Last Hymn

Technology Report

Jefferson Le
Tyler Pinho
Curtis Spence

Table of Contents

Introduction

Abstract

Last Hymn was created by the team of Tyler Pinho, Jefferson Le, and Curtis Spence with the desire to create an eccentric Role Playing Game focused on the exploration of a strange, dying world. Battles in the game are based off of rhythm games like Dance Dance Revolution using a procedural generation algorithm that makes every encounter unique. This is then complemented with the path system where each enemy has unique rhythm patterns to give them different types of combat opportunities.

In Last Hymn, the player arrives on a train at the World's End Train Station where they are greeted by a mysterious figure and guided to the Forest where they witness the end of the world and find themselves back at the train station before they left for the Forest. With only a limited amount of time per cycle of the world, the player must constantly weigh the opportunity cost of each decision, and only with careful thought, conviction, and tenacity will the player find a conclusion from the never ending cycle of rebirth.

Blending both Shinto architecture and modern elements, Last Hymn used a "fantasy-chic" aesthetic in order to provide memorable locations and dissonant imagery. As the player explores they will struggle against puzzles and dynamic, rhythm based combat while trying to unravel the mystery of the world's looping time. Last Hymn was designed to develop innovative and dynamic new solutions for combat, exploration, and mapping. From this project all three team members were able to grow their software development and game design skills, achieving goals like improved level design, improved asset pipelines while simultaneously aiming to craft an experience that will be unforgettable for players everywhere.

Team

Jefferson Le

Jefferson was the primary artist for Last Hymn and produced the art assets used in the project. His goals participating in the project were to refine his production pipeline to be more efficient and produce consistent work quickly, improve in areas he was previously lacking like animation, and gain experience working in a team for a long term project. The first two goals were blockers for previous projects and placed a hard limit and the scope of current and future projects, while the third goal was due to a belief that such experience was necessary for the modern engineering field.

By the end of Last Hymn's development based on user feedback most of the aforementioned goals were achieved. Over one-hundred unique assets, including numerous animations, were produced for the game greatly eclipsing production on previous projects and reception to the game's aesthetics was very positive, as was reception to the game as a whole, making the entire experience well fitting to fulfill the third goal.

Tyler Pinho

Tyler was the level designer and programmer for Last Hymn. His goals coming into the project were to focus on creating environments that were memorable and fun for the player to explore. Level design had been a challenge in previous projects and required Tyler to spend many hours restructuring levels and iterating on their design to make sure they were fun.

At the end of Last Hymn's development, it was evident that the level design of Last Hymn was one of its strongest points as referenced through user testing. Many players wanted to explore the world even more than they were and made frequent positive remarks about the high-quality locations they were visiting. Tyler also implemented the graphics and subsystems of the game such as working on Shaders used for cutscene transitions, the dynamic layering system, turn controllers, and more, all of which were highly praised.

Curtis Spence

Curtis was the systems and framework developer for the team. He approached coding Last Hymn with the intention of creating sleek and effective system designs through out the game's underlying core functions. Using the culmination of the past 3 years of experience in developing game systems, this project posed a perfect challenge to match Curtis's skill level in coding seeing as it aimed at being more in depth and large scale than anything previously worked on.

To compare the goals with the results, the final game contained many key features including: dynamically generated menu system, arrow and NPC pathing, train station auto-generated arrivals, and time delegate events. All of these features were constructed with the initial design intentions in mind which promoted clean, readable, reusable code throughout the project.

Definition of Terms

Throughout this report a variety of terms will be used that are specific to the game development field and/or are going to be used as acronyms. In order to avoid language problems we have defined important terms and acronyms here.

Term	**Definition**
Role Playing Game (RPG)	A game in which each participant assumes the role of a character, generally in a fantasy or science fiction setting, that can interact within the game's imaginary world.
Non-Player Character (NPC)	Any character in the game that the player interacts with and/or controls that is not the player.
Quest Line	A series of linked together tasks the player has to complete in order to obtain a reward or outcome.

Purpose and Goals

Last Hymn was produced by our team for the purpose of crafting an immersive RPG experience that focuses on developing a strong visual aesthetic centered around incorporating older Eastern architecture with modern sprite designs and environment while incorporating unique rhythm elements. We wanted to create two sections in our gameplay: an exploration-based RPG side that focuses on the completion of quest lines and a rhythm battle system. However, we wanted to blend both sides of the game into each other so they complemented one another instead of feeling completely distinct.

A heavy emphasis was placed on creating many different RPG areas for the player to explore. In Last Hymn we wanted the player to be given very little direction throughout the world and instead have the focus brought to the player’s journey through the world. We desired to craft our game to have a warm, melancholic feeling to it where the player felt lonely in our world without ever feeling lost or abandoned. This was done through creating tight-knit environments and giving the world a personality that the player would interact with instead of other characters.

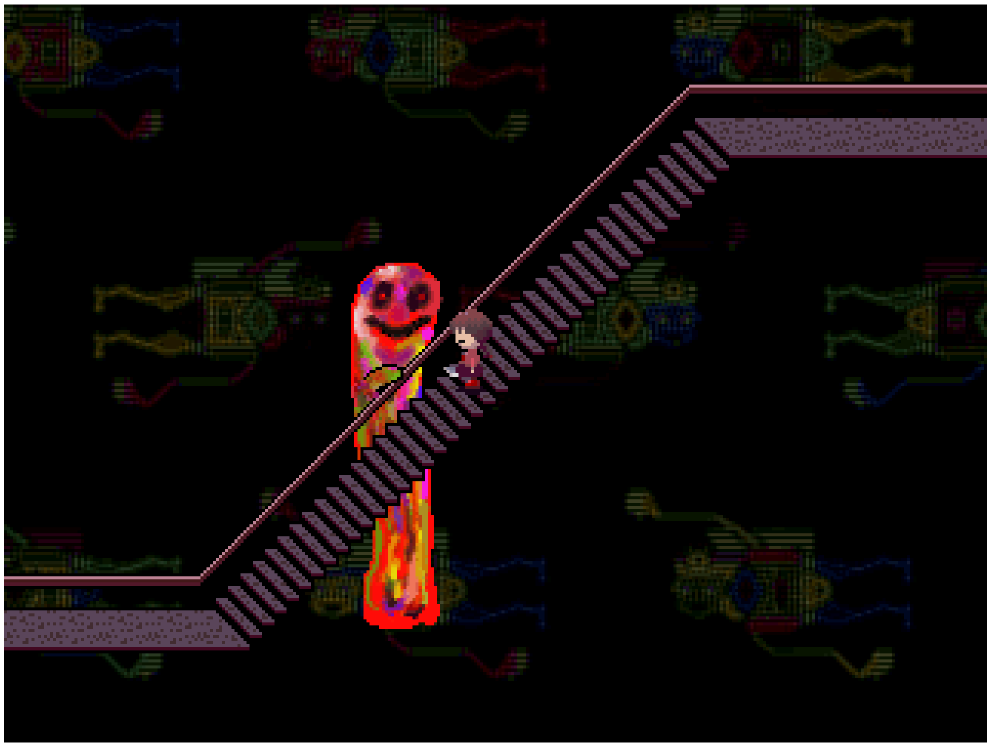
We also wanted to have a unique battle system that has never been made made before. We developed a rhythm engine that takes in any song input and would create procedural patterns so every fight was different while conforming to a particular set of rules. This allowed for dynamic difficulty settings and creating a rhythm game that is more accessible to different skill levels.

Finally we took a developmental approach that placed the player first. Any and all decisions made were with the player’s experience as top priority with the objective of producing a memorable experience as well as the potential to have a product that could be marketed in the future to an audience.

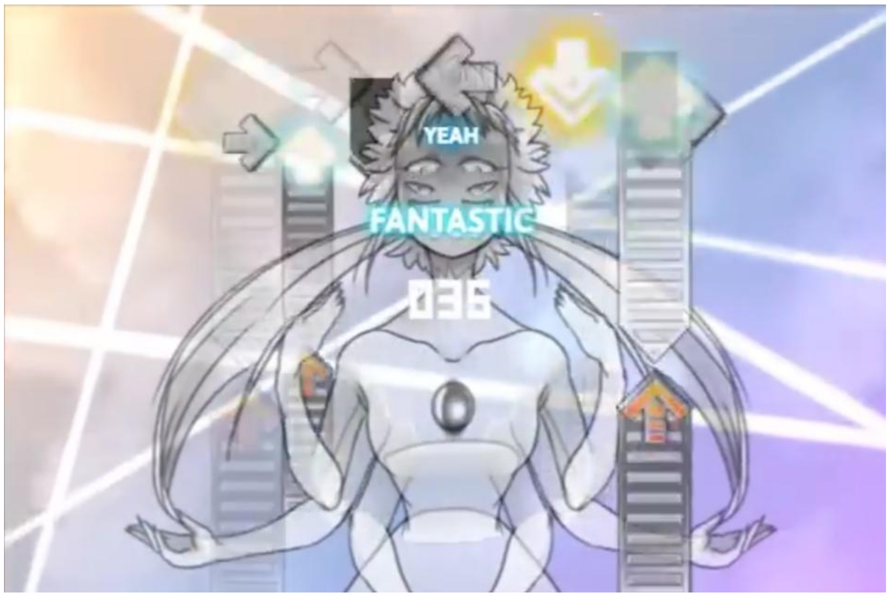
Inspirations

The Last Hymn game finds itself deeply inspired by Yume Nikki a RPGMaker 2003 RPG created by the Japanese solo developer Kikiyama (Project YUMENIKKI). Literally translating to "Dream Diary", Yume Nikki centers around the dreams of a young girl named Madotsuki as she attempts to find a variety of effects around a multitude of interconnected dream worlds. The game has no explicit plot laid out and focuses more on player speculation and a strong motif of loneliness and exploration.

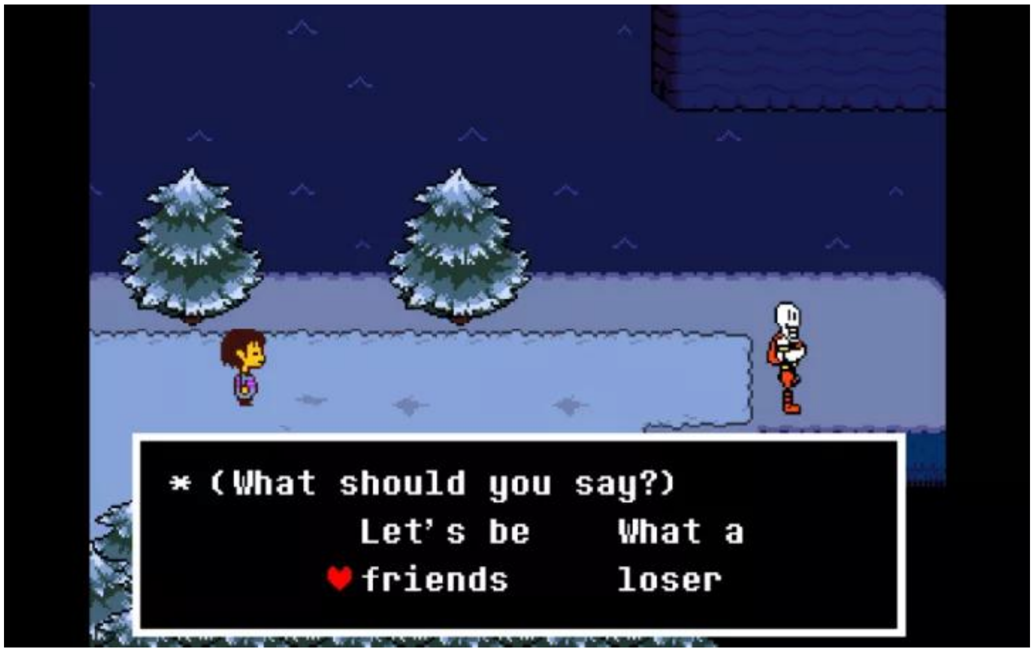
While Kikiyama has a cult following and is a progenitor of similarly developed games, we felt that the game had several potential areas of improvement so we endeavored to frame our game using the same developmental techniques and feelings of the game. We wanted to incorporate well-defined combat into the game that complements the RPG quests and serves to break up some of the monotony of exploring a deserted world. We also endeavoured to incorporate more NPCs into the game to allow for some more dialogue and the development of other characters which Yume Nikki did not do to the same degree. However, we wanted to borrow the same atmospheric qualities that Yume Nikki held and develop a wide variety of areas with different aesthetics and their own personalities so that exploration of the world was one of the main focuses of the game.



Our battle side of the game is an amalgam of different genres inspired primarily by the “WinDEU Hates You” and Undertale (http://undertale.com/). WinDEU Hates You is a long running set of rhythm game mods for In The Groove made to be harder and visually different than any other rhythm game experience available (WinDEU). It's designed around a single song having several different visual patterns and attacks that are added onto the base gameplay of hitting arrows with the correct timing. It influenced our combat system where a single enemy has several different attack patterns that are accomplished through rhythm gameplay. However, WinDEU designs his beatmaps to have a very high skill required to pass them and are not accessible for a casual audience. Our battle system was made to have the same qualities as WinDEU’s maps but aimed for a lower skilled group of people who may or may not have prior experience with rhythm games.



Undertale by Toby Fox has a unique system by which all enemies in the game do not have to be killed and instead can be spared via an alternate win condition. The ability to choose who you want to kill or avoiding killing all together makes each encounter have a bit more importance behind it and introduces a bit of emotional attachment to enemies knowing you could've easily let it live (Fox). We integrated this by giving every enemy encounter in our game an alternate win condition to not kill it, as well as integrating a permanence to our world where every enemy you kill leave behind a permanent corpse so that when you leave an area and come back you are reminded of the fact you could of spared the enemy.



Outside of previous games, much of our inspiration came from our travels across Japan with certain areas meant to be direct parallels to real life equivalents. Our Cherry Blossom Forest area was designed to be reminiscent of Ueno Park, by including two major areas of the park in the game design: the paths and the lake shrine. The paths are littered with cherry blossom petals and petals still continue to fall while the lake areas have many benches and Torii Gates showing off the mix between park and shrine.

Our Train System, which is another of the major focuses of the game, is also inspired by our trips to Japan. The direction of the trains and their style are modeled after the Yamanote Line in Tokyo while our World's End Train Station takes inspiration in layout to the Ikebukuro Station by including various shops and an area like a park scattered around the train station general area. There are many more examples in our game that mimic Japanese architecture or areas from different aspects of Japanese culture. Some of these have our own twist such as the Shirahebi Shrine area in the game that features a favela shrine that combines aspects of Shinto architecture with the favela style of South America

Tools

Unity

The Unity Game Engine (www.unity3d.com) is a freeware 2D and 3D game development engine that uses C# or Javascript as its primary programming language. It uses DirectX as its graphics library and also contains the Mechanim System, which is an animation tool geared towards designers and programmers used to create custom in-engine animations without the need of an artist (Unity). Unity is an engine our team has used before. We chose it over other available engines like Game Maker Studio (http://www.yoyogames.com/gamemaker), Unreal Engine (https://www.unrealengine.com), etc. to develop Last Hymn primarily because of its reliance on the C# language, which the majority of us use outside of game development projects, as well as the Mechanim System. This system allowed everyone to produce art assets and construct features like transitions, walk cycles, and other integral animations that are not possible in any other commercially available product.

Paint.net

Considering the art style that the team wanted to achieve, it was determined early on that more advanced art programs such as Photoshop are both unfamiliar and provide a user-interface through which it is challenging to develop low fidelity pixel art, and therefore unsuited to achieve the artistic benchmarks we wanted for our game. However, a very basic program like the default paint application that comes with Microsoft operating systems is not feature rich enough for our goals. Therefore it was decided after examining many popular art programs that Paint.net (www.getpaint.net) would be the optimal tool for the project. The program is not only free, but it boasts a very large user-base and community. These groups not only maintain the project and keep it updated, but also have created a vast array of easy to use plugins as well, expanding the already solid suite of features the programs includes. This default suite alone covered the main features we needed, namely image layers and advanced selection tools such as a magic wand.

Spriter

Animation is normally a major roadblock for many game development projects. Like any art-form incorporated into software projects, animation deals with a highly specialized and technical discipline, which traditional is separated from the engineering disciplines. Consequently, many animation production workflows have been designed to streamline the task and make it more scalable beyond the traditional team of draftsman constructing scenes by hand. One of these many innovations is what’s commonly known as “puppet animation” wherein 3D animation principles utilizing bones are adapted for the 2D style of animation. While this method is quickly growing in usage, specialized programs for the method are still young and in most cases costly. Considering these constraints, we discovered that Spriter (https://brashmonkey.com/), a puppet animation program which boasts an impressive feature list in its free version, had enough functionality for the art-style we wanted to achieve while also being fast to learn without requiring significant financial investment.

Trello

Our team utilized an Agile-based team structure using week long sprints to complete the project in the most efficient way possible. An Agile team revolves around three aspects of team composition. First, the team has to as a whole make up all the skills needed with each person being a generalist who can take on tasks not traditionally associated with their role. For example, a programmer may make the music, some art, and do level design in addition to programming. This allows for team members to rapidly switch between assignments based off of need and causes fewer blocks on an item. Secondly, Agile teams focus on rapid iteration. Theoretically Last Hymn should be playable after every sprint and is made to be improved upon constantly instead of a standard approach where a product may not work until part way through development. Lastly, the schedule and people need to be flexible, meaning that they could be moved off their current task at any moment to assist on another issue or work on something else in order to facilitate rapid development. In order to properly manage this approach we used Trello which is a website where all members of the project can create lists, tasks, and upload content (www.trello.com). Using Trello,e created a list for each of our major milestones. Each milestone has its associated tasks required for that version to be feature complete. Weekly, we would assign members to tasks based on skillset, desire, and importance to that milestone. This allowed our team to not only identify what we have done and what we are doing but what we will do and the steps we took to accomplish each of these.

Github

For version control of documents, the game itself, and other files we used a private repository on Github. Github is a source control service that features several collaboration features such as bug tracking, feature requests, task management, and wikis for every project (https://github.com/). We chose to use Github mainly for the ease of use of Github for Windows and the fact that we had prior experience using Git Shell prior to this project. It is also a handy tool for ensuring that if someone accidently introduced bugs into the game the team can identify when it was introduced and in the worst-case scenario rollback the change to when it previously worked.

Technical Details

In tackling the many issues which Last Hymn posed as a project, we utilized some pre-existing tools to help us in certain cases. One example of this was in the music analyzer tool RhythmTool (https://www.assetstore.unity3d.com/en/#!/content/15679) for battle attack generation. We felt it wasn't necessary to "reinvent the wheel" by delving into the complexity of sound analyzing software on our own and so used the help of a utility which allowed us to focus more on the content of the game rather than spend weeks of time learning about the technical procedure for sound analyzing. In another example, Bezier pathing within the game was constructed via an online tutorial (http://catlikecoding.com/unity/tutorials/curves-and-splines/). While not an original idea of the team, the implementation and manipulation of the Bezier functionality was unique to our team. All other systems, art, and code were original creations by our team.

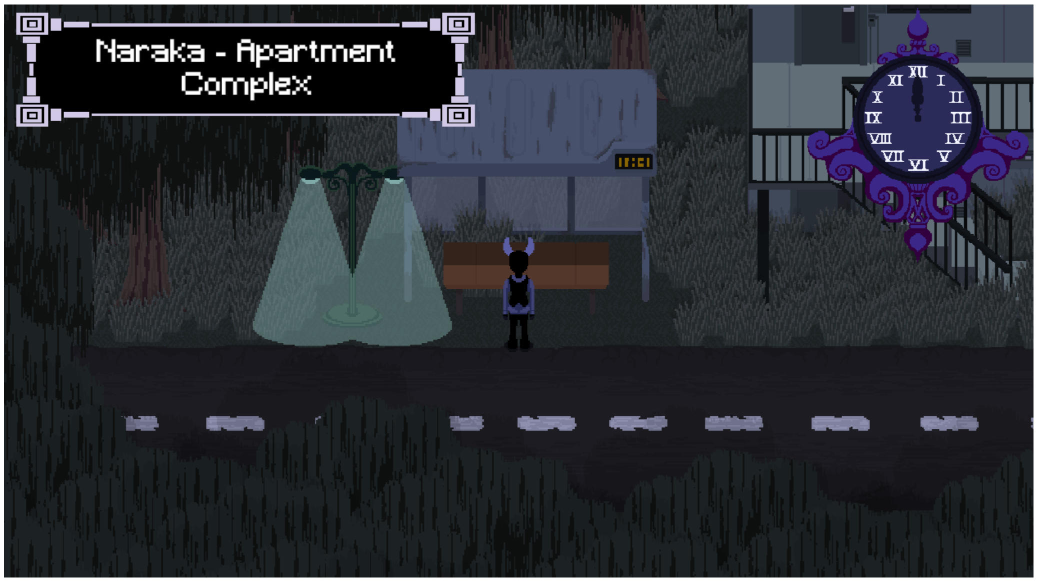
As for the team exclusive creations, many complex systems were created to control the various aspects of the game. The menu system worked off of a delegate based, dynamically generating list of intractable menu options resulting in an overall simplistic and rapidly expandable game component. Another example of a uniquely designed system is seen in the Crystal Flowers Cavern puzzle mechanics. Detailed more heavily later on in this report, the puzzle mechanics worked off of a cause effect reaction that would automatically interact with various objects in an easily programmable chain reaction system which allowed for quick and unique puzzle creation. One final example of technical achievement within Last Hymn can be seen in the time delegate system which, upon loading into any area in the game, automatically generates a list of timed events that automatically trigger based on the current time of day. This functionality is used in many ways including scheduling train arrivals and controlling the end of the world event.

Design and Implementation

General

Design

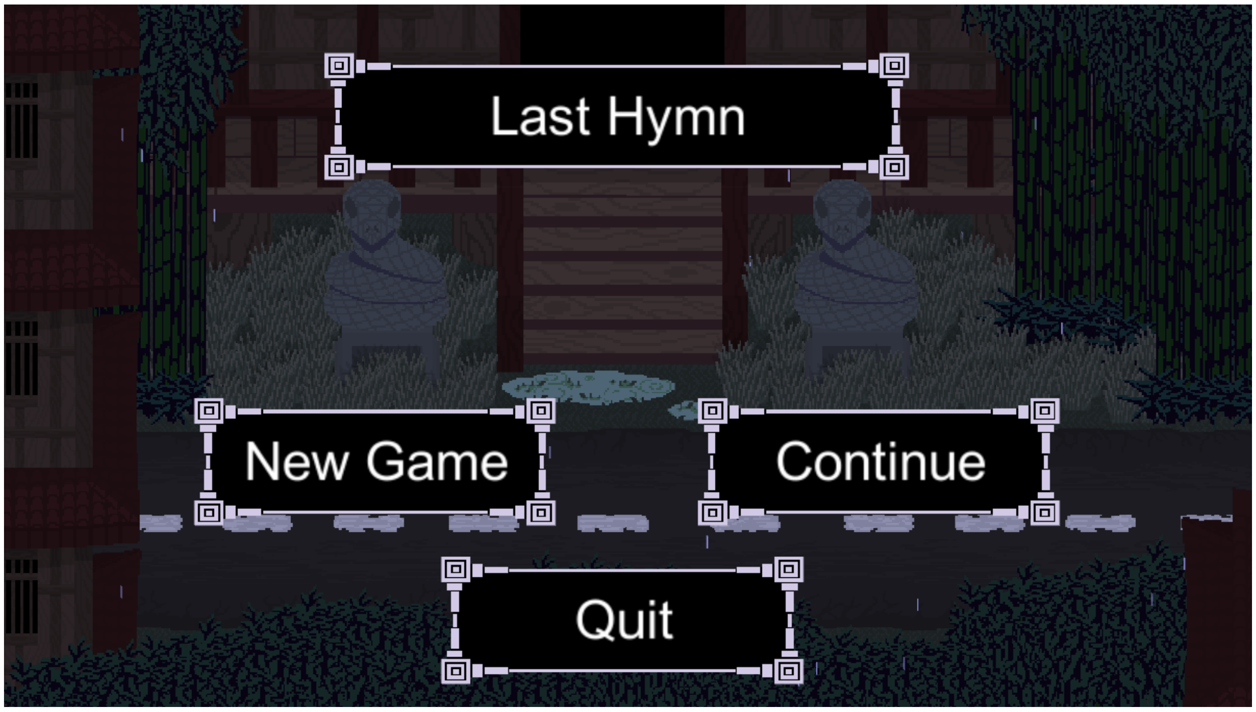
The goal of the game was to produce an RPG world that a player could freely explore and encounter new areas and events without any set progression path. A heavy emphasis was placed on using a “fantasy chic” style which is one that takes traditional and modern elements and interweaves them into the same environment.



In the above screenshot, a modern bus stop and apartment complex are placed in an overgrown forest environment. This area is blended into the other areas of the game in the Forest seamlessly to have it so two seemingly opposite environments, a modern apartment complex and a Shinto inspired Forest, can coexist. The team’s goal is to take these areas and focus on the development of a location’s mood and atmosphere.

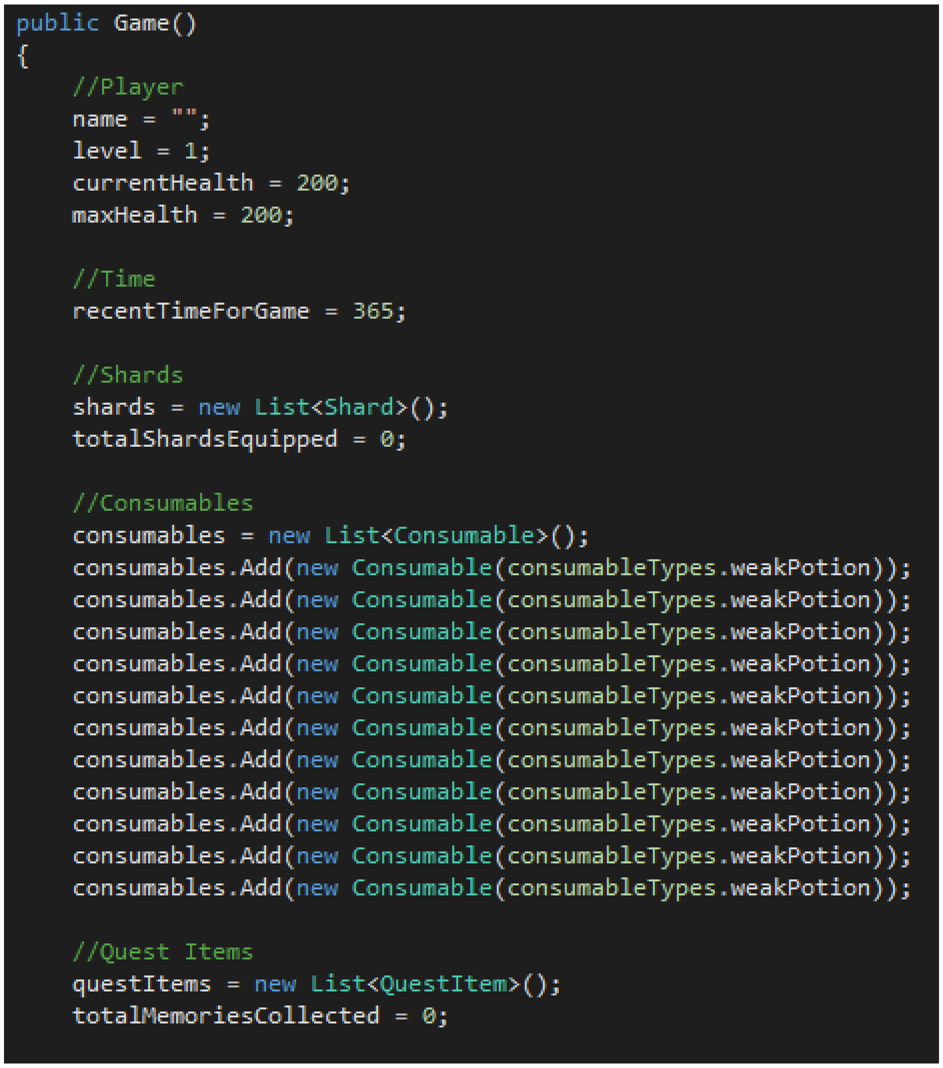
The story of the game takes place over the course of one day that is repeated over and over as the player tries to solve the mystery of and prevent their own death at the end of the day. This is done by collecting five essences by completing five different quest lines that are spread throughout the world. The player then has to deliver them to Ariadne at the lowest depths of the Crystal Flowers Cavern. The game begins with the player arriving on a train at the World’s End Train Station where they are greeted by a mysterious figure and guided to the Forest where they witness the end of the world and find themselves back at the train station before they left for the Forest. With only a limited amount of time per cycle of the world, the player must constantly weigh the opportunity cost of each decision, and only with careful thought, conviction, and tenacity will the player find a conclusion from the never ending cycle of rebirth.

Players will be taken on a journey through an eclectic world that mixes together modern and traditional fantasy elements in order to form a unique and original experience that seeks to draw in the player as they unravel the mysteries that surround them. Reinforcing the world and its themes, Last Hymn blends a JRPG world and story with procedurally generated rhythm game elements. Players will explore the world and grow their abilities through collecting Essences through their story choices, making each playthrough unique to the decisions of that player and tying together Last Hymn's story, gameplay, and lore into a single cohesive experience where each aspect seeks to reinforce the other to create a memorable experience for the player.



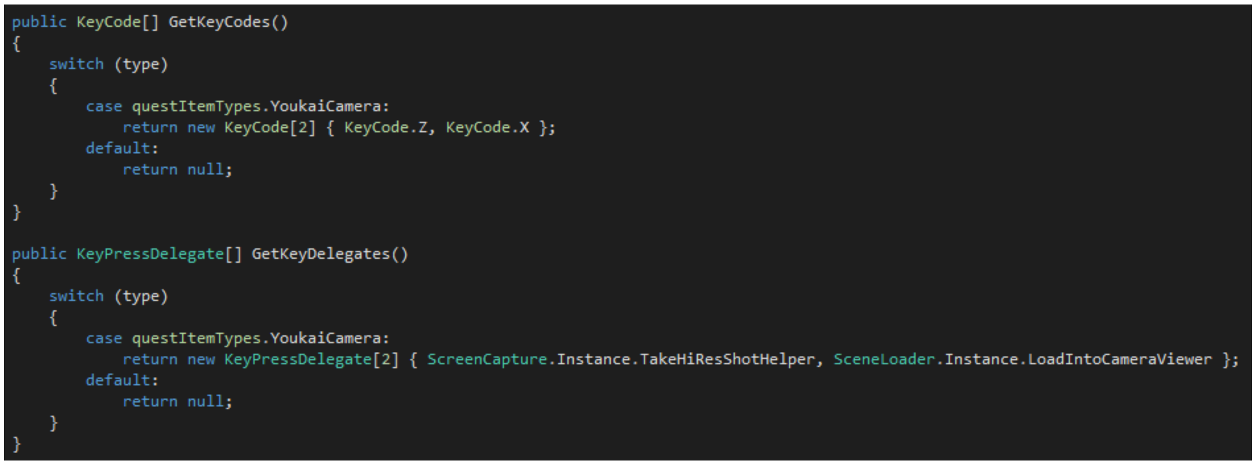
Game

Any time the player enters the game world a corresponding Game object is created to correspond to that game session. This Game object is singleton that contains all the information that needs to be saved about the state space of the game. It holds information on the player’s inventory, name, and health as well as the state of enemies in each of the game’s different scenes. We also implemented a trigger system to keep track of what stage a player is at on each of the game’s main quest lines.



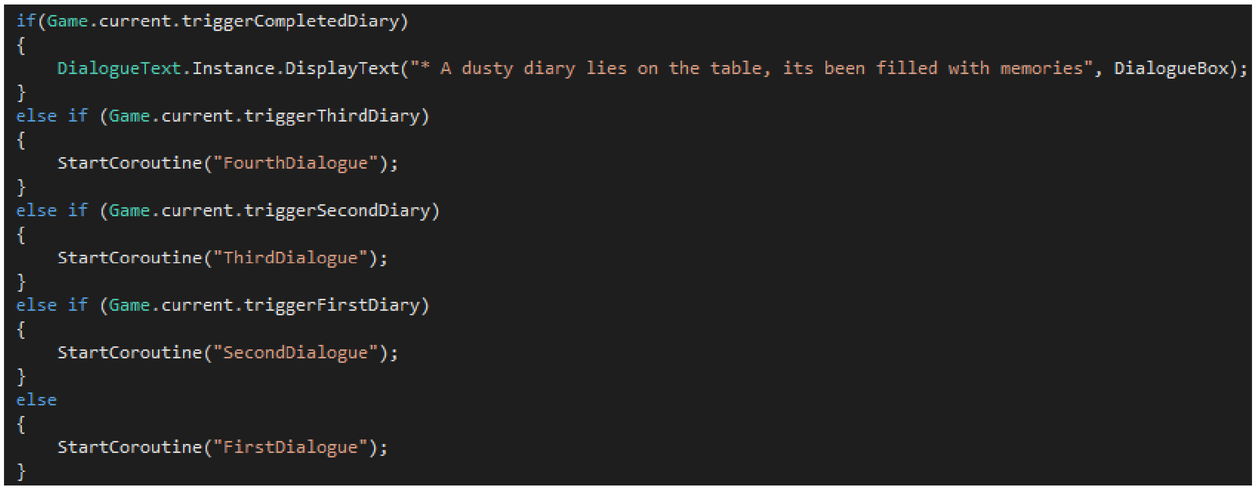
```
public Game()
{
    //Player
    name = "";
    level = 1;
    currentHealth = 200;
    maxHealth = 200;

    //Time
    recentTimeForGame = 365;

    //Shards
    shards = new List<Shard>();
    totalShardsEquipped = 0;

    //Consumables
    consumables = new List<Consumable>();
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));
    consumables.Add(new Consumable(consumableTypes.weakPotion));

    //Quest Items
    questItems = new List<QuestItem>();
    totalMemoriesCollected = 0;
```

Also contained in this class are several subclasses that are used in the Battle Scenes and Menu System: Shard, Consumable, and QuestItem. Each of these contain an enumerable that lists all the possible types of each item and a basic description for each of them which are used as part of the Menu.There are also KeyPressDelegate methods in these classes that are triggered when the player selects each of these items on the Menu screen and the associated outcome with each item type.



```
public KeyCode[] GetKeyCodes()
{
    switch (type)
    {
        case questItemTypes.YoukaiCamera:
            return new KeyCode[2] { KeyCode.Z, KeyCode.X };
        default:
            return null;
    }
}

public KeyPressDelegate[] GetKeyDelegates()
{
    switch (type)
    {
        case questItemTypes.YoukaiCamera:
            return new KeyPressDelegate[2] { ScreenCapture.Instance.TakeHiResShotHelper, SceneLoader.Instance.LoadIntoCameraViewer };
        default:
            return null;
    }
}
```

The main Game class and the subclasses can all be referenced in any other script and have their methods called through the usage of Game.current. This is the instance of the current version of the game that the player is playing through and is used as a central hub for retrieving and manipulating data of the world in order to reduce errors and to keep a centralized repository of information.

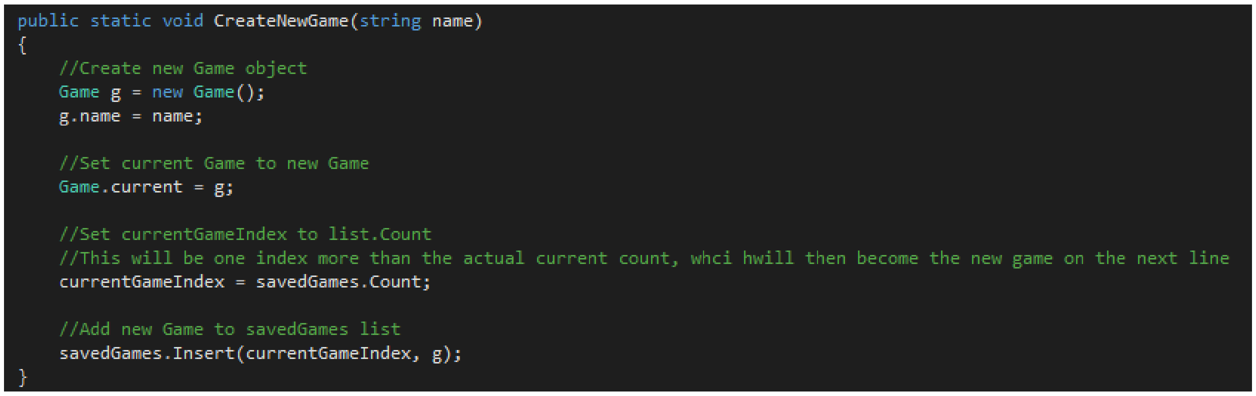


```
if(Game.current.triggerCompletedDiary)
{
    DialogueText.Instance.DisplayText("* A dusty diary lies on the table, its been filled with memories", DialogueBox);
}
else if (Game.current.triggerThirdDiary)
{
    StartCoroutine("FourthDialogue");
}
else if (Game.current.triggerSecondDiary)
{
    StartCoroutine("ThirdDialogue");
}
else if (Game.current.triggerFirstDiary)
{
    StartCoroutine("SecondDialogue");
}
else
{
    StartCoroutine("FirstDialogue");
}
```

SaveLoad

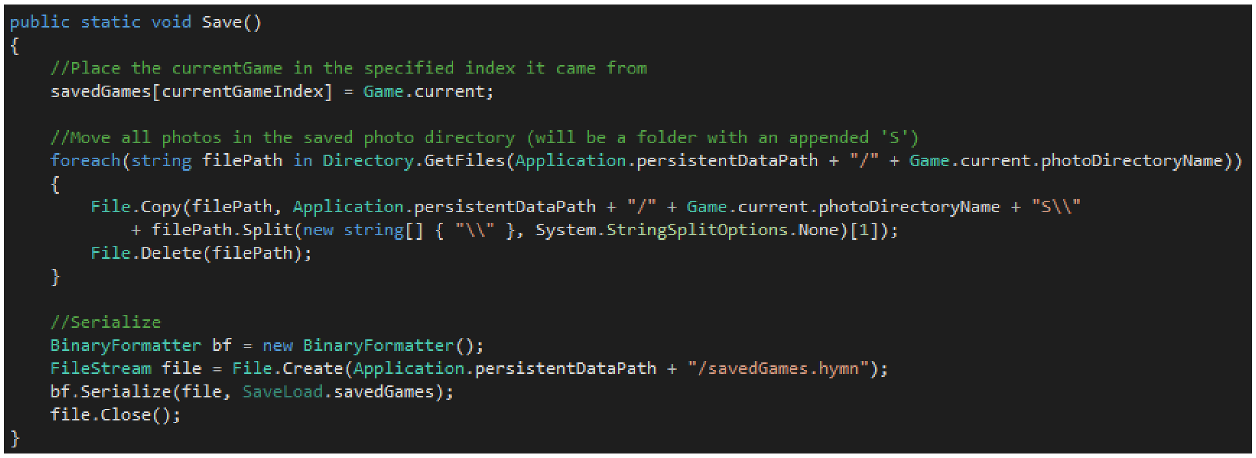
As a team, we desired to have a minimal amount of UI Elements as a way to keep the player immersed in the experience of the game and the world itself. We started this off on the Title Screen where we limit the choices available to the player to only being able to start a new game, continue a previous save file, or quit.

When a player chooses to start a new game, we need to generate a new Game object for them that has all of the game's triggers, the player's inventory, and other key variables set to their default values. We store all of the Game objects for multiple save files in a list so they can be referenced later.



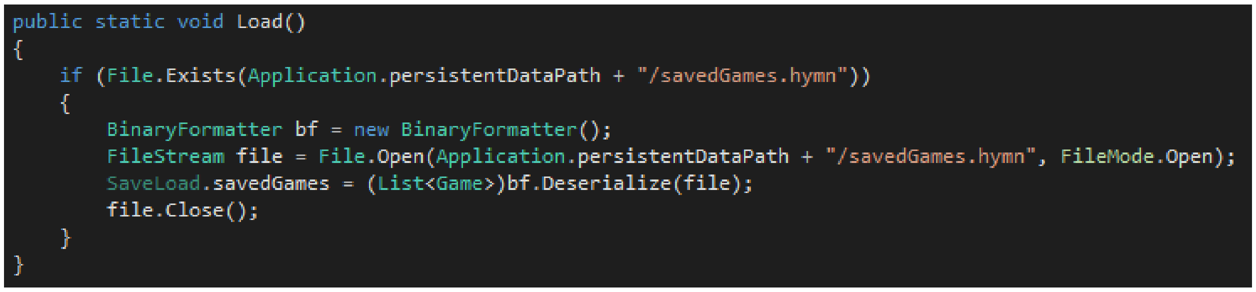
```
public static void CreateNewGame(string name)
{
    //Create new Game object
    Game g = new Game();
    g.name = name;

    //Set current Game to new Game
    Game.current = g;

    //Set currentGameIndex to list.Count
    //This will be one index more than the actual current count, whci hwill then become the new game on the next line
    currentGameIndex = savedGames.Count;

    //Add new Game to savedGames list
    savedGames.Insert(currentGameIndex, g);
}
```

Anytime in the process of playing the game the player decides to save, we reference the Save method, which takes the current Game object and overwrites itself in the list of saved game files. Our list of Game objects is serialized and saved to the AppLocal of a computer so it can then be loaded from later. As part of one of the game's major questlines we have an in-game camera that takes and saves photos to the computer. As part of the save functionality we have to move the photo from a temporary directory to a permanent one so that a player who takes a picture then quits without saving won't have that picture as part of their game as well as making it so that a player that does save the game can access the photo.



```
public static void Save()
{
    //Place the currentGame in the specified index it came from
    savedGames[currentGameIndex] = Game.current;

    //Move all photos in the saved photo directory (will be a folder with an appended 'S')
    foreach(string filePath in Directory.GetFiles(Application.persistentDataPath + "/" + Game.current.photoDirectoryName))
    {
        File.Copy(filePath, Application.persistentDataPath + "/" + Game.current.photoDirectoryName + "S\\"
            + filePath.Split(new string[] { "\\" }, System.StringSplitOptions.None)[1]);
        File.Delete(filePath);
    }

    //Serialize
    BinaryFormatter bf = new BinaryFormatter();
    FileStream file = File.Create(Application.persistentDataPath + "/savedGames.hymn");
    bf.Serialize(file, SaveLoad.savedGames);
    file.Close();
}
```

The last part of our SaveLoad.cs system is the loading functionality which checks if the saved list of games exists and if it does it deserializes it, setting the current list of saved games equal to what is found in the file. Since SaveLoad is a singleton it is able to be used throughout the game without an object reference and ensuring we only have one current game and one list of saved games.

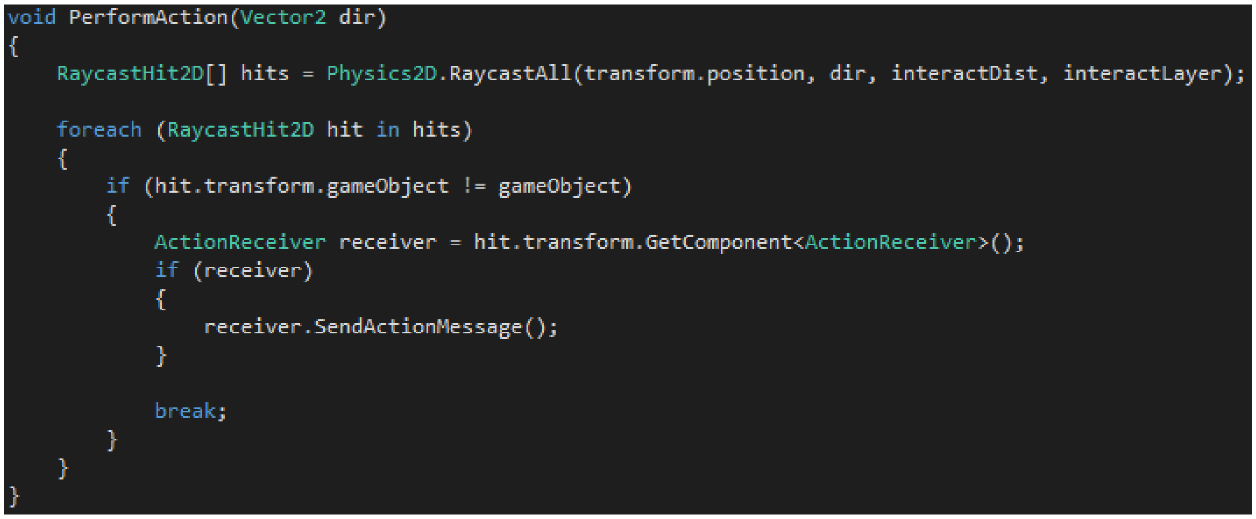


```
public static void Load()
{
    if (File.Exists(Application.persistentDataPath + "/savedGames.hymn"))
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file = File.Open(Application.persistentDataPath + "/savedGames.hymn", FileMode.Open);
        SaveLoad.savedGames = (List<Game>)bf.Deserialize(file);
        file.Close();
    }
}
```

Action Scripts

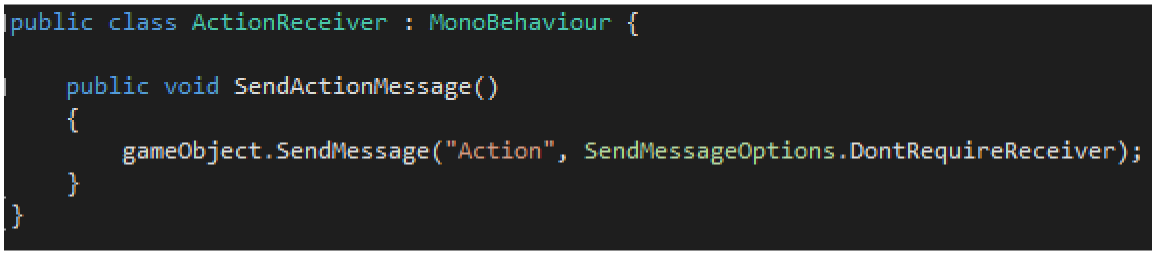
In Last Hymn a majority of the RPG sections of the game are spent interacting with objects, characters, and the environment. In order to facilitate different types of interactions, we created a general framework using a Caller and Receiver System.

In this system, the player serves as a caller and whenever they hit the Z Key to interact with objects in the RPG sections of the game the PerformAction method is called on their character.



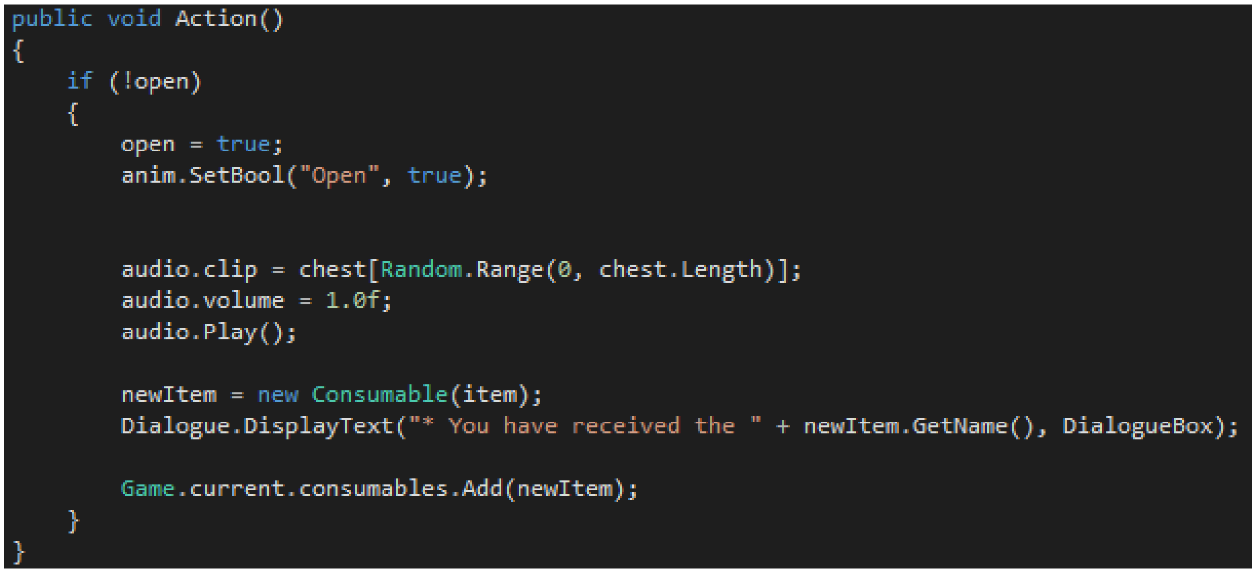
```
void PerformAction(Vector2 dir)
{
    RaycastHit2D[] hits = Physics2D.RaycastAll(transform.position, dir, interactDist, interactLayer);

    foreach (RaycastHit2D hit in hits)
    {
        if (hit.transform.gameObject != gameObject)
        {
            ActionReceiver receiver = hit.transform.GetComponent<ActionReceiver>();
            if (receiver)
            {
                receiver.SendActionMessage();
            }

            break;
        }
    }
}
```

It performs a Raycast from the center of the character and goes out a small distance from the player in the direction they are facing. We then ensure the object returned from the Raycast is not itself and then we call SendActionMessage on that object the Raycast hit.



```
public class ActionReceiver : MonoBehaviour {

    public void SendActionMessage()
    {
        gameObject.SendMessage("Action", SendMessageOptions.DontRequireReceiver);
    }
}
```

Every object in the game that can be interacted with has the ActionReceiver script which then calls an “Action” method on the same object. We then can create different types of Action methods all of which can be called by this Caller and Receiver setup but their outcomes can be different. For example, a chest after being interacted with by the player will then have the following Action Method executed:



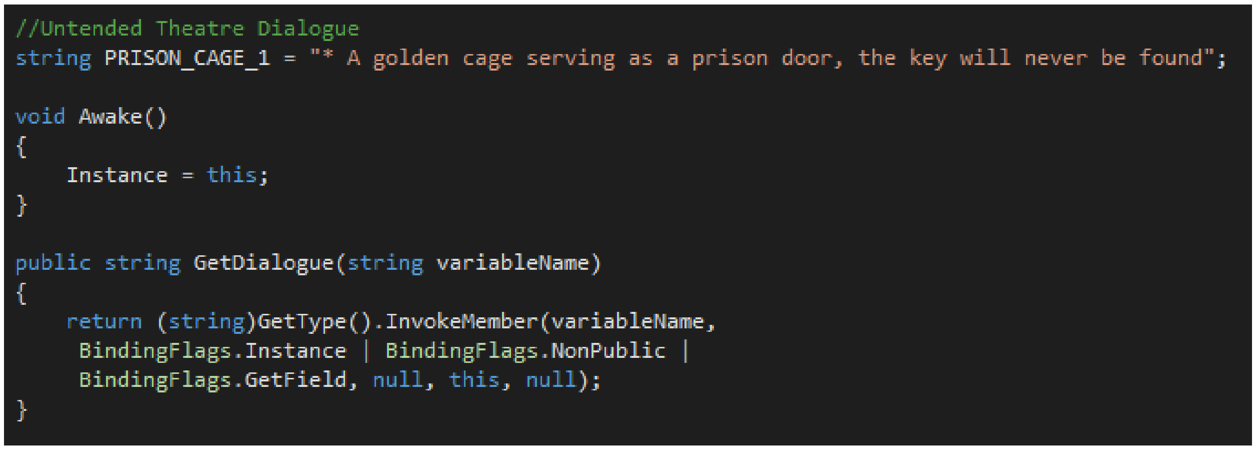
```
public void Action()
{
    if (!open)
    {
        open = true;
        anim.SetBool("Open", true);


        audio.clip = chest[Random.Range(0, chest.Length)];
        audio.volume = 1.0f;
        audio.Play();

        newItem = new Consumable(item);
        Dialogue.DisplayText("* You have received the " + newItem.GetName(), DialogueBox);

        Game.current.consumables.Add(newItem);
    }
}
```

This Action method plays a sound effect, displays dialogue, and then adds a new item to the player’s inventory. The advantage of this system is that instead of making a new system to interact with enemies we just have to make a new Action method to apply to the enemy and the interaction with it and calls are the exact same.

Dialogue System

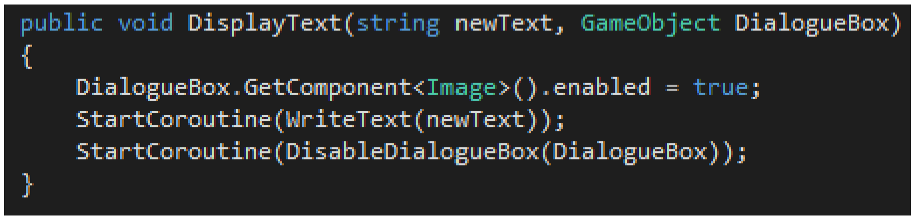
Last Hymn, while not being a story-centric game, does contain hundreds of lines of dialogue which are used for NPC conversation, the player interacting with items, events, etc. In order to create a generic structure to display text we created a Dialogue System with two major components: DialogueRepo.cs and DialogueText.cs.

The DialogueRepo is a singleton class that has private strings that contain a singular line of text. It also has the GetDialogue method which is a function used in other classes to access the line of text through the variable's name via reflection. This was deliberately done to aggregate nearly all lines of text into a single file to allow for easy editing. For example, if a chest opens it will always display CHEST_OPEN dialogue instead of specifying for each chest individually what it needs to print which makes typos harder to eliminate and editing of the game's dialogue a harder endeavor.

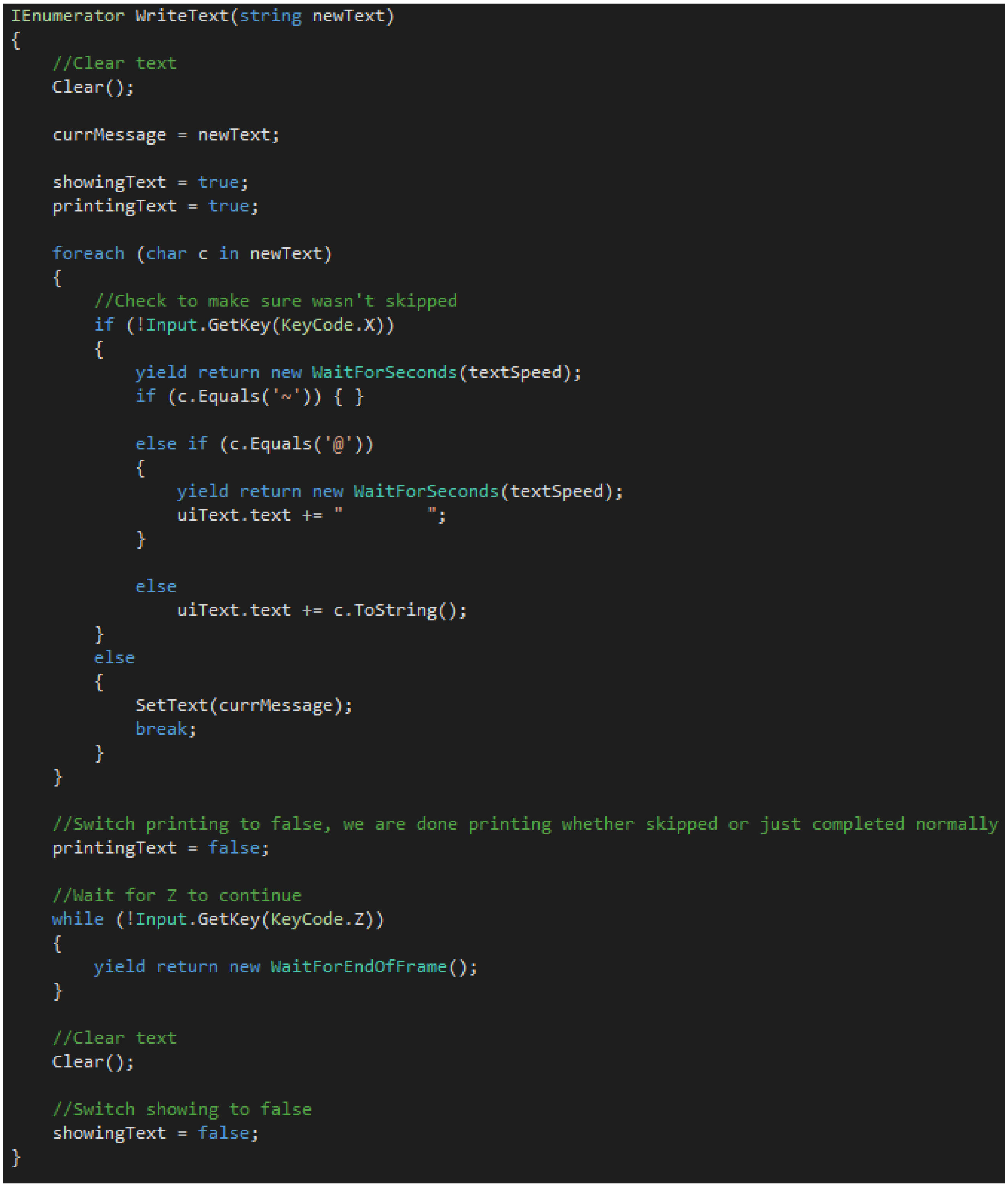


```
//Untended Theatre Dialogue
string PRISON_CAGE_1 = "* A golden cage serving as a prison door, the key will never be found";

void Awake()
{
    Instance = this;
}

public string GetDialogue(string variableName)
{
    return (string)GetType().InvokeMember(variableName,
     BindingFlags.Instance | BindingFlags.NonPublic |
     BindingFlags.GetField, null, this, null);
}
```

DialogueText.cs is the class that handles the text input received from the DialogueRepo and contains functions to support different ways of displaying text. Most text in the game is printed via the Display Textmethod which will display the box that serves as the background for the dialogue, print out each character individually to give a typewriter impression and make it visually more appealing, and disable the dialogue box after the player has hit the Z key. However, we've added functionality to skip the printing and display the full text through the usage of the X key. This was done in order to allow players to skip dialogue they had no interest in or had read before without forcing them to re-read content.



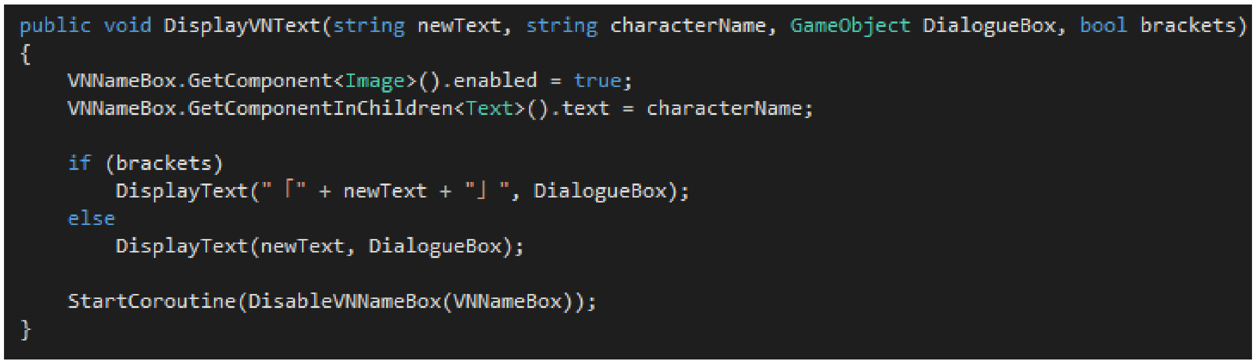
```
public void DisplayText(string newText, GameObject DialogueBox)
{
    DialogueBox.GetComponent<Image>().enabled = true;
    StartCoroutine(WriteText(newText));
    StartCoroutine(DisableDialogueBox(DialogueBox));
}
```



```
IEnumerator WriteText(string newText)
{
    //Clear text
    Clear();

    currMessage = newText;

    showingText = true;
    printingText = true;

    foreach (char c in newText)
    {
        //Check to make sure wasn't skipped
        if (!Input.GetKey(KeyCode.X))
        {
            yield return new WaitForSeconds(textSpeed);
            if (c.Equals('~')) { }

            else if (c.Equals('@'))
            {
                yield return new WaitForSeconds(textSpeed);
                uiText.text += "        ";
            }

            else
                uiText.text += c.ToString();
        }
        else
        {
            SetText(currMessage);
            break;
        }
    }

    //Switch printing to false, we are done printing whether skipped or just completed normally
    printingText = false;

    //Wait for Z to continue
    while (!Input.GetKey(KeyCode.Z))
    {
        yield return new WaitForEndOfFrame();
    }

    //Clear text
    Clear();

    //Switch showing to false
    showingText = false;
}
```

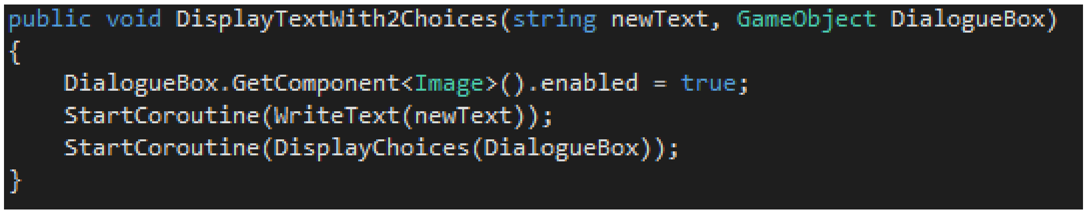
However, in addition to displaying text we also had to handle Dialogue that had choices associated with it and NPC conversations.

NPC conversations require an additional box to be displayed with an NPC’s name in it so it will be easy for a player to identify who they are talking to as well as who is speaking with each line of dialogue. As a stylistic choice we have two ways of writing dialogue. If a character is speaking, we wrap the dialogue in brackets (e.g.「Hello World!」). All text inside of brackets are spoken words and have punctuation, while character actions or third-person observations do not have punctuation and are preceded by an asterisk (e.g. * You obtained the Shard of Death). This distinction is made so a player never confuses an action with what is said.



```
public void DisplayVNText(string newText, string characterName, GameObject DialogueBox, bool brackets)
{
    VNNameBox.GetComponent<Image>().enabled = true;
    VNNameBox.GetComponentInChildren<Text>().text = characterName;

    if (brackets)
        DisplayText("「" + newText + "」", DialogueBox);
    else
        DisplayText(newText, DialogueBox);

    StartCoroutine(DisableVNNameBox(VNNameBox));
}
```

There are also instances where a dialogue may lead to two choices where the player has to decide which option to choose before proceeding. The display of these boxes and dialogue is handled by the DialogueText while the options and the results of those actions are handled by Action scripts.



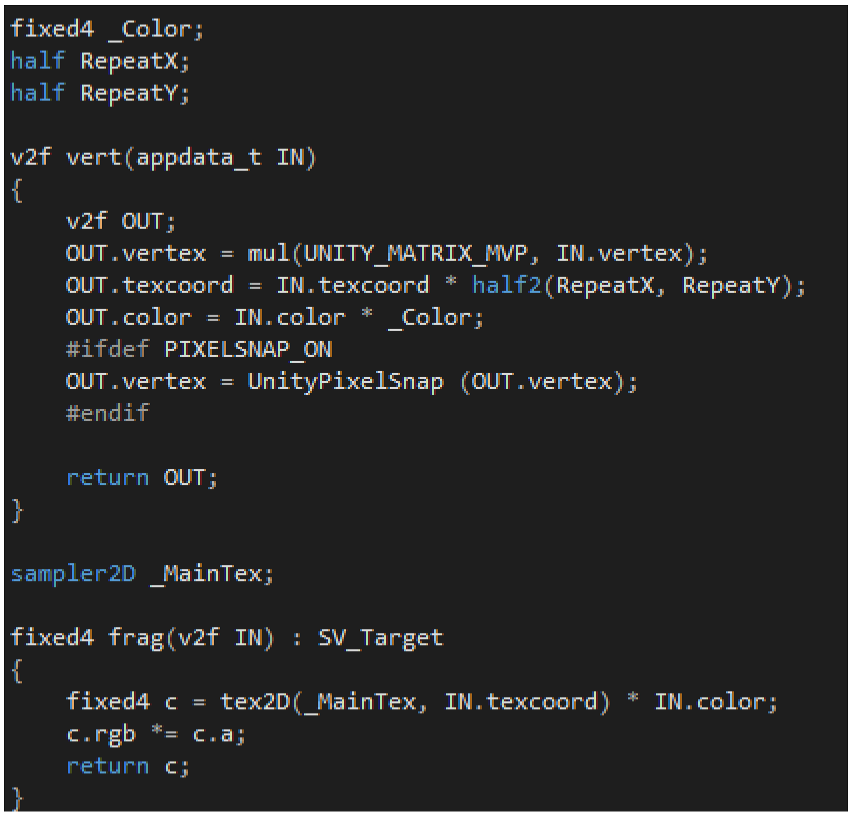
```
public void DisplayTextWith2Choices(string newText, GameObject DialogueBox)
{
    DialogueBox.GetComponent<Image>().enabled = true;
    StartCoroutine(WriteText(newText));
    StartCoroutine(DisplayChoices(DialogueBox));
}
```

Nearly all variations of the game’s text will follow one of the three base cases of DisplayText, DisplayVNText, and DisplayTextWith2Choices with parts of it slightly manipulated. For example, all text displayed on a menu, which doesn’t use a typewriter to print the text is a modified version of DisplayText that functions as if the player immediately pressed x to skip printing the text.

Shaders

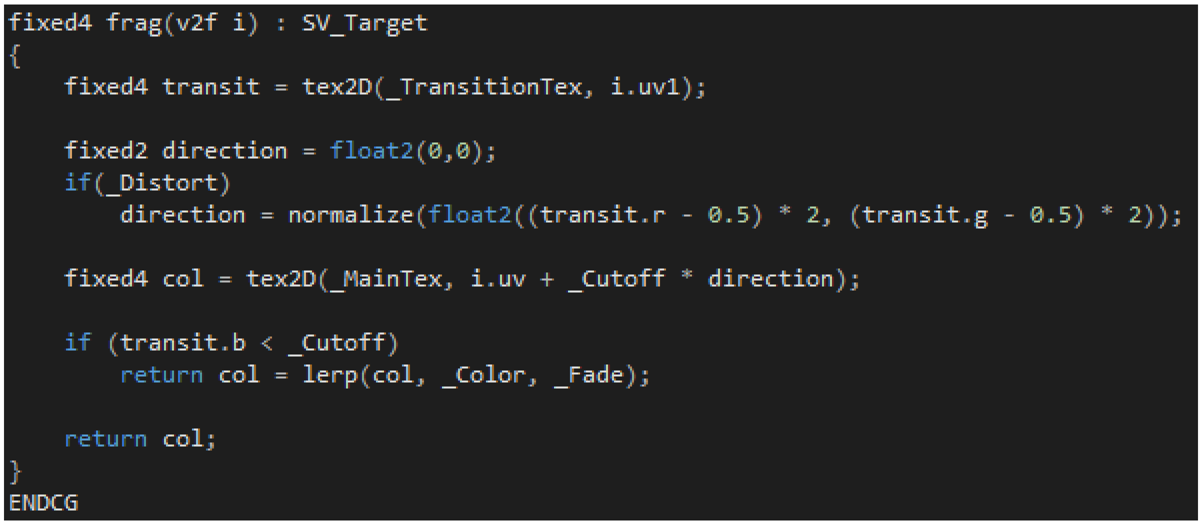
Unity by default does not include a tiling system for sprites and instead only has native support for scaling a sprite. For large areas like the Forest where we have a large grid of grass tiles that form the background, it would be difficult to manually duplicate a sprite thousands of times to create a grid for the player to walk on. To address this problem, we implemented a shader to handle the rendering for us and only use a single game object.

We used a shader that takes in a sprite and repeats itself given an input of how you want to scale in the x and y directions. This effect requires very little GPU processing but is effective in quickly rendering large areas of content.



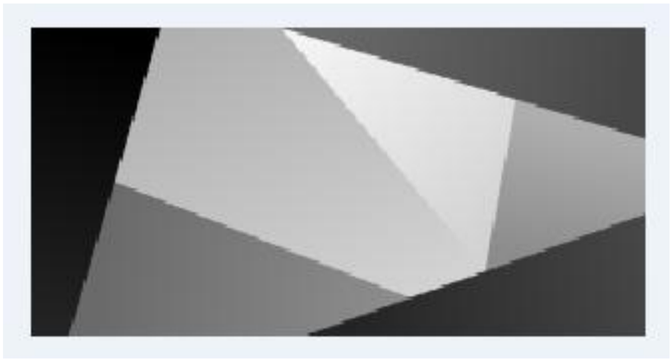
```
fixed4 _Color;
half RepeatX;
half RepeatY;

v2f vert(appdata_t IN)
{
    v2f OUT;
    OUT.vertex = mul(UNITY_MATRIX_MVP, IN.vertex);
    OUT.texcoord = IN.texcoord * half2(RepeatX, RepeatY);
    OUT.color = IN.color * _Color;
    #ifdef PIXELSNAP_ON
    OUT.vertex = UnityPixelSnap (OUT.vertex);
    #endif

    return OUT;
}

sampler2D _MainTex;

fixed4 frag(v2f IN) : SV_Target
{
    fixed4 c = tex2D(_MainTex, IN.texcoord) * IN.color;
    c.rgb *= c.a;
    return c;
}
```

We also have implemented a Battle Transition System through the usage of shaders. To create a battle transition normally would require a custom animation to be made, which takes time to create and more time to modify to precisely fit our vision of the game. Instead of taking that route we use a shader that take in an input of a grey-scale sprite and will process it and render a transition onto the camera based off of it.



```
fixed4 frag(v2f i) : SV_Target
{
    fixed4 transit = tex2D(_TransitionTex, i.uv1);

    fixed2 direction = float2(0,0);
    if(_Distort)
        direction = normalize(float2((transit.r - 0.5) * 2, (transit.g - 0.5) * 2));

    fixed4 col = tex2D(_MainTex, i.uv + _Cutoff * direction);

    if (transit.b < _Cutoff)
        return col = lerp(col, _Color, _Fade);

    return col;
}
ENDCG
```

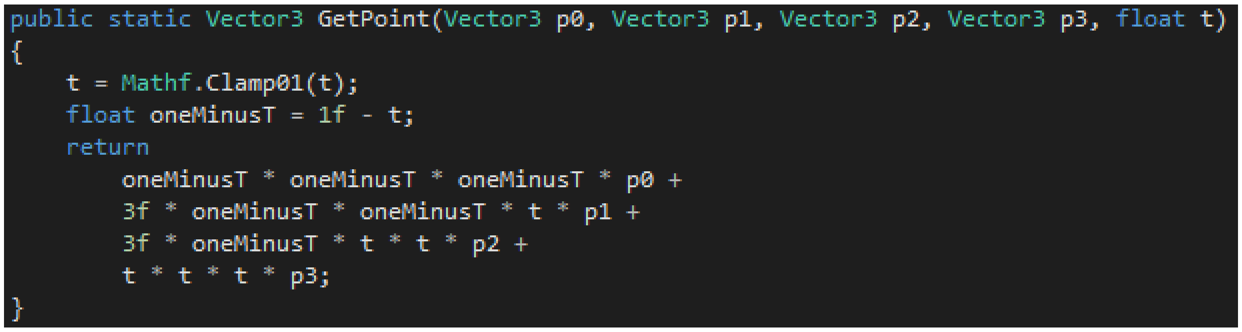
We lerp between the solid areas going from black to white, which creates a smooth transition that is visually appealing but only requires our team to spend a minute to create a new black and white image compared to hours of custom animations. This shader is also reusable for all other types of transition functions which a custom animation would not account for.



Paths

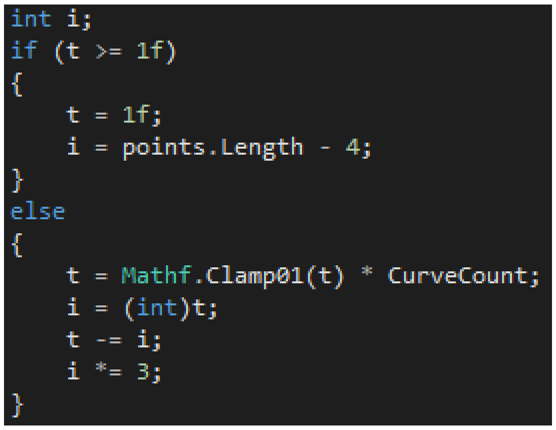
Paths are used in various areas throughout the game to allow fluid motion for objects. Arrows, enemies, and camera cut scenes are all controlled by paths. Paths work through a linear interpolation between predefined points that result in smooth waveforms across distances.

A key feature when using paths is the GetPoint method, which takes in predefined points and then linearly interpolates between them to determine the physical point of where the path should be.



```
public static Vector3 GetPoint(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3, float t)
{
    t = Mathf.Clamp01(t);
    float oneMinusT = 1f - t;
    return
        oneMinusT * oneMinusT * oneMinusT * p0 +
        3f * oneMinusT * oneMinusT * t * p1 +
        3f * oneMinusT * t * t * p2 +
        t * t * t * p3;
}
```

All Paths used within the game are fed in a “0 to 1” value which correlates to a position on the path. When using a Path with multiple segments of sets of points, it is important to be able to locate which portion of the Path is needed and which points will correlate to the correct Path section. This code shows the functionality of choosing such points where “i” is the index of the first point needed. Notice how through this clever functionality, any input value maybe be traced back to the set of indices needed.



```
int i;
if (t >= 1f)
{
    t = 1f;
    i = points.Length - 4;
}
else
{
    t = Mathf.Clamp01(t) * CurveCount;
    i = (int)t;
    t -= i;
    i *= 3;
}
```

Scene States and Serialized Objects

A problem that arises when loading and unloading various portions of a game world based on Unity's scene structure is the issue of recording changes that have occurred in previously visited scenes and recreating those changes in a manageable and practical way. The answer for this issue created within Last Hymn is the Serialized Object and Scene State system.

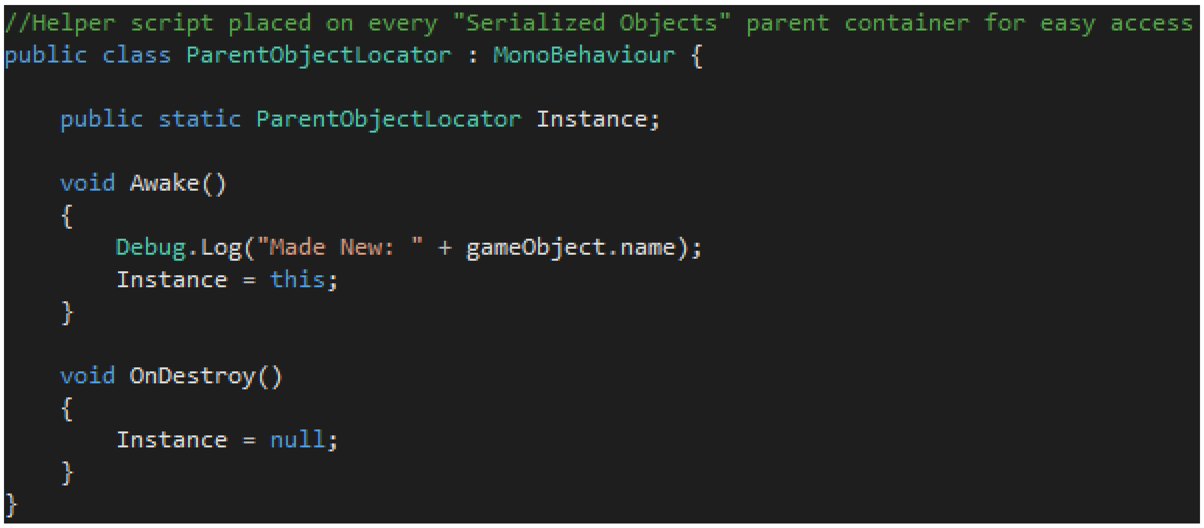
Serialized objects are objects that need their positions and states stored between scenes. The SerializedObject script is a container that is used by many other scripts to lookup information at run time about the object's previous state as well as to handle updating state information when needed.

ObjectStates are the serialized container objects used for storing information on SerializedObjects in a serializable manner. For example, ObjectStates turn non serializable Vector3s into 3 float values when storing a SerializedObject's position.

SceneState objects are simply serializable Dictionaries with (string,ObjectState) key-value pairs. Each SceneState is set up with the expectation that all objects to be serialized within a given scene will be uniquely named so that object names can correlate to the Dictionary key when attempting to find ObjectStates later on.

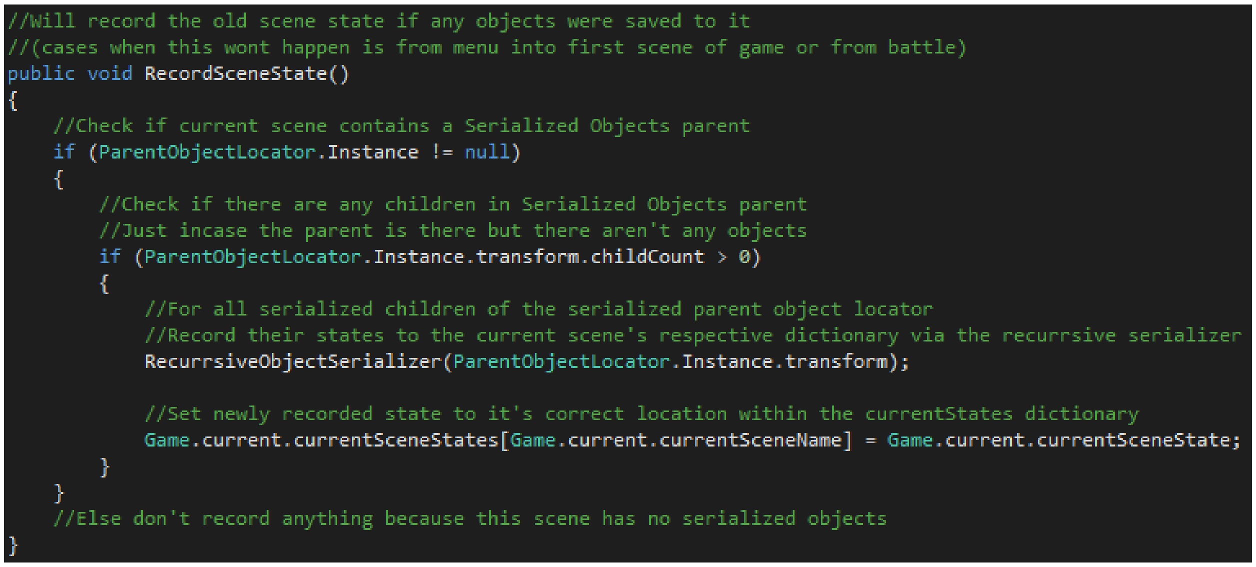
Upon loading into a scene, all SerializedObject scripts attempt to lookup a record of their previous ObjectState in the Game's current SceneState.

Any scene within the game that has some SerializedObject will have a ParentObjectLocator static script attached to an empty object. This empty object is the parent to all objects that need to be serialized and its sole purpose is to be easily locatable for finding the serialized objects within each scene.



```
//Helper script placed on every "Serialized Objects" parent container for easy access
public class ParentObjectLocator : MonoBehaviour {

    public static ParentObjectLocator Instance;

    void Awake()
    {
        Debug.Log("Made New: " + gameObject.name);
        Instance = this;
    }

    void OnDestroy()
    {
        Instance = null;
    }
}
```

Within the LoadIntoScene method of the SceneLoader script, there are two method calls which attempt to store the current scene's serialized object state and load in the new scene's serialized object state. Below, the two save and load methods can be seen.

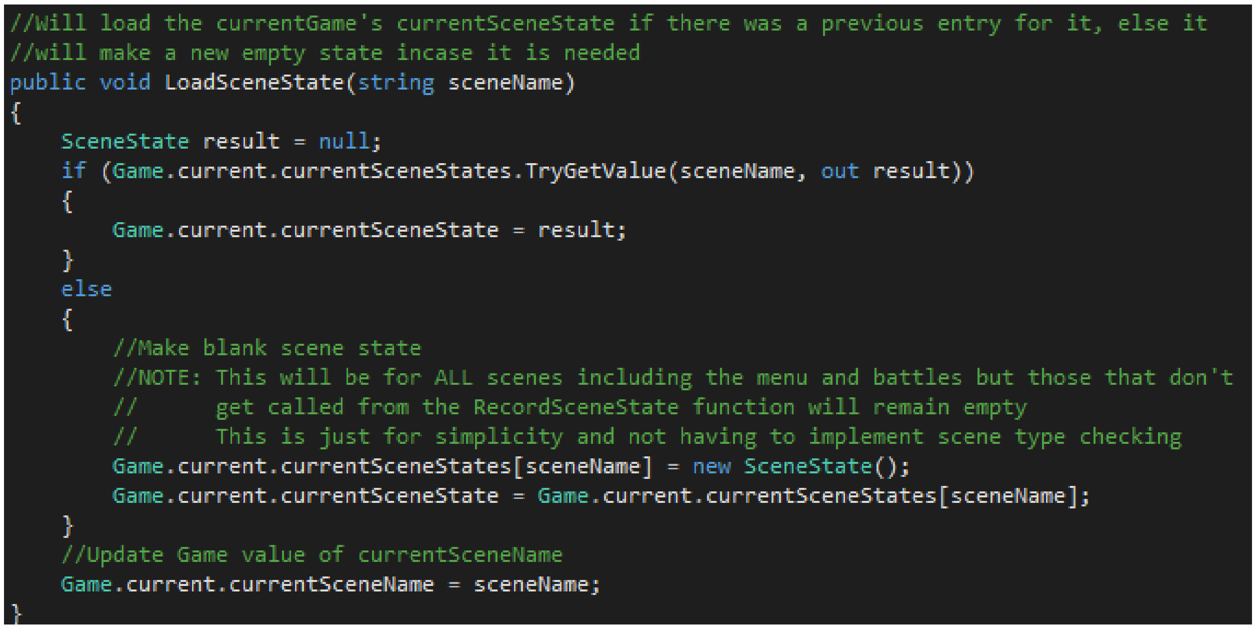


```
//Will record the old scene state if any objects were saved to it
//(cases when this wont happen is from menu into first scene of game or from battle)
public void RecordSceneState()
{
    //Check if current scene contains a Serialized Objects parent
    if (ParentObjectLocator.Instance != null)
    {
        //Check if there are any children in Serialized Objects parent
        //Just incase the parent is there but there aren't any objects
        if (ParentObjectLocator.Instance.transform.childCount > 0)
        {
            //For all serialized children of the serialized parent object locator
            //Record their states to the current scene's respective dictionary via the recurrsive serializer
            RecurrsiveObjectSerializer(ParentObjectLocator.Instance.transform);

            //Set newly recorded state to it's correct location within the currentStates dictionary
            Game.current.currentSceneStates[Game.current.currentSceneName] = Game.current.currentSceneState;
        }
    }
    //Else don't record anything because this scene has no serialized objects
}
```

Note how recording attempts to store the current scene's state to the current Game object's list of states by first checking that there are any objects to serialize in the scene and if so, then updating all serialized objects' info before finally overwriting the old scene state entry to the current one.

The RecursiveSerializer is used to find all SerializedObjects beneath the ParentObjectLocator so that even complex objects with multiple serialized parts can be stored and accessed again successfully.

Loading is much simpler in that all it attempts to do is lookup if there was a scene state already in the current Game object for the new scene, making a new scene if there isn't one already and flagging the scene state if it already exists.



```
//Will load the currentGame's currentSceneState if there was a previous entry for it, else it
//will make a new empty state incase it is needed
public void LoadSceneState(string sceneName)
{
    SceneState result = null;
    if (Game.current.currentSceneStates.TryGetValue(sceneName, out result))
    {
        Game.current.currentSceneState = result;
    }
    else
    {
        //Make blank scene state
        //NOTE: This will be for ALL scenes including the menu and battles but those that don't
        //      get called from the RecordSceneState function will remain empty
        //      This is just for simplicity and not having to implement scene type checking
        Game.current.currentSceneStates[sceneName] = new SceneState();
        Game.current.currentSceneState = Game.current.currentSceneStates[sceneName];
    }
    //Update Game value of currentSceneName
    Game.current.currentSceneName = sceneName;
}
```

All of the above systems work together to do the following:

1. Load a SceneState from the current Game's list of SceneStates with information on SerializedObjects before changing into the new scene
2. Switch scenes and have all SerializedObjects lookup their previous ObjectStates states and set themselves accordingly
3. Once attempting to be deloaded, make each SerializedObject record it's state in the current SceneState's list of ObjectStates at the appropriate (key, value) location based on it's unique name
4. Place the current SceneState back into the Game's list of SceneStates for it to be stored until needed again

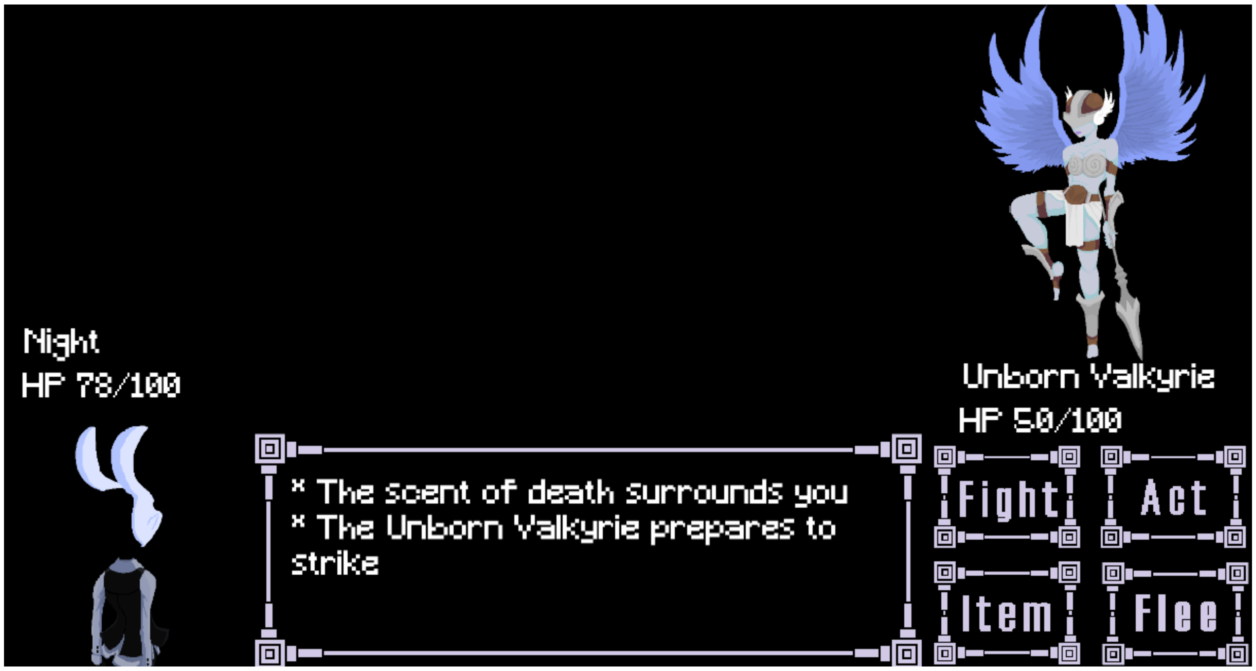
By storing serialized objects in this way, all SceneState information can be serialized by the SaveLoad system and stored as a simple text file.

Battle

Design of Battles

Battles within Last Hymn follow a turn-based attack system where the player acts and then the enemy responds until some ending condition has been met. Battles can end from the player dying, the enemy dying, or in some cases through certain actions taken during battle that result in non-death related endings.

Both player attacks and enemy attacks work on a rhythm-based fighting system. Attacks consist of directional arrows moving onto the screen towards designated final destinations. These arrows move along defined paths and are preemptively generated in a way that causes them to reach their ending destinations at the same time as distinguishable moments of the background music. The paths for each attack are themed in a way that attempts to correlate to the attacks source. For example, enemy attacks attempt to thematically resemble the underlying essence of the enemy itself.

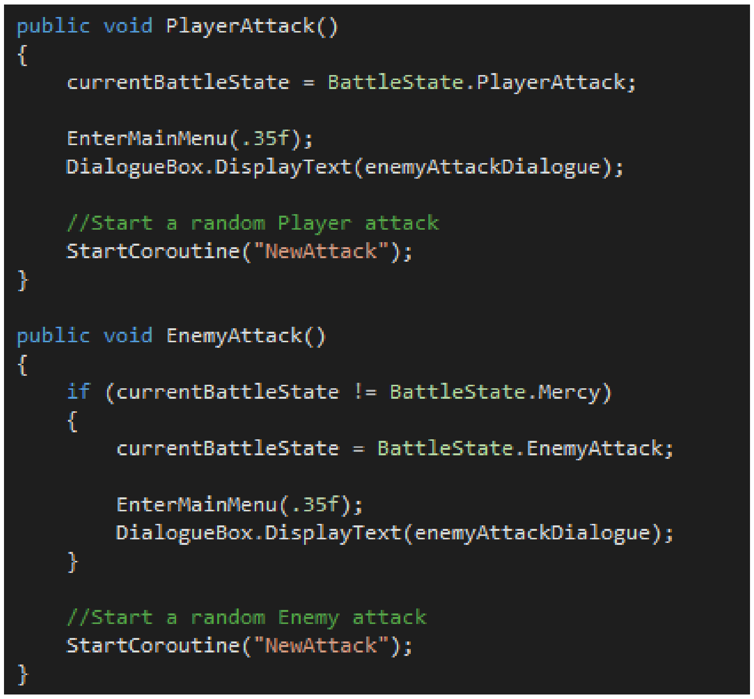


* The scent of death surrounds you
* The Unborn Valkyrie prepares to strike

Turn Controllers

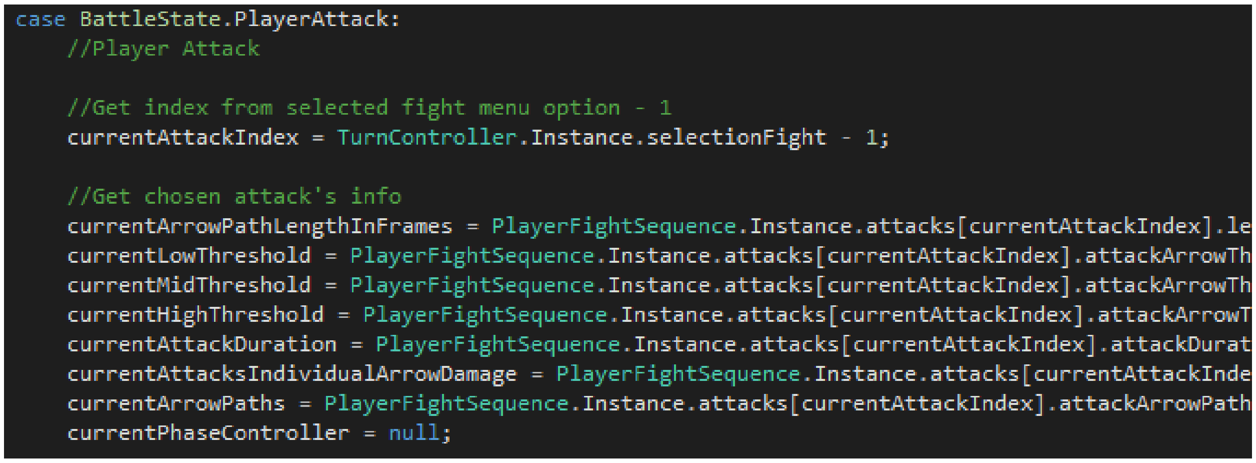
Fights in Last Hymn vary in dialogue as well as in the situations which cause them to end before someone dies. Turn Controllers are the scripts created to handle distinguishing these unique features within each battle. With a base Turn Controller class defined that implements key basic functionality of all Turn Controllers, each battle has an inherited Turn Controller containing that battle's specifics.

Below, the functionality of entering attacks can be seen.

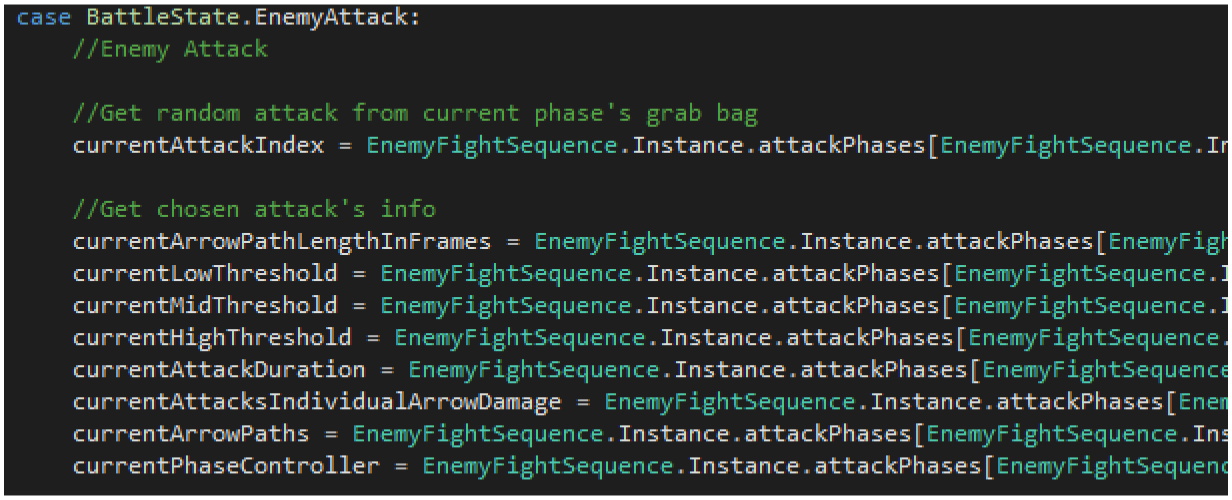


```
public void PlayerAttack()
{
    currentBattleState = BattleState.PlayerAttack;

    EnterMainMenu(.35f);
    DialogueBox.DisplayText(enemyAttackDialogue);

    //Start a random Player attack
    StartCoroutine("NewAttack");
}

public void EnemyAttack()
{
    if (currentBattleState != BattleState.Mercy)
    {
        currentBattleState = BattleState.EnemyAttack;

        EnterMainMenu(.35f);
        DialogueBox.DisplayText(enemyAttackDialogue);
    }

    //Start a random Enemy attack
    StartCoroutine("NewAttack");
}
```

Note the functions change BattleState values before calling the same generalized NewAttack coroutine. This approach is used so that the scripts used to generate attacks can be reused for both the player attacks as well as the enemy attacks.



```
case BattleState.PlayerAttack:
    //Player Attack

    //Get index from selected fight menu option - 1
    currentAttackIndex = TurnController.Instance.selectionFight - 1;

    //Get chosen attack's info
    currentArrowPathLengthInFrames = PlayerFightSequence.Instance.attacks[currentAttackIndex].le
    currentLowThreshold = PlayerFightSequence.Instance.attacks[currentAttackIndex].attackArrowTh
    currentMidThreshold = PlayerFightSequence.Instance.attacks[currentAttackIndex].attackArrowTh
    currentHighThreshold = PlayerFightSequence.Instance.attacks[currentAttackIndex].attackArrowT
    currentAttackDuration = PlayerFightSequence.Instance.attacks[currentAttackIndex].attackDurat
    currentAttacksIndividualArrowDamage = PlayerFightSequence.Instance.attacks[currentAttackInde
    currentArrowPaths = PlayerFightSequence.Instance.attacks[currentAttackIndex].attackArrowPath
    currentPhaseController = null;
```

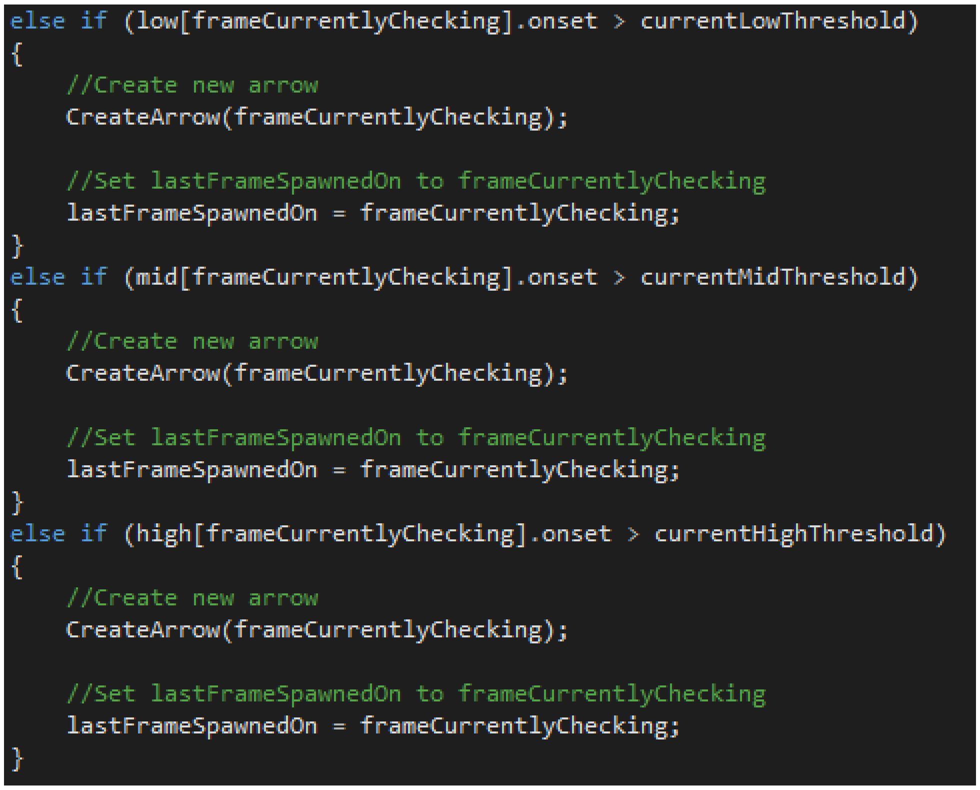


```
case BattleState.EnemyAttack:
    //Enemy Attack

    //Get random attack from current phase's grab bag
    currentAttackIndex = EnemyFightSequence.Instance.attackPhases[EnemyFightSequence.In

    //Get chosen attack's info
    currentArrowPathLengthInFrames = EnemyFightSequence.Instance.attackPhases[EnemyFigh
    currentLowThreshold = EnemyFightSequence.Instance.attackPhases[EnemyFightSequence.I
    currentMidThreshold = EnemyFightSequence.Instance.attackPhases[EnemyFightSequence.I
    currentHighThreshold = EnemyFightSequence.Instance.attackPhases[EnemyFightSequence.
    currentAttackDuration = EnemyFightSequence.Instance.attackPhases[EnemyFightSequence
    currentAttacksIndividualArrowDamage = EnemyFightSequence.Instance.attackPhases[Enem
    currentArrowPaths = EnemyFightSequence.Instance.attackPhases[EnemyFightSequence.Ins
    currentPhaseController = EnemyFightSequence.Instance.attackPhases[EnemyFightSequenc
```

Attack Arrow Generation

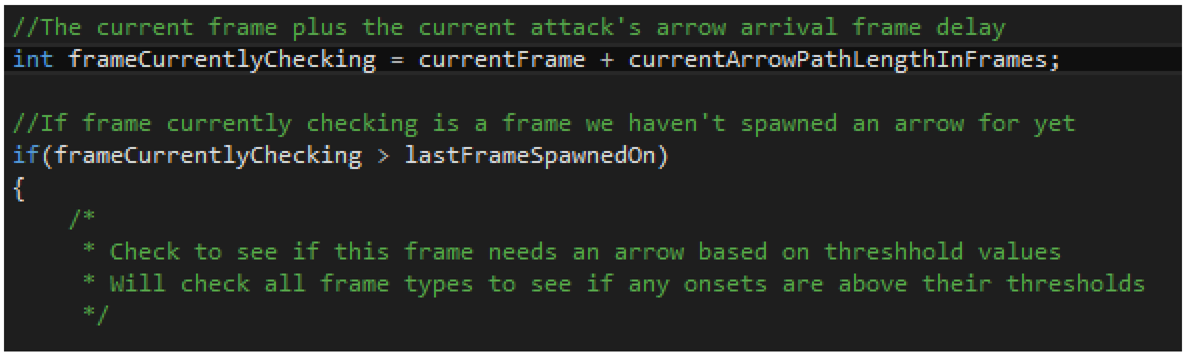
The arrows that appear during player and enemy attacks are generated to the beat of the background music.

To do this, the we utilized a rhythm analyzer which is setup to interpret music files on a per frame basis. Each resulting analyzed song frame is then interpreted by our DDRGeneration script to see if the frame's sound thresholds pass the attack's predefined cutoff values and in cases where it does, an arrow is created.



```
else if (low[frameCurrentlyChecking].onset > currentLowThreshold)
{
    //Create new arrow
    CreateArrow(frameCurrentlyChecking);

    //Set lastFrameSpawnedOn to frameCurrentlyChecking
    lastFrameSpawnedOn = frameCurrentlyChecking;
}
else if (mid[frameCurrentlyChecking].onset > currentMidThreshold)
{
    //Create new arrow
    CreateArrow(frameCurrentlyChecking);

    //Set lastFrameSpawnedOn to frameCurrentlyChecking
    lastFrameSpawnedOn = frameCurrentlyChecking;
}
else if (high[frameCurrentlyChecking].onset > currentHighThreshold)
{
    //Create new arrow
    CreateArrow(frameCurrentlyChecking);

    //Set lastFrameSpawnedOn to frameCurrentlyChecking
    lastFrameSpawnedOn = frameCurrentlyChecking;
}
```

Arrows are preemptively created so that they have time to traverse the visual playing field and can be recognized by the player. To do this, the arrow generator looks ahead in the song's frame array by the current attack's arrow path frame length value as shown below.



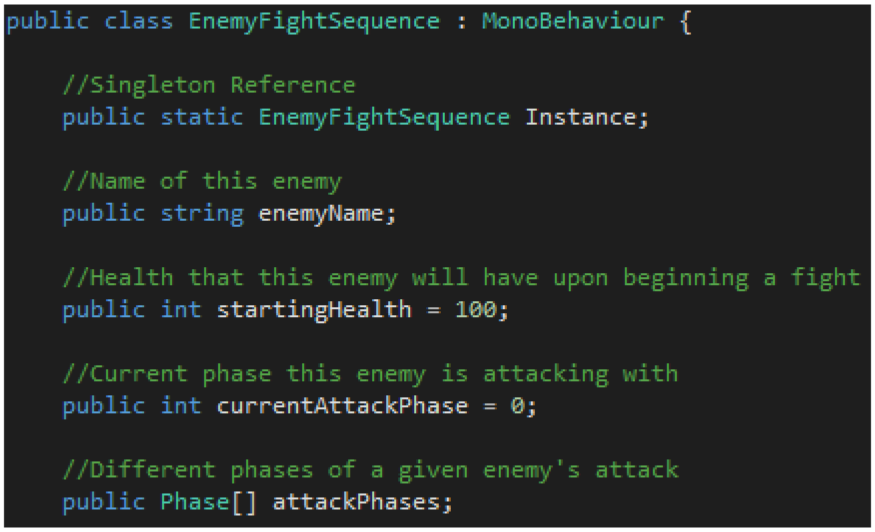
```
//The current frame plus the current attack's arrow arrival frame delay
int frameCurrentlyChecking = currentFrame + currentArrowPathLengthInFrames;

//If frame currently checking is a frame we haven't spawned an arrow for yet
if(frameCurrentlyChecking > lastFrameSpawnedOn)
{
    /*
     * Check to see if this frame needs an arrow based on threshhold values
     * Will check all frame types to see if any onsets are above their thresholds
     */
```

This causes any arrow generated be on the beat of the music when it arrives at the end of its movement path.

Fight Sequences

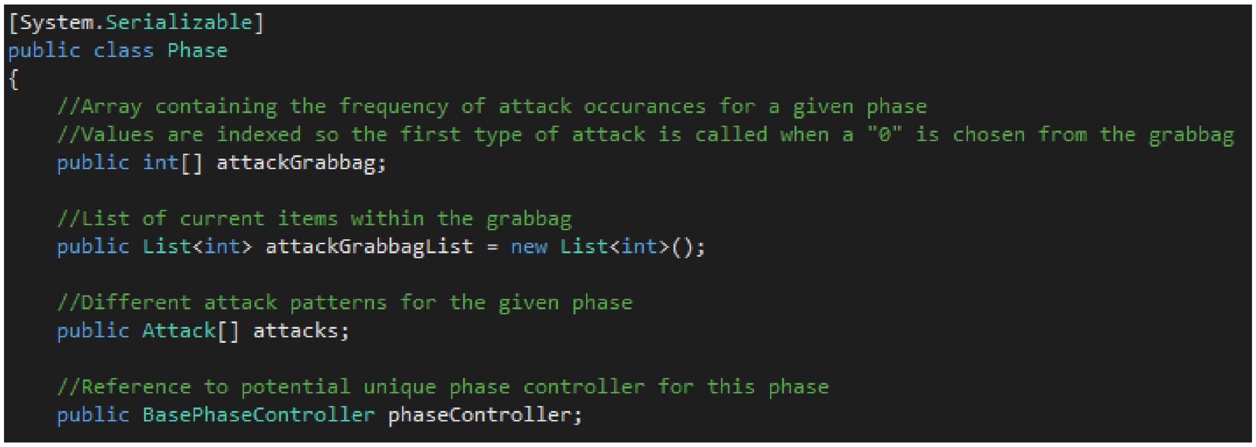
WIthin each battle, there are two scripts which contain information on the player and enemy attack patterns which are known as FightSequences. FightSequences are structured in a way that allows for unique attack definitions to be made and changed on a per battle basis. The structure for these objects is explained below.

Sequences contain the basic, unchanging information for a specific battle. In the case below, the EnemyFightSequence is shown detailing starting health, name, and Phase information.



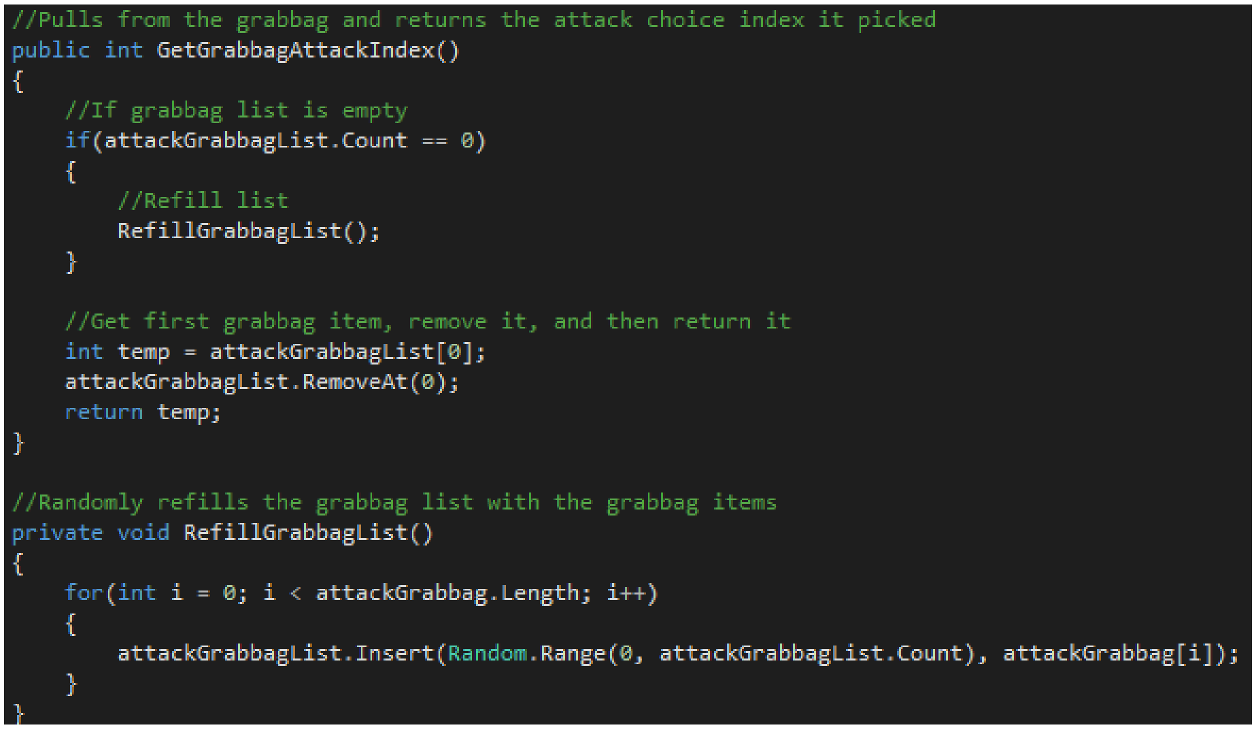
```
public class EnemyFightSequence : MonoBehaviour {

    //Singleton Reference
    public static EnemyFightSequence Instance;

    //Name of this enemy
    public string enemyName;

    //Health that this enemy will have upon beginning a fight
    public int startingHealth = 100;

    //Current phase this enemy is attacking with
    public int currentAttackPhase = 0;

    //Different phases of a given enemy's attack
    public Phase[] attackPhases;
```

Each sequence contains one or more Phases. Phases are used to break up different sections of a fight, for example the main boss fight has multiple phases where attack patterns change and become more difficult as the phases progress.



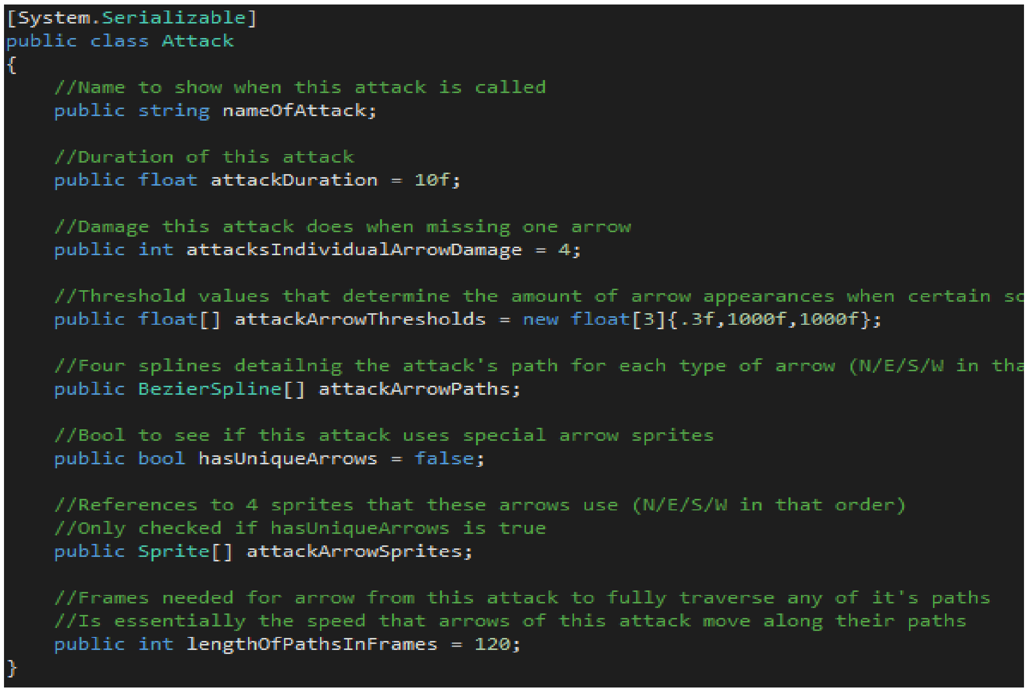
```
[System.Serializable]
public class Phase
{
    //Array containing the frequency of attack occurances for a given phase
    //Values are indexed so the first type of attack is called when a "0" is chosen from the grabbag
    public int[] attackGrabbag;

    //List of current items within the grabbag
    public List<int> attackGrabbagList = new List<int>();

    //Different attack patterns for the given phase
    public Attack[] attacks;

    //Reference to potential unique phase controller for this phase
    public BasePhaseController phaseController;
```

A unique feature within enemy Phases is the attack grabbag list. The attack grabbag causes enemies to cycle through attacks in a semi-random way, which makes it so that battles won’t be too difficult or too repetitive. The grabbag pop and refill functionality can be seen below.



```
//Pulls from the grabbag and returns the attack choice index it picked
public int GetGrabbagAttackIndex()
{
    //If grabbag list is empty
    if(attackGrabbagList.Count == 0)
    {
        //Refill list
        RefillGrabbagList();
    }

    //Get first grabbag item, remove it, and then return it
    int temp = attackGrabbagList[0];
    attackGrabbagList.RemoveAt(0);
    return temp;
}

//Randomly refills the grabbag list with the grabbag items
private void RefillGrabbagList()
{
    for(int i = 0; i < attackGrabbag.Length; i++)
    {
        attackGrabbagList.Insert(Random.Range(0, attackGrabbagList.Count), attackGrabbag[i]);
    }
}
```

Phases have one or more Attacks which are used mainly to store details on arrow pathing, timing, damage and spawning thresholds among other things. Attacks are looked up by the DDRGeneration script when a new attack is called, thus having this detailed object definition allows for clear and straight-forward battle creation to occur with just the use of a few indexed variables.

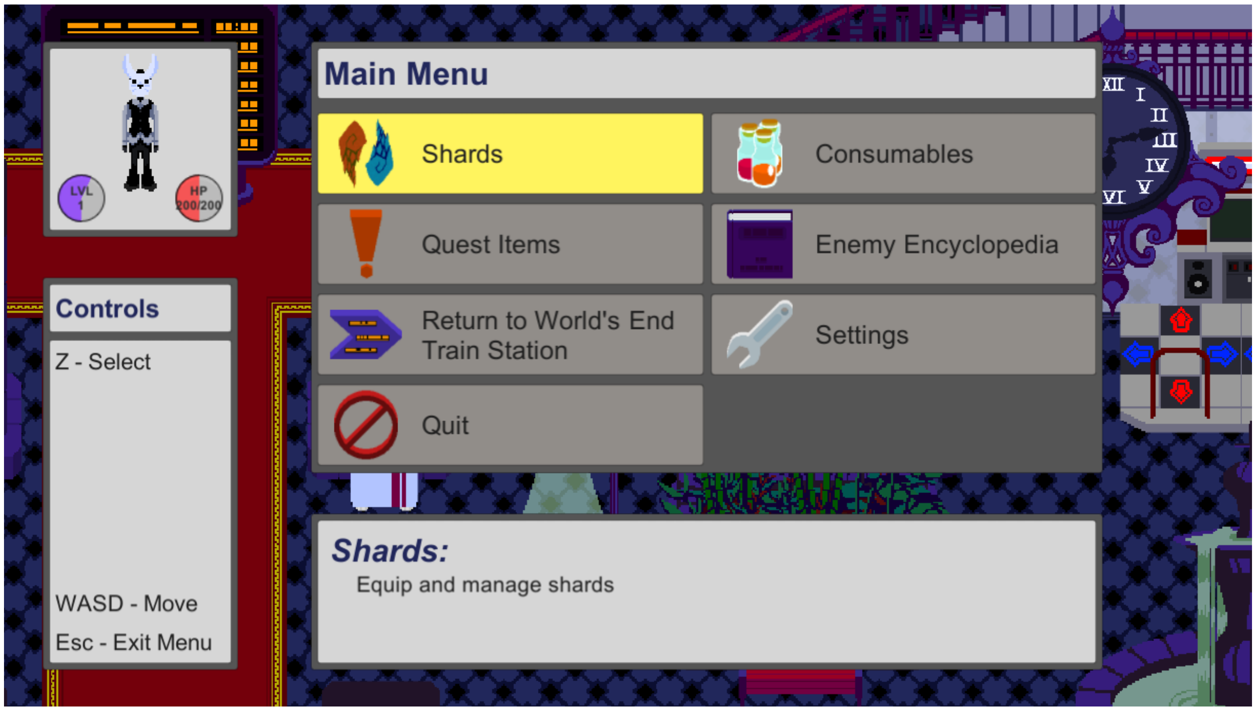


```
[System.Serializable]
public class Attack
{
    //Name to show when this attack is called
    public string nameOfAttack;

    //Duration of this attack
    public float attackDuration = 10f;

    //Damage this attack does when missing one arrow
    public int attacksIndividualArrowDamage = 4;

    //Threshold values that determine the amount of arrow appearances when certain sc
    public float[] attackArrowThresholds = new float[3]{.3f,1000f,1000f};

    //Four splines detailnig the attack's path for each type of arrow (N/E/S/W in tha
    public BezierSpline[] attackArrowPaths;

    //Bool to see if this attack uses special arrow sprites
    public bool hasUniqueArrows = false;

    //References to 4 sprites that these arrows use (N/E/S/W in that order)
    //Only checked if hasUniqueArrows is true
    public Sprite[] attackArrowSprites;

    //Frames needed for arrow from this attack to fully traverse any of it's paths
    //Is essentially the speed that arrows of this attack move along their paths
    public int lengthOfPathsInFrames = 120;
}
```

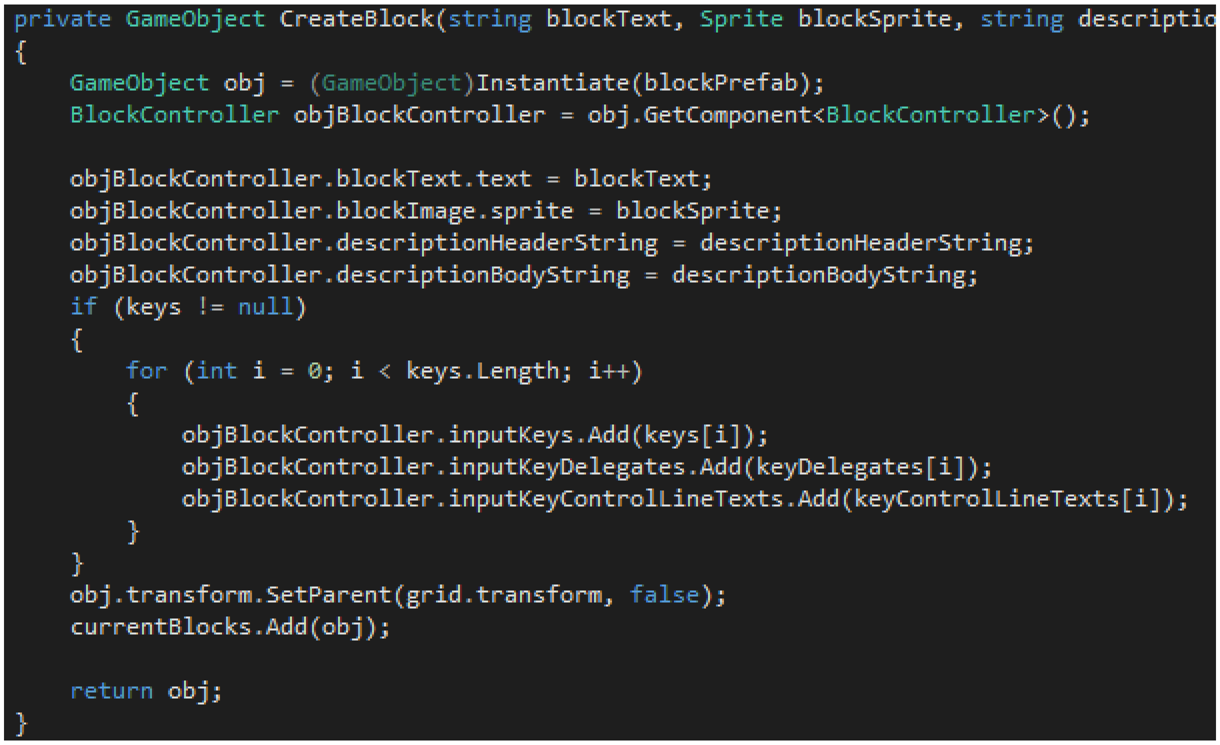
RPG

Menu System

The in-game menu of Last Hymn is an adaptive, dynamically populating view featuring a delegate-based method calling for its interactive components, which makes for a simple and sleek approach to menu option creation.



Depending on the current menu state, different views are generated. Each view has many of the same components, each one varying only by the option buttons which are created based on the current state. These option buttons are known as list blocks and they are created by the MenuListBlocksController. This script runs a specific method based on the current menu state to create the necessary list blocks. Shown below is the function which creates these list blocks.

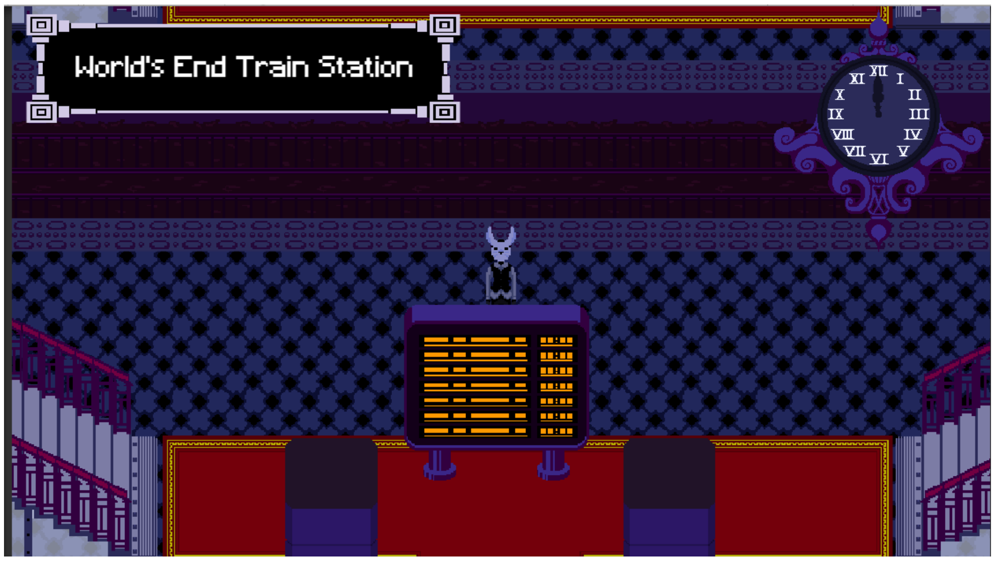


```
private GameObject CreateBlock(string blockText, Sprite blockSprite, string descriptio
{
    GameObject obj = (GameObject)Instantiate(blockPrefab);
    BlockController objBlockController = obj.GetComponent<BlockController>();

    objBlockController.blockText.text = blockText;
    objBlockController.blockImage.sprite = blockSprite;
    objBlockController.descriptionHeaderString = descriptionHeaderString;
    objBlockController.descriptionBodyString = descriptionBodyString;
    if (keys != null)
    {
        for (int i = 0; i < keys.Length; i++)
        {
            objBlockController.inputKeys.Add(keys[i]);
            objBlockController.inputKeyDelegates.Add(keyDelegates[i]);
            objBlockController.inputKeyControlLineTexts.Add(keyControlLineTexts[i]);
        }
    }
    obj.transform.SetParent(grid.transform, false);
    currentBlocks.Add(obj);

    return obj;
}
```

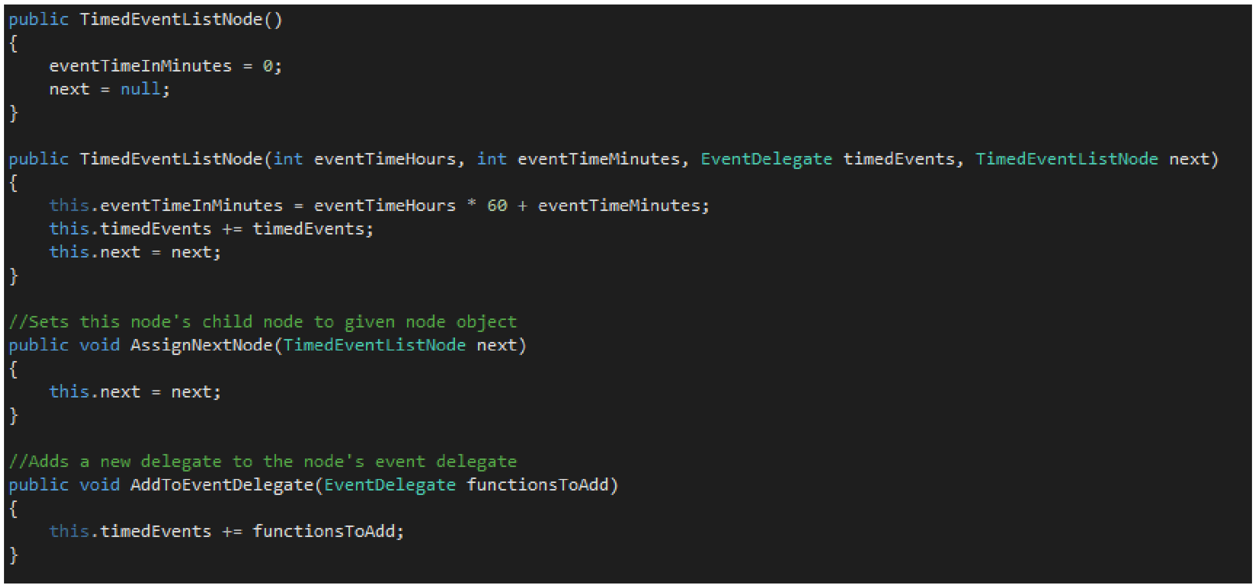
Note how each block has a BlockController which contains the inputKeys and inputKeyDelegates lists. These cause a given list block to respond to a press of inputKeys[x] by calling the inputKeyDelegates[x] method. This is how menu buttons for items and menu buttons for exiting the game can run off the same base button object but have different functionality due to differences in inputKeys and inputKeyDelegates.

Time

Last Hymn uses time as one of its main gimmicks associated with the RPG world. Most NPCs have some reliance on the clock that is featured in the upper-right corner of the screen. Our game takes place between the hours of 6 AM and Midnight with the player having to complete all the quest lines in that time period. If the clock reaches midnight, the world is destroyed and the player returns to the World's End Train Station with the world resetting back to 6 AM.



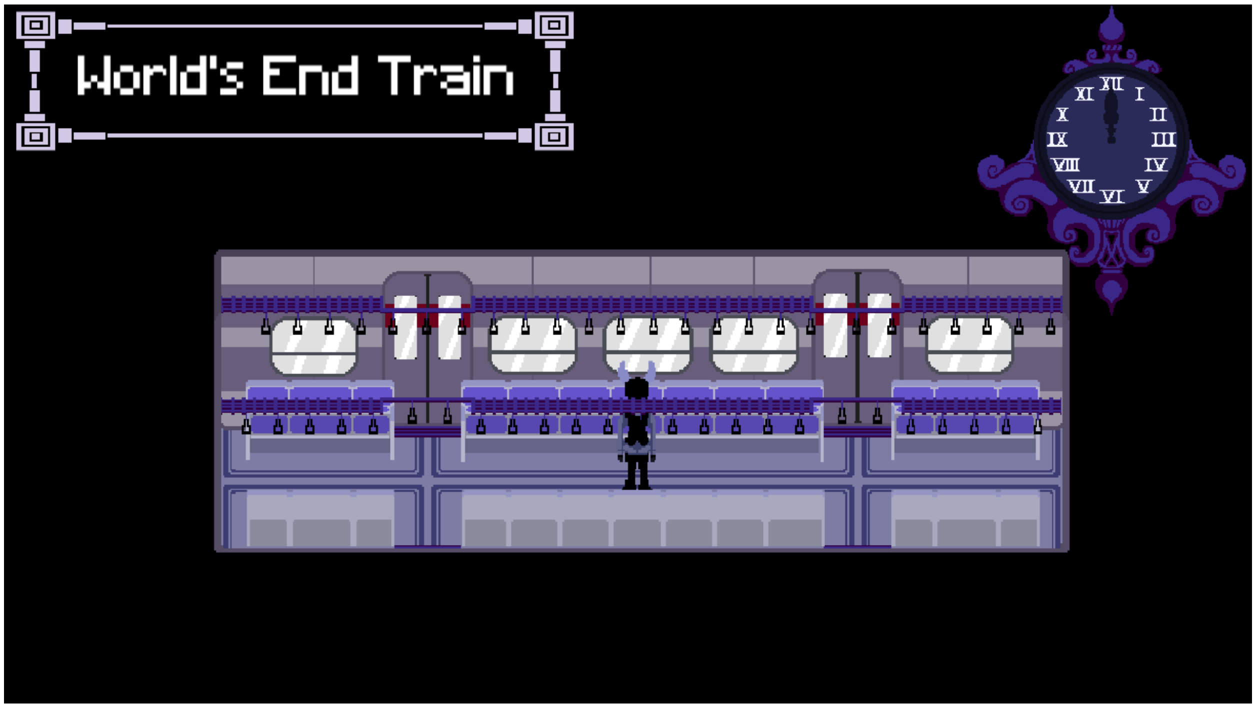
Other gameplay systems rely on the clock through a Caller and Listener System. Instead of every object in the game that relies on time constantly checking to see if it needs to trigger or not we have a TimedEventController.



```
public TimedEventListNode()
{
    eventTimeInMinutes = 0;
    next = null;
}

public TimedEventListNode(int eventTimeHours, int eventTimeMinutes, EventDelegate timedEvents, TimedEventListNode next)
{
    this.eventTimeInMinutes = eventTimeHours * 60 + eventTimeMinutes;
    this.timedEvents += timedEvents;
    this.next = next;
}

//Sets this node's child node to given node object
public void AssignNextNode(TimedEventListNode next)
{
    this.next = next;
}

//Adds a new delegate to the node's event delegate
public void AddToEventDelegate(EventDelegate functionsToAdd)
{
    this.timedEvents += functionsToAdd;
}
```

This controller has a linked list of events that are inserted into it by various objects. That way only this linked list has to check for the time to trigger events. It is arranged chronologically so only the next node has to be checked to see if it should trigger. Having a one to one relationship between the clock and the events, ordering of events is made easier, bug testing is made easier, and computation time is heavily reduced. It is also useful for implementing schedules with the Train System inserting nodes into this event queue to generate trains.

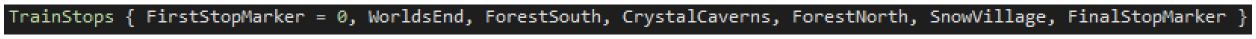
An interesting aspect of this system is that when you load into an area all events that should of happened prior to that time (e.g. it is 3 PM and a vendor leaves an area at noon) will occur as well chronologically as part of the load time of a scene. This is done to maintain a consistent event store and to ensure that events are always occurring.



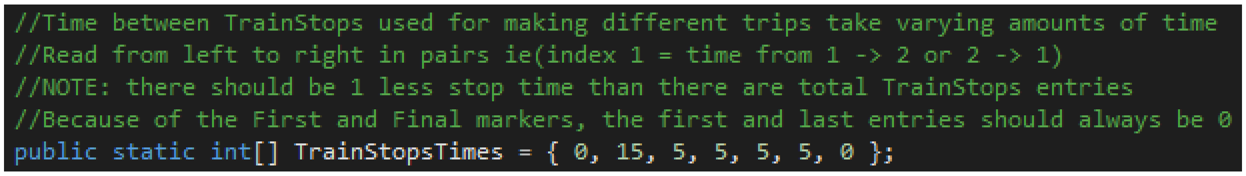
Train Controller

Trains are a key game component. The underlying systems used to control how trains are scheduled and generated, how train stations link between each other, and how the player interacts with the various types of trains is all controlled via the TrainStationController.

Below, the hard coded information necessary for all train stations is shown. The details for train stop names as well as travel times between stops is important to have universally the same.

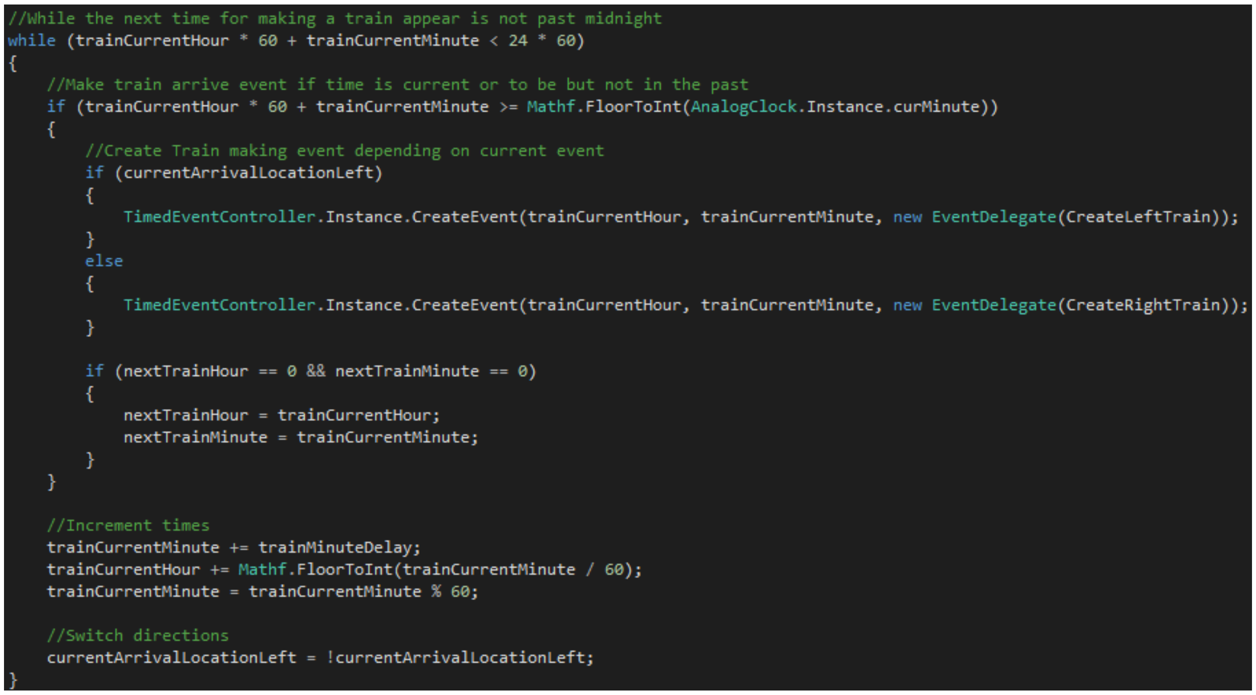


```
TrainStops { FirstStopMarker = 0, WorldsEnd, ForestSouth, CrystalCaverns, ForestNorth, SnowVillage, FinalStopMarker }
```



```
//Time between TrainStops used for making different trips take varying amounts of time
//Read from left to right in pairs ie(index 1 = time from 1 -> 2 or 2 -> 1)
//NOTE: there should be 1 less stop time than there are total TrainStops entries
//Because of the First and Final markers, the first and last entries should always be 0
public static int[] TrainStopsTimes = { 0, 15, 5, 5, 5, 5, 0 };
```

For each train station, there is a need to have trains arrive in given intervals and to do this, the method below is used to create a dynamically generated list of train arrival events to be added to the delegate-calling time system.



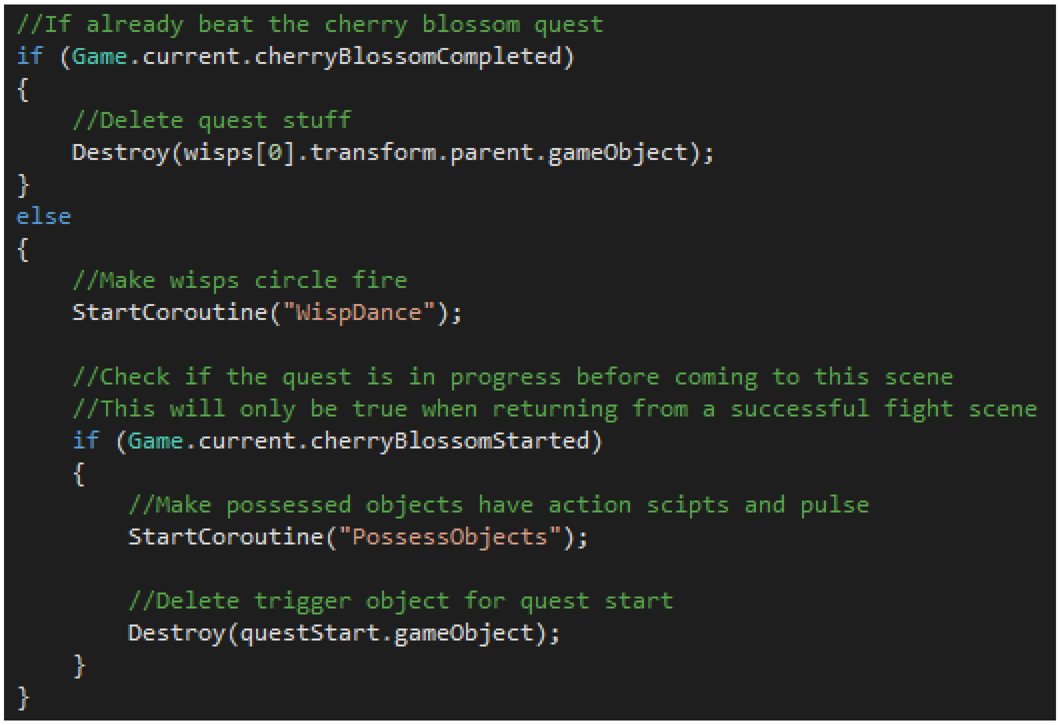
```
//While the next time for making a train appear is not past midnight
while (trainCurrentHour * 60 + trainCurrentMinute < 24 * 60)
{
    //Make train arrive event if time is current or to be but not in the past
    if (trainCurrentHour * 60 + trainCurrentMinute >= Mathf.FloorToInt(AnalogClock.Instance.curMinute))
    {
        //Create Train making event depending on current event
        if (currentArrivalLocationLeft)
        {
            TimedEventController.Instance.CreateEvent(trainCurrentHour, trainCurrentMinute, new EventDelegate(CreateLeftTrain));
        }
        else
        {
            TimedEventController.Instance.CreateEvent(trainCurrentHour, trainCurrentMinute, new EventDelegate(CreateRightTrain));
        }

        if (nextTrainHour == 0 && nextTrainMinute == 0)
        {
            nextTrainHour = trainCurrentHour;
            nextTrainMinute = trainCurrentMinute;
        }
    }

    //Increment times
    trainCurrentMinute += trainMinuteDelay;
    trainCurrentHour += Mathf.FloorToInt(trainCurrentMinute / 60);
    trainCurrentMinute = trainCurrentMinute % 60;

    //Switch directions
    currentArrivalLocationLeft = !currentArrivalLocationLeft;
}
```

With the above scheduled train arrivals, each train station upon loading a new scene will automatically sync up with the current Game time and create the appropriate train arrivals as needed.

Cutscene System

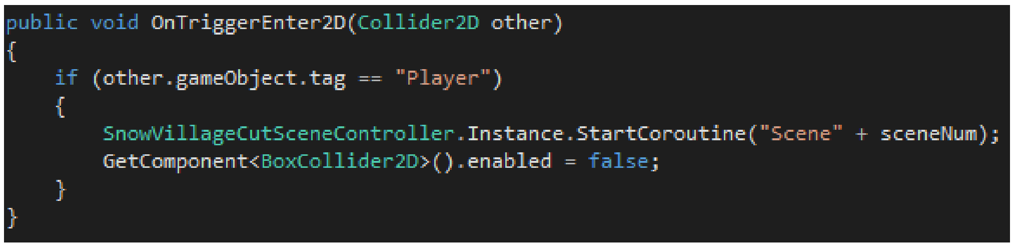
Cutscenes are used in situations where many story elements need to be conveyed in quick succession via dialogue displays and unique game state changes.

For this game’s usage of cutscenes, unique cutscene controller scripts exist to handle these. Working off of an underlying system of recorded quest triggers, cutscene controllers are able to retain quest-related game state changes. This means that moving between areas of the game won't cause loss of quest progression or odd visual disparities when reloading a previously affected region. Seen below is an example of one such controller’s method used to recreate quest changes using these recorded game triggers.



```
//If already beat the cherry blossom quest
if (Game.current.cherryBlossomCompleted)
{
    //Delete quest stuff
    Destroy(wisps[0].transform.parent.gameObject);
}
else
{
    //Make wisps circle fire
    StartCoroutine("WispDance");

    //Check if the quest is in progress before coming to this scene
    //This will only be true when returning from a successful fight scene
    if (Game.current.cherryBlossomStarted)
    {
        //Make possessed objects have action scipts and pulse
        StartCoroutine("PossessObjects");

        //Delete trigger object for quest start
        Destroy(questStart.gameObject);
    }
}
```

Cutscene controllers also work with these triggers during gameplay through various methods which activate after certain conditions are met. The main situation this pertains to is when the player walks into an invisible collider which causes a cutscene to begin. An example of this activator script is shown below.

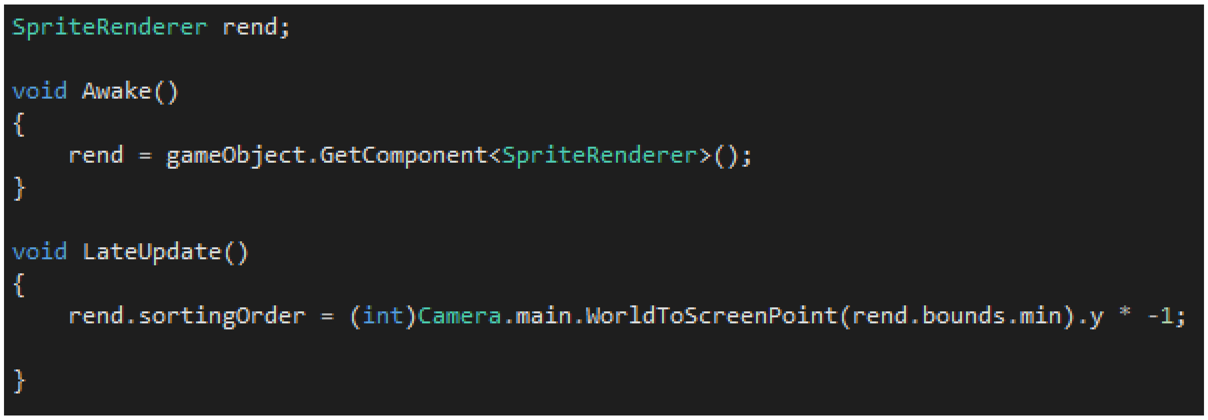


```
public void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Player")
    {
        SnowVillageCutSceneController.Instance.StartCoroutine("Scene" + sceneNum);
        GetComponent<BoxCollider2D>().enabled = false;
    }
}
```

Dynamic Layering

The RPG levels of Last Hymn were crafted using a two-tiered layering system: the strict layers and dynamic layers. Strict Layers are sprite layers that are rendered in a set order. We have a Background Layer that holds the base grass, snow, and objects that should be beneath the player. On top of that is the Addon Layer that holds rivers, cliffs, and objects that should always be rendered above the ground but beneath the player. Next is the Default layer that has the Player, Enemies, Buildings, Grass, and the majority of objects that are interacted with. Finally there is a Foreground Layer that contains items that are always above the player like roofs.

Inside of each of these Strict Layers we apply a Dynamic Layering script to each object that reorders every object in the layer depending on its position in the world so that objects on the same layer as the player will be properly layered with grass in front of or behind the player depending on the player’s current positioning. This gives the game a 3D feel to it while still being rendered with 2D sprites.



```
SpriteRenderer rend;

void Awake()
{
    rend = gameObject.GetComponent<SpriteRenderer>();
}

void LateUpdate()
{
    rend.sortingOrder = (int)Camera.main.WorldToScreenPoint(rend.bounds.min).y * -1;

}
```

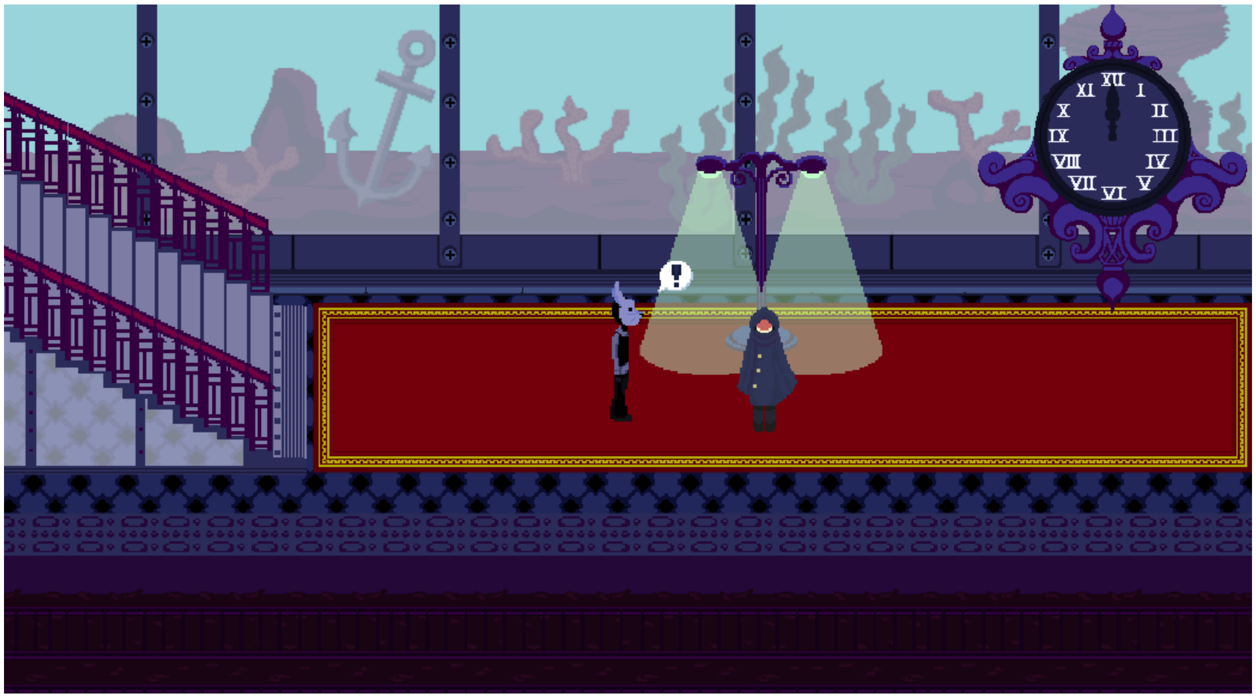
A LateUpdate is used so that the sprite’s layer positioning will change after all movement is finished in the player’s, enemies’, and NPCs’ Update functions. We also have to multiply by a negative constant so that objects that are below the player can be rendered on top of them instead of behind.

World’s End Train Station

Design of the Area

The World’s End Train Station was one of the first areas designed for the game. This was due to the fact that it functioned as the hub area that the player would return to throughout the game, such as when they died. This presented a unique issue though, in that while it was the hub area, the narrative of the game dictated that the train station feel distinctly separate from the rest of the game while still maintaining the same visual style cohesively. As such, key concepts and guidelines for the rest of the assets were also produced in conjunction with the design of The World’s End Train Station.

Such decisions include the limited color palette to be utilized for all the assets, restricting shading values to two or three bands with the exception of rare cases. Outlines were ruled out as well and strong hue shifting was employed as the primary method of shading objects. After these guidelines were established, the train station was then made to stand out by employing more saturated colors compared to the rest of the game while also implementing a higher degree of modern objects and trappings compared to the later more nature based areas of the game.

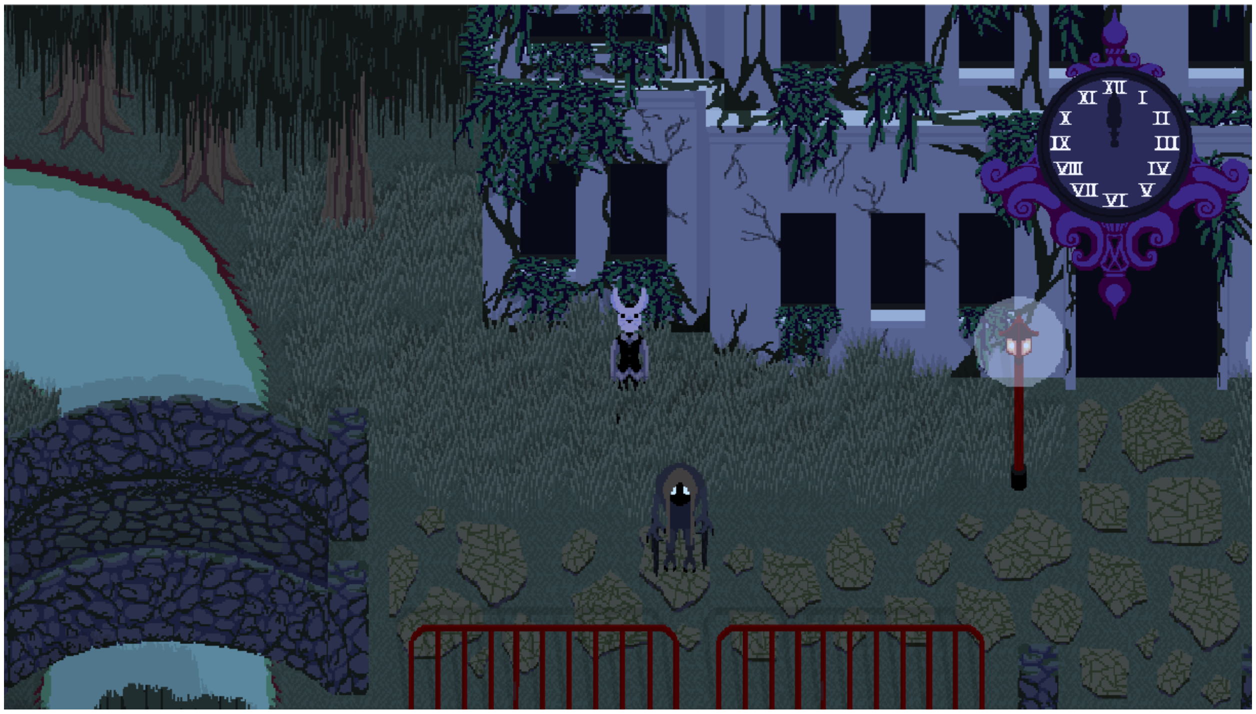


Forest

Design of the Area

The Forest is the largest area in the game and serves as an informal overworld that the player walks through to access different areas of the game. As a consequence, the Forest also contains the highest number of enemies and overall contains the highest number of unique assets anywhere in game. This wide discrepancy in locales was the deciding factor in determining that the area would be a forest at all, as it was found that a forest would not create a style conflict with any of the planned locales and in fact would actually help to bring them closer together and unify the artstyle. To accomplish this, many design decisions and patterns made in the forest were replicated across the game.

The way in which wood would be textured throughout the game was created in the Forest and used throughout all the other objects in the game made out of wood to contribute to a consistent and unified artstyle. On a similar note, this was also true of the color palette used for the wood as well. This was utilized as the point of reference for setting the tone of more demure, subdued colors in general for areas outside of The World’s End Train Station.

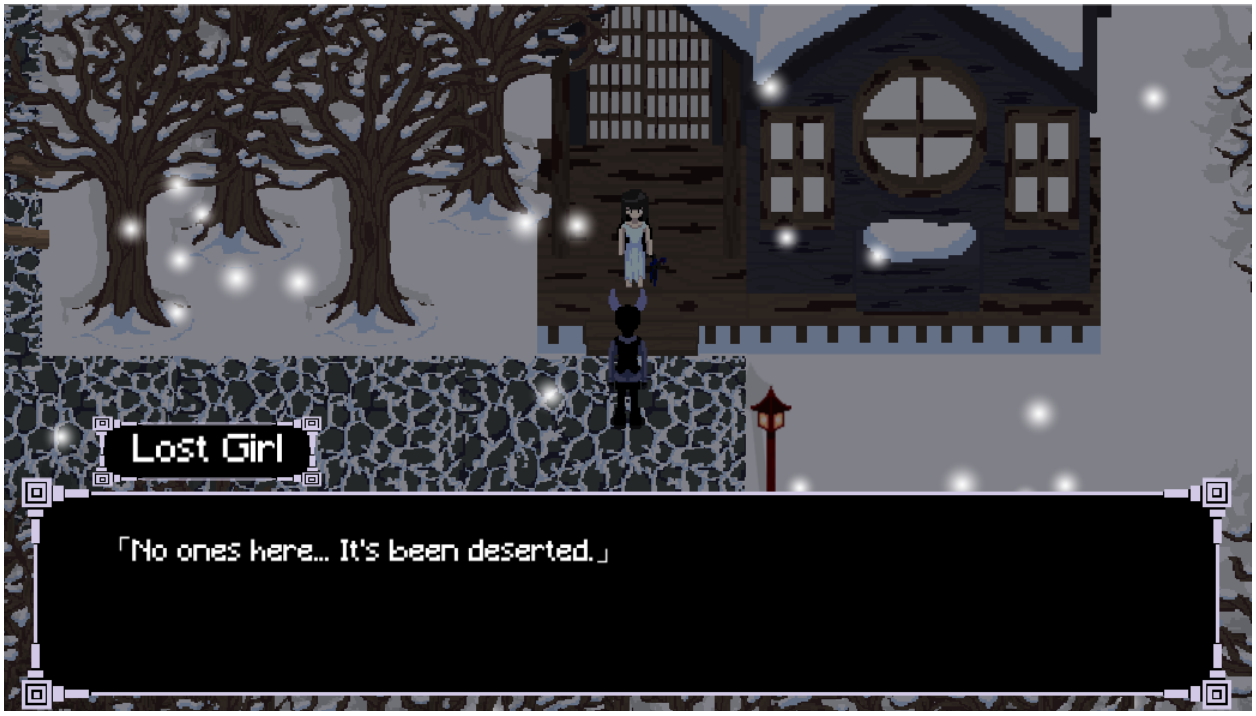


Snow Village Quest Line

Design of the Area

The Snow Village was a key sub-area in the game that challenged the team on what decisions had to be made about how to represent snow in the game's artstyle. Ultimately a highly simplified and "fluffy" look was decided for the snow that proved that the simplified shading bands was the right choice for our artstyle. This simplified look integrated well with the other design patterns present in the game such as the wood. It was also the first instance of a secondary effect being implemented into an area to enhance the look and feel, namely the footprints the player leaves behind as they walk in the snow and the blizzard that ramps up as the player progresses.

This area also marks the first time complex outdoor structures were created for the game, solidifying their style and cementing the "Oblique-like" perspective we had been using up until that point in development. The Snow Village is also where the style used to draw concrete and other "man-made" stone materials was decided as well, implementing the simple geometric design and shading used in the World's End Train Station.

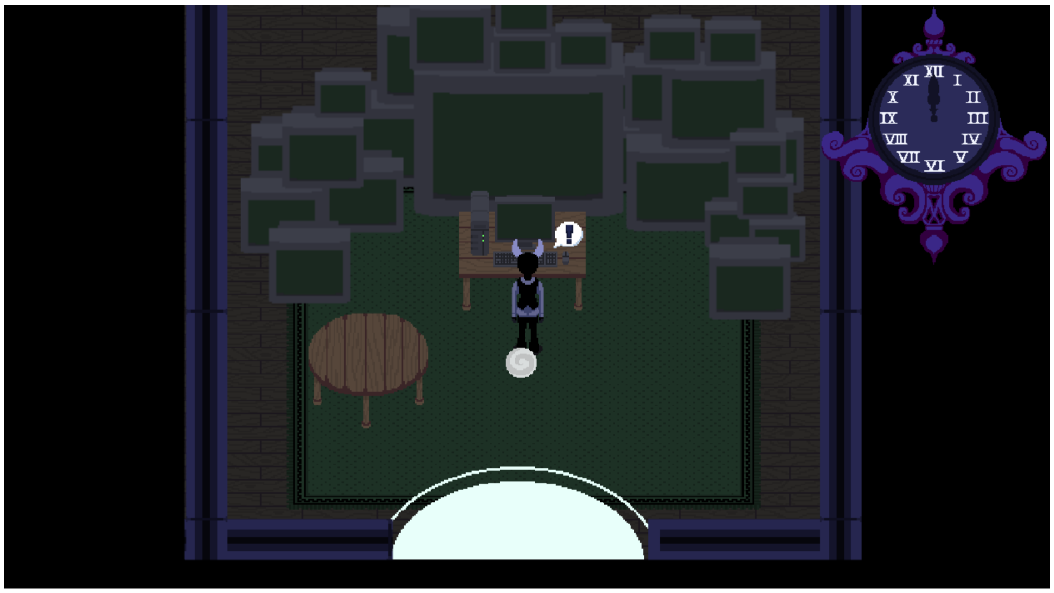


「No ones here... It's been deserted.」

Apartment Quest Line

Design of the Area

The Apartment Quest Line was designed to focus on a smaller area with a mix of indoor and outdoor areas. This was done to make the area feel much larger than it was since the main quest of the area features you exploring the world and taking photos of certain set pieces. The wood texture of the bookshelves and the simple style of the computer were reflections of previous design decisions intended to create a warm, cozy environment. The development of the Apartment area solidified this duality between “modern” and “naturalistic” components for the rest of the game. The actual building itself is a recreation of a modern apartment complex, while the level of technology on the inside is more reminiscent of a cabin style. Some of the assets used in the area were recreations of items in the World’s End Train Station, like the walls, meant to help provide the rooms with a more mysterious feel to them.



Crystal Flower Caverns

Design of the Area

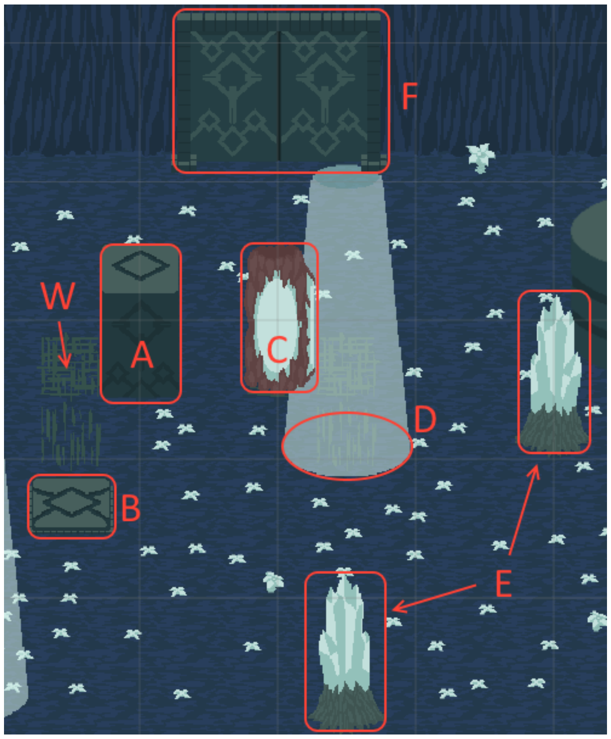
The Crystal Flower Caverns was the first area in which rock-like textures were developed for the game. To stay within the design boundaries set by other areas, it was decided that the darker of the two tones would represent the crevices that form in rocks while the other would naturally be used for highlighted areas caught by the light. In addition, two tones were used for the rocks to match better and unify them with the man-made stone structures present in the cave, which also used two tones. The hue shifting principle was also used here to better blend each of the individual items into each other. An example of this are the man-made structures. They were given a green color scheme while the cave and its crystal flowers remained primarily blue creating a varied, but unified color palette for the whole area.

Puzzle Mechanics

The Crystal Flower Caverns is the game's ending area. It consists of an entrance floor with train access and intro to the cave's puzzle mechanics, 3 one-time puzzle floors, and a final floor where the main quest line reaches and ending boss fight takes place.

The cave's puzzles were meant to incorporate the essence of the cave itself. The puzzles involve moving pillars, placing mirrors, and redirecting cavern light in order to illuminate crystal outcroppings and activate ancient mechanisms to descend further into the caves.

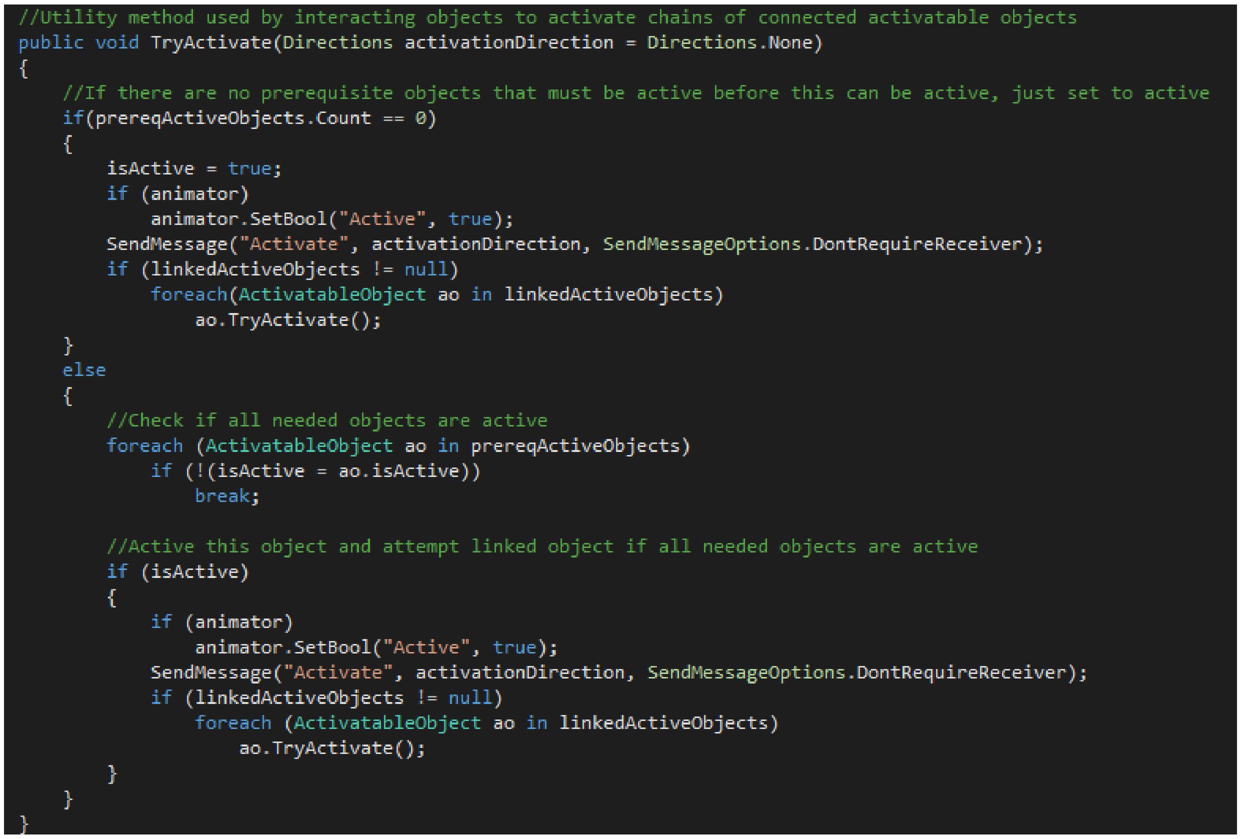
The puzzle mechanics work on a system of what we call ActivatableObjects. ActivatableObjects are essentially fancy booleans. Each ActivatableObject has an isActive state boolean value which determines if this object is active or not. When an ActivatableObject changes its isActive state, it attempts to make any other objects it is linked to active or deactive as well. This is done a predefined list of puzzle dependencies which allows us to create physical "if, then" conditions with the puzzle objects. The following is a detailed image of our intro puzzle area which will be used to further explain the ActivatableObject functionality.



A is a push block the player may walk into and move along the path of worn ground it is connected with. When the player attempts to move a pushable object, it checks the WorngroundController of the worn ground object beneath it to see if there is any worn ground in the pushing direction and if the worn ground allows motion in that direction. If all these checks pass, the pushable object moves along the path and connects to the next worn ground.

W is worn ground. Upon loading into a scene with worn ground, each worn ground object attempts to connect itself with surrounding pieces by checking one unit in all it's cardinal directions for overlaps with other Wornground layer objects. This linking is used when a pushable object is moved by the player as described above. Worn ground can be one of three direction types: LeftRight, UpDown, or Both. The wornground pointed at above is a Both type, while directly below it is an UpDown type. This type distinction places restrictions on which ways pushable objects may move when overtop certain types.

B is a pressure pad which utilizes the worn ground functionality of linking so that pushable objects may be placed onto them. In this example, the push block **A** is expected to be pushed onto this pressure pad which will in turn activate the cavern light **D**. This is the first example of ActivatableObject linking. **B** is an ActivatableObject which has no prerequisite objects, thus when a push block or the player is overtop it, it will activate itself. Upon activation, **B** then looks at its list of linked ActivatableObjects, in this case the list is just **D**, and attempts to activate each one by calling the TryActivate method on each. TryActivate may be seen below:



```
//Utility method used by interacting objects to activate chains of connected activatable objects
public void TryActivate(Directions activationDirection = Directions.None)
{
    //If there are no prerequisite objects that must be active before this can be active, just set to active
    if(prereqActiveObjects.Count == 0)
    {
        isActive = true;
        if (animator)
            animator.SetBool("Active", true);
        SendMessage("Activate", activationDirection, SendMessageOptions.DontRequireReceiver);
        if (linkedActiveObjects != null)
            foreach(ActivatableObject ao in linkedActiveObjects)
                ao.TryActivate();
    }
    else
    {
        //Check if all needed objects are active
        foreach (ActivatableObject ao in prereqActiveObjects)
            if (!(isActive = ao.isActive))
                break;

        //Active this object and attempt linked object if all needed objects are active
        if (isActive)
        {
            if (animator)
                animator.SetBool("Active", true);
            SendMessage("Activate", activationDirection, SendMessageOptions.DontRequireReceiver);
            if (linkedActiveObjects != null)
                foreach (ActivatableObject ao in linkedActiveObjects)
                    ao.TryActivate();
        }
    }
}
```

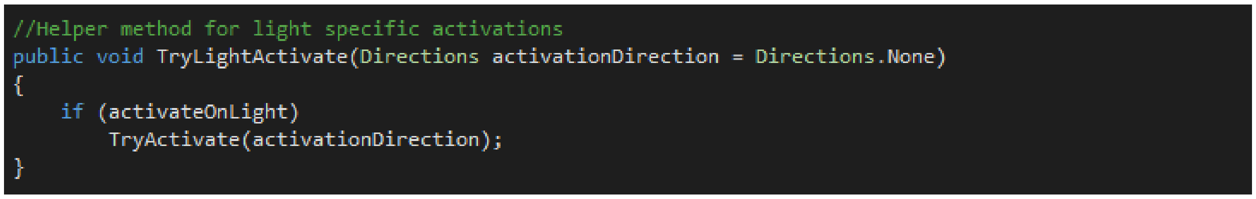
C is a pushable mirror pillar. There are two types of mirror pillars, Regular and Static, and both types may be immovable or pushable. The red coloring shows that this is a Static pillar, meaning that the mirrors seen on it's right and bottom sides are permanent and may not be removed. The fact that it is next to worn ground shows that it is pushable. Regular pillars are grey in color and their mirrors may be removed and replaced onto any of their sides. In this situation, the pillar is expected to be placed underneath the light **D**. When this occurs, the pillar will activate in a unique way as compared to the above described method for the pressure pad which is seen below:

C - Mirrors are placed on mirror pillars. In the above picture, **C** has mirrors on its right and bottom sides. Mirrors are a unique activatable object which always have one prerequisite being its parent mirror pillar. When the parent pillar activates, it activates all mirrors placed on itself. When a mirror activates, which will only ever happen due to light hitting a pillar, the mirror begins shooting a beam of light not shown in the above picture. This light beam will travel in the mirror's cardinal facing direction until coming into contact with something which can block it. When these light beams contact something which is an ActivatableObject, it checks to see if that object responds to light (ie pillars and pylons) and then performs the unique light activation on said colliding object.

D is cavern light. These light sources are used to activate mirror pillars placed beneath them. In the above image, **C** is expected to be moved onto the bottom most portion of it’s worn ground path which then causes it to become active when the cavern light is active. These lights are ActivatableObjects which will visually fade in and begin making any other ActivatableObjects beneath it activated with light in the same way mirrors do to objects.

E are pylons. These are ActivatableObjects which only respond to light like mirror pillars do.

F is a rune door. There are multiple types of rune doors, all doing essentially the same thing. These ActivatableObjects have no linked ActivatableObjects. Instead, they function as physical switches by opening and disabling their collisions so that the player may pass through areas they previously couldn’t.



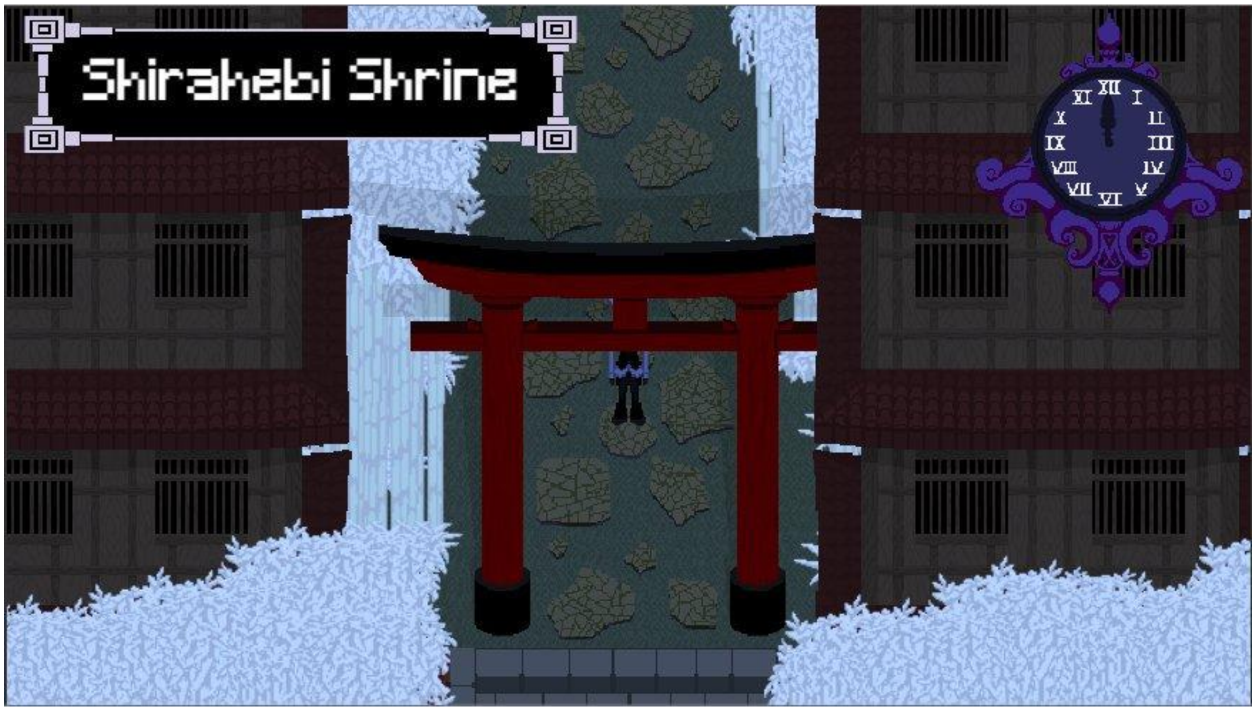
```
//Helper method for light specific activations
public void TryLightActivate(Directions activationDirection = Directions.None)
{
    if (activateOnLight)
        TryActivate(activationDirection);
}
```

Once a player has passed all the puzzle floors, they arrive to the boss area. Leaving and returning to the caverns will not cause the player to need to re-solve all puzzles. Instead the player enters the boss area directly, passing all puzzle floors.

Snake Shrine Quest Line

Design of the Area

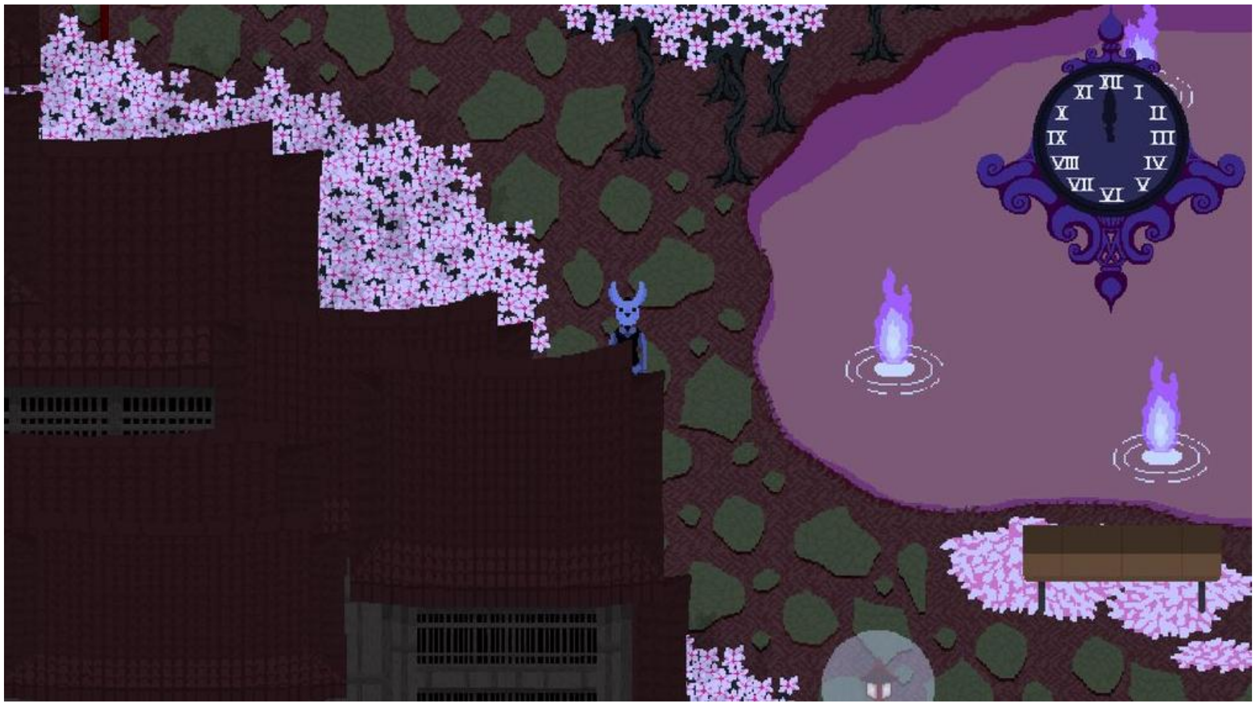
One of the most difficult areas to develop, the Snake Shrine utilized many real world elements with a very high detail level. The Chinese style hip and gable roofs in particular featured a very distinctive style that could not be heavily abstracted without hurting the overall image and signature look they conveyed. However once the main shrine structure was completed, progress on the area accelerated and it quickly became one of the highest quality areas in the area. It represented a blend of “modern” and “classic” elements that we wanted to achieve with the game’s aesthetic, contrasting the classic styled buildings with modern elements like the metal railing and favela style structure used throughout the area.



Cherry Blossom Quest Line

Design of the Area

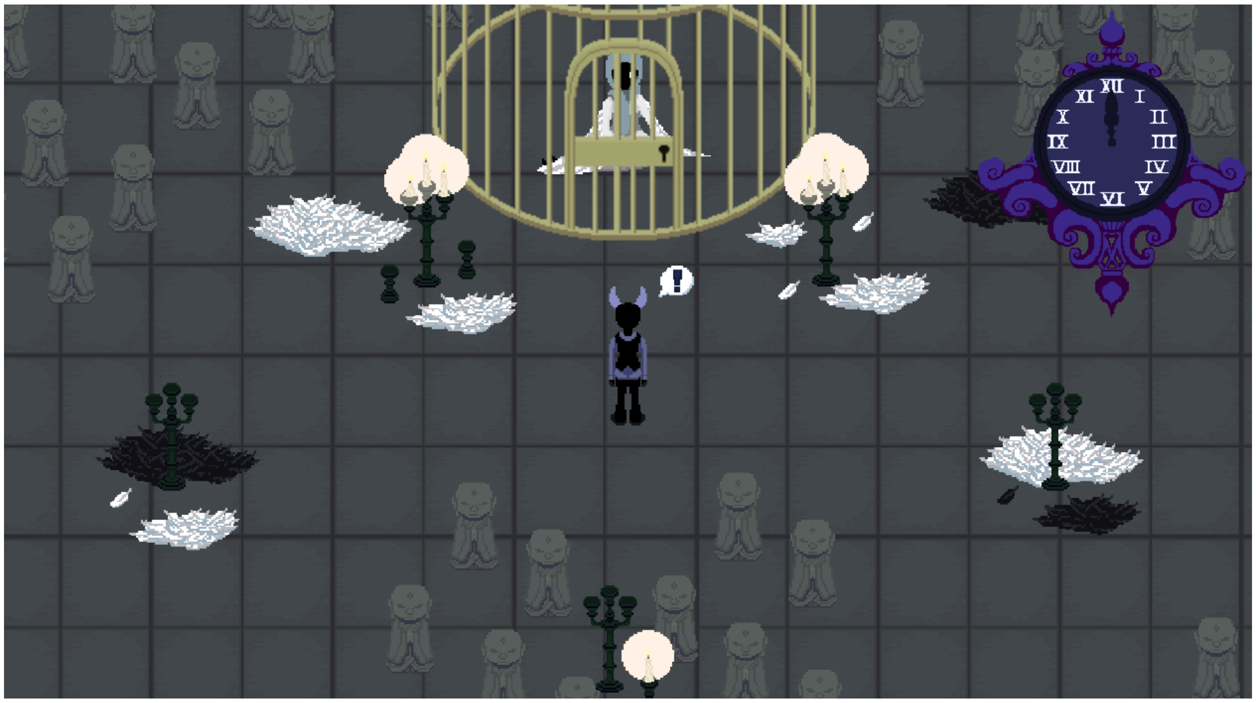
The first sub-area developed in the game, the Cherry Blossom Quest Line was key in creating visual diversity in the Forest, using bright reds and pinks to contrast the duller browns and greens of the main forest. During the development of this area, the wood texture was adapted to a different color for the first time and the methodology would later be replicated for things like the red wood used throughout the game such as in the shrine. It was also in the Cherry area that we decided to implement many Japanese elements into the game, such as the Torii gate, as a core part of the game's unique visual identity.



Theatre Quest Line

Design of the Area

The Theatre Quest Line was the last area developed for the game. The Theatre represents the culmination of all the other design decisions made for each area throughout the game. Decisions such as how to represent concrete structures and structural damage were utilized in the making of the Theatre area. The overall structure uses the simplified modern stylings that other objects in the game use while being layered with natural elements that while being more detailed, are not so different as to be jarring to see side by side. The enemies developed in conjunction with this area also received trickle-down benefits from past decisions as well, having an existing broad color palette to draw from for their design as well as a more mature animation pipeline for faster iteration and implementation into the game.



User Testing

After developing a working version of the Last Hymn game, our team conducted two rounds of User Testing. As a team we decided that user experience testing was imperative for developing the best video game possible as it allows for better fine tuning and honing in on our target audience. The purpose of the testing was to understand how a target audience would respond to the challenges in game. Participants talked through their experiences with the game, allowing us to gain a better understanding of what parts of the game are too challenging, too easy, or are hard to understand/navigate. Feedback from user testing was used to redevelop parts of the game to create a better user experience.

Our first round of testing was informal group sessions where groups of five people collectively played through the game and switched off frequently. This was done several times as a way to attempt to catch bugs that didn't emerge from our own playtests. It was also used to gauge interest in the systems of the game.

After removing any bugs found in group testing we performed individual testing. Participants were expected to spend between one to two hours playing through the entire sequence of the game. Data was collected during the play-through session by doing a "Think Aloud" where the participant was told to constantly talk through their thoughts as they played through the game. We did audio recordings during this time, as well as capturing the screen of the video game. No video recording of the participants was done. No previously collected data was used. No surveys or interview questions were administered. After the completion of the game or at two hours (whichever occurred first), the participant was stopped and thanked for their time and that concluded the study. The data was analyzed later for game play trends to analyze areas for possible improvement in the game.

15 users were run through the game with 10 in the group sessions and 5 in individual sessions. Our demographic was a young adult 20-24 years old who had some experience with video games and is a casual player. We had 12 females and 3 males play through the game in total with the group sessions having 9 females and 1 male while individual sessions were 3 females and 2 males.

Some of the major changes made from the user testing included:

- Making the default battle attack more readable by changing the skull pattern to a x-shaped pattern
- Adding in scalable difficulty to the enemies so that enemies are easier at the beginning of the game and more challenging at the end regardless of which order the player completes the questlines
- Added in more sound effects and changed level design to make the Forest easier to navigate and create better set pieces
- Updated dialogue that was identified as uninformative or not clear in intent

- Altered the number of fights that a player could expect to fight and retuned health and damage numbers of the player and enemies so that the amount of fighting necessary was reduced

User testing also led to the creation of a long bug list of issues that were unidentified prior which was very useful in providing a cleaner playing experience. After the initial playtest session, we made changes based off of the feedback and conducted a second round of user testing with different individuals to ensure the issues were positively corrected and impactful. Many more changes were made based off of user-testing but the above list includes the most impactful changes made as seen through our second playtest session.

All users revealed varying amounts of positive feedback on the game especially praising the game’s diverse environments and fun exploration of the world. However, in the first several rounds of testing the battle system was heavily critiqued. Most of it had to do with the frequency of encounters, as players started the game excited about combat but progressively went from “Oh cool you can fight enemies” to “Wow another fight” or “Do I really have to fight another one....”. These frequent progression of comments almost across the entire group led to the reduction of enemy encounters and then making fights easier to beat.

One user in particular mentioned that, “This game looks and feels great I could easily see myself paying money for this,” as part of their Think Aloud after considerable changes were made to previous issues while others echoed similar praise upon the game. It was at this point that the culmination of our changes from previou user testing started to show results.

Future Developments

In the future, we want to expand the game further adding in the following features:

- More time-based events and NPCs that have set schedules throughout the world
- Add side areas that tell smaller stories and award additional items that are not necessary to complete the game
- Expand on the lore of the world and implement a better story
- Provide a wider variation on enemy types
- Redevelop the Act System in Battles to be more rhythmically oriented
- Add in longer questlines that tell a more in-depth story that interweaves with the main story instead of being one-off questlines that have very little impact on the game world
- Optimize our code to allow for higher and better stabilized frame rates
- Include support for Mac and Linux Devices

We desire to make all of these changes with the express goal of releasing our product on Steam (http://store.steampowered.com/), an online game distribution platform. This would be done through a process known as Steam Greenlight (https://steamcommunity.com/greenlight) where the users of Steam would vote if they would like to play our game or not. If enough users give a positive response, Last Hymn can be released to the public. It is our belief that our game is of a high-enough quality and has an audience that is willing to pay for the experience we created if we can apply more polish to the core experience we have already created.

In addition to the mechanical changes, our other goal is to perform a complete overhaul of the game's code, refactoring it with better practices in mind. Our code has several areas where it could be made cleaner and allow for better development practices which would make future developments easier than the current codebase. A majority of the game's systems were made in isolation and developed over time when other helper classes may not of existed which makes our approach to the same problem in two different classes have two different approaches instead of a standardized single call. We began a refactor after all major systems were put in, which led to the creation of the Dialogue Repo system which has saved our team many hours. We would like to get the same time save by creating smarter systems throughout the rest of our code.

Reflection and Conclusions

Overall we are proud of the work we ended up producing. While we have created several games as a team before, this is easily the game that is of the highest quality and provides the best experience. Our game has a very strong aesthetic behind it that was noted in every single one of our playtests as well as having a unique gameplay that isn't offered in any other mainstream game. We believe that it has the potential to become something considered great if we make further enhancements to the already strong base we have created over the last two semesters.

Over the development period of the game, we ramped up the amount of development time we placed into the game, which led to some weeks of putting in a hundred or more hours a week between the three of us. In the beginning, our sprints were very small with a week being dedicated to tasks like "having movement implemented", "add in attacks for a single enemy", "create three trees", etc. This caused our later sprints to be filled with more items to be done which made for a harder crunch period. We also had to limit our scope halfway through in order to create a better player experience which we believe was mainly caused by having easier early sprints.

We also struggled in the beginning of the project in creating smarter systems that can be reused in other areas of the game. We had to refactor our battle code three times throughout the development cycle instead of spending the time to have it working properly in one cycle. We also added in extra features that were never utilized in the game instead of redirecting the same development time to other features that we wish could have been added but we did not have the time to complete.

However, this project has dramatically increased our skill levels across the board. On the programming side, we learned about shaders and how/when to utilize them, Bezier Splines, how to utilize the GPU to reduce CPU load, singletons, and other concepts that we were not exposed to in the classroom environment. The project allowed us to develop our ability to work in a group setting and to create clean, readable code so as not to inconvenience other members making our workflow more efficient. We had to develop systems in parallel and had to mesh all of our coding styles together to ensure consistency and that we were not breaking each other's code with our own systems.

Design is the area we improved the most as it was always our weakest aspect in previous projects. Particular care was placed into the level design and making each area feel distinct and memorable for the player. Some areas had their design reconfigured dozens of times in order to provide certain feelings to the player. Our game's aesthetic and design is the highest praised part of the project by players, which is the exact opposite recognition and a huge change from past projects. We also spent many meetings on discussing the lore of the world to ensure the game's world is consistent with itself and each of our visions of how the game would unfold.

Art was an area of concern for our team in the beginning as we noticed through planning that the amount of assets required to produce our game was considerably larger than initially expected. We had to learn to use proper forward planning in producing assets that were easy to animate while also fitting the aesthetic developed across the entire game. Proper color management by utilizing similar colors across the world to produce a sense of cohesiveness was critical for creating an experience that flowed together instead of disjointed areas. Each major areas needed to feel a part of the world and make thematic and structural sense. We had to work on both macro and micro scales balancing time spent producing large set pieces and smaller items to fill in the open fields of the Forest.

As a team we worked really well together and while the initial several months were slow production-wise by the end of development we were producing a large amount of content with very little confusion among team members. We meshed really well together and were able to fill in each other's knowledge and ability gaps. Our game has many different areas, fights, and quest lines each with their own gimmicks which went far beyond the original scope of the game which planned to only have two quest lines. We feel that there is no singular area of our game that stands out as particularly negative and that overall we produced a solid, worthwhile experience which is supported by user reviews of our game.

References

Curves and Splines, a Unity C# Tutorial - Catlike Coding. (n.d.). Retrieved October 20, 2016, from http://catlikecoding.com/unity/tutorials/curves-and-splines/

Fox, T. (n.d.). UNDERTALE. Retrieved October 20, 2016, from http://undertale.com/

Game Maker. (n.d.). Retrieved October 20, 2016, from http://www.yoyogames.com/gamemaker/

GitHub. (n.d.). Retrieved October 20, 2016, from https://github.com/

Paint.NET - Free Software for Digital Photo Editing. (n.d.). Retrieved October 20, 2016, from http://www.getpaint.net/index.html

Project YUMENIKKI. (n.d.). Retrieved October 20, 2016, from http://yumenikki.net/

Spriter - 2D sprite animation character creation software. (n.d.). Retrieved October 20, 2016, from https://brashmonkey.com/

Trello. (n.d.). Retrieved October 20, 2016, from https://trello.com/

Unity - Game Engine. (n.d.). Retrieved October 20, 2016, from http://unity3d.com/

Unreal Engine 4. (n.d.). Retrieved October 20, 2016, from https://www.unrealengine.com/

WinDEU. (2012, Oct 15). WinDEU Hates You 5EVR - Extra stage. [Video File]. Retrieved from https://www.youtube.com/watch?v=oQTzrWp9yoc.