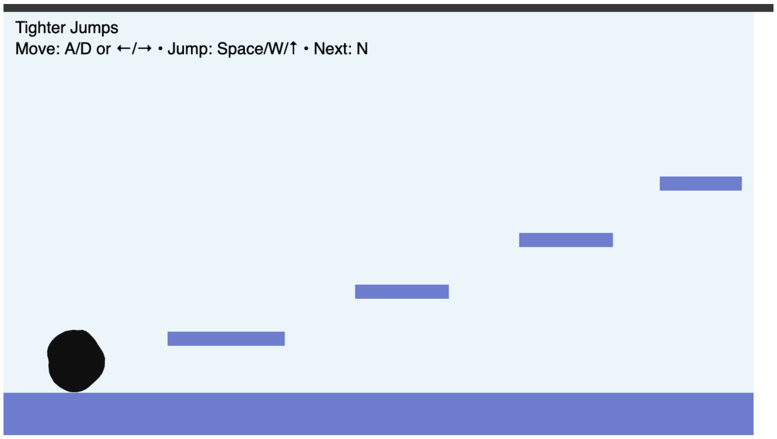
Process & Decision Documentation

Project/Assignment Decisions

- I used GenAI to generate a new level instead of designing it entirely from scratch to save time. I then manually edited the platform positioning and spacing in the generated code to ensure the level design remained my own work.

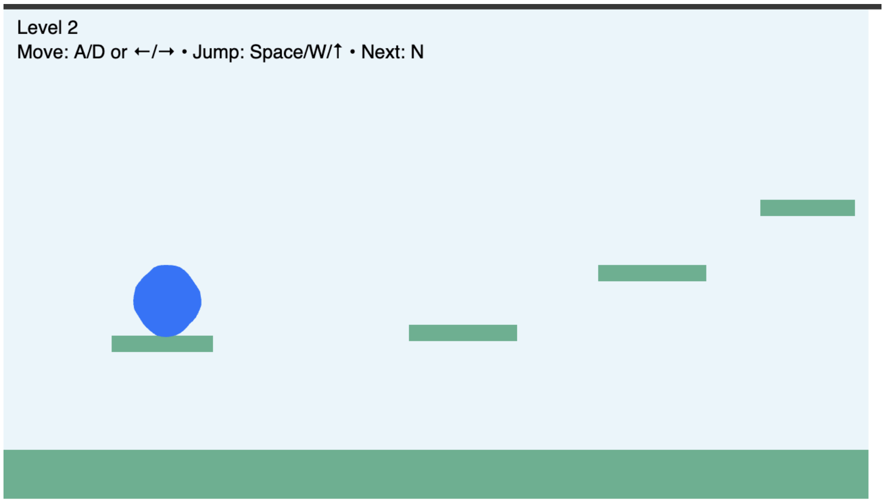
Role-Based Process Evidence

The original level generated by AI:



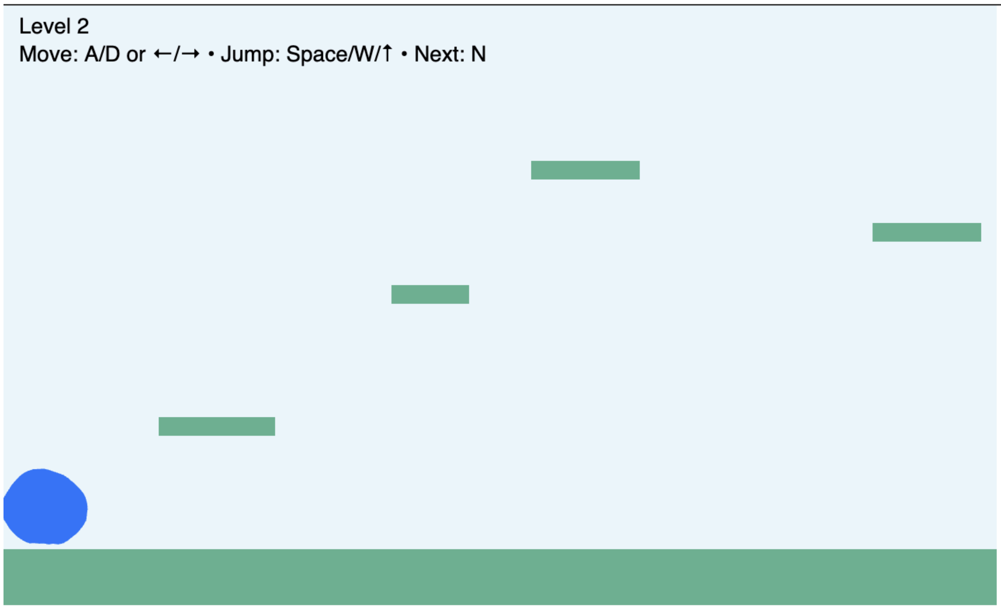
Tighter Jumps
Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N

Change #1: I began by changing the colour of the platforms and the blob. I also changed the first platform to be higher and narrower. I used trial and error to figure out the best position for the platform as I wanted to ensure it was challenging yet still reachable by the player.

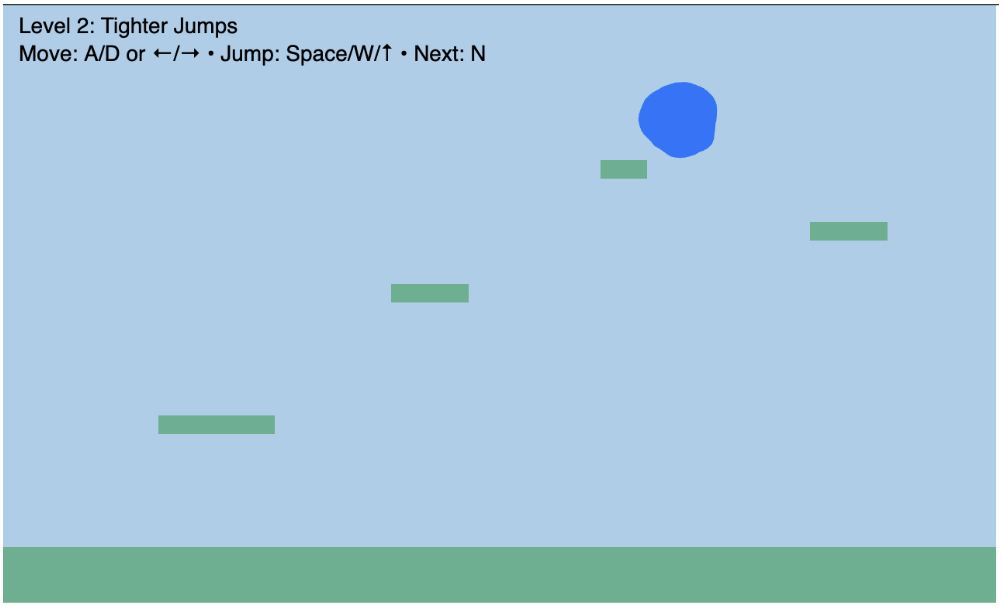


Level 2
Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N

Change #2: I lowered the base of the game so there could be more vertical spacing between platforms. I also adjusted the positioning and spacing of platforms 2, 3, and 4.


Level 2
Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N

Change #3 (final version of the game): I made platform three narrower to increase difficulty, and I adjusted the spacing and positioning of platforms 3 and 4. I also changed the background colour.


Level 2: Tighter Jumps
Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N

Entry Header

Name: Nila Sivapragasam

Role(s): Programmer

Primary responsibility for this work: Programming

Goal of Work Session:

Briefly describe what you were trying to accomplish during this phase of the assignment.

Examples:

- To generate a new level of the platform jumping game using JSON data

Tools, Resources, or Inputs Used

- Lecture Notes
- GDBA 302 Week 4 Example 5 Code
- Microsoft Copilot

GenAI Documentation

Date Used: February 10th 2026

Tool Disclosure: Microsoft Copilot

Purpose of Use: I used GenAI to clarify the lecture content and better understand the Side Quest prompt. I used it to generate the layout of a new level, which I edited and refined myself.

Summary of Interaction: GenAI explained the concepts of classes and JSON files, explained what needed to be done for this week's Side Quest, and provided guidance on how to approach this task. It also used the Example 05 files to generate a new level for the game.

Human Decision Point(s): I manually edited the generated code by changing the colour scheme and adjusting the platforms' positioning and spacing to ensure the new level was still my own design. I also lowered the base of the game to create more vertical space between the platforms. Additionally, I removed the comments added by Copilot to the levels.json file, as JSON doesn't support comments.

Integrity & Verification Note: I carefully read through all GenAI outputs to ensure they were accurate and aligned with what we learned in class. Additionally, I reviewed the generated code to make sure I fully understood how it worked, and I tested the code through gameplay to confirm it ran properly and met the Side Quest requirements.

Scope of GenAI Use: I used GenAI to explain class concepts, clarify Side Quest instructions and to generate a new level. However, I edited the generated code to change the colour scheme and alter the platforms' spacing, positioning, and width.

Limitations or Misfires: Although Copilot made a new level, it was not more difficult than the original level in the Example 5 files, so many changes were needed. It also added comments to the JSON file, which are not supported.

Appendix

*Note: My messages are highlighted in orange and Copilot's responses are highlighted in green.

My message: In my GBDA 302 class, we are learning about classes and JSON files. I've attached the lecture slides from this week but I'm still struggling to understand the concepts of classes and JSON files, can you explain it to me?

Copilot Response: Absolutely — I can break down **classes** and **JSON files** in a way that connects directly to your **GBDA 302 Week 4 slides** and the kind of code you're writing.
No jargon, no assumptions — clear, visual, and example-based.

🎯 What You Need to Understand (in Simple Terms)

The lecture slides are teaching you two BIG concepts:

1. **JSON = where your DATA lives**
   (things like level layout, player start position, gravity, platforms…)
2. **Classes = where your LOGIC lives**
   (functions + data bundled together to describe one "thing," like a Level or Player)

Together, JSON + classes help you avoid messy, repetitive code and make your games scalable (multiple levels, characters, layouts).

Let’s break them down.

🟦 1. What Is JSON? (And Why Your Course Uses It)

✔ JSON stands for *JavaScript Object Notation*

It’s just a file that stores data in a structured way.

✔ JSON is basically a text file that looks like JavaScript objects:

```
{
  "levelNumber": 1,
  "grid": [
    [1,1,1,1],
    [1,0,0,1],
    [1,0,2,1],
    [1,1,1,1]
  ]
}
```

✔ Why your instructors love JSON:

Your slides say:

- “Data lives in a file, not hard-coded in sketch.js.”
- “You can add levels without touching the code”.

🎮 Example:

Instead of writing five maze layouts in code, you save **five JSON files**.

Then your program does:

loadJSON("level1.json")

loadJSON("level2.json")

This means:

- Your file holds **content**
- Your JavaScript holds **logic**

This is a **core industry pattern**.

🟩 2. What Are Classes? (In the way GBDA 302 uses them)

✔ A class is a blueprint for creating objects.

Think of it like cookie cutters:

- **Class = cookie cutter**
- **Object = actual cookie**

✔ Classes in your course bundle:

- **Data** (like the grid, player position)
- **Methods** (functions, like draw(), update(), move())

From the slides:
“Bundle data + behaviour into one object.”

Example: A Level class.

```
class Level {
  constructor(grid) {
    this.grid = grid;
  }

  draw() {
    // code that draws the level
  }
}
```

Now you can create multiple different levels with the same blueprint:

```
let level1 = new Level(level1Grid);
let level2 = new Level(level2Grid);
```

🟧 Why Use Both? (The exact “problem” your slides explain)

Your slides list 3 issues with beginner code:

❌ **Spaghetti code** — everything inside `draw()`
❌ **Repeated logic** — copy/pasted code for each level
❌ **Hard to reuse** — want 5 levels? copy/paste 5x?

JSON + classes solve these:

➢ JSON makes levels reusable

You change the file, **not the code**.

➢ Classes make logic reusable

`Level.draw()` works for ANY level you pass into it.

🟪 How JSON + Classes Work Together (GBDA 302 version)

The process your slides show:

1. **Load JSON** → gives you the raw data
2. **Pass JSON data into a class** → class organizes it
3. **Use the class to draw / update the game**

Example:

Step 1: Load JSON

```
let levelData;



function preload() {

  levelData = loadJSON("level1.json");

}
```

Step 2: Build a class using JSON

```
let currentLevel;
```

```
function setup() {

  currentLevel = new Level(levelData.grid);

}
```

Step 3: Use the class

```
function draw() {

  currentLevel.draw();

}
```

This is exactly the flow on *pages 9–11* of your slides.

🟩 Putting It All Together (Your Course Flow)

Your Week 4 examples show this progression:

Example	Concept	Meaning
1	Static grid	Hard-coded layout
2	Level class	Logic bundled in a class
3	JSON + Level	Data stored externally
4	Level + Player classes	Interactive game

5	Platformer	Multiple classes + JSON

By Step 5, you’re doing what real game engines do:

- Level files
- Player classes
- Physics
- Collision
- Themes

- Worlds

🎉 Quick Analogies (if you’re a visual learner)

JSON is like:

🗂️ A binder full of world information (levels, enemies, settings)

Classes are like:

🛠️ Tools that know **how** to use that information
(draw the world, move the player, detect collisions)

Your game engine (what you're building!) is the combination of the two.

📌 Want me to go deeper or show you code examples?

I can walk you through:

- A full Level class example
- How loadJSON actually works
- How to design clean classes
- How to structure multiple levels

- How to connect Start → Level → Win screens (like in your current project!)

Since you're working on a game with screens/buttons, I can also show you how JSON and classes plug into your *Start Screen → Choice1 Screen* project.

Just tell me what part you want next!

My message: I'm going to provide you with multiple files from the example code now. Store each file but don't do anything yet. I'll tell you once I'm done sending all the files

Copilot Response: Sounds good — go ahead and upload each file.

I'll store them as you send them and won't take any action until you say you're finished.

My message:

/* BlobPlayer.js (Example 5)
BlobPlayer owns all "dynamic" player state:
- position (x,y), radius (r)
- velocity (vx,vy)
- movement tuning (accel, friction, max run)
- jump state (onGround)
- blob rendering animation parameters (noise wobble)

It also implements:
- update() for physics + collision against platforms
- jump() for input
- draw() for the "breathing blob" look

The algorithm is the same as the original blob world example from Week 2:
- Apply input acceleration
- Apply friction
- Apply gravity
- Compute an AABB (box) around the blob
- Move box in X and resolve collisions
- Move box in Y and resolve collisions
- Write back box center to blob position */

class BlobPlayer { constructor() { // ----- Transform ----- this.x = 0; this.y = 0; this.r = 26;
// ----- Velocity -----
this.vx = 0;
this.vy = 0;

```
// ----- Movement tuning (matches your original values) -----
this.accel = 0.55;
this.maxRun = 4.0;

// Physics values that are typically overridden per level.
this.gravity = 0.65;
this.jumpV = -11.0;

// State used by jumping + friction choice.
this.onGround = false;

// Friction:
// - in air: almost no friction (keeps momentum)
// - on ground: more friction (stops more quickly)
this.frictionAir = 0.995;
this.frictionGround = 0.88;

// ----- Blob rendering / animation -----
this.t = 0;
this.tSpeed = 0.01;
this.wobble = 7;
this.points = 48;
this.wobbleFreq = 0.9;

}
/* Apply level settings + spawn the player. We reset velocities so each level starts consistently. */
spawnFromLevel(level) { this.gravity = level.gravity; this.jumpV = level.jumpV;
this.x = level.start.x;
this.y = level.start.y;
this.r = level.start.r;

this.vx = 0;
this.vy = 0;
this.onGround = false;

}
/* Update movement + resolve collisions against all platforms.
Input is polled with keyIsDown to get smooth movement (held keys). This keeps the behavior
aligned with your original blob example. */ update(platforms) { // 1) Horizontal input (A/D or
arrows) let move = 0; if (keyIsDown(65) || keyIsDown(LEFTARROW)) move -= 1; if
(keyIsDown(68) || keyIsDown(RIGHTARROW)) move += 1;
// 2) Apply horizontal acceleration based on input
this.vx += this.accel * move;
```

```
// 3) Apply friction (ground vs air)
this.vx *= this.onGround ? this.frictionGround : this.frictionAir;

// 4) Clamp max run speed
this.vx = constrain(this.vx, -this.maxRun, this.maxRun);

// 5) Apply gravity every frame
this.vy += this.gravity;

// 6) Build an AABB around the blob (center/radius -> box)
let box = {
  x: this.x - this.r,
  y: this.y - this.r,
  w: this.r * 2,
  h: this.r * 2,
};

// 7) Move in X and resolve collisions
box.x += this.vx;

for (const s of platforms) {
  if (overlapAABB(box, s)) {
    // If moving right, snap to the left side of the platform.
    if (this.vx > 0) box.x = s.x - box.w;
    // If moving left, snap to the right side of the platform.
    else if (this.vx < 0) box.x = s.x + s.w;

    // Cancel horizontal velocity after collision.
    this.vx = 0;
  }
}

// 8) Move in Y and resolve collisions
box.y += this.vy;

// Reset and recompute onGround each frame during Y resolution.
this.onGround = false;

for (const s of platforms) {
  if (overlapAABB(box, s)) {
    if (this.vy > 0) {
      // Falling: snap to platform top
      box.y = s.y - box.h;
      this.vy = 0;
      this.onGround = true;
    } else if (this.vy < 0) {
      // Rising: snap to platform bottom (head bump)
      box.y = s.y + s.h;
      this.vy = 0;
    }
  }
}

// 9) Write back blob center from box position
this.x = box.x + box.w / 2;
this.y = box.y + box.h / 2;

// 10) Optional: keep player within canvas horizontally.
this.x = constrain(this.x, this.r, width - this.r);

// 11) Advance blob animation time
this.t += this.tSpeed;

}
//Jump: only possible when on ground. jump() { if (!this.onGround) return; this.vy = this.jumpV;
this.onGround = false; }
/* Draw the blob with a wobbly outline:
    ●   we sample a noise value around the circle
    ●   perturb the radius slightly per vertex
    ●   this creates an organic "breathing"”" look

         This is the same technique as the original drawBlob() function. */ draw(colourHex) {
        fill(color(colourHex)); beginShape();

         for (let i = 0; i < this.points; i++) { const a = (i / this.points) * TAU;

         // Noise input: circle coordinates + time. const n = noise( cos(a) * this.wobbleFreq + 100,
        sin(a) * this.wobbleFreq + 100, this.t, );

         // Map noise to a small radius offset. const rr = this.r + map(n, 0, 1, -this.wobble,
        this.wobble);

         // Place the vertex around the center. vertex(this.x + cos(a) * rr, this.y + sin(a) * rr); }

         endShape(CLOSE); } }
/* Collision function: AABB overlap test.
    ●   a is the moving player "box"
    ●   b is a platform rectangle
```

We accept b as either:

- a Platform instance (with x,y,w,h)
- or a plain object with x,y,w,h This keeps it flexible. */ function overlapAABB(a, b) { return (a.x < b.x + b.w && a.x + a.w > b.x && a.y < b.y + b.h && a.y + a.h > b.y); }

Copilot Response: Got it — I've stored this file (`BlobPlayer.js`).

Go ahead and send the next ones, and I'll keep storing them without taking any action until you say you're finished.

My message: <!-- Load classes BEFORE sketch.js (order matters) -->

```
<script src="platform.js"></script>
<script src="worldLevel.js"></script>
<script src="blobPlayer.js"></script>

<!-- Main sketch -->
<script src="sketch.js"></script>
```

Copilot Response: Stored!

I've saved this second file (your `index.html`) as well.

You can continue sending the remaining files — I'll keep storing them and won't take any action until you say **"I'm done"**.

My message: { "include": ["*.js", "libraries/*.js", "c:\Users\david\.vscode\extensions\wmcicompsci.cs30-p5-1.9.3\p5types\global.d.ts"] }

Copilot Response: Stored!

This file (your config/JSON include file) is saved as well.

Feel free to continue sending the rest — I'm still just storing and not taking any action until you say **you're done**.

My message:

```
{
  "schemaVersion": 1,
  "levels": [
    {
      "name": "Intro Steps",
      "gravity": 0.65,
      "jumpV": -11.0,
      "theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
      "start": { "x": 80, "y": 220, "r": 26 },
      "platforms": [
        { "x": 0, "y": 324, "w": 640, "h": 36 },
        { "x": 120, "y": 254, "w": 120, "h": 12 },
        { "x": 300, "y": 204, "w": 90, "h": 12 },
        { "x": 440, "y": 144, "w": 130, "h": 12 },
        { "x": 520, "y": 254, "w": 90, "h": 12 }
      ]
    }
  ]
}
```

Stored!

I’ve saved this file as well — this is your **levels.json** (World configuration file).

Still holding everything and doing nothing with it until you say **“I’m done sending files.”**

/* Platform.js (Example 5)
A Platform is a single axis-aligned rectangle in the world.
Why a class for something "simple"?

- It standardizes the shape of platform data.
- It makes later upgrades easy (e.g., moving platforms, icy platforms, spikes).
- It keeps drawing code in the object that knows what it is.

In JSON, platforms are stored like: { "x": 0, "y": 324, "w": 640, "h": 36 } */
class Platform { constructor({ x, y, w, h }) { // Position is the top-left corner. this.x = x; this.y = y;
// Size (width/height).
this.w = w;
this.h = h;

}
draw(fillColor) { fill(fillColor); rect(this.x, this.y, this.w, this.h); } }

Copilot Response: Stored!

I’ve saved this file (`Platform.js`) as well.

You can keep sending the rest — I’ll just store them and won’t take any action until you say **you’re done**.

My message:/* Week 4 — Example 5: Example 5: Blob Platformer (JSON + Classes) Course: GBDA302 Instructors: Dr. Karen Cochrane and David Han Date: Feb. 5, 2026

This file orchestrates everything:

- load JSON in preload()
- create WorldLevel from JSON
- create BlobPlayer
- update + draw each frame

- handle input events (jump, optional next level)

This matches the structure of the original blob sketch from Week 2 but moves details into classes. */

let data; // raw JSON data let levelIndex = 0;

let world; // WorldLevel instance (current level) let player; // BlobPlayer instance

function preload() { // Load the level data from disk before setup runs. data = loadJSON("levels.json"); }

function setup() { // Create the player once (it will be respawned per level). player = new BlobPlayer();

// Load the first level. loadLevel(0);

// Simple shared style setup. noStroke(); textFont("sans-serif"); textSize(14); }

function draw() { // 1) Draw the world (background + platforms) world.drawWorld();

// 2) Update and draw the player on top of the world player.update(world.platforms); player.draw(world.theme.blob);

// 3) HUD fill(0); text(world.name, 10, 18); text("Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N", 10, 36); }

function keyPressed() { // Jump keys if (key === " " || key === "W" || key === "w" || keyCode === UP_ARROW) { player.jump(); }

// Optional: cycle levels with N (as with the earlier examples) if (key === "n" || key === "N") { const next = (levelIndex + 1) % data.levels.length; loadLevel(next); } }

/* Load a level by index:

- create a WorldLevel instance from JSON
- resize canvas based on inferred geometry
- spawn player using level start + physics */ function loadLevel(i) { levelIndex = i;

  // Create the world object from the JSON level object. world = new WorldLevel(data.levels[levelIndex]);

  // Fit canvas to world geometry (or defaults if needed). const W = world.inferWidth(640); const H = world.inferHeight(360); resizeCanvas(W, H);

```
// Apply level settings + respawn. player.spawnFromLevel(world); }
```

Copilot Response: Stored!
I’ve saved this file as `sketch.js`.

You can keep sending the remaining files — I’ll continue storing them and won’t take any action until you say **you’re done**.

My Message:
```
html, body { margin: 0; padding: 0; }

canvas { display: block; }
```

Copilot Response: Stored!
Your `style.css` file is saved as well.

If you have more files, go ahead and send them. I’m still just storing everything until you say **“I’m done.”**

My Message: /* WorldLevel.js (Example 5)
WorldLevel wraps ONE level object from levels.json and provides:
- Theme colours (background/platform/blob)
- Physics parameters that influence the player (gravity, jump velocity)
- Spawn position for the player (start)
- An array of Platform instances
- A couple of helpers to size the canvas to fit the geometry

This is directly inspired by your original blob sketch’s responsibilities:
- parse JSON
- map platforms array
- apply theme + physics
- infer canvas size

Expected JSON shape for each level (from your provided file): { "name": "Intro Steps", "gravity": 0.65, "jumpV": -11.0, "theme": { "bg":"…", "platform":"…", "blob":"…" }, "start": { "x":80, "y":220, "r":26 }, "platforms": [{x,y,w,h}, …] } */

```
class WorldLevel { constructor(levelJson) { // A readable label for HUD. this.name =
levelJson.name || "Level";
// Theme defaults + override with JSON.
this.theme = Object.assign(
  { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },
  levelJson.theme || {},
);

// Physics knobs (the blob player will read these).
this.gravity = levelJson.gravity ?? 0.65;
this.jumpV = levelJson.jumpV ?? -11.0;

// Player spawn data.
// Use optional chaining so levels can omit fields safely.
this.start = {
  x: levelJson.start?.x ?? 80,
  y: levelJson.start?.y ?? 180,
  r: levelJson.start?.r ?? 26,
};

// Convert raw platform objects into Platform instances.
this.platforms = (levelJson.platforms || []).map((p) => new Platform(p));

}
/* If you want the canvas to fit the world, you can infer width/height by finding the maximum x+w
and y+h across all platforms. */ inferWidth(defaultW = 640) { if (!this.platforms.length) return
defaultW; return max(this.platforms.map((p) => p.x + p.w)); }
inferHeight(defaultH = 360) { if (!this.platforms.length) return defaultH; return
max(this.platforms.map((p) => p.y + p.h)); }
/* Draw only the world (background + platforms). The player draws itself separately, after the
world is drawn. */ drawWorld() { background(color(this.theme.bg)); for (const p of this.platforms)
{ p.draw(color(this.theme.platform)); } } }
```

Copilot Response: Stored!

I've saved this file (`WorldLevel.js`) as well.

Keep sending any remaining files. I'll continue to just store them and won't take any action until you say **"I'm done."**

My Message: The above files I sent make up the Example 05 code. I need to generate a level using arrays or JSON data and I need to use loops to dynamically place tiles, obstacles, or platforms.How would I go about doing this?

Copilot Response: Awesome — you've already got a clean Example 05 foundation (WorldLevel + Platform + BlobPlayer). To satisfy the Side Quest requirement:

> **"Generate a level using arrays or JSON data; use loops to dynamically place tiles, obstacles, or platforms."**

you'll do two things:

1. **Represent the layout as data** (array or JSON).
2. **Loop over that data to create `Platform` objects** (and optionally other things like spikes or coins).

Below are two drop-in patterns you can pick from. They both plug directly into your current files (no rewrites needed), and they follow the exact JSON/classes pattern from the Week 4 slides. [Week 4 - part 2 | PDF]

Option A — Use a 2D array (quickest)

Idea: A grid of numbers (tile map). Each number means “what to place here.” The loop reads each cell and instantiates objects at the correct pixel location.

1) Create a new level entry that uses a grid

You can keep your existing `levels.json` and **add** a second level that includes a `tileSize` and a `grid`. You’ll still keep `platforms` if you want a floor; or you can generate everything from the grid.

levels.json (append another level):

```
{
 "schemaVersion": 1,
 "levels": [
	{
	"name": "Intro Steps",
	"gravity": 0.65,
	"jumpV": -11.0,
	"theme": { "bg": "#F0F0F0", "platform": "#C8C8C8", "blob": "#1478FF" },
	"start": { "x": 80, "y": 220, "r": 26 },
	"platforms": [
	{ "x": 0, "y": 324, "w": 640, "h": 36 },
	{ "x": 120, "y": 254, "w": 120, "h": 12 },
{ "x": 300, "y": 204, "w": 90, "h": 12 },
{ "x": 440, "y": 144, "w": 130, "h": 12 },
{ "x": 520, "y": 254, "w": 90, "h": 12 }
]
},

{
"name": "Grid Level (Array)",
"gravity": 0.65,
"jumpV": -11.0,
"theme": { "bg": "#F7F3FF", "platform": "#7A61FF", "blob": "#222" },
"start": { "x": 60, "y": 60, "r": 24 },

"tileSize": 40,
"grid": [
[1,1,1,1,1,1,1,1,1,1,1,1,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,0,2,0,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
[1,0,0,1,1,1,0,0,0,0,0,0,1],
[1,0,0,0,0,0,0,0,0,0,3,0,1],
[1,0,0,0,0,0,0,0,0,0,0,0,1],
        [1,1,1,1,1,1,1,1,1,1,1,1]
        ]
        }
  ]
}
```

Legend (your choice, just be consistent):

- `0` = empty
- `1` = platform tile
- `2` = spike (obstacle)
- `3` = small platform tile (or coin, etc.)

2) Teach `WorldLevel` to build platforms from the grid using loops

Add the **optional** grid support in your `WorldLevel.js`. You'll loop rows/cols and create `Platform` rectangles wherever the grid says so.

WorldLevel.js (augment constructor):

```
class WorldLevel {
  constructor(levelJson) {
        this.name = levelJson.name || "Level";


        this.theme = Object.assign(
        { bg: "#F0F0F0", platform: "#C8C8C8", blob: "#1478FF" },
        levelJson.theme || {}
        );


        this.gravity = levelJson.gravity ?? 0.65;
this.jumpV = levelJson.jumpV ?? -11.0;


this.start = {

x: levelJson.start?.x ?? 80,

y: levelJson.start?.y ?? 180,

r: levelJson.start?.r ?? 26,

};


// Base platforms from JSON (already in Example 05)

this.platforms = (levelJson.platforms || []).map(p => new Platform(p));


// --- NEW: grid-based generation (dynamic placement via loops) ---

if (levelJson.grid) {

const tileSize = levelJson.tileSize ?? 40;

const grid = levelJson.grid;


for (let row = 0; row < grid.length; row++) {

for (let col = 0; col < grid[row].length; col++) {

const cell = grid[row][col];

const x = col * tileSize;

const y = row * tileSize;
```

```
        if (cell === 1) {
        // Full tile as a platform
        this.platforms.push(new Platform({ x, y, w: tileSize, h: tileSize }));
        }
        else if (cell === 3) {
        // Example: “small platform” half height
        this.platforms.push(new Platform({ x, y: y + tileSize * 0.5, w: tileSize, h: tileSize * 0.5 }));
        }
        // If you want spikes or other obstacles, see Option C below
        }
        }
        }
  }

  inferWidth(defaultW = 640) {
        if (!this.platforms.length) return defaultW;
        return max(this.platforms.map(p => p.x + p.w));
  }

  inferHeight(defaultH = 360) {
        if (!this.platforms.length) return defaultH;
        return max(this.platforms.map(p => p.y + p.h));
  }


  drawWorld() {
        background(color(this.theme.bg));
        for (const p of this.platforms) {
        p.draw(color(this.theme.platform));
        }
  }
}
```

What you just did: you **used nested loops** (`for` row, `for` col) to **dynamically place platforms** based on the grid. This is exactly what your slides mean by “use loops to place tiles.” [Week 4 - part 2 | PDF]

Option B — Use JSON with a tile map (clean for content editing)

Prefer to keep all layout data in JSON (so you never touch code to make levels)? Put `tileSize` + `grid` in your JSON (as above) and **don’t** list `platforms`. The same `WorldLevel` constructor will generate platform rectangles entirely from the `grid`. That’s the “data lives in a file; code stays generic” approach emphasized in the lecture. [Week 4 - part 2 | PDF]

Option C — Add obstacles (e.g., spikes) generated via loops

Because your collision is AABB vs. rectangles (see `overlapAABB` in `BlobPlayer.js`), the simplest obstacle is just another rectangle type you don't want to land on. Two easy paths:

1. **Make spikes "harmful platforms"** (same geometry, different behavior).
2. **Give them their own class** (e.g., `Spike`) if you want different drawing.

Minimal approach: spikes as rectangles

- Extend `WorldLevel` to collect `this.spikes = []`.
- In the grid loop, if `cell === 2`, push a spike rect into `this.spikes`.

WorldLevel.js (snippet inside grid loop):

```
// Before loops

this.spikes = [];



…



if (cell === 2) {

  this.spikes.push({ x, y, w: tileSize, h: tileSize * 0.5 });

}
```

Drawing spikes (still in WorldLevel):

```
drawWorld() {

  background(color(this.theme.bg));

  // platforms

  for (const p of this.platforms) p.draw(color(this.theme.platform));



  // spikes (simple triangles on top of a thin rect)
  push();

  fill('#E74C3C');

  noStroke();

  for (const s of this.spikes) {

        rect(s.x, s.y + s.h - 6, s.w, 6); // base

        // draw 3 triangles across the tile

        const tW = s.w / 3;

        triangle(s.x, s.y + s.h - 6, s.x + tW/2, s.y, s.x + tW, s.y + s.h - 6);

        triangle(s.x + tW, s.y + s.h - 6, s.x + 1.5tW, s.y, s.x + 2tW, s.y + s.h - 6);

        triangle(s.x + 2tW, s.y + s.h - 6, s.x + 2.5tW, s.y, s.x + 3*tW, s.y + s.h - 6);

  }

  pop();

}
```

Collision / fail state (in `BlobPlayer.update` or in `draw()` after update):\ Do a separate AABB overlap check between the player’s box and each spike; on hit, respawn.

sketch.js (simple respawn after update):

```
function draw() {

  world.drawWorld();



  player.update(world.platforms);

  player.draw(world.theme.blob);



  // Spike collision check (simple)
```

```
 if (world.spikes) {
	const box = { x: player.x - player.r, y: player.y - player.r, w: player.r2, h: player.r2 };
	for (const s of world.spikes) {
	if (overlapAABB(box, s)) {
	// Respawn player at start
	player.spawnFromLevel(world);
	break;
	}
	}
 }
 fill(0);
 text(world.name, 10, 18);
 text("Move: A/D or ←/→ • Jump: Space/W/↑ • Next: N", 10, 36);
}
```

> This is still “dynamic placement with loops” because spikes are **generated** by iterating the grid and placing geometry at computed positions.

Tips & gotchas (so it “just works”)

- **Tile/world sizing:** Your `inferWidth/Height` already looks at `platforms`. If you generate everything via grid, the inferred size will match the furthest platform edge—nice!
- **Start position:** If you also want start to be defined by the grid, pick a cell code (e.g., `9`), then in the grid loop set `this.start` when you encounter `9`, and render that cell as empty.
- **Performance:** For small student projects, loops over a 2D grid are totally fine.

- **Separation of concerns:** Keep *drawing logic* in classes (`Platform.draw`, spike drawing in `WorldLevel.drawWorld`) and *physics/collision* in `BlobPlayer.update`—this mirrors the slides' "Single Responsibility" principle. [Week 4 - part 2 | PDF]

How this meets the Side Quest rubric

- **"Generate a level using arrays or JSON data"**\ → You defined a `grid` in JSON (or an in-code array).
- **"Use loops to dynamically place tiles, obstacles, or platforms"**\ → Your nested `for` loops translate each grid cell into placed rectangles (platforms/spikes).
- **Bonus ("Add a second level that loads automatically when the first is finished")**\ → You already have `N` for next level. To auto-advance, detect a win condition (e.g., reaching a `goal` tile code like `4`), then call `loadLevel((levelIndex + 1) % data.levels.length)`.

Want me to tailor it to your exact aesthetic?

If you tell me:

- tile size you want (e.g., 32 or 48),
- what numbers you want to map (e.g., `1=platform`, `2=spike`, `3=goal`),
- whether you prefer JSON-only or quick in-code array,

I'll generate the ready-to-paste JSON and the minimal code diffs for your files.

My Message: I want to use the JSON approach. Create a new level that is more difficult than the original level. The platforms should be a bit less wide and a bit more spaced apart to make it harder (but the blob should still be able to jump from platform to platform). There should be 5 platforms

Copilot Response: Love it — you can add a second, tougher level **entirely in JSON** and your existing code will pick it up automatically (your `sketch.js` already cycles with **N** and calls `loadLevel(next)`).

Below is a drop-in level that's **harder than "Intro Steps"**:

- same physics (`gravity`, `jumpV`) so your blob's jump arc stays familiar
- **5 platforms total** (including the ground)
- upper platforms are **narrower** and **more spaced** (but still jumpable with your current `maxRun`, friction, and jump settings)
- a slight **staircase** pattern so difficulty ramps up from left → right

✅ With `jumpV = -11` and `gravity = 0.65`, the blob's airtime is ~34 frames; with `maxRun ≈ 4 px/frame`, horizontal gaps of **40–60 px** between platforms are safely jumpable.

➕ Add this to your `levels.json` (as a second entry in `"levels"`)

```
{
  "name": "Tighter Jumps",
  "gravity": 0.65,
  "jumpV": -11.0,
  "theme": { "bg": "#EEF6FF", "platform": "#6B7FD7", "blob": "#111" },

  // Start the blob standing on the ground, near the left.
  // Ground top is at y=324, blob radius is 26 → center y ≈ 324 - 26 = 298
  "start": { "x": 60, "y": 298, "r": 26 },

  "platforms": [
        // Ground (1)
        { "x": 0,   "y": 324, "w": 640, "h": 36 },

        // Narrow + spaced steps (2–5)
        // Gaps between right edge of previous and left edge of next: ~60, 60, 40 px (all jumpable)
        { "x": 140, "y": 272, "w": 100, "h": 12 },  // right edge 240 → next starts 300 (60 px gap)
        { "x": 300, "y": 232, "w": 80,  "h": 12 },  // right edge 380 → next starts 440 (60 px gap)
        { "x": 440, "y": 188, "w": 80,  "h": 12 },  // right edge 520 → next starts 560 (40 px gap)
        { "x": 560, "y": 140, "w": 70,  "h": 12 }   // final, a bit higher + narrower
  ]
}
```

Where to put it:\ Open `levels.json` and append that object as the **second element** of the `"levels"` array (after your existing “Intro Steps”). Make sure commas are correct between entries.

🔧 How to test it

1. Save `levels.json`.
2. Refresh your page / Live Server.
3. Press **N** to switch from “Intro Steps” to **“Tighter Jumps.”**
4. Try the jumps. If anything feels *too* tight:
    - reduce a gap slightly (e.g., move a platform 10 px left),
    - or widen one platform (`w += 10–20`).

🧩 Want obstacles or a goal later?

Because you’re using JSON + classes, you can extend this pattern easily:

- Add a `grid` and use loops to generate extra tiles (e.g., spikes on `2`),
- Or add a `"goal": { "x": ..., "y": ..., "w": ..., "h": ... }` and check overlap to auto-advance to the next level.

If you want, tell me the exact vibe (difficulty target, color palette, or whether you want a “goal” tile), and I’ll generate the JSON + tiny code diff to auto-advance when the blob reaches it.