Final Project Report: Drum-based VR Rhythm Game

Nathan Chay (200403221)
CS 458, University of Regina

1 Introduction

This report is a detailed overview of the research, design, and development process for my final project, a drum-based VR rhythm game built in Unity for standalone Meta Quest. The main gameplay involves 4 columns in which note blocks move towards the player. At the bottom of each column is a drum, which the player must hit in time as the note block approaches it. The player can hit the drum by simply moving the drum stick, which is attached to their controller, into the drum. The note blocks themselves are timed in such a way that they line up with the accompanying music, so that it feels as though the player is playing the song with their drums. The player is judged based on how many note blocks they can successfully hit in a row, and the timing accuracy of said hits. The game includes a level select menu, which allows the player to choose from a number of songs and difficulty levels, and an options menu, which allows the player to customize their gameplay experience.

2 Background

The gameplay is primarily inspired by *osu!mania*, a game mode of the popular free-to-play open-source PC rhythm game *osu!*. The gameplay of *osu!mania* itself is inspired by arcade rhythm games such as *Dance Dance Revolution* and *Beatmania*. All of the aforementioned games fall into a colloquial category of rhythm games known as vertical scrolling rhythm games, or VSRGs, in which the defining characteristic is that the notes scroll vertically towards the player in columns, and are intended to be hit at a certain point in said columns.

When deciding on an idea for this project, I immediately knew that I wanted to create a rhythm game. I am an avid rhythm game player, and creating one of my own has been a personal goal for a long time. In the current VR gaming space, the market for rhythm games is relatively untapped. The most popular VR rhythm game by far is Beat Saber, and it has virtually no competition. In fact, Beat Saber generated more revenue in 2021 than the other top 5 VR apps combined, proving that rhythm games have huge potential in VR (Wöbbeking, 2022).

My personal issue with Beat Saber is that it requires a lot of physical movement - players are required to swing their arms to hit notes and move to the side, duck, or jump to avoid obstacles. This becomes a problem for players like me who are confined to a very small VR playspace in which such movement can be infeasible or simply dangerous. For this reason, I decided to create a VR rhythm game that requires minimal movement. I chose a gameplay style similar to *osu!mania* simply for ease of implementation - the level files (known as "beatmaps") are made to be human-readable and easily parseable. I predicted that this would greatly reduce development time, as I would not need to design and impelment my own custom level and timing management systems. With this idea in mind, I began development on the project.

3 Development

3.1 Project Setup

The project was developed entirely in Unity using Meta's official Oculus Integration SDK. I chose Unity simply due to its relevance to our course. As my level of familiarity with Unity was fairly low, and my familiarity with C# was non-existent, I knew that the learning curve would be steep regardless. By choosing Unity I at least had the assurance of learning the basics through the course as opposed to being

entirely on my own. The Oculus Integration SDK was also chosen due to its relevance to the course, but it is extremely useful in its own right. The SDK implements all of the functionality needed for a VR game, primarily head/controller tracking and simple interactions. By using the SDK, I was able to focus entirely on the game itself without having to worry much about VR-specific considerations.

3.2 Testing

The majority of testing for this project was simply done via mouse and keyboard through Unity's play mode. I implemented controls for the drums via keyboard by simply mapping a key to each drum, and I added an alternate input handler to allow the mouse to interact with the world space UI. For VR testing, I opted to use Quest Link. When the Quest was linked to my PC, I was able to simply use Unity's play mode to control the game via the Quest, despite the game not yet being built for Quest. Since build times were upwards of 5 minutes on average, this significantly reduced the time cost of testing. Despite this, the game still occasionally needed to be built and ran on the Quest, since inputs via Quest Link suffered from latency issues.

3.3 Basic Mechanics

3.3.1 Models and Collision

The first step in development was simply creating the basic gameplay mechanics, in which the notes move towards the player, and the player can hit the notes by swinging a drumstick into the corresponding drum at the right time. For the models of the drum and drumsticks I simply used plain, untextured cylinders. I made the drums semi-transparent so that the player can see their drumstick enter the cylinder. For the drumsticks, I simply parented the cylinders to the controller children of the OVR camera rig prefab. The OVR camera rig is included with the Oculus Integration SDK, and handles head and controller tracking. By parenting the drumsticks to the controller children, the player can swing the drumsticks by simply moving their controller. Collision detection was covered in the course and therefore was easy to implement. When a drumstick collides with the drum, the drum turns red and a hit sound is emitted to inform the player that the drum has been hit. Finally, I added some basic thumbstick locomotion so that the player can adjust their position relative to the drums depending on their VR playspace.

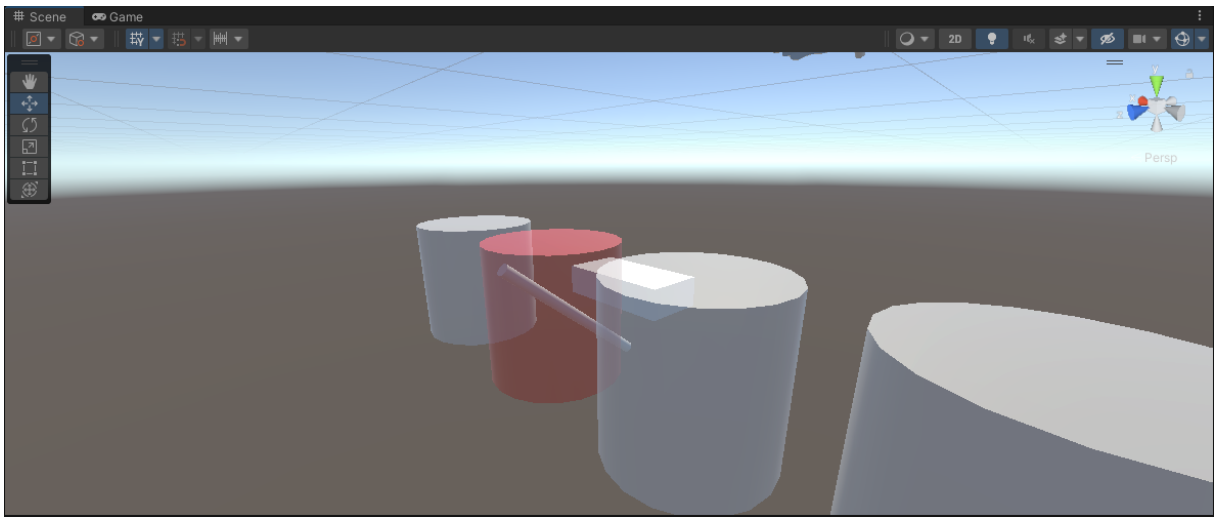

Figure 1: Drum stick colliding with a drum, turning it red.

3.3.2 Note Spawning

The next step was to spawn the note blocks so that the player has something to hit. The spawning mechanism is fairly simple. The note blocks themselves are prefabs which are instantiated by the note controller. The spawning function takes a drum index parameter, which it uses to find the corresponding game object and calculate the position at which the new note block should be spawned. The note block

prefabs have rigidbody components, and move towards the player with a fixed velocity value. An initial issue with this approach was that the note blocks would collide with the drums and stop moving. This was due to the drums and the note blocks both having rigidbody components - similarly, the player could knock note blocks out of their columns by hitting them with the drumsticks. This was simply fixed by placing the note blocks on their own collision layer, so they phased through the drums as expected. Finally, an invisible block was placed behind the drums which destroyed the note blocks on collision.

3.4 File Parsing

Before starting work on hit detection, I wanted to add a proper level into the game, which necessitated parsing of *osu!mania* beatmap files. As mentioned earlier, *osu!mania* beatmap files are made to be human-readable and are therefore fairly easy to parse. All of the note data is contained in a section of the file with a header of `[HitObjects]`. Each line in this section represents a single note and its values are comma-separated. This facilitates trivial parsing of the notes using C#'s `string.Split()` method.

```
...
[HitObjects]
64,192,584,5,12,0:0:0:0:
448,192,584,1,0,0:0:0:0:
448,192,935,1,12,0:0:0:0:
64,192,935,1,0,0:0:0:0:
...
```

Listing 1: Notes from an osu!mania beatmap file.

For the sake of this project, there are only two values of interest: the first, which is the column index of the note, and the third, which is the time in milliseconds at which the note should be hit. The column index value needs to be computed through an algorithm after being parsed: `floor(x * columnCount / 512)`. It is important to note that this project was only developed with 4 columns in mind, and therefore the `columnCount` value is fixed at 4. In *osu!mania*, there are types of notes which need to be held down by the player instead of briefly tapped - the length of said hold notes is defined by the 6th value in its line (osu! wiki, n.d.). Since there is no such type of note implemented in this project, that value can simply be ignored and there is no workaround needed for dealing with hold notes. This was another huge benefit of working with *osu!mania* beatmap files for this project. After parsing the notes from the beatmap file, I placed them into a struct to easily handle them later.

```
...
private struct Note
{
    public int time;
    public int drum;

    public Note(int time, int drum)
    {
        this.time = time;
        this.drum = drum;
    }
}
...
```

Listing 2: The note struct.

3.4.1 Note Timing

In order to spawn the notes on time I opted to use coroutines, which are essentially just functions that execute asynchronously after a specified amount of time. Timing the spawns of the notes was a bit tricky, since the time defined in the beatmap file is the time when the note should be hit. If notes were spawned in at that exact time, they would arrive to the drums late; therefore, the notes have to be spawned in early so that they can arrive at the drum on time. The time a note travels from its spawn point to the drum can simply be calculated by `noteBlockSpawnDist / velocity`, where `noteBlockSpawnDist` is the distance between the note block spawner and the drums, and `velocity` is the velocity of the note. This presents another problem, however: if the accompanying song is set to play when the scene is loaded, there is no time to spawn notes early. The simple solution is to delay the music by the same amount of

time that the notes would be spawned in early - this is feasible since said time is the same for every note and therefore does not need to be calculated on a per-note basis. With these considerations in mind, the parsing block is complete.

```
...
List<Note> notes = new List<Note>();
string text = map.text;

// 14 = length of [HitObjects] + 2
int hitObjectsStart = text.IndexOf("[HitObjects]") + 14;
text = text.Substring(hitObjectsStart);

string[] lines = text.Trim().Split('\n');

foreach (string line in lines)
{
    string[] lineSplit = line.Split(',');

    // Drum no. conversion algo from https://osu.ppy.sh/wiki/en/Client/File_formats/Osu_%28file_format%29
    Note newNote = new Note(Int32.Parse(lineSplit[2]), Int32.Parse(lineSplit[0]) * 4 / 512);
    notes.Add(newNote);
}

StartCoroutine(StartMusic(noteBlockSpawnDist / velocity));

foreach (Note note in notes)
{
    float timeToSpawn = (note.time + noteDelayMs) / 1000;

    StartCoroutine(SpawnNote(note.drum, timeToSpawn));
}
...
```

Listing 3: The note parsing block.

3.5 Hit Detection

At this point in development, notes were successfully spawning in time with the music, and the player was able to interact with the drums. The next step was allowing the player to hit the notes by hitting the drum on time. I initially took the simplest approach to implementing this by checking whether the note was positioned inside of the drum when the drum was hit. After some testing I realized that the window for a successful hit was very small, due to the small size of the drum object. Instead of explicitly checking if the note block is inside the drum, I extended the position range for a successful hit to be slightly outside the confines of the drum object. This approach worked well at first, and I added the first UI element for the game: a hit counter which incremented when a note was successfully hit, and a miss counter which incremented when a note collided with the invisible note-deleting block.

3.5.1 Timing Windows: Background

Before starting on a more complex scoring system, I needed a way to judge the timing accuracy of the player's hits in a more detailed manner than simply hit or miss. Most rhythm games, *osu!mania* included, incorporate the idea of a timing window. A note can be successfully hit at any time in a timing window, but the closer a player's hit is to the edge of the timing window, the worse their judgement they receive for that hit is. Traditionally, a perfect hit is in the exact center of the timing window and the judgments on both sides are even, but some rhythm games choose to deviate from this.

With my original hit detection implementation, it was possible to create a hit window by simply extending the range of the position check further, and calculating the judgment based on how far from the center of the drum the note was hit. There is a glaring flaw in this implementation, though: a low note velocity value would make the hit window unreasonably large and a high velocity value would make the hit window extremely small. Since I intended to make the velocity value adjustable by players, the hit detection method needed to be changed.

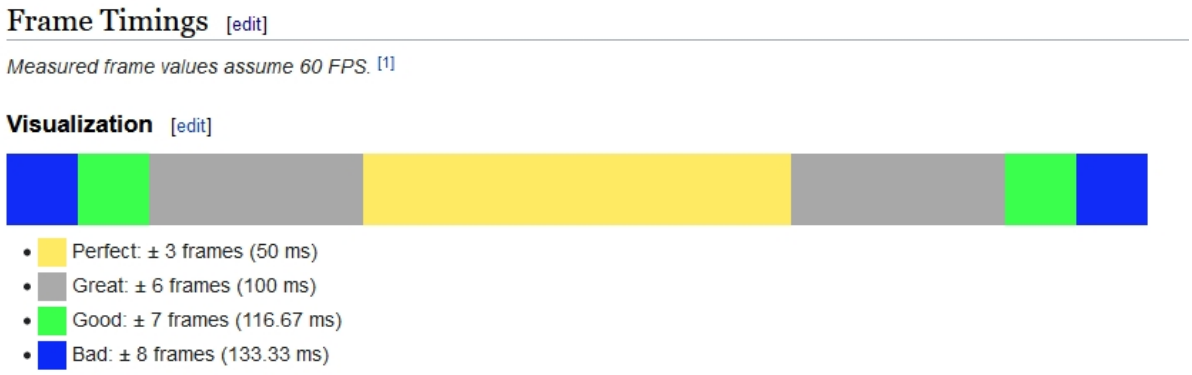

Figure 2: Example of what a rhythm game timing window might look like (u/Safyire, 2022).

3.5.2 Timing Windows: Implementation

My first proposed alternative to hit detection was to store a value in each individual note that holds the elapsed time since it was spawned. When a drum is hit, the time-since-spawn value of the nearest note in its respective column could be checked and the judgement for that hit could be determined. Since the values would need to be updated every frame, I was concerned that this approach might cause performance issues. I decided to simplify the value stored in each note to a single character representing the judgement that would be awarded when it is hit. This way, the value only needed to be updated a few times during the note's lifespan. This was easily achieved by using coroutines to wait the length of each timing window phase before applying the judgement change to the note object. For the judgements themselves, I went with a four-tier system consisting of perfect, good, OK, and miss. For sake of simplicity, I set each judgment phase to last an equal amount of time.

```
...
if (beforeHalf)
{
    switch (nextJudgement)
    {
        case 'o':
            StartCoroutine(HitWindowStartPhase(obj, timeToNextPhase, 'g', true));
            break;
        case 'g':
            StartCoroutine(HitWindowStartPhase(obj, timeToNextPhase, 'p', true));
            break;
        case 'p':
            StartCoroutine(HitWindowStartPhase(obj, timeToNextPhase, 'g', false));
            break;
    }
}
else
{
    switch (nextJudgement)
    {
        case 'g':
            StartCoroutine(HitWindowStartPhase(obj, timeToNextPhase, 'o', false));
            break;
        case 'o':
            StartCoroutine(HitWindowClose(obj, timeToNextPhase));
            break;
    }
}
...
```

Listing 4: Switch statement portion of the timing window management function.

3.5.3 Hit Feedback

Once this approach had been implemented, I realized that it was not only extremely confusing but that the repeated use of coroutines could still cause performance issues. Despite this, I decided to leave it

as-is so I could focus on adding other functionality to the game. The final step was to add some feedback for the player, so they would be aware of the timing accuracy of their hits. Previously, the drum would turn red when hit, but I added some additional colors for the judgements: yellow for perfect, green for good, and purple for OK. I set the red color to appear if the player hits a drum and there is no note in the column to be hit. If a player misses a note, the color of the drum does not change color. This was an oversight that I was aware of, but did not bother to change due to time constraints. A future approach may be to show UI elements above each drum indicating the judgements for each note.

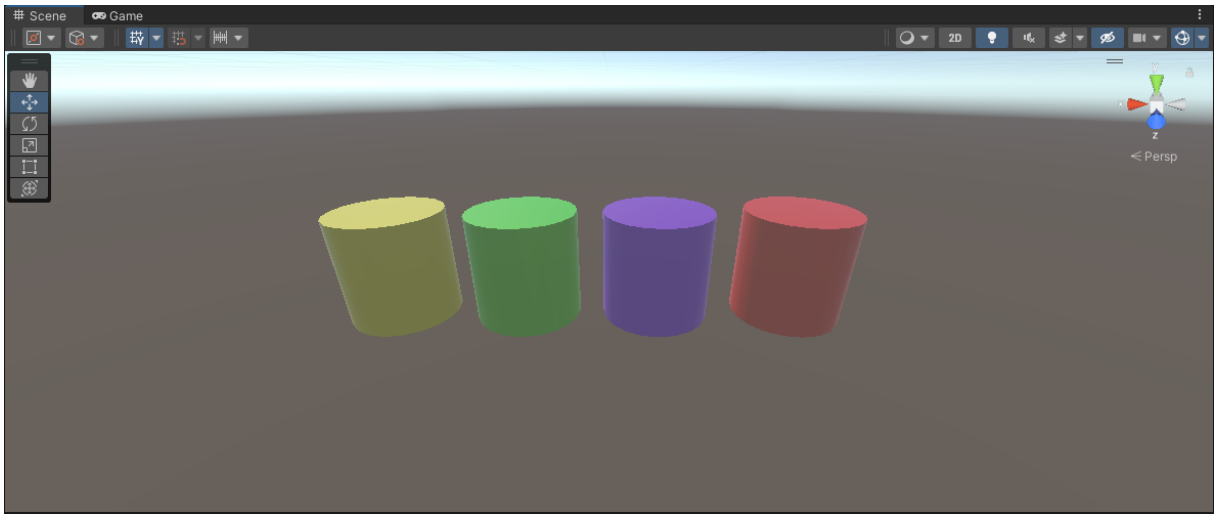

Figure 3: The judgement-based drum colors.

3.6 Scoring

With a proper timing window and judgement system implemented, it became possible to create a scoring system. The scoring system I had initially proposed consisted of three parts: numeric score, hit combo, and timing accuracy. I began work on the accuracy system first. I took a similar approach to the accuracy system in *osu!mania* by simply assigning a percentage value to each judgement: 100% for perfect, 75% for good, 50% for OK, and 0% for miss. I made sure to serialize each judgement-percentage value, making it appear in Unity's inspector, to ensure that I could easily tweak these values. Since the hit counts for each judgement were already being tracked, I set the accuracy value to be calculated as a simple average over all hits.

```
...
float hitTotal = oks + goods + perfects + misses;
float accTotal = oks * okAcc + goods * goodAcc + perfects * perfectAcc;
acc = accTotal / hitTotal * 100f;
...
```

Listing 5: Formula for accuracy calculation.

Implementing the hit combo was trivial. This was done by simply incrementing the hit counter by 1 until a note is missed, then resetting the hit counter to 0. With an accuracy and combo system in place, I was able to implement numeric scoring. Initially, I assigned score values to each judgement similarly to the accuracy system: 10 for perfect, 5 for good, 1 for OK, and 0 for miss. When a note was hit, the score value for that judgement was multiplied by the current combo, and the result was added to the score total. I quickly realized that this formula was extremely biased towards combo - a player's final score would be almost entirely dependent on the maximum combo they achieved during that play. Since I wanted the numeric score to equally represent accuracy and combo, I added an adjustable `comboNerf` value which simply acts as a divisor to the current combo. This significantly improved the balance of the numeric scoring system.

3.6.1 Gameplay UI

With the entire scoring system in place, the final step was to add some UI so that the player could keep track of their score, accuracy, and combo while playing. I took a similar approach to *osu!mania* and most other VSRGs when deciding where to place the UI components. The score and accuracy are placed to the left of the note columns. This ensures that the player can easily check their accuracy or score during gameplay, as they only have to move their eyes slightly away from the playfield - or, with practice, they can read the values in their peripheral vision. The combo on the other hand is placed directly on top of the note columns so that the player has conscious knowledge of their current combo at all times. Since the combo will rarely exceed 3 digits and never exceed 4, this is non-obstructive to gameplay.

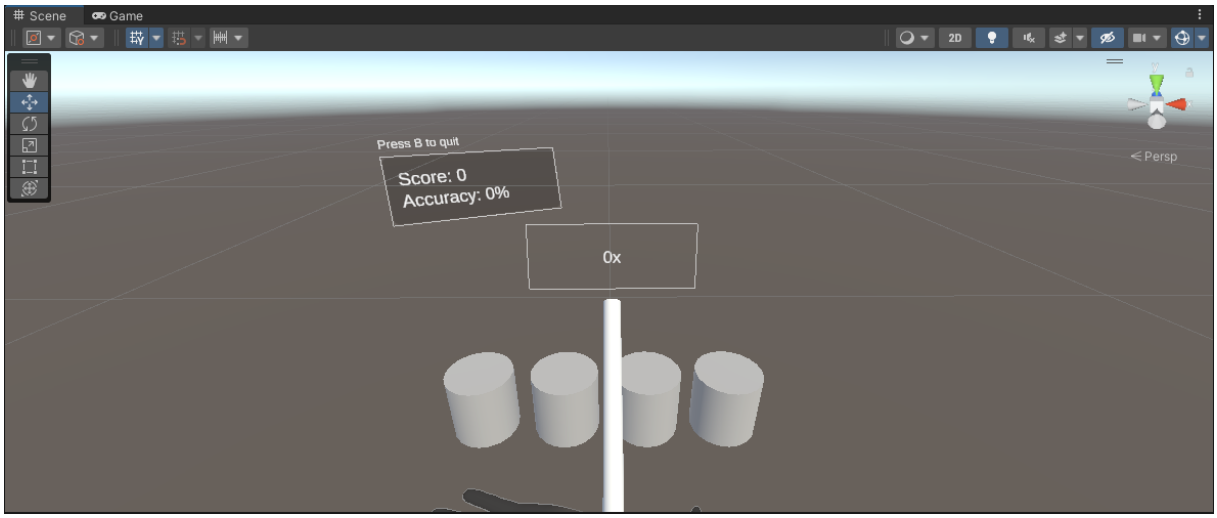

Figure 4: The finalized gameplay UI.

The next step was to create a post-song results screen, where the player can see a full detailed overview of their performance. This was fairly simple to implement, as all of the values present on the proposed results screen were already being tracked. The only addition was a calculated letter grade, based on the player's overall accuracy - this is yet another feature commonly seen in rhythm games. I assigned a letter grade to a number of accuracy thresholds: 100% for S+, 99%+ for S, 90%+ for A, 80%+ for B, 70%+ for C, 50%+ for D, and F for anything under 50%. With everything in place, I designed the results screen prefab and set it to appear at the end of the song, while simultaneously hiding the gameplay UI. Alongside the values shown during gameplay and the letter grade, I added a detailed breakdown of judgements and information about the song itself.

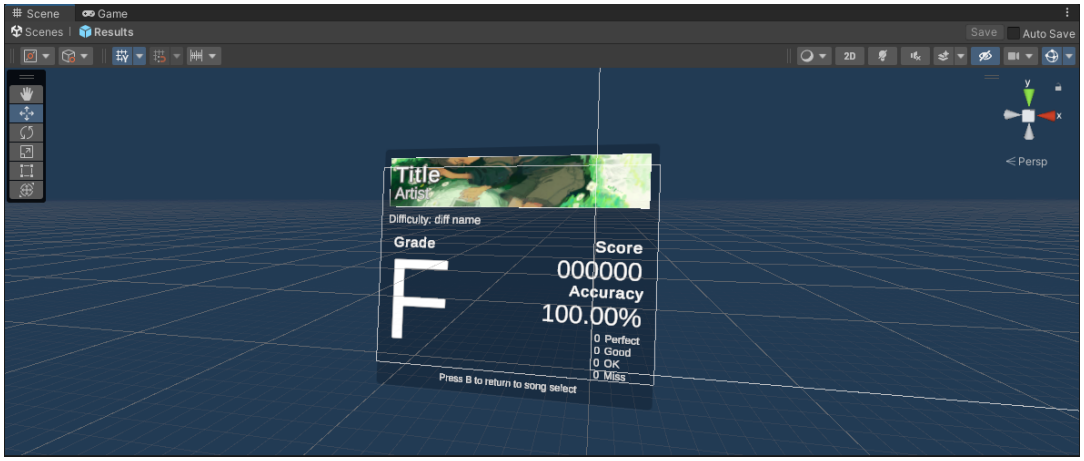

Figure 5: The post-song results screen.

3.7 Level Select

3.7.1 UI Setup

At this point in development, I was satisfied with the main gameplay. I decided that I would switch my focus to adding content rather than gameplay functionality. In order to add content to the game I first had to add a menu to access said content, so I began work on a level select system. First, I had to find a way to make sure that all of the additional content was present in the game at build time. By default, Unity does not include all assets when a project is built - an asset must be associated with a game object for it to be included. There are workarounds to this, such as the deprecated `Resources` folder, or the newly added `AssetBundles` feature, but after looking into these I decided that they were too complex for the purposes of this project. My final approach was to simply associate all of the song and map assets with a single game object, even though this meant I had to manually add all assets for a new song into the inspector window.

Figure 6: Organization of the maps in the inspector window.

For each song, I needed the song file itself, a banner image, and each difficulty as a text file. This resulted in a pretty complex setup, but it allowed me to dynamically populate the level select UI. I first created a prefab for the song select button, simply displaying the artist and title of the song. Alongside hit object information, *osu!mania* beatmap files contain metadata for the associated song, so I opted to parse the files instead of manually inputting the artist and title for each song. I then mapped the array of songs from the inspector into a list of buttons, and added functionality so that details for a song appear on the right when a button is clicked.

3.7.2 Song Details

The details panel was relatively simple to design. Two important attributes I wanted to include alongside the artist and title were the length and BPM of the song. Getting the length of the song was trivial, as Unity's `AudioSource` component contains a length field which can be read and converted into mm:ss format. For the BPM, I had to parse the beatmap file once more. *osu!mania* beatmap files also contain a `[TimingPoints]` section that is visually similar to the `[HitObjects]` section, but each line represents a point where a BPM is either explicitly set or inherited from a previous point (in order to change other timing-related options). For the purposes of simply obtaining the song's BPM, I only needed to parse the first timing point in the file. At this point I decided to parse each file only once, and store the song's metadata in a struct associated with the rest of the song's assets.

```
...
[Serializable]
public class MapInfo
{
    public AudioClip song;
```

```
    public Texture2D banner;
    public TextAsset[] difficulties;
    public Metadata metadata;
}

public struct Metadata
{
    public string title;
    public string artist;
    public int length;
    public float bpm;
    public float previewPoint;
}
...
```

Listing 6: Definitions for the MapInfo class and Metadata struct.

The next step was to populate the difficulties list for the selected song. I used a similar approach to the song list by simply creating a button prefab and instantiating it for each difficulty. Each button displays the difficulty name (parsed from the beatmap file) and a notes per second (NPS) value. The NPS value is calculated by simply counting the lines in the `[HitObjects]` section of the file and dividing by the song length in seconds. For added accuracy, I trimmed the time before the first note and the time after the last note, so that any note-less intro and outro sections would not artifically lower the NPS value. The purpose of the NPS value is to provide players with a consistent method of comparing difficulty levels between songs, as *osu!mania*'s difficulty naming system is rather unconventional.

Figure 7: The populated level select screen.

3.7.3 Interaction

The next step was to add VR interactivity to the UI. In this course, we were taught a method of adding interactivity using physics raycasting; however, this method was simply a workaround to the lack of proper VR support for UI interaction at the time that the course was designed. I decided to try the newly-supported graphics raycasting system, as the Oculus Interaction SDK has a built-in implementation of it. Unfortunately, this turned out to be the most difficult part of the project overall. Meta's documentation for this functionality is barely existent, and the documentation that does exist is very incomplete. I had to watch multiple obscure YouTube tutorials to get the interactivity working, and I was stuck for a long period of time due to a collision box that I had accidentally made extremely small. It was very relieving to finally get the UI interaction working in VR.

The final step was to simply transmit the data to the main gameplay scene. I did this by simply creating a map manager object with a `DontDestroyOnLoad()` call in its script, allowing its properties to persist between scenes. I also had to make some minor changes in the gameplay scene to load the data from the map manager object. Finally, I added the ability for the player to return to the level select screen at any time by simply pressing B on their controller, and some additional UI elements to indicate this functionality.

3.8 Options

The final feature I had time to implement in the project was an options menu. I decided to place this menu to the right of the level select screen, so that the player could easily adjust their settings right before playing a song. The first options I added were note delay and note velocity. The note delay is extra time that is waited before a note is spawned, in order to compensate for timing errors on different hardware setups. Note velocity simply refers to the speed at which the notes travel from their spawn point to the drums. Both of these options are commonly found in rhythm games. Since these values were already serialized in the gameplay scene, it was simple to transfer and overwrite them via the map manager object.

The next option I added to the menu was a toggle for autoplay. In autoplay mode, the player can no longer interact with the drums - instead, the drums are automatically hit at the right time, obtaining a perfect score. This feature is mostly just for entertainment, but I found it particularly useful for finding the right note delay value for my system. Finally, I added a volume slider while recording takes for the demonstration video, for the sole purpose of allowing myself to talk while on the level select screen. I had previously omitted it since volume is easily adjustable via the Quest headset, but I think it is a useful feature regardless.

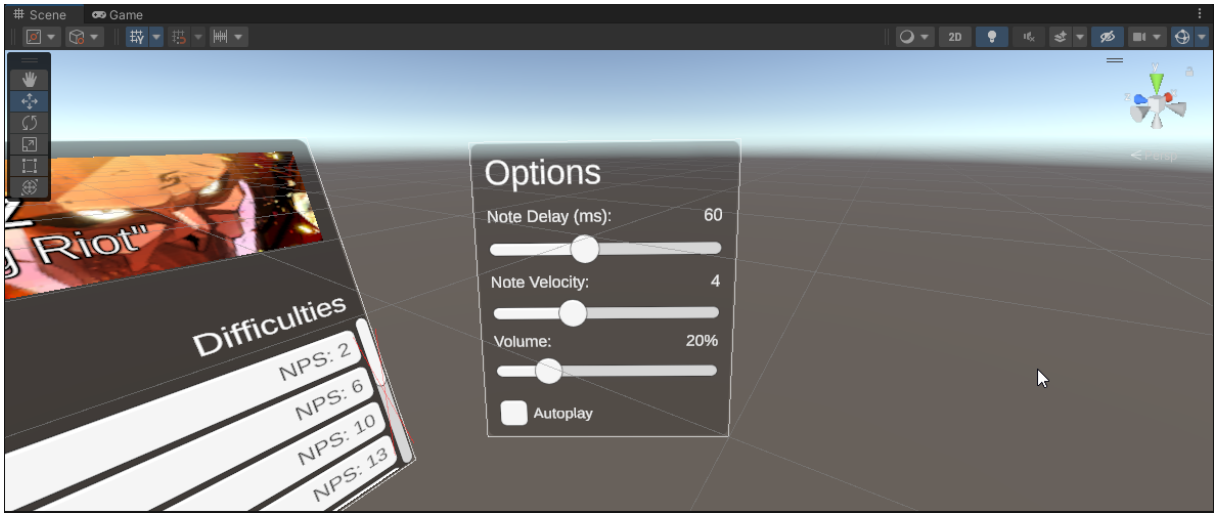

Figure 8: The finalized options menu.

4 Milestones

4.1 Milestone 1: Basic Gameplay

This first milestone concerns the basic mechanics of the gameplay. The goals for this milestone were very simple, including having a single song and difficulty level, four drum objects, notes spawning and moving towards the drums, players being able to hit the notes when they arrive at the drums, hit and miss indicators, and simple untextured objects. All of these goals were achieved very early in development, and I did not have any particular struggles here.

4.2 Milestone 2: Scoring System

This milestone concerns the scoring system for the game. The goals included are timing accuracy calculation, numeric score calculation, a post-song results screen, and an HP system. I achieved all of these goals except for the HP system. The idea for the HP system was initially taken from *osu!* but can be found in other rhythm games as well. The player has a health bar that decreases when a note is missed and replenishes when a note is hit. If a player reaches 0 HP, the gameplay is stopped and they are taken to a game over screen. I chose not to implement this feature simply because I personally think it is better to allow the player to continue playing an entire song regardless if it is above their skill level. This

milestone was marked as the "vertical slice", suitable for the in-class demonstration, and I am satisfied that I was able to achieve it.

4.3 Milestone 3: Menu and Level Selection

This milestone involved implementing UI allowing for additional game content. The goals include a title screen, level select with multiple songs, difficulty levels, and song details, and a leaderboard display for previous scores. Despite its simplicity, I chose not to implement the title screen since, with my limited time, I wanted to focus on functionality over flair. Implementing the leaderboard was a strong consideration for me, but I figured that it would most likely involve reading and writing text files and designing a whole new UI section, which would have been too time consuming.

4.4 Milestones 4-6

Milestones 4-6 mostly concern special effects and stretch goals, including animations for UI and object interactions, properly modeled and textures objects, 2 - 8 drum support, alternate note types, etc. I have grouped these together in one section because I did not achieve any goals in these milestones with the exception of the options menu. In hindsight, the options menu was fairly easy to implement and would have been better suited for an earlier milestone. As for the other goals, the most easily achievable ones were mostly cosmetic upgrades, and the unreasonable ones could be entire projects on their own (level editor, online leaderboards). Throughout development, I tried to focus on reasonably achievable functionality, which is why none of these features were implemented.

5 Future Development

In its current state, the project is not suitable for public release. Despite this, I believe it is a very solid foundation and could become suitable for public release with more development time. The most important features I would like to add would be mostly those that I missed in earlier milestones such as leaderboards and a title screen. Additionally, support for 2 - 8 drum songs and alternate note types would add a lot of variety to the game. Online leaderboards would also be a priority for a public release. Finally, the game would need an entire graphical overhaul with proper models, textures, artwork, and effects. As I am very passionate about rhythm games, I genuinely enjoyed developing this project and may actually continue work on it in the future. An alternative approach may be making it open-source, so that players could contribute to it and speed up development. Regardless, I am very satisfied with the outcome of this project overall and I am excited to potentially take it further in the future.

References

osu! wiki. (n.d.). *.osu (file format).* Retrieved from https://osu.ppy.sh/wiki/en/Client/File_formats/Osu_%28file_format%29

u/Safyire. (2022). *A "data wiki" that covers timing windows and other important data for rhythm games.* Retrieved from https://www.reddit.com/r/rhythmgames/comments/r7sop6/a_data_wiki_that_covers_timing_windows_and_other/

Wöbbeking, J. (2022). *Beat saber generated more revenue in 2021 than the next five biggest apps combined.* Retrieved from https://mixed-news.com/en/beat-saber-generated-more-revenue-in-2021-than-the-next-five-biggest-apps-combined/