

Process & Decision Documentation

Project/Assignment Decisions

In the initial stages of the game's development, I used GenAI to help with keyboard input and movement. It suggested I incorporate a shift + arrow key function to move similarly to Celeste. When I tested this as a user I found that I was confused over which key to start the game with as the instructions were not clear. This led me to add various areas of text to the game, indicating how to play, score, level, and feedback as game is being played.

Side Quests and A4 (Individual Work)

In order to fit the caption that appeared when the player hit the mountain, I had to do several iterations of changes. Initially, the caption bled outside of the frame. To fix its visibility, I increased the size of the frame which, being a square, only required the alteration of a single property instead of my other attempts to adjust the length of the frame through width * 2. I found that this adjustment of the frame's size made the game more visually pleasing in its larger scale than the original smaller size before this issue.

Role-Based Process Evidence

Side Quest: Week 4 – Mountain Maze

Name: Chloe Victoria Hodgins

Role(s): Programmer

Primary responsibility for this work: Created a grid-based maze with arrays, loops and an automatic second level, inspired by Celeste.

Goal of Work Session

- Code a playable level using arrays and JSON data.
- Use loops to place tiles and obstacles.
- Revising visual verbal feedback after playtesting.
- Adjusting the caption size and frame size to fit caption.
- Design a self-loading second level when first level is reached.

Tools, Resources, or Inputs Used

- Week 4 lecture slides
- Playtesting feedback from myself and a friend
- GenAI tools (ChatGPT 5.2)

GenAI Documentation

Date Used: February 10, 2026

Tool Disclosure: ChatGPT 5.2

Purpose of Use: Understanding movement logic and arrays and improving visuals (such as sizing).

Summary of Interaction: I used ChatGPT 5.2 to clarify how the array corresponds with tile types, ideas for communicating to the player game constraints, and to adjust the size of the frame and caption.

Human Decision Point(s): I rejected the GenAI's suggested methods for fitting the caption into the frame, such as through a wrapped text box as the options provided were usually overly complicated.

Integrity & Verification Note: I separately tested suggested code made by ChatGPT using the Live Server and additionally referenced the lecture slides to compare.

Scope of GenAI Use: GenAI did not contribute to the text added, basic functions such as showMessage and draw, the overall concept, level design, and final layout decisions as these were my own. I would look at GenAI code suggestions and use it as inspiration for my work rather than copied word for word or for understanding suggestions given.

Limitations or Misfires: Movement was misunderstood as a suggested text area indicated the use of the SHIFT key to move although it had to be paired with the arrow key. I had to clarify this, which led me to including text at the top of the game, indicating how to move, and captions after that. The text from the caption initially also overflowed and required manual adjustments to the frame size.

Summary of Process (Human + Tool)

- Testing with myself then revision, then testing with a friend, more revision
- Uncertainty with trying to fit in the caption – inquired using ChatGPT 5.2

Decision Points & Trade-offs

Increasing Frame Width: In order to fit the caption for the mountain being hit, I considered adjusting the caption size or adjusting just the length of the frame, rather than the whole frame. I found that adjusting the tile size (entire frame) made the game more visually appealing and enjoyable to play.

Verification & Judgement

I playtested often to check my decisions (myself and friend). I made sure the player stayed in the walls of the maze, hitting a mountain would reset the level and player location, the dash function worked, and that the second level began automatically after completion of the first.

Limitations, Dead Ends, or Open Questions

There are no sound effects or animations. If I were to expand on this side quest further I would improve it through the addition of sound of better visuals.

Appendix

Prompt:

I am a third year University of Waterloo student taking Global Business and Digital Arts. In my GDBA302 class, "Global Digital Project 2", I have an in-class assignment in which I must generate a level using arrays or JSON data and use loops to dynamically place tiles, obstacles, or platforms. This is meant to be completed at the end of the class. The first part of the class was spent playing Celeste and Codenames the boardgame. How could I incorporate an aspect related to one of these games in my own?

Response:

You can keep the **coding requirement** super clear (level generated from an array/JSON + loops), and then add **one recognizable “Celeste” or “Codenames” twist** on top so it feels connected to class. Week 4’s prompt explicitly wants level generation with arrays/JSON + loops, and suggests a 2nd level auto-load as a bonus.

Here are strong, easy-to-implement ideas that fit the time limit:

Celeste-inspired (platformer/precision vibe)

1) “Dash crystal” tile (most iconic, fast to code)

- Your array has tile types:
0 = empty, 1 = wall/platform, 2 = spikes, 3 = dash crystal, 4 = goal
- Use loops to draw tiles.
- Mechanic: player has **1 dash**. If they touch a crystal tile, dash resets.
- Why it reads as Celeste: dash management + crystals is instantly recognizable.

2) Spikes that punish sloppy movement

- Add 2 = spikes tiles placed by the array.
- If player overlaps spikes, respawn to start.
- Super simple but gives that “tight platforming” vibe.

3) “Strawberry” optional collectible

- Add 5 = collectible tiles.
- Track `score++` when collected.
- It’s extra polish without making the game complex.

Codenames-inspired (word-grid/hidden roles vibe)

1) Word tiles with “red/blue/neutral/assassin”

- Your grid generates *cards* (rectangles) using loops.
- Each tile has a word (can be from an array), and a hidden role:
 - Blue team, Red team, Neutral, Assassin
- Click to flip a card and reveal color.
- Why it reads as Codenames: flipping cards + assassin is the whole game feel.

2) “One-word clue” overlay (light version)

- At the top, display a “clue” like: “**Cold: 2**”
 - Player has to click 2 related words (you can hardcode which ones count as correct).
 - This is very doable in a short time because it’s mostly UI + click checks.
-

Easiest combos (if you want it to feel creative but still finish)

Combo A (recommended): Celeste maze + dash crystal

- You build a tile maze from a 2D array (meets requirement).
- You add *one* Celeste mechanic: dash resets when touching crystal.
- Optional: auto-load a second level when reaching goal (bonus).

Combo B: Codenames grid from JSON

- Your JSON contains an array of words + roles.
- Loops place cards.
- Clicking reveals role colors.
- Very “clean” use of JSON.

Quick example of what your array could look like (Celeste version)

```
// 0 empty, 1 wall, 2 spikes, 3 dashCrystal, 4 goal
let level = [
  [1,1,1,1,1,1],
  [1,0,0,0,3,1],
  [1,0,1,0,2,1],
  [1,0,1,0,0,1],
  [1,0,0,0,4,1],
  [1,1,1,1,1,1],
];
```

Then loops read each number and draw the correct tile + run collision logic.

If you tell me which direction you want (**Celeste platformer** vs **Codenames word grid**), I'll write you a tight plan (and starter code structure) that hits the requirement and feels themed—without becoming a huge project.

Prompt: I really like the Celeste maze idea with the incorporation of an auto-loading second level. How should I go about doing this?

Response:

Awesome — this is a clean Week 4 solution because it hits **arrays + loops + dynamic tiles**, and the **bonus auto-load level**.

Here's the simplest, most reliable way to build it (maze + player + goal + auto level-switch), with a Celeste-ish touch.

1) Represent each level as a 2D array (numbers = tile types)

Use a legend like:

- 0 = floor
- 1 = wall
- 2 = spikes (Celeste danger)
- 3 = dash crystal (optional but very Celeste)
- 4 = goal/exit

```
const LEVELS = [
  [
    [1,1,1,1,1,1,1],
    [1,0,0,0,0,3,1],
    [1,0,1,1,0,1,1],
    [1,0,0,1,0,0,1],
    [1,1,0,1,2,0,1],
    [1,0,0,0,0,4,1],
    [1,1,1,1,1,1,1],
  ],
  [
    [1,1,1,1,1,1,1],
    [1,0,0,0,1,0,3,1],
    [1,0,1,0,1,0,1,1],
    [1,0,1,0,0,0,0,1],
    [1,0,1,1,1,1,0,1],
    [1,0,0,0,2,0,4,1],
    [1,1,1,1,1,1,1,1],
  ]
];
```

2) Track which level you're on + load it

Key idea: a **loadLevel(index) function** that resets:

- the current grid
- canvas size
- player start position
- any state (dash, etc.)

```

let tileSize = 40;

let levelIndex = 0;
let grid;

let player;

function setup() {
  loadLevel(0);
}

function loadLevel(idk) {
  levelIndex = idk;
  grid = LEVELS[levelIndex];

  // set player start (simple: first open tile you find, or fixed spot)
  player = { r: 1, c: 1, hasDash: true };

  createCanvas(grid[0].length * tileSize, grid.length * tileSize);
}

```

If you want a “real” start tile, you can add 5 = start later, but for an in-class version, r=1, c=1 is totally fine.

3) Draw the level using loops (this is the graded core)

```

function draw() {
  background(230);

  for (let r = 0; r < grid.length; r++) {
    for (let c = 0; c < grid[r].length; c++) {
      let t = grid[r][c];

      // floor base
      fill(245);
      rect(c * tileSize, r * tileSize, tileSize, tileSize);

      if (t === 1) { // wall
        fill(50, 80, 120);
        rect(c * tileSize, r * tileSize, tileSize, tileSize);
      } else if (t === 2) { // spikes
        fill(200, 70, 70);
        rect(c * tileSize, r * tileSize, tileSize, tileSize);
      } else if (t === 3) { // dash crystal
        fill(80, 200, 255);
        ellipse(c * tileSize + tileSize/2, r * tileSize + tileSize/2,
tileSize * 0.6);
      } else if (t === 4) { // goal
        fill(120, 220, 120);
        rect(c * tileSize, r * tileSize, tileSize, tileSize);
      }
    }
  }
}

```

```

    // draw player
    fill(255, 140, 200);
    rect(player.c * tileSize + 6, player.r * tileSize + 6, tileSize - 12,
tileSize - 12);

    // simple HUD
    fill(0);
    textSize(14);
    text(`Level ${levelIndex + 1}`, 10, 18);
    text(`Dash: ${player.hasDash ? "READY" : "USED"}`, 10, 36);
}

```

4) Movement + collision (grid-based, super reliable)

This is the fastest way to make it feel like a maze game.

```

function keyPressed() {
    let dr = 0, dc = 0;
    if (keyCode === UP_ARROW) dr = -1;
    if (keyCode === DOWN_ARROW) dr = 1;
    if (keyCode === LEFT_ARROW) dc = -1;
    if (keyCode === RIGHT_ARROW) dc = 1;

    // dash (Celeste-ish): hold SHIFT to move 2 tiles if you have dash
    let steps = 1;
    if (keyIsDown(SHIFT) && player.hasDash) {
        steps = 2;
        player.hasDash = false;
    }

    tryMove(dr, dc, steps);
}

```

And the move logic:

```

function tryMove(dr, dc, steps) {
    for (let i = 0; i < steps; i++) {
        let nr = player.r + dr;
        let nc = player.c + dc;

        let tile = grid[nr][nc];

        if (tile === 1) return; // wall blocks

        // move
        player.r = nr;
        player.c = nc;

        // react to tile
        if (tile === 2) { // spikes
            respawn();
            return;
        }
    }
}

```



```

    }

    if (tile === 3) { // dash crystal
      player.hasDash = true;
      // optional: "consume" crystal so it disappears
      grid[player.r][player.c] = 0;
    }

    if (tile === 4) { // goal -> next level
      nextLevel();
      return;
    }
  }
}

function respawn() {
  player.r = 1;
  player.c = 1;
  player.hasDash = true;
}

```

5) Auto-load the second level (the bonus)

```

function nextLevel() {
  let next = levelIndex + 1;

  if (next < LEVELS.length) {
    loadLevel(next); // AUTO switch
  } else {
    // finished all levels
    console.log("You win!");
    // you can show a win screen if you want
  }
}

```

That's it. When the player steps on the goal tile, it automatically loads the next level.

What to keep *very* simple (so you finish in-class)

- Use **grid movement** (tile-to-tile), not physics.
 - Use **one player rectangle**, not animations.
 - Use **2 levels max**.
 - Optional Celeste flair: dash crystal + spikes (already above).
-

If you want, paste your current `sketch.js` and I'll merge this into your file structure (or match whatever Week 4 starter code you're using).

Prompt:

Could you explain the 1 0 1 walls after const LEVELS = I would like to understand how the legend corresponds with the different tile types in code.

Response:

YES. I love this question because once this clicks, level design suddenly makes sense instead of feeling like random numbers.

Let's slow it down and decode it properly.

What This Actually Is

When you write:

```
const LEVELS = [  
  [  
    [1,1,1,1,1],  
    [1,0,0,0,1],  
    [1,0,1,0,1],  
    [1,0,0,4,1],  
    [1,1,1,1,1]  
  ]  
];
```

You are creating a **2D array**.

Think of it like a spreadsheet or pixel art grid.

Each number is ONE tile.

Visualize It Like a Map

Take this row:

```
[1,0,1,0,1]
```

That means:

| 1 | 0 | 1 | 0 | 1 |

Which becomes:

| Wall | Floor | Wall | Floor | Wall |

So the maze is literally drawn from numbers.

The Legend = Translation Key

The legend is just a rule you invented:

```
// 0 = floor
// 1 = wall
// 2 = spikes
// 3 = dash crystal
// 4 = goal
```

That means:

When the code sees a number, it decides what to draw.

Where the Translation Happens

Inside your draw loop:

```
let t = grid[r][c];
```

This grabs the number at row `r` and column `c`.

Then:

```
if (t === 1) {
  fill(50, 80, 120); // wall color
  rect(...)
}
else if (t === 2) {
  fill(200, 70, 70); // spikes
}
else if (t === 3) {
  fill(80, 200, 255); // dash crystal
}
```

```
else if (t === 4) {  
    fill(120, 220, 120); // goal  
}
```

So:

- If tile number = 1 → draw wall
- If tile number = 2 → draw spikes
- If tile number = 3 → draw crystal
- If tile number = 4 → draw goal
- If tile number = 0 → just leave floor

The numbers themselves do nothing.
The **if statements** give them meaning.

You are basically saying:

“When I see a 1, treat it as a wall.”

Why 1s Are Around the Edges

Look at this:

```
[1, 1, 1, 1, 1],  
[1, 0, 0, 0, 1],  
[1, 0, 1, 0, 1],  
[1, 0, 0, 4, 1],  
[1, 1, 1, 1, 1]
```

All the outside numbers are 1.

That creates a border of walls so the player can't walk off the map.

It's like building the container of the maze.

What grid[r][c] Means

This is important:

- `r` = row (vertical position)

- `c` = column (horizontal position)

So:

```
grid[2][3]
```

Means:

- Row 2
- Column 3

Which corresponds to a tile on screen at:

```
x = c * tileSize  
y = r * tileSize
```

That's how array positions become visual positions.

Think of It Like Pixel Art

Each number is like a colored pixel.

Instead of:

- 0 = black
- 1 = white

You're doing:

- 1 = wall
- 0 = floor
- 2 = spikes
- 3 = dash crystal
- 4 = goal

It's literally symbolic drawing.

Why This Is Powerful

Instead of writing:

```
rect(40, 80, 40, 40);  
rect(80, 80, 40, 40);  
rect(120, 80, 40, 40);
```

You let the loop read the grid and generate everything automatically.

That's why this satisfies the assignment:

- It uses arrays
 - It uses loops
 - Tiles are dynamically placed
-

If You Changed the Legend...

You could literally change:

```
// 1 = lava instead of wall
```

And now your maze becomes a lava world.

The numbers don't change.
The meaning does.

If you want, I can:

- Draw a tiny grid and show exactly how the nested loops move through it
- Or explain how collision uses those same numbers

You're asking the right questions btw. This is exactly how you move from "copying code" to actually understanding game systems.

Prompt:

When I press "shift" nothing happens. What is happening here and how to fix?

Response:

Totally normal — in your current code, **SHIFT by itself is not supposed to do anything**. The dash only triggers when an **arrow key press happens while SHIFT is being held**.

What's happening

Your `keyPressed()` only runs when a key is pressed (like an arrow). Inside that function you have:

```
if (keyIsDown(SHIFT) && player.hasDash) {  
    steps = 2;  
    player.hasDash = false;  
}
```

So:

- Pressing **SHIFT alone** → no movement code runs → “nothing happens”
- Holding **SHIFT + pressing an ARROW** → dash tries to move 2 tiles

Fix / Improve (so it feels like it's working)

1) Make sure you're testing it correctly

- ✓ Click the canvas once (so it has focus)
- ✓ Hold SHIFT, then press an arrow key

If you only tap SHIFT, you'll see nothing.

2) If it *still* doesn't dash: use the `keycode 16` (more reliable)

Sometimes `SHIFT` constant can be inconsistent depending on environment. Replace:

```
keyIsDown(SHIFT)
```

with:

```
keyIsDown(16) // 16 is SHIFT
```

So:

```
if (keyIsDown(16) && player.hasDash) {  
    steps = 2;  
    player.hasDash = false;  
}
```

3) Prevent the browser from eating your key input (common on laptops)

Browsers sometimes hijack arrow keys / shift+arrow.

At the **end** of `keyPressed()` add:

```
return false;
```

Example:

```
function keyPressed() {  
  // ... your movement code ...  
  return false; // stops browser default behaviour  
}
```

4) Dash might be “working” but blocked by a wall on the 2nd tile

Your dash moves **tile-by-tile**, so if the second tile is a wall, you’ll only move 1 tile (which can feel like “no dash”).

Quick test: print the step count:

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

Or add a dash message on screen when dash triggers.

Quick “best” dash snippet to paste

Replace your dash check with:

```
let steps = 1;  
if (keyIsDown(16) && player.hasDash) {  
  steps = 2;  
  player.hasDash = false;  
}
```

And add `return false;` at the end of `keyPressed()`.

If you paste your current `keyPressed()` function, I’ll point to the exact line that’s causing your particular “SHIFT does nothing” behavior (whether it’s focus, browser hijack, or the dash getting blocked).

Prompt:

Oh! Okay that makes sense. How could I communicate to the user in text underneath "Dash: READY (hold SHIFT)" to use the arrow keys to move?

Response:

Ahhh yes. This is such a good polish move — it bumps your **Design & Clarity** marks immediately because now the interaction is obvious.

Right now you probably have something like:

```
text(`Dash: ${player.hasDash ? "READY (hold SHIFT)" : "USED"}`, 12, 40);
```

All you need to do is add another `text()` line under it.

Simple Version (clean and done in 10 seconds)

Right under your dash text, add:

```
text("Move with ARROW KEYS", 12, 60);
```

So your HUD section becomes:

```
fill(255);
textSize(14);
text(`Level ${levelIndex + 1}/${LEVELS.length}`, 12, 20);
text(`Dash: ${player.hasDash ? "READY (hold SHIFT)" : "USED"}`, 12, 40);
text("Move with ARROW KEYS", 12, 60);
```

Done. Clean. Clear.

Slightly More Polished Version (looks intentional)

You could separate movement + dash instructions:

```
text("Move: ARROW KEYS", 12, 60);
text("Dash: HOLD SHIFT + ARROW", 12, 80);
```

This is extra clear because it explains that SHIFT must be combined with movement.

Even Nicer: Fade Instruction After First Move

If you want it to feel more “game-like” (not required but cool):

Add a variable:

```
let hasMoved = false;
```

In `tryMove()` after the player moves:

```
hasMoved = true;
```

Then in your HUD:

```
if (!hasMoved) {  
  fill(255);  
  text("Move: ARROW KEYS", 12, 60);  
  text("Dash: HOLD SHIFT + ARROW", 12, 80);  
}
```

Now the instructions disappear after they move once — very clean UX move.

One Important Thing

If your text ever overlaps tiles, make sure:

- It's drawn **after** the tiles
- Or you add a small semi-transparent background rectangle behind it

Example:

```
fill(0, 150);  
rect(0, 0, 260, 100);  
  
fill(255);  
text("Move: ARROW KEYS", 12, 60);
```

That gives you a proper HUD panel.

If you want, I can also show you how to:

- Center the instructions at the bottom
- Add a tiny animated arrow icon

- Or make it feel more “Celeste mountain tutorial energy” instead of classroom energy 😊

Prompt:

I would like to communicate at the bottom of the screen as soon as the player hits the arrow key the first time that they should try to get the blue circle to gain a point. I would also like that if the player hits the red triangle (moving back to the beginning) there is a message such as "Oh no! You hit the mountain and got catapulted back to the beginning!!)

Response:

Perfect — do this with a tiny **message system**: a `message` string + a `messageTimer` that counts down, so text appears at the bottom for a couple seconds.

1) Add these globals near the top of `sketch.js`

```
let hasMovedOnce = false;

let message = "";
let messageTimer = 0; // frames to show message
```

2) Add this helper function (anywhere in the file)

```
function showMessage(txt, seconds = 2.5) {
  message = txt;
  messageTimer = int(seconds * 60); // assuming ~60fps
}
```

3) Draw the message at the bottom (inside `draw()`, after you draw the player + HUD)

Put this near the end of `draw()`:

```
// bottom message UI
if (messageTimer > 0) {
  messageTimer--;

  // dark bar behind the text for readability
  noStroke();
  fill(0, 160);
  rect(0, height - 44, width, 44);

  fill(255);
  textSize(14);
  textAlign(CENTER, CENTER);
  text(message, width / 2, height - 22);
  textAlign(LEFT, BASELINE);
}
```

4) Trigger the “blue circle” message on the FIRST arrow-key move

In your `tryMove(dr, dc, steps)` function, right after you update the player position (right after `player.r = nr; player.c = nc;`), add:

```
if (!hasMovedOnce) {
    hasMovedOnce = true;
    showMessage("Try to reach the blue circle to gain a point!");
}
```

So it'll show the moment they successfully move for the first time.

5) Show the “catapulted” message when they hit the red triangle

In your spikes logic inside `tryMove()`:

```
if (tile === 2) {
    showMessage("Oh no! You hit the mountain and got catapulted back to the beginning!!");
    respawn();
    return;
}
```

6) Give a point when they collect the blue circle (dash crystal)

Add a score variable near the top:

```
let score = 0;
```

When you load a level, decide if score resets or carries over. If you want it to carry across levels, **don't reset it** in `loadLevel()`. If you want it per-level, reset there.

Then in your crystal logic:

```
if (tile === 3) {
    player.hasDash = true;
    score += 1;
    showMessage("Nice! +1 point!");
    grid[player.r][player.c] = 0; // remove crystal
}
```

And show the score in your HUD (in `draw()` where you print Level/Dash):

```
text(`Score: ${score}`, 12, 60);
```

(If you add score, bump your “Move with ARROW KEYS” line down a bit, like `y=80`.)

Quick note (so your first-move message actually triggers)

This only fires when a move succeeds. If the player presses an arrow into a wall first, they won't "move," so the message won't appear yet — which is usually fine. If you want it to appear on *any* arrow press (even if blocked), tell me and I'll tweak it.

If you paste your current `draw()` + `tryMove()` + `respawn()` functions, I'll stitch these edits into your exact code so nothing conflicts.

Prompt:

The text that appears at the bottom of the frame when the player hits the mountain is too long and is being cut off. How could I fix this?

Response:

Ooo yes let's scale this mountain up 🏔️🌟

Right now your canvas size is probably being set like this inside `loadLevel()`:

```
createCanvas(grid[0].length * tileSize, grid.length * tileSize);
```

So the **canvas size depends on tileSize**.

✓ Easiest Way to Make Everything 2x Bigger

Just double `tileSize`.

If you currently have:

```
let tileSize = 48;
```

Change it to:

```
let tileSize = 96;
```

That's it.

Because your canvas size is calculated from `tileSize`, the whole world will scale up automatically.

Why This Works

Your tile positions are calculated like this:

```
x = c * tileSize;  
y = r * tileSize;
```

So if `tileSize` doubles:

- Tiles get bigger
- Player gets bigger
- Canvas gets bigger
- Everything scales proportionally

You designed it properly, so scaling is easy.

If Text Looks Tiny After Scaling

If you double tile size, your world grows — but your text size stays the same.

So increase text size slightly:

```
textSize(18); // instead of 14
```

For HUD.

And maybe:

```
textSize(24);
```

For win screen.



Alternative: Double the Entire Canvas Without Changing Tile Size

If instead you want:

- Same tile size
- But more space around the game

You can multiply canvas size manually:

```
createCanvas(grid[0].length * tileSize * 2,  
             grid.length * tileSize * 2);
```

But this stretches empty space — not the game itself.

For your assignment, increasing `tileSize` is the correct move.



Pro Tip

If you want clean scaling control, define:

```
let SCALE = 2;  
let tileSize = 48 * SCALE;
```

Then later you can easily adjust scale without rewriting numbers.

If you want, I can also show you how to:

- Add smooth camera zoom
- Or scale using `scale()` instead of changing `tileSize`

But for Week 4 side quest energy, doubling `tileSize` is the move.

Prompt:

Where should I put `createCanvas`?

Response:

Great question — and this actually matters a lot in p5.

Short answer:

👉 **`createCanvas()` must be inside `setup()`**, or inside a function that is called from `setup()` (like your `loadLevel()`).

Why this works:

- `setup()` calls `loadLevel()`
- `loadLevel()` creates the canvas
- When you switch levels, `loadLevel()` runs again
- Canvas resizes properly

Prompt:

How to make the message after hitting the mountain and then moving with the arrow key go away?

Response:

Ooo okay this is a good UX instinct — you don't want the "Oh no!" message lingering once they start moving again.

Right now your message system probably works like this:

```
let message = "";  
let messageTimer = 0;
```

And you display it while:

```
if (messageTimer > 0) { ... }
```

So the message disappears when `messageTimer` hits 0.

But you want it to disappear immediately when they move again.

Easiest Fix: Clear Message on Arrow Key Press

Inside your `keyPressed()` function, right after detecting arrow movement, add:

```
// If player presses an arrow key, clear message
if (
  keyCode === UP_ARROW ||
  keyCode === DOWN_ARROW ||
  keyCode === LEFT_ARROW ||
  keyCode === RIGHT_ARROW
) {
  message = "";
  messageTimer = 0;
}
```

Put this near the top of `keyPressed()` before movement logic.

Now:

- They hit spike → message appears
- They press arrow again → message immediately disappears

Clean and simple.

Slightly Better Version (Only Clear if They Actually Move)

If you want it to disappear only when a move succeeds (not if they press into a wall):

Inside `tryMove()` after successful movement:

```
player.r = nr;
player.c = nc;

// Clear message when they successfully move
message = "";
messageTimer = 0;
```

This feels more intentional because:

- Message stays if they keep hitting a wall

- Disappears once they actually start playing again
-

✳ Even Cleaner: Only Clear the “Mountain” Message

If you later have multiple types of messages and don’t want to clear everything, you can do:

```
if (message.includes("mountain")) {  
  message = "";  
  messageTimer = 0;  
}
```

But that’s optional — probably overkill for Week 4.

My Recommendation for This Assignment

Use the “clear on successful move” version inside `tryMove()`.

It feels polished and not glitchy.

If you paste your `keyPressed()` and `tryMove()` functions, I’ll point to the exact line where to insert it so nothing breaks.