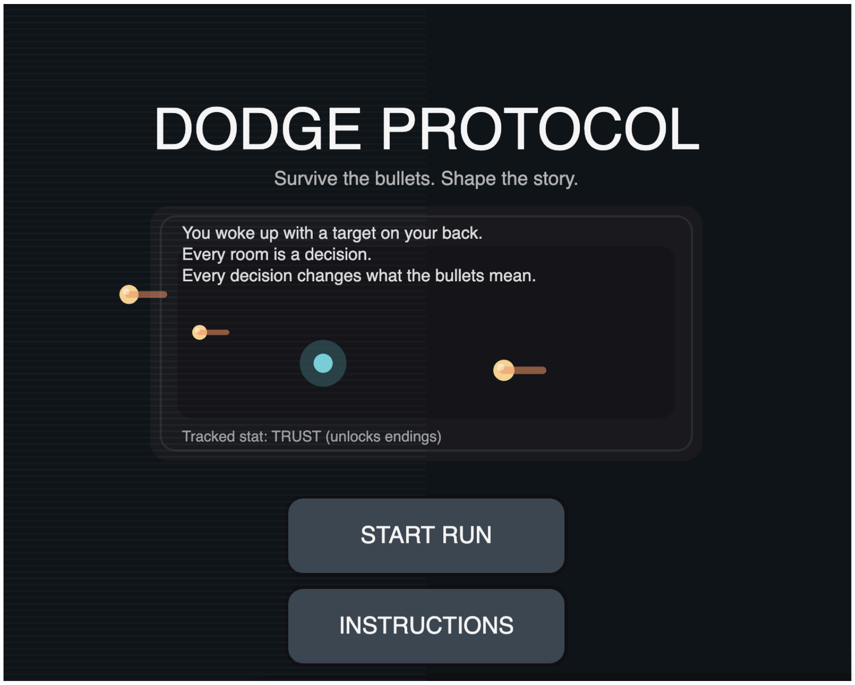
Process & Decision Documentation

Project/Assignment Decisions

- Changed the approach to the game after it failed multiple times because the losing screen and winning screen was not functioning properly. After a player won/lost the game, the game did not reset properly.
- The player is supposed to dodge the bullets using keys on their keyboard. To win, the player must dodge all the bullets without getting hit by when the timer runs out.

Role-Based Process Evidence



DODGE PROTOCOL

Survive the bullets. Shape the story.

You woke up with a target on your back.
Every room is a decision.
Every decision changes what the bullets mean.

Tracked stat: TRUST (unlocks endings)

Entry Header

Name: Jenny Dang

Primary responsibility for this work: To create an interactive game

Tools, Resources, or Inputs Used

- GenAI

GenAI Documentation

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

Date Used: Feb 1, 2026

Tool Disclosure: ChatGPT 5.2

Purpose of Use: To help me create an interactive game using p5.js

Summary of Interaction: Created code/comments for my interactive game

Human Decision Point(s): I redirected the AI and regenerated my prompts when the code in the game was not working correctly

Integrity & Verification Note: I checked AI for accuracy by looking to see if the game created is accurate with the instructions I set.

Scope of GenAI Use: I made the final decisions for the code and developed the story of the game.

Limitations or Misfires:It misunderstood my instructions at the beginning and didn't understand

Summary of Process (Human + Tool)

- Redirected the GenAI tool by changing my prompts to get the correct output I'm looking for.

Decision Points & Trade-offs

- Changed the storyline of the game. Instead of unlocking to new levels, the game ends when the player dodges the bullets successfully or unsuccessfully.

Verification & Judgement

- Playtesting the game to see if it works correctly and smoothly

Appendix

Please include a full transcript of your conversation with the GenAI. GenAI transcripts are used for transparency and verification only and are not assessed for writing quality or completeness.

Note: If GenAI is used in another language, you must include both the original language and a translated transcript in the appendix

ME: okay can you make an interactive story where the player has to dodge bullets make the starting screen for it

AI: // start.js

```
// NOTE: Do NOT add setup() or draw() in this file
// setup() and draw() live in main.js
// This file defines:
// 1) drawStart() → start/menu screen visuals
// 2) input handlers for this screen
// 3) helper function(s) used only on this screen
//
// Assumptions (safe defaults):
// - main.js owns: currentScreen, resetRun()
// - utils.js provides: isHover()
// - main.js routes input to startMousePressed/startKeyPressed when currentScreen === "start"

function drawStart() {
  // Dark, tense backdrop
  background(18, 20, 26);

  // Subtle scanlines / texture
  noStroke();
  for (let y = 0; y < height; y += 6) {
    fill(255, 255, 255, 6);
    rect(0, y, width, 2);
  }

  // Title
  fill(245);
  textAlign(CENTER, CENTER);
  textSize(54);
  text("DODGE PROTOCOL", width / 2, 110);

  // Tagline
  fill(180);
  textSize(18);
  text("An interactive story told through survival choices.", width / 2, 155);

  // Minimal “scene preview” panel
  drawPreviewPanel(width / 2, 300, 520, 240);

  // Buttons
  const startBtn = { x: width / 2, y: 480, w: 260, h: 70, label: "START RUN" };
  const instrBtn = { x: width / 2, y: 560, w: 260, h: 70, label: "INSTRUCTIONS" };

  drawMenuButton(startBtn);
  drawMenuButton(instrBtn);

  // Footer hints
  fill(140);
```

```
  textSize(14);
  text("ENTER = start   |   I = instructions", width / 2, height - 24);

  cursor(isHover(startBtn) || isHover(instrBtn) ? HAND : ARROW);
}

function startMousePressed() {
  const startBtn = { x: width / 2, y: 480, w: 260, h: 70 };
  const instrBtn = { x: width / 2, y: 560, w: 260, h: 70 };

  if (isHover(startBtn)) {
    if (typeof resetRun === "function") resetRun();
    currentScreen = "game";
  } else if (isHover(instrBtn)) {
    currentScreen = "instr";
  }
}

function startKeyPressed() {
  if (keyCode === ENTER) {
    if (typeof resetRun === "function") resetRun();
    currentScreen = "game";
  }
  if (key === "i" || key === "I") {
    currentScreen = "instr";
  }
```

```
}

// ------------------------------------------------------------
// Helpers (start screen only)
// ------------------------------------------------------------

function drawMenuButton(btn) {
  const hover = isHover(btn);

  rectMode(CENTER);
  noStroke();

  if (hover) {
    fill(235, 90, 75);
    drawingContext.shadowBlur = 18;
    drawingContext.shadowColor = color(235, 90, 75, 120);
  } else {
    fill(60, 70, 85);
    drawingContext.shadowBlur = 10;
    drawingContext.shadowColor = color(0, 0, 0, 90);
  }

  rect(btn.x, btn.y, btn.w, btn.h, 14);
  drawingContext.shadowBlur = 0;

  fill(245);
  textAlign(CENTER, CENTER);
  textSize(22);
  text(btn.label, btn.x, btn.y);
}

function drawPreviewPanel(cx, cy, w, h) {
  const x = cx - w / 2;
  const y = cy - h / 2;

  // Panel base
  rectMode(CORNER);
  noStroke();
  fill(255, 255, 255, 10);
  rect(x, y, w, h, 18);

  // Inner frame
  stroke(255, 255, 255, 18);
  strokeWeight(2);
  noFill();
  rect(x + 10, y + 10, w - 20, h - 20, 14);
  noStroke();

  // “Arena” area
  const ax = x + 26;
  const ay = y + 34;
  const aw = w - 52;
  const ah = h - 68;

  fill(0, 0, 0, 55);
  rect(ax, ay, aw, ah, 14);

  // Fake bullets (animated by frameCount, still works without draw() here)
  // They sweep across to imply dodging.
  const t = frameCount * 0.03;

  // Player silhouette
  const px = ax + aw * 0.28 + sin(t * 1.2) * 10;
  const py = ay + ah * 0.62 + cos(t * 0.9) * 10;

  // Player glow
  fill(90, 210, 220, 60);
  ellipse(px, py, 44, 44);

  // Player body
  fill(90, 210, 220);
  ellipse(px, py, 18, 18);

  // Bullets
  drawBullet(ax + aw * 0.95 - (t * 140) % (aw + 80), ay + ah * 0.28, 18);
  drawBullet(ax + aw * 0.95 - (t * 190 + 70) % (aw + 120), ay + ah * 0.48, 14);
  drawBullet(ax + aw * 0.95 - (t * 230 + 140) % (aw + 160), ay + ah * 0.70, 20);
```

```
  // Story prompt text
  fill(220);
  textAlign(LEFT, TOP);
  textSize(16);
  text(
    "You woke up with a target on your back.\n" +
      "Every room is a decision.\n" +
      "Every decision changes what the bullets mean.",
    x + 30,
    y + 18
  );

  // Stat hint (fits the “interactive story” requirement)
  fill(160);
  textSize(14);
  text("Tracked stat: TRUST (unlocks endings)", x + 30, y + h - 30);
}

function drawBullet(x, y, s) {
  // Core
  noStroke();
  fill(255, 210, 140);
  ellipse(x, y, s, s);

  // Streak
  fill(255, 140, 90, 140);
  rectMode(CENTER);
  rect(x + s * 0.9, y, s * 2.2, s * 0.35, 8);


  // Hot highlight
  fill(255, 255, 255, 120);
  ellipse(x - s * 0.15, y - s * 0.15, s * 0.35, s * 0.35);
}
```

ME: edit the instructions so that it tells how to play the game // NOTE: Do NOT add setup() or draw() in this file // setup() and draw() live in main.js // This file only defines: // 1) drawInstr() → what the instructions screen looks like // 2) input handlers → how the player returns to the start screen // 3) helper functions specific to this screen // ------------------------------ // Main draw function for instructions screen // ------------------------------ // drawInstr() is called from main.js // only when currentScreen === "instr" function drawInstr() { // Light neutral background background(240); // ---- Screen title ---- fill(0); textAlign(CENTER, TOP); textSize(36); text("Instructions", width / 2, 80); // ---- Instruction text ---- textSize(18); // \n creates a line break in the text // This is useful for simple multi-line instructions const lines = "Press the game button.\n" + "You have a chance to win or lose!"; text(lines, width / 2, 160); // ---- Back button ---- // This button lets the player return to the start screen const backBtn = { x: width / 2, // centred horizontally y: 560, w: 220, h: 70, label: "BACK", }; // Draw the back button drawInstrButton(backBtn); // Change cursor when hovering over the button cursor(isHover(backBtn) ? HAND : ARROW); } // ------------------------------ // Mouse input for instructions screen // ------------------------------ // Called from main.js only when currentScreen === "instr" function instrMousePressed() { // Button data must match the draw position const backBtn = { x: width / 2, y: 560, w: 220, h: 70 }; // If the button is clicked, return to the start screen if (isHover(backBtn)) { currentScreen = "start"; } } // ------------------------------ // Keyboard input for instructions screen // ------------------------------ // Provides keyboard-only navigation function instrKeyPressed() { // ESC is a common “go back” key in games and apps if (keyCode === ESCAPE) { currentScreen = "start"; } // B key is an additional, explicit shortcut for “back” if (key === "b" || key === "B") { currentScreen = "start"; } } // ------------------------------ // Button drawing helper (instructions screen) // ------------------------------ // This function is only responsible for drawing the button. // It is kept separate so the visual style can be changed // without touching input or game logic. function drawInstrButton({ x, y, w, h, label }) { rectMode(CENTER); // Check whether the mouse is hovering over the button const

```
hover = isHover({ x, y, w, h }); noStroke(); // Subtle colour change on hover for visual feedback fill(hover ? color(200, 200, 255, 200) : color(220, 220, 255, 170)); // Draw the button shape rect(x, y, w, h, 12); // Draw the button text fill(0); textSize(26); textAlign(CENTER, CENTER); text(label, x, y); }
```

AI:

```
// NOTE: Do NOT add setup() or draw() in this file
// setup() and draw() live in main.js
// This file only defines:
// 1) drawInstr() → what the instructions screen looks like
// 2) input handlers → how the player returns to the start screen
// 3) helper functions specific to this screen

// ------------------------------
// Main draw function for instructions screen
// ------------------------------
// drawInstr() is called from main.js
// only when currentScreen === "instr"
function drawInstr() {
  // Light neutral background
  background(240);

  // ---- Screen title ----
  fill(0);
  textAlign(CENTER, TOP);
  textSize(36);
  text("How to Play", width / 2, 80);
```

```
// ---- Instruction text ----
textSize(18);
textAlign(CENTER, TOP);

const lines =
  "You are trapped in a series of rooms under fire.\n\n" +
  "Your goal is to survive by dodging bullets and making decisions.\n\n" +
  "• Move your character using the arrow keys or WASD.\n" +
  "• Bullets will fly across the screen — avoid getting hit.\n" +
  "• Each room presents a choice that affects the story.\n\n" +
  "Your choices change your TRUST level.\n" +
  "High or low trust unlocks different endings.\n\n" +
  "Survive long enough to discover the truth.";

text(lines, width / 2, 160);

// ---- Back button ----
const backBtn = {
  x: width / 2,
  y: 560,
  w: 220,
  h: 70,
  label: "BACK",
};

drawInstrButton(backBtn);
```

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

// ------------------------------
// Mouse input for instructions screen
// ------------------------------
function instrMousePressed() {
  const backBtn = { x: width / 2, y: 560, w: 220, h: 70 };

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

// ------------------------------
// Keyboard input for instructions screen
// ------------------------------
function instrKeyPressed() {
  if (keyCode === ESCAPE) {
    currentScreen = "start";
  }

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

```
}


// ------------------------------
// Button drawing helper (instructions screen)
// ------------------------------
function drawInstrButton({ x, y, w, h, label }) {
  rectMode(CENTER);


  const hover = isHover({ x, y, w, h });


  noStroke();
  fill(hover ? color(200, 200, 255, 200) : color(220, 220, 255, 170));
  rect(x, y, w, h, 12);


  fill(0);
  textSize(26);
  textAlign(CENTER, CENTER);
  text(label, x, y);
}
```

now edit the game // 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: "PRESS HERE", // 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(
"Click the button (or press ENTER) for a random result.", 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"; } }
```

ME: // 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

//

// Requires globals from main.js (or other files):

// - currentScreen

// - player = { trust: number }   // 0–100

// - gameState = { nodeId: string, history: [], lastChoice: null }

// - STORY object in storyData.js

// - UI helpers: drawChoiceButton(), drawTopBar()  (from ui.js)

// - util: isHover(), clamp() (from utils.js)


// ------------------------------

// Local game objects (only used on the game screen)

// ------------------------------

let dodger = {

  x: 0,

  y: 0,

  r: 10,

  speed: 4.2,

};


let bullets = [];

let spawnTimer = 0;


// “Room” survival requirement to unlock choices
```

```
let room = {
  secondsToSurvive: 5, // increases per room
  startedAtMs: 0,
  survived: false,
};

function drawGame() {
  // Initialize on first frame of this screen
  if (room.startedAtMs === 0) initRoom();

  // Background
  background(18, 20, 26);

  // HUD
  if (typeof drawTopBar === "function") drawTopBar();
  drawRoomHeader();

  // Arena
  const arena = getArenaRect();
  drawArena(arena);

  // Update sim
  updateDodger(arena);
  updateBullets(arena);

  // Draw entities
  drawDodger();
  drawBullets();

  // Check hits + time condition
  const hit = checkCollision();
  if (hit) {
    onHit();
  }

  updateSurvival(arena);

  // Story panel + choices (unlocked after surviving)
  drawStoryPanel();

  // Cursor hint for choices
  if (room.survived) {
    const anyHover = isAnyChoiceHovered();
    cursor(anyHover ? HAND : ARROW);
  } else {
    cursor(ARROW);
  }
}

// ------------------------------
// Input
// ------------------------------
```

```
function gameMousePressed() {
  if (!room.survived) return;

  const node = STORY[gameState.nodeId];
  if (!node || !node.choices) return;

  const btns = buildChoiceButtons(node);
  for (let i = 0; i < btns.length; i++) {
    if (isHover(btns[i])) {
      applyChoice(node.choices[i]);
      return;
    }
  }
}

function gameKeyPressed() {
  // Number keys select choices when unlocked
  if (!room.survived) return;

  const node = STORY[gameState.nodeId];
  if (!node || !node.choices) return;

  const idx = keyCode - 49; // '1' -> 0
  if (idx >= 0 && idx < node.choices.length) {
    applyChoice(node.choices[idx]);
  }
```

```
}

// ------------------------------
// Room / arena setup
// ------------------------------
function initRoom() {
  const arena = getArenaRect();

  dodger.x = arena.x + arena.w * 0.22;
  dodger.y = arena.y + arena.h * 0.55;

  bullets = [];
  spawnTimer = 0;

  room.startedAtMs = millis();
  room.survived = false;

  // Slightly scale difficulty by “depth”
  const depth = gameState.history.length;
  room.secondsToSurvive = clamp(5 + depth * 1.5, 5, 12);
}

function getArenaRect() {
  // Leave space on the right for story/choices
  return {
    x: 40,
    y: 80,
    w: width * 0.55,
    h: height - 140,
  };
}

function drawArena(a) {
  noStroke();
  fill(0, 0, 0, 70);
  rectMode(CORNER);
  rect(a.x, a.y, a.w, a.h, 16);

  stroke(255, 255, 255, 18);
  strokeWeight(2);
  noFill();
  rect(a.x + 10, a.y + 10, a.w - 20, a.h - 20, 12);
  noStroke();
}

function drawRoomHeader() {
  const node = STORY[gameState.nodeId];
  const title = node ? node.title : "Unknown";

  fill(235);
  textAlign(LEFT, TOP);
  textSize(18);
  text(`ROOM: ${title}`, 40, 44);

  fill(170);
  textSize(14);
  text("Dodge until choices unlock.", 40, 64);
}

// ------------------------------
// Dodger movement
// ------------------------------
function updateDodger(arena) {
  let dx = 0;
  let dy = 0;

  if (keyIsDown(LEFT_ARROW) || keyIsDown(65)) dx -= 1; // A
  if (keyIsDown(RIGHT_ARROW) || keyIsDown(68)) dx += 1; // D
  if (keyIsDown(UP_ARROW) || keyIsDown(87)) dy -= 1; // W
  if (keyIsDown(DOWN_ARROW) || keyIsDown(83)) dy += 1; // S

  // Normalize diagonal
  if (dx !== 0 && dy !== 0) {
    dx *= 0.7071;
    dy *= 0.7071;
  }

  dodger.x += dx * dodger.speed;
  dodger.y += dy * dodger.speed;

  dodger.x = clamp(dodger.x, arena.x + dodger.r, arena.x + arena.w - dodger.r);
  dodger.y = clamp(dodger.y, arena.y + dodger.r, arena.y + arena.h - dodger.r);
}

function drawDodger() {
  // glow
  noStroke();
  fill(90, 210, 220, 70);
  ellipse(dodger.x, dodger.y, dodger.r * 5.0, dodger.r * 5.0);

  // core
  fill(90, 210, 220);
  ellipse(dodger.x, dodger.y, dodger.r * 2, dodger.r * 2);
}

// ------------------------------
// Bullets
// ------------------------------
function updateBullets(arena) {
  // spawn cadence
  spawnTimer -= deltaTime;
  if (spawnTimer <= 0) {
    spawnBullet(arena);
    const depth = gameState.history.length;
    const base = 520;
    spawnTimer = clamp(base - depth * 35, 260, 520); // ms
  }

  // move
  for (let i = bullets.length - 1; i >= 0; i--) {
    const b = bullets[i];
    b.x += b.vx;
    b.y += b.vy;

    // cull offscreen
    if (
      b.x < arena.x - 80 ||
      b.x > arena.x + arena.w + 80 ||
      b.y < arena.y - 80 ||
      b.y > arena.y + arena.h + 80
    ) {
      bullets.splice(i, 1);
    }
  }
}

function spawnBullet(arena) {
  // spawn from right edge, aim roughly toward the dodger with variance
  const y = random(arena.y + 20, arena.y + arena.h - 20);
  const x = arena.x + arena.w + 40;
  const targetX = dodger.x;
  const targetY = dodger.y + random(-80, 80);

  const angle = atan2(targetY - y, targetX - x);
  const depth = gameState.history.length;

  const speed = clamp(4.2 + depth * 0.35, 4.2, 7.6);
  const vx = cos(angle) * speed;
  const vy = sin(angle) * speed;

  bullets.push({
    x,
    y,
    r: random(7, 11),
    vx,
    vy,
  });
}

function drawBullets() {
  for (const b of bullets) {
    // streak
    noStroke();
    fill(255, 140, 90, 140);
    push();
    translate(b.x, b.y);
    const ang = atan2(b.vy, b.vx);
    rotate(ang);
    rectMode(CENTER);
    rect(10, 0, b.r * 3.2, b.r * 0.5, 8);
    pop();

    // core
    fill(255, 210, 140);
    ellipse(b.x, b.y, b.r * 2, b.r * 2);

    // highlight
    fill(255, 255, 255, 120);
    ellipse(b.x - b.r * 0.3, b.y - b.r * 0.3, b.r * 0.7, b.r * 0.7);
  }
}

// ------------------------------
// Collision + consequences
// ------------------------------
function checkCollision() {
  for (const b of bullets) {
    const d = dist(dodger.x, dodger.y, b.x, b.y);
    if (d < dodger.r + b.r) return true;
  }
  return false;
```

```
}

function onHit() {
  // Penalty: trust drops; restart the same node/room
  player.trust = clamp(player.trust - 12, 0, 100);
  initRoom();
}

// ------------------------------
// Survival gating for story choices
// ------------------------------
function updateSurvival(arena) {
  const elapsed = (millis() - room.startedAtMs) / 1000;
  const remaining = max(0, room.secondsToSurvive - elapsed);

  // Progress bar in arena
  const barW = arena.w - 40;
  const barH = 10;
  const bx = arena.x + 20;
  const by = arena.y + 18;

  noStroke();
  fill(255, 255, 255, 55);
  rectMode(CORNER);
  rect(bx, by, barW, barH, 8);
  const pct = clamp(1 - remaining / room.secondsToSurvive, 0, 1);

  fill(90, 210, 220, 160);

  rect(bx, by, barW * pct, barH, 8);


  fill(220);

  textAlign(LEFT, TOP);

  textSize(12);

  text(`UNLOCK IN: ${remaining.toFixed(1)}s`, bx, by + 14);


  if (remaining <= 0 && !room.survived) {

    room.survived = true;

    // small reward for surviving the room

    player.trust = clamp(player.trust + 3, 0, 100);

  }

}


// ------------------------------

// Story panel + choices

// ------------------------------

function drawStoryPanel() {

  const node = STORY[gameState.nodeId];


  const panel = {

    x: width * 0.62,

    y: 80,

    w: width * 0.35,
    h: height - 140,
  };

  // Panel base
  rectMode(CORNER);
  noStroke();
  fill(255, 255, 255, 18);
  rect(panel.x, panel.y, panel.w, panel.h, 16);

  // Title
  fill(245);
  textAlign(LEFT, TOP);
  textSize(20);
  const title = node ? node.title : "Missing node";
  text(title, panel.x + 18, panel.y + 16);

  // Body
  fill(210);
  textSize(14);
  const body = node ? node.body : "No story text found for this nodeId.";
  text(body, panel.x + 18, panel.y + 48, panel.w - 36, 200);

  // Choices / end status
  if (!node || !node.choices || node.choices.length === 0) {
    drawEndingBlock(panel, node);
    return;
  }

  if (!room.survived) {
    fill(170);
    textSize(14);
    text("Survive the arena to unlock choices.", panel.x + 18, panel.y + 260);
    return;
  }

  // Draw choice buttons
  const btns = buildChoiceButtons(node);
  for (let i = 0; i < btns.length; i++) {
    drawChoiceButton(btns[i]);

    // small keyboard hint
    fill(180);
    textAlign(LEFT, CENTER);
    textSize(12);
    text(`${i + 1}.`, btns[i].x - btns[i].w / 2 + 10, btns[i].y);
  }
}

function buildChoiceButtons(node) {
  const baseX = width * 0.62 + (width * 0.35) / 2;
  const startY = 360;
  const gap = 76;
  const btns = [];
  for (let i = 0; i < node.choices.length; i++) {
    btns.push({
      x: baseX,
      y: startY + i * gap,
      w: width * 0.33,
      h: 56,
      label: node.choices[i].label,
    });
  }
  return btns;
}

function isAnyChoiceHovered() {
  const node = STORY[gameState.nodeId];
  if (!node || !node.choices) return false;
  const btns = buildChoiceButtons(node);
  for (const b of btns) if (isHover(b)) return true;
  return false;
}

function applyChoice(choice) {
  // update stat
  const delta = choice.trustDelta || 0;
  player.trust = clamp(player.trust + delta, 0, 100);
  gameState.history.push(gameState.nodeId);
  gameState.lastChoice = choice.id;

  // jump to next node
  gameState.nodeId = choice.next;

  // If the next node is an ending that depends on trust, resolve it here
  resolveGatedEndingIfNeeded();

  // start new room
  room.startedAtMs = 0;
}

function resolveGatedEndingIfNeeded() {
  const node = STORY[gameState.nodeId];
  if (!node) return;

  if (node.gate && node.gate.type === "trust") {
    const t = player.trust;
    if (t >= node.gate.high) {
      gameState.nodeId = node.gate.highNext;
    } else if (t <= node.gate.low) {
      gameState.nodeId = node.gate.lowNext;
    } else {
      gameState.nodeId = node.gate.midNext;
    }
  }
}


function drawEndingBlock(panel, node) {
  fill(200);
  textAlign(LEFT, TOP);
  textSize(14);


  const endingText =
    (node && node.endingText) ||
    "END.\n\nThis path ends here.\nYour TRUST determines which endings you can reach.";


  text(endingText, panel.x + 18, panel.y + 260, panel.w - 36, 220);


  // Restart hint
  fill(170);
  textSize(12);
  text("Press ESC to return to start.", panel.x + 18, panel.y + panel.h - 34);
}
```

ME: its freezing my game

AI:

```
// game.js
// NOTE: Do NOT add setup() or draw() in this file
// setup() and draw() live in main.js

// ------------------------------------------------------------
// Fallbacks (prevents freeze if other files aren't loaded)
// ------------------------------------------------------------
if (typeof clamp !== "function") {
  function clamp(v, lo, hi) {
    return Math.max(lo, Math.min(hi, v));
  }
}

if (typeof isHover !== "function") {
  function isHover({ x, y, w, h }) {
    return (
      mouseX > x - w / 2 &&
      mouseX < x + w / 2 &&
      mouseY > y - h / 2 &&
      mouseY < y + h / 2
    );
  }
}

if (typeof drawChoiceButton !== "function") {
  function drawChoiceButton(btn) {
    const hover = isHover(btn);
    rectMode(CENTER);
    noStroke();
    fill(hover ? 240 : 225);
    rect(btn.x, btn.y, btn.w, btn.h, 12);
    fill(20);
    textAlign(CENTER, CENTER);
    textSize(16);
    text(btn.label, btn.x, btn.y);
  }
}

if (typeof drawTopBar !== "function") {
  function drawTopBar() {
    if (typeof player !== "object") return;
    const bx = 28, by = 22, bw = width - 56, bh = 14;

    noStroke();
    fill(255, 255, 255, 120);
    rectMode(CORNER);
    rect(bx, by, bw, bh, 8);

    const pct = clamp((player.trust || 0) / 100, 0, 1);
    fill(80, 170, 180, 200);
    rect(bx, by, bw * pct, bh, 8);

    fill(235);
    textAlign(LEFT, TOP);
    textSize(12);
    text(`TRUST: ${player.trust}`, bx, by + bh + 6);
  }
}

// Minimal STORY fallback (prevents STORY undefined freeze)
if (typeof STORY === "undefined") {
  var STORY = {
    intro: {
      title: "Intro",
      body: "STORY is missing.\nLoad storyData.js before game.js.",
      choices: [{ id: "x", label: "Back to Start", next: "intro", trustDelta: 0 }],
    },
  };
}

// Ensure required globals exist
if (typeof player !== "object") {
  var player = { trust: 50 };
}
if (typeof gameState !== "object") {
  var gameState = { nodeId: "intro", history: [], lastChoice: null };
}

// ------------------------------------------------------------
// Local game objects
// ------------------------------------------------------------
let dodger = { x: 0, y: 0, r: 10, speed: 4.2 };
let bullets = [];
let spawnTimer = 0;

let room = {
  secondsToSurvive: 5,
  startedAtMs: 0,
  survived: false,
};

// ------------------------------------------------------------
// Main draw
// ------------------------------------------------------------
function drawGame() {
  // Guard against undefined nodeId
  if (!gameState.nodeId || !STORY[gameState.nodeId]) {
    gameState.nodeId = "intro";
  }

  if (room.startedAtMs === 0) initRoom();

  background(18, 20, 26);

  drawTopBar();
  drawRoomHeader();
  const arena = getArenaRect();
  drawArena(arena);

  updateDodger(arena);
  updateBullets(arena);

  drawDodger();
  drawBullets();

  if (checkCollision()) onHit();

  updateSurvival(arena);
  drawStoryPanel();

  if (room.survived && isAnyChoiceHovered()) cursor(HAND);
  else cursor(ARROW);
}

// ------------------------------------------------------------
// Input
// ------------------------------------------------------------
function gameMousePressed() {
  if (!room.survived) return;

  const node = STORY[gameState.nodeId];
  if (!node || !node.choices) return;

  const btns = buildChoiceButtons(node);
  for (let i = 0; i < btns.length; i++) {
    if (isHover(btns[i])) {
      applyChoice(node.choices[i]);
      return;
    }
  }
}

function gameKeyPressed() {
  // ESC goes back to start (prevents “stuck” feeling)
  if (keyCode === ESCAPE) {
    if (typeof currentScreen !== "undefined") currentScreen = "start";
    return;
  }

  if (!room.survived) return;

  const node = STORY[gameState.nodeId];
  if (!node || !node.choices) return;

  const idx = keyCode - 49; // '1' -> 0
  if (idx >= 0 && idx < node.choices.length) {
    applyChoice(node.choices[idx]);
  }
}

// -----------------------------------------------------------
// Room / arena
// -----------------------------------------------------------
function initRoom() {
  const arena = getArenaRect();

  dodger.x = arena.x + arena.w * 0.22;
  dodger.y = arena.y + arena.h * 0.55;

  bullets = [];
  spawnTimer = 0;

  room.startedAtMs = millis();
  room.survived = false;

  const depth = (gameState.history && gameState.history.length) || 0;
  room.secondsToSurvive = clamp(5 + depth * 1.5, 5, 12);
}

function getArenaRect() {
  return { x: 40, y: 80, w: width * 0.55, h: height - 140 };
}
```

```
function drawArena(a) {
  noStroke();
  fill(0, 0, 0, 70);
  rectMode(CORNER);
  rect(a.x, a.y, a.w, a.h, 16);

  stroke(255, 255, 255, 18);
  strokeWeight(2);
  noFill();
  rect(a.x + 10, a.y + 10, a.w - 20, a.h - 20, 12);
  noStroke();
}

function drawRoomHeader() {
  const node = STORY[gameState.nodeId];
  fill(235);
  textAlign(LEFT, TOP);
  textSize(18);
  text(`ROOM: ${node ? node.title : "Unknown"}`, 40, 44);

  fill(170);
  textSize(14);
  text("Dodge until choices unlock.", 40, 64);
}

// ------------------------------------------------------------
// Dodger
// ---------------------------------------------------------
function updateDodger(arena) {
  let dx = 0, dy = 0;

  if (keyIsDown(LEFT_ARROW) || keyIsDown(65)) dx -= 1;  // A
  if (keyIsDown(RIGHT_ARROW) || keyIsDown(68)) dx += 1; // D
  if (keyIsDown(UP_ARROW) || keyIsDown(87)) dy -= 1;    // W
  if (keyIsDown(DOWN_ARROW) || keyIsDown(83)) dy += 1;  // S

  if (dx !== 0 && dy !== 0) { dx *= 0.7071; dy *= 0.7071; }

  dodger.x += dx * dodger.speed;
  dodger.y += dy * dodger.speed;

  dodger.x = clamp(dodger.x, arena.x + dodger.r, arena.x + arena.w - dodger.r);
  dodger.y = clamp(dodger.y, arena.y + dodger.r, arena.y + arena.h - dodger.r);
}

function drawDodger() {
  noStroke();
  fill(90, 210, 220, 70);
  ellipse(dodger.x, dodger.y, dodger.r * 5.0, dodger.r * 5.0);

  fill(90, 210, 220);
  ellipse(dodger.x, dodger.y, dodger.r * 2, dodger.r * 2);
```

```
}

// ------------------------------------------------------------
// Bullets
// ------------------------------------------------------------
function updateBullets(arena) {
  spawnTimer -= deltaTime || 16;

  if (spawnTimer <= 0) {
    spawnBullet(arena);
    const depth = (gameState.history && gameState.history.length) || 0;
    spawnTimer = clamp(520 - depth * 35, 260, 520);
  }

  for (let i = bullets.length - 1; i >= 0; i--) {
    const b = bullets[i];
    b.x += b.vx;
    b.y += b.vy;

    if (
      b.x < arena.x - 80 || b.x > arena.x + arena.w + 80 ||
      b.y < arena.y - 80 || b.y > arena.y + arena.h + 80
    ) {
      bullets.splice(i, 1);
    }
  }
```

```
}

function spawnBullet(arena) {
  const y = random(arena.y + 20, arena.y + arena.h - 20);
  const x = arena.x + arena.w + 40;

  const targetX = dodger.x;
  const targetY = dodger.y + random(-80, 80);

  const angle = atan2(targetY - y, targetX - x);
  const depth = (gameState.history && gameState.history.length) || 0;

  const speed = clamp(4.2 + depth * 0.35, 4.2, 7.6);
  bullets.push({
    x, y,
    r: random(7, 11),
    vx: cos(angle) * speed,
    vy: sin(angle) * speed,
  });
}

function drawBullets() {
  for (const b of bullets) {
    noStroke();
    fill(255, 140, 90, 140);
    push();
    translate(b.x, b.y);
    rotate(atan2(b.vy, b.vx));
    rectMode(CENTER);
    rect(10, 0, b.r * 3.2, b.r * 0.5, 8);
    pop();

    fill(255, 210, 140);
    ellipse(b.x, b.y, b.r * 2, b.r * 2);

    fill(255, 255, 255, 120);
    ellipse(b.x - b.r * 0.3, b.y - b.r * 0.3, b.r * 0.7, b.r * 0.7);
  }
}

// -----------------------------------------------------------
// Collision + consequence
// -----------------------------------------------------------
function checkCollision() {
  for (const b of bullets) {
    if (dist(dodger.x, dodger.y, b.x, b.y) < dodger.r + b.r) return true;
  }
  return false;
}

function onHit() {
  player.trust = clamp((player.trust || 0) - 12, 0, 100);
  initRoom();
}

// ------------------------------------------------------------
// Survival gate
// ------------------------------------------------------------
function updateSurvival(arena) {
  const elapsed = (millis() - room.startedAtMs) / 1000;
  const remaining = Math.max(0, room.secondsToSurvive - elapsed);

  const barW = arena.w - 40, barH = 10;
  const bx = arena.x + 20, by = arena.y + 18;

  noStroke();
  fill(255, 255, 255, 55);
  rectMode(CORNER);
  rect(bx, by, barW, barH, 8);

  const pct = clamp(1 - remaining / room.secondsToSurvive, 0, 1);
  fill(90, 210, 220, 160);
  rect(bx, by, barW * pct, barH, 8);

  fill(220);
  textAlign(LEFT, TOP);
  textSize(12);
  text(`UNLOCK IN: ${remaining.toFixed(1)}s`, bx, by + 14);
  if (remaining <= 0 && !room.survived) {
    room.survived = true;
    player.trust = clamp((player.trust || 0) + 3, 0, 100);
  }
}

// ------------------------------------------------------------
// Story panel + choices
// ------------------------------------------------------------
function drawStoryPanel() {
  const node = STORY[gameState.nodeId];

  const panel = { x: width * 0.62, y: 80, w: width * 0.35, h: height - 140 };

  rectMode(CORNER);
  noStroke();
  fill(255, 255, 255, 18);
  rect(panel.x, panel.y, panel.w, panel.h, 16);

  fill(245);
  textAlign(LEFT, TOP);
  textSize(20);
  text(node.title || "Story", panel.x + 18, panel.y + 16);

  fill(210);
```

```
textSize(14);
text(node.body || "", panel.x + 18, panel.y + 48, panel.w - 36, 220);

if (!node.choices || node.choices.length === 0) {
  fill(190);
  textSize(14);
  text("END.", panel.x + 18, panel.y + 280);
  return;
}

if (!room.survived) {
  fill(170);
  textSize(14);
  text("Survive the arena to unlock choices.", panel.x + 18, panel.y + 280);
  return;
}

const btns = buildChoiceButtons(node);
for (let i = 0; i < btns.length; i++) {
  drawChoiceButton(btns[i]);

  fill(180);
  textAlign(LEFT, CENTER);
  textSize(12);
  text(`${i + 1}.`, btns[i].x - btns[i].w / 2 + 10, btns[i].y);
}
```

```
}

function buildChoiceButtons(node) {
  const baseX = width * 0.62 + (width * 0.35) / 2;
  const startY = 360;
  const gap = 76;

  const btns = [];
  for (let i = 0; i < node.choices.length; i++) {
    btns.push({
      x: baseX,
      y: startY + i * gap,
      w: width * 0.33,
      h: 56,
      label: node.choices[i].label,
    });
  }
  return btns;
}

function isAnyChoiceHovered() {
  const node = STORY[gameState.nodeId];
  if (!node || !node.choices) return false;
  const btns = buildChoiceButtons(node);
  for (const b of btns) if (isHover(b)) return true;
  return false;
```

```
}


function applyChoice(choice) {
  player.trust = clamp((player.trust || 0) + (choice.trustDelta || 0), 0, 100);


  gameState.history = gameState.history || [];
  gameState.history.push(gameState.nodeId);
  gameState.lastChoice = choice.id;


  gameState.nodeId = choice.next;


  room.startedAtMs = 0;
}
```