SPONGE BRO'S ADVENTURE

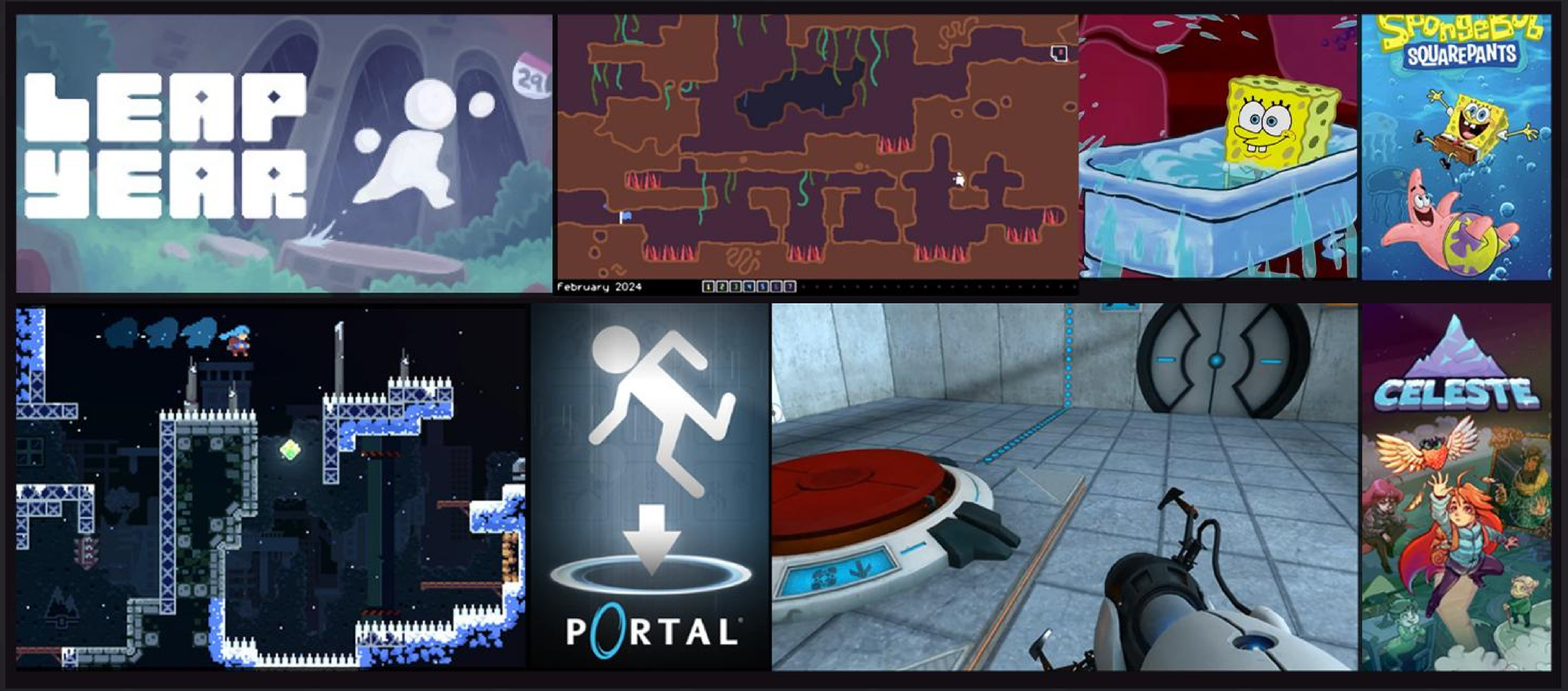
VIDEO LINK: https://youtu.be/YeX-gLpjrfw

In a dark, humid dungeon, a little sponge seeks a way out using the power of water. Absorb droplets to reload your ammo, then shoot to trigger mechanisms or boost your jumps with recoil. Solve puzzles with buttons and reflectors, master physics-based movement, and find freedom deep within the dungeon.

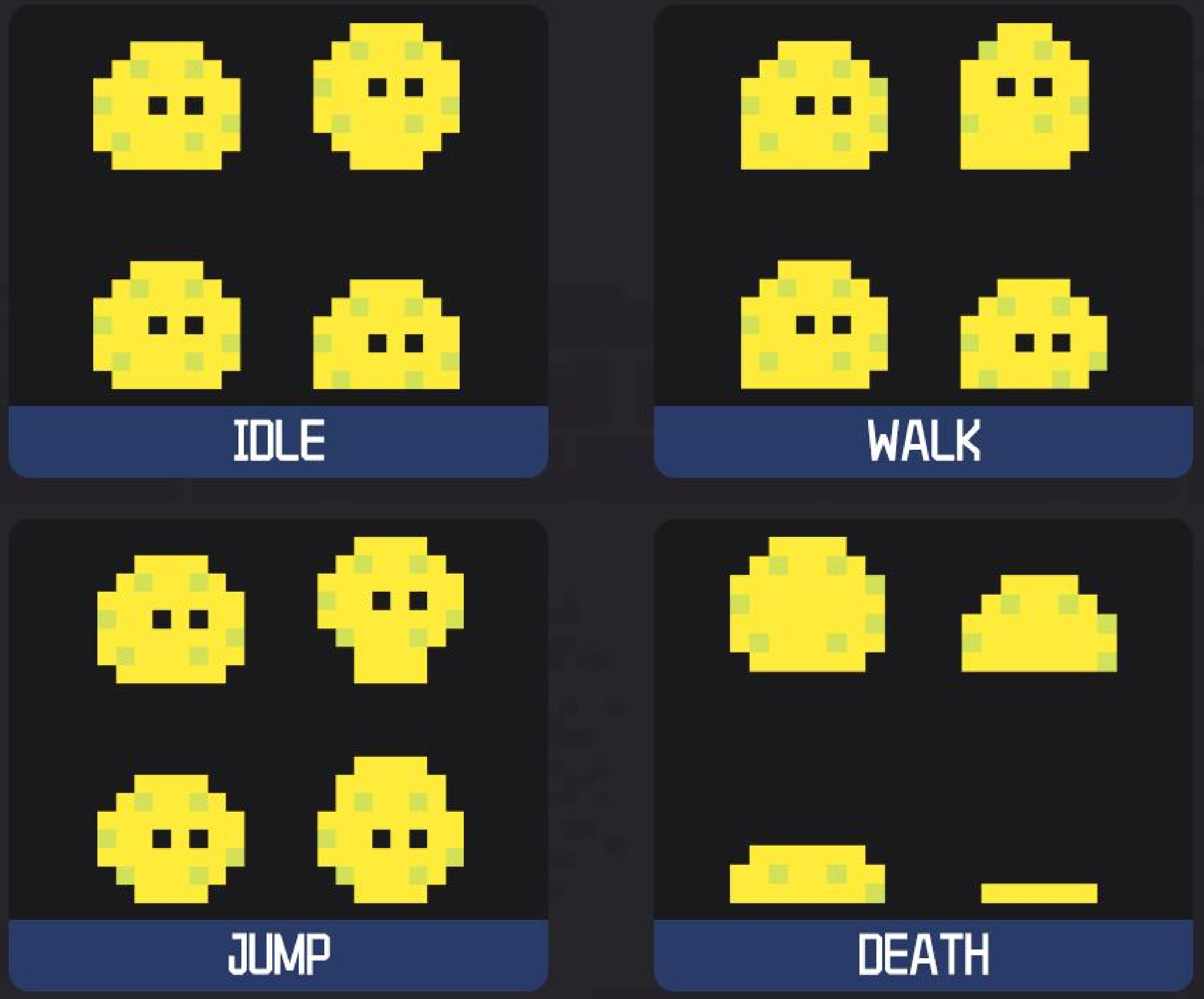
INSPIRATION & CHARACTER DESIGN

INSPIRATION

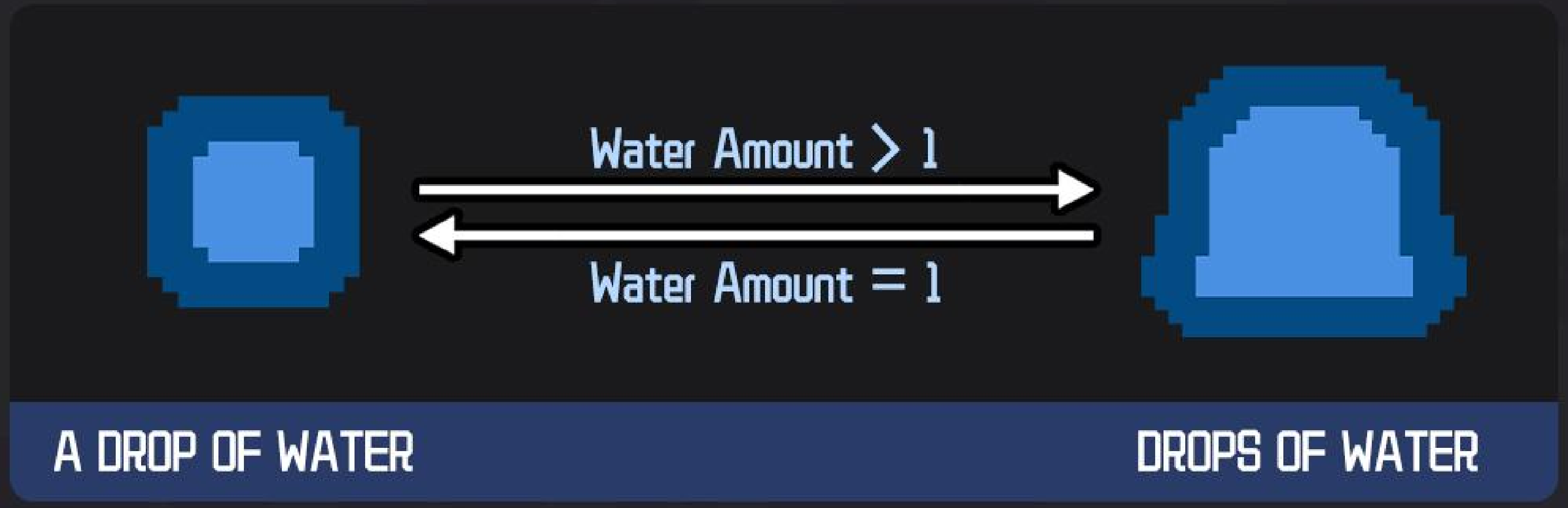
- The idea for SpongeBro's Adventure began when I watched a gameplay video of Leap Year. I was deeply impressed by how it combined platforming and puzzle-solving in such a simple yet elegant way. That experience sparked my interest in designing levels that blend jumping mechanics with thoughtful challenges.
- Since I wasn't very familiar with platformers or puzzle games before, I decided to study two classics — Celeste and Portal. From Celeste, I learned how carefully crafted movement and jump physics can make every action feel satisfying. From Portal, I was fascinated by how the game builds puzzles around logical "knowledge locks," especially the clever use of momentum conservation.
- These inspirations helped me shape my own design philosophy: to create a hero whose core ability — a water-powered jump — interacts naturally with the environment. And the sponge character was inspired by SpongeBob, whose absorb-and-spray nature fit perfectly with the game's mechanics.



CHARACTER ANIMATION



WATER DROPS



MECHANINCS

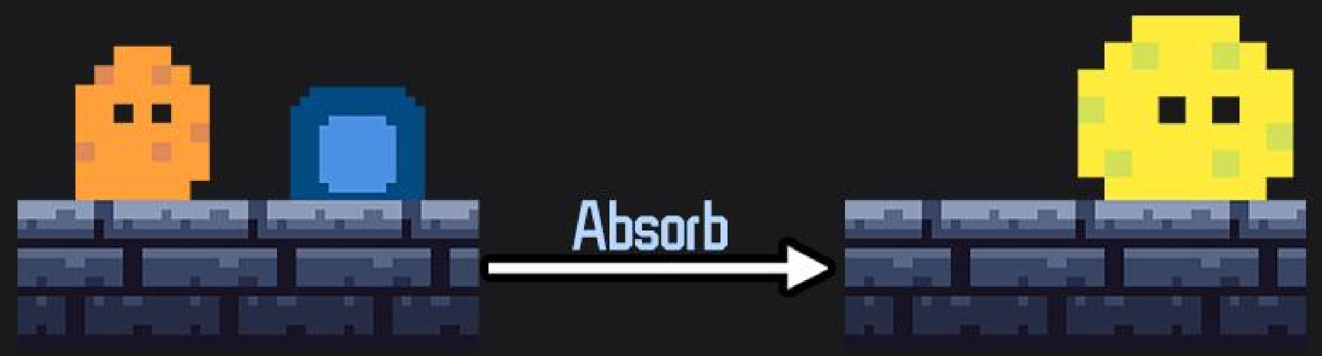
MECHANICS OVERVIEW

- In SpongeBro' s Adventure, the core mechanic centers on absorption and propulsion. The sponge protagonist collects scattered water droplets throughout the dungeon to refill its internal water reserve. Each time it absorbs water, the sponge swells in size, affecting its movement and jump dynamics. Players can release stored water to activate mechanisms, interact with the environment, or boost jumps using recoil force. Meanwhile, water droplets in the environment merge upon collision, and their motion follows the law of momentum conservation, adding a layer of realistic physics and dynamic puzzle-solving. Balancing absorption, release, and environmental interaction is the key to mastering this physics-driven dungeon adventure.

CORE MACHANINCS

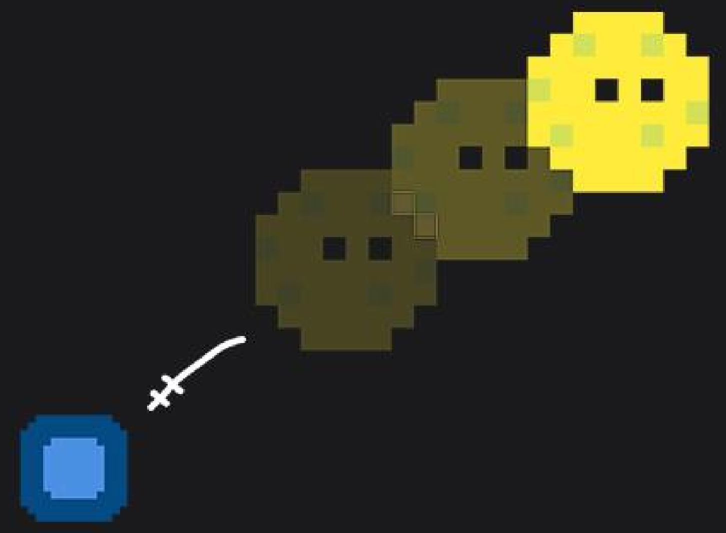
CORE MACHANINCS 1 : WATER COLLECTION

- SpongeBro grows as it absorbs water.
- And its color shifting only between dry and wet states.



CORE MACHANINCS 2 : WATER SHOOTING

- SpongeBro shoots water droplets to gain recoil.



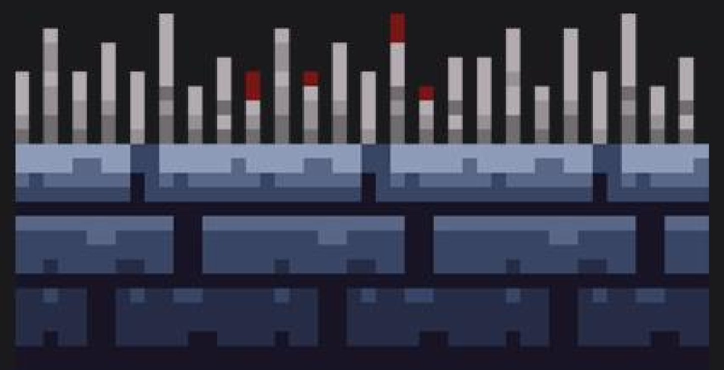
CORE MACHANINCS 3 : PHYSICS-BASED PUZZLE SOLVING

- MASS INTERACTION : When the player shoots a droplet, the sponge is pushed backward with an equal and opposite force, creating a balance between movement and action. This means every shot isn' t just an attack—it' s also a movement tool.
- MOMENTUM CONSERVATION : As droplets travel and collide, they merge into larger droplets, conserving momentum in the process. Their weight and size affect how they move, bounce, and interact with the world, leading to creative and sometimes unexpected outcomes.

ENVIORMENTAL INTERACTION

INTERACTIVE MECHANISMS 1 : SPIKE TRAP

- SpongeBro dies on contact with spikes and the level will restart automatically.

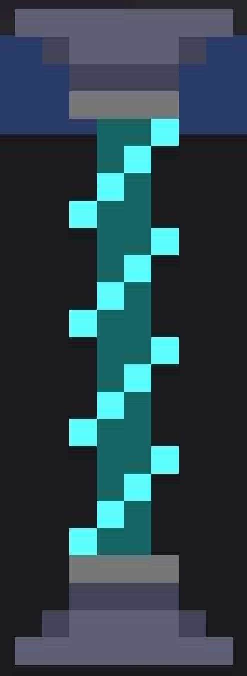


INTERACTIVE MECHANISMS 2 : BUTTON & DOOR

- When SpongeBro or a water droplet steps on a button, it changes the door' s open or closed state.

INTERACTIVE MECHANISMS 3 : REFLECTER

- When SpongeBro (in a soaked state) or a water droplet hits a reflector, it bounces off.

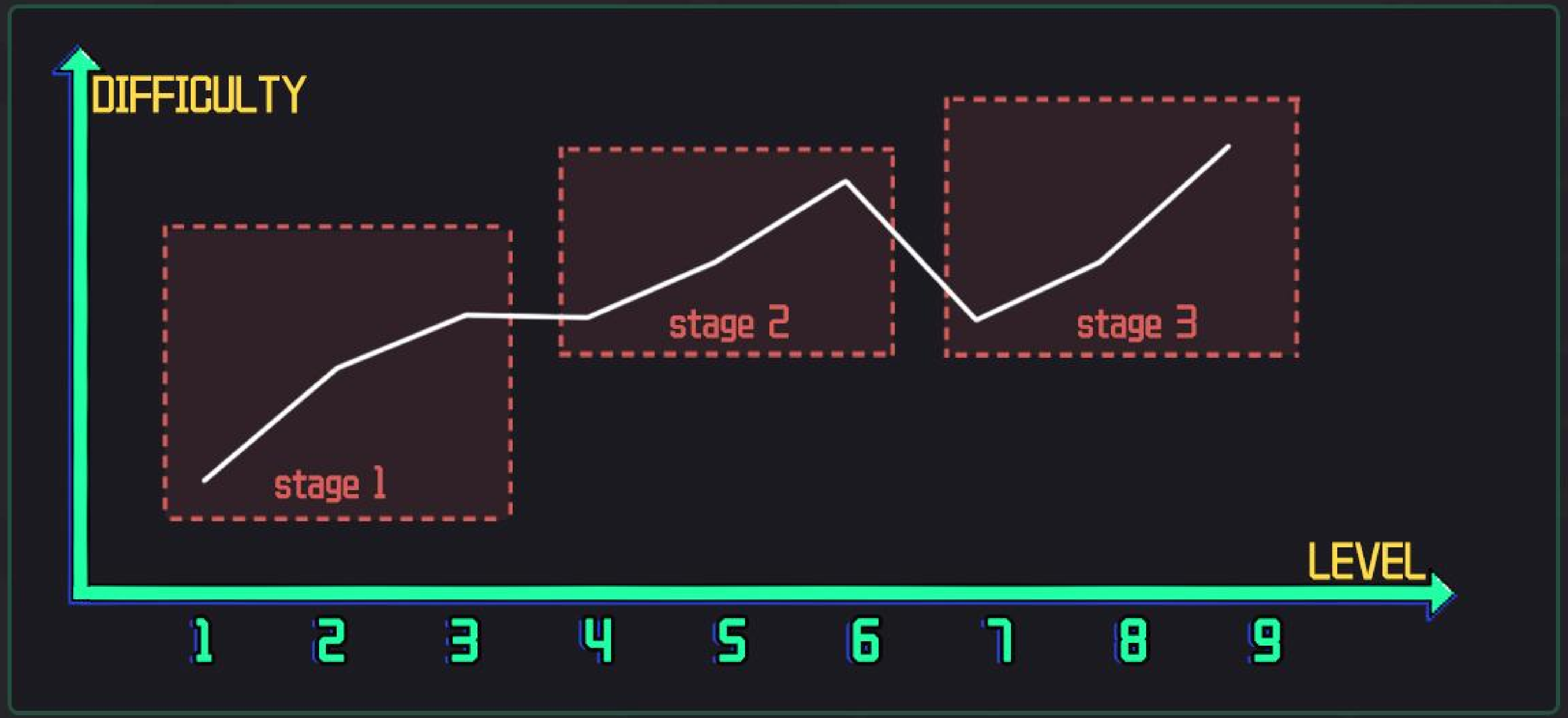


LEVEL DESIGN

LEVEL OVERVIEW

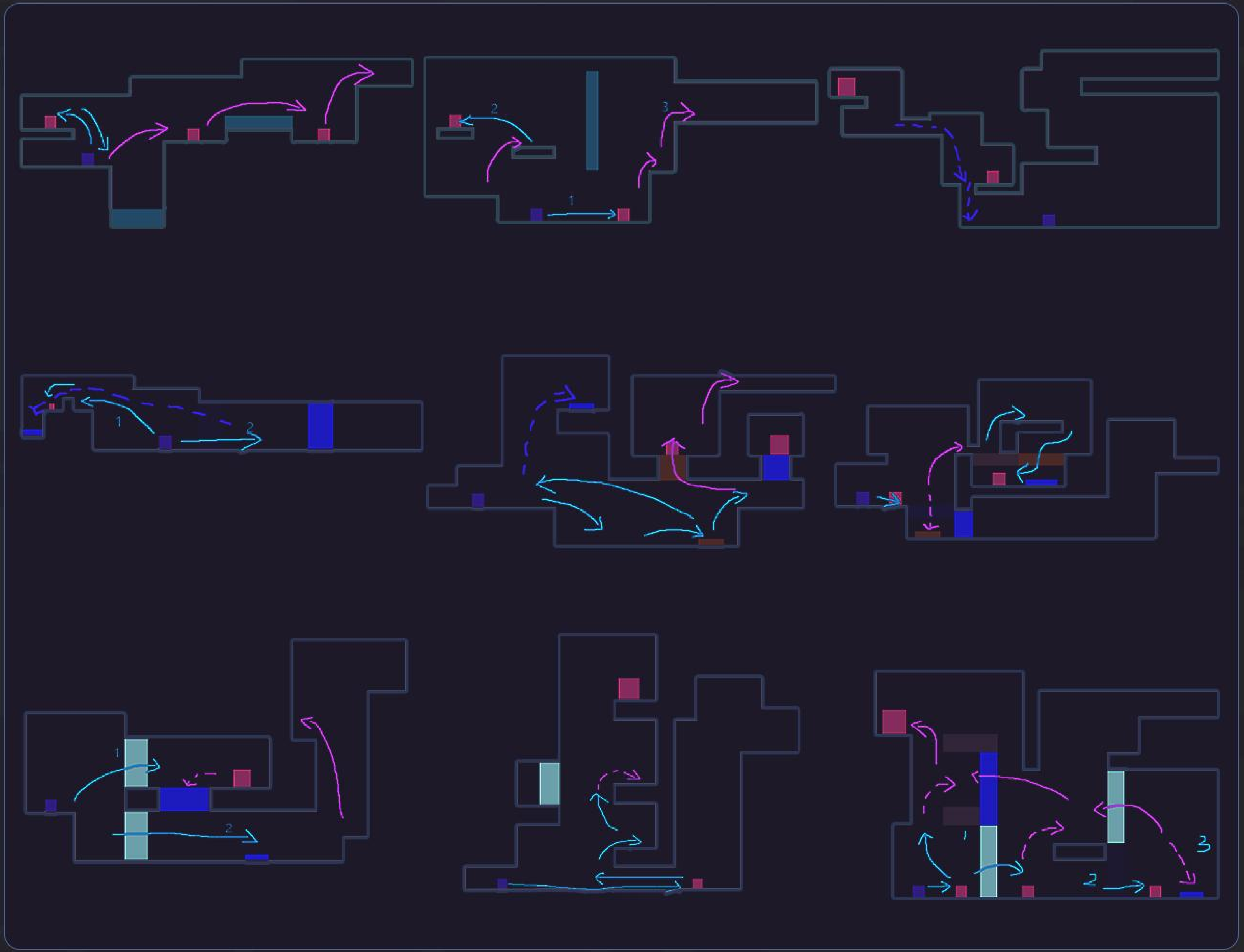
- The game is structured into three stages, each consisting of three levels that gradually expand the player's understanding of the core mechanics. Each stage in this game follows a LEARN-TEST-TWIST progression. Early levels introduce a new mechanic in a safe environment where the player can experiment freely. Subsequent stages then test the player's understanding through more complex applications, and finally, a twist level combines previously learned elements in unexpected ways to deepen mastery and create memorable challenges.

LEVEL DIFFICULTY CURVE

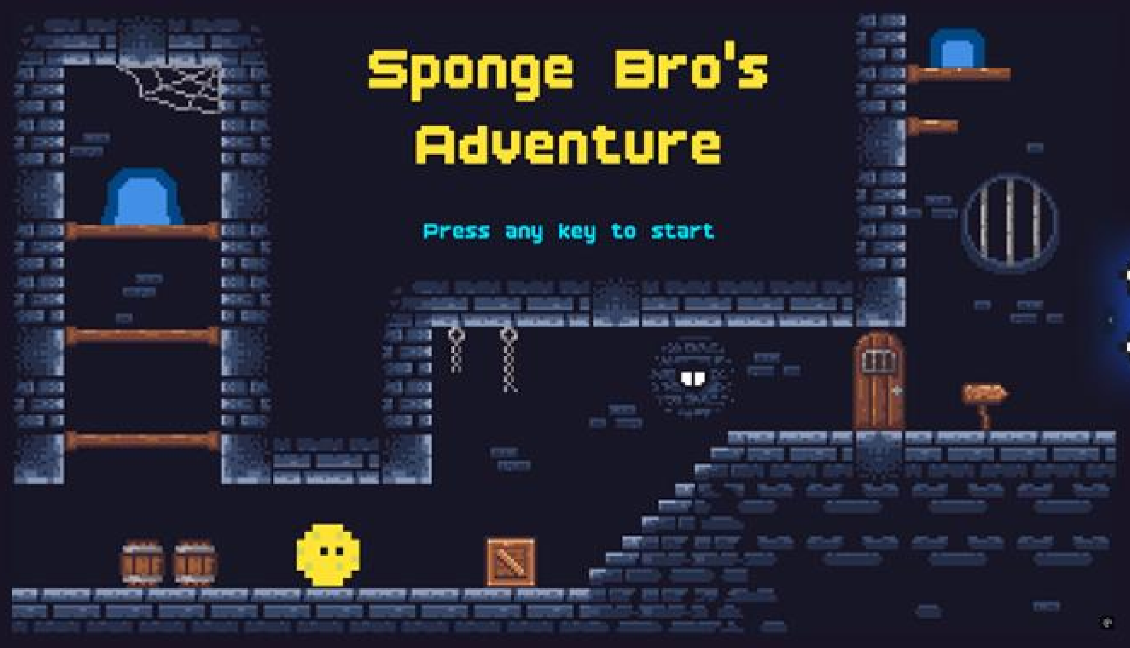


STAGE	LEVELS	TEACHING FOCUS	NEW MECHANIC	KNOWLEDGE LOCKS
I	1-3	Basic movement & slime mechanics	Water drops absorption & shooting	Jumping, shooting recoil control, size adaptation
II	4-6	Button & door puzzles	Single & multi-button doors, chain reactions	Timing, sequencing, planning
III	7-9	Reflection puzzles	Reflectors, combined mechanics	Momentum Reflection, advanced timing, path planning

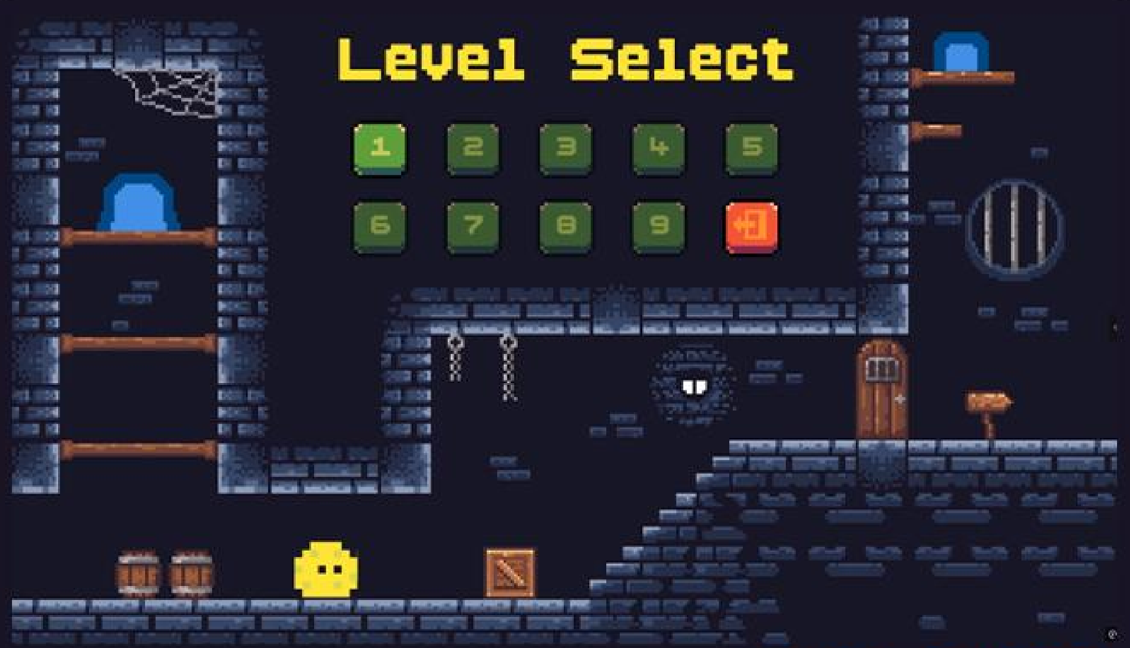
LEVEL BLOCKOUT



GAME INTERFACE & LEVEL DESIGN



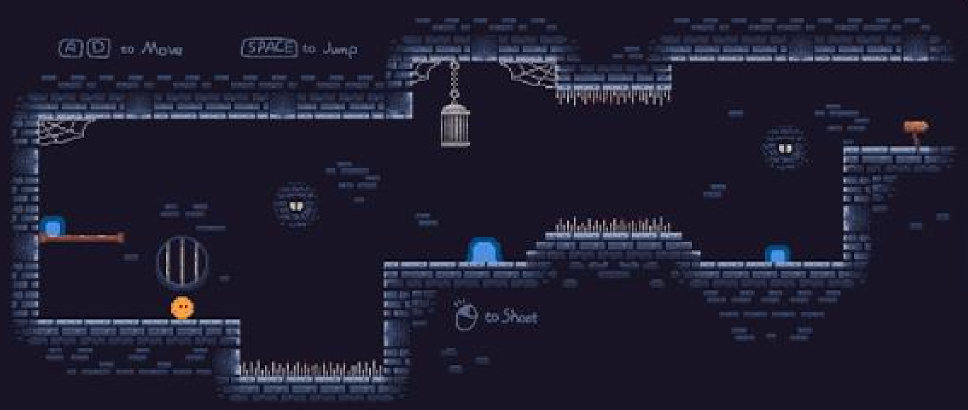
MAIN INTERFACE



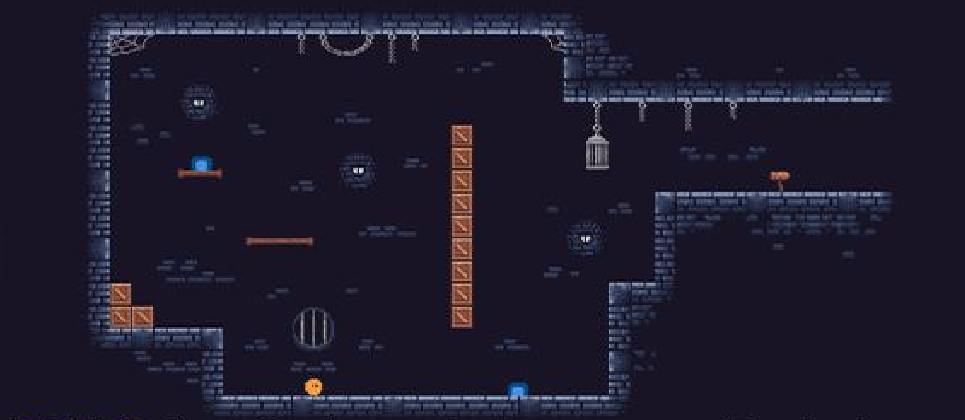
LEVEL SELECT

LEVEL SELECT SYSTEM

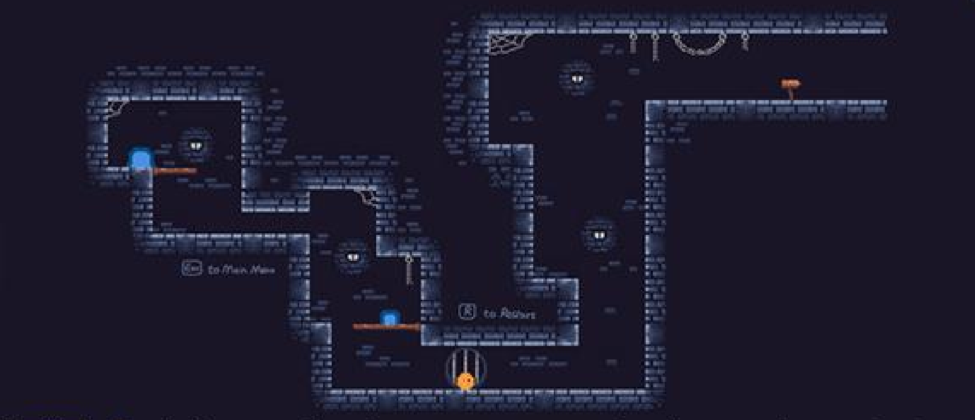
The level select system combines gameplay design with local save logic. Each button' s interactivity and transparency reflect the player' s unlocked progress, giving immediate visual feedback and guiding progression. Progress data is stored locally using Unity' s PlayerPrefs, while UnlockNextLevel() updates both the UI and saved state when a level is completed. This implementation ensures players can smoothly track and access unlocked levels, supporting both usability and game flow design.



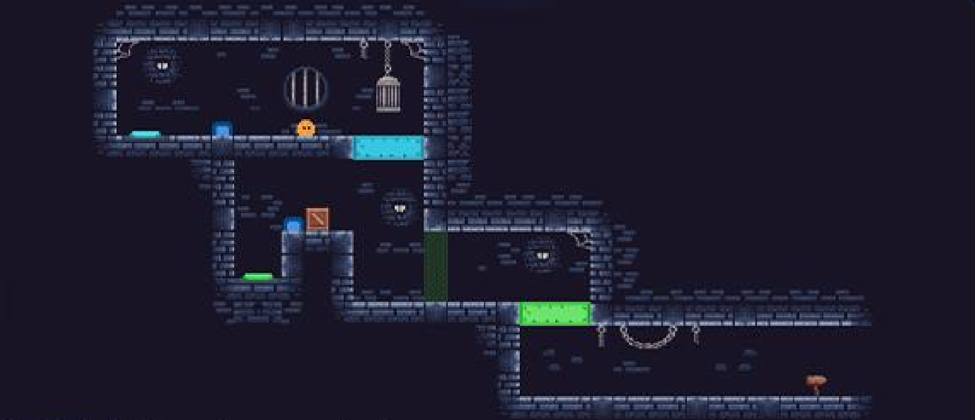
LEVEL 1 introduces the player to the recoil jump mechanic through environmental pressure. Two spike traps block the path forward, forcing the player to experiment with shooting to gain extra height.



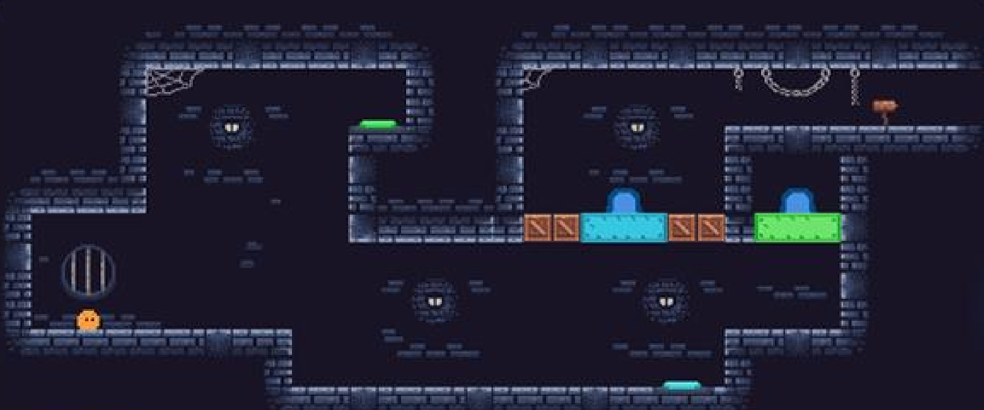
LEVEL 2 allows the player to further master recoil jumping by collecting two water drops to reach a high platform. The first drop also unlocks access to the second, encouraging strategic use of resources.



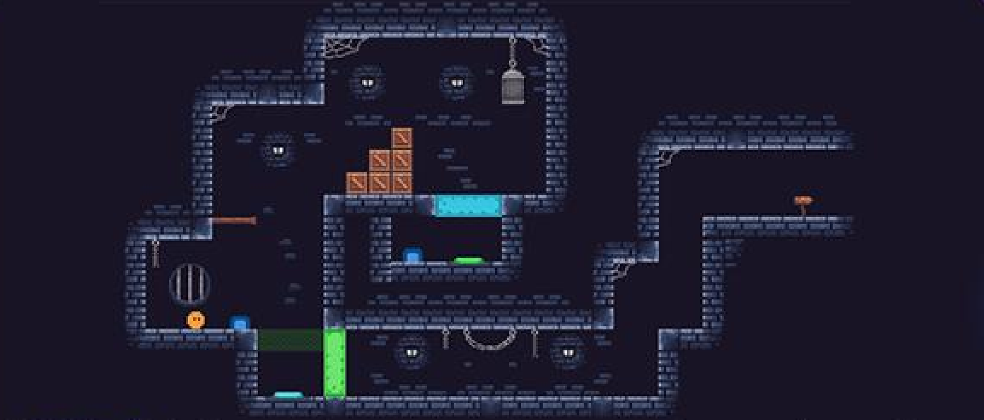
LEVEL 3 challenges the player to manage size changes by navigating differently sized openings. Mastery of volume-based mechanics is required, as adjusting body size through absorbing and shooting water drops is key to passing obstacles.



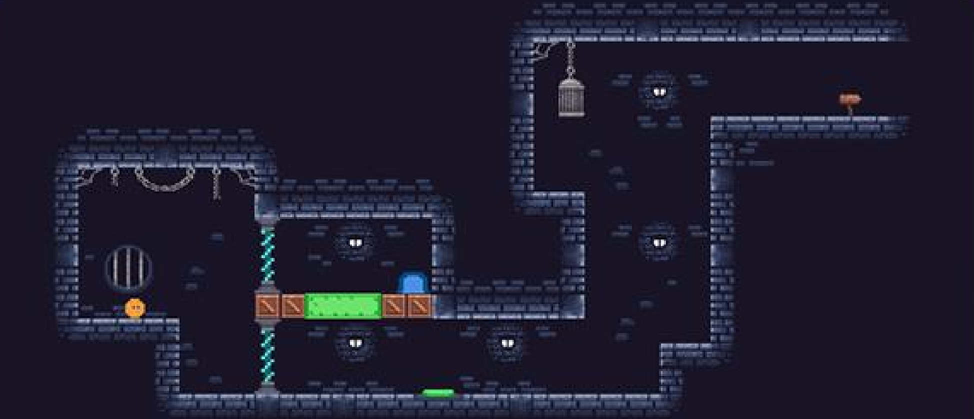
LEVEL 4 introduces switch-operated doors, requiring the player to shoot water drops or stand on buttons to open gates. The level layout encourages careful timing and positioning to successfully progress.



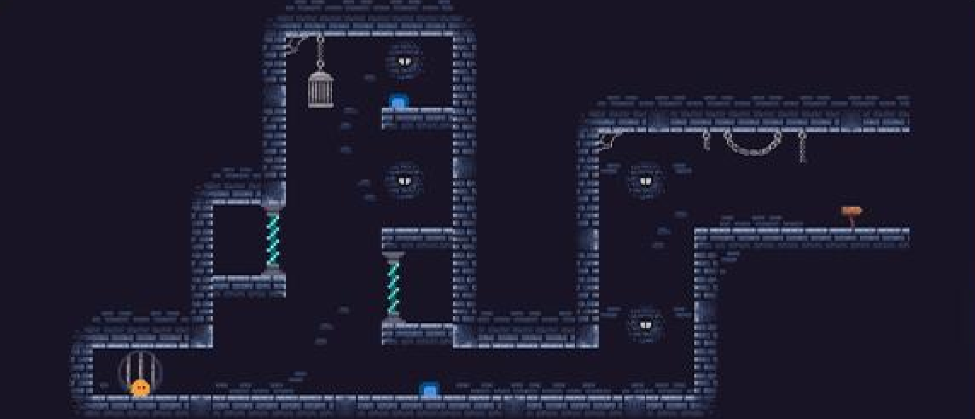
LEVEL 5 reinforces the concept of multiple switch-door systems. The player must manage two sets of switches simultaneously, planning water drop placement to open doors and reach new areas.



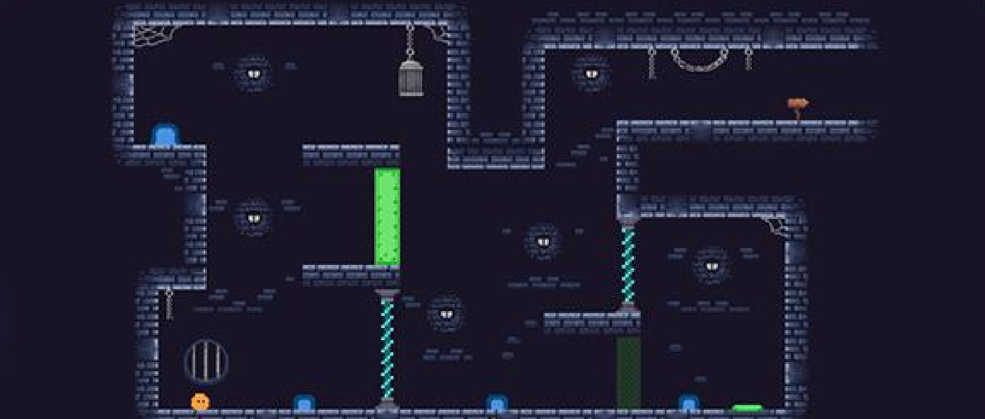
LEVEL 6 challenges players to preplace water drops on doors, then activate switches to trigger chain reactions. This reinforces foresight and strategic use of resources to navigate complex puzzles.



LEVEL 7 introduces reflectors that bounce the wet player or water drops. Players must move water drops onto switch-doors while navigating the room, learning how to use reflection trajectories to solve puzzles.



LEVEL 8 builds on reflection mechanics, requiring players to bounce water drops to higher platforms and then use recoil jumps to reach them. The level emphasizes combining player movement with projectile dynamics.



LEVEL 9 integrates switches and reflectors in one stage. Players must preplace water drops carefully and coordinate their own movement with reflections to reach the goal, synthesizing all previously learned mechanics.

CORE MECHANICS CODE

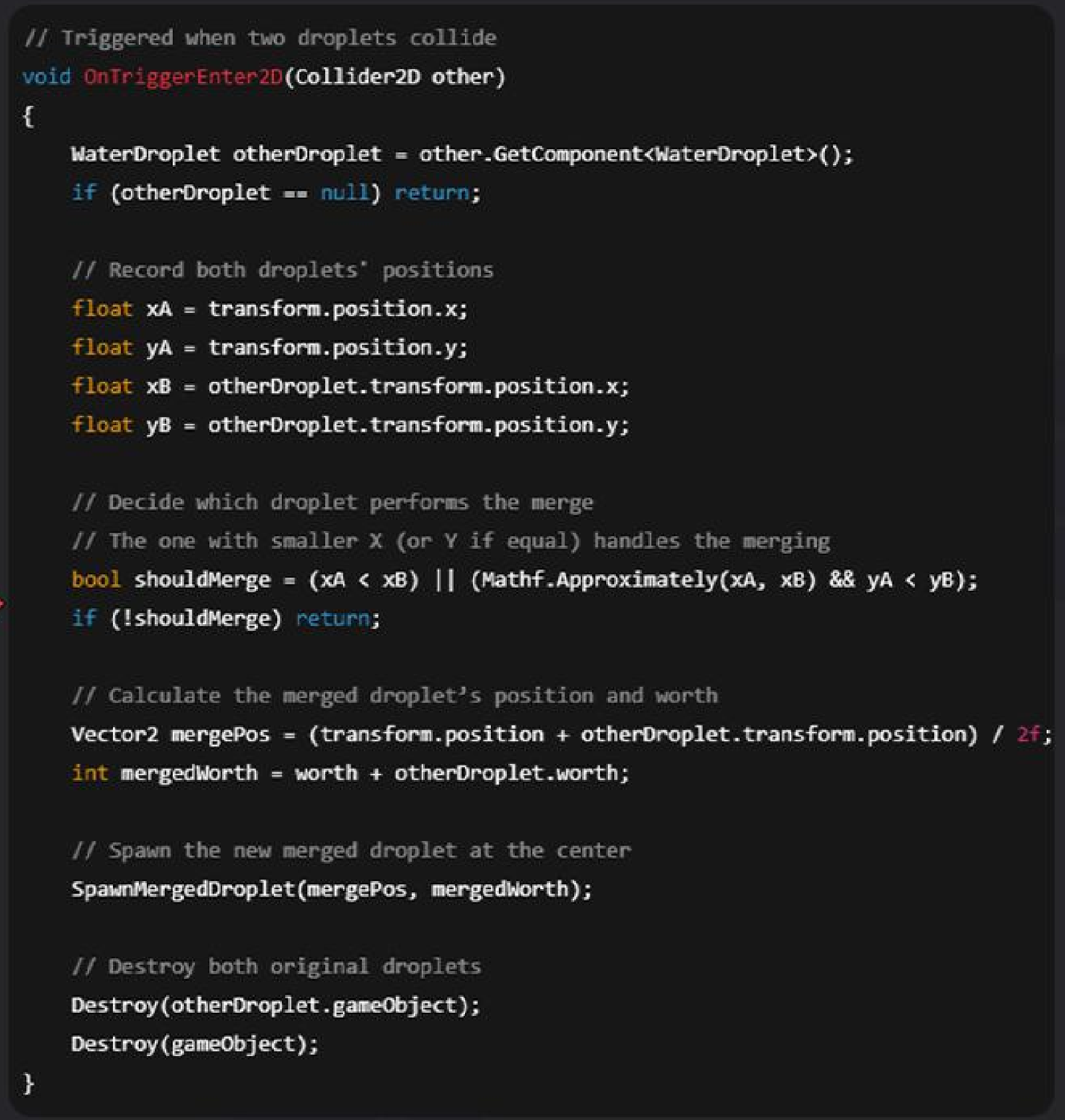
WATER SYSTEM

- **WATER DROPLET ATTRIBUTES AND VISUAL EFFECTS :**
  Each water drop carries an independent "worth" value that determines its size and sprite. A coroutine smoothly interpolates the scale, creating a natural transition when drops merge or are collected.

- **GRAVITY AND PHYSICS CONTROL**
  The gravity system clamps the maximum falling speed to prevent excessive acceleration and ensure consistent collision behavior.

- **DROPLET MERGING LOGIC**
  When two drops collide, the system compares their positions to decide which instance spawns the merged drop, inheriting combined momentum and preventing recursive merges.

- **INERTIA AND DELAYED FRICTION RECOVERY**
  A short friction delay allows the newly merged drop to glide briefly, adding a sense of fluidity before friction gradually restores realism.



```csharp
// Triggered when two droplets collide
void OnTriggerEnter2D(Collider2D other)
{
    WaterDroplet otherDroplet = other.GetComponent<WaterDroplet>();
    if (otherDroplet == null) return;

    // Record both droplets' positions
    float xA = transform.position.x;
    float yA = transform.position.y;
    float xB = otherDroplet.transform.position.x;
    float yB = otherDroplet.transform.position.y;

    // Decide which droplet performs the merge
    // The one with smaller X (or Y if equal) handles the merging
    bool shouldMerge = (xA < xB) || (Mathf.Approximately(xA, xB) && yA < yB);
    if (!shouldMerge) return;

    // Calculate the merged droplet's position and worth
    Vector2 mergePos = (transform.position + otherDroplet.transform.position) / 2f;
    int mergedWorth = worth + otherDroplet.worth;

    // Spawn the new merged droplet at the center
    SpawnMergedDroplet(mergePos, mergedWorth);

    // Destroy both original droplets
    Destroy(otherDroplet.gameObject);
    Destroy(gameObject);
}
```

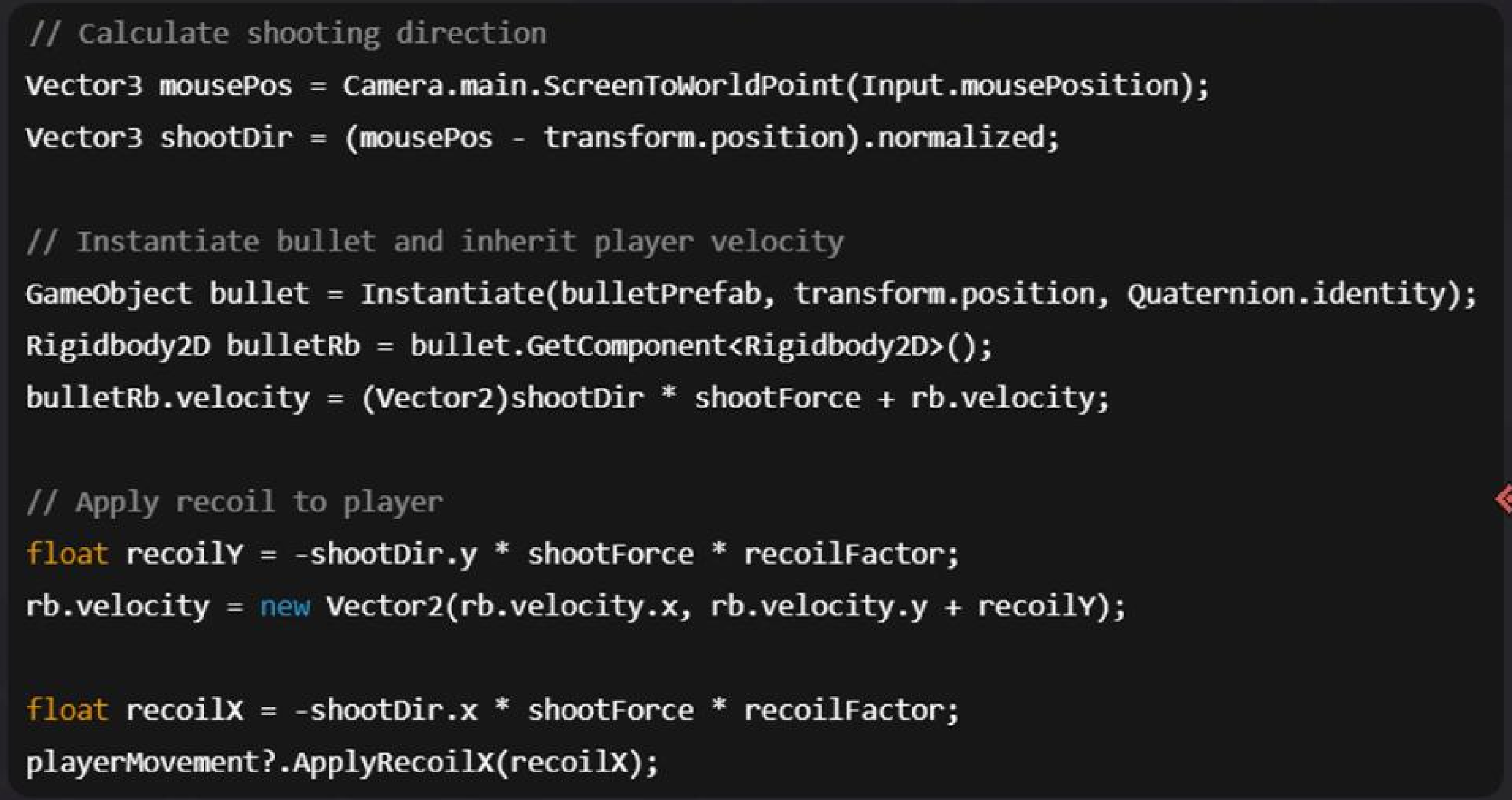
SHOOTING SYSTEM

- **SHOOTING & DIRECTION CALCULATION**
  Shooting water droplets consumes one unit of water and generates a recoil force on the player in the opposite direction. The shot direction is calculated from the player to the mouse position, and the bullet inherits the player's current velocity to maintain smooth motion.

  Vertical recoil is applied instantly, while horizontal recoil is gradually damped through the movement system, allowing precise control during recoil jumps.

- **VISUAL FEEDBACK & PARTICLES**
  Water splash particles are spawned dynamically based on the shooting direction to provide immediate visual feedback. Shots aimed upwards offset the particles slightly above the player, while downward shots offset them slightly below.

  The particle system is rotated to match the shot angle, reinforcing the connection between the action and its visual effect, and enhancing the player's sense of responsiveness and control.



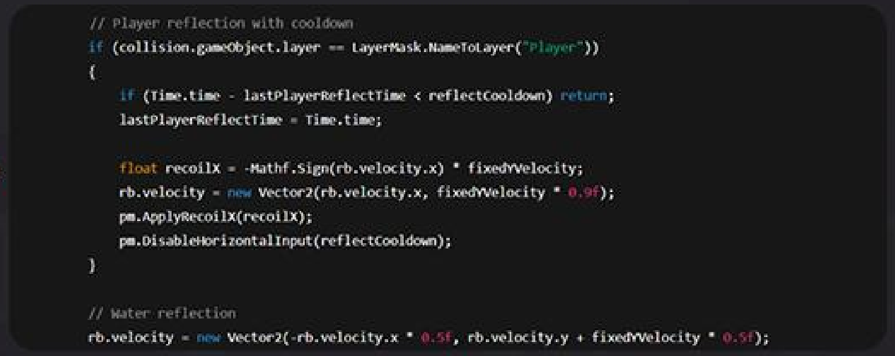
```csharp
// Calculate shooting direction
Vector3 mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
Vector3 shootDir = (mousePos - transform.position).normalized;

// Instantiate bullet and inherit player velocity
GameObject bullet = Instantiate(bulletPrefab, transform.position, Quaternion.identity);
Rigidbody2D bulletRb = bullet.GetComponent<Rigidbody2D>();
bulletRb.velocity = (Vector2)shootDir * shootForce + rb.velocity;

// Apply recoil to player
float recoilY = -shootDir.y * shootForce * recoilFactor;
rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y + recoilY);

float recoilX = -shootDir.x * shootForce * recoilFactor;
playerMovement?.ApplyRecoilX(recoilX);
```

WATER SYSTEM

- **SPIKE TRAP**
  Spike traps instantly kill the player upon contact. The trap checks if the colliding object is the player and then calls the GameController's death method, ensuring immediate and clear feedback for failure states in the level.

- **REFLECTER**
  The reflecter allows players and water droplets to bounce off surfaces with fixed vertical velocity. Players are subject to a short cooldown to prevent repeated bounces, while water droplets are reflected immediately. Horizontal recoil is applied to players to create a dynamic movement effect.

- **DOOR & BUTTON**
  The door system allows doors to be opened or closed based on player interaction with buttons. Each door can start in an open or closed state, and its state can be toggled at runtime. This system provides designers with a simple way to link player actions to environmental changes.



```csharp
// Player reflection with cooldown
if (collision.gameObject.layer == LayerMask.NameToLayer("Player"))
{
    if (Time.time - lastPlayerReflectTime < reflectCooldown) return;
    lastPlayerReflectTime = Time.time;

    float recoilX = -Mathf.Sign(rb.velocity.x) * fixedYVelocity;
    rb.velocity = new Vector2(rb.velocity.x, fixedYVelocity * 0.9f);
    pm.ApplyRecoilX(recoilX);
    pm.DisableHorizontalInput(reflectCooldown);
}

// Water reflection
rb.velocity = new Vector2(-rb.velocity.x * 0.5f, rb.velocity.y + fixedYVelocity * 0.5f);
```

TESTING & FEEDBACK

PLAYER CONTROL SYSTEM

- **CORE MOVEMENT AND PHYSICS HANDLING**

The player' s movement system is built upon Unity' s 2D physics engine, providing smooth and responsive motion through direct manipulation of Rigidbody2D velocity. It integrates several advanced mechanics to refine control precision and gameplay feel :

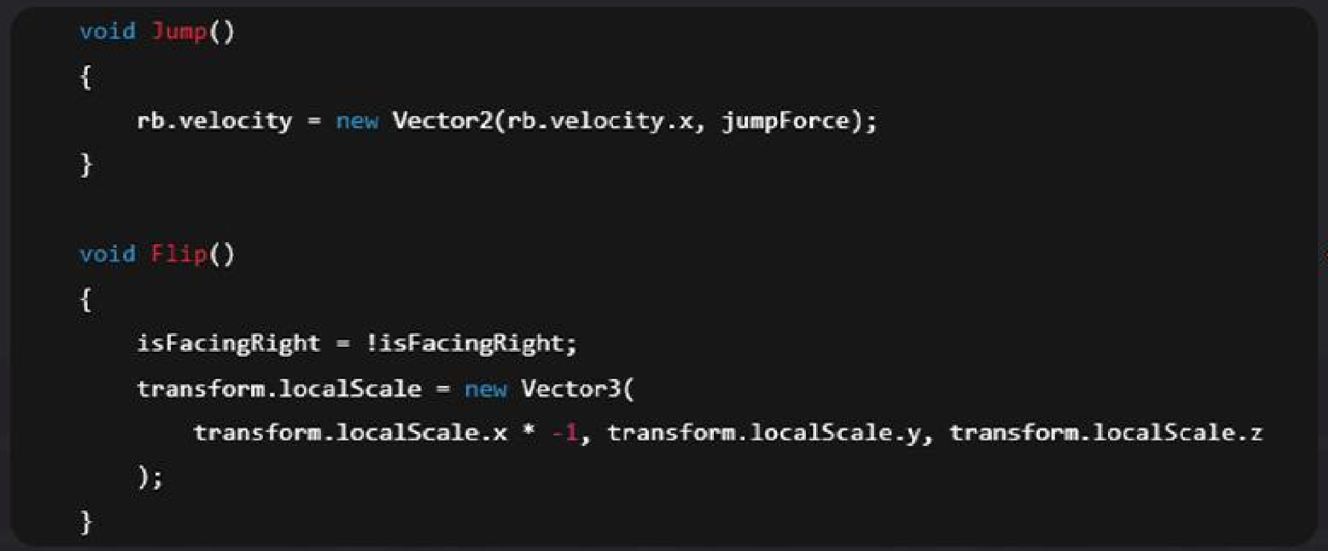
- Coyote Time : Allows the player to jump shortly after leaving a platform, improving input forgiveness and gameplay fluidity.
- Variable Gravity : Applies different gravity scales during ascent and descent, creating a more natural and responsive jump arc.
- Speed Limitation : Caps the player' s maximum velocity on both X and Y axes to maintain stable physics behavior.
- Directional Flip : Smoothly flips the player' s facing direction based on horizontalMovement, ensuring consistent animation flow.

- **RECOIL AND INPUT MANAGEMENT**

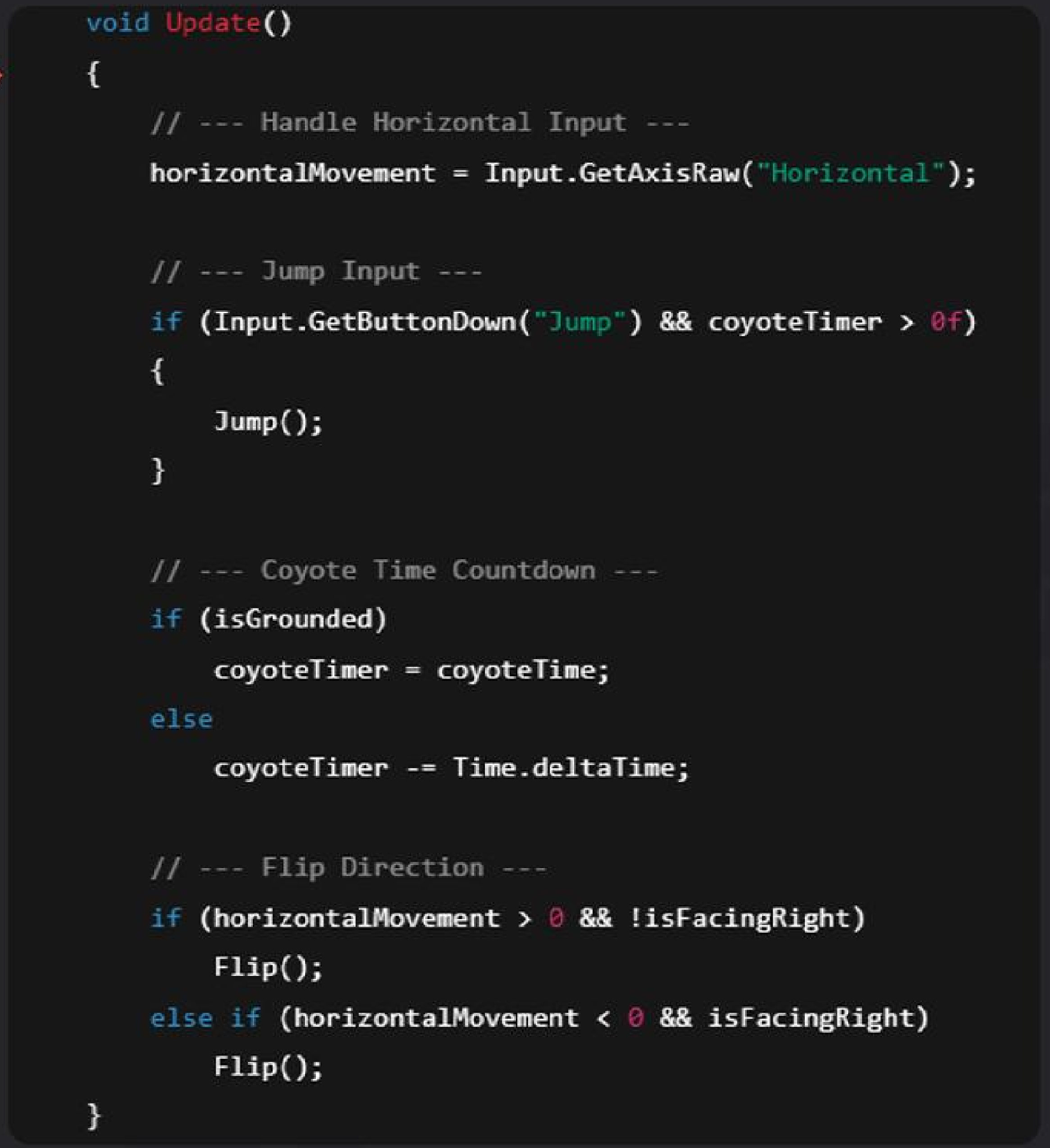
When the player shoots, a recoil force is applied to the movement system. Horizontal recoil decays gradually to simulate inertia, while a short input lock ensures consistent physics behavior and prevents control conflicts during reflection or collision events.

- **DYNAMIC SIZE AND VISUAL FEEDBACK**

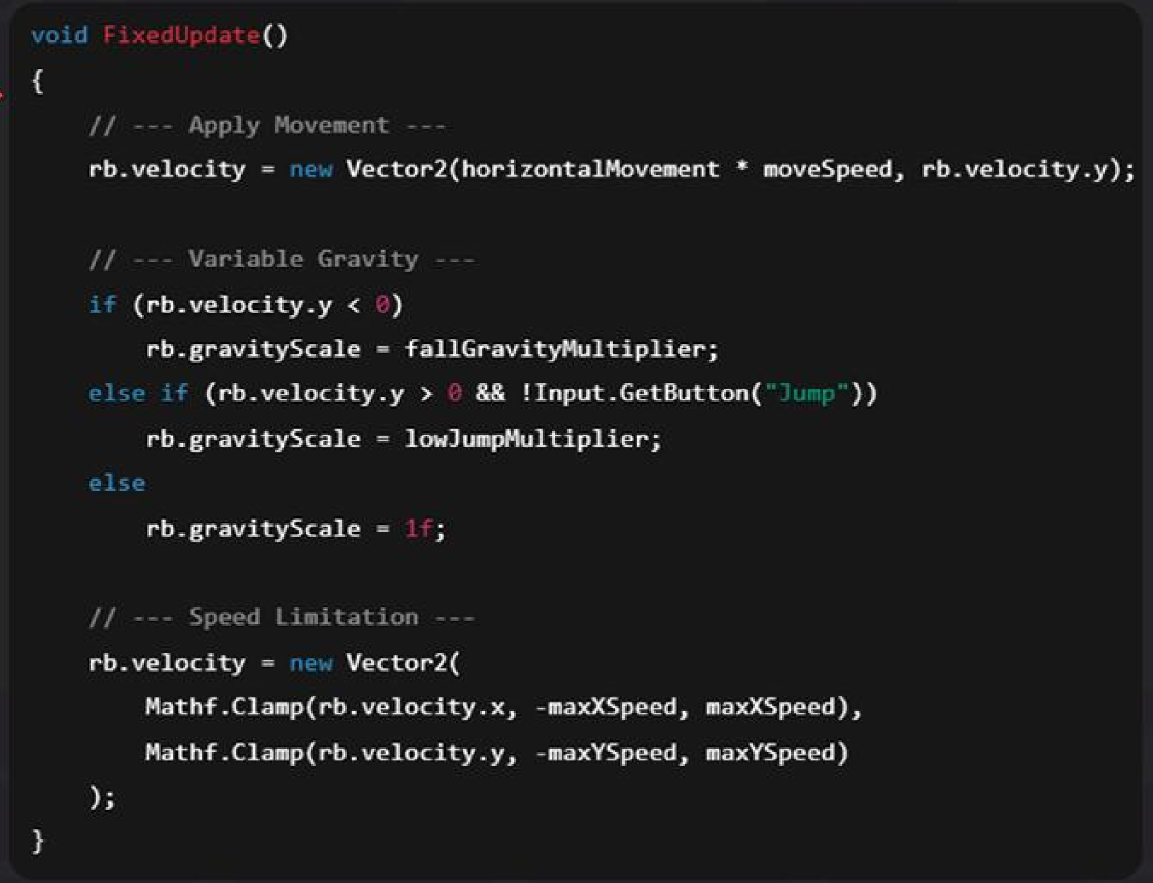
The player' s size and material dynamically respond to water resource changes. When water is consumed, the player gradually shrinks and desaturates, providing immediate visual feedback that links gameplay mechanics with the visual identity of the character.



```
void Jump()
{
    rb.velocity = new Vector2(rb.velocity.x, jumpForce);
}

void Flip()
{
    isFacingRight = !isFacingRight;
    transform.localScale = new Vector3(
        transform.localScale.x * -1, transform.localScale.y, transform.localScale.z
    );
}
```



```
void Update()
{
    // --- Handle Horizontal Input ---
    horizontalMovement = Input.GetAxisRaw("Horizontal");

    // --- Jump Input ---
    if (Input.GetButtonDown("Jump") && coyoteTimer > 0f)
    {
        Jump();
    }

    // --- Coyote Time Countdown ---
    if (isGrounded)
        coyoteTimer = coyoteTime;
    else
        coyoteTimer -= Time.deltaTime;

    // --- Flip Direction ---
    if (horizontalMovement > 0 && !isFacingRight)
        Flip();
    else if (horizontalMovement < 0 && isFacingRight)
        Flip();
}
```



```
void FixedUpdate()
{
    // --- Apply Movement ---
    rb.velocity = new Vector2(horizontalMovement * moveSpeed, rb.velocity.y);

    // --- Variable Gravity ---
    if (rb.velocity.y < 0)
        rb.gravityScale = fallGravityMultiplier;
    else if (rb.velocity.y > 0 && !Input.GetButton("Jump"))
        rb.gravityScale = lowJumpMultiplier;
    else
        rb.gravityScale = 1f;

    // --- Speed Limitation ---
    rb.velocity = new Vector2(
        Mathf.Clamp(rb.velocity.x, -maxXSpeed, maxXSpeed),
        Mathf.Clamp(rb.velocity.y, -maxYSpeed, maxYSpeed)
    );
}
```

GAME TESTING

During development, I followed a level-based testing approach, where each stage was designed to introduce and validate specific core mechanics.

In the early phase of development — particularly during Level 1 — I completed and thoroughly tested the core gameplay systems, including:the Water Droplet System, the Shooting System, the Spike Trap, and the Player Movement System.

As new mechanics were introduced in later stages, I performed targeted feature testing within the first level where each mechanic appeared.
For example:

- Level 4 served as the initial testing ground for the Button & Door Mechanism.
- Level 7 introduced the Reflecter Mechanism, which required extensive debugging of physics-based collision logic and reflection timing.

Throughout playtesting, I intentionally explored edge cases—such as rapid input sequences, overlapping collisions, or abnormal droplet interactions—to ensure the stability and robustness of the system.

GAME FEEDBACK

After completing and exporting the game, I invited a friend to participate in playtesting. Instead of offering direct instructions, I observed his playthrough closely, noting where he struggled or hesitated during specific levels. This allowed me to reflect on whether the in-game guidance and level design were sufficiently clear and intuitive.

Through this process, I discovered—thanks to his feedback—that the final level contained a design flaw allowing players to bypass part of the intended challenge and clear the stage too easily. I quickly adjusted the layout and refined the level structure, successfully eliminating the unintended shortcut.