
# Legacy of Magic — Technical Design Documentation

# 『魔法の遺産』 — 技術設計書

ArctynFox

Source code and game download ／ ソースコードやダウンロード：
https://github.com/ArctynFox/Legacy-of-Magic

Last updated: 2025/11/27

Unity Version: 2020.3.49f1

# Table of 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

Total Development Time: Roughly 500 hours.

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 updated the game in late 2025, making many optimizations and cleaning up a large portion of the code.

## 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, given that this was a class project focusing on developing a functioning game, and I did not have any intention of selling the game commercially, I wanted to focus on the fundamentals of development and fleshing out the underlying systems that support the type of game I was trying to

make, rather than learn the graphical or audio technical skills to fully round out those aspects of the game. 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).

Everything I did not make myself or commission for this project is used under United States (in which the game was developed and presented) fair use clauses, on the grounds of being for educational purposes.

## 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, greater magic uses, and score.

## Design

This section covers the graphics, music, and game mechanics of the project, showing various aspects of the final product and what inspired them.

## 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).

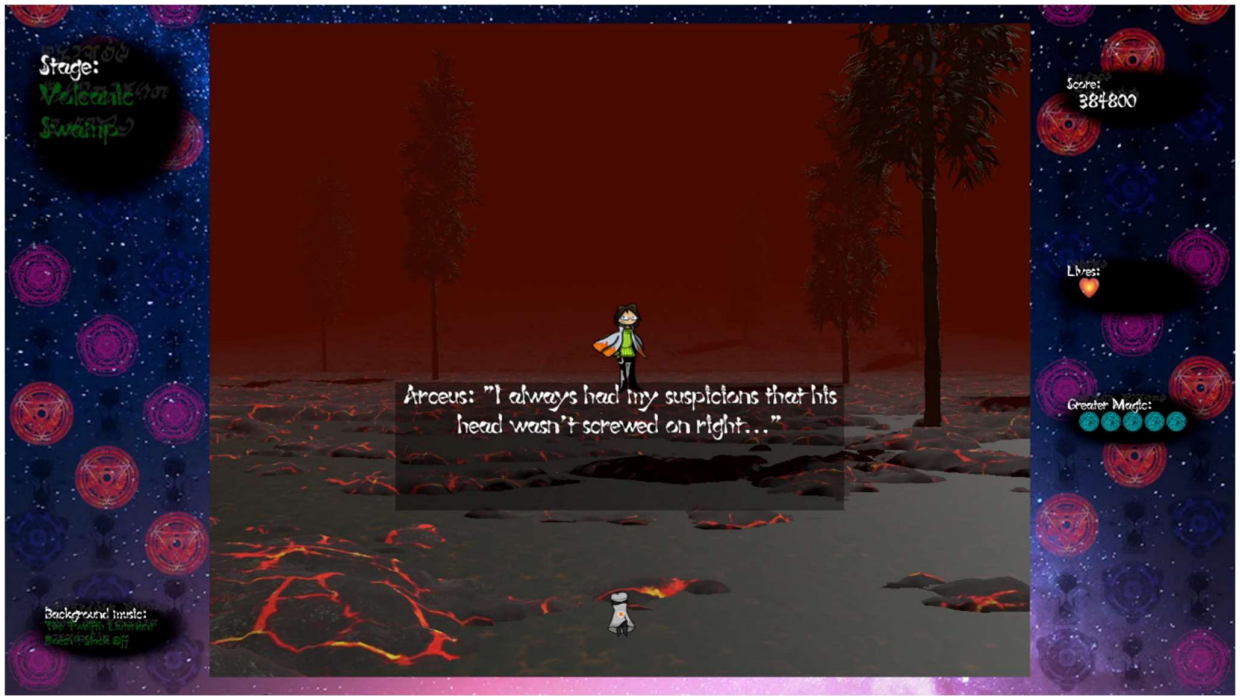


*Figure 7: Example of dialogue and UI font.*

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).

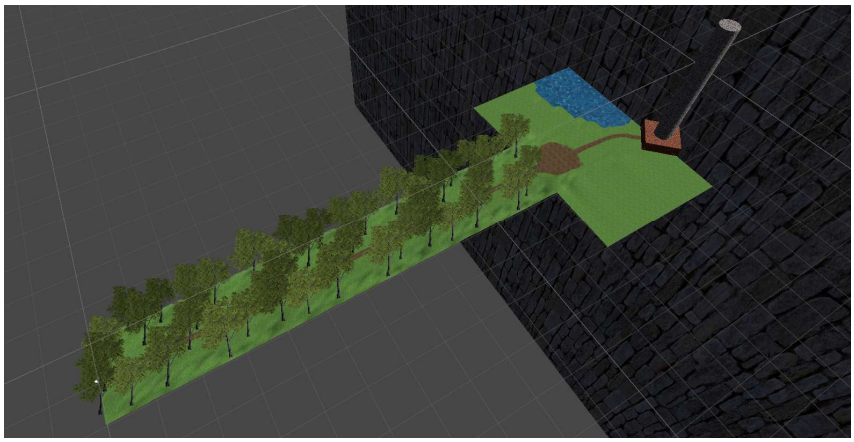

*Figure 9: Stage 1 scene view without fog.*

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.

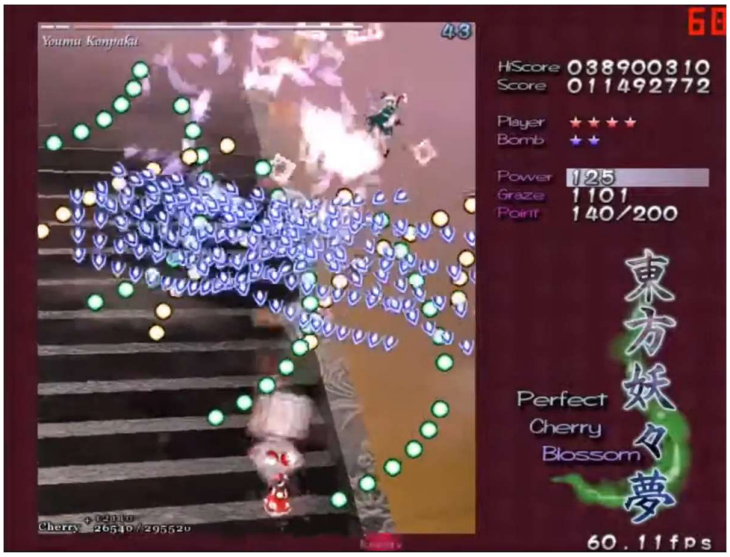


*Figure 10: 3d stairs visible in the background of one of the stages in Touhou 7.*

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).

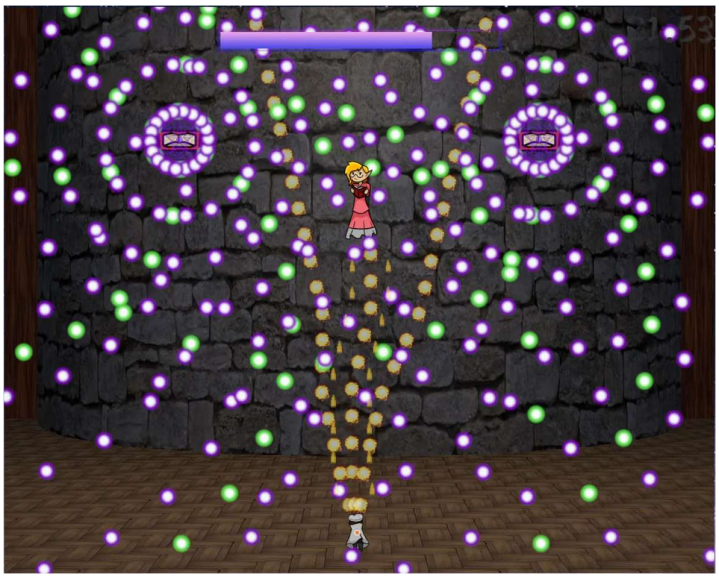

*Figure 11: Example of an attack pattern of a boss.*

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.

When I started this project, I was not knowledgeable on music composition, music theory, or the tools involved in digital music composition. This meant that, in order to include effective music in the game, I needed to find an alternative or acquire these skills myself rapidly. I spent a lot of time toward the end of the 10 weeks researching these topics myself as well as consulting with composers I know about important points like how to design the flow of the piece. By doing so, I was able to learn a little about music composition and challenge myself to gaining an entirely new skill through this project.

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, the tracks effectively match the requested tones and are used suitably given the story, setting, and other factors.

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 to build these emotions, and so I specifically requested such.

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. However, despite any prior music composition experience or tools, I rose to the task and gained the skills required to compose simple music that solidified 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 greater magic you start with, within reason (figure 2). Just like Touhou Project games, your greater magic uses 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 the 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.
- 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 that handles the use of greater magic:

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

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

    ...

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

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

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

    ...
}
```

When I updated the game, I also added controller support provisionally, but I have yet to thoroughly test it throughout the whole game, so I have not included information about it in the game manual yet. When I am confident that it works without issues just like the keyboard input method, I will add it to this document and the in-game manual.

### *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, so I will only explain the math happening here. GitHub has some issues with encoding and may not display the Japanese comments correctly, so you may need to download the source code to view the comments properly.

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;
    }
}
```

I was worried about the performance of this function, as it would need to recalculate a large number of vectors in cases where bullet count per round fired is very high, so I created copies of the modifiable settings (the dSPS, dCo, dD, and dCe variables) to store the last known values of their respective settings (see the Update method above to see the respective setting variables). This makes it possible to only run performance-expensive vector calculations once every time the settings change, instead of once every frame.

Similarly, any time there is vector arithmetic in the code, I make sure the vectors always come last in the order of operations when optimal to reduce the number of arithmetic calls in the compiled code. For example, any time a Vector3 is used in an expression that contains two or more primitive values, ordering the Vector3 such that it is used last in order of operations means performing the arithmetic on each value in the Vector3 once, instead of multiple times if it came first.

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).

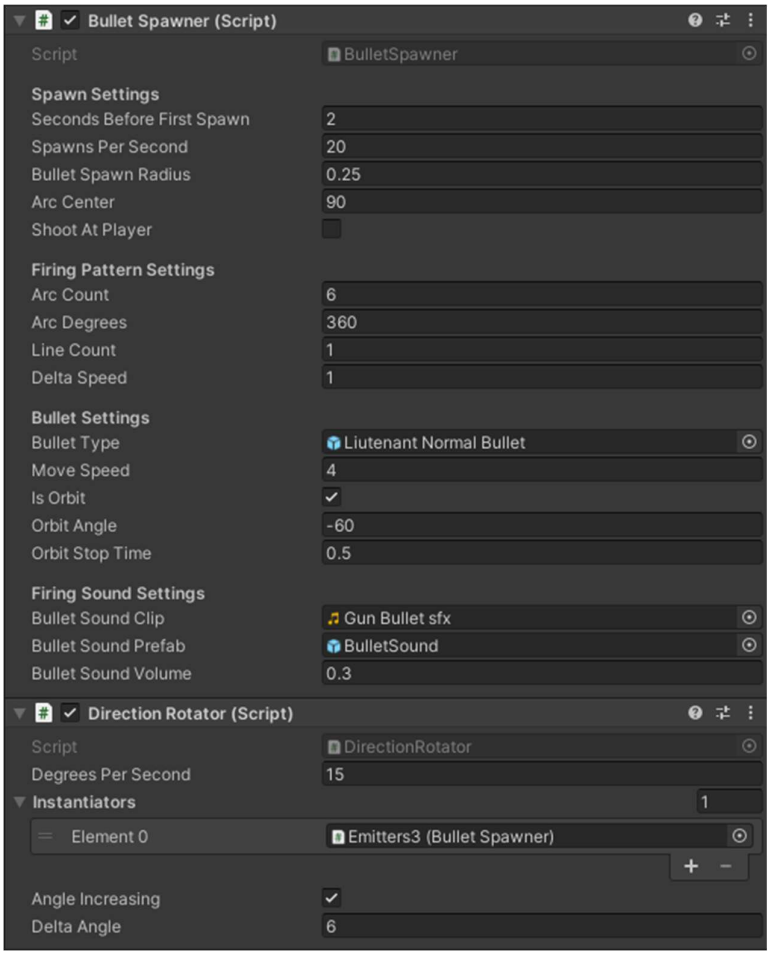


*Figure 14: Example of BulletSpawner and a decorator being used to modify it.*

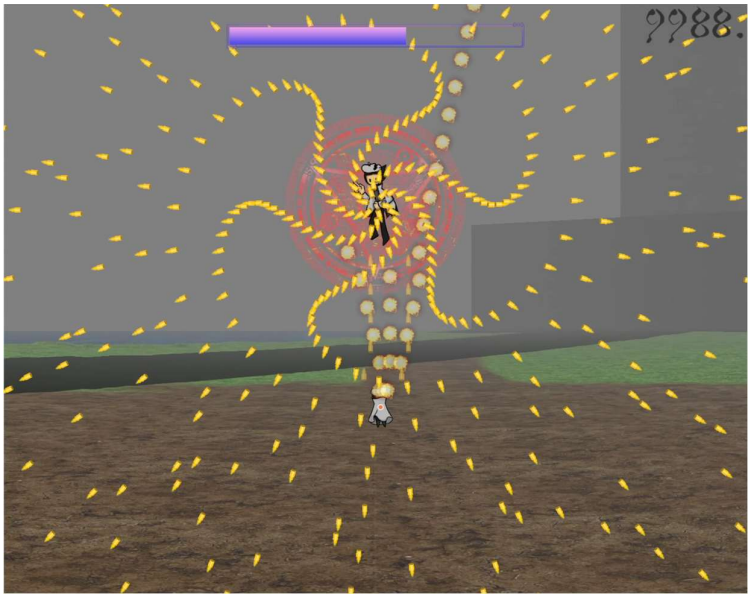

*Figure 15: Bullet pattern for the final phase of the stage 1 boss.*

### *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;
        }
    }
}
```

Combining spawning functionality and bullet functionality together, Figure 16 is a UML class diagram showing a typical enemy.

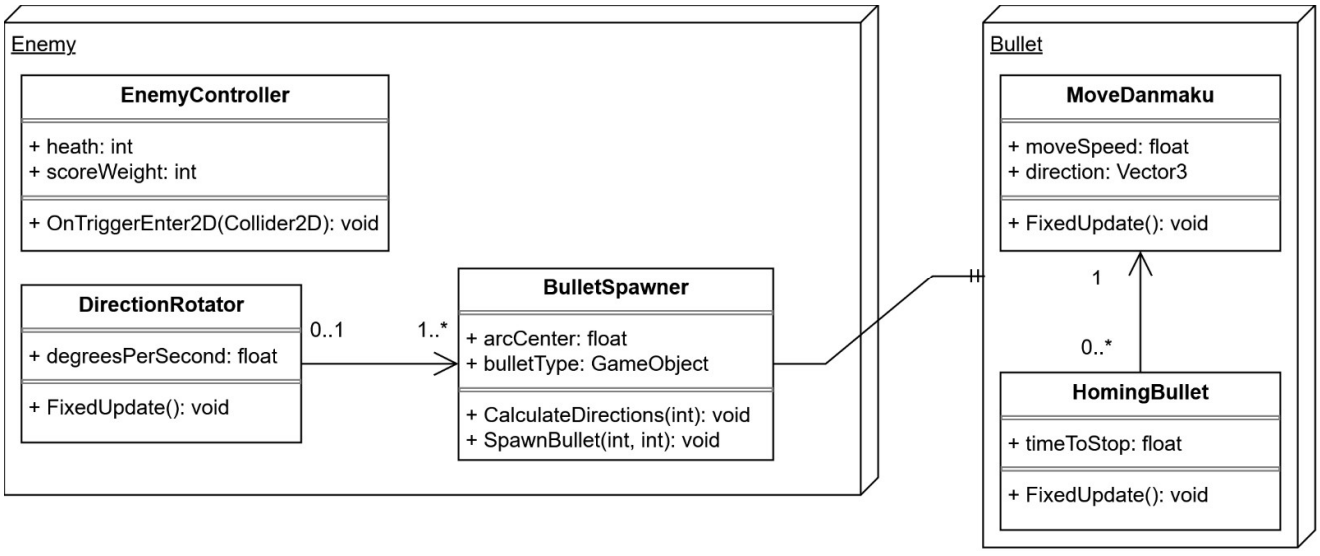


*Figure 16: Class diagram showing relationship between enemy classes and bullet classes.*

Figure 16 describes an enemy that uses a DirectionRotator instance to modify the firing direction of one or more BulletSpawner instances. These instances then spawn instances of a bullet prefab that includes an optional HomingBullet script to make the bullets home toward the player after a set time.

### *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 greater magic uses
    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) 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 greater magic 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.

As a singleton, there is only ever one instance of the Parameters script at a time. This instance handles the overall state of the game, as seen in the flowchart in Figure 17.

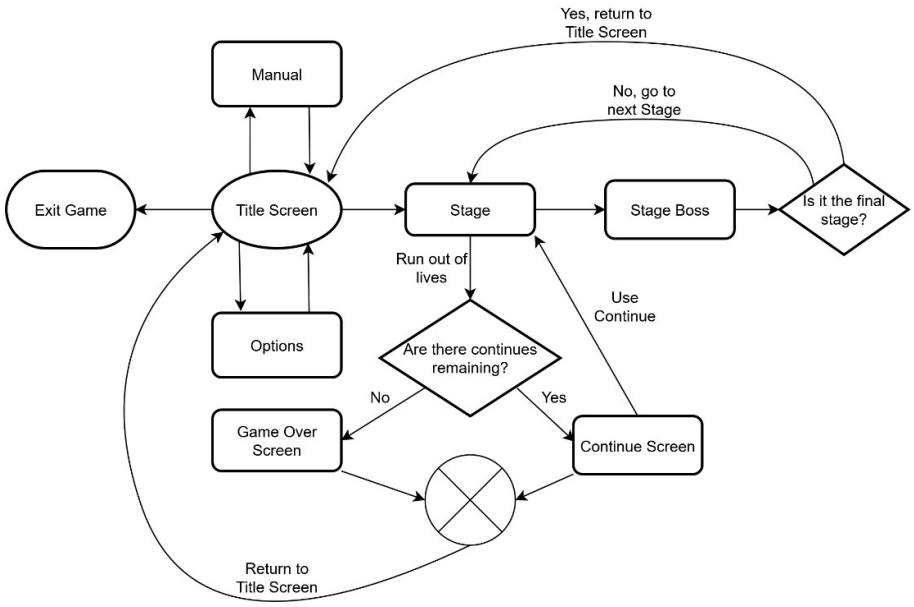


*Figure 17: Flowchart of game states.*

### *Level Design*

Due to short development time, I couldn't spend a lot of time designing areas and modeling them out for stage backgrounds. Instead, I modeled a small area of terrain to be looped repeatedly and seamlessly until the boss of an area appears, after which a second piece of terrain with different details would come into view (Figure 9). I then used thick distance fog to hide this technique from the player.

Here is a link to a video of the scene view with distance fog disabled to show what this looks like:

https://www.youtube.com/watch?v=S7l5qfUKVfI

The themes of the levels themselves were based around the storyline. This was done to emphasize the implications of the story on the game's world.

## Story

When I started working on this project, I had just one goal in mind: "I want to make a storyline that can be engaging even if it's only a few minutes long". I decided that the best way to do this would be to have a plot twist halfway through. Once I determined the setting for the story, I worked backward from there.

I wanted the story to be based around the events of an incident, so I chose the theme of "anachronism". In short, the geography of the region in which the game's events take place has quickly changed, such as a lake suddenly turning volcanogenic as you traverse it, as can be seen when progressing from stage 2 to stage 3.

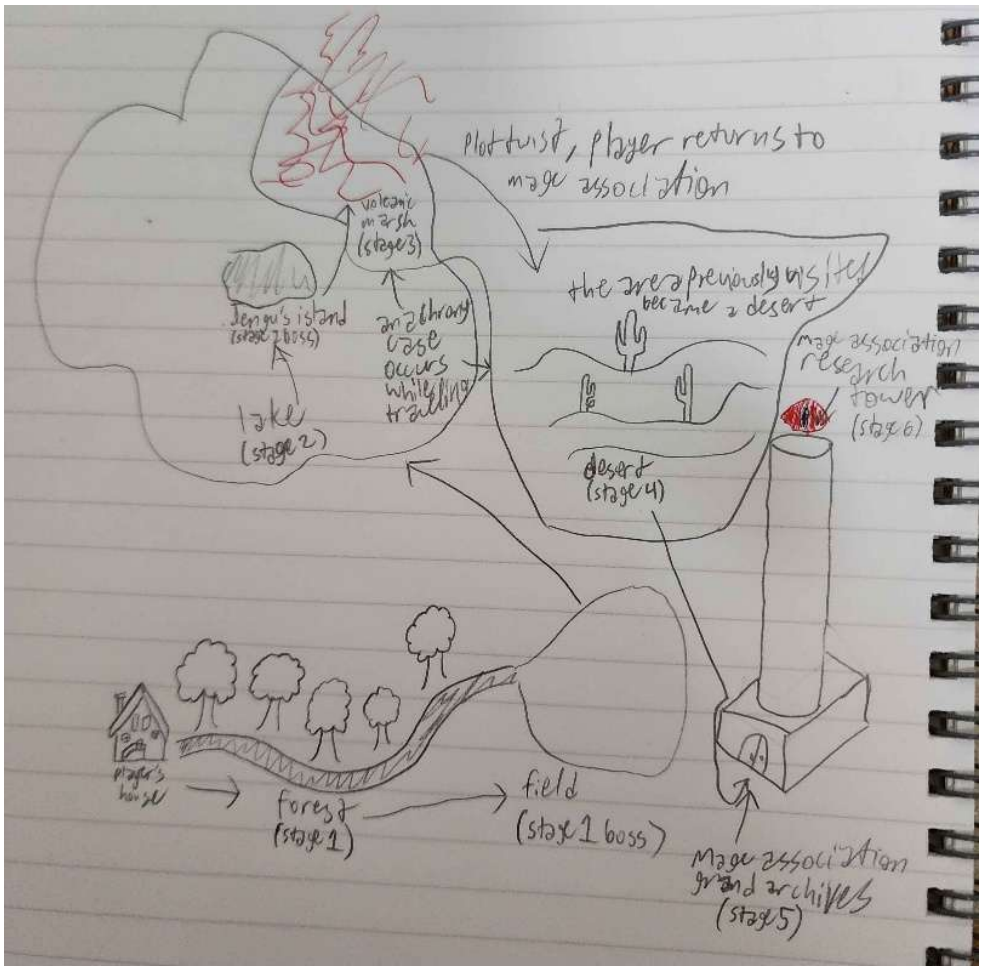


*Figure 18: Region map and stage order*

Figure 16 is a rough draw-up of the map of the region with a flowchart of the stages from planning the game out. Unfortunately, due to time constraints, the original stage 4, which was supposed to be a desert that replaces the location of the field at the end of the first stage during a case of anachronism, was cut, and the final product came to have the 5 stages it does now.

The rough storyline of the final product is as follows:

1. You are called to the field in front of the Mage Association Research Tower by your mage battalion lieutenant.
2. Tests your strength and then gives you a mission to solve the anachronism cases.
3. You start by checking out the lake near the plains in front of the research tower, as it appeared in a previous case.
4. You meet a jengu on an island in the middle of the lake who just took up residence here. Believing it to be suspicious, you fight, but it turns out that the jengu was unrelated.
5. The jengu points you toward another area near the opposite edge of the lake that seemed to recently have a case of anachronism.
6. As you proceed there, the area becomes volcanic.
7. When you get there, you meet another member of the mage association, but he attacks you when you call out.
8. After defeating him, he reveals that the mage association's grand master is the cause of the anachronism incidents.
9. You proceed back to the research tower and enter through the ground floor library.
10. The books in the library recognize you as a threat to the association and attack you, also causing the librarian to attack.
11. After clearing up the misunderstanding, you decide to proceed up the outside of the tower as it will be more efficient.
12. You are stopped on the way by other mages acting on the grand master's orders.
13. You reach the top of the tower where the grand master is waiting and fight him.
14. At this point, the ending you get depends on how many continues you used throughout the game.
    a. If you used any, you get the bad ending where the grand master stops but the cases don't get reverted.
    b. If you didn't use any, you get the good ending where everything gets reverted and you get promoted.

## Conclusion

The main goals I had when starting this project were to learn the basics of both 2d and 3d game development and to develop a fully functional game that has progression and a storyline. Some of the largest technical challenges with the game resided in designing the bullet and spawning systems such that they can be modified and expanded on without needing to largely modify the base scripts. If I were to use parts of this project in the future, I would want to redesign them to be much more modular, solving such issues as the previously mentioned hard-coded stage order. Some of the most important things that I learned through this project were how to manage time when working on a lot of features in parallel and determining what to cut from originally planned features in order to finish within the time allotted.

# 日本語版

## ドキュメント概要

これは、Unity エンジンで開発した『魔法の遺産』の機能と設計の詳細を説明する技術設計文書である。目的は、Unity での開発経験がある方なら誰でも、大きな困難なくコンテンツを修正または追加できるよう、重要な機能をすべて詳細に説明することだ。また、ゲーム設計時の私の考えや、なぜその選択をしたのかを明確にすることも目的としている。C#スクリプトを参照する場合には関連するコードを表示する。

## ゲーム概要

『魔法の遺産』は当方 Project にインスパイアされた弾幕シューティングゲーム。アーケードのコンティニュー方式を採用し、魔術士協会の審問官であるあなたは、王国で発生した奇妙な時代錯誤現象の原因を解明せねばならない。この再起の正体は？引き起こしているのは何か、あるいはだれか？様々な人々や生物の魔弾をかすめながら、この 2.5D 弾幕ゲームで自らその真相を突き止めよ！

## 開発時間

総開発時間：約 500 時間

本ゲームは 2021 年、ゲームデザイン大学講座において 10 週間にわたり開発された。授業外では毎日約 6 時間を費やし、一般的なゲームデザイン技法の研究、C#プログラミングの習得、Unity での開発・デザイン手法の学習を行い、ゲーム機能のプログラミングに加え、UI アセットやゲームスプライトを作成した。

さらに、Unity に影響を与える最近の CVE 脆弱性を契機として、2025 年末にゲームをアップデートし、多くの最適化を行い、コードの大部分を整理した。

本脆弱性：https://www.cve.org/CVERecord?id=CVE-2025-59489

## 開発環境と借用／委託アセット

プロジェクト全体の開発者兼デザイナーとして、ゲーム開発、グラフィック制作、音楽制作のために多様なツールを使用し習得した。使用ツールとその目的は以下の通りである：

- Unity — ゲームの各要素を開発し、最後製品に統合するため。主に公式 Unity ドキュメントを通じて独学で習得。
- VSCode — ゲーム内で使用する C#スクリプトの記述。
- GIMP — UI アセットとスプライトのデザイン。
- MIDI — 音楽の作曲。作曲の経験がなく音楽理論も知らなかったため、主に試行錯誤で習得。

とはいえ、これは機能するゲーム開発に焦点を当てた授業プロジェクトであり、販売の意図はなかったため、グラフィックやオーディオの技術を習得してそれらの側面を完全に整えるよりも、開発の基本と、作りたいゲームを支える基盤システムの構築に注力した。グラフィックやオーディオの一部のみを自分で作成した。

弾のスプライトの多くはオンラインで見つけたロイヤルティフリー素材であり、UI 要素は全てロイヤルティフリー素材と NASA の宇宙写真を背景画像として使用し、自ら編集して組み立てた。キャラクターアートとスプライト、並びにステージ 2、3，4 で使用された音楽は全て Maltolyte（https://www.youtube.com/@Maltolyte）によって製作された。

本プロジェクトにおいて私が制作または委託しなかったすべての素材は、教育目的であるという根拠のもと、米国（本ゲームが開発された国）のフェアユース条項に基づき使用されている。

## 要件

- プレイヤーと衝突時にダメージを与える弾が存在する
- 敵とプレイヤーは弾を発射できる
- 弾は様々な特徴を持ち、独自の移動パターンを実現できる
- 敵は攻撃パターンの難易度に基づき明確に区別できる
- プレイヤーのライフは有限である
- プレイヤーはライフを使い切った場合、コンティニューを使用してライフをリセットし、プレイを継続できる
- プレイヤーは困難な状況を乗り切るための有限リソースを所持する
- 複数のステージが存在する
- ステージごとに難易度が上昇する
- 各ステージの最後にボスが存在する
- ゲームには操作説明と世界観を解説するマニュアルを同梱する
- ゲームにストーリーラインを設ける
- ストーリーはボス敵との会話を通じてプレイヤーに伝える
- ステージとボスはそのデザインやキャラクターに合った音楽テーマが採用される
- ゲームは、ストーリーを主題的に表現する一貫した UI スタイルを採用する
- 敵を倒すと増加するスコアカウンターがある
- 獲得スコア値は敵の種類とステージによって異なる
- ゲームプレイ中に一時停止と再開が可能である
- 一時停止時には、タイトル画面への復帰や第一ステージからの再スタートが可能なメニューを表示する
- プレイヤーが現在のライフ数、大魔法の使用回数、スコアを確認できる UI を備える

## デザイン

このセクションでは、プロジェクトのグラフィック、音楽、ゲームメカニクスについて取り上げ、完成品の様々な側面とその着想減を紹介する。

### グラフィック

海洋で述べた通り、本作は ZUN の『東方 Project』から多大な影響を受けている。この長寿シリーズはシューティングゲームの一種である「弾幕」と呼ばれるジャンルに属す。特に本作に多大な影響を与えたのは『東方紅魔郷』と『東方妖々夢』である。物語の展開に調和した統一感のある魔法的な雰囲気をゲーム内に創出するため、UI の全体に星や魔法陣のモチーフを多用した。これはゲームを起動した直後のタイトル画面（図 1）で一目瞭然である。


*図1：タイトル画面*

背景画像には、魔法と神秘のテーマに沿って、NASA 提供のさんかく座銀河内にある NGC 604 領域の写真（https://www.nasa.gov/image-article/ngc-604/）を使用しました。タイトル画面の背景に明瞭に確認できます（図 1）。


*図2：オプションメニュー。ボタンアセットの拡大表示*

上記の図（図 2）に UI ボタンが表示されている。魔法のテーマを表現するため、タイトル画面とメニューで使用する左側に魔法陣を配した比較的シンプルなグラフィックを作成した。

ゲーム UI も同様のデザイン原則に従い、タイトル画面と同じ背景画像を使用していますが、今回は切り取り位置を変え、魔法陣のパターンを加えて雰囲気を演出している（図 3）。


*図3：3D ステージ背景を用いた 2D ゲームプレイ*

これは意図的に東方 Project のデザイン選択を反映したものである（図 4）。東方 Project では UI は視覚的に非常にシンプルだが、繰り返される模様によって繊細さを創出し、鮮やかでカラフルかつ慌ただしいゲームプレイと強い対比を成している。


*図4：『東方妖々夢』のゲームプレイのスクリーンショット*

その他の視覚的類似点には、弾のデザイン（図 5）とプレイヤーの可視ヒットボックス（図 6）が含まれる。

*図5：ゲーム内の弾の例*

*図6：ヒットボックスが表示されたプレイヤーキャラクター*

また、物語の舞台設定に合うと思われるフォントも選択した。最初に、精霊のようなファンタジーフォント（https://font.heartx.info/c86/）を使用している。これはタイトル画面（図 1）や会話文（図 7）を含む、表示されるテキストの大部分に使用されている。

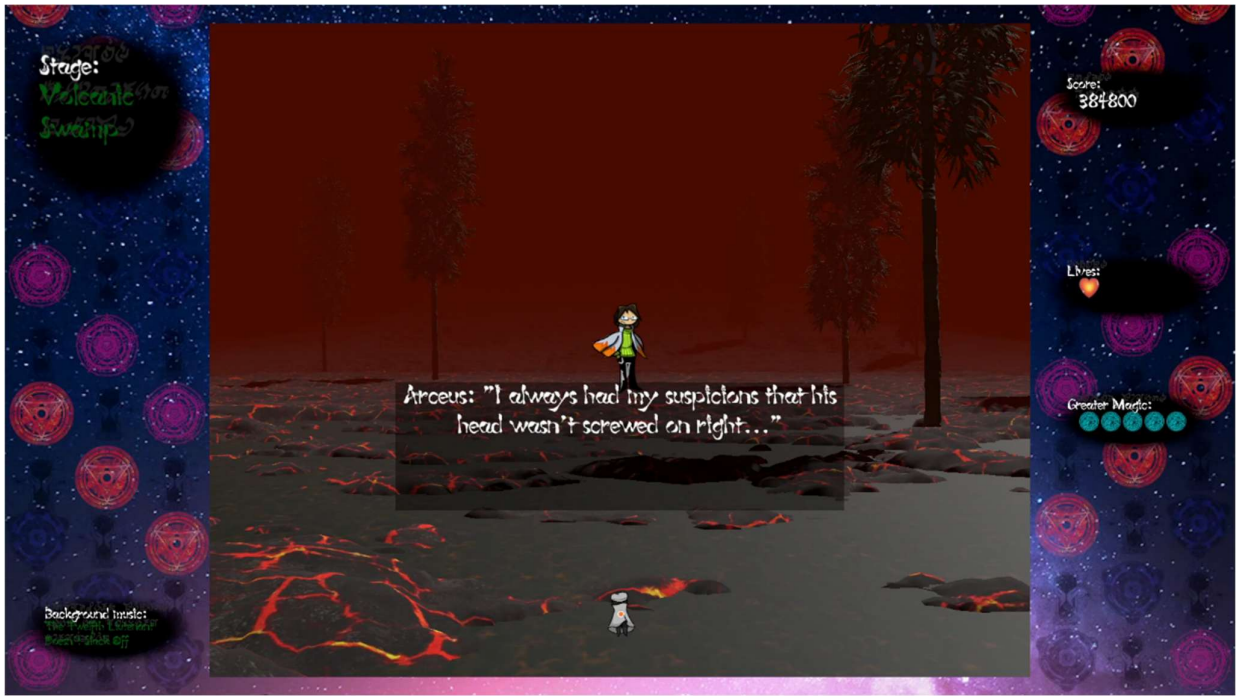


*図7：ダイアログと UI フォントの例*

次に、雰囲気をさらに高め、ビジュアルを物語に深く結びつけるため、全ての UI テキストに古代風フォントを影として追加した（図 8）。

(https://www.deviantart.com/ozziescribbler/art/Ancient-Quill-FREE-FANTASY-SCI-FI-FONT-297679343)


*図8：ステージ名の背後の古風な影付きテキスト*

東方 Project から得たもう一つの視覚的インスピレーションは、3D ステージ背景の上に 2D ゲームプレイを重ねた構成である。3D ステージでは、細部が簡素な小さな地形アセットを使用し、濃

い霧で遠景をぼかした。これは前述の通り（図 3）私のゲームでも確認できる。以下は、霧を除いた完全な状態の同一ステージの見た目である（図 9）。

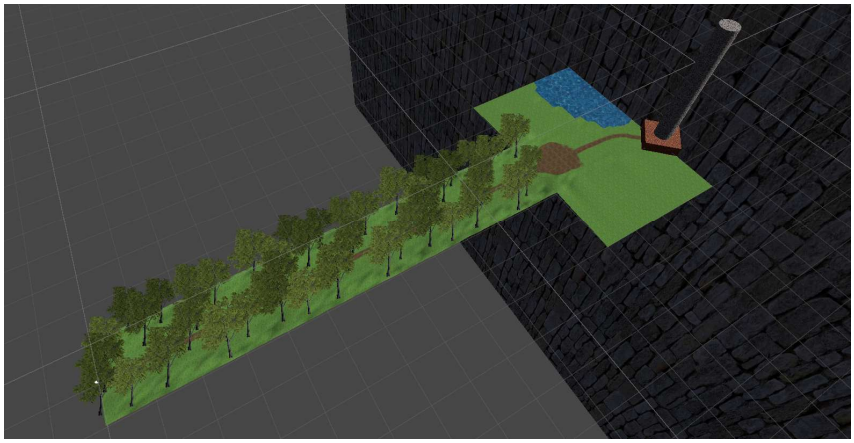

*図9：霧のないステージ1 のシーン全体*

見た目とは異なり、このステージはボスが出現するまで無限にループするように設計されている。右側の広いエリアは、ステージのボスが出現するまで視界に入らない。この仕組みの詳細は「ゲームメカニクス」セクションで説明する。

この設計選択は東方 Project にも着想を得ており、東方紅魔郷以降の東方 Project 公式作品では同様の手法が採用されている（図 10）。当時多くの弾幕系ゲームがタイル素材を用いた 2D スクロール背景を採用していた中、この手法はゲームプレイの背景を興味深くユニークに演出する非常に優れた方法だと考え、取り入れた。

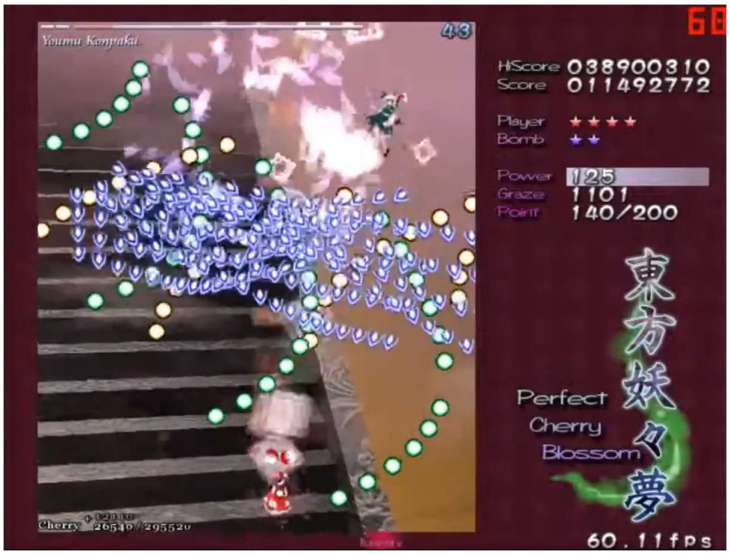


*図10：『東方妖々夢』のステージ背景に映る 3D 階段*

また、東方 Project の強みの一つは、各ゲームが独自の攻撃パターンを持ち、それらが視覚的なスペクタクルを構成することである。敵の攻撃パターンを鑑賞する際に感じる「芸術作品を眺めているような感覚」を再現しようと試みた。その例が上図（図 3）と下図（図 11）に示されている。

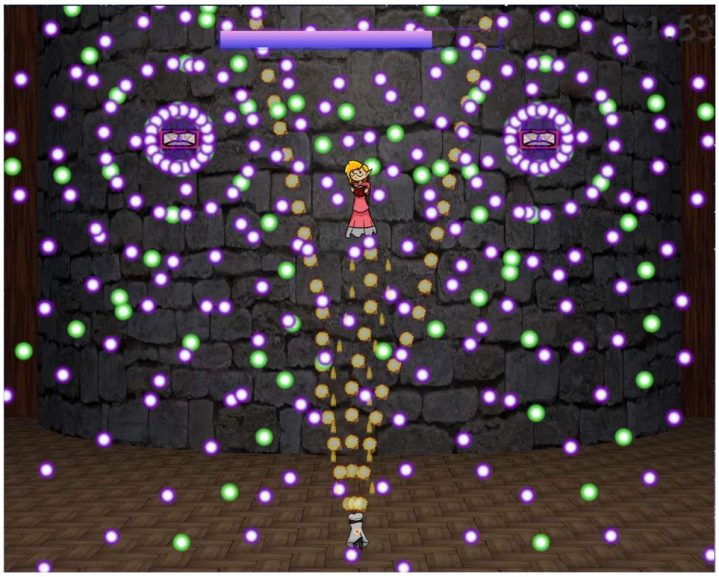

*図11：ボスの攻撃パターンの例*

これら全てが一体となって、魔法と神秘の非常にまとまりのある感覚を生み出していると思う。これはゲームのあらゆる側面で伝えようとしたモチーフである。

## 音楽

ゲームの音楽は、私と Maltolyte の共同作曲によるもので、楽曲がゲームの雰囲気に合うよう頻繁に意見交換を行った。私が作曲した楽曲も、ZUN 作品の幻想的な世界観からインスピレーションを得ている。ほとんどの楽曲でパワーコードを多用した。この文書内に音楽を添付することはできないが、ゲーム内の楽曲を参照する際には、その使用箇所を明記する。

このプロジェクトを始めた当初、私は音楽制作や音楽理論、デジタル音楽制作に関わるツールについて知識がなかった。つまり、効果的な音楽をゲームに組み込むためには、代替手段を見つけるか、自らこれらのスキルを急速に習得する必要があった。10 週間の終盤にかけて、楽曲の流れの設計といった重要なポイントについて、自らこれらのテーマを研究するとともに、知人である作曲家にも相談を重ねた。こうした取り組みを通じて、作曲について多少の知識を習得し、このプロジェクトを通じて全く新しいスキルを獲得するという挑戦を成し遂げることができた。

通常のステージ背景音楽については、常に高揚感があり、ボス戦というクライマックスへと積み上がっていくような感覚を持たせたかった。これらの楽曲には、ステージそのものの舞台を彷彿とさせる要素を込めた。例えばステージ 1 のテーマは森が舞台であるため、主にハープと柔らかなシンセサイザーを用いて、ファンタジー探検を想起させる雰囲気を演出した。このような落ち着いた楽曲については、日本ファルコムの『空の軌跡』やスタジオジブリの『天空の城ラピュタ』といった古典的なファンタジー作品からインスピレーションを得ている。

同様に、もう一人の作曲家である Maltolyte にもこの方針に沿うよう依頼した。彼の作曲に対する私の管理は最小限に留めたが、結果として各楽曲は求められたトーンに見事に合致し、物語や設定などの要素を考慮した適切な使用が実現していた。

ボス戦で使用される音楽については、ほぼ全てが「ボス最強攻撃でクライマックスを迎え、一息つく静寂の瞬間を経て、再び頂点に達してボス戦を締めくくる」という展開を意図して作曲されている。この特徴はステージ 5 の最終ボス戦で使用される楽曲で最も顕著に表れている。

このパターンから外れる楽曲が 2 曲あり、いずれもステージ 3 に配置されている。これには物語上の理由があり、ステージ 3 の会話で主人公の名が明かされると同時に、物語の展開に重大な転換が訪れるためである。そのため、Maltolyte にはサスペンスを喚起し、混乱の雰囲気を醸し出す楽曲の作曲を依頼した。特にステージ 3 のボス戦テーマでは、タイトル画面のモチーフを反転させることでこうした感情を構築するのが適切と考え、具体的にそのように依頼した。

開発期間がもっとあれば、私の作曲と Maltolyte 氏の作曲のクオリティ差を縮める努力をしただろう。しかし、音楽制作の経験やツールが全くない状態から、この課題に取り組み、ゲームが影響を受けた東方 Project の要素を確固たるものにするシンプルな音楽を作曲するスキルを身につけた。

## ゲームメカニクス

初めて手掛けたゲームプロジェクトとして、提出期限までの 10 週間という期間内に完成させるためには、ゲームシステムとコンテンツをシンプルに保つ必要があることを理解していた。そのため、『東方 Project』のようなゲームプレイ体験を創出することを要件の中心とした。ゲームシステムの殆どは、私のゲームに直接反映されている。移動、複雑な弾幕を避けるためのスローフォーカス、自身の弾発射、そして画面上の敵弾をすべて消去する「大魔法」の使用が可能である。

### *難易度*

当初は難易度設定を実装したかったが、時間制限内で実現するのは困難だと判断し、妥協案として『東方 Project』初の Windows ゲームから着想を得たシステムを採用した。これは合理的な範囲内で、開始時のライフ数と大魔法の数を設定できる仕組みだ（図２）。『東方 Project』と同様に、ライフを失うと大魔法の使用回数は最大値にリセットされる。また、ライフが尽きた場合、スコアをリセットする代償として限られたコンティニュー回数を使用しプレイを継続できる。

さらに、敵の発射パターンは後半ステージになるほど難易度が上がり、ゲーム終盤に向けて自然な挑戦の増加が図られている。この例として、序盤（図 12）と終盤（図 13）のボス戦が挙げられる。


*図12：初期ボスの発射パターン*

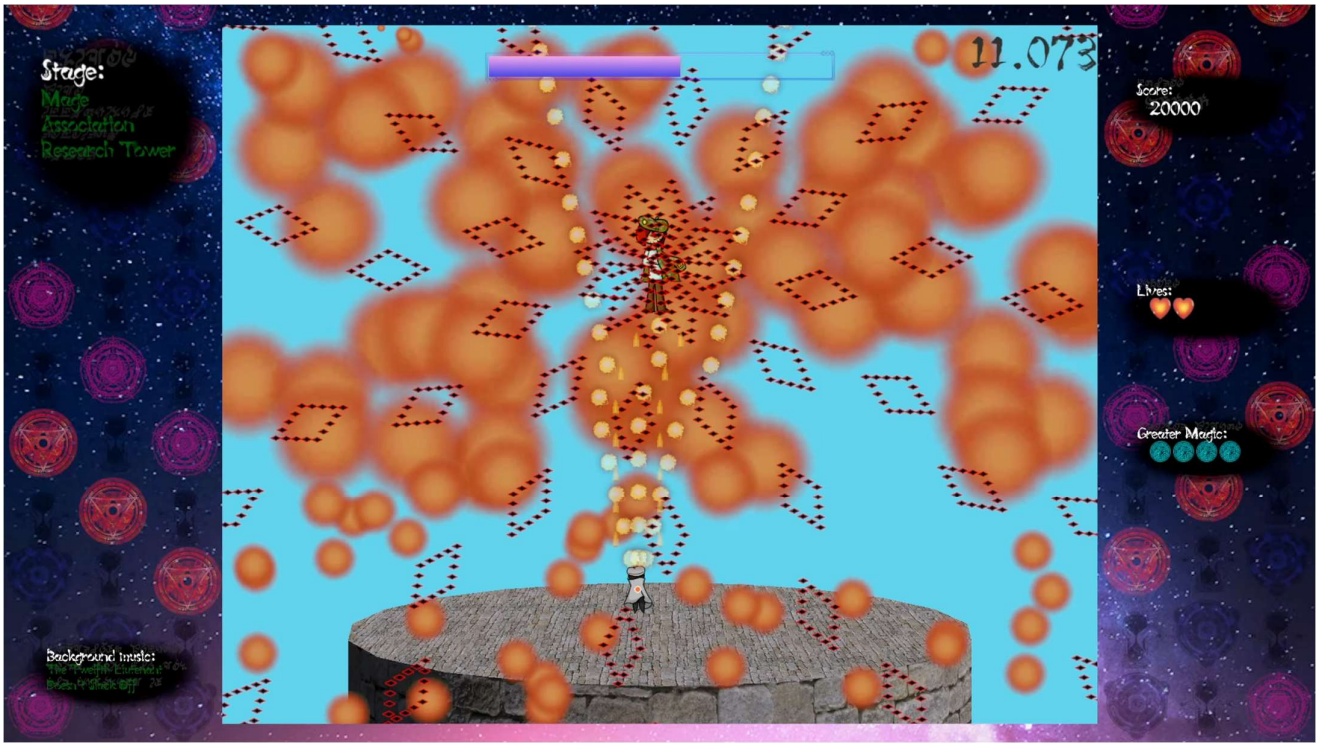


*図13：周期ボスの発射パターン*

実際のバランス調整については、難易度オプションが存在しないため、様々な『東方 Project』作品での自身のクリア実績を参考に、当時の自分のスキルレベル（コンティニューをすべて使用すればノーマル難易度でクリア可能な程度）を基準に難易度を設計した。この意図は、この種のゲームにおいて私よりも熟練したプレイヤーであれば、コンティニューを一切使わずにクリアでき、ゲームがグッドエンディングを表示するようにすることだった。一方、プレイヤーが与えられた 5 回のコンティニューのうち 1 回以上を使用した場合、代わりにバッドエンディングが表示される仕組みである。

### *ユーザー入力*

ユーザー入力はイベント駆動型入力処理である。パフォーマンスへの影響はほぼない。入力方法はキーボードだ。ゲームの操作方法は以下の通り：

- Z　—　弾発射／選択
- X　—　大魔法使用
- シフト　—　フォーカス（移動悠長）
- 方向キー　—　移動／UI 操作
- Esc　—　ポーズメニュー表示

プレイヤーが新たな操作方法を習得する必要がないよう、敢えて『東方 Project』の操作体系を模倣している。

ユーザー入力機能の管理は、シングルトン設計パターンを用いて何のスクリプトからでもアクセス可能なため簡素化されている：

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();
        }
    }
}
```

それから、大魔法の使用を扱う PlayerController スクリプトの以下のセクションのように、アクションを定義して有効にすることができる：

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

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

    ...

    //大魔法入力有効
    void EnableBombAction()
    {
        bomb.performed += OnBombPressed;
        bomb.Enable();
    }

    //大魔法入力無効
    void DisableBombAction()
    {
        bomb.performed -= OnBombPressed;
        bomb.Disable();
    }

    //大魔法ボタンが押されたら起動
    void OnBombPressed(InputAction.CallbackContext callbackContext)
    {
        //大魔法が残っているかを確認
        if(Parameters.singleton.bombs >= 0)
        {
            //大魔法を起動
            Instantiate(spell, transform);
            Parameters.singleton.bombs--;
            Parameters.singleton.updateBombDisplay();
            //プレイヤーを一旦無効にする
            StartCoroutine(BombInvulnerability());
```

```
        }
    }

    ...
}
```

2025 年にゲームをアップデートした際、暫定的にコントローラー対応も追加したが、ゲーム全体での徹底的なテストはまだ行っていないため、現時点ではゲームマニュアルにその情報を記載していない。キーボード入力方式と同様に問題なく動作すると確認できた時点で、この文書及びゲームマニュアルに追加する予定だ。

### *弾スポナー*

弾幕シューティングゲームにおいて、複数なパターンで頻繁に弾を発生させることは重要な要素の一つであるため、設定された条件で自動的に弾を発射する、無限に再利用可能な弾発生器を作成したいと考えた。これを実現する最良の方法はファクトリー設計パターンだと判断した。このパターンでは、複数の弾の要素を一度に生成・管理することができる。その「BulletSpawner」というスクリプトは以下の URL で全体的に見ることができる。

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

関連する変数とそれぞれの目的に関するコメントは上記のコードに記載されているので、ここでは行われている数学的な処理のみ説明しておく。GitHub では文字コードの問題が発生する可能性があり、日本語コメントが正しく表示されない場合もある。そのため、こめんとを正しく閲覧するにはソースコードをダウンロードする必要があるかもしれない。

ゲームの弾発射タイミングは Unity の組み込み関数 fixedDeltaTime を基にしており、スプーン率や移動などの機能を実時間に連動されるため、関連する移動計算は FixedUpdate 内で実行される。これにより弾の速度は Time.timeScale で制御可能となり、単一の変数を 0〜1 で変更だけでゲームを一時停止できる。将来的にゲーム機能を拡張する場合、時間を加速・減速させる機能の実装にも応用可能だ。

弾の発生位置と方向は、Unity の Update メソード内で比較的単純な三角関数と代数を用いて計算される：

```
void Update()
{

    //shootAtPlayer が true だったらプレイヤーを狙う
    if (shootAtPlayer)
    {
        AimAtPlayer();
    }
```

```
    //発射設定が変わった場合、directions と spawnLocations を再計算
    if(SpawnDetailsHaveChanged())
    {
        CalculateSpawnDetails();

        dSPS = spawnsPerSecond;
        dCo = arcCount;
        dD = arcDegrees;
        dCe = arcCenter;
    }
}

//前フレームのスポーン設定が今のと違うかどうか
bool SpawnDetailsHaveChanged()
{
    return dSPS != spawnsPerSecond || dCo != arcCount || dD != arcDegrees || dCe != arcCenter;
}

//全てのスポーンに関する変数を計算
void CalculateSpawnDetails()
{
    //中心放射状ラインの角度を計算
    int center = (int)Mathf.Floor((arcCount - 1f) / 2);
    ///角度のオフセットを計算
    centerPLC = arcCenter;
    if (arcCount % 2 == 0)
    {
        centerPLC -= arcDegrees / arcCount / 2;
    }

    //全放射状ラインの角度を計算
    CalculateDirections(center);

    //放射状ラインごとのスポーン位置を計算
    CalculateSpawnLocations();

    //スポーン間のフレーム数を計算
    framesPerSpawn = (int)(Mathf.Floor((int)(1 / Time.fixedDeltaTime) / spawnsPerSecond));
}

//放射状ラインの方向角度を計算
void CalculateDirections(int center)
{
    directions = new float[arcCount];

    for(int i = 0; i < directions.Length; i++)
    {
        //中心から与えられた方向を基に、反時計回り方向の最も端の放射状ラインから時計回り方向の最も端の放射状ラインまでの
        //総角度、放射状ラインの総数をもちいて、すべての放射状ラインの方向を算出。各放射状ラインの角度方向を計算。
        directions[i] = centerPLC + (arcDegrees / arcCount * ((float)i - center - (((arcCount % 2) - 1) /
2)));
    }
}

//放射状ラインごとのスポーン位置を計算
void CalculateSpawnLocations()
{
    spawnLocations = new Vector3[arcCount];

    for(int i = 0; i < spawnLocations.Length; i++)
    {
        spawnLocations[i].x = Mathf.Cos(directions[i] / 180 * Mathf.PI);
        spawnLocations[i].y = Mathf.Sin(directions[i] / 180 * Mathf.PI);
        spawnLocations[i] = spawnLocations[i] * bulletSpawnRadius;
```

```
    }
}
```

発射される弾は一発当たりの弾数が非常に多い場合、多数のベクトルを再計算する必要があるため、この関数のパフォーマンスが懸念された。そこで、変更可能な設定（dSPS、dCo、dD、dCe 変数）のコピーを作成し、それぞれの設定の最終既知値を保存するようにした（各設定変数については上記の Update メソードを参照）。これにより、設定が毎フレームではなく、変更された場合だけに一度だけパフォーマンス負荷の高いベクトル計算を実行できるようになる。

同様に、コード内でベクトル演算が発生する場合には、コンパイル後のバイナリコードにおける演算呼び出し回数を削減するため、最適化可能な場合、ベクトル演算を常に演算順序の最後に配置するようにしている。例えば、2 つ以上のプリミティブ値（int、float 等）を含む式で Vector3 を使用する場合、演算順序の最後に Vector3 を配置することで、Vector3 の各要素に対する演算を一回だけ実行できる。Vector3 が先頭にある場合、演算が 3 つの値に対して実行されるため、後続の核プリミティブごとに計算回数が三倍になる。要するに：

Vector3 * int * int → 3 + 3 = 6 演算

int * int * Vector3 → 1 + 3 = 4 演算

それから、弾の発射時に各弾を生成・配置するために生成位置と方向が使用される：

```
//spawnsPerSecond で定義された発射率で弾を発射
void FixedUpdate()
{

    //クールダウンが終わったら発射
    if (framesSinceLastSpawn >= framesPerSpawn)
    {
        FireBulletRound();
    }
    //このフレームで弾が生成じゃなかったら、発射からのフレーム数を上がる
    else framesSinceLastSpawn++;
}

...

//設定を使用して弾を発射
void FireBulletRound()
{
    int spawnLocationIndex = 0;
    //放射状ラインごとに
    foreach (Vector3 _ in spawnLocations)
    {
        //そのラインの弾数によって（lineCount）
        for (int linePositionIndex = 0; linePositionIndex < lineCount; linePositionIndex++)
        {
            //弾をスポーン
            SpawnBullet(spawnLocationIndex, linePositionIndex);
        }
        spawnLocationIndex++;
    }
```

```
    //発射音を再生
    PlayShootSound();
    //発射クールダウンをリセット
    framesSinceLastSpawn = 0;
}

//一つの弾をスポーン
void SpawnBullet(int spawnPositionIndex, int linePositionIndex)
{
    //BulletPool が存在していたら、スポーンする弾をそれの子オブジェクトにする
    GameObject bulletPoolGameObject = GameObject.Find("BulletPool");
    Transform bulletPoolTransform = null;
    if(bulletPoolGameObject != null)
    {
        bulletPoolTransform = bulletPoolGameObject.transform;
    }

    //弾をスポーン
    GameObject bullet = Instantiate(bulletType, spawnLocations[spawnPositionIndex] + transform.position,
Quaternion.Euler(0, 0, directions[spawnPositionIndex]), bulletPoolTransform);

    //スポーンした弾の移動方向と速度を設定
    if (bullet.TryGetComponent<MoveDanmaku>(out var mD))
    {
        mD.moveSpeed = moveSpeed + (deltaSpeed * linePositionIndex);
        mD.direction = spawnLocations[spawnPositionIndex].normalized;
    }

    //スポーンした弾の軌道設定を設定
    if (bullet.TryGetComponent<AngleOrbit>(out var aO))
    {
        aO.firedFrom = gameObject;
        aO.offsetAngle = orbitAngle;
        aO.isOrbit = isOrbit;
        aO.stopTime = orbitStopTime;
    }
}
```

このようにスポーン機能を実装し、設定を時間経過とともに更新できるようにすることで、スポナーを他のスクリプトで装飾し、時間経過とともに変更を加えることが可能になる。これにより興味深いパターンを生成できる。例えば、発射方向を時間経過とともに回転させるだけのシンプルな使用例を以下に示す：

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

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

    //一秒あたりの角度差
    public float degreesPerSecond = 15f;
    //対象スポナー
    public BulletSpawner[] instantiators;
    //角度は時間が立つと変化するかどうか
    public bool angleIncreasing = false;
    //一秒ごとの角度変更
    public float deltaAngle = 15f;
    //フレーム当たりの角度差
    float degreesPerFrame;
```

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

    void FixedUpdate()
    {
        if (angleIncreasing)
        {
            degreesPerFrame += deltaAngle * Time.fixedDeltaTime;
        }
        foreach(BulletSpawner current in instantiators)
        {
            //発射方向を回転
            current.arcCenter += degreesPerFrame;
        }
    }
}
```

同一のオブジェクトのコンポーネントとして追加し、DirectionRotator インスタンスの instantiators 配列を BulletSpawner インスタンスに参照させ、両方を設定する。エディター上での表示例として、ステージ 1 ボス用の最終形態のスポナーの設定（図 14）と、生成されたパターン自体のスクリーンショット（図 15）は以下にある。

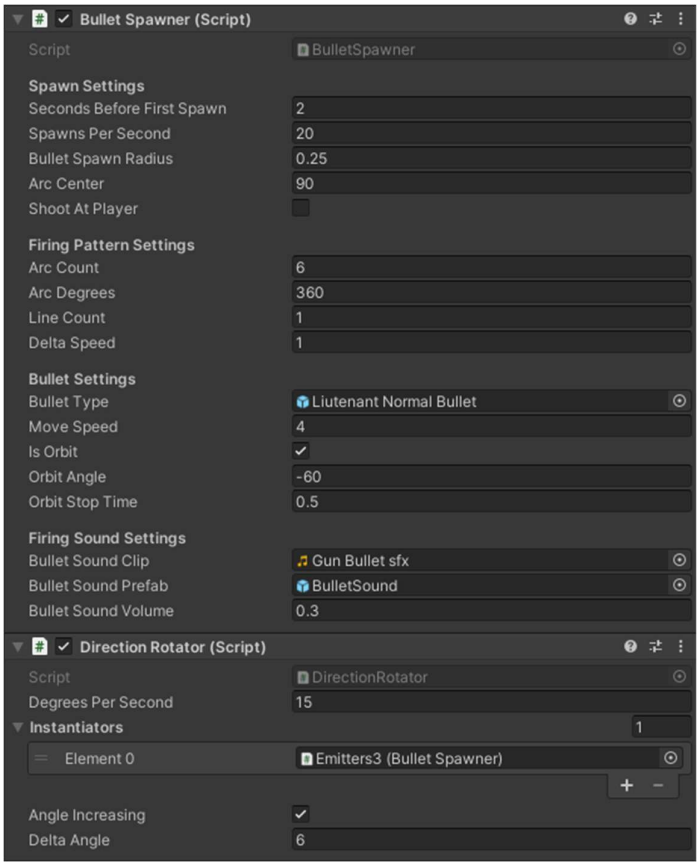


*図14：BulletSpawner とそれを変更するデコレーターの設定*

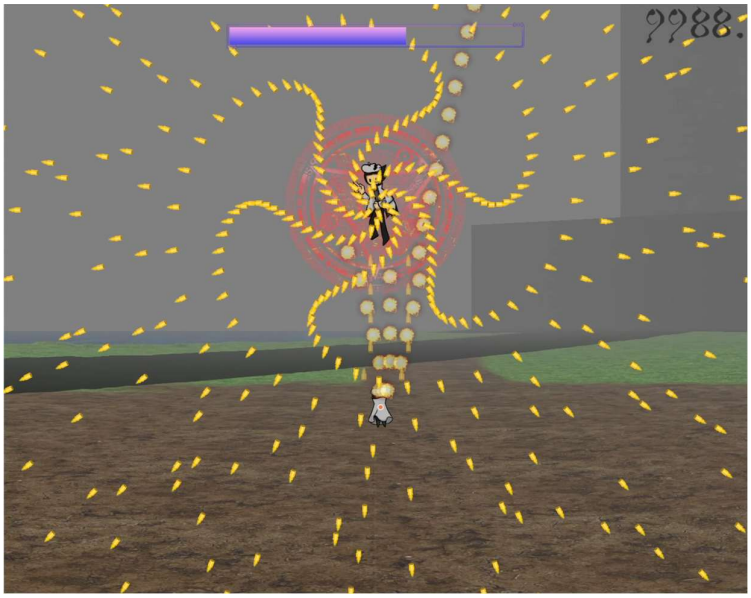

*図15：ステージ1 ボスの最終形態の弾幕パターン*

### *弾*

弾自体もモジュール化された機能を実現するため、デコレーター設計パターンを使用している。標準的な弾には MoveDanmaku スクリプトのみが実装されており、これは単独では非常に単純で、指定された方向と速度で弾を移動させるだけである：

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

```
public class MoveDanmaku : MonoBehaviour
{
    //移動速度
    public float moveSpeed = 1;
    //移動方向ベクトル
    public Vector3 direction = new Vector3(0,0);

    void FixedUpdate()
    {
        //移動速度で移動方向に移動
        transform.position += moveSpeed * Time.fixedDeltaTime * direction;
        //弾のグラフィックを移動方向に向ける
        float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;
        transform.rotation = Quaternion.Euler(new Vector3(0, 0, angle));
    }
}
```

さらに複雑な動きパターンを実現するため、追加機能でデコレートできる。例として次のスクリプトでは、弾が指定時刻に減速・停止した後、方向転換してプレイヤーの現在位置へ向かって移動する：

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

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

    //停止までの秒数
    public float timeToStop = 3f;

    //ベース弾移動スクリプト
    public MoveDanmaku moveScript;
    //フレーム当たりの速度差ベクトル
    Vector3 deltaDirection;
    //弾からプレイヤーまでの方向ベクトル
    Vector3 directionTowardPlayer;
    //プレイヤーの位置に回転したかどうか
    bool hasHomed = false;

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

    void FixedUpdate()
    {
        //もう回転したなら何もしない
        if (hasHomed)
        {
            return;
        }

        //移動はまだ停止していないことを確認
        if(moveScript.direction != new Vector3())
        {
            //移動を減速
            moveScript.direction -= deltaDirection;
        }
        else
        {
            //プレイヤーの位置に回転
            directionTowardPlayer = (PlayerController.singleton.transform.position -
transform.position).normalized;
            moveScript.direction = directionTowardPlayer;
            hasHomed = true;
        }
    }
}
```

弾のスポーン機能と自体機能を組み合わせた図 16 は、典型的な敵を示す UML クラス図だ。

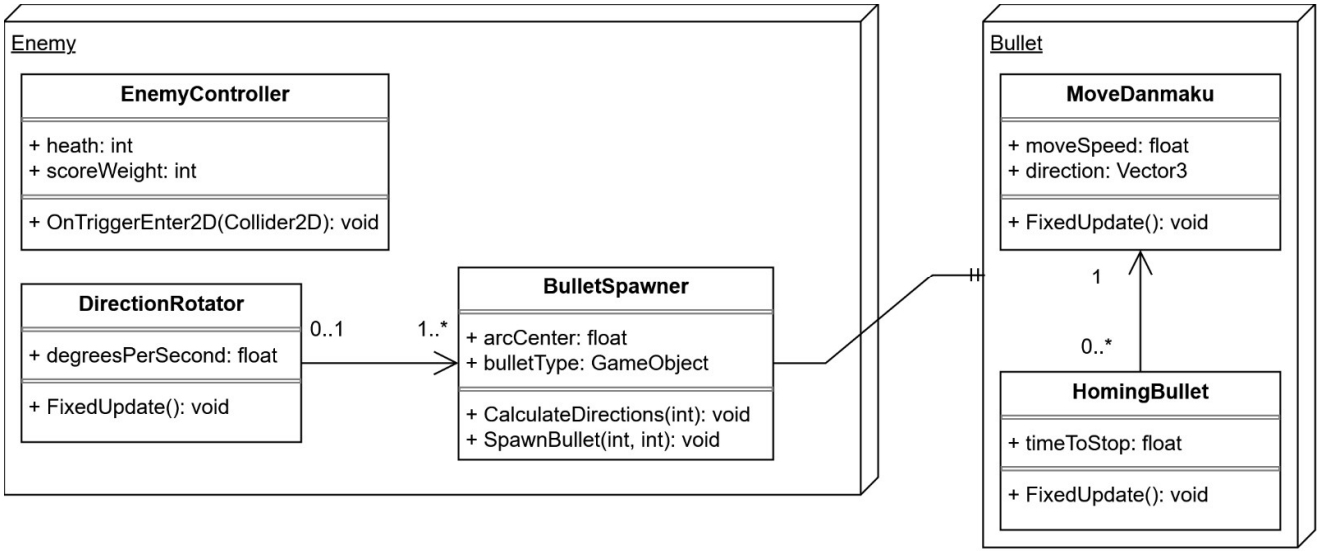


*図16：敵クラスと弾クラスの関係*

図 16 では、DirectionRotator インスタンスを使用して 1 つ以上の BulletSpawner インスタンスの発射方向を変更する敵を記述している。これら BulletSpawner は、指定 HomingBullet スクリプトを含む弾プレハブのインスタンスを生成する。これにより、弾は設定時間後にプレイヤーに向かってホーミングする。

*ゲーム状態管理*

*レベルデザイン*

ストーリー

結論

# Table of Contents