A thorough overview of my game, “Legacy of Magic”, and how it was developed.

ArctynFox

Contents

English Version

Overview

This is a technical design document providing an explanation of the functional and design details of my game, “Legacy of Magic”, which I developed in the Unity engine. The goal is to explain all important functionality in detail such that anyone who has experience developing in Unity can modify or add content without much difficulty. It should also make clear my thoughts when designing the game and why I made the choices I made. Any time a C# script is referenced, the relevant code will be shown, however most comments for non-critical functionality are in Japanese only.

Game Synopsis

“Legacy of Magic” is a Touhou-inspired danmaku game using the same arcade-continue style in which you, an inquisitor of the mage association, must uncover the cause of a strange case of anachronism occurring within the kingdom. Of what nature is this disaster? Who or what may be causing it? Discover for yourself as you graze past the magic of various people and creatures in this 2.5d bullet hell!

Development Time

This game was developed for a Game Design university course in 2021 over the span of 10 weeks, where I spent roughly 6 hours every day outside of class researching common game design techniques, learned how to program in C# as well as how to develop and design in Unity, and programmed the game functionality as well as created some of the UI assets and game sprites.

Additionally, with the recent CVE vulnerability affecting Unity (https://www.cve.org/CVERecord?id=CVE-2025-59489) as the impetus, I recently updated the game, making many optimizations and cleaning up a large portion of the code.

Total Development Time: Roughly 500 hours.

Development Environment and Borrowed/Commissioned Assets

Being the sole developer and designer of the entire project, I used and learned a wide range of tools to develop the game, create graphics, and compose the music. The tools used and their purposes are as follows:

- Unity – To develop and integrate the different aspects of the game into the final product. I learned how to use this mostly on my own through the official Unity documentation.
- VSCode – To write the C# scripts used in the game.
- Gimp – To design the UI assets and sprites.

- MIDI – To compose the music. I do not have a compositional background and don't know music theory, so I learned this mainly through trial and error.

That being said, I only created some of the graphics and audio myself. Many of the bullet sprites were royalty free assets I found online, and all of the UI elements were edited together by myself using royalty assets, as well as NASA space photography for the background images. All character artworks and sprites, as well as the music used on stages 2, 3, and 4, were created by Maltolyte (https://www.youtube.com/@Maltolyte).

Requirements

- There must be bullets that can damage the player on collision.
- Enemies and players can fire bullets.
- Bullets should have various features that allow for unique movement patterns.
- Enemies should be recognizably different based on their attack pattern difficulties.
- The player should have a limited number of lives.
- The player should be able to spend a continue to reset their lives and continue playing should they run out.
- The player should have a limited resource that allows them to survive difficult situations.
- There should be multiple stages.
- The difficulty should increase per stage.
- There should be a boss at the end of each stage.
- The game should include a manual that explains the controls and the exposition for the game.
- The game should have a storyline.
- The story should be told to the player through dialogue with boss enemies.
- Stages and bosses should have music themes that fit their design or character.
- The game should have a consistent UI style that thematically represents the story.
- There should be a score counter that increases when enemies are defeated.
- The score value gained should differ based on the enemy type and stage.
- The game should be able to be paused and un-paused during gameplay.
- Pausing should show a menu that allows for returning to the title screen and restarting from the first stage.
- There should be UI that allows the player to see their current lives, spell cards, and score.

Design

This section covers the graphics, music, and game mechanics of the project, showing both the inspirations and the final product.

Graphics

As mentioned in the overview, this game was heavily inspired by ZUN's "Touhou Project", a long-standing series that is a subvariety of shoot-em-up game called a "bullet hell", or "danmaku". The specific games that inspired this project the most are "Touhou 6 Koumakyou – The Embodiment of Scarlet Devil" and "Touhou 7 Youyoumu – Perfect Cherry Blossom".

To create a cohesive magical feeling in the game that matches the events of the story, I incorporated many star and magic circle motifs throughout the UI. This is immediately apparent when you first open the game, with the title screen (Figure 1).

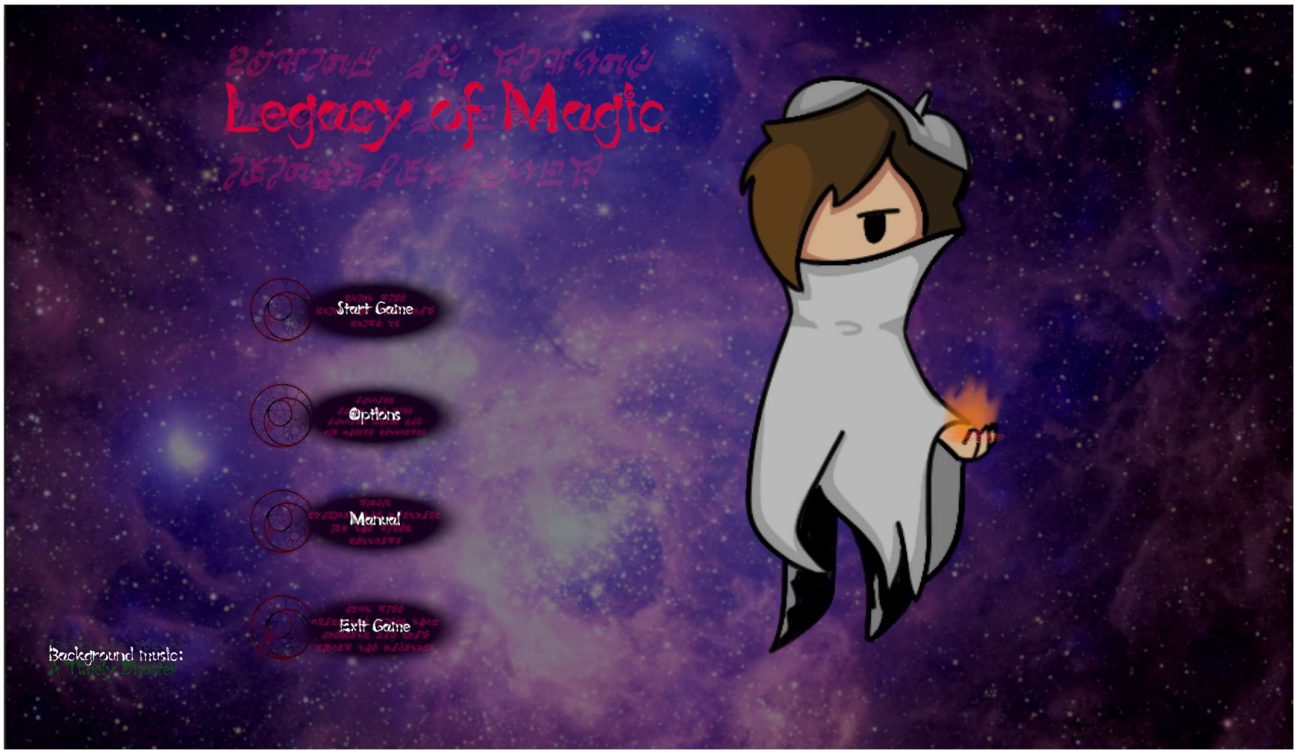

Figure 1: The title screen.

For the background image, keeping with the theme of magic and mystery, I used a photograph of the region NGC 604 in the galaxy Messier 33 (https://www.nasa.gov/image-article/ngc-604/), courtesy of NASA. It can be seen clearly in the background of the title screen (Figure 1).

Figure 2: The options menu, showing closeups of the button assets.

The UI buttons are also visible above (Figure 2). To portray the theme of magic, I made a relatively simple graphic with a magic circle to the left for use on the title screen and menus.

The game UI follows the same design principle, using a different cropping of the same background image as the title screen, but this time incorporating a pattern of magic circles to flavor it (Figure 3).

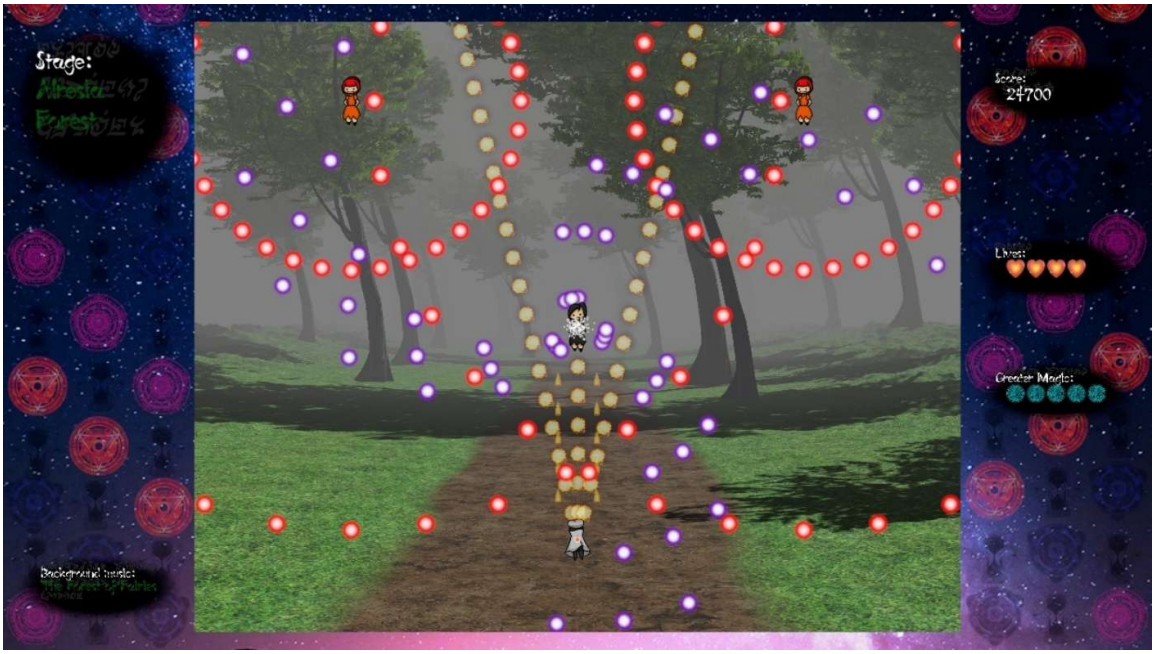

Figure 3: 2d gameplay with a 3d stage background.

This is intentionally to reflect a similar design choice from Touhou Project (Figure 4), where the UI is very simple visually, but has a repeating pattern to create subtlety that strongly contrasts the vibrantly colorful and hectic gameplay.

Figure 4: Screenshot of gameplay of Touhou 7.

Other visual similarities in my game include bullet designs (Figure 5) and the player's visible hitbox (Figure 6) shown below.

Figure 5: Example of a bullet that can be found in the game.

Figure 6: The player character with a visible hitbox.

I also chose fonts I believed to fit the setting of the story. The first of which is that I used a cartoonish, fantasy font (https://font.heartx.info/c86/). It is used for most of the visible text, including the title screen (Figure 1) and dialogue (Figure 7).

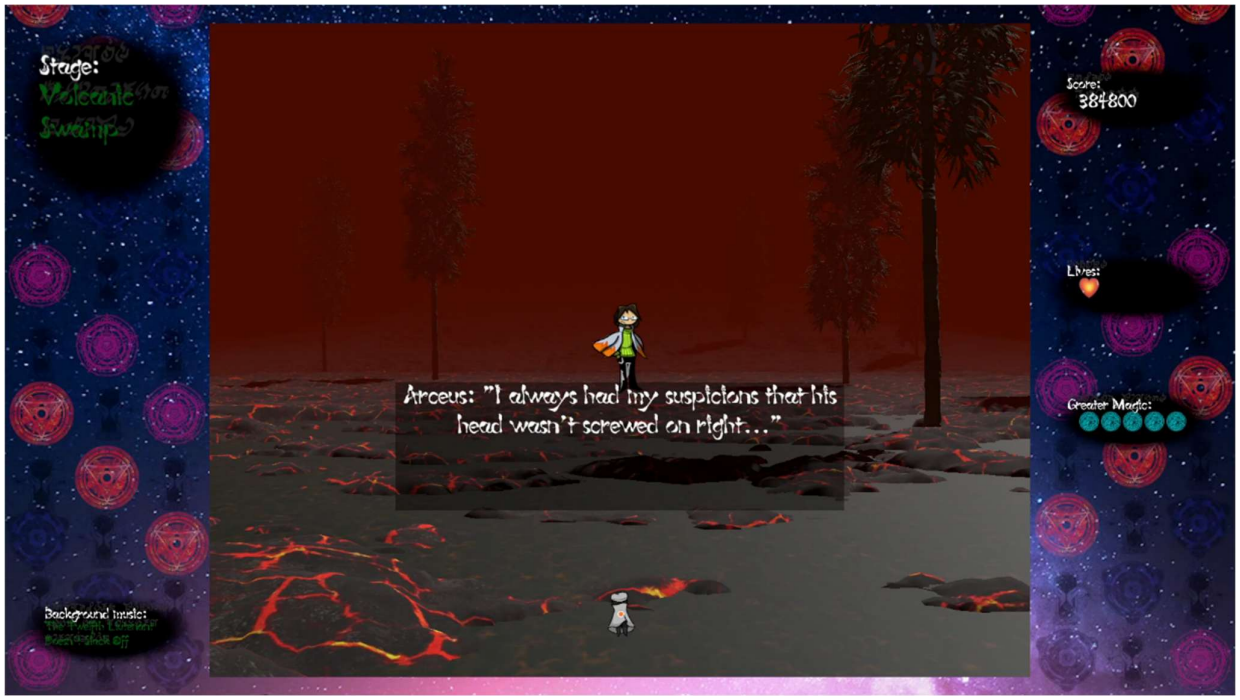

Figure 7: Example of dialogue and UI font.

For the second of those, in order to further enhance the feeling and tie the visuals into the story even more, I also added an ancient style font as a shadow to all UI text (Figure 8).
(https://www.deviantart.com/ozziescribbler/art/Ancient-Quill-FREE-FANTASY-SCI-FI-FONT-297679343)

Figure 8: Ancient style shadow text behind the stage title.

Another visual inspiration from Touhou Project is the 3d stage backgrounds with the 2d gameplay overlayed on top. For the 3d stages, I used small terrain assets with sparse details on them and obscured the distance with thick fog. This can be seen in my game as shown previously (Figure 3). The following is what the same stage looks like in full without the fog (Figure 9).

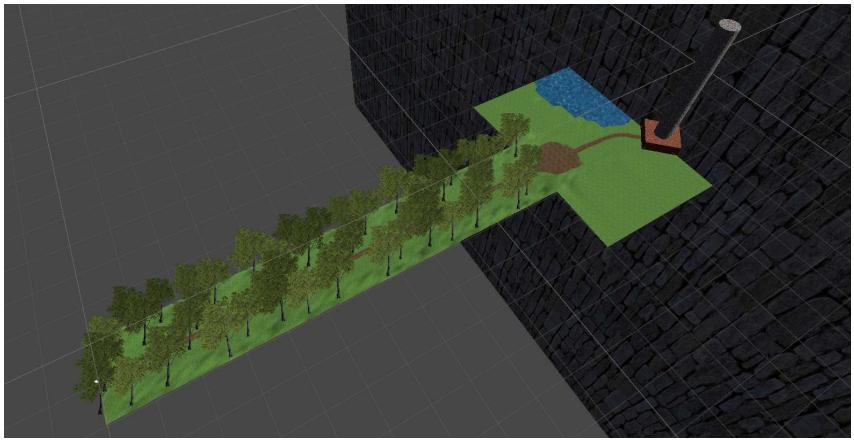

Figure 9: Stage 1 scene view without fog.

Despite how it looks, the stage is designed to loop indefinitely until the boss appears. The larger area on the right does not come into view until the stage's boss appears. I detail how I made this happen in the Game Mechanics section.

This design choice is also inspired by Touhou Project, with every game since Touhou 6 doing the same (Figure 10). I incorporated this as I thought that this was a very interesting and unique way of making the background of the gameplay interesting, as many other danmaku games at the time simply used tile assets to make a 2d scrolling background.

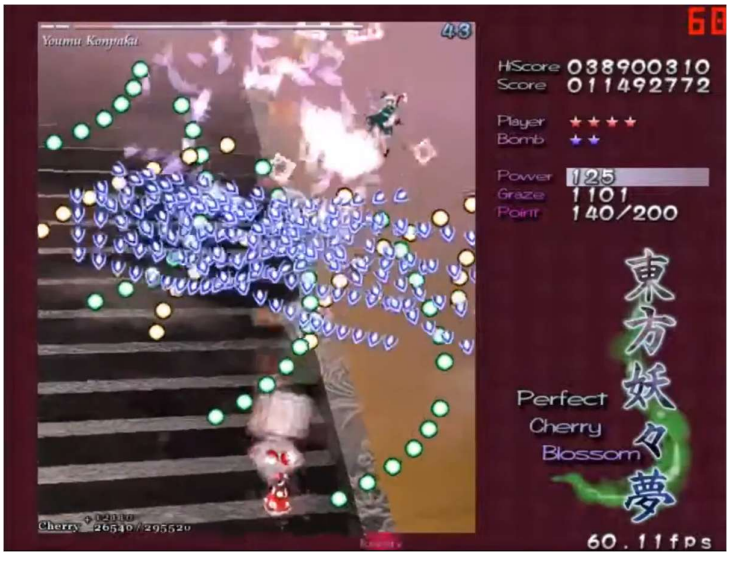

Figure 10: 3d stairs visible in the background of one of the stages in Touhou 7.

One of the strong points of Touhou Project is that every game contains unique attack patterns that come together to be a visual spectacle. I attempted to recreate this feeling of looking at a work of art when observing enemy attack patterns, with examples visible above (Figure 3) and below (Figure 11).

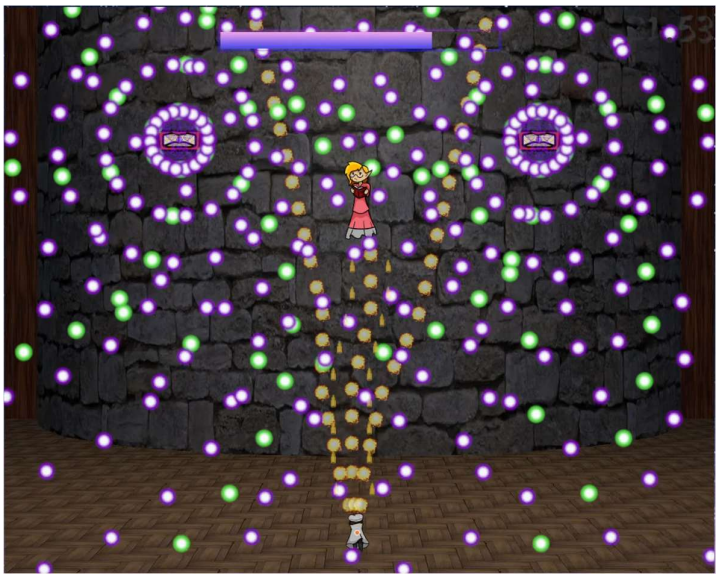

Figure 11: Example of an attack pattern of a boss.

I believe that all of this comes together to create a very cohesive feel of magic and mystery, a motif I tried to convey with every aspect of the game.

Music

The music in the final game was composed by both myself and Maltolyte, with frequent communication to make sure the tracks fit the feel of the game. The music that I composed myself was also inspired by the fantastical feel of ZUN's works. I made heavy use of power chords in most of the tracks I composed. I can't attach the music inside this document, but any time I reference a track from the game, I will say where it is used.

For the normal stage background music, I wanted it to always have a sort of rising feeling that builds up to the climax that is the boss. For these tracks, I wanted them to be reminiscent of the locale of the stage itself. For example, I gave the stage 1 theme a very fantasy-exploration-invoking feel using mainly harps and soft synthesizers because it takes place in a forest. For calmer tracks like this, I took some inspiration from older fantasy games and movies such as Nihon Falcom's "Trails in the Sky" and Studio Ghibli's "Castle in the Sky".

In a similar vein, I requested that Maltolyte, the other composer, to try to keep this rule as well, and, though I left the extent of my management of his composition to a minimum beyond that, I am happy with how the tracks he composed turned out for the places they are used.

As for the music used for boss fights, they were almost all composed with the intention of creating a rising action that climaxes with the boss' most powerful attack, falling back to a moment of calm, and then reaching a climax again at the end, bringing a close to the boss fight. This can be heard most strongly in the music that plays in the fight against the final boss on stage 5.

There are two tracks that deviate from these patterns, both being on stage 3. There is a story reason for this, as the dialogue on stage 3 both reveals the protagonist's name and provides the story's plot twist. For that reason, I asked Maltolyte to compose tracks that invoke suspense and try to give an air of confusion. Particularly, I thought it would be fitting for the stage 3 boss' music theme to reverse the motif of the title screen, and I am very happy with what he produced.

If I had more development time, I would have worked to try to make the quality disparity between my compositions and Maltolyte's compositions smaller, but without any music composition experience or tools, I worked with what I was able to and I think it worked to solidify the Touhou Project inspirations the game takes.

Game Mechanics

As the first game project I ever worked on, I knew I would have to keep the mechanics and content simple to complete it within the 10-week timeframe until the submission deadline. For this reason, I based my requirements around creating a gameplay experience like that of the Touhou Project. Most of the game mechanics are directly mirrored in my game. You can move, focus to move slower to avoid bullets in complex patterns, fire your own projectiles, and use "greater magic", which clears all enemy bullets on the screen.

Difficulty

Originally, I wanted to provide difficulty levels, but knew it would be difficult to do so within the time limit, so I compromised and instead pulled a mechanic from the early Windows Touhou games where you can set the amount of lives and spell cards you start with, within reason (figure 2). Just like Touhou Project games, your spell cards reset to the maximum when you lose a life, and if you run out of lives, you can use one of a limited number of continues to continue playing, at the expense of resetting your score.

Additionally, enemies' attack patterns become more challenging as you reach later stages, providing a natural increase in challenge as you near the end of the game. One example of this can be seen with bosses in the early (Figure 12) and late (Figure 13) stages.

Figure 12: Simpler attack pattern of an earlier boss.

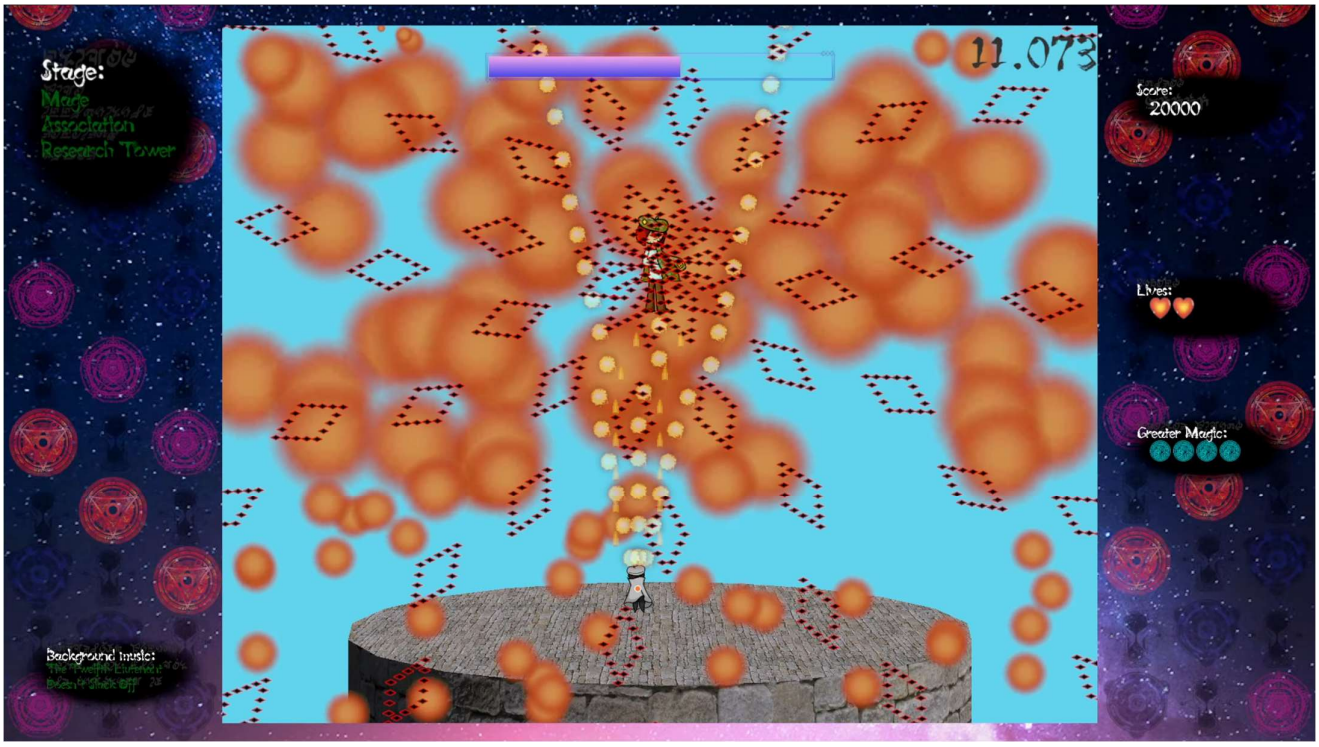

Figure 13: More difficult attack of a late stage boss.

Regarding the actual balancing, since there are no difficulty options, I sampled how well I did in various Touhou games and designed the difficulty in my game around my own skill level, which, at the time, was being able to clear a Touhou game on normal difficulty if I used all my continues. The intention behind this was that anyone who is more skilled than I am at this type of game would be able to clear without needing to use any continues, which would make the game show the good ending. On the other hand, if the player uses one or more of the five allotted continues, they will instead be shown the bad ending.

User Input

The user input is handled using an interrupt-based system, so there is nearly no effect on performance. The input method is the keyboard. The controls for the game are as follows:

- Z – Fire bullets/Select.
- X – Use greater magic (spell card).
- Shift – Focus (slow movements).
- Arrow Keys – Movement/UI navigation.
- Escape – Show the pause menu.

This intentionally mimics the controls of the Touhou games so that players don't need to learn a new control scheme.

Managing the functionality of user input is simple, as it can be accessed from anywhere using a singleton design pattern:

https://github.com/ArctynFox/Legacy-of-Magic/blob/main/Assets/Scripts/UserInput/InputManager.cs

```
public class InputManager : MonoBehaviour
{
    public static InputActions inputActions;

    void Awake()
    {
        if (inputActions == null)
        {
            inputActions = new InputActions();
        }
    }
}
```

I can then define and enable an action like in the following section of the PlayerController script:

https://github.com/ArctynFox/Legacy-of-Magic/blob/main/Assets/Scripts/Player/PlayerController.cs

```
public class PlayerController : MonoBehaviour
{
    InputAction bomb;

    ...

    //enable the use of bombs
    void EnableBombAction()
    {
        bomb.performed += OnBombPressed;
        bomb.Enable();
    }

    //disable the use of bombs
    void DisableBombAction()
    {
        bomb.performed -= OnBombPressed;
        bomb.Disable();
    }

    //called whenever the bomb action is pressed
    void OnBombPressed(InputAction.CallbackContext callbackContext)
    {
        //check whether the player has bombs remaining
        if(Parameters.singleton.bombs >= 0)
        {
            //use a bomb
            Instantiate(spell, transform);
            Parameters.singleton.bombs--;
            Parameters.singleton.updateBombDisplay();
            //make the player temporarily invulnerable
            StartCoroutine(BombInvulnerability());
        }
    }

    ...

}
```

Spawners

Since spawning bullets frequently and in complex patterns is one of the key points of a danmaku shooting game, I wanted to make an endlessly reusable spawner for bullets that would fire them automatically with given settings. I decided that the best way to do this would be using a factory design pattern, where there is one script to spawn and manage elements of many bullets at once. That BulletSpawner script can be found at the following link:

https://github.com/ArctynFox/Legacy-of-Magic/blob/main/Assets/Scripts/Spawner/BulletSpawner.cs

The relevant variables and a comment for each with their purpose can be found in the code above (in English and in Japanese, however GitHub has some issues with encoding and does not display the

Japanese correctly, so you may need to download the source code to view the comments properly), so I will only explain the math happening here.

The game bases its bullet firing timing off of Unity's built-in fixedDeltaTime and does relevant movement calculations in FixedUpdate so that spawn rates, movement, and other functionality can be based off real time. This also means their speed can be controlled with Time.timeScale, which allows for pausing the game by changing a single variable between 0 and 1, and could be used to allow game features that speed up or slow down time if the game were expanded on in the future.

The game calculates where and in what direction to shoot bullets from in Unity's Update function with relatively simple trigonometry and algebra:

```
void Update()
{
    if (shootAtPlayer)
    {
        AimAtPlayer();
    }
    if(SpawnDetailsHaveChanged())
    {
        CalculateSpawnDetails();
        //update the values to check for updates against with the new values
        dSPS = spawnsPerSecond;
        dCo = arcCount;
        dD = arcDegrees;
        dCe = arcCenter;
    }
}

//if firing settings have changed, recalculate directions and
//spawnLocations arrays
bool SpawnDetailsHaveChanged()
{
    return dSPS != spawnsPerSecond || dCo != arcCount || dD != arcDegrees || dCe != arcCenter;
}

//recalculate all of the parameters involved in spawning bullets based on the current settings
void CalculateSpawnDetails()
{
    //calculate the angle of the central-most line of bullets
    int center = (int)Mathf.Floor((arcCount - 1f) / 2);
    //calculate the angle of offset if the number of lines is even
    centerPLC = arcCenter;
    if (arcCount % 2 == 0)
    {
        centerPLC -= arcDegrees / arcCount / 2;
    }

    //recalculate the directions array
    CalculateDirections(center);

    //recalculate the spawnLocations array
    CalculateSpawnLocations();

    //recalculate the number of frames between each round of bullets
```

```
    framesPerSpawn = (int)(Mathf.Floor((int)(1 / Time.fixedDeltaTime) / spawnsPerSecond));
}

//calculate the directions for each line of bullets to fire
void CalculateDirections(int center)
{
    directions = new float[arcCount];

    for(int i = 0; i < directions.Length; i++)
    {
        //based around the direction given by center, calculate the directions of all of the firing lines
        //using the total angle from the counterclockwise-most line to the clockwise-most line and the
        //number of lines total to fire. This gives the angle direction of each firing line.
        directions[i] = centerPLC + (arcDegrees / arcCount * ((float)i - center - (((arcCount % 2) - 1) /
2)));
    }
}

//calculate the spawn positions for each line of bullets to fire
void CalculateSpawnLocations()
{
    spawnLocations = new Vector3[arcCount];

    for(int i = 0; i < spawnLocations.Length; i++)
    {
        //the directions are stored as floats, so we need to calculate a direction vector from that
        spawnLocations[i].x = Mathf.Cos(directions[i] / 180 * Mathf.PI);
        spawnLocations[i].y = Mathf.Sin(directions[i] / 180 * Mathf.PI);
        //and then we multiply the vector by the spawn radius to find where the bullets need to be spawned
        spawnLocations[i] = spawnLocations[i] * bulletSpawnRadius;
    }
}
```

The spawn locations and directions are then used to spawn and position each bullet when firing:

```
void FixedUpdate()
{

    //fire a round of bullets once the cooldown is up
    if (framesSinceLastSpawn >= framesPerSpawn)
    {
        FireBulletRound();
    }
    //if bullets were not spawned on this frame, increment the frame count
    else framesSinceLastSpawn++;
}

...

//fire a round of bullets using the current settings
void FireBulletRound()
{
    int spawnLocationIndex = 0;
    //for every bullet line
    foreach (Vector3 _ in spawnLocations)
    {
```

```
        //for the number of bullets in the line (lineCount)
        for (int linePositionIndex = 0; linePositionIndex < lineCount; linePositionIndex++)
        {
            //spawn a bullet
            SpawnBullet(spawnLocationIndex, linePositionIndex);
        }
        spawnLocationIndex++;
    }
    //play the firing sound
    PlayShootSound();
    //reset the firing cooldown
    framesSinceLastSpawn = 0;
}

//spawn one bullet and set its location, direction and speed given its spawn position and line position
void SpawnBullet(int spawnPositionIndex, int linePositionIndex)
{
    //if BulletPool exists, add the bullet as it's child object to prevent cluttering the scene hierarchy in
editor
    GameObject bulletPoolGameObject = GameObject.Find("BulletPool");
    Transform bulletPoolTransform = null;
    if(bulletPoolGameObject != null)
    {
        bulletPoolTransform = bulletPoolGameObject.transform;
    }

    //spawn the bullet
    GameObject bullet = Instantiate(bulletType, spawnLocations[spawnPositionIndex] + transform.position,
Quaternion.Euler(0, 0, directions[spawnPositionIndex]), bulletPoolTransform);

    //set the spawned bullet's direction and movement speed
    if (bullet.TryGetComponent<MoveDanmaku>(out var mD))
    {
        mD.moveSpeed = moveSpeed + (deltaSpeed * linePositionIndex);
        mD.direction = spawnLocations[spawnPositionIndex].normalized;
    }

    //set the orbit settings for the bullet
    if (bullet.TryGetComponent<AngleOrbit>(out var aO))
    {
        aO.firedFrom = gameObject;
        aO.offsetAngle = orbitAngle;
        aO.isOrbit = isOrbit;
        aO.stopTime = orbitStopTime;
    }
}
```

Writing the spawning functionality in this way such that the settings can be updated over time allows for decorating the spawner with other scripts that modify them over time, allowing for the creation of interesting patterns. For example, here is a simple use case where I just rotate the firing direction over time:

https://github.com/ArctynFox/Legacy-of-Magic/blob/main/Assets/Scripts/Spawner/DirectionRotator.cs

```
[RequireComponent(typeof(BulletSpawner))]
public class DirectionRotator : MonoBehaviour
{

    //change in angle per second
    public float degreesPerSecond = 15f;
    //the BulletSpawner scripts to affect
    public BulletSpawner[] instantiators;
    //whether the rotation speed should change over time
    public bool angleIncreasing = false;
    //change in rotation speed per second
    public float deltaAngle = 15f;
    //degree rotation per FixedUpdate call
    float degreesPerFrame;

    void Start()
    {
        degreesPerFrame = degreesPerSecond / (int)(1 / Time.fixedDeltaTime);
    }

    void FixedUpdate()
    {
        if (angleIncreasing)
        {
            degreesPerFrame += deltaAngle * Time.fixedDeltaTime;
        }
        foreach(BulletSpawner current in instantiators)
        {
            //rotate the firing direction
            current.arcCenter += degreesPerFrame;
        }
    }
}
```

I then add them as components of the same object and refer the instantiators array of the DirectionRotator instance to the BulletSpawner instance and set them both up. I have attached an example of how this looks in the editor, as taken from the final phase prefab for the stage 1 boss (Figure 14), as well as a screenshot of the resulting pattern itself (Figure 15).

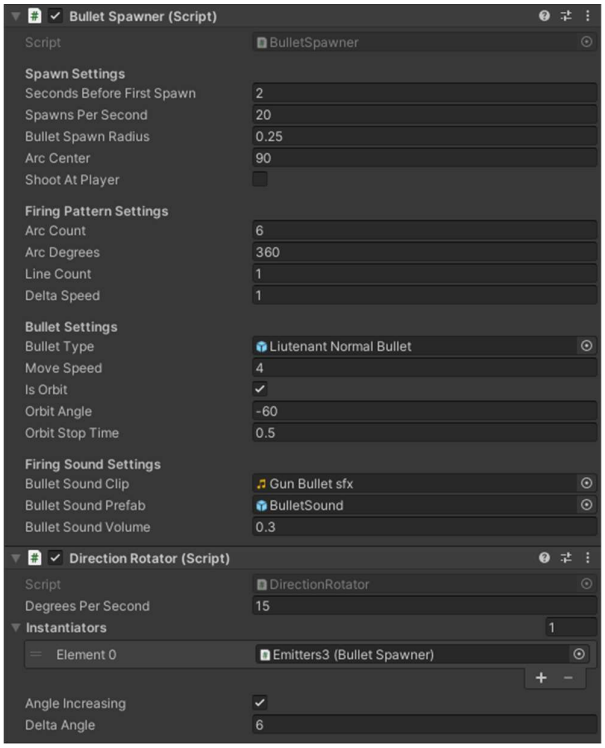

Figure 14: Example of BulletSpawner and a decorator being used to modify it.

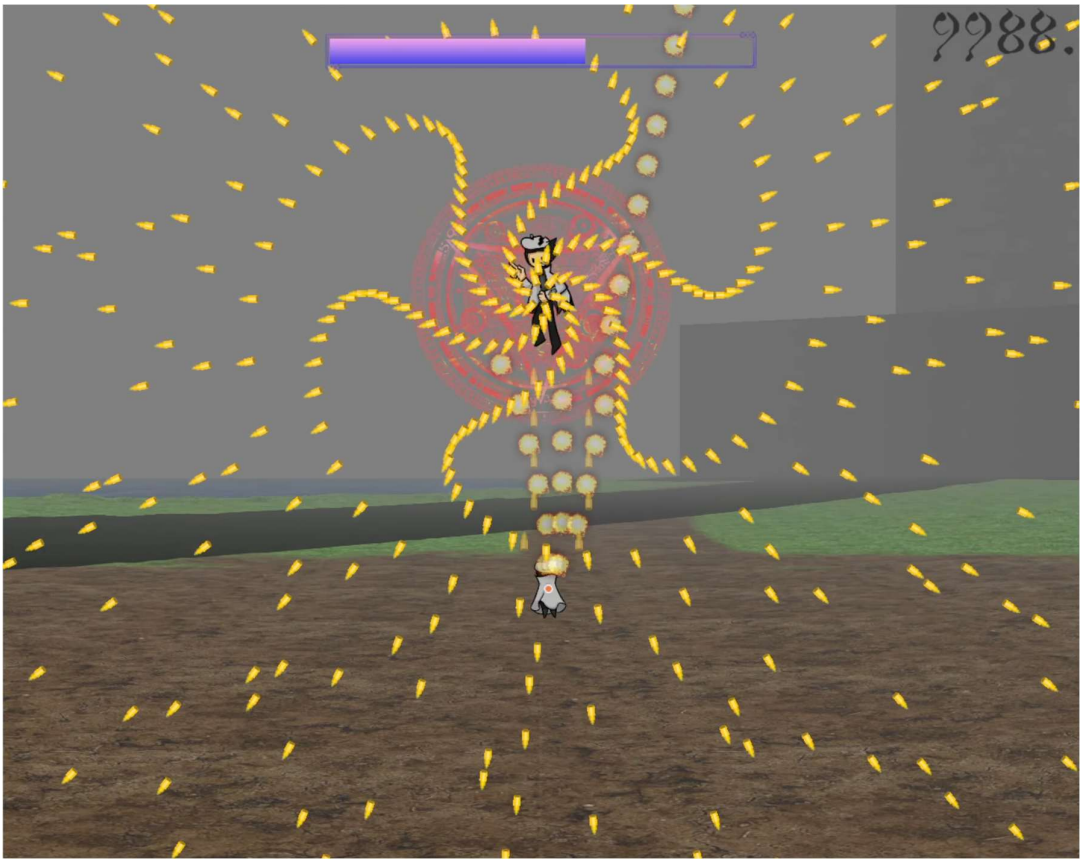

Figure 15: Bullet pattern for the final phase of the stage 1 boss.

Bullets

Bullets themselves also follow a decorator design pattern to allow for modular functionality. A standard bullet has only the MoveDanmaku script, which is, on its own, very simple, as it just moves the bullet in a given direction at a given speed:

https://github.com/ArctynFox/Legacy-of-Magic/blob/main/Assets/Scripts/Bullet%20Controllers/MoveDanmaku.cs

```
public class MoveDanmaku : MonoBehaviour
{
    public float moveSpeed = 1;
    public Vector3 direction = new Vector3(0,0);

    void FixedUpdate()
    {
        //move in the specified direction at the specified speed
        transform.position += moveSpeed * Time.fixedDeltaTime * direction;
        //rotate the bullet's sprite to match its movement direction
        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        transform.rotation = Quaternion.Euler(new Vector3(0, 0, angle));
    }
}
```

It can then be decorated with additional functionality to provide more complex movement patterns, such as the following script which causes bullets to slow down and stop at a specified time and then turn and move toward the player's current location:

https://github.com/ArctynFox/Legacy-of-Magic/blob/main/Assets/Scripts/Bullet%20Controllers/HomingBullet.cs

```
[RequireComponent(typeof(MoveDanmaku))]
public class HomingBullet : MonoBehaviour
{

    //time it will take to slow to a stop and then move toward the player
    public float timeToStop = 3f;

    //base MoveDanmaku script to decorate
    public MoveDanmaku moveScript;
    //speed change per frame vector
    Vector3 deltaDirection;
    //direction vector from bullet location to player
    Vector3 directionTowardPlayer;
    //whether the bullet has homed toward the player's location
    bool hasHomed = false;

    void Start()
    {
        deltaDirection = moveScript.direction / (timeToStop * (int)(1 / Time.fixedDeltaTime));
    }

    void FixedUpdate()
    {
        //if the bullet has already homed in, don't do anything
        if (hasHomed)
        {
```

```
            return;
        }

        //if the bullet has not yet come to a complete stop
        if(moveScript.direction != new Vector3())
        {
            //slow the bullet's movements
            moveScript.direction -= deltaDirection;
        }
        else
        {
            //turn toward the player's location
            directionTowardPlayer = (PlayerController.singleton.transform.position -
transform.position).normalized;
            moveScript.direction = directionTowardPlayer;
            hasHomed = true;
        }
    }
}
```

Game State Management

The game wouldn't be engaging without some sort of progression as well as lose conditions. Therefore, I implemented the previously mentioned life and continues systems as well as multiple stages to progress through. For replay value, I also added a score that gets added to on defeating enemies. With this, however, comes the necessity to not only store this data but allow it to persist between scenes as well as make it accessible by whatever needs it at any point in time.

For this purpose, I devised the Parameters script, which handles all functionality related to the game state and is accessible via a singleton pattern. The full source code file can be found here:

https://github.com/ArctynFox/Legacy-of-Magic/blob/main/Assets/Scripts/Parameters/Parameters.cs

The first thing this script handles, as mentioned, is lives. Here is the relevant code:

```
public class Parameters : MonoBehaviour
{
    //life-related values and graphics
    [Header("Life Data")]
    [Tooltip("Number of lives to spawn with per continue.")]
    public int lifeSetting = 3;
    [Tooltip("Current number of lives.")]
    public int lives = 3;
    [Tooltip("Life UI object reference.")]
    GameObject[] hearts = new GameObject[5];

    ...

    //set the hearts variable to the passed array
    public void setLifeGraphicArray(GameObject[] newArray)
    {
        hearts = newArray;
    }
```

```
    //whether there are no lives remaining
    bool outOfLives()
    {
        return lives < 0 && stageID > 0 && stageID < 6;
    }

    //update the graphics indicating the remaining lives and bombs
    public void updateHeartAndBombDisplay()
    {
        if (hearts[0] != null)
        {
            for (int i = 0; i < hearts.Length; i++)
            {
                if (i <= lives)
                {
                    hearts[i].SetActive(true);
                }
                else
                {
                    hearts[i].SetActive(false);
                }
            }
        }
        updateBombDisplay();
    }

    //this is called whenever the player takes damage
    public void playerTookDamage()
    {
        lives--;

        //if lives have run out, display the continue or game over screen based on the number of remaining
continues
        if(outOfLives() && menusAreNotVisible())
        {
            EventSystem.current.SetSelectedGameObject(null);
            Time.timeScale = 0;
            if (continues > 0)
            {
                displayContinueScreen();
            }
            else
            {
                displayGameOverScreen();
            }
        }
       updateHeartAndBombDisplay();
    }

    ...

}
```

Greater magic (sometimes referred to in the code as bombs or spell cards) function very similarly, so I will not show the code here.

The Parameters script also manages the boss battle game state:

```
//enables boss state and starts the boss' pre-fight dialogue
public GameObject instantiatePreBossDialogue(GameObject preDialogue)
{
    if (!isBoss)
```

```
    {
        isBoss = true;
        //prevent the player from attacking or using bombs while the boss dialogue is active
        PlayerController.singleton.DisableDestructiveInputActions();
        bossDialoguePre = Instantiate(preDialogue, canvas.transform);
        return bossDialoguePre;
    }
    else
    {
        return GameObject.Find("Boss Dialogue Pre(Clone)");
    }
}
```

The final main function of the Parameters script is to manage stage loading:

```
//check the stage variable for which scene to transition to and initiate the transition
public void NextStage()
{
    //if already loading, do nothing
    if(stageSwitchInitiated) { return; }
    stageSwitchInitiated = true;

    switch (stageID)
    {
        case 5:
            sceneName = "Ending";
            stageName = "End";
            break;
       case 4:
            sceneName = "Stage 5";
            stageName = "Mage Association Research Tower";
            break;
        case 3:
            sceneName = "Stage 4";
            stageName = "Grand Magus Archives";
            break;
        case 2:
            sceneName = "Stage 3";
            stageName = "Volcanic Swamp";
            break;
        case 1:
            sceneName = "Stage 2";
            stageName = "Lake of Spirits";
            break;
        case 0:
            sceneName = "Stage 1";
            stageName = "Alresia Forest";
            break;
       default:
            sceneName = "Title Screen";
            stageName = "Title Screen";
            stageID = -1;
            break;
    }
    StartCoroutine(StageSwitch());
}
```

```
//load the next scene asynchronously
IEnumerator StageSwitch()
{
    //pre-scene change setup-------------------------------
    //fade the screen to black
    float alpha = 0;
    while(alpha <= 1)
    {
        alpha += .02f;
        fadeTransition.color = new Color(0, 0, 0, alpha);
        yield return null;
    }

    //disable boss state
    isBoss = false;
    //pre-scene change setup END---------------------------

    //load the scene and increment the stage counter
    SceneManager.LoadScene(sceneName);
    stageID++;
    stageSwitchInitiated = false;

    //post-scenee change setup-----------------------------
    yield return new WaitForSeconds(.25f);
    //if the current scene is a gameplay scene
    if (stageID > 0 && stageID < 6)
    {
        //update UI elements for current gameplay info
        GameObject stageNameDisp = GameObject.Find("Stage Indicator YYK");
        stageNameDisp.GetComponent<Text>().text = stageName;
        stageNameDisp.transform.parent.gameObject.GetComponent<Text>().text = stageName;
        updateHeartAndBombDisplay();
    }

    fadeTransition = GameObject.Find("FadeTransition").GetComponent<Image>();

    //fade the screen in from black
    while (alpha >= 0)
    {
        alpha -= .02f;
        fadeTransition.color = new Color(0, 0, 0, alpha);
        yield return null;
    }
    //post-scene change setup END--------------------------
}
```

One thing I would change if I were to expand or reuse code from this project is to swap this scene changing method out for one that uses asset references to the scenes, ordered in an actual array. This would increase scalability and make it easier to keep track of stage order without having to read this code.

Story/Level Design