Overview

For "after us, the flood"--a narrative game with rhythm mechanics made in Unity--I was tasked with re-engineering and iterating on the rhythm game since it was buggy and difficult to play. The problems were caused by a mix of technical design and game design. Originally, it was implemented using physics to detect when notes were overlapping with the hit zone. I re-implemented it using a clock to detect when timing windows were open. The choice to re-engineer it was because physics and colliders often incorrectly detected overlaps.

How to Play the Rhythm Game

The Rhythm Game is played with WASD and Arrow Keys and the Player must hit the correct combination. For example, if the combination is two arrows pointing up, the Player must hit the W and Up Arrow simultaneously.

The Rhythm Game has two phases:

1. In Phase 1 (fig.1), the Fret (the pink circle) will show the combination the Player must hit. Phase 1 lasts 10 combinations. Combinations are scripted. Failing will restart the rhythm game.
2. In Phase 2 (fig.2), the Fret will show the combination the Player must hit and notes will also move in from the right side of the screen. Phase 2 lasts the rest of the song. Combinations are randomly generated. The Player has 5 lives and failing 5 times will restart the rhythm game.

Game Design and Balancing

Most players initially reported that the Rhythm Game was very difficult and felt like the system was working against them. I playtested the game myself and facilitated playtests to iterate on the game. These were the major changes implemented in the final game:

- Rhythm game is played by hitting every other beat. Originally played hitting every third beat, but playtesting showed that players were more inclined to hit every other beat.
- UI sprites with arrows are a venn diagram shape. Originally a single circle shape, but playtesting showed that players were having a hard time deciphering which arrow represented which hand when it was one circle.
- Increasing timing windows. Playtesting showed that players enjoyed the game more when the timing of hitting a beat was more forgiving, rather than being given more chances.

- Bias in how often combinations show up. Playtesting showed that players struggled the most with combinations like W and Left Arrow (where the arrows do not point in the same or direct opposite direction), so the easier combinations show up more frequently.

Technical Design

Pages 4-12 show the script for controlling the Rhythm Game. These parts of the code have been cut for brevity:

- Calculating offsets caused by the clock.
- Transitioning between intro and closing animations + intro and closing animations.
- Calls to packages used.

Components not managed by the RhythmGameController

- **The Clock**. There is a clock that begins counting at the beginning of the Rhythm Game. The clock returns the current measure and beat at any given point in the song.
- **Note Objects**. Each note object stores its own combination. Combinations are generated by the RhythmGameController

Components managed by the RhythmGameController

RhythmGameController is the event manager for the Rhythm Game.

- **The Song**. A 2D array. The outer array is measures, the inner array is beats. The game is played by hitting every other beat. The first and third elements of each inner array holds a Note Object, the second and fourth hold null values.
- **Timing Window Management**. A State Machine with two states: InWindow and OutOf Window.
    - **InWindow**: Lasts 96 ticks. Player can hit a correct combination. The WASD and Arrow Key do not need to be hit on the same frame, but must be within the window. Once a key is pressed, it is registered and cannot be changed. For example, if the player hits W and then A, the A does not register. Once a key press is registered on both hands, the window closes, even if the InWindow 96 ticks are not over.
    - **OutOfWindow**: Lasts 96 ticks. Player can do nothing. Hitting keys will not result in a penalty.
- **Player Input Evaluation**. Compares the Player entered combination to the current expected combination. The current expected combination is retrieved by getting the current Measure and Beat from the Clock and then evaluating the 2D array that

represents the song. The combination is read from the Note Object and returned to the RhythmGameController.

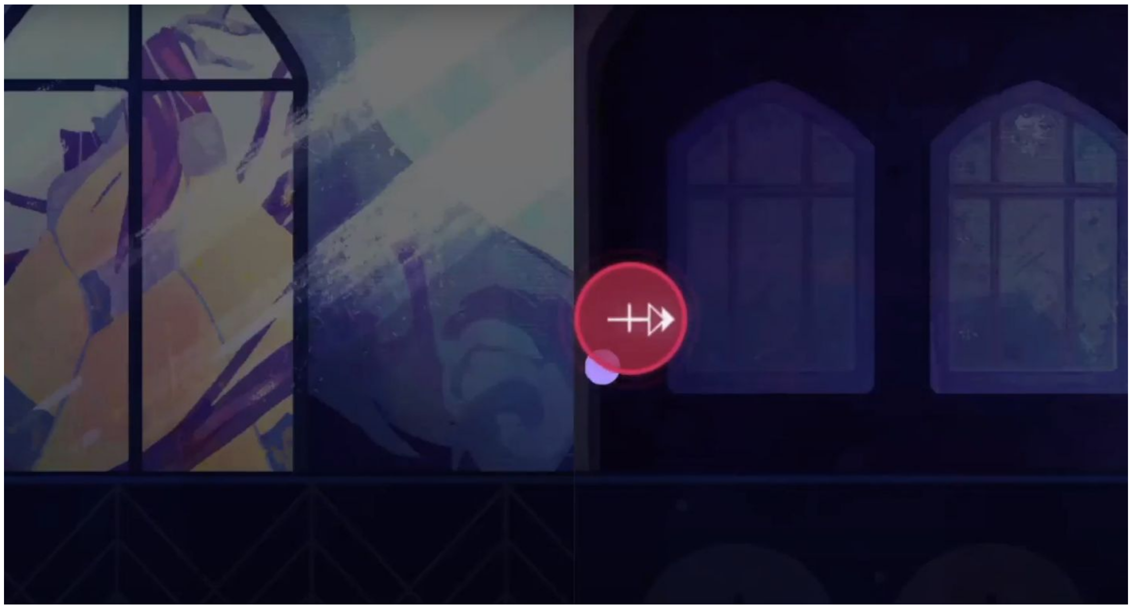

Fig. 1. Phase 1 of the Rhythm Game. The Fret shows the combination for the player to press.

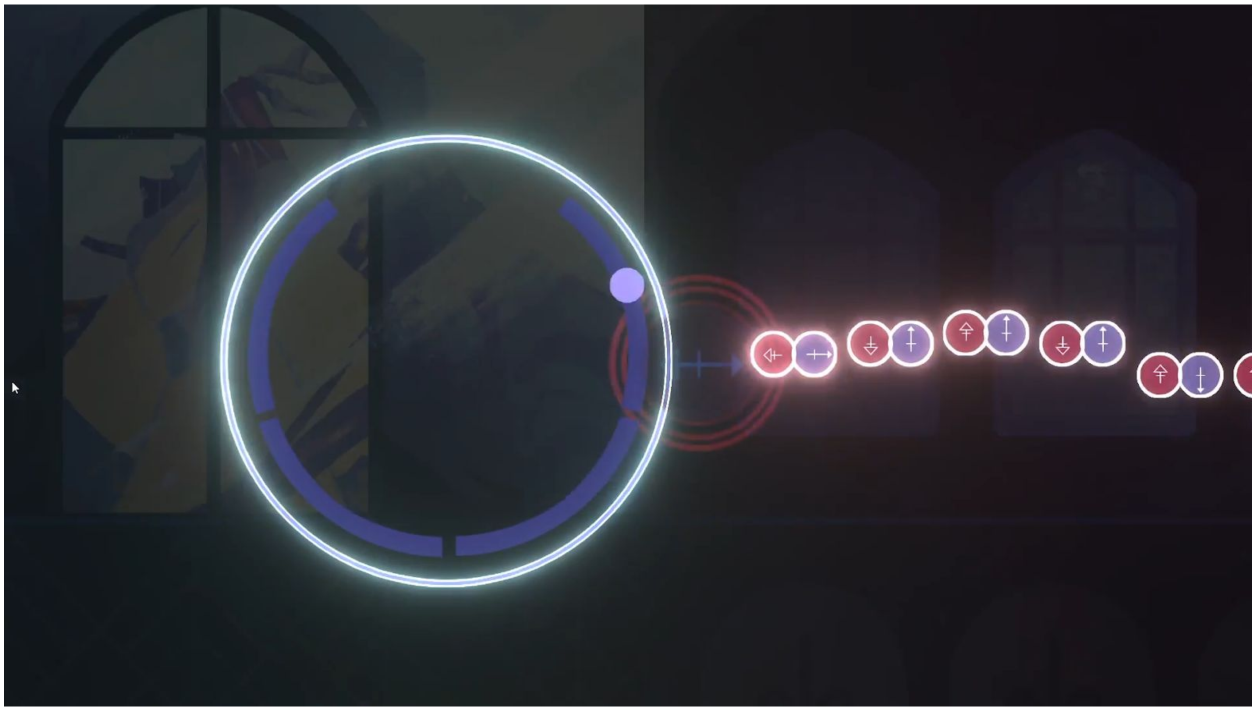

Fig. 2. Phase 2 of the Rhythm Game. The Fret shows the combination for the player to press as well as notes moving in from off screen.

```
public class RhythmGameController : MonoBehaviour
{
    FiniteStateMachine<RhythmGameController> rhythmGameStateMachine;

    List<string> notesCombinations = new List<string>() { "UU", "DD", "LL", "RR", "UD", "LR", "UU", "DD", "LL", "RR", "UD", "LR" }; //first 10 notes are scripted

    string[] mostLikelyCombos = { "UU", "DD", "LL", "RR" }; //these have a 40% chance of appearing
    string[] secondLikelyCombos = { "UD", "DU", "LR", "RL" }; //these have a 40% chance of appearing
    string[] leastLikelyCombos = { "UL", "UR", "DL", "DR", "LU", "LD", "RU", "RD" }; //these have a 20% chance of appearing

    //outer array represents measures, inner array represents beats per measure.
    GameObject[,] thisSong = new GameObject[77, 4];
    string[] thisSongSequence;

    public int phase1Threshold;

    public int currMeasure;
    public int currBeat;

    private bool completed = false;

    void Start()
    {
        rhythmGameStateMachine = new FiniteStateMachine<RhythmGameController>(this);
        rhythmGameStateMachine.TransitionTo<IntroAnimation>();

        //generate the random combination for the second phase of the song and make the song into one string
        GenerateCombinations();

        this.thisSongSequence = notesCombinations.ToArray();

        GenerateNotes();
    }

    void Update()
    {
        //CODE CALCULATING CLOCK OFFSET OMITTED
        currMeasure = SimpleClock.Instance.Measures;
        currBeat = SimpleClock.Instance.Beats;

        rhythmGameStateMachine.Update();
    }

    //generate the list of combinations (strings)
    private void GenerateCombinations()
    {
        int combosToGenerate = 154 - notesCombinations.Count;
        string thisNotesCombo = "";

        for (int i = 0; i < combosToGenerate; i++)
        {
            //set a bias, certain combinations are more likely than others
            int comboBias = Random.Range(0, 5);
            int getComboIndex = 0;

            if (comboBias == 0 || comboBias == 1)
            {
                getComboIndex = Random.Range(0, mostLikelyCombos.Length);
                thisNotesCombo = mostLikelyCombos[getComboIndex];
            }
            else if (comboBias == 2)
            {
                getComboIndex = Random.Range(0, secondLikelyCombos.Length);
                thisNotesCombo = secondLikelyCombos[getComboIndex];
            }
            else
            {
                getComboIndex = Random.Range(0, leastLikelyCombos.Length);
                thisNotesCombo = leastLikelyCombos[getComboIndex];
            }

            notesCombinations.Add(thisNotesCombo);
            thisNotesCombo = "";
        }
    }

    //generate note objects (gameobjects)
    private void GenerateNotes()
    {
        string thisNotesCombo = "";
        int combinationStepper = 0;

        for (int i = 0; i < thisSong.GetLength(0); i++)
        {
            for (int j = 0; j < thisSong.GetLength(1); j++)
            {
                //Starting index 0, second and fourth beats are not hit. Set to null
                if (j == 1 || j == 3)
                    thisSong[i, j] = null;

                else
                {
                    GameObject newNote = Instantiate(note);
                    //CODE FOR SETTING NOTE PROPERTIES (POSITION, COMBINATION, ETC) OMITTED

                    thisSong[i, j] = newNote;
                    thisNotesCombo = "";
                    combinationStepper++;
                }
            }
        }
    }

    public string GetArrowKeys()
    {
        if (Input.GetKeyDown(KeyCode.UpArrow))
            return "U";
        else if (Input.GetKeyDown(KeyCode.LeftArrow))
            return "L";
        else if (Input.GetKeyDown(KeyCode.DownArrow))
            return "D";
        else if (Input.GetKeyDown(KeyCode.RightArrow))
            return "R";

        return "";
    }

    public string GetWASD()
    {
        if (Input.GetKeyDown(KeyCode.W))
            return "U";
        else if (Input.GetKeyDown(KeyCode.A))
            return "L";
        else if (Input.GetKeyDown(KeyCode.S))
            return "D";
        else if (Input.GetKeyDown(KeyCode.D))
            return "R";

        return "";
    }

    private string GetExpectedCombination()
    {
        string expectedCombo = "";

        if (currBeat < 5)   //bounds check
        {
            int expectedNoteBeat = currBeat;
            int expectedNoteMeasure = currMeasure;

            if (currBeat == 1)      //if we're at the second beat in a measure, want to get the third beat
            {
                expectedNoteMeasure = currMeasure;
                expectedNoteBeat = 2;
            }

            else if (currBeat == 3)     //if we're at the fourth beat in a measure, then to get the first beat of the next one
            {
                expectedNoteMeasure = currMeasure + 1;
                expectedNoteBeat = 0;
            }

            //bounds check for end of song
            if (expectedNoteMeasure < 77 && expectedNoteBeat <= 3)
            {
                GameObject posInSong = thisSong[expectedNoteMeasure, expectedNoteBeat];

                if (posInSong != null)
                    expectedCombo = posInSong.gameObject.GetComponent<NewNote>().GetCombination();
            }
        }

        return expectedCombo;
    }

    private bool CombinationCheck(string pressedKeys, string expectedCombo)
    {
        if (pressedKeys.Equals(expectedCombo))
            return true;
        else
            return false;
    }

    public void CallCoroutine(string coroutineToCall)
    {
        if (coroutineToCall.Equals("StartMovement"))
        {
            //out of bounds check: always looking to move the note that is 4 measures ahead.
            if (currMeasure < 73)
                MoveNote(currMeasure + 4, currBeat);
        }
    }

    //tell a note to move from offscreen towards the Fret
    public void MoveNote(int currMeasure, int currBeat)
    {
        if (currBeat == 1 || currBeat == 3)
            currBeat--;

        StartCoroutine(thisSong[currMeasure, currBeat].gameObject.GetComponent<NewNote>().WaitAndMove(0f));
    }

    public bool WindowCheck()
    {
        //hitting the third beat of a measure
        if (SimpleClock.Instance.Beats == 0)
            return true;

        if ((SimpleClock.Instance.Beats == 2 && (SimpleClock.Instance.Ticks >= 48)) || (SimpleClock.Instance.Beats == 3 && (SimpleClock.Instance.Ticks <= 48)))
            return true;

        //hitting the first beat of a measure
        else if ((SimpleClock.Instance.Beats == 4 && (SimpleClock.Instance.Ticks >= 48)) || (SimpleClock.Instance.Beats == 1 && (SimpleClock.Instance.Ticks <= 48)))
        {
            if (SimpleClock.Instance.Measures == 2) //SimpleClock edge case
                return false;

            return true;
        }

        else if (SimpleClock.Instance.Beats == 5)   //SimpleClock edge case
            return true;

        return false;
    }

    private class RhythmGame : FiniteStateMachine<RhythmGameController>.State
    {
        //nested state machine for detecting timing windows: when a player can and cannot hit a note combo.
        //the parent state machine manages rhythm game phases, the nested state machine manages timing windows and is controlled by the parent
        FiniteStateMachine<Phase1> phaseWindowStateMachine;
        private bool started = false;

        private bool phase1 = true;
        private bool phase2 = false;

        private int strikes = 0;
        private int noteCounter;

        public override void OnEnter()
        {
            phase1 = true;
            phase2 = false;
            started = false;
            strikes = 0;

            phaseWindowStateMachine = new FiniteStateMachine<Phase1>(this);
            phaseWindowStateMachine.TransitionTo<Resting>();
        }

        public override void Update()
        {
            phaseWindowStateMachine.Update();

            //transition to phase 2
            if (noteCounter > Context.phase1Threshold && phase1)
            {
                //CODE TO BEGIN SHOWING NOTES IN PHASE 2 OMITTED
                phase1 = false;
                phase2 = true;
            }

            //if in the window and NOT in InWindow state, transition to InWindow
            if (Context.WindowCheck() && (phaseWindowStateMachine.CurrentState.GetType() != typeof(InWindow)) && started)
                phaseWindowStateMachine.TransitionTo<InWindow>();

            else if (!Context.WindowCheck() && (phaseWindowStateMachine.CurrentState.GetType() == typeof(InWindow)) && started)
                phaseWindowStateMachine.TransitionTo<OutOfWindow>();

            if (SimpleClock.Instance.Measures > 78)
            {
                Context.completed = true;           //player beat the rhythm game
                RestartRhythmGame();
                TransitionTo<ClosingAnimation>();
            }
        }

        public void StrikeCheck()
        {
            Context.lifeSprites[strikes].GetComponent<HPShatter>().CallShatterAnim();
        }

        public void RestartRhythmGame()
        {
            phase1 = true;
            phase2 = false;

            noteCounter = 0;
            strikes = 0;

            //reset all notes. notes are not destroyed when they reach the goal, they just turn invisible and teleport somewhere irrelevant
            //CODE FOR STOPPING MUSIC AND RESETTING NOTE SPRITES AND FRET OMITTED

            started = false;

            TransitionTo<RhythmGame>();
        }

        //Nested state machine
        private class Resting : FiniteStateMachine<Phase1>.State
        {
            string pressedCombo;
            string expectedCombo;
            string pressedArrow;
            string pressedWASD;
            bool firstComboPressed;
            float bufferTimer;

            public override void OnEnter()
            {
                pressedCombo = "";
                expectedCombo = Context.Context.thisSongSequence[0];
                pressedArrow = "";
                pressedWASD = "";
                firstComboPressed = false;
                bufferTimer = 1f;   //player does not need to hit both WASD and arrow keys at the exact same frame, but within 1 second of each other
            }
            public override void Update()
            {//
                if (!pressedArrow.Equals("") && bufferTimer >= 0)
                {
                    bufferTimer -= Time.deltaTime;
                    pressedWASD = Context.Context.GetWASD();
                }

                else if (!pressedArrow.Equals("") && bufferTimer >= 0)
                {
                    bufferTimer -= Time.deltaTime;
                    pressedArrow = Context.Context.GetArrowKeys();
                }
                else
                {
                    pressedWASD = Context.Context.GetWASD();
                    pressedArrow = Context.Context.GetArrowKeys();
                }

                pressedCombo = pressedArrow + pressedWASD;

                if (bufferTimer < 0)
                {
                    pressedArrow = "";
                    pressedWASD = "";
                    pressedCombo = "";
                    bufferTimer = 1f;
                }
```

```csharp

            if (expectedCombo.Equals(pressedCombo) && !firstComboPressed)
                StartRhythmGame();

            if (pressedCombo.Length == 2 && !expectedCombo.Equals(pressedCombo))    //rhythm game only starts when player hits correct first combo
            {
                pressedArrow = "";
                pressedWASD = "";
                pressedCombo = "";
            }

            if (firstComboPressed && !Context.Context.WindowCheck())
                TransitionTo<OutOfWindow>();
        }

        private void StartRhythmGame()
        {
            //CODE FOR STARTING MUSIC AND CHANGING THE FRET OMITTED
            Context.noteCounter += 1;
            firstComboPressed = true;
        }

        public override void OnExit()
        {
            Context.started = true;
            firstComboPressed = false;
        }
    }

    private class InWindow : FiniteStateMachine<Phase1>.State
    {
        string pressedCombo = "";
        string expectedCombo = "";
        string pressedArrow;
        string pressedWASD;

        public override void OnEnter()
        {
            pressedCombo = "";
            expectedCombo = Context.Context.GetExpectedCombination();

            pressedArrow = "";
            pressedWASD = "";
        }

        public override void Update()
        {
            if (pressedArrow.Equals("")) //if a pressed key has not yet been registered,
                pressedArrow = Context.Context.GetArrowKeys(); //then check
```

```
                             for a pressed key
                    if (pressedWASD.Equals(""))
                        pressedWASD = Context.Context.GetWASD();
                }

                public override void OnExit()
                {
                    //ALL CODE FOR VISUAL FEEDBACK OMITTED
                    pressedCombo = pressedArrow + pressedWASD;

                    Context.noteCounter += 1;

                    //phase 2 check: if an incorrect combination was pressed, grant a strike
                    if (!Context.Context.CombinationCheck(pressedCombo, expectedCombo) && Context.phase2)
                    {
                        Context.strikes++;
                        if (Context.strikes > 5)
                            Context.RestartRhythmGame();
                    }
                }
            }

            //empty state just to denote being out of the timing window
            private class OutOfWindow : FiniteStateMachine<Phase1>.State
            {
                void Update()
                { }
            }
        }
    }
```