HO2

FYP Final Report

**Music Rhythm Game**

[**https://johnnyn2.github.io/MusicRhythmGameBuild/index.html**](https://johnnyn2.github.io/MusicRhythmGameBuild/index.html)

by

Victor Fung, Sam Ho, Sarah Ho

**HO2**

Advised by

Prof. Andrew B. HORNER

Submitted in partial fulfillment

of the requirements for COMP 4982

in the

Department of Computer Science

The Hong Kong University of Science and Technology

2019-2020

Date of submission: February 28, 2020

**Tables Of Contents**

[**Abstract**](#_ugrqgxa3v65w) **5**

[**1. Introduction**](#_8kbzxojn1m5n) **6**

[1.1 Overview](#_13tq1mawdzhc) 6

[1.2 Objectives](#_wt52q9nzqga1) 7

[1.3 Literature Survey](#_7ka60hqcjcir) 8

[1.3.1 Cytus](#_s4d7sudn3jcf) 8

[1.3.2 Thumper](#_mm0yfhcul18t) 9

[1.3.3 Beat Saber](#_5zpk7e9dkfsm) 10

[**2. Methodology**](#_97pzwqk28kyf) **11**

[2.1 Design](#_2m6lp74po318) 11

[2.1.1 Storyline](#_x758jasbpc40) 11

[2.1.2 Gameplay](#_ne2gzrvd0wa1) 11

[2.1.3 Navigation](#_nzfbo9lhfaaq) 12

[2.1.4 Scenes, Character, Equipments, Final Bosses and Minions](#_pk2j9rllp08i) 13

[2.1.5 Physics](#_lqunxyloiv9o) 13

[2.1.6 Animation and Special Effects](#_a71lssfd7vad) 13

[2.1.7 Audio and Sound Effects](#_u0upp3x5werm) 14

[2.1.8 Collaboration and Source Control](#_hc098y4z17nz) 15

[2.2 Implementation](#_agfd7dip9cit) 15

[Game Scene 1: Main Menu](#_drutqvd5a6j9) 15

[Game Scene 2: Song Menu](#_v2ng9mlhqhvh) 15

[Game Scene 3: Game](#_mcwh2kpngn0p) 16

[Game Scene 4: Shop](#_sebg9xkkm0pr) 16

[1. User Interface](#_cqe9zt7hawve) 17

[2. Game Objects](#_w59jrnikvi28) 18

[Game Scene 1: Main Menu](#_kd3rhze070xx) 18

[Game Scene 2: Song Menu](#_3gj4nhh13o4z) 18

[Game Scene 3: Game](#_bo8oy9n6wj2k) 18

[Game Scene 4: Shop](#_o39c76rxmnb) 19

[2.1 Camera View (CameraMotor.cs)](#_u0r1pz3mj5cn) 19

[2.2 Character Control (PlayerMotor.cs)](#_dpirpdvpq2le) 20

[2.3 Runway Control (TileManager.cs)](#_bcbgod6vw2le) 20

[2.4 Sound Control (SoundManager.cs and Song.cs)](#_cpxuwxpgk3fp) 20

[2.5 Audio Visualization (Beat Tracking and Onset Detection)](#_vyih2o2g0574) 21

[2.5.1 Beat Tracking (SoundManager.cs)](#_fn27wdkz8mhj) 22

[2.5.2 Onset Detection (SpectralFluxAnalyzer.cs and DSPLib.cs)](#_xdmy22x3df6i) 25

[2.6 Minion Control (MinionManager.cs)](#_rvxoic2g2v27) 36

[2.6.1 Individual Minion (Minion.cs)](#_ah0jhsuvol2n) 37

[2.7 Game Management (HealthBar.cs, Score.cs, DeathMenu.cs, PauseMenu.cs)](#_xij5b2tccoq) 37

[2.7.1 Health Bar (HealthBar.cs)](#_wo1ek93shy2z) 37

[2.7.2 Score (Score.cs)](#_7buqy0v7f76g) 37

[2.7.3 Death Menu (DeathMenu.cs)](#_l7xgdte2m5ad) 37

[2.7.4 PauseMenu (PauseMenu.cs)](#_u7efzca8xj97) 38

[3. Animators](#_qecnu311j57b) 38

[2.3 Testing](#_7br2qdnsuqzt) 40

[2.3.1 Physical Objects](#_97ivdnpgslab) 40

[2.3.3 User Interface](#_newsv2rkjs9v) 40

[**3. Discussion**](#_qp479kbnjaq7) **42**

[**4. Conclusion**](#_x938lpqc4t9g) **42**

[**5. References**](#_bverslmhips3) **44**

[**6. Appendix A: Meeting Minutes**](#_9d1794ngpkul) **45**

[6.1 Minutes of the 1st Project Meeting](#_o4bijm9454np) 45

[**6.2 Project Planning**](#_cwsckphx8ayb) **46**

[6.2.1 Distribution of Work](#_5zgh75pnqo4k) 46

[**6.3. Required Hardware & Software**](#_xq586pwvq6v5) **50**

[6.3.1 Hardware](#_x3r6qix14hry) 50

[6.3.2 Software​​​​​​​​​​​​](#_kvv8b46mrhid) 50

[6.2 Minutes of the 2nd Project Meeting](#_vu1d5gwynpat) 51

# **Abstract**

The music rhythm game we developed is a game that integrate music component with RPG element. As a warrior, you need to kill the monsters that want to invade the village and the monsters will attack the warrior in synchronous with the beat or the onset detected with our algorithm.

The music of the game is processed through a on-set detection algorithm and beat detection to produce 2 types of gameplay. We pre-process the music and store in separate json files and loaded the file when the player choose to play that song.

As the Unity do not provide a pre-process audio analysis, so we developed a 5 step approach to do the onset detection. This reduce the loading time of each song as the real-time audio analysis API provided by Unity require long processing time before the level is loaded and which is not user friendly.

# **1.** **Introduction**

## **1.1** **Overview**

Music rhythm game is a game genre that the music of a game plays a crucial role. One typical type in this genre is one that requires users to press buttons at a certain time, which in a sense simulates musical instruments playing or dance steps. Dance Dance Revolution is one of the classical and widespread example of arcade machine dance-rhythm game and Cytus is another popular game on mobile platforms. Various versions of rhythm games are available across different platforms PC and mobile games game consoles.

Most of the music rhythm games available on the market focus merely on the music and thus lack variations, making the games repetitive and it becomes hard to retain the interest of players. Meanwhile, a limited number of rhythm games have included different elements like fighting, car-racing and puzzle. Adding non-musical features indeed makes the game more exciting and attractive. However, some of them are too challenging for average players as the additional content complicates the game, making the game less enjoyable.

This project aims at developing an accessible and engaging PC music rhythm game and combining it with role-playing game (RPG) element. The game’s plot sets in doomsday when giant monsters invade Earth. It is combined to the music rhythm aspect by representing the music beats with minions sent by the enemies and obstacles, while the player is playing a warrior character defeating the minions and avoiding the obstacles according to the rhythm. Our team’s goal is to create an enjoyable and unique version of music rhythm game.

## **1.2** **Objectives**

Our project will mainly focus on the following objectives:

1. Combine RPG elements with music rhythm game

2. Develop a user-friendly user interface for players

3. Integrate a ranking system into the game

4. Convert MIDI files into usable format for rhythm processing

To achieve the objectives, we will:

1. Develop 3D model and write game logic with Unity

2. Research audio processing tools to analyzing songs to useful data

3. Devote our time and effort to build an enjoyable and relaxing rhythm game

4. Consult people from friends to professor to examine and comment the game

The biggest challenge we expect to face will be to explore the area of audio processing and working with music, which our team is not particularly familiar with. To addresses this challenge, we will actively research and consult more on different aspect of audio processing. For example, study which music elements are crucial to our rhythm game. Using online material and documentation of some audio processing tools to understand which aspects of music we can process and what data we will use to build our game. Also, we will search some tools to help us analysis the different music element of some music pieces. After doing some experiment on the audio processing tool, we can conclude which information of the music we will use and become more familiarize with audio processing.

## 

## **1.3** **Literature Survey**

We did an online survey and found some games related to our project.

### **1.3.1** **Cytus**

  
(Figure 1: Cytus: rhythm game)

Cytus is the first music rhythm game developed by Rayark in 2012 and it result in over 5 million downloads[2] in 2019. In 2013, “Cytus” was one of the Best Game of Year of Google Play.

The mechanism of the game is straightforward and simple. Players tap on the pop-up notes when the “Active Scan Line” contact the notes. The score of the game is determined by the timing of pressing the note when the “Active Scan Line” passes by and the score is evaluated in 5 levels. Each level has a different score and decrease from level “colored perfect” to “miss”. Combo is a system to reward player with higher score that continuous well performance in the song. Combo reset if the note results in “bad” or “miss”.

This game poses a challenge to the players to achieve highest score and lots of video post online for showing how to obtain the highest. However, this game maybe too intense for some players, the difficulties in some level and need to put a lot of effort to obtain high score. Our game integrates the style of RPG and music rhythm, therefore the stress to achieve perfect in replaying the level can be lower.

### **1.3.2** **Thumper**

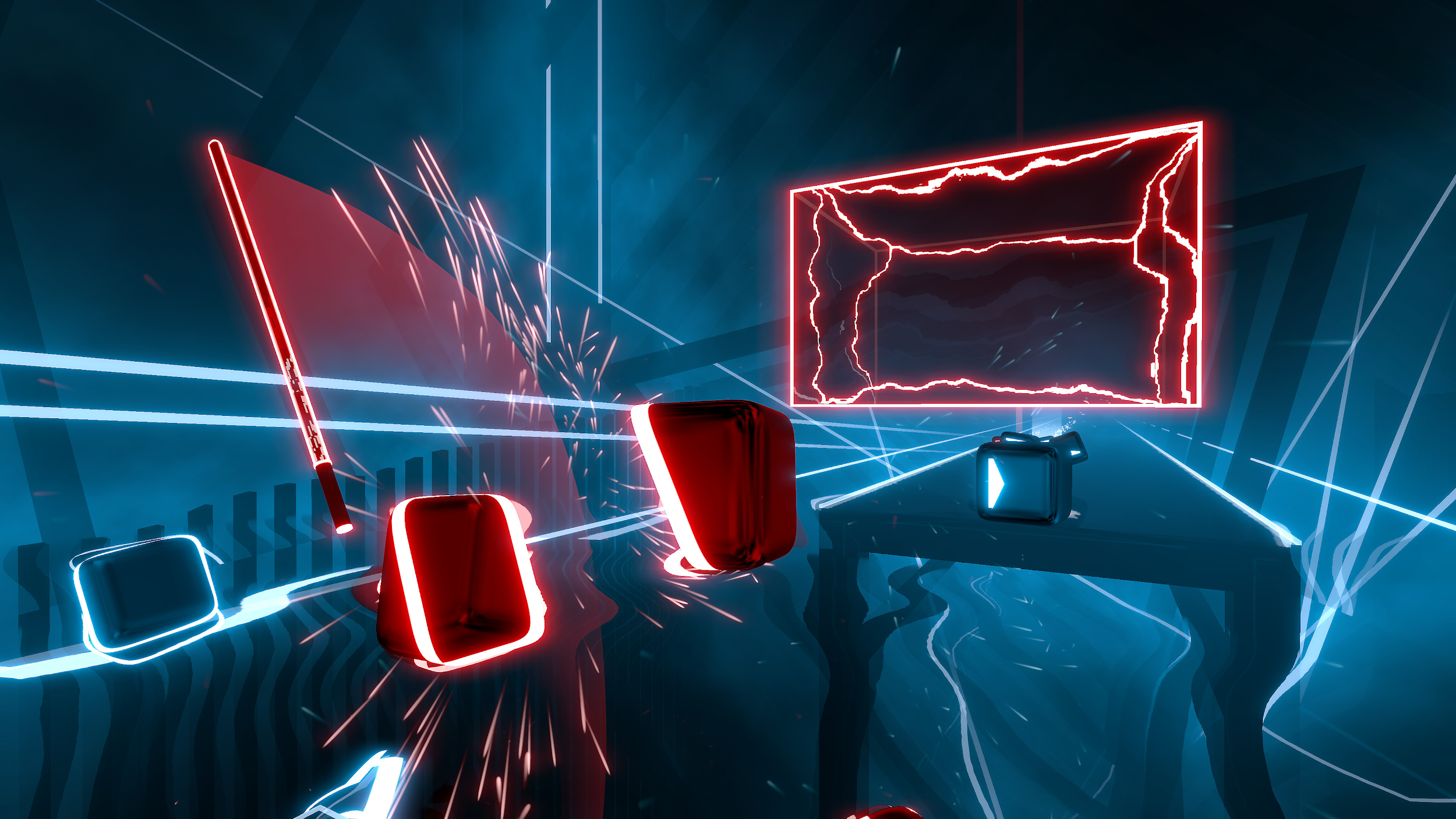
  
(Figure 2: Thumper: Rhythm violence game)

Thumper is an rhythm-action game that provide forceful gaming experience. This game has won lots of awards like the 2016 Paste Magazine Game of the Year [4]. The game has been published in multiple platforms like PC, Android, and iOS. Also, some platforms support VR and allow players to play the game with virtual reality (VR) device like oculus rift.

The gameplay mechanism is similar to endless runner game but require players to react more quickly than other endless runner game in the market such as Temple Run. The music rhythm help players to act according to the course. If the player’s miss an action, player will lose a life and lose the game eventually after few misses of actions.

This game include action element in a rhythm game and let the players have a incentive other than replaying the songs perfectly. However, the game is quite hard for beginner and not suitable for players who want to play game casually. Our game also involves battle element, but with a more strategic approach to deal with the enemies rather than test for players’ reaction time.

### **1.3.3** **Beat Saber**

   
(Figure 3: VR rhythm game)

Beat Saber is a VR rhythm game developed and published by Beat Games and fully released in May 2019. This game is the winner of the 2018 NAVGTR Awards in the Game, Music or Performance-Based category [3].

Player need to use the controller as a saber and slash the box in the dedicated direction. Sometimes, the player need to move in real life to avoid the obstacles. The uniqueness of this game is the control of this game. Players swing and slash the saber to destroy the boxes. Unlike other rhythm games, they are highly relied on the use of buttons. The boxes will have 2 colors which according to the color of the sabers you choose and cut the colored boxes with appropriate color of sabers.

The minimalist design allow players focus on the gameplay which let them familiarize with the interface and game mechanics. However, as the level go harder, this game require good stamina to play as swinging and slashing repeatedly require lots of energy. Our game will design and learn the minimalist style of interface and scene from this game and build a game with similar art style.

# **2.** **Methodology**

## **2.1** **Design**

The design phase of the project started in early July. We decided to make a music rhythm game integrated with 3D RPG elements. Our game takes inspiration from a classic type of music rhythm game which requires the player to catch objects according to the beats of a song. RPG elements like narratives, experience points, character levels, equipment, monsters and minions will be incorporated in our game to enhance the player’s experience.

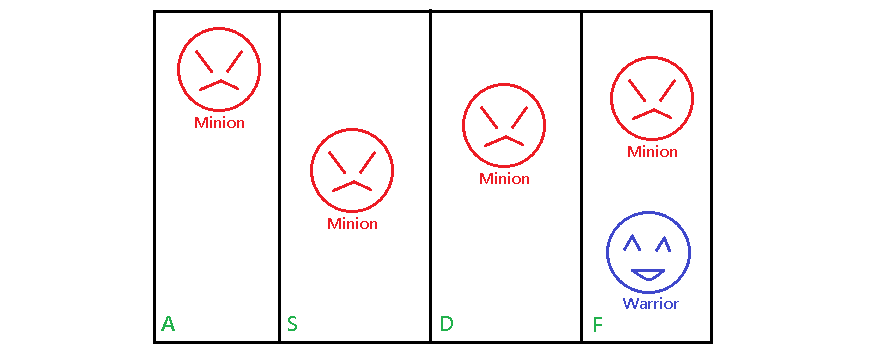
### **2.1.1** **Storyline**

The world is approaching doomsday because of the invasion by giant monsters from the outer space. The player would be playing a warrior role hunting down minions sent by the enemies, hoping to defeat the monsters and save the day.

### **2.1.2** **Gameplay**

The game consists of multiple levels with different songs being played in each level. In each level, a big monster will generate minions according to the music beats and moving towards the warrior along four runways. The player needs to move the warrior onto the correct runway by pressing “a”, “s”, “d” or “f” key to collide and hit the minions.

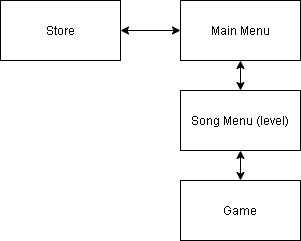
If the player has missed any minion, the warrior would lose life points, calculated based on the attack and defense ability of the warrior and the minions. If the warrior survives when the song ends, the level ends and the player wins. Otherwise, the player loses. After each level, the warrior would gain money and score for that level. The game has a store system and the player is allowed to purchase equipment to enhance the warrior’s abilities.



(Fig. 2.1.2 Game field)

### **2.1.3** **Navigation**

Upon entering the game, the first page would be Main Menu. The player can click the “Play” button to enter the Song Menu. He or she can then select a song menu and then enjoy the game. The player can also click on the “Store” button to purchase equipment.



(Fig 2.1.3 Navigation)

### **2.1.4** **Scenes, Character, Equipments, Final Bosses and Minions**

We have planned to look into the available assets from Unity store and create models for the game objects, such as the warrior, equipments, monsters, minions, lightning, camera, texture, colors, shapes and space.

### **2.1.5** **Physics**

We have planned to apply Unity’s physics engines which provide the components that handles the physical simulation for the warrior and minions. As the warrior is chasing the boss, he should be running under normal gravity and friction force. The minions should also come from the boss and walking toward the warrior under gravity and friction. We have planned to use Unity’s mesh collider to define the shape of the warrior and the minions. The physical object’s scripting system will then be used to detect the collisions when one collider enters the space of another.

### **2.1.6** **Animation and Special Effects**

We have planned to look into the available animation assets from Unity store for the warrior and the minions. The detailed animator specification is still on discussion but we already have a draft on what events the animation would be involved. The following events or objects have been designed with animation or special effects.

|  |
| --- |
| Click on the Store icon |
| 2 seconds to get ready for the game start (the warrior and the boss are idling) |
| Warrior running under gravity and friction automatically |
| Warrior changes the current runway (“a”, “s”, “d” or “f” key is pressed) |
| Minions running under gravity and friction automatically |
| Minions are eliminated by the warrior |
| Warrior fails to eliminate the minions (warrior loses life points) |
| Game ends (player wins) |
| Game ends (player loses) |

### 

### **2.1.7** **Audio and Sound Effects**

We will look into the available music and sound effect assets from Unity store. When the game starts, the music would be playing automatically after 2 seconds (animation duration). Minions would be generated according to the beats or onsets of the music. The following events or objects have been designed with sound effects.

|  |
| --- |
| Click on the Store icon |
| Click on the “Purchase” button |
| Click on the “Close” button |
| 2 seconds to get ready for the game start (the warrior and the boss are idling) |
| Minions are eliminated by the warrior |
| Warrior fails to eliminate the minions (warrior loses life points) |
| Game ends (player wins) |
| Game ends (player loses) |

### **2.1.8** **Collaboration and Source Control**

We have decided to use GitHub for the project repository and Git for collaboration and source control.

## **2.2** **Implementation**

We will implement the game according to the Unity’s documentation for 3D game development. Meanwhile, we may download or buy some assets from Unity store. After we obtain the assets, we will customize their attributes to fit them into our game. Below is our implementation for the game scenes of the game.

### **Game Scene 1: Main Menu**

### 

### **Game Scene 2: Song Menu**



### **Game Scene 3: Game**



### **Game Scene 4: Shop**



### 

### **1.** **User Interface**

Based on our design, we use Unity UI to develop the user interface. We place the UI elements such as the store icon and the menu of levels inCanvas and set the layout for them through Unity’s inspector. We used TextMeshPro package and create color gradient assets to make fancy text in the game. We also scraped some free images from internet for the button icons and cursor icon.



Fig 1.1 The UI of main menu

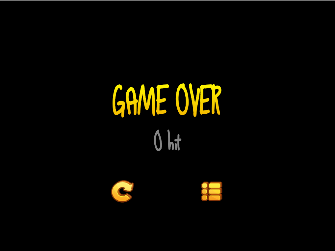


Fig 1.1 The UI when game overs

### **2.** **Game Objects**

For each game object, we will develop a corresponding script to control its behavior.

#### **Game Scene 1: Main Menu**

|  |  |  |  |
| --- | --- | --- | --- |
| **Main Menu (menu.unity)** | | | |
| MainMenu  MainMenu.cs | PlayButton  MainMenu.ToGame() | StoreButton  MainMenu.ToStore() | QuitButton  MainMenu.Quit() |

#### **Game Scene 2: Song Menu**

|  |  |
| --- | --- |
| **Song Menu (songmenu.unity)** | |
| SongMenu  SongMenu.cs | PlayButton  SongMenu.ToGame(String song) |
|

#### **Game Scene 3: Game**

|  |  |  |
| --- | --- | --- |
| **Game (game.unity)** | | |
| Warrior  PlayMotor.cs | | StatusContainer  StatusContainer.cs |
| Camera  CameraMotor.cs | | ScoreContainer  Score.cs |
| TileManager  TileManager.cs | | DeathMenu  DeathMenu.cs |
| MinionManager  MinionManager.cs | | PlayButton  DeathMenu.Restart() |
| SoundManager  SoundManager.cs | | MenuButton  DeathMenu.ToMenu() |
| HealthBar  HealthBar.cs | | |

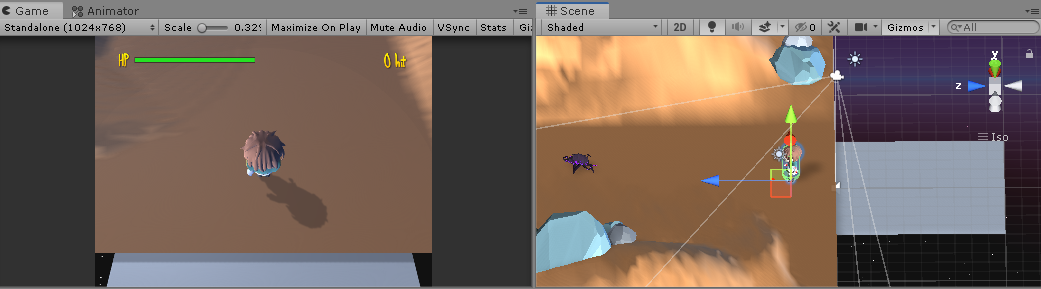
#### **Game Scene 4: Shop**

|  |  |  |
| --- | --- | --- |
| **Shop (Shop.unity)** | | |
| Canvas  Shop.cs | | Confirm  Shop.confirm() |
| ReturnBtn  Shop.returnBtn() | | Cancel  Shop.cancel() |

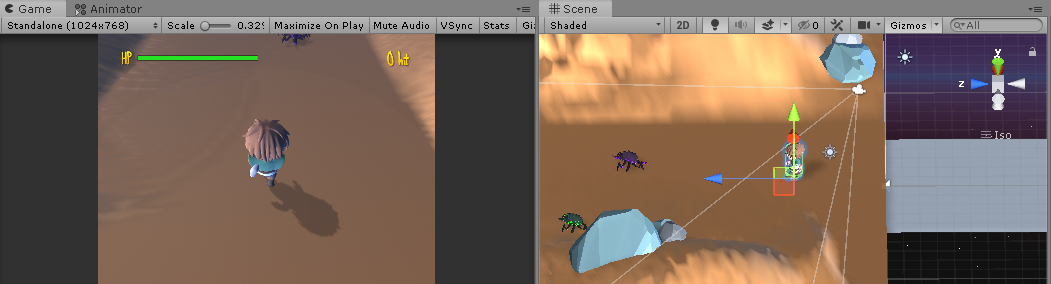
### 

#### **2.1 Camera View (CameraMotor.cs)**

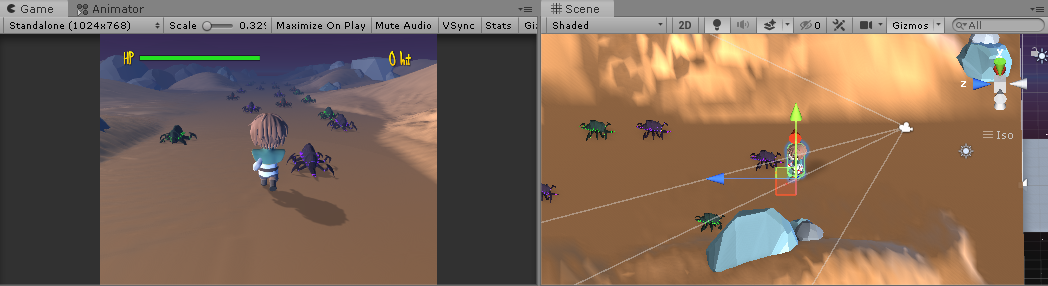
We wrote CameraMotor.cs to control the camera view. In CameraMotor.cs, we initially set the Transform of the camera object to “look at” the warrior. Then, we calculated the move vector (it determines the x,y and z index) for the camera. Finally, we updated the position of the Transform by move vector and animation duration.



(Figure 2.1.1 At the beginning)



(Figure 2.2.2 After 1 second)



(Figure 2.1.3 After 2 seconds)

#### **2.2 Character Control (PlayerMotor.cs)**

This script is used to control the warrior movement and action. We calculated the move vector for the warrior to update its x,y and z position. The x-index of the move vector would not be updated during the camera animation so the warrior would not move when the player pressed “a”, “s”, “d” or “f” key. After the animation, the player can press “a”, “s”, “d” or “f” to change the runway. If the player presses “a”, the warrior moves to the runway A, so on so forth. If the player collides a minion on runway A, the player needs to press “a” key again to attack the minion. Then the minion dying animation displays and the score increases. Otherwise, minion attacking animation displays.

#### **2.3 Runway Control (TileManager.cs)**

We created a manager to generate the segments for the track. It records the z-index of the mount point to spawn the segments of the tile in a variable, which was also the total length of the track. When the z-position of the Transform of the warrior has been increased by 10 meters (the length of a segment). One segment would be generated and the z-index of the mount point would be added by 10 meters too. The mechanism for generating tiles is that we limit the maximum amount of tiles that can exist in the game to prevent memory leak. When a tile is out of the safe zone, we remove it and we generate new tile at the same time.

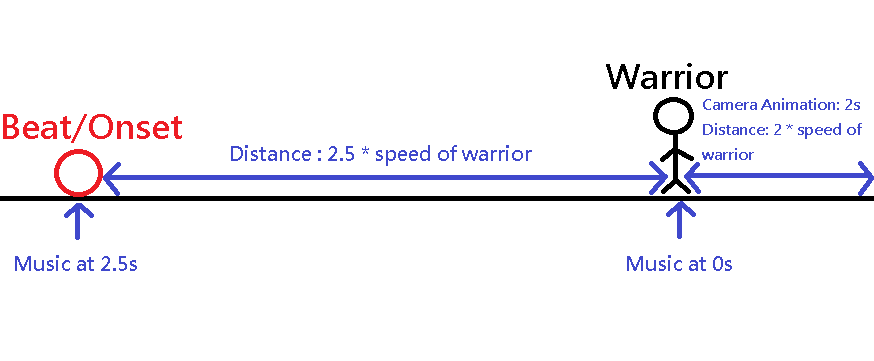
#### **2.4 Sound Control (SoundManager.cs and Song.cs)**

We implemented Song.cs to for a song object. Class Song includes audio clip, pitch and volume as class members. SoundManager.cs is responsible for using the Song object to play the song selected by the users. The script also set up all the sound effects that are used in the game.

Apart from the basic audio control, this script is also playing a core role for audio visualization. It visualizes the music according to the music’s beats and onsets as the game has two modes: beat tracking mode and onset detection mode. For the beat tracking mode, it generates minions according to the beats. For the onset detection mode, it generates minions according to the onsets.

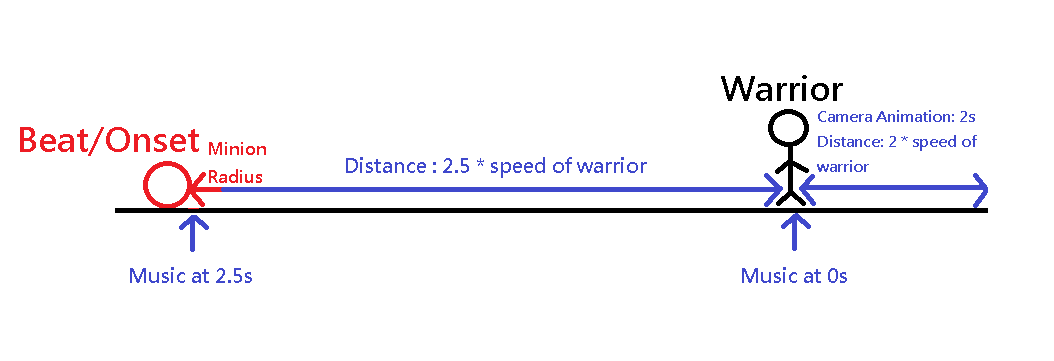
#### **2.5 Audio Visualization (Beat Tracking and Onset Detection)**

We visualize the audio by calculating the timestamp whether its beats or onsets occur. We then generate a minion when a beat or a onset occurs which depends on whether the user selects beat tracking mode or onset detection mode.



(Fig 2.5.1. Z-position of a minion calculation)

When the game starts, there is a camera animation and the warrior will keep moving forward. The music won’t start during the camera animation. After the camera animation, the music starts. The warrior will collide the minion when there is a beat/onset at that timestamp of the music. For instance, if there is a beat/onset at 2.5s, the warrior should encounter a minion after he has ran for 2.5s. The z-position of the minion is the timestamp of that beat/onset times the speed of warrior plus the camera animation duration times the speed of warrior.



However, we need to take into account of the volume of minion. If the minion radius is 0.5m, we need to add 0.5m to the spawning z-position of the minion. By doing so, the contact surface of the minion with the warrior would be exactly where the beat/onset happens. Let the speed of warrior, camera animation duration, beat/onset occurring timestamp and radius of minion be s, a, t and r respectively. The final z-position is calculated as, sa + st + r.

##### **2.5.1 Beat Tracking (SoundManager.cs)**

Beat tracking algorithm first gets the beats per minute (BPM) of the selected song(bpm = getBpm()). After that, it calculates the total number of beats occurred throughout the whole song which is also the total number of minions that will be generated (totalNumOfBeats = clipLength \* 60 / bpm). It then calculates the time interval for occurring a beat (interval = clipLength / totalNumOfBeats). Once we obtains the time interval, total number of beats and the total length of the audio clip, we can can place a minion for each timestamp.

**Pseudo Code:**

bpm = getBpm();

totalNumOfBeats = clipLength \* 60 / bpm;

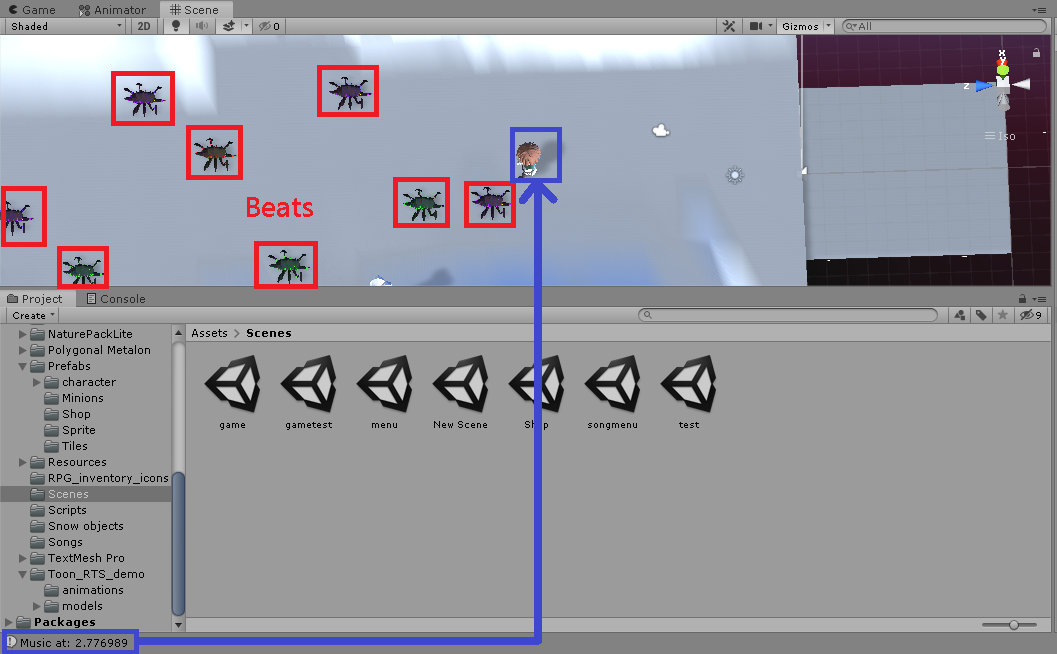
interval = clipLength / totalNumOfBeats;

for (i=0;i<totalNumOfBeats;i++) {

timestamp = interval \* i;

SpawnMinion(timestamp);

}

****

(Fig. 2.5.1.1 The future beats that the character will encounter when the current music is at time 2.77s)

**Beat tracking algorithm:**

SoundManager.cs:

public class SpecificSpectralFluxInfo {

public int time;

public float spectralFlux;

public float threshold;

public float prunedSpectralFlux;

public bool isPeak;

}

void Start() {

timer = 0f;

peakOfPeakSamples = new List<SpectralFluxInfo>();

BpmAnalysis();

InitializeMinions();

Play(PlayerPrefs.GetString("selectedSong"));

}

private void BpmAnalysis() {

Sound s = Array.Find(songs, song => song.name == PlayerPrefs.GetString("selectedSong"));

int bpm = GetBpm(PlayerPrefs.GetString("selectedSong"));

float totalNumOfBeats = s.clip.length / 60 \* bpm;

float interval = s.clip.length / totalNumOfBeats;

for(float i=1.0f;i<=totalNumOfBeats - 5.0f;i+=1.0f) { // Beats at the last can be ignored since the song is fading out and there is not any sound

SpectralFluxInfo info = new SpectralFluxInfo();

info.time = interval \* i;

peakOfPeakSamples.Add(info);

}

}

private int GetBpm(String song) {

switch(song) {

case "faded":

return 90;

case "theFatRat":

return 90;

case "kimetsunoYaiba":

return 135;

case "nonono":

return 106;

default:

}

}

private void InitializeMinions() {

GameObject minionManager = GameObject.Find("MinionManager");

for(int i=0;i<20;i++) { // Generate 20 minions first. Later on when we will delete out of view minions and generate new minions according to the beats, to ensure there are 20 minions at most since generate a gameObject consume many resources.

minionManager.GetComponent<MinionManager>().SpawnMinion(peakOfPeakSamples[i].time);

}

}

public void Play(string name) {

Sound s = Array.Find(songs, song => song.name == name);

if (s == null) {

return;

}

s.source.PlayDelayed(animationDuration);

}

MinionManager.cs:

public void SpawnMinion(float z) {

GameObject go;

float[] spawnPos = {-5.0f, -1.66f, 1.66f, 5.0f};

go = Instantiate(minionPrefabs[RandomPrefabIndex()]) as GameObject;

go.transform.SetParent(transform);

// 0.5f is half of the z of the minion

go.transform.position = new Vector3(spawnPos[Random.Range(0, 4)], 0.53f, z \* speed + tileLength);

go.transform.localEulerAngles = new Vector3(0.0f, 180f, 0.0f);

go.transform.localScale = new Vector3(0.5f, 0.5f, 0.5f);

activeMinions.Add(go);

}

##### 

##### **2.5.2 Onset Detection (SpectralFluxAnalyzer.cs and DSPLib.cs)**

Since Unity only provides API for real-time audio analysis which does not fit our pre-process purpose, we referenced to an article by Jesse Scam for doing preprocessed audio analysis [1]. We use a 5-step approach to achieve onset detection.

The first step is to get the original sample data and attributes from the song. We initialize AudioSource(which is class provided by Unity) by input the .mp3 file of the song. The AudioSource returns the audio clip’s sampling rate, channels and number of samples. Then, it uses Unity’s function GetData(samples, 0) to get all the sample data from the song for audio analysis.

**Pseudo Code:**

AudioSource song = getSelectedSong();

float samplingRate = s.frequency;

int channels = s.channels;

int numOfSamples = s.samples;

float clipLength = s.length;

float[] samples = new float[song.samples \* song.channels];

samples = GetData(song);

The second step is to calculate pre-processed audio signal from the samples, we implemented function GetOutputData() to process all channel samples for each timestamp and output the average of the channels combined.

**Pseudo Code:**

float[] getOuputData(float[] samples) {

float[] preprocessedSamples = new float[numOfSamples];

int numProcessed = 0;

float combinedChannelAvg = 0f;

for(int i=0;i<samples.length;i++) {

if ((i+1)%channels == 0) {

preprocessedSamples[numProcessed] = combinedChannelAvg / channels;

numProcessed++;

combinedChannelAvg = 0f;

}

}

return preprocessedSamples;

}

The third step is to use a reduction function that extracts spectral features and reduces the complexity of the signal. We invoke GetSpectrumData() to execute Steve’s Hageman’s fast fourier transform (FFT) implementation on the pre-processed samples to get the spectrum (complex number). We choose to apply Hanning window for FFT because it forces the amplitude of the time record to zero at the beginning and end of the sample interval. Hanning weighted record is zero at both ends. Therefore, no distortion is created during FFT looping process and only actual frequencies in the signal are calculated. The calculated spectrum values can be more accurate. Finally, we scale and convert the two-sided spectrum in complex form (real and imaginary parts) to polar form to obtain magnitude.

**Pseudo Code:**

function getSpectrumData(float[] samples) {

float[] preProcessedSamples = GetOutputData(samples);

int spectrumSampleSize = 1024;

int iterations = preProcessedSamples.length / spectrumSampleSize;

FFT fft = new FFT();

double sampleChunk = new double[spectrumSampleSize];

for(int i=0;i<iterations;i++) {

Array.Copy(preProcessedSamples, i \* spectrumSampleSize, sampleChunk, 0, spectrumSampleSize);

double[] windowCoefs = DSP.Window.Coefficients(DSP.Window.Type.Hanning, spectrumSampleSize);

double[] scaledSpectrumChunk = DSP.Math.Multiply(sampleChunk, windowCoefs);

double scaleFactor = DSP.Window.ScaleFactor.Signal(windowCoefs);

Complex[] fftSpectrum = fft.Execute(scaledSpectrumChunk);

double[] scaledFFTSpectrum = DSP.ConvertComplex.ToMagnitude(fftSpectrum);

scaledFFTSpectrum = DSP.Math.Multiply(scaledFFTSpectrum, scaledFactor);

float curSongTime = getTimeFromIndex(i) \* spectrumSampleSize;

preProcessedSpectralFluxAnalyzer.analyzeSpectrum(Array.ConvertAll(scaledFFTSpectrum, x => (float)x), curSongTime);

}

}

function getTimeFromIndex(int index) {

return ((1f / (float) samplingRate) \* index);

}

The fourth step is peak detection function. We pass the magnitude data to SpectralFluxAnalyzerwhich it a library created by Jesse Scam. It is used to analyze the spectral information of magnitude data. It gets the rectified spectral flux by aggregate positive changes in spectrum data. It then adds the spectral flux samples to a sample list. If it has enough samples(threshold window size = 50) for detecting a peak, it calculates the flux threshold. The flux threshold is the average of spectral flux over the window times the threshold multiplier (1.5). After threshold calculation, it keeps amp amount above threshold to allow peak filtering. Finally, it compares the current spectral flux sample’s amp amount with its previous candidate’s and the next candidate’s amp amount. If it is the highest, it should be a peak.

The fifth step is peak picking function. After we analyze the song and get all its peak, we then compare a peak with the previous peak and the next peak. If that peak has higher spectral flux than the previous and the next one, we recognize that peak as an onset. We then place a minion at that timestamp when the onset occurs.

**Pseudo Code:**

function getPeakOfPeakSamples(SpectralFluxInfo[] peakSamples) {

for(int i=0;i<peakSamples.length;i++) {

if(i == 0) {

if (peakSamples[0].prunedSpectralFlux > peakSamples[i].prunedSpectralFlux) {

peakOfPeakSamples.Add(peakSamples[i]);

} else if (i == peakSamples.length - 1) {

if (peakSamples[i].prunedSpectralFlux > peakSamples[i-1].prunedSpectralFlux) {

peakOfPeakSamples.Add(peakSamples[i]);

}

} else {

if (peakSamples[i-1].prunedSpectralFlux < peakSamples[i].prunedSpectralFlux && peakSamples[i+1].prunedSpectralFlux < peakSamples[i].prunedSpectralFlux) {

peakOfPeakSamples.Add(peakSamples[i]);

}

}

}

}

}

**Onset detection algorithm:**

SoundManager.cs

public class SpecificSpectralFluxInfo {

public int time;

public float spectralFlux;

public float threshold;

public float prunedSpectralFlux;

public bool isPeak;

}

void Start() {

timer = 0f;

peakOfPeakSamples = new List<SpectralFluxInfo>();

SpectralFluxAnalysis();

InitializeMinions();

Play(PlayerPrefs.GetString("selectedSong"));

}

private void SpectralFluxAnalysis() {

Sound s = Array.Find(songs, song => song.name == PlayerPrefs.GetString("selectedSong"));

preProcessedSpectralFluxAnalyzer = new SpectralFluxAnalyzer();

samplingRate = s.source.clip.frequency;

channels = s.source.clip.channels;

numOfSamples = s.source.clip.samples;

clipLength = s.source.clip.length;

samples = new float[s.source.clip.samples \* s.source.clip.channels];

// the sample data is stored as [L, R, L, R, L, R, ...] in the array

s.source.clip.GetData(samples, 0);

GetSpectrumData(samples);

List<SpectralFluxInfo> peakSamples = preProcessedSpectralFluxAnalyzer.spectralFluxSamples.FindAll(IsPeakSample);

GetPeakOfPeakSamples(peakSamples);

}

private bool IsPeakSample(SpectralFluxInfo info) {

if (info.isPeak) {

return true;

} else {

return false;

}

}

private void GetPeakOfPeakSamples(List<SpectralFluxInfo> peakSamples) {

for(int i=0;i<peakSamples.Count;i++) {

// Find the strongest onset from a set of peaks. Filter the peaks that are too close to each other

if (i==0) {

if (peakSamples[0].prunedSpectralFlux > peakSamples[1].prunedSpectralFlux) {

peakOfPeakSamples.Add(peakSamples[i]);

}

} else if (i== peakSamples.Count-1) {

if (peakSamples[i].prunedSpectralFlux > peakSamples[i-1].prunedSpectralFlux) {

peakOfPeakSamples.Add(peakSamples[i]);

}

} else {

if (peakSamples[i-1].prunedSpectralFlux < peakSamples[i].prunedSpectralFlux && peakSamples[i+1].prunedSpectralFlux < peakSamples[i].prunedSpectralFlux) {

peakOfPeakSamples.Add(peakSamples[i]);

}

}

}

}

// return data that is of the same format as what’s returned in real-time by AudioSource.GetOutputData, but it contains samples from the beginning of the track until the end instead of just the samples for the currently playing audio.

private float[] GetOutputData(float[] samples) {

float[] preprocessedSamples = new float[numOfSamples];

int numProcessed = 0;

float combinedChannelAvg = 0f;

for (int i=0; i<samples.Length; i++) {

combinedChannelAvg += samples[i];

// Each time we have processed all channels samples for a point in time, we will store the average of the channels combined

if ((i + 1) % channels == 0 ) {

preprocessedSamples[numProcessed] = combinedChannelAvg / channels;

numProcessed++;

combinedChannelAvg = 0f;

}

}

return preprocessedSamples;

}

private void GetSpectrumData(float[] samples) {

try{

float[] preProcessedSamples = GetOutputData(samples);

Debug.Log("Combine Channels done");

Debug.Log(preProcessedSamples.Length);

// Once we have our audio sample data prepared, we can execute an FFT to return the spectrum data over the time domain

int spectrumSampleSize = 1024;

int iterations = preProcessedSamples.Length / spectrumSampleSize;

FFT fft = new FFT();

fft.Initialize ((UInt32)spectrumSampleSize);

Debug.Log (string.Format("Processing {0} time domain samples for FFT", iterations));

double[] sampleChunk = new double[spectrumSampleSize];

for(int i=0;i<iterations;i++) {

// Grab the current 1024 chunk of audio sample data

Array.Copy (preProcessedSamples, i \* spectrumSampleSize, sampleChunk, 0, spectrumSampleSize);

// Apply our chosen FFT Window

double[] windowCoefs = DSP.Window.Coefficients (DSP.Window.Type.Hanning, (uint)spectrumSampleSize);

double[] scaledSpectrumChunk = DSP.Math.Multiply (sampleChunk, windowCoefs);

double scaleFactor = DSP.Window.ScaleFactor.Signal (windowCoefs);

// Perform the FFT and convert output (complex numbers) to Magnitude

Complex[] fftSpectrum = fft.Execute (scaledSpectrumChunk);

double[] scaledFFTSpectrum = DSPLib.DSP.ConvertComplex.ToMagnitude (fftSpectrum);

scaledFFTSpectrum = DSP.Math.Multiply (scaledFFTSpectrum, scaleFactor);

// These 1024 magnitude values correspond (roughly) to a single point in the audio timeline

float curSongTime = getTimeFromIndex(i) \* spectrumSampleSize;

// Send our magnitude data off to our Spectral Flux Analyzer to be analyzed for peaks

preProcessedSpectralFluxAnalyzer.analyzeSpectrum (Array.ConvertAll (scaledFFTSpectrum, x => (float)x), curSongTime);

Debug.Log ("Spectrum Analysis done");

Debug.Log ("Background Thread Completed");

}

} catch (Exception e) {

Debug.Log(e.ToString());

}

}

public float getTimeFromIndex(int index)

{

return ((1f / (float)samplingRate) \* index);

}

public void Play(string name) {

Sound s = Array.Find(songs, song => song.name == name);

if (s == null) {

return;

}

s.source.PlayDelayed(animationDuration);

}

SpectralFluxAnalyzer.cs

public class SpectralFluxInfo {

public float time;

public float spectralFlux;

public float threshold;

public float prunedSpectralFlux;

public bool isPeak;

}

public class SpectralFluxAnalyzer {

int numSamples = 1024;

// Sensitivity multiplier to scale the average threshold.

// In this case, if a rectified spectral flux sample is > 1.5 times the average, it is a peak

float thresholdMultiplier = 1.5f;

// Number of samples to average in our window

int thresholdWindowSize = 50;

public List<SpectralFluxInfo> spectralFluxSamples;

float[] curSpectrum;

float[] prevSpectrum;

int indexToProcess;

public SpectralFluxAnalyzer () {

spectralFluxSamples = new List<SpectralFluxInfo> ();

// Start processing from middle of first window and increment by 1 from there

indexToProcess = thresholdWindowSize / 2;

curSpectrum = new float[numSamples];

prevSpectrum = new float[numSamples];

}

public void setCurSpectrum(float[] spectrum) {

curSpectrum.CopyTo (prevSpectrum, 0);

spectrum.CopyTo (curSpectrum, 0);

}

public void analyzeSpectrum(float[] spectrum, float time) {

// Set spectrum

setCurSpectrum(spectrum);

// Get current spectral flux from spectrum

SpectralFluxInfo curInfo = new SpectralFluxInfo();

curInfo.time = time;

curInfo.spectralFlux = calculateRectifiedSpectralFlux ();

spectralFluxSamples.Add (curInfo);

// We have enough samples to detect a peak

if (spectralFluxSamples.Count >= thresholdWindowSize) {

// Get Flux threshold of time window surrounding index to process

spectralFluxSamples[indexToProcess].threshold = getFluxThreshold (indexToProcess);

// Only keep amp amount above threshold to allow peak filtering

spectralFluxSamples[indexToProcess].prunedSpectralFlux = getPrunedSpectralFlux(indexToProcess);

// Now that we are processed at n, n-1 has neighbors (n-2, n) to determine peak

int indexToDetectPeak = indexToProcess - 1;

bool curPeak = isPeak (indexToDetectPeak);

if (curPeak) {

spectralFluxSamples [indexToDetectPeak].isPeak = true;

}

indexToProcess++;

}

else {

Debug.Log(string.Format("Not ready yet. At spectral flux sample size of {0} growing to {1}", spectralFluxSamples.Count, thresholdWindowSize));

}

}

float calculateRectifiedSpectralFlux() {

float sum = 0f;

// Aggregate positive changes in spectrum data

for (int i = 0; i < numSamples; i++) {

sum += Mathf.Max (0f, curSpectrum [i] - prevSpectrum [i]);

}

return sum;

}

float getFluxThreshold(int spectralFluxIndex) {

// How many samples in the past and future we include in our average

int windowStartIndex = Mathf.Max (0, spectralFluxIndex - thresholdWindowSize / 2);

int windowEndIndex = Mathf.Min (spectralFluxSamples.Count - 1, spectralFluxIndex + thresholdWindowSize / 2);

// Add up our spectral flux over the window

float sum = 0f;

for (int i = windowStartIndex; i < windowEndIndex; i++) {

sum += spectralFluxSamples [i].spectralFlux;

}

// Return the average multiplied by our sensitivity multiplier

float avg = sum / (windowEndIndex - windowStartIndex);

return avg \* thresholdMultiplier;

}

float getPrunedSpectralFlux(int spectralFluxIndex) {

return Mathf.Max (0f, spectralFluxSamples [spectralFluxIndex].spectralFlux - spectralFluxSamples [spectralFluxIndex].threshold);

}

bool isPeak(int spectralFluxIndex) {

if (spectralFluxSamples [spectralFluxIndex].prunedSpectralFlux > spectralFluxSamples [spectralFluxIndex + 1].prunedSpectralFlux &&

spectralFluxSamples [spectralFluxIndex].prunedSpectralFlux > spectralFluxSamples [spectralFluxIndex - 1].prunedSpectralFlux) {

return true;

} else {

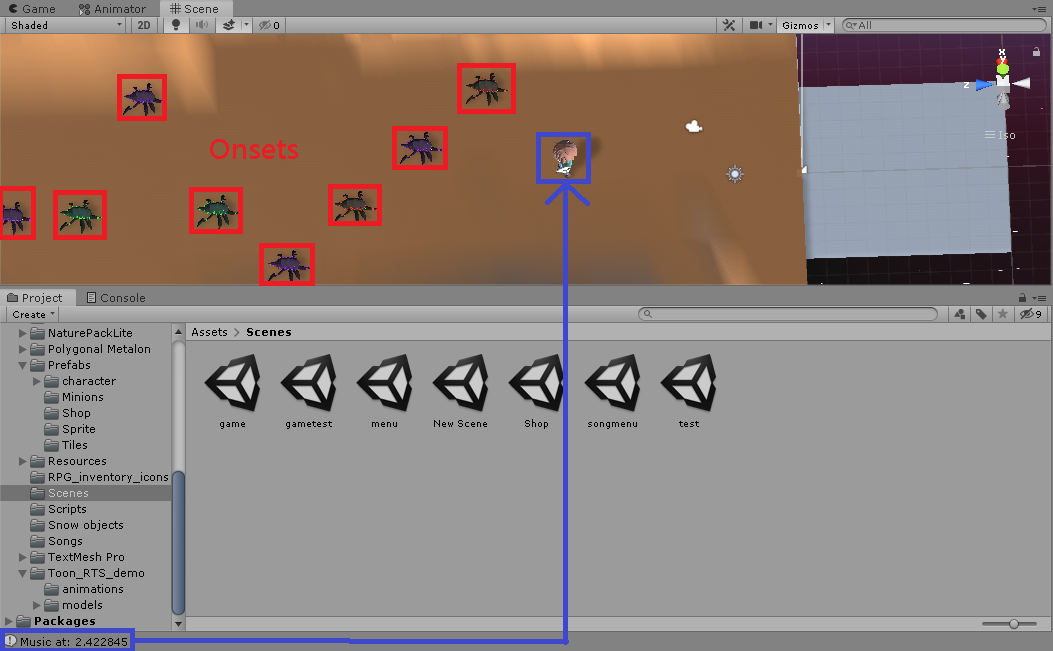
return false;

}

}

}

The computation time for FFT is O(nlogn), n is the number of pre-processed samples (all channel samples for each timestamp). For example, Faded (44100 Hz sampling rate, stereo channels, 3:32, MP3 format) has 9373824 pre-processed samples. There will be 65353521 iterations. It is very time-consuming in executing the algorithm. Therefore, we decided to avoid it during the game. We would execute the algorithm in Unity and output the extracted spectral flux information (including the timestamps) as .json formatted data and store the file in Resource/Json folder. Therefore, in onset mode, we can directly read the calculated timestamp from spectral flux information .json file of the selected song and print the minions according to the timestamp instead of executing the algorithm when the user enter this mode.



(Fig. 2.5.1.2 The future onsets that the character will encounter when the current music is at time 2.42s)

#### 

#### **2.6 Minion Control (MinionManager.cs)**

We implemented MinionManager.cs to spawn minion and the minion z-position is the character’s speed times the timestamp of that beat or onset plus a tile length plus 0.5 meters (which is half of the z-length of the minion). SpawnMinion() is invoked in SoundManager.cs since the minions are spawn according to the beats or onsets we extract from the song.

##### **2.6.1 Individual Minion (Minion.cs)**

We implemented a separate script that each minions will run to control each minions’ behaviors. The destruction and animator control will be done in this C# script. We destroy minion when its z-position is smaller than or equal to the z-position of the warrior - 6f. When we delete a minion, we check whether it is killed by the warrior or not. If it is not killed by warrior, we reduce the health of warrior by 10. We then spawn a new minion according to the beat or onset. Finally, we destroy that minion as it is out of view to prevent memory leak.

#### **2.7 Game Management (HealthBar.cs, Score.cs, DeathMenu.cs, PauseMenu.cs)**

We implemented the health bar, score and death menu to decide whether the game ends or not and manage the behaviors after the game ends.

##### **2.7.1 Health Bar (HealthBar.cs)**

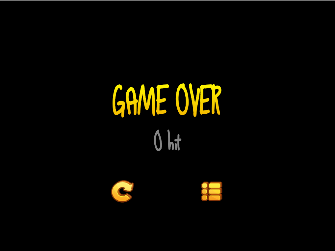
HealthBar.cs manages the length of the health bar. OnTakeDamage() is invoked in Minion.cs since it detects whether the warrior has missed any minion. It triggers the death menu to pop up when the warrior takes too much damage.

##### **2.7.2 Score (Score.cs)**

Score.cs records how many minions have been destroyed by the character and displays the score value on the score label in UI. If the warrior is dead, it invokes OnDeath(score) to pop up the death menu and pass the score to display on the death menu.

##### **2.7.3 Death Menu (DeathMenu.cs)**

DeathMenu.cs creates menu that displays the value of the score and provides buttons for users to navigate back to the main menu or restart. Death menu pops up when the character lose all his health or the music is ended.



(Fig 2.7.3 Death Menu)

##### **2.7.4 PauseMenu (PauseMenu.cs)**

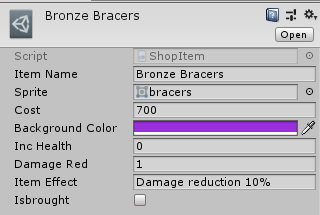
Users can press “Esc” to pause the game when s/he is playing. The game and music will then pause. A menu would pop up and it provides 3 options, including “Resume”, “Restart” and “Quit”.



(Fig 2.7.4 Pause Menu)

#### **2.8 Shop and Status(Shop.cs)**

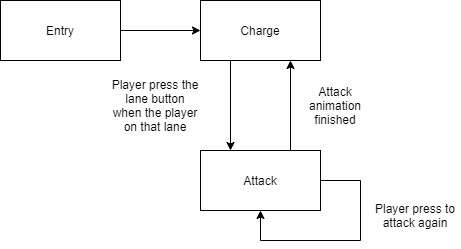
The user can check the status, health and number of coins, inside this scene. If the user is the first time to play the game, the health will be 100 and the number of coins will be 0. The shop item will be populated by the script Shop.PopulateShop(). The function will go through each asset files that are defined inside Unity. Each items are created within the Unity environment with a scriptable object class. The data of each items can be edited inside Unity. It is easier for developer to add new or edit the existing items. There are also a return button at the bottom right corner to back to the main screen.



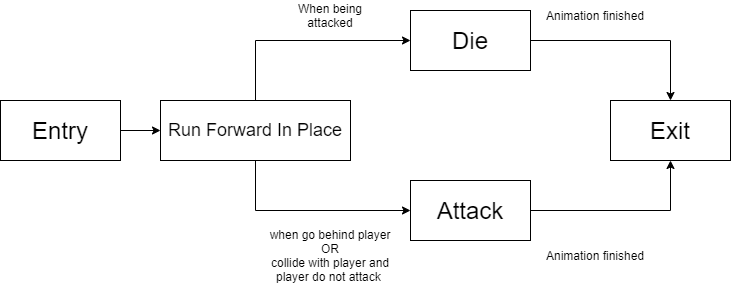
(Fig. 2.8.1 A example of the ShopItem Scriptable Object)

### **3.** **Animators**

We would develop a **state machine** to include set of states in **Animator Controller** that the warrior and the minions can be in, along with a set of transitions between these states and a variable to remember the current state.



(Fig. 2.2.3.1 State diagram for the warrior)



(Fig. 2.2.3.2 State diagram for the minions)

**4. Data storage**

Unity provides **PlayPrefs** that stores and access player preferences between game sessions. We use **PlayerPrefs** to store the highest score of each game level and status of equipments in shop. Therefore, the users can know the highest score of each game level and whether the equipments are purchased whenever s/he comes back to the game.

## **2.3** **Testing**

During the development process, unit testing will be done to ensure all modules are built correctly. System integration testing will be done after we have built all the components and combined them into the application.

### **2.3.1** **Physical Objects**

We will use Unity’s build-in game simulation environment to test whether the animation for physical motions of the warrior, minions and the boss, can be displayed properly under the player’s control (left, right, up, down key). Then, we will integrate the warrior and minions to test whether the collision can be detected accurately and also the corresponding animation.

### **2.3.3** **User Interface**

We will click on each icon or button to test whether it can perform a proper update.

**2.4** **Evaluation**

After we have finished all the testing, we will evaluate the system to check whether it fulfills our objectives or not.

1. Does the game level successfully generate minions according to the music beats?

2. Does the game generate proper sound effect (music beat) when the warrior collides with the minions?

3. Can the ranking system update and calculate the best result for each game level among all players properly?

4. Can the warrior, minions and the boss behave (run, change track and attack) expectedly without any bug?

5. Can RPG elements (character experience, money, store, equipments, game progress) be implemented and integrated successfully?

# **3. Discussion**

1. Combine RPG elements with music rhythm game

2. Develop a user-friendly user interface for players

3. Integrate a ranking system into the game

4. Convert MIDI files into usable format for rhythm processing

To achieve the objectives, we will:

1. Develop 3D model and write game logic with Unity

2. Research audio processing tools to analyzing songs to useful data

3. Devote our time and effort to build an enjoyable and relaxing rhythm game

4. Consult people from friends to professor to examine and comment the game

# **4. Conclusion**

**4.1 The key things we did and learned**

1. Digital signal processing. We are all computer science students and we have never studied any subject about digital signal processing as it is the core course of electronic engineering. Therefore, it is difficult for us to implement the beat tracking and onset detection algorithm as we don’t have any knowledge about audio analysis. Yet, we find a lot of readings and code examples from internet so we can implemented the beat tracking and onset detection algorithm finally.
2. Unity. We learned how to use this game engine to create scene navigation, game scenes, 2D plane image, UI (font style, font color, font shadow), 3D game objects (character controller, animator state machine, collision detection, C# scripting, tag system), audio assets and graphic assets management. We also learned to get and set data from PlayerPrefs and JSON formatted data storage.
3. Git. We learned to put the project on github so the team members can collaborate. Each of us own a branch and develop new features on the branch. Once it is finished, s/he create a pull request to merge the new features on the branch master. We also learned to take advantage of github free hosting service to host our game on github page.

**4.1 Improvement**

There are still a lot of rooms for our beat tracking algorithm, onset detection algorithm and character customization.

For the current beat tracking algorithm, we get the bpm directly by hard-coding in the code because a song is usually created with its beat. However, we can improve it by creating a beat calculation algorithm that outputs the bpm of a song and all we need to do is just input the song data to the algorithm.

Onset detection is still an active research area. Therefore, our onset detection algorithm may be not the most accurate algorithm. Onset detection methods can include looking for increase in spectral energy, change in spectral energy distribution (spectral flux), changes in detected pitch and spectral pattern found by neural networks. Thus, we can improve our system by providing different methods and take the best result to generate the minions.

By the current stage, the users can buy equipments from shop. However, the equipments are not mounted on the character in the game and they are just a 2D image. We can improve it by creating the equipments in 3D and its corresponding mount point so that the character can actually equip the items.

# **5.** **References**

[1] Jesse, S. (2018). Algorithmic Beat Mapping in Unity. [online] Available at <https://medium.com/giant-scam/algorithmic-beat-mapping-in-unity-preprocessed-audio-analysis-d41c339c135a> [Accessed 17 Feb. 2020]

[2] Play.google.com. (2019). Cytus - Google Play Application. [online] Available at: https://play.google.com/store/apps/details?id=com.rayark.Cytus.full [Accessed 1 Sep. 2019].

[3] Martin, G. (2019). The 25 Best Videogames of 2016. [online] pastemagazine.com. Available at: https://www.pastemagazine.com/articles/2016/12/the-25-best-videogames-of-2016.html?p=3 [Accessed 3 Sep. 2019].

[4] Navgtr.org. (2019). Winner List for 2018: God of War Breaks Record | NAVGTR. [online] Available at: https://navgtr.org/2019/03/13/winner-list-for-2018-god-of-war-breaks-record/ [Accessed 3 Sep. 2019].

[5] Steve, H. (2018). DSPLib - FFT / DFT Fourier Transform Library for .NET 4. [online] Available at: <https://www.codeproject.com/Articles/1107480/DSPLib-FFT-DFT-Fourier-Transform-Library-for-NET-6> [Accessed 28 Apr 2020]

# 

# **6. Appendix A: Meeting Minutes**

## **6.1** **Minutes of the 1st Project Meeting**

Date: Aug 1, 2019

Time: 4:00 pm

Place: Room 3549

Present: Sam, Sarah, Victor, Prof. Honor

Absent: None

Recorder: Sarah

1. Approval of minutes

This was the first formal group meeting, so there were no minutes to approve.

2. Report on progress

2.1 All team members have read the instructions of the Final Year Project online and have done research for the topic.

2.2 All team members present their idea of music rhythm game

3. Discussion items

3.1 The expectation of music rhythm game as a FYP.

3.2 The important grading criteria using music rhythm game as a FYP.

3.3 What is the previous experience of doing a game FYP.

3.4 Platform to use or tools to develop the game.

4. Goals for the coming week

4.1 All group members will read more information related to the project topic, e.g. genre, processing audio.

4.2 All group members will need to study and compare the languages and software being considered for implementation of the project.

4.3 All group members will continue to think about the game idea.

5. Meeting adjournment and next meeting

The meeting was adjourned at 6:00 pm.

The date and time of the next meeting will be set later by e-mail.

# **6.2 Project Planning**

## **6.2.1** **Distribution of Work**

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Victor** | **Sam** | **Sarah** |
| Do the Literature Survey | ● | ○ | ○ |
| Design the Game Logic | ○ | ● | ○ |
| Design the Database | ● | ○ | ○ |
| Design Game Scenes and Characters | ○ | ○ | ● |
| Design the User Interface | ○ | ● | ○ |
| Design the ranking system | ○ | ● | ○ |
| Process MIDI | ○ | ○ | ● |
| Build the Database | ● | ○ | ○ |
| Develop the Game | ○ | ● | ○ |
| Build the User Interface | ○ | ○ | ● |
| Test the Game Logic | ● | ○ | ○ |
| Test the Database | ● | ○ | ○ |
| Test the Ranking System | ○ | ○ | ● |
| Test the User Interface | ○ | ● | ○ |
| Perform Integration Testing | ● | ○ | ○ |
| Write the Proposal | ● | ○ | ○ |
| Write the Monthly Reports | ○ | ● | ○ |
| Write the Progress Report | ○ | ○ | ● |
| Write the Final Report | ● | ○ | ○ |
| Prepare for the Presentation | ○ | ○ | ● |
| Design the Project Poster | ○ | ○ | ● |

● Leader ○ Assistant

**6.2.2** **GANTT Chart**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Task | July | Aug | Sep | Oct | Nov | Dec | Jan | Feb | Mar | Apr |
| Do the Literature Survey |  |  |  |  |  |  |  |  |  |  |
| Design the Game Logic |  |  |  |  |  |  |  |  |  |  |
| Design the Database |  |  |  |  |  |  |  |  |  |  |
| Design Game Scenes and Characters |  |  |  |  |  |  |  |  |  |  |
| Design the User Interface |  |  |  |  |  |  |  |  |  |  |
| Design the ranking system |  |  |  |  |  |  |  |  |  |  |
| Process MIDI |  |  |  |  |  |  |  |  |  |  |
| Build the Database |  |  |  |  |  |  |  |  |  |  |
| Develop the Game |  |  |  |  |  |  |  |  |  |  |
| Build the User Interface |  |  |  |  |  |  |  |  |  |  |
| Test the Game Logic |  |  |  |  |  |  |  |  |  |  |
| Test the Database |  |  |  |  |  |  |  |  |  |  |
| Test the Ranking System |  |  |  |  |  |  |  |  |  |  |
| Test the User Interface |  |  |  |  |  |  |  |  |  |  |
| Perform Integration Testing |  |  |  |  |  |  |  |  |  |  |
| Write the Proposal |  |  |  |  |  |  |  |  |  |  |
| Write the Monthly Reports |  |  |  |  |  |  |  |  |  |  |
| Write the Progress Report |  |  |  |  |  |  |  |  |  |  |
| Write the Final Report |  |  |  |  |  |  |  |  |  |  |
| Prepare for the Presentation |  |  |  |  |  |  |  |  |  |  |
| Design the Project Poster |  |  |  |  |  |  |  |  |  |  |

# 

# **6.3. Required Hardware & Software**

## **6.3.1** **Hardware**

For development,

|  |  |
| --- | --- |
| Operating system | Windows: 7 SP1+, 8, 10, 64-bit versions only  macOS: 10.12+  Linux: Fixed at: Ubuntu 16.04, 18.04 and CentOS 7  Server versions of Windows and OS X are untested. |
| CPU | SSE2 instruction set support |
| GPU | Graphics card with DX10 (shader model 4.0) capabilities |

For running the game,

|  |  |
| --- | --- |
| WebGL | Any recent desktop version of Firefox, Chrome, Edge or Safari |

## **6.3.2** **Software​​​​​​​​​​​​**

|  |  |
| --- | --- |
| **Software** | **Purpose** |
| Unity | Game development |
| Github | Source control and version control |

## 

## **6.2** **Minutes of the 2nd Project Meeting**

Date: Aug 26, 2019

Time: 10:00 am

Place: Barn A

Present: Sarah, Sam, Victor

Absent: Prof. Honor

Recorder: Sarah

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

2.1 All team members have read the instructions of the Final Year Project online and have done research for the topic.

2.2 Sam has done research on Unity.

2.3 Sarah and Victor have done research on music rhythm game in the market.

2.4 All team members have understood the guidance provided by Prof. Honor.

3. Discussion items

3.1 Genre of game include apart from music rhythm.

3.2 Programming software, framework to develop the game.

3.3 Tools or framework to convert Midi to usable file format.

3.4 The project plan needs to include a list of the main tasks, who will work on each task and a GANTT chart.

4. Goals for the coming week

4.1 All group members will continue thinking the game idea.

4.2 All group members will continue researching platform other than Unity to develop the game.

4.3 Sam will research Unity as Unity will probably be our development platform.

4.4 Sam will research in Unity store to find some audio processing framework.

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 1:00 pm.

5.2 The next meeting will be set later by e-mail.