Summary

For my senior capstone project--a narrative game with rhythm mechanics made in Unity--I was tasked with re-engineering the rhythm game since it was buggy and difficult to play. 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.

Pages 3-11 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
- Visual feedback code (resetting objects, changing background colors)
- Calls to packages used

Technical Design

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

When the Player can or cannot hit a combination is managed with a state machine that has 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.

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.

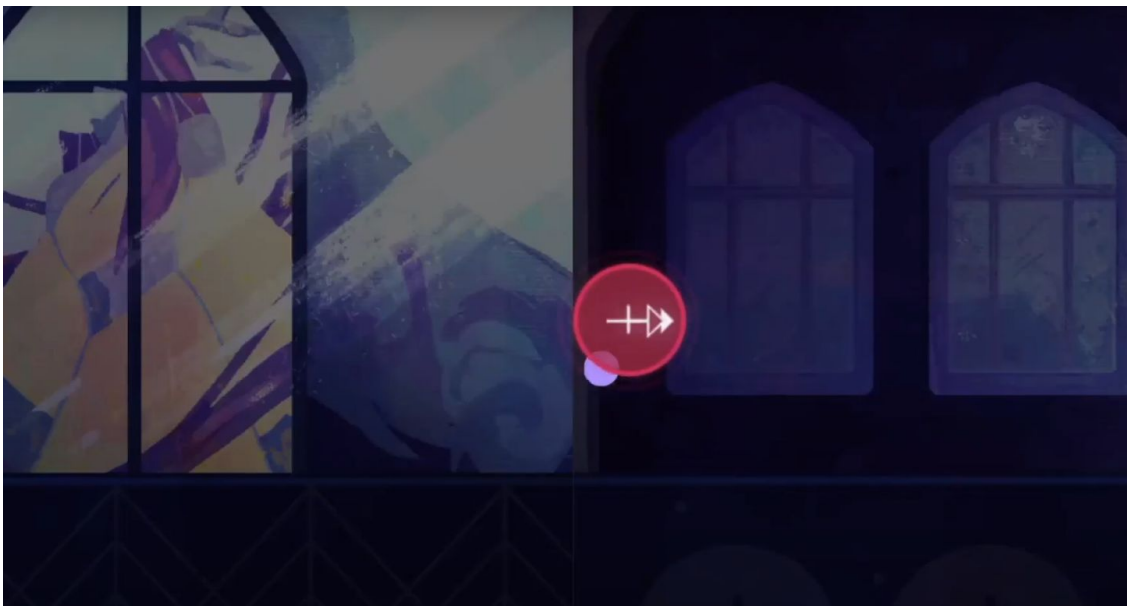

Fig. 1. Phase 1 of the Rhythm Game. The Fret shows the combination for the player to press.

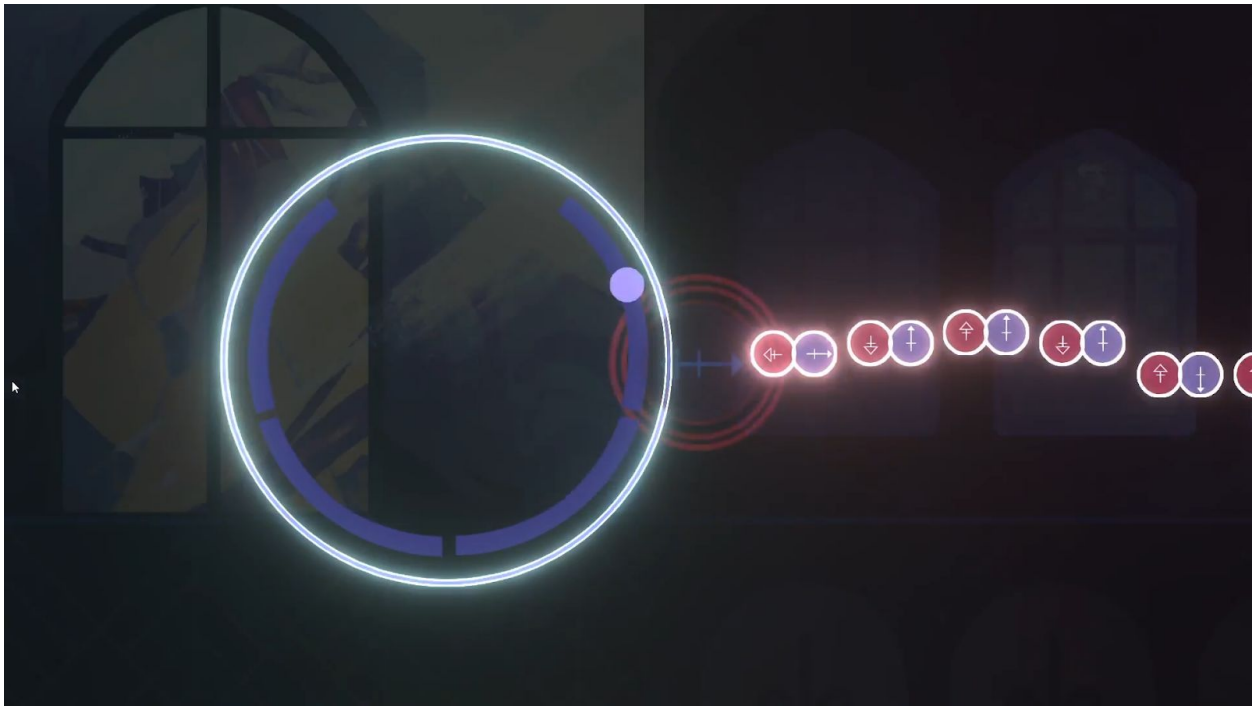

Fig. 2. Phase 2 of the Rhythm Game. The Fret shows the combination for the player to press as well as notes moving in from off screen.

```
1  public class RhythmGameController : MonoBehaviour
2  {
3      FiniteStateMachine<RhythmGameController> rhythmGameStateMachine;
4
5      List<string> notesCombinations = new List<string>() { "UU", "DD", "LL", "RR",
           "UD", "LR", "UU", "DD", "LL", "RR", "UD", "LR" }; //first 10 notes are
         scripted
6
7      string[] mostLikelyCombos = { "UU", "DD", "LL", "RR" }; //these have a 40%
         chance of appearing
8      string[] secondLikelyCombos = { "UL", "UR", "DL", "DR", "LU", "LD", "RU",
         "RD" }; //these have a 40% chance of appearing
9      string[] leastLikelyCombos = { "UD", "DU", "LR", "RL" }; //these have a 20%
         chance of appearing
10
11     //outer array represents measures, inner array represents beats per measure.
12     GameObject[,] thisSong = new GameObject[77, 4];
13     string[] thisSongSequence;
14
15     public int phase1Threshold;
16
17     public int currMeasure;
18     public int currBeat;
19
20     //VARIABLES FOR VISUAL FEEDBACK (LIVES, BACKGROUND IMAGE, FRET VISUAL
         FEEDBACK, ORBITER) OMITTED
21     private bool completed = false;
22
23     void Start()
24     {
25         rhythmGameStateMachine = new FiniteStateMachine<RhythmGameController>
             (this);
26         rhythmGameStateMachine.TransitionTo<IntroAnimation>();
27
28         //generate the random combination for the second phase of the song and
             make the song into one string
29         GenerateCombinations();
30
31         this.thisSongSequence = notesCombinations.ToArray();
32
33         GenerateNotes();
34     }
35
36     void Update()
37     {
38         //CODE CALCULATING CLOCK OFFSET OMITTED
39         currMeasure = SimpleClock.Instance.Measures;
40         currBeat = SimpleClock.Instance.Beats;
41
42         rhythmGameStateMachine.Update();
43     }
44
```

```
45      //generate the list of combinations (strings)
46      private void GenerateCombinations()
47      {
48          int combosToGenerate = 154 - notesCombinations.Count;
49          string thisNotesCombo = "";
50
51          for (int i = 0; i < combosToGenerate; i++)
52          {
53              //set a bias, certain combinations are more likely than others
54              int comboBias = Random.Range(0, 5);
55              int getComboIndex = 0;
56
57              if (comboBias == 0 || comboBias == 1)
58              {
59                  getComboIndex = Random.Range(0, mostLikelyCombos.Length);
60                  thisNotesCombo = mostLikelyCombos[getComboIndex];
61              }
62              else if (comboBias == 2)
63              {
64                  getComboIndex = Random.Range(0, secondLikelyCombos.Length);
65                  thisNotesCombo = secondLikelyCombos[getComboIndex];
66              }
67              else
68              {
69                  getComboIndex = Random.Range(0, leastLikelyCombos.Length);
70                  thisNotesCombo = leastLikelyCombos[getComboIndex];
71              }
72
73              notesCombinations.Add(thisNotesCombo);
74              thisNotesCombo = "";
75          }
76      }
77
78      //generate note objects (gameobjects)
79      private void GenerateNotes()
80      {
81          string thisNotesCombo = "";
82          int combinationStepper = 0;
83
84          for (int i = 0; i < thisSong.GetLength(0); i++)
85          {
86              for (int j = 0; j < thisSong.GetLength(1); j++)
87              {
88                  //Starting index 0, second and fourth beats are not hit. Set to
                        null
89                  if (j == 1 || j == 3)
90                      thisSong[i, j] = null;
91
92                  else
93                  {
94                      GameObject newNote = Instantiate(note);
95                      //CODE FOR SETTING NOTE PROPERTIES (POSITION, COMBINATION,
```

```
                    ETC) OMITTED
 96
 97                     thisSong[i, j] = newNote;
 98                     thisNotesCombo = "";
 99                     combinationStepper++;
100                 }
101             }
102         }
103     }
104
105     public string GetArrowKeys()
106     {
107         if (Input.GetKeyDown(KeyCode.UpArrow))
108             return "U";
109         else if (Input.GetKeyDown(KeyCode.LeftArrow))
110             return "L";
111         else if (Input.GetKeyDown(KeyCode.DownArrow))
112             return "D";
113         else if (Input.GetKeyDown(KeyCode.RightArrow))
114             return "R";
115
116         return "";
117     }
118
119     public string GetWASD()
120     {
121         if (Input.GetKeyDown(KeyCode.W))
122             return "U";
123         else if (Input.GetKeyDown(KeyCode.A))
124             return "L";
125         else if (Input.GetKeyDown(KeyCode.S))
126             return "D";
127         else if (Input.GetKeyDown(KeyCode.D))
128             return "R";
129
130         return "";
131     }
132
133     private string GetExpectedCombination()
134     {
135         string expectedCombo = "";
136
137         if (currBeat < 5)   //bounds check
138         {
139             int expectedNoteBeat = currBeat;
140             int expectedNoteMeasure = currMeasure;
141
142             if (currBeat == 1)      //if we're at the second beat in a measure,
                  want to get the third beat
143             {
144                 expectedNoteMeasure = currMeasure;
145                 expectedNoteBeat = 2;
```

```
146             }
147
148             else if (currBeat == 3)     //if we're at the fourth beat in a
                  measure, then to get the first beat of the next one
149             {
150                 expectedNoteMeasure = currMeasure + 1;
151                 expectedNoteBeat = 0;
152             }
153
154             //bounds check for end of song
155             if (expectedNoteMeasure < 77 && expectedNoteBeat <= 3)
156             {
157                 GameObject posInSong = thisSong[expectedNoteMeasure,
                      expectedNoteBeat];
158
159                 if (posInSong != null)
160                     expectedCombo = posInSong.gameObject.GetComponent<NewNote>
                          ().GetCombination();
161             }
162         }
163
164         return expectedCombo;
165     }
166
167     private bool CombinationCheck(string pressedKeys, string expectedCombo)
168     {
169         if (pressedKeys.Equals(expectedCombo))
170             return true;
171         else
172             return false;
173     }
174
175     public void CallCoroutine(string coroutineToCall)
176     {
177         if (coroutineToCall.Equals("StartMovement"))
178         {
179             //out of bounds check: always looking to move the note that is 4
                  measures ahead.
180             if (currMeasure < 73)
181                 MoveNote(currMeasure + 4, currBeat);
182         }
183     }
184
185     //tell a note to move from offscreen towards the Fret
186     public void MoveNote(int currMeasure, int currBeat)
187     {
188         if (currBeat == 1 || currBeat == 3)
189             currBeat--;
190
191         StartCoroutine(thisSong[currMeasure,
              currBeat].gameObject.GetComponent<NewNote>().WaitAndMove(0f));
192     }
```

```
193
194     public bool WindowCheck()
195     {
196         //hitting the third beat of a measure
197         if (SimpleClock.Instance.Beats == 0)
198             return true;
199
200         if ((SimpleClock.Instance.Beats == 2 && (SimpleClock.Instance.Ticks >=
              48)) || (SimpleClock.Instance.Beats == 3 && (SimpleClock.Instance.Ticks
               <= 48)))
201             return true;
202
203         //hitting the first beat of a measure
204         else if ((SimpleClock.Instance.Beats == 4 && (SimpleClock.Instance.Ticks
              >= 48)) || (SimpleClock.Instance.Beats == 1 &&
              (SimpleClock.Instance.Ticks <= 48)))
205         {
206             if (SimpleClock.Instance.Measures == 2) //SimpleClock edge case
207                 return false;
208
209             return true;
210         }
211
212         else if (SimpleClock.Instance.Beats == 5)   //SimpleClock edge case
213             return true;
214
215         return false;
216     }
217
218     private class RhythmGame : FiniteStateMachine<RhythmGameController>.State
219     {
220         //nested state machine for detecting timing windows: when a player can
              and cannot hit a note combo.
221         //the parent state machine manages rhythm game phases, the nested state
              machine manages timing windows and is controlled by the parent
222         FiniteStateMachine<Phase1> phaseWindowStateMachine;
223         private bool started = false;
224
225         private bool phase1 = true;
226         private bool phase2 = false;
227
228         private int strikes = 0;
229         private int noteCounter;
230
231         public override void OnEnter()
232         {
233             phase1 = true;
234             phase2 = false;
235             started = false;
236             strikes = 0;
237
238             phaseWindowStateMachine = new FiniteStateMachine<Phase1>(this);
```

```
239             phaseWindowStateMachine.TransitionTo<Resting>();
240         }
241
242         public override void Update()
243         {
244             phaseWindowStateMachine.Update();
245
246             //transition to phase 2
247             if (noteCounter > Context.phase1Threshold && phase1)
248             {
249                 //CODE TO BEGIN SHOWING NOTES IN PHASE 2 OMITTED
250                 phase1 = false;
251                 phase2 = true;
252             }
253
254             //if in the window and NOT in InWindow state, transition to InWindow
255             if (Context.WindowCheck() &&
                  (phaseWindowStateMachine.CurrentState.GetType() != typeof
                  (InWindow)) && started)
256                 phaseWindowStateMachine.TransitionTo<InWindow>();
257
258             else if (!Context.WindowCheck() &&
                  (phaseWindowStateMachine.CurrentState.GetType() == typeof
                  (InWindow)) && started)
259                 phaseWindowStateMachine.TransitionTo<OutOfWindow>();
260
261             if (SimpleClock.Instance.Measures > 78)
262             {
263                 Context.completed = true;           //player beat the rhythm game
264                 RestartRhythmGame();
265                 TransitionTo<ClosingAnimation>();
266             }
267         }
268
269         public void StrikeCheck()
270         {
271             Context.lifeSprites[strikes].GetComponent<HPShatter>
                  ().CallShatterAnim();
272         }
273
274         public void RestartRhythmGame()
275         {
276             phase1 = true;
277             phase2 = false;
278
279             noteCounter = 0;
280             strikes = 0;
281
282             //reset all notes. notes are not destroyed when they reach the goal,
                  they just turn invisible and teleport somewhere irrelevant
283             //CODE FOR STOPPING MUSIC AND RESETTING NOTE SPRITES AND FRET OMITTED
284
```

```
285             started = false;
286 
287             TransitionTo<RhythmGame>();
288         }
289 
290         //Nested state machine
291         private class Resting : FiniteStateMachine<Phase1>.State
292         {
293             string pressedCombo;
294             string expectedCombo;
295             string pressedArrow;
296             string pressedWASD;
297             bool firstComboPressed;
298             float bufferTimer;
299 
300             public override void OnEnter()
301             {
302                 pressedCombo = "";
303                 expectedCombo = Context.Context.thisSongSequence[0];
304                 pressedArrow = "";
305                 pressedWASD = "";
306                 firstComboPressed = false;
307                 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
308             }
309             public override void Update()
310             {//
311                 if (!pressedArrow.Equals("") && bufferTimer >= 0)
312                 {
313                     bufferTimer -= Time.deltaTime;
314                     pressedWASD = Context.Context.GetWASD();
315                 }
316 
317                 else if (!pressedArrow.Equals("") && bufferTimer >= 0)
318                 {
319                     bufferTimer -= Time.deltaTime;
320                     pressedArrow = Context.Context.GetArrowKeys();
321                 }
322                 else
323                 {
324                     pressedWASD = Context.Context.GetWASD();
325                     pressedArrow = Context.Context.GetArrowKeys();
326                 }
327 
328                 pressedCombo = pressedArrow + pressedWASD;
329 
330                 if (bufferTimer < 0)
331                 {
332                     pressedArrow = "";
333                     pressedWASD = "";
334                     pressedCombo = "";
```

```
335                     bufferTimer = 1f;
336                 }
337
338                 if (expectedCombo.Equals(pressedCombo) && !firstComboPressed)
339                     StartRhythmGame();
340
341                 if (pressedCombo.Length == 2 && !expectedCombo.Equals
                      (pressedCombo))    //rhythm game only starts when player hits
                      correct first combo
342                 {
343                     pressedArrow = "";
344                     pressedWASD = "";
345                     pressedCombo = "";
346                 }
347
348                 if (firstComboPressed && !Context.Context.WindowCheck())
349                     TransitionTo<OutOfWindow>();
350             }
351
352             private void StartRhythmGame()
353             {
354                 //CODE FOR STARTING MUSIC AND CHANGING THE FRET OMITTED
355                 Context.noteCounter += 1;
356                 firstComboPressed = true;
357             }
358
359             public override void OnExit()
360             {
361                 Context.started = true;
362                 firstComboPressed = false;
363             }
364         }
365
366         private class InWindow : FiniteStateMachine<Phase1>.State
367         {
368             string pressedCombo = "";
369             string expectedCombo = "";
370             string pressedArrow;
371             string pressedWASD;
372
373             public override void OnEnter()
374             {
375                 pressedCombo = "";
376                 expectedCombo = Context.Context.GetExpectedCombination();
377
378                 pressedArrow = "";
379                 pressedWASD = "";
380             }
381
382             public override void Update()
383             {
384                 if (pressedArrow.Equals("")) //if a pressed key has not yet been
```

```
                    registered,
385                     pressedArrow = Context.Context.GetArrowKeys(); //then check
                         for a pressed key
386                 if (pressedWASD.Equals(""))
387                     pressedWASD = Context.Context.GetWASD();
388             }
389
390             public override void OnExit()
391             {
392                 //ALL CODE FOR VISUAL FEEDBACK OMITTED
393                 pressedCombo = pressedArrow + pressedWASD;
394
395                 Context.noteCounter += 1;
396
397                 //phase 2 check: if an incorrect combination was pressed, grant a
                     strike
398                 if (!Context.Context.CombinationCheck(pressedCombo,
                    expectedCombo) && Context.phase2)
399                 {
400                     Context.strikes++;
401                     if (Context.strikes > 5)
402                         Context.RestartRhythmGame();
403                 }
404             }
405         }
406
407         //empty state just to denote being out of the timing window
408         private class OutOfWindow : FiniteStateMachine<Phase1>.State
409         {
410             void Update()
411             { }
412         }
413     }
414 }
```