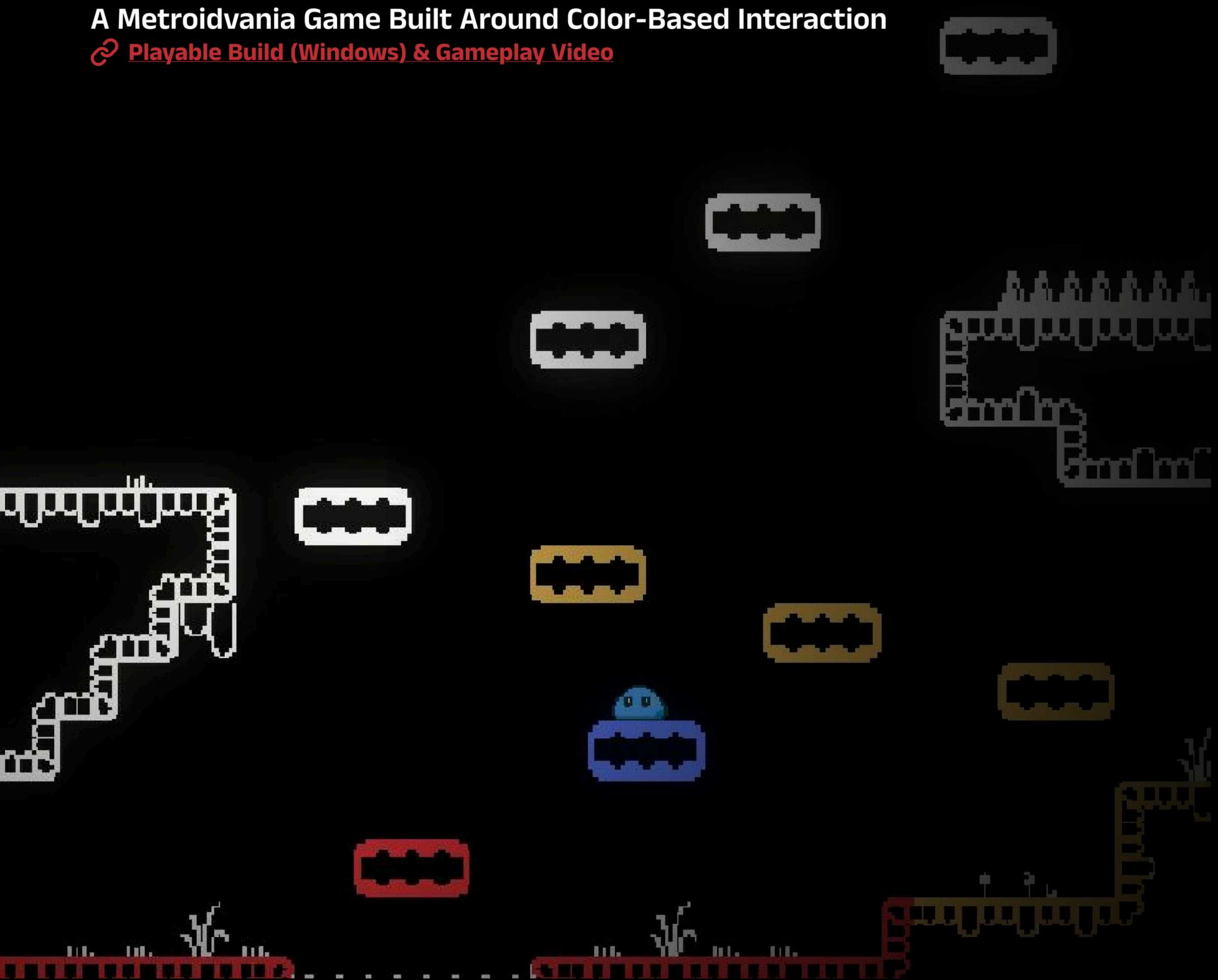
Coloring

A Metroidvania Game Built Around Color-Based Interaction

Playable Build (Windows) & Gameplay Video



Inspiration

Reference

Animal Well

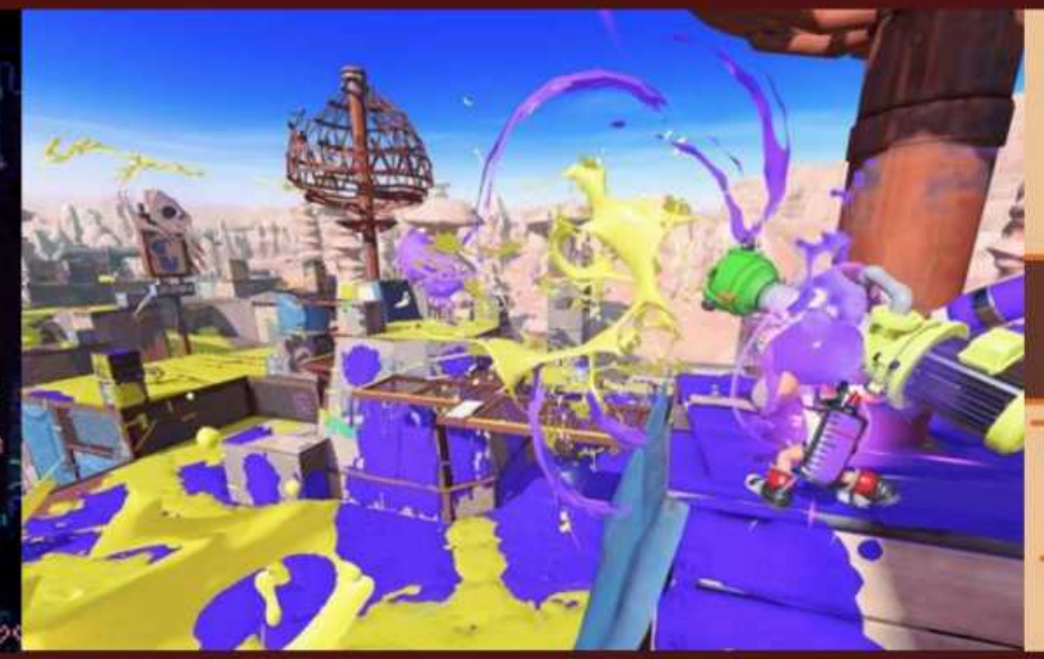
Non-combat Metroidvania with environmental puzzles and nonlinear exploration.

Splatoon

Strong emphasis on color as feedback, dynamically altering the environment with ink.

Leap Year

Minimalist design where abilities are discovered through intuition and experimentation.



Overview of My Game

This is a non-linear, combat-free Metroidvania focused on exploration and puzzle-solving. Without explicit instructions, players discover hidden abilities through experimentation and environmental cues.

Color is both a tool and a form of expression. Players coloring the world as they move through it, they reshape their surroundings, unlock new possibilities, and leave behind a visible trail of their journey.

Art Style

The visual style takes inspiration from early arcade games such as Space Invaders and Pac-Man, using a bold, high-contrast pixel aesthetic.

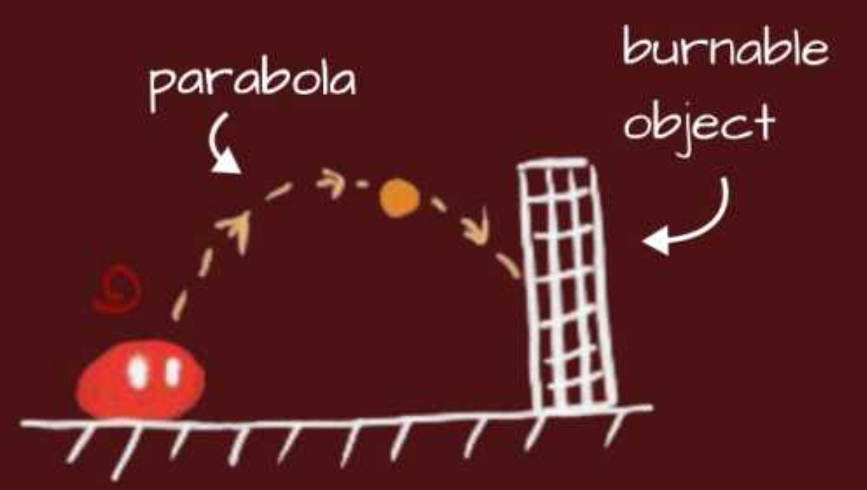
Game Setting

The Colorful Kingdom is divided into three rival nations: Red, Blue, and Yellow. When a black-and-white storm steals all color from the world, each nation sends a knight. Though they start as enemies, they learn to work together, combining their powers to restore color and bring back a brighter future.

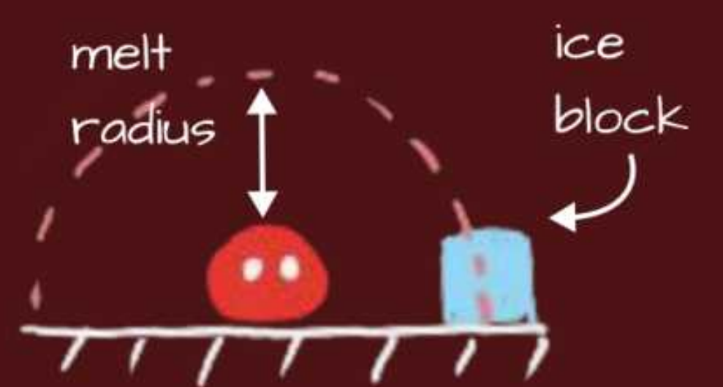
Game Mechanic

There are three primary colors and three secondary (mixed) colors. Each color represents a unique ability and interacts differently with the environment.

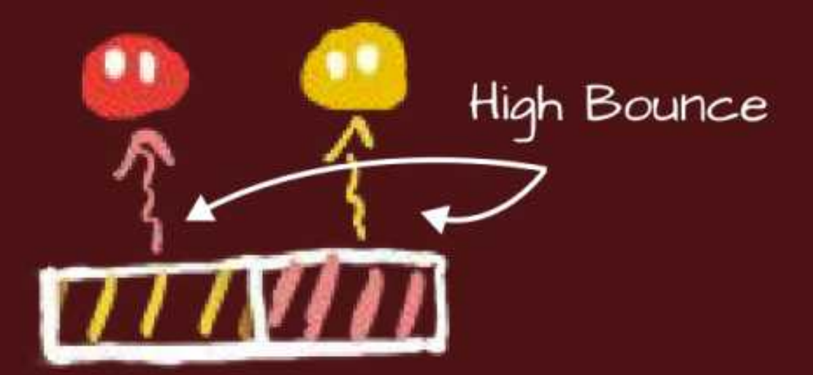
Red



Shoots a fireball in a parabolic arc. Direction and power are chargeable.



Melts nearby ice blocks within a radius around the player.

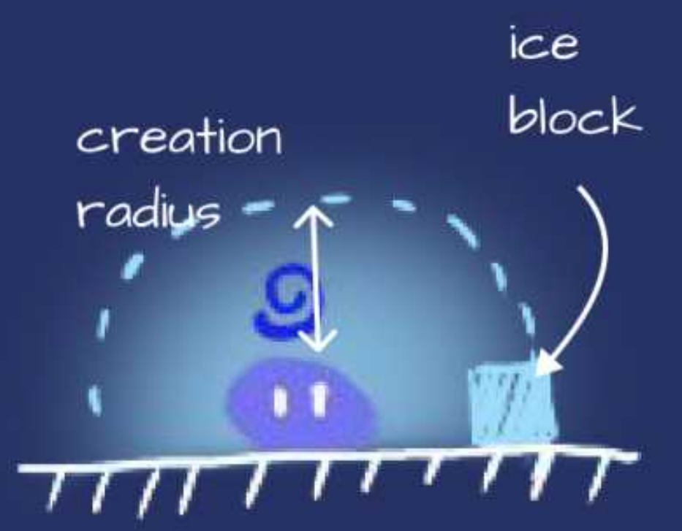


Red on Yellow tile → High bounce

Yellow on Red tile → High bounce

These interactions make the player jump higher than normal.

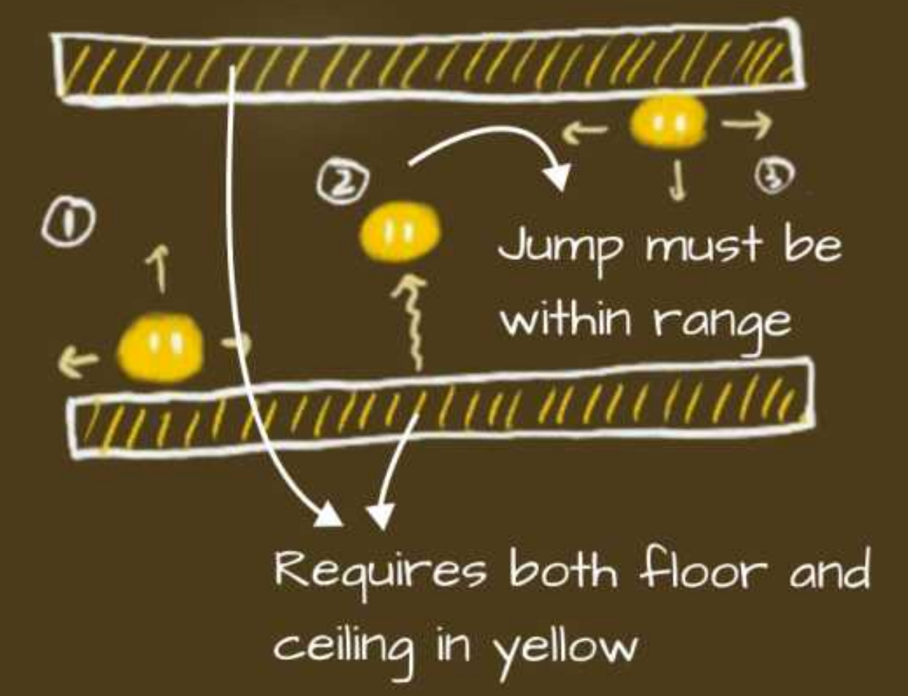
Blue



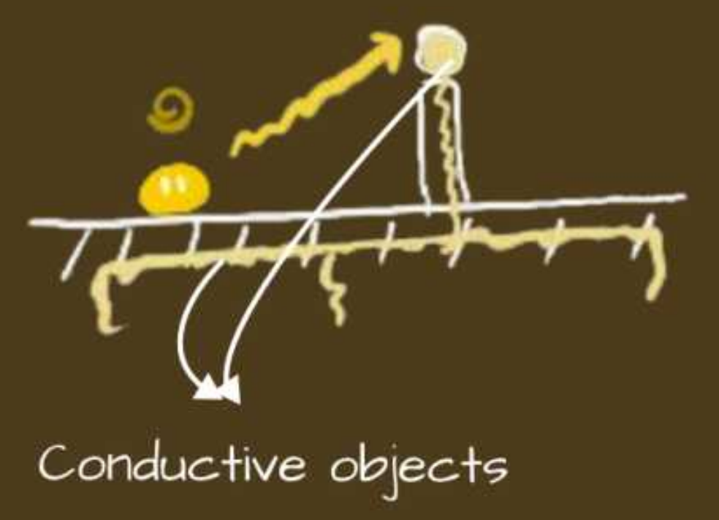
Creates ice blocks in a set area that can be used as platforms or to press trigger.

Only color able to move underwater: 4-directional movement with buoyancy.

Yellow

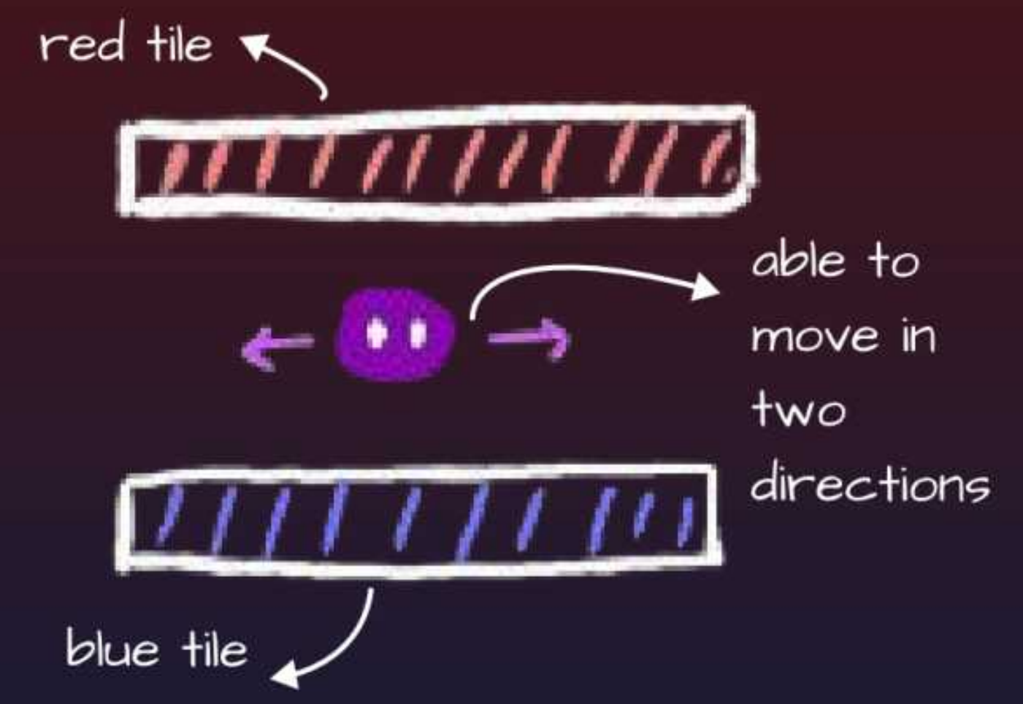

①
②
Jump must be within range
Requires both floor and ceiling in yellow

Inverts gravity on yellow platforms, letting the player walk along both floor and ceiling.



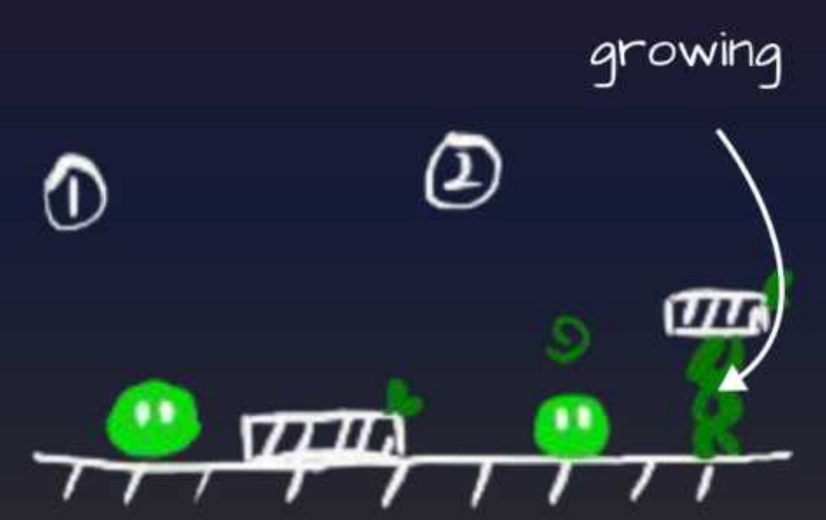
Sends electricity through conductive objects (including water) to activate distant mechanisms.

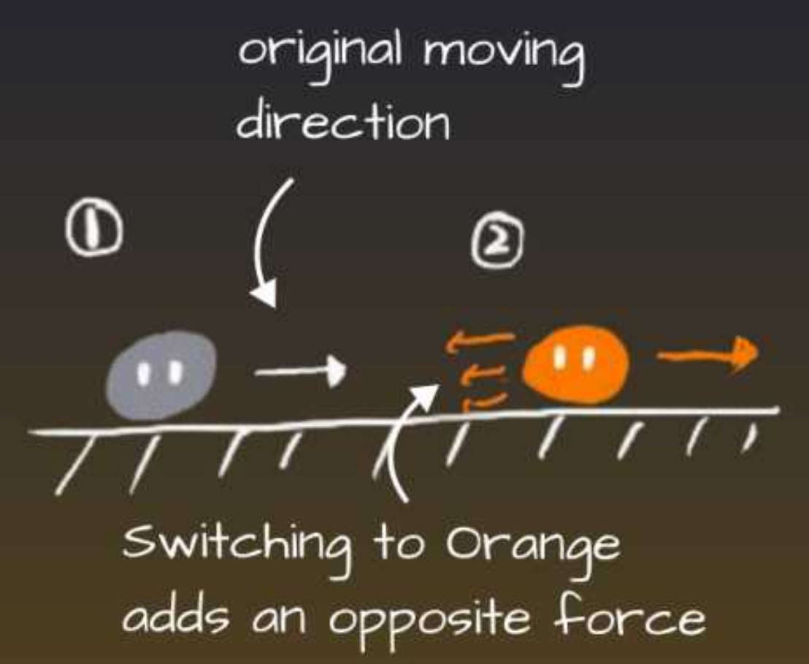
Secondary Colors



Allows the player to hover between red and blue platforms, giving limited control in mid-air. The player can only move left and right while hovering.

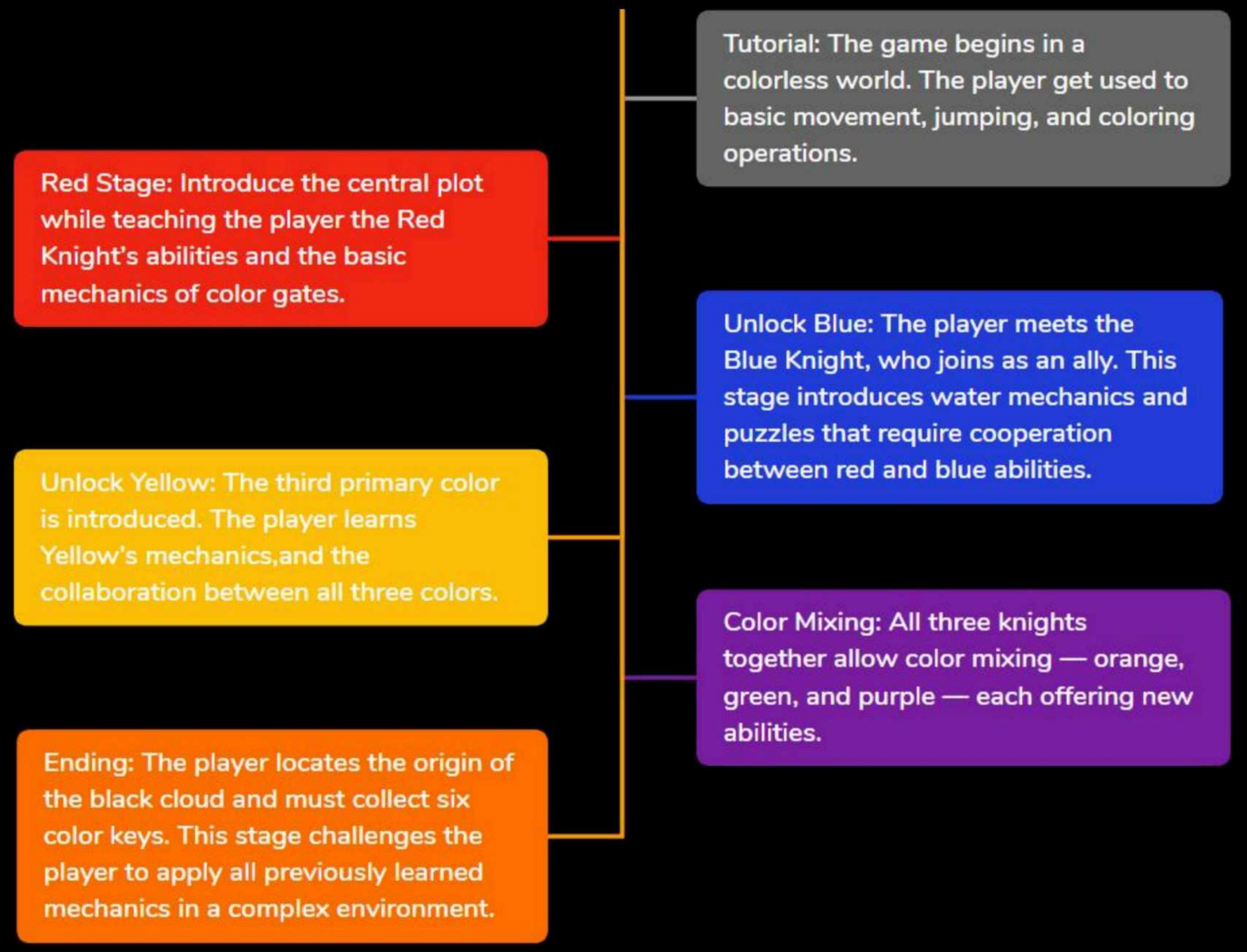
Stimulates plant growth, creating vines and platforms.




original moving direction
①
②
Switching to Orange adds an opposite force

Shoots a burst of ink in the opposite direction of movement, pushing the player forward.

Game Flow



Tutorial: The game begins in a colorless world. The player get used to basic movement, jumping, and coloring operations.

Red Stage: Introduce the central plot while teaching the player the Red Knight's abilities and the basic mechanics of color gates.

Unlock Blue: The player meets the Blue Knight, who joins as an ally. This stage introduces water mechanics and puzzles that require cooperation between red and blue abilities.

Unlock Yellow: The third primary color is introduced. The player learns Yellow's mechanics,and the collaboration between all three colors.

Color Mixing: All three knights together allow color mixing — orange, green, and purple — each offering new abilities.

Ending: The player locates the origin of the black cloud and must collect six color keys. This stage challenges the player to apply all previously learned mechanics in a complex environment.

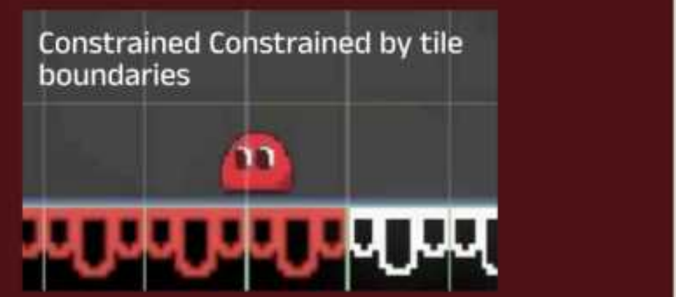
Collectible System: The game includes a collectible system where optional color pieces are hidden along side paths. Each piece restores part of a painting in a special room, and as more are found, the image gradually becomes complete, marking the player's progress.

Core System Design & Implementation

Coloring is one of the core mechanics of the game, and I invested significant effort into building a system that feels smooth and efficient.

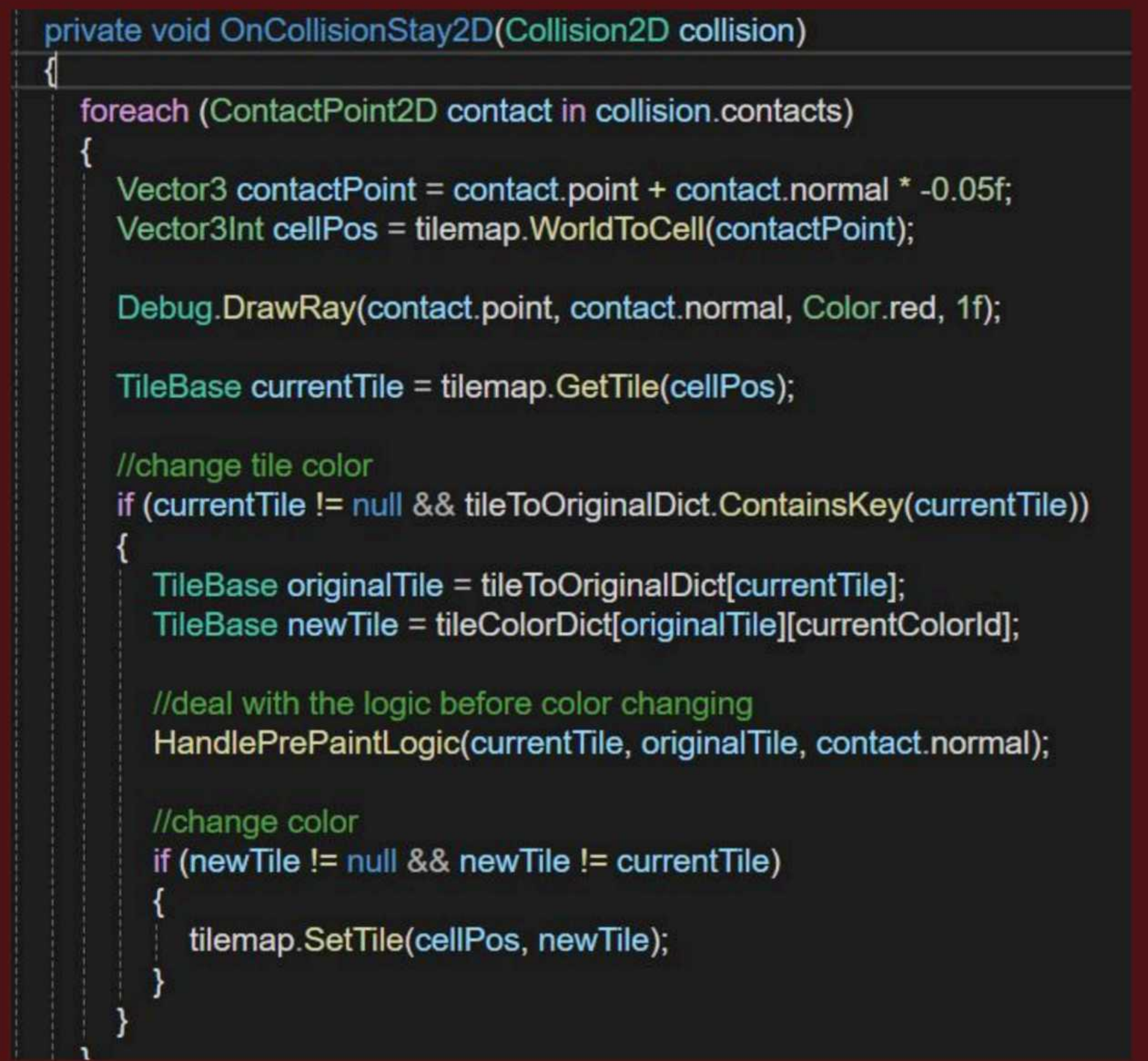
Shader-Based Coloring

V1: Coloring by detecting the player's collisions with the **tilemap**.



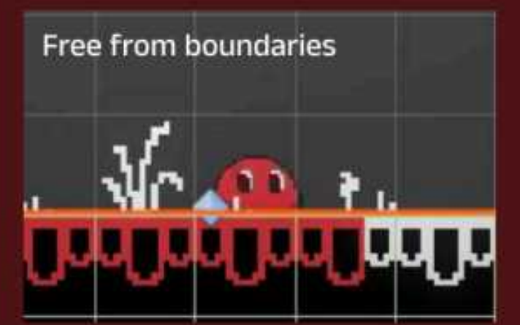
Limitation

- The coloring constrained by tile boundaries.
- Difficult for saved game



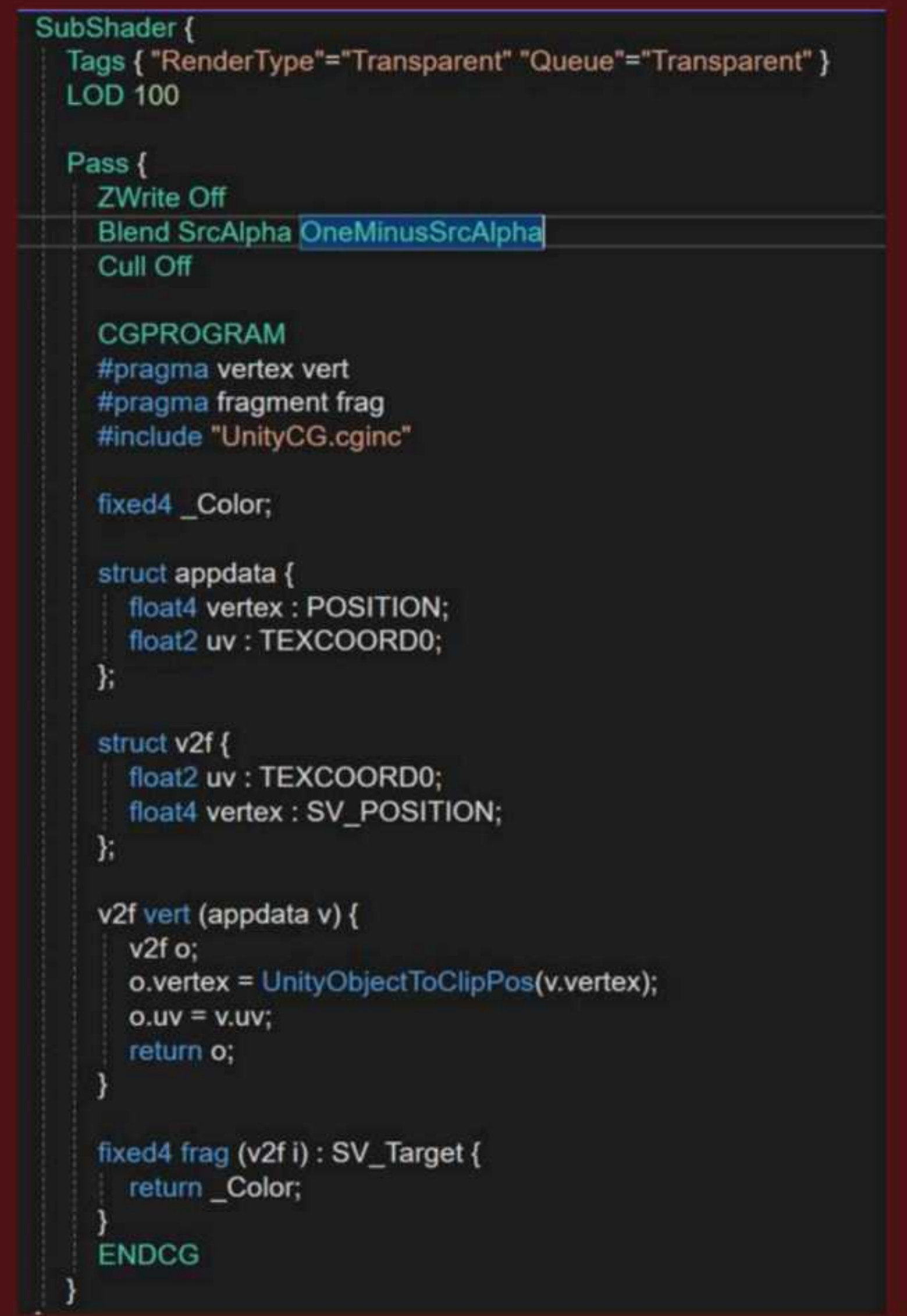
```
private void OnCollisionStay2D(Collision2D collision)
{
    foreach (ContactPoint2D contact in collision.contacts)
    {
        Vector3 contactPoint = contact.point + contact.normal * -0.05f;
        Vector3Int cellPos = tilemap.WorldToCell(contactPoint);

        Debug.DrawRay(contact.point, contact.normal, Color.red, 1f);

        TileBase currentTile = tilemap.GetTile(cellPos);

        //change tile color
        if (currentTile != null && tileToOriginalDict.ContainsKey(currentTile))
        {
            TileBase originalTile = tileToOriginalDict[currentTile];
            TileBase newTile = tileColorDict[originalTile][currentColorId];

            //deal with the logic before color changing
            HandlePrePaintLogic(currentTile, originalTile, contact.normal);

            //change color
            if (newTile != null && newTile != currentTile)
            {
                tilemap.SetTile(cellPos, newTile);
            }
        }
```

V2: switched to shader-based solution by using RenderTexture system.

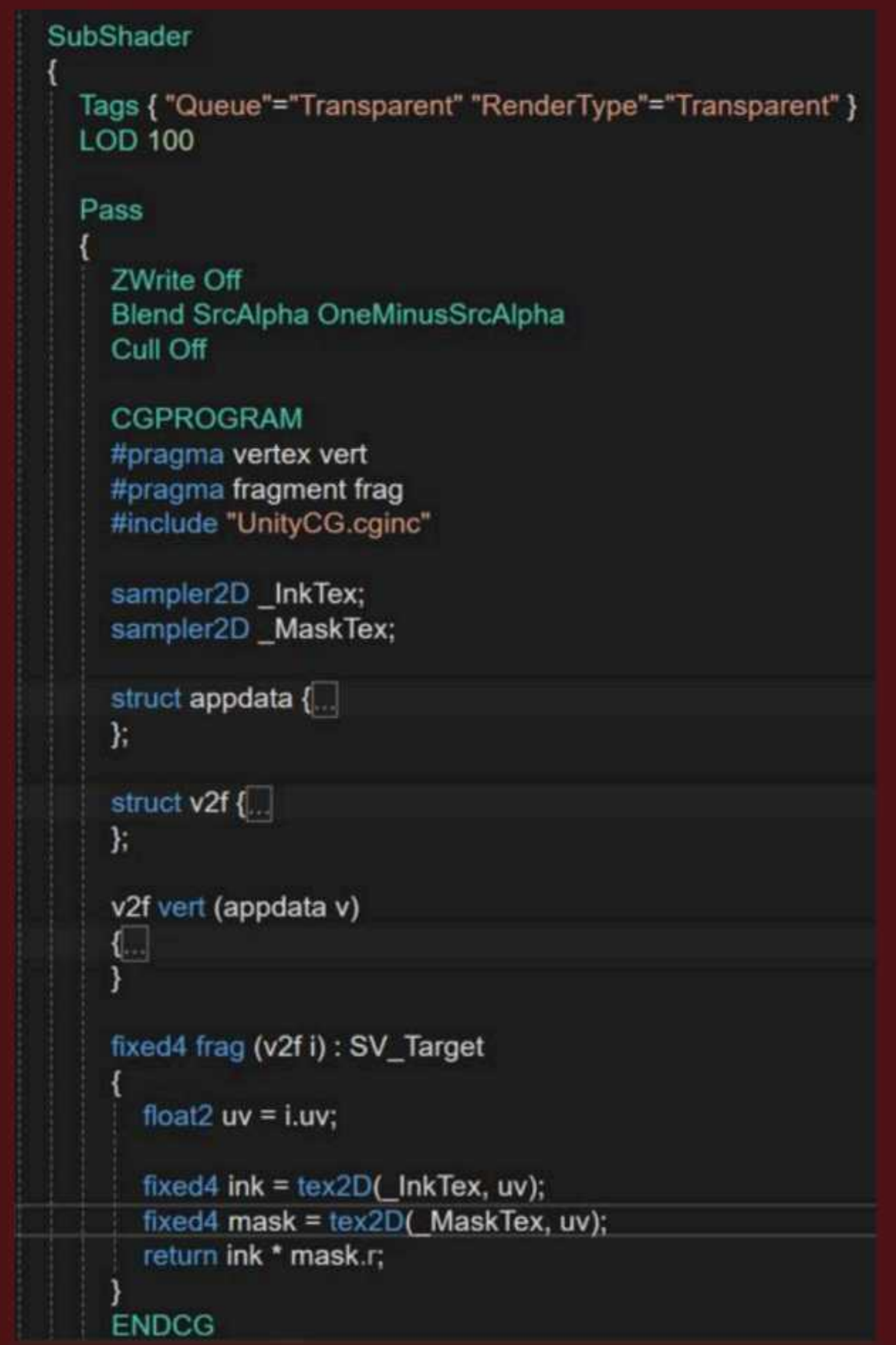


Advantage

- An Ink Mask layer is rendered on top of the map.
- The player's world position is used to draw color directly onto this mask in real time.
- A mask texture is used to restrict coloring only to designated white areas of the map



```
SubShader {
    Tags { "RenderType"="Transparent" "Queue"="Transparent" }
    LOD 100

    Pass {
        ZWrite Off
        Blend SrcAlpha OneMinusSrcAlpha
        Cull Off

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        fixed4 _Color;

        struct appdata {
            float4 vertex : POSITION;
            float2 uv : TEXCOORD0;
        };

        struct v2f {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;
        };

        v2f vert (appdata v) {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = v.uv;
            return o;
        }

        fixed4 frag (v2f i) : SV_Target {
            return _Color;
        }
        ENDCG
    }
```



```
SubShader
{
    Tags { "Queue"="Transparent" "RenderType"="Transparent" }
    LOD 100

    Pass
    {
        ZWrite Off
        Blend SrcAlpha OneMinusSrcAlpha
        Cull Off

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        sampler2D _InkTex;
        sampler2D _MaskTex;

        struct appdata {...
        };

        struct v2f {...
        };

        v2f vert (appdata v)
        {...
        }

        fixed4 frag (v2f i) : SV_Target
        {
            float2 uv = i.uv;

            fixed4 ink = tex2D(_InkTex, uv);
            fixed4 mask = tex2D(_MaskTex, uv);
            return ink * mask.r;
        }
        ENDCG
```

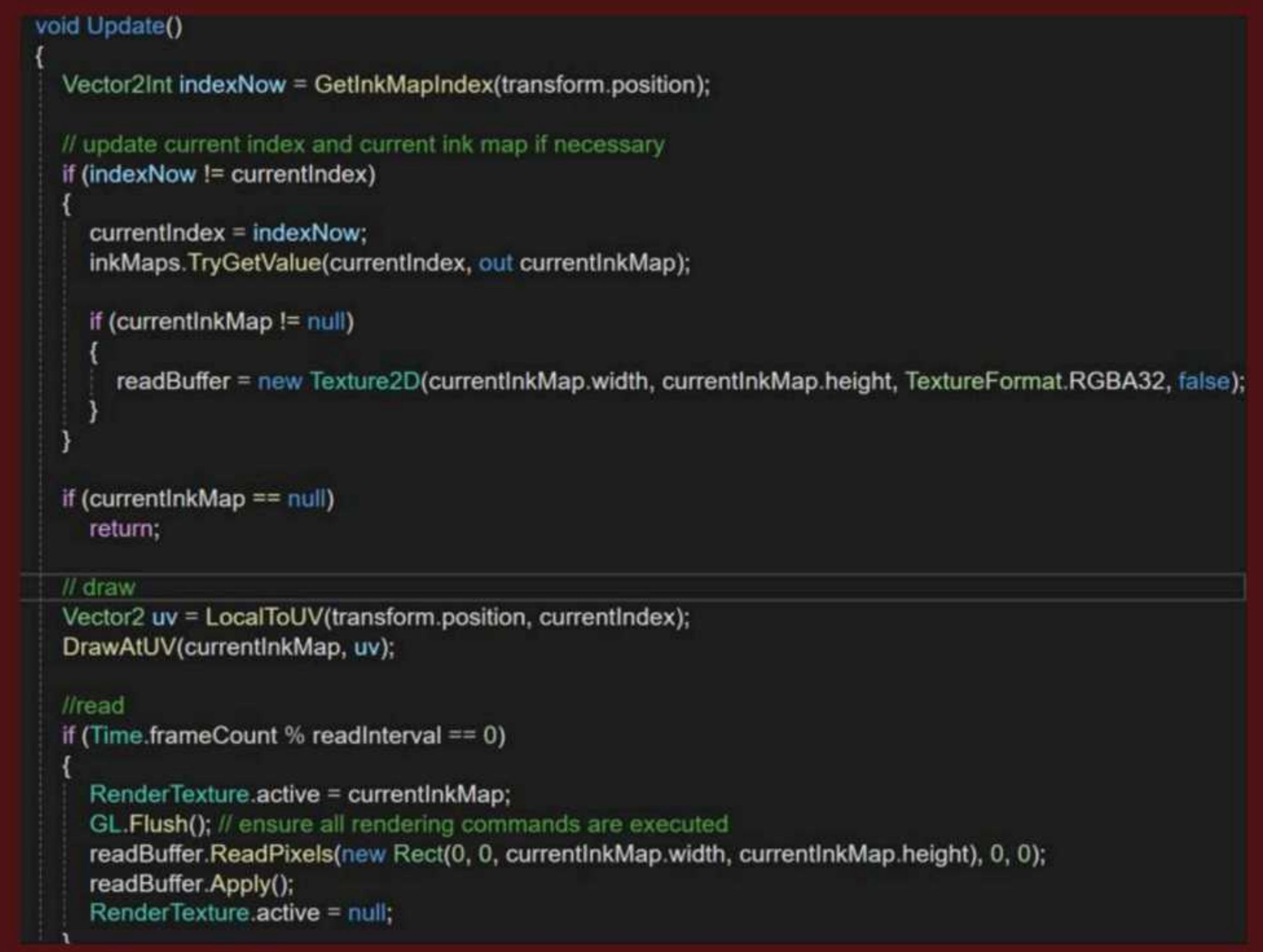
V3: Regional Ink Masks for Performance Optimization

Problem

- Needed for mechanics like "player reacts to the color underfoot."
- GPU-to-CPU readback from a large, high-resolution Ink Mask caused frame drops.

Solution

- Divided the map into multiple regions, each with its own smaller Ink Mask.
- At runtime, the system calculates which region the player is in and only reads from the corresponding small Ink Mask. The performance was much better.

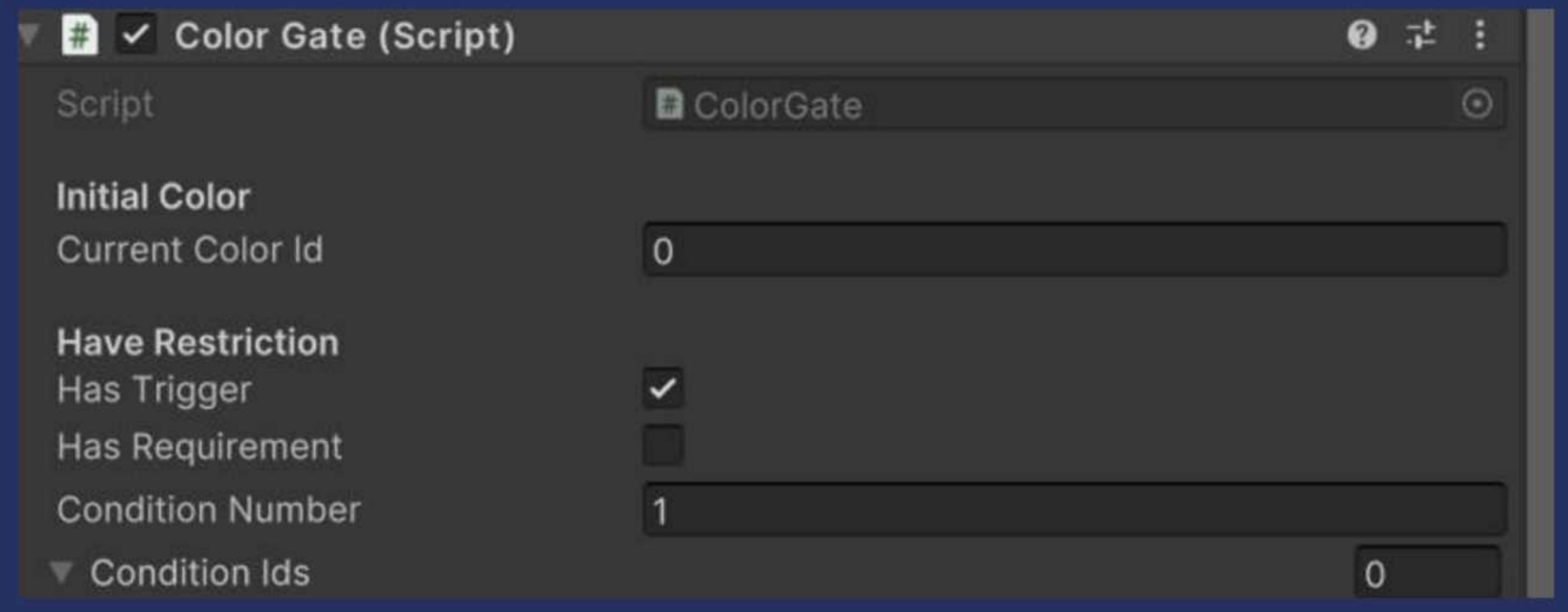


```
void Update()
{
    Vector2Int indexNow = GetInkMapIndex(transform.position);

    // update current index and current ink map if necessary
    if (indexNow != currentIndex)
    {
        currentIndex = indexNow;
        inkMaps.TryGetValue(currentIndex, out currentInkMap);

        if (currentInkMap != null)
        {
            readBuffer = new Texture2D(currentInkMap.width, currentInkMap.height, TextureFormat.RGBA32, false);
        }
    }

    if (currentInkMap == null)
        return;

    // draw
    Vector2 uv = LocalToUV(transform.position, currentIndex);
    DrawAtUV(currentInkMap, uv);

    //read
    if (Time.frameCount % readInterval == 0)
    {
        RenderTexture.active = currentInkMap;
        GL.Flush(); // ensure all rendering commands are executed
        readBuffer.ReadPixels(new Rect(0, 0, currentInkMap.width, currentInkMap.height), 0, 0);
        readBuffer.Apply();
        RenderTexture.active = null;
    }
```

Color Gate

Color gates restrict access based on the player's current color. They are designed to reinforce color-based puzzle logic and progression.

- **Fixed Color Gates**: Can only be passed by characters matching the gate's specific color.
- **Paintable Gates**: start as gray and must be manually painted the according trigger to open.
- **Multi-Trigger Gates:** Require multiple color buttons to be activated. Variants include:
  - All buttons must be the same color
  - Each button must be a specific different color
- **Contextual Gates:** Appear flexible but have an optimal solution based on context — for example, painting a gate blue near water.



Color Gate (Script)

Script: ColorGate

Initial Color

Current Color Id: 0

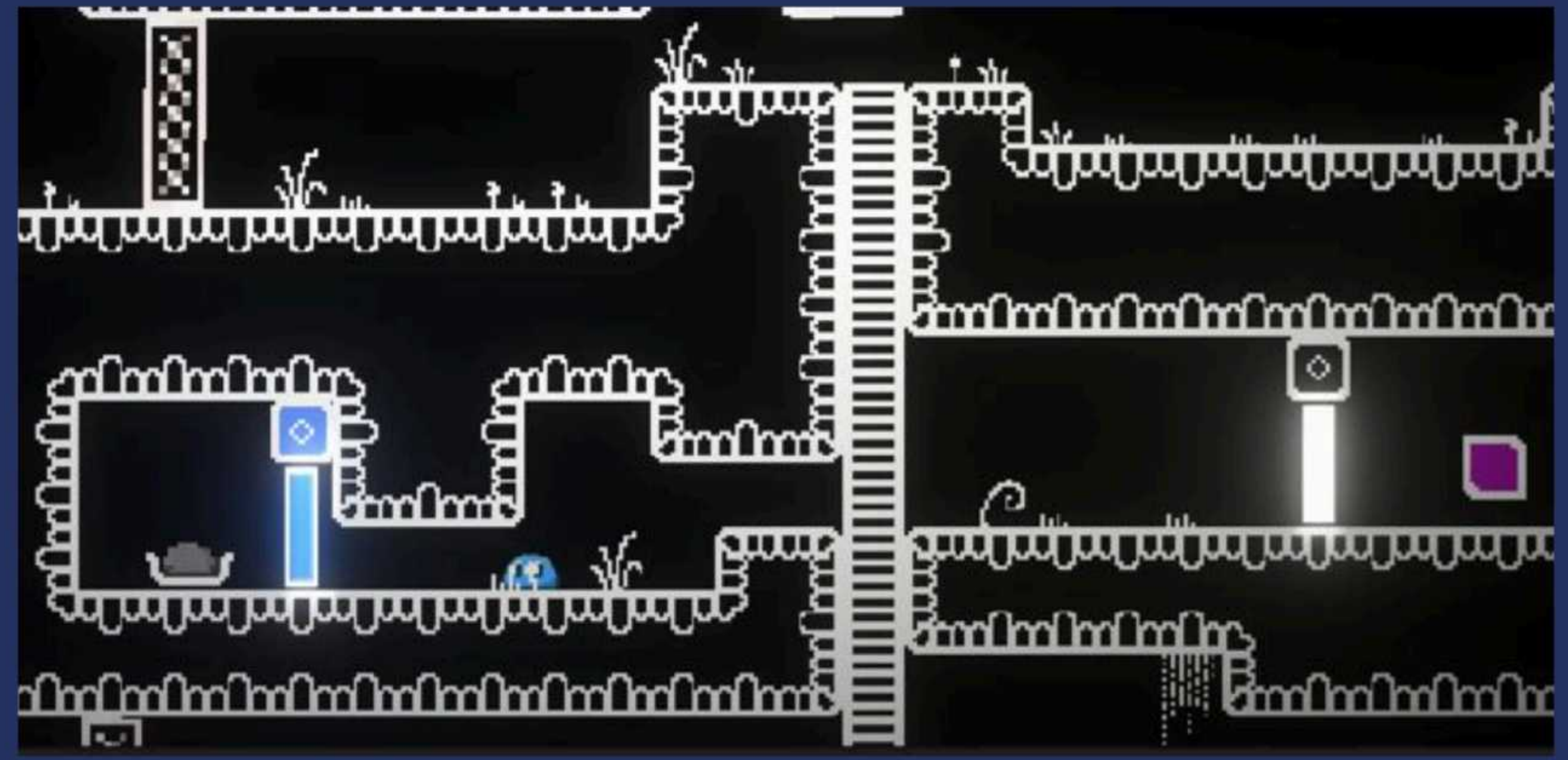
Have Restriction

Has Trigger: ☑

Has Requirement: ☐

Condition Number: 1

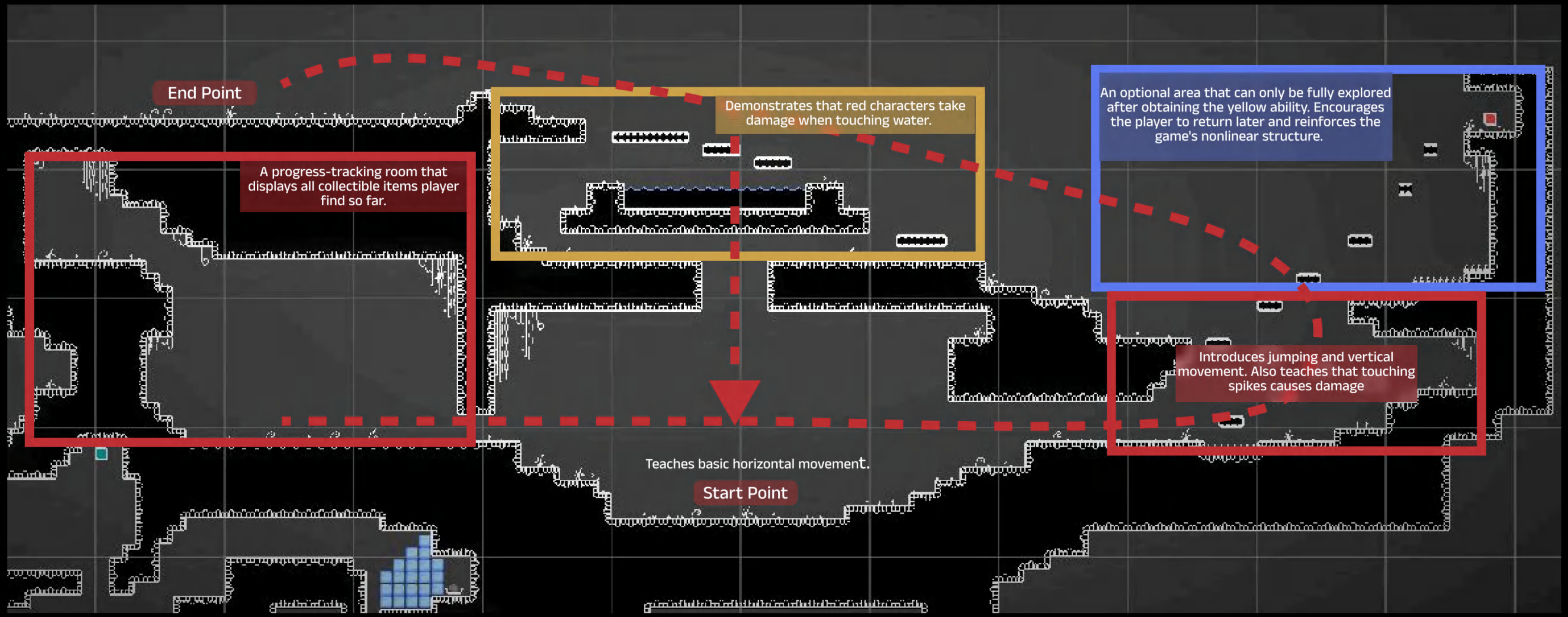
Condition Ids: 0



Level Design

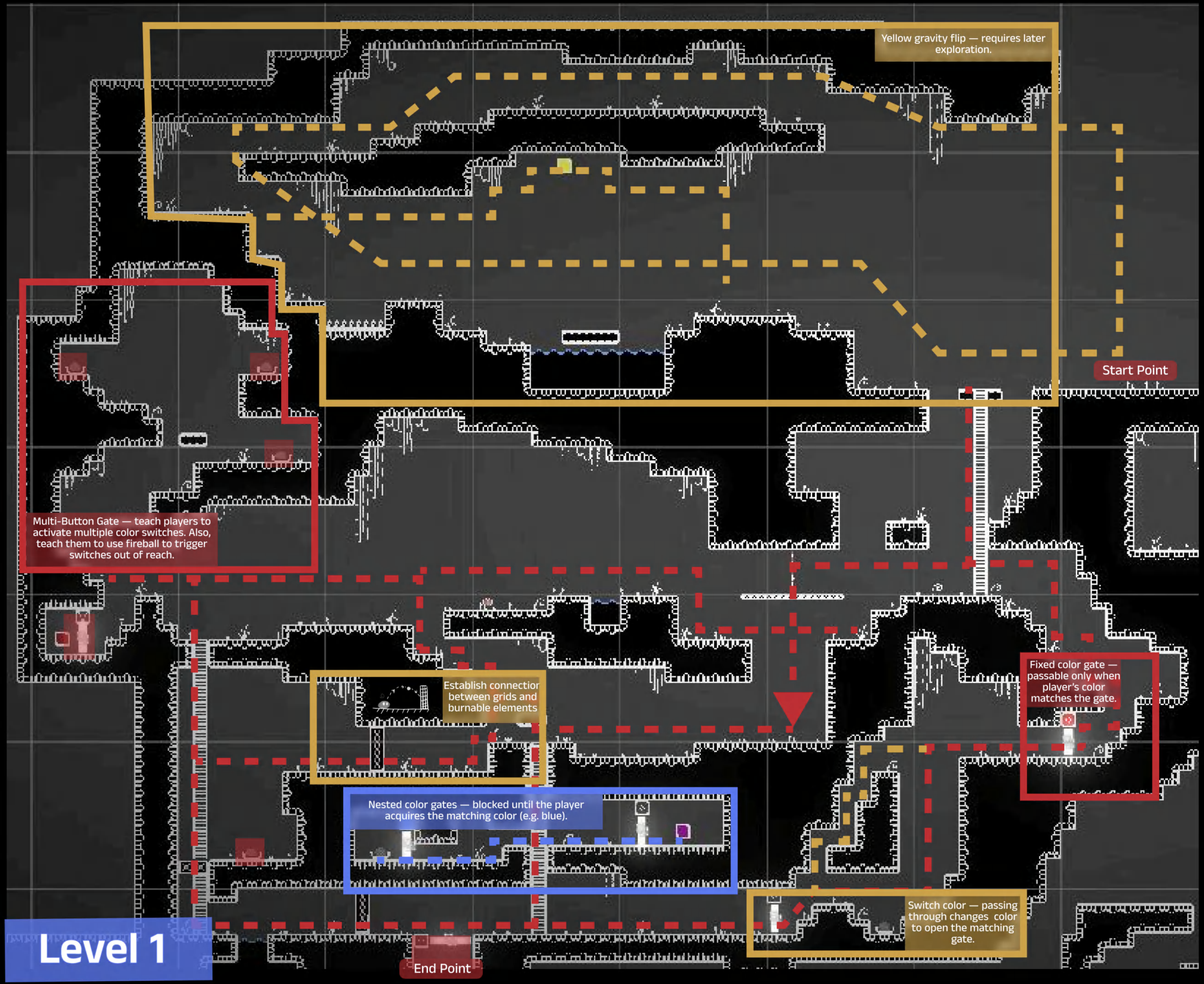
Tutorial

- Teaches basic movement and the coloring mechanic.
- Introduces the achievement system for early engagement. In the final map, a shortcut will lead back here, allowing players to revisit.


End Point
Demonstrates that red characters take damage when touching water.
An optional area that can only be fully explored after obtaining the yellow ability. Encourages the player to return later and reinforces the game's nonlinear structure.
A progress-tracking room that displays all collectible items player find so far.
Introduces jumping and vertical movement. Also teaches that touching spikes causes damage
Teaches basic horizontal movement.
Start Point

Level 1

- Offers multiple paths, with some areas currently inaccessible.
- Collectibles often require experimentation, or must be revisited with unlocked abilities.
- Regardless of the chosen route, before reaching Level 2 the player will learn:
  - How to use the fireball and its burning property
  - The basic mechanics of color gates


Yellow gravity flip — requires later exploration.
Start Point
Multi-Button Gate — teach players to activate multiple color switches. Also, teach them to use fireball to trigger switches out of reach.
Establish connection between grids and burnable elements
Fixed color gate — passable only when player's color matches the gate.
Nested color gates — blocked until the player acquires the matching color (e.g. blue).
Switch color — passing through changes color to open the matching gate.
Level 1
End Point

Level Design

Level 2

The pink and light blue routes form the main level. The pink section introduces basic blue knight mechanics, while the light blue section builds on them and requires more strategic thinking and collaboration with the red knight.

- Pink Stage:
  - Yellow: The player learns swimming.
  - Red:
    - Creating ice blocks as platforms;
    - Using blue dye to unlock color gate underwater.

- Light Blue Stage: Contain 3 side routes.
  - Blue: Melt ice with red;
  - Green: Use ice to press switches;
  - Purple: Use pipes and melt to trigger multiple switches

Start Point

End Point

Learn the blue ability: swimming, and become familiar with buoyancy in water. A more difficult platforming challenge is placed on the left side.

Learn to use ice blocks: stack them to reach previously inaccessible heights.

Learn the red ability: melt ice blocks by approaching them, and understand the synergy between red and blue.

Learn the mechanic of pressing switches simultaneously, using ice blocks to assist.

Advanced challenge: combine two color abilities to press three switches simultaneously.

Learn to use the property that ice melts into water to press multiple switches at once and open a pipeline.

Screenshot

Reflection and Plan

Add clearer visual links between doors and triggers to improve player understanding.

Add visual landmarks and a minimap to guide players, and adjust camera rules to improve navigation.

Resolve input conflicts by refining control logic and adding clearer UI prompts.

Complete unfinished systems such as yellow ability, secondary colors, achievements, UI, and saving.

Redesign color data handling with asynchronous reading to ensure stable and accurate updates.