="okayPath.js"></script>
<script src="disappointPath.js"></script>
<script src="game2.js"></script>
```

✅ Why I'm confident this is the fix

You have now:

✔ correct button object
✔ correct hover
✔ correct click logic
✔ correct screen name
✔ correct draw function

The **only remaining failure point** in p5 is:

the mouse handler never firing

And that is **always** routing or file loading.

🎯 What to do right now (in order)

1. Add the `console.log` test
2. Confirm whether it fires
3. If not → check `mousePressed()`
4. If still not → check `index.html` script order

If you want, paste **your index.html** next and I'll circle the exact missing line in under 10 seconds.

You're *very* close — this is the last wiring issue, not a logic one 💪

scratch the next button idea, how do i add a key that takes you back to the start screen

All good — this is actually **way simpler** and more reliable 👍
A keyboard shortcut is perfect for this stage.

We'll add **one key press** (like **B** or **ESC**) that always sends you back to the start screen.

✅ Option 1 (Recommended): Press B to go back to Start

This keeps things explicit and easy to demo.

1️⃣ Add this to each screen file you want it to work on

(for example: `okayPath.js`, `disappointPath.js`, `game2.js`)

```
function okayPathKeyPressed() {
  if (key === 'b' || key === 'B') {
    currentScreen = "start";
  }
}
```

(Replace `okayPath` with the correct screen name in each file.)

2️⃣Make sure `main.js` routes key presses

Your `main.js` already has `keyPressed()`, but it **does not yet route okayPath**.

Add these lines:

```
function keyPressed() {
  if (currentScreen === "start") startKeyPressed?.();
  else if (currentScreen === "instr") instrKeyPressed?.();
  else if (currentScreen === "game") gameKeyPressed?.();
  else if (currentScreen === "okayPath") okayPathKeyPressed?.();          // ✅
  else if (currentScreen === "disappointPath") disappointPathKeyPressed?.(); // ✅
  else if (currentScreen === "game2") game2KeyPressed?.();               // ✅
}
```

✅ Option 2: Use ESC key (very natural)

Inside any screen file:

```
function okayPathKeyPressed() {
  if (keyCode === ESCAPE) {
    currentScreen = "start";
  }
}
```

This feels very “app-like”.

🧠 Best Practice (Cleaner + DRY)

If you want this key to work **on every screen**, you can do it **once** in `main.js`:

```
function keyPressed() {

  // GLOBAL back shortcut
  if (key === 'b' || key === 'B') {
    currentScreen = "start";
    return;
  }

  if (currentScreen === "start") startKeyPressed?.();
  else if (currentScreen === "instr") instrKeyPressed?.();
  else if (currentScreen === "game") gameKeyPressed?.();
  else if (currentScreen === "okayPath") okayPathKeyPressed?.();
  else if (currentScreen === "disappointPath") disappointPathKeyPressed?.();
  else if (currentScreen === "game2") game2KeyPressed?.();
}
```

🔥 This is the cleanest solution.

🎮 Tell the player (important UX)

Add a small hint on screens:

```
textSize(14);
text("Press B to return to start", width/2, height - 40);
```

✅ Summary

✔ No buttons
✔ No hover bugs
✔ No routing headaches
✔ Keyboard-only = reliable

If you want next:

- show key hints visually
- add sound on key press
- prevent accidental exits

Just say what you want to polish ✨

how do I basically do the exact same thing as the initial disappointed and okay screen and add two buttons that lead to two new pathways onto those screens

You're basically asking:

"How do I repeat the same branching pattern again, cleanly, without breaking anything?"

And the answer is: **yes — do the exact same thing you already did**, just with **new screen names and new files**. This is *the* correct mental model.

I'll lay it out as a **repeatable recipe** you can follow every time.

🧠 The Pattern (this never changes)

For *every* new decision screen, you always do **4 things**:

1. **Create two button objects**
2. **Draw them**
3. **Handle clicks → set `currentScreen`**
4. **Create the destination screens**

That’s it. No extra logic.

✅ Step 1: On an existing screen (example: `okayPath.js`)

You already have one choice screen.
Now you want **two new choices** on it.

Add two buttons at the top of the file

```
const choiceABtn = {
  x: 400,
  y: 500,
  w: 260,
  h: 90,
  label: "Talk it out",
};

const choiceBBtn = {
  x: 400,
  y: 620,
  w: 260,
  h: 90,
  label: "Move on",
};
```

✅ Step 2: Draw them on the screen

Inside `drawOkayPath()`:

```
drawGameButton(choiceABtn);
drawGameButton(choiceBBtn);

cursor(
  isHover(choiceABtn) || isHover(choiceBBtn)
    ? HAND
    : ARROW
);
```

✅ Step 3: Handle clicks → NEW pathways

Inside `okayPathMousePressed()`:

```
function okayPathMousePressed() {

  if (isHover(choiceABtn)) {
    currentScreen = "talkPath";
  }

  if (isHover(choiceBBtn)) {
    currentScreen = "moveOnPath";
  }
}
```

⚠️ These screen names **do not exist yet** — that's normal.

✅ Step 4: Create the new pathway screens

📁 `talkPath.js`

```
function drawTalkPath() {
  background(180, 220, 200);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You decide to talk things out.", width/2, height/2);
}
```

📁 `moveOnPath.js`

```
function drawMoveOnPath() {
  background(200, 200, 200);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You decide to move on.", width/2, height/2);
}
```

✅ Step 5: Register them in `main.js`

In `draw()`:

```
else if (currentScreen === "talkPath") drawTalkPath();
else if (currentScreen === "moveOnPath") drawMoveOnPath();
```

In `mousePressed()`:

```
else if (currentScreen === "talkPath") talkPathMousePressed?.();
else if (currentScreen === "moveOnPath") moveOnPathMousePressed?.();
```

(Handlers optional for now.)

🧠 That's the whole system

Every new branch is just:

```
Screen
 ├── Button A → newScreenA
 └── Button B → newScreenB
```

You are **not supposed** to invent new logic each time.
Repetition here = good architecture.

📌 Visualizing your story now

```
game
 ├── okayPath
 │      ├── talkPath
 │      └── moveOnPath
 └── disappointPath
        ├── withdrawPath
        └── confrontPath
```

This is *exactly* how narrative games are built.

🎓 What a prof wants to see

You can literally explain it like this:

“Each narrative choice is its own screen with explicit routing, allowing branching without shared state complexity.”

That’s a strong answer.

🚀 If you want next

I can help you:

- add **one more layer of depth**
- prevent infinite loops
- track past choices
- add visual continuity (weather carries forward)
- refactor into a data-driven system (optional, advanced)

Just tell me how deep you want to go.

what do i do for the disappoint path

Great question — and this is actually where your project can start to feel *designed*, not just coded.

For the **disappointPath**, you should mirror the **same structure** as `okayPath`, but make the **choices, visuals, and tone different**. Think: *same system, different emotional outcome.*

Below is the **exact, repeatable pattern**, plus **concrete ideas** you can drop straight in.

🧠 Concept for Disappoint Path

The player already feels disappointed.
Now they choose **how they deal with it**.

Two strong, simple options:

1. **Withdraw** (internalize it)
2. **Confront** (externalize it)

These feel natural, readable, and easy to justify in a write-up.

✅ Step 1: Add two buttons to `disappointPath.js`

At the top of the file:

```
const withdrawBtn = {
  x: 400,
  y: 520,
  w: 260,
  h: 90,
  label: "Keep it to myself",
};

const confrontBtn = {
  x: 400,
  y: 640,
  w: 260,
  h: 90,
  label: "Say something",
};
```

✅ Step 2: Draw them

Inside `drawDisappointPath()`:

```
drawGameButton(withdrawBtn);
drawGameButton(confrontBtn);

cursor(
  isHover(withdrawBtn) || isHover(confrontBtn)
    ? HAND
    : ARROW
);
```

You'll still keep your rain visuals here — that's good contrast.

✅ Step 3: Handle clicks → new paths

Add / update this function:

```
function disappointPathMousePressed() {

  if (isHover(withdrawBtn)) {
    currentScreen = "withdrawPath";
  }

  if (isHover(confrontBtn)) {
    currentScreen = "confrontPath";
  }
}
```

Again: these screen names don’t exist yet — that’s fine.

✅ Step 4: Create the new screens

📁 `withdrawPath.js`

```
function drawWithdrawPath() {
  background(120, 120, 160);

  fill(255);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You keep it to yourself.", width / 2, height / 2);
}
```

📁 `confrontPath.js`

```
function drawConfrontPath() {
  background(160, 120, 120);

  fill(255);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You decide to say something.", width / 2, height / 2);
}
```

✅ Step 5: Register them in `main.js`

In `draw()`:

```
else if (currentScreen === "withdrawPath") drawWithdrawPath();
else if (currentScreen === "confrontPath") drawConfrontPath();
```

In `mousePressed()` (optional for now):

```
else if (currentScreen === "withdrawPath") withdrawPathMousePressed?.();
else if (currentScreen === "confrontPath") confrontPathMousePressed?.();
```

🧠 That’s literally it

You now have **parallel branching**:

```
okayPath
 ├── talkPath
```

```
  └── moveOnPath

disappointPath
  ├── withdrawPath
  └── confrontPath
```

Same mechanics. Different emotional meaning.

🎓 Why this is good design

You can say:

"The disappointed path explores internal vs external coping strategies, represented through contrasting choices."

That shows intention, not randomness.

🔥 Optional polish (only if you want)

- Withdraw → heavier rain / darker screen
- Confront → lightning flash or red tint
- Add subtle text shake on confront
- Add "Press B to return" hint

If you want, next I can help you:

- bring both paths back to a shared ending
- visualize accumulated emotion
- keep track of decisions
- write a short design rationale for submission

Just tell me what the assignment needs 👌

can you make it the same buttons as the okaypath

Yes — absolutely.
If you want the **disappointPath** to work **exactly the same way as okayPath**, you literally **copy the structure** and only change:

- the **text**
- the **screen names it goes to**
- (optionally) the background / mood

This is the **correct and intended approach** for your project.

Below is a **drop-in pattern** you can follow with confidence.

🧠 What “same buttons as okayPath” means

If `okayPath` has:

- 2 buttons
- same size
- same layout
- same logic

Then `disappointPath` should have:

- the **same two-button layout**
- just **different labels + destinations**

✅ Step 1: Copy the button setup

In **`disappointPath.js`**, add the SAME structure as `okayPath`.

At the top of the file:

```
const option1Btn = {
  x: 400,
  y: 520,
  w: 260,
  h: 90,
  label: "Say something",
};

const option2Btn = {
  x: 400,
  y: 640,
  w: 260,
  h: 90,
  label: "Keep it inside",
};
```

(These mirror the okayPath buttons exactly — just different labels.)

✅ Step 2: Draw them (same pattern)

Inside `drawDisappointPath()`:

```
drawGameButton(option1Btn);
drawGameButton(option2Btn);

cursor(
  isHover(option1Btn) || isHover(option2Btn)
    ? HAND
    : ARROW
);
```

Leave your rain/background code as-is.

✅ Step 3: Handle clicks (same logic)

```
function disappointPathMousePressed() {

  if (isHover(option1Btn)) {
    currentScreen = "confrontPath";
  }

  if (isHover(option2Btn)) {
    currentScreen = "withdrawPath";
  }
}
```

Same logic as okayPath — just different destinations.

✅ Step 4: Create the destination screens (blank is fine)

📁 `confrontPath.js`

```
function drawConfrontPath() {
  background(160, 120, 120);

  fill(255);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You decide to speak up.", width / 2, height / 2);
}
```

📁 `withdrawPath.js`

```
function drawWithdrawPath() {
  background(120, 120, 160);

  fill(255);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You keep it to yourself.", width / 2, height / 2);
}
```

✅ Step 5: Register them in `main.js`

In `draw()`:

```
else if (currentScreen === "confrontPath") drawConfrontPath();
else if (currentScreen === "withdrawPath") drawWithdrawPath();
```

In `mousePressed()` (optional for now):

```
else if (currentScreen === "confrontPath") confrontPathMousePressed?.();
else if (currentScreen === "withdrawPath") withdrawPathMousePressed?.();
```

🧠 Why this is the RIGHT way

You now have **symmetry**:

```
okayPath
 ├── option 1
 └── option 2

disappointPath
 ├── option 1
 └── option 2
```

Same UI
Same mechanics
Different emotional meaning

This is exactly how branching narratives are designed.

🎓 What to say if asked

“Both emotional paths use the same interaction structure to keep the experience consistent, while the narrative meaning changes.”

That’s a strong design justification.

If you want next, I can help you:

- bring all paths back to a shared ending
- show accumulated emotion visually
- prevent infinite loops
- cleanly refactor repeated code (optional)

Just tell me what you want to build next 👌

can you make it so that it also has the 2 buttons leading to the talk page or move on page

Yes 👍 — and this is actually **very clean design**.

You want **disappointPath** to have **the same two choices as okayPath**, and **both paths lead to the SAME next screens**:

- **Talk it out**
- **Move on**

So the *emotion* changes the context, but the *options* stay consistent. That’s solid.

Below is the **exact way to do it**, step by step, with no new concepts.

🧠 Goal

From **both**:

- `okayPath`
- `disappointPath`

You can go to:

- `talkPath`

- `moveOnPath`

✅ Step 1: Use the SAME button structure in `disappointPath.js`

At the top of **`disappointPath.js`**:

```
const talkBtn = {
  x: 400,
  y: 520,
  w: 260,
  h: 90,
  label: "Talk it out",
};

const moveOnBtn = {
  x: 400,
  y: 640,
  w: 260,
  h: 90,
  label: "Move on",
};
```

These should match the **same size and positions** as in `okayPath`.

✅ Step 2: Draw them on the disappointing screen

Inside `drawDisappointPath()`:

```
drawGameButton(talkBtn);
drawGameButton(moveOnBtn);

cursor(
  isHover(talkBtn) || isHover(moveOnBtn)
    ? HAND
    : ARROW
);
```

Keep your rain visuals — that contrast is good.

✅ Step 3: Handle clicks (same destinations as okayPath)

```
function disappointPathMousePressed() {

  if (isHover(talkBtn)) {
    currentScreen = "talkPath";
  }

  if (isHover(moveOnBtn)) {
    currentScreen = "moveOnPath";
  }
}
```

That’s it. No new logic.

✅ Step 4: Make sure the destination screens exist

You should already have these, but just to confirm:

`talkPath.js`

```
function drawTalkPath() {
  background(180, 220, 200);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You decide to talk it out.", width / 2, height / 2);
}
```

`moveOnPath.js`

```
function drawMoveOnPath() {
  background(200, 200, 200);

  fill(0);
  textAlign(CENTER, CENTER);
  textSize(28);
  text("You decide to move on.", width / 2, height / 2);
}
```

✅ Step 5: Confirm `main.js` routing (one-time check)

In `draw()`:

```
else if (currentScreen === "talkPath") drawTalkPath();
else if (currentScreen === "moveOnPath") drawMoveOnPath();
```

In `mousePressed()`:

```
else if (currentScreen === "talkPath") talkPathMousePressed?.();
else if (currentScreen === "moveOnPath") moveOnPathMousePressed?.();
```

(Mouse handlers optional.)

🧠 What your story structure now looks like

```
game
├── okayPath
│    ├── talkPath
│    └── moveOnPath
└── disappointPath
     ├── talkPath
     └── moveOnPath
```

Same choices.
Different emotional context.
Very intentional.

what is a new scenario i can use that will lead to an anger page or a funny/joyful page

Love this direction — an **anger vs funny/joyful split** gives you *clear emotional contrast* and is easy to communicate visually and conceptually.

I'll give you **one strong scenario**, plus **a few backups**, and then show **exactly how it maps to your existing structure** so it plugs right into what you've already built.

⭐ Best Scenario (Highly Recommended)

Scenario: Group Chat Message

Prompt text (neutral):

You open a group chat and see messages sent about you while you were gone.

This is perfect because:

- everyone relates to it
- it can be interpreted multiple ways
- it naturally splits into anger vs humor

🔥 Anger Path

Button label:

- "That's messed up."

Outcome:

- Goes to **angerPath**
- Visuals: red tint, shake, jagged shapes, harsh text

Anger page text idea:

“You feel heat rush in your chest.”

😂 Funny / Joyful Path

Button label:

- “Honestly… that’s kinda funny.”

Outcome:

- Goes to **joyPath**
- Visuals: warm colors, confetti dots, bouncy text

Joy page text idea:

“You laugh it off and screenshot it.”

🧠 Why this works well for your project

- Same situation
- Different interpretation
- Emotion drives outcome
- Very easy to justify in a design write-up

🔄 How it fits your existing structure

From **talkPath** or **moveOnPath**, you can do:

```
talkPath
 └── groupChatScene
        ├── angerPath
        └── joyPath
```

or

```
moveOnPath
 └── groupChatScene
       ├── angerPath
       └── joyPath
```

✏️ Alternate Scenarios (Quick Picks)

2️⃣ Accidental Photo Upload

You post a photo by accident.

- “Delete it now” → anger
- “Leave it, who cares” → joy

3️⃣ Missed Transit

Your bus/train leaves just as you arrive.

- “This is ridiculous” → anger
- “Guess I’m walking 😂” → joy

4️⃣ Barista Gets Your Order Wrong

They give you the wrong drink.

- “Say something” → anger
- “Try it anyway” → joy

5️⃣ Auto-Correct Disaster

Your phone sends the wrong word.

- “Cringe” → anger
- “Lean into it” → joy

🎨 Visual Direction (Simple)

Anger Page

- red / dark background
- thick lines
- slight screen shake

Joy Page

- yellow / pastel background
- circles / confetti
- playful font size changes

🎓 How to explain this academically

You can literally say:

“The same ambiguous social situation branches into contrasting emotional interpretations, highlighting how perception shapes experience.”

That’s gold for a critique.

👉 Want me to scaffold it?

If you want, I can:

- write the **exact screen file** for the group chat scenario
- design the anger + joy visuals
- wire it cleanly into your current flow

Just tell me which scenario you like best and I’ll build it with you step by step 🔥

i renamed the talkPath to frustratingPath and the moveOnPath to funnyPath, can you fix this: let stillRain = []; let rainSet = false; const talkBtn = { x: 400, y: 520, w: 260, h: 90, label: "Talk it out", }; const moveOnBtn = { x: 400, y: 640, w: 260, h: 90, label: "Move on", }; function drawStillRain() { stroke(200, 220, 255); strokeWeight(2); for (let drop of stillRain) { line(drop.x, drop.y, drop.x, drop.y + 12); } } function drawDisappointPath() { if (!rainSet) { initStillRain(); rainSet = true; } background(100); drawStillRain(); drawGameButton(talkBtn);

drawGameButton(moveOnBtn); cursor(isHover(talkBtn) || isHover(moveOnBtn) ? HAND : ARROW); fill(0); textAlign(CENTER, CENTER); textSize(32); text("That's disappointing :(", width / 2, height / 2 - 150); text("New scenario: You open a group chat and see your", width / 2, height / 2 - 150,); text("friends making jokes about you, while you were away", width / 2, height / 2,); } function disappointPathMousePressed() { if (isHover(talkBtn)) { currentScreen = "talkPath"; } if (isHover(moveOnBtn)) { currentScreen = "moveOnPath"; } } function initStillRain() { stillRain = []; for (let i = 0; i < 120; i++) { stillRain.push({ x: random(width), y: random(height), }); } }

Yep — you're *very* close. You just need a **clean rename pass** + **one tiny layout fix**.

Right now the logic still points to the **old screen names**, so clicks won't go where you expect.

I'll show you **exactly what to change** and then give you the **fixed version**.

🚨 What's broken (quick diagnosis)

You renamed:

- `talkPath` → **`frustratingPath`**
- `moveOnPath` → **`funnyPath`**

But your click handler still says:

```
currentScreen = "talkPath";
currentScreen = "moveOnPath";
```

So those screens no longer exist.

✅ Fix 1: Update the destinations

Change **only this function**:

❌ OLD

```
function disappointPathMousePressed() {
  if (isHover(talkBtn)) {
    currentScreen = "talkPath";
  }

  if (isHover(moveOnBtn)) {
    currentScreen = "moveOnPath";
  }
}
```

✅ NEW (correct)

```
function disappointPathMousePressed() {
  if (isHover(talkBtn)) {
    currentScreen = "frustratingPath";
  }

  if (isHover(moveOnBtn)) {
    currentScreen = "funnyPath";
  }
}
```

That's the **main fix**.

✅ Fix 2 (optional but recommended): Update button labels to match tone

Right now the labels still say "Talk it out" / "Move on".
If these now represent **anger vs humor**, make the intent clearer:

```
const talkBtn = {
  x: 400,
  y: 520,
  w: 260,
  h: 90,
  label: "That really annoys me",
};

const moveOnBtn = {
  x: 400,
  y: 640,
  w: 260,
  h: 90,
  label: "Honestly, that's funny",
};
```

This helps your story *read* better.

✅ Fix 3 (small layout bug): overlapping text Y positions

You currently draw two lines at the same Y:

```
text("That's disappointing :(", width / 2, height / 2 - 150);
text("New scenario: You open a group chat and see your", width / 2, height /
2 - 150);
```

So they overlap.

✅ Adjust spacing:

```
text("That's disappointing :(", width / 2, height / 2 - 220);
text(
  "New scenario: You open a group chat and see your",
  width / 2,
  height / 2 - 180
);
text(
  "friends making jokes about you while you were away",
  width / 2,
  height / 2 - 140
);
```

✅ Final (cleaned + fixed) version

Here’s your **corrected `disappointPath.js` core**, with only necessary changes:

```
let stillRain = [];
let rainSet = false;

const talkBtn = {
  x: 400,
  y: 520,
  w: 260,
  h: 90,
  label: "That really annoys me",
};

const moveOnBtn = {
  x: 400,
  y: 640,
  w: 260,
  h: 90,
  label: "Honestly, that's funny",
};

function drawStillRain() {
  stroke(200, 220, 255);
  strokeWeight(2);

  for (let drop of stillRain) {
    line(drop.x, drop.y, drop.x, drop.y + 12);
  }
}

function drawDisappointPath() {
  if (!rainSet) {
    initStillRain();
    rainSet = true;
  }

  background(100);
  drawStillRain();

  drawGameButton(talkBtn);
  drawGameButton(moveOnBtn);

  cursor(
    isHover(talkBtn) || isHover(moveOnBtn)
      ? HAND
      : ARROW
  );

  fill(255);
  textAlign(CENTER, CENTER);
  textSize(32);
  text("That's disappointing :(", width / 2, height / 2 - 220);

  textSize(18);
  text(
    "You open a group chat and see your friends",
    width / 2,
    height / 2 - 180
  );
  text(
    "making jokes about you while you were away.",
    width / 2,
    height / 2 - 140
  );
}

function disappointPathMousePressed() {
  if (isHover(talkBtn)) {
    currentScreen = "frustratingPath";
  }

  if (isHover(moveOnBtn)) {
    currentScreen = "funnyPath";
  }
}

function initStillRain() {
  stillRain = [];

  for (let i = 0; i < 120; i++) {
    stillRain.push({
      x: random(width),
      y: random(height),
    });
  }
}
```

🧠 Your story structure now (very solid)

```
disappointPath
 ├── frustratingPath (anger)
 └── funnyPath (humor)
```

That’s **clear, intentional, and emotionally readable**.

If you want next, I can help you:

- design the **frustratingPath visuals** (red, shake, harsh shapes)
- design the **funnyPath visuals** (confetti, bounce, warm colors)
- or bring both back to a shared ending

Just tell me which one you want to build next 👌