HO 2 Music Rhythm Game
Music Warrior

Ho Wai Kin (Sam, 2044 7589)
Ho Tsz Kiu (Sarah, 20424236)
Fung Hing Lun (Victor, 20448193)

Introduction

Music Warrior 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 are incorporated in our game to enhance the player's experience.

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

Overview

01 Design

- **Game engine**
- **Storyline**
- **Gameplay**
- **Character design**
- **Game Scenes**

02 Implementation

- **Game level**
  Camera | Character | Runway
  Animation | Game Management
  Data Storage
- **Audio Processing & visualization**

03 Testing

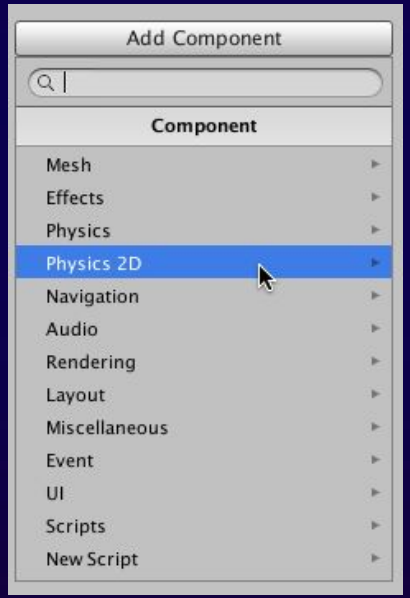
04 Evaluation


DESIGN

Design - Game engine

- cross-platform games
- functions for 2D/3D games
- game objects+components+C# scripts
- comprehensive API for scripting

Components + Scripts

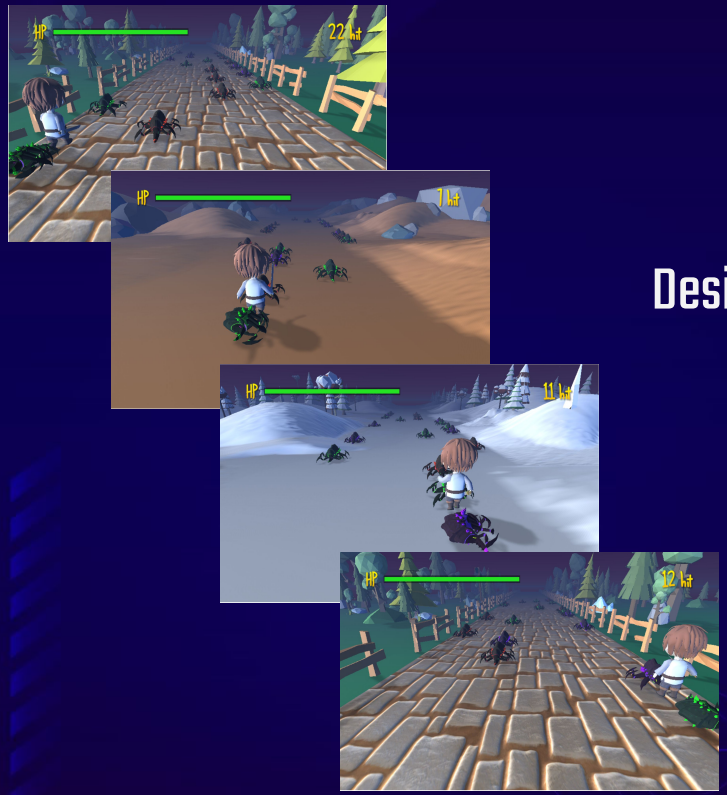



unity
WebGL™

- cross-platform games
- functions for 2D/3D games
- game objects+components+C# scripts
- comprehensive API for scripting

Design - Storyline

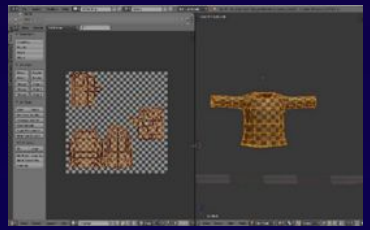
- The world is approaching doomsday
- Invasion by giant monsters
- Player plays a warrior role hunting down minions sent by the enemies, hoping to defeat the monsters and save the day



Design - Gameplay

- 4 levels with different songs
- Minions are generated according to the music beats or onset
- Player move onto the correct runway by pressing “A”, “S”, “D” or “F” key to hit the minions
- Missed any minion → lose life points

CHARACTER DESIGN



2. UV Mapping

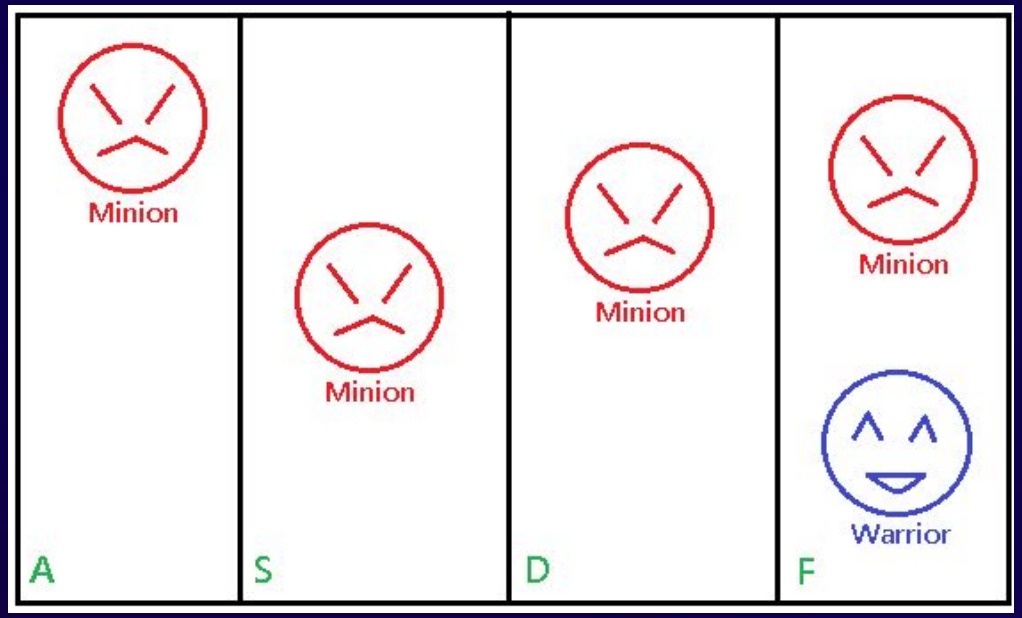
4. Rigging

3. Texture painting

1. Modelling

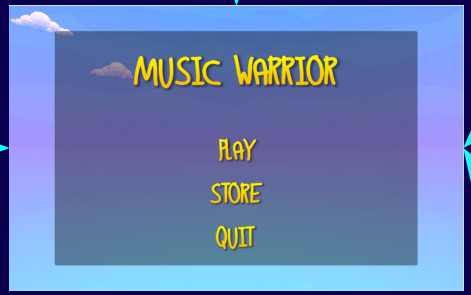
5. Animating

Gameplay



Navigation

start



Game scene 1 - Main Menu

Game scene 2 - Song Menu

Game scene 3 - Game

Game Pause in “Game” Scene

Game Over in “Game” Scene

Game scene 4 - Shop

Game scene 2: Song Menu

Game scene 3: Game

Game Pause

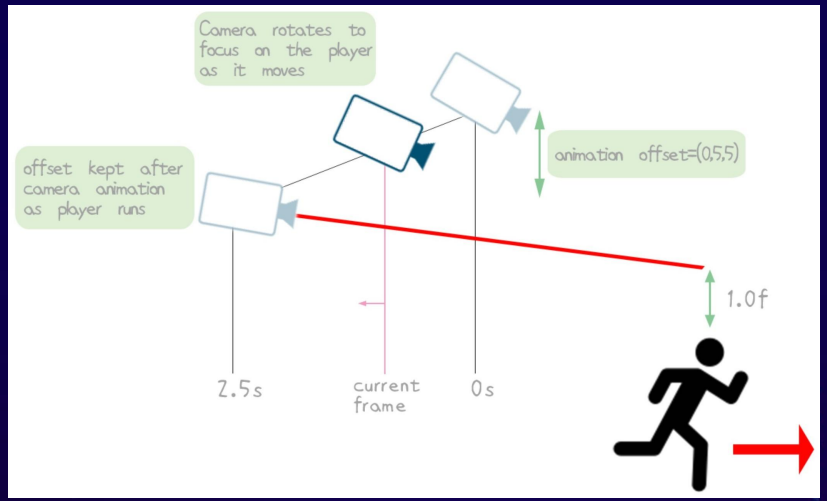
Game Over

Game scene 4: Shop

IMPLEMENTATION

Implementation - Camera Control

- Camera motion: from top to behind
- first 2.5 second
  - look at the Warrior
- after 2.5 second
  - frozen the rotation and follow the Warrior



Implementation - Character Control

Forward movement

- The speed: designated speed / frame per second(fps)
- Player do not have any control of the forward motion

Horizontal movement (switching lane)

- The first 2.5 second is freeze.
- Character move between 4 lanes by pressing “A”, “S”, “D”, and “F”
- The speed: the distance of warrior to the targeted lane / 0.15

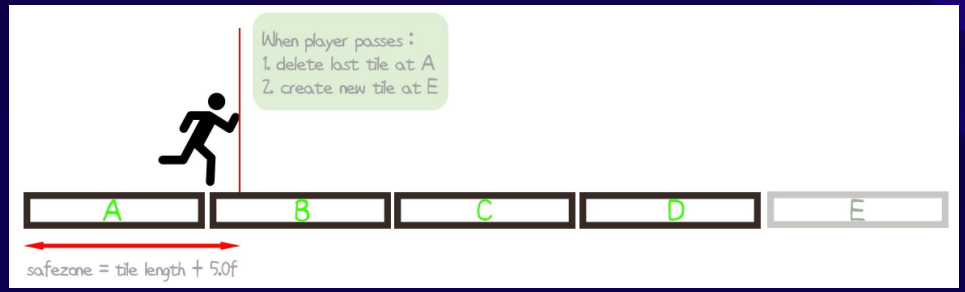
Attack action

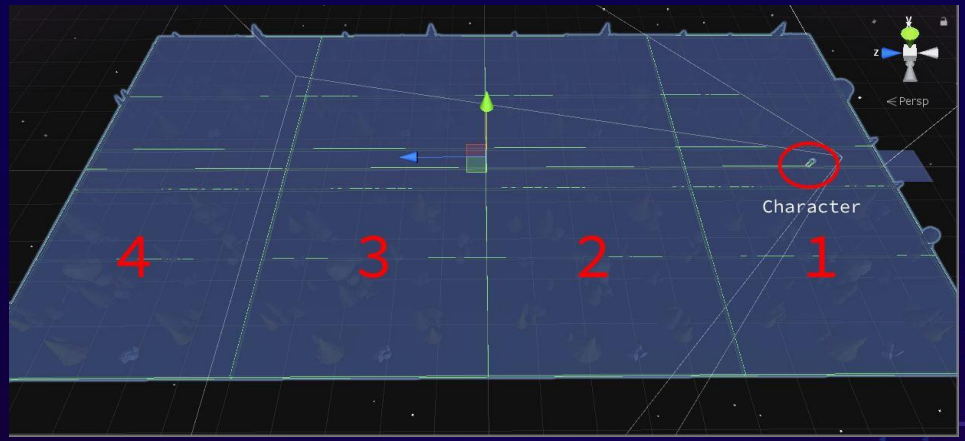
- similar to horizontal movement, pressing “A”, “S”, “D”, “F” will start attack
- The attack animation is inturrupted when another key is pressed

```
// determine the Waarrior is on the Lane or not
if (transform.position.x > target - 0.2f && transform.position.x < target + 0.2f)
{
    if (transform.position.x > target - 0.1f && transform.position.x < target + 0.1f)
        leftOrRight = 0;
}
else
    leftOrRight = 1;
// X - Left and Right
moveVector.x = leftOrRight * (target - transform.position.x) / 0.15f;
```

Implementation - Runway Control

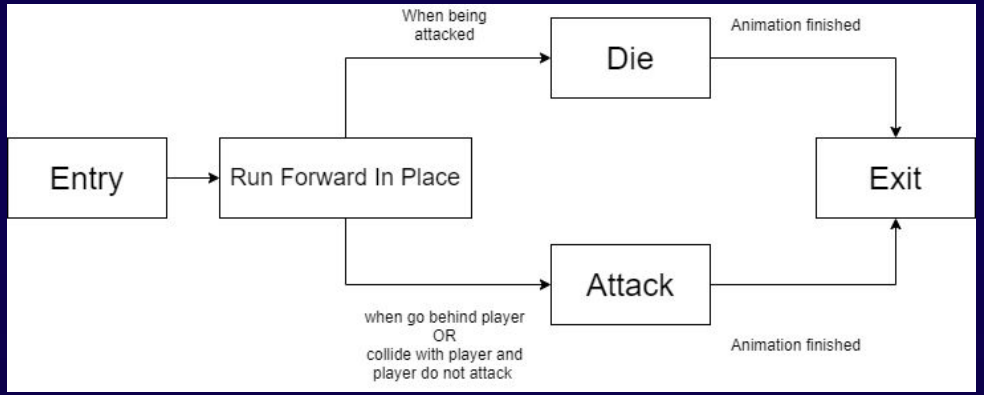
- Tile style is randomize and fixed when spawning the first tile
- Width of tiles: 50 unit
- spawnZ keep track of the position of new tiles
- Tile out of safe zone → destroyed





Implementation: Minion Animation State Diagram

- Controlled by PlayerMotor.cs and Minion.cs
- PlayerMotor.cs : play the animation based on the collision detection function
- Minion.cs : play the “attack” animation based on the distance from the player



Implementation - Game Management

Health Bar

- Misses a minion → Health bar decreased
- Health point reach 0 → death menu pop up

Score

- represent number of minions have been hit

Death Menu

- Display score
- Restart button
- Main menu button

Pause Menu

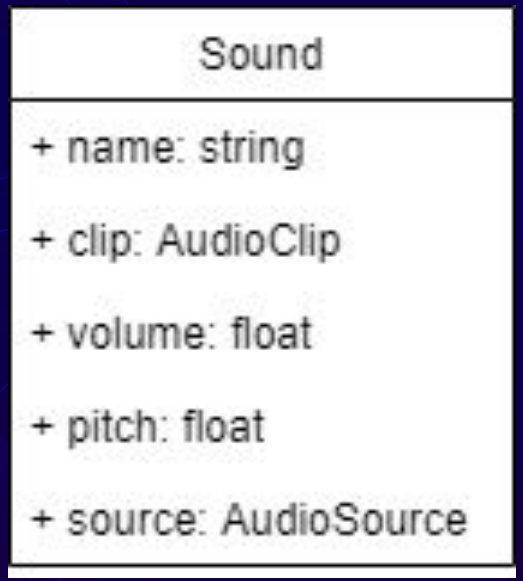
- Press “Esc” to pause the game
- Resume, Restart, Quit

Implementation - Data storage

- PlayPrefs that stores and access player preferences between game sessions
- Store the highest score of each game level
- Each equipment will have a variable to store the status(sold not or) of equipments in shop

Implementation: Sound, Sound Manager

Class: Sound



Core: Sound Manager

1. Audio processing (Beat/Onset Detection)
2. Audio Visualization (Spawning Minions)
3. Play the Song

Implementation: Audio Processing (Beat Tracking)

1. BPM = getBpm()
2. Total Number of Beats = Clip Length / 60 * BPM
3. Time interval = Clip Length / Total Number of Beats
4.
```
for (i=0;i<totalNumOfBeats;i++) {
  timestamp = interval * i;
  SpawnMinion(timestamp);
}
```

Audio Processing (Beat Tracking)

1. BPM = getBpm()

2. Total Number of Beats = Clip Length / 60 * BPM

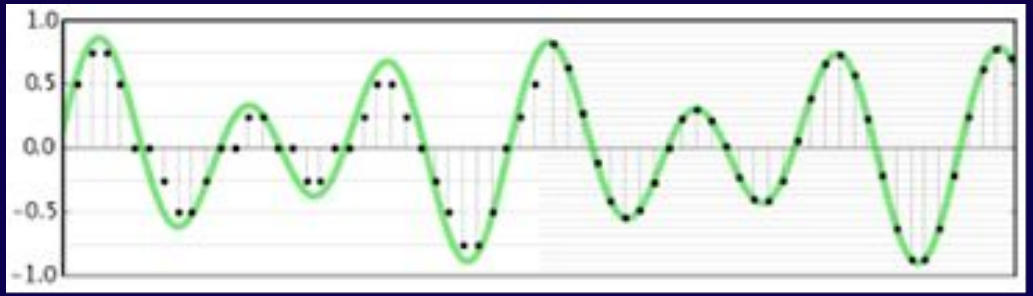
3. Time interval = Clip Length / Total Number of Beats

4.
```
for (i=0;i<totalNumOfBeats;i++) {
  timestamp = interval * i;
  SpawnMinion(timestamp);
}
```

Implementation: Audio Processing (Onset Detection)

Audio Processing (Onset Detection)

Step 1: Getting sample data




```
Pseudo Code:
   AudioClip selectedSong
   selectedSong.samples()                //Get the number of samples
   selectedSong.channels()             //Get the number of channels
   samples = float[numOfSamples * numOfChannels]
   samples = GetData(selectedSong)
```

0.1 Channel 0	0.04 Channel 1	-0.32 Channel 2	-0.01 Channel 0	0.27 Channel 1	0.71 Channel 2	...
|----------Timestamp: 0.0123s ----------|			|----------Timestamp: 0.0456s ----------|		

Step 2: Combining channels

```
Pseudo Code of GetOutputData():
  preprocessedSamples = float[numOfSamples]
  int numProcessed = 0
  float combinedChannelAvg = 0
  for (int i=0; i < numOfAllSamples; i++)
     combinedChannelAvg += allSamples[i]
     if ((i+1) % numOfChannels == 0)                //next sample is in the 0th channel
          //Average out the cumulated amplitude
          preprocessedSamples[numProcessed++] = combinedChannelAvg / numOfChannels;
          combinedChannelAvg = 0f
  return preprocessedSamples
```

0.1 Channel 0	0.04 Channel 1	-0.32 Channel 2	-0.01 Channel 0	0.27 Channel 1	0.71 Channel 2	...
|----------Timestamp: 0.0123s ----------|			|----------Timestamp: 0.0456s ----------|		

-0.06	0.323	...
|-------Timestamp: 0.0123s---------|	|-------Timestamp: 0.0456s----------|	

Step 3: Spectral analysis with fast fourier transform

1. A song is a complex waveform and can be represented by series of sines

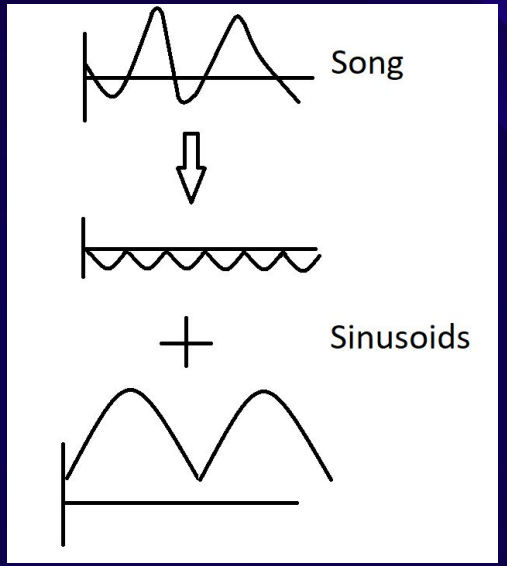
2. Fourier transform deconstructs a time domain representation of a signal into the frequency domain representation to analyze the different frequencies in a signal → deconstructing a signal

3. Discrete fourier transform (DFT, $O(N^2)$) vs Fast fourier transform (FFT, O(Nlog(N)))

4. FFT Size(N): 2^p (512/1024/2048/…) → Larger FFT size, higher spectral resolution but longer computation time

5. Limitation of FFT: Non-integer cycles → Spectral leakage

6. Solution: windowing function(Hann Window (95% cases))



Step 3: Spectral analysis with fast fourier transform

1. DSPLib - FFT / DFT library
2. Provide Hanning window, Hamming window and Blackman-harris window
3. Configurations:
   a. FFT Size: 1024
   b. Window: Hanning

```
Pseudo Code of GetSpectrumData():
  For every 1024 of samples
    Apply Hanning Window
    Execute FFT
    Pass the spectrum of the current time to SpectralFluxAnalyzer.analyzeSpectrum()
```

Step 4: Peak detection

1. SpectralFluxAnalyzer library analyzes spectral information
2. Store previous 1024 samples spectrum and next 1024 samples spectrum
3. Calculate and rectify spectral flux values

```
for (int i=0; i<1024; i++) {
   sum += Mathf.max(0f, curSpectrum[i] - prevSpectrum[i]);
}
```

4. For every 50 SpectralFlux, calculate threshold (avg. of SpectralFlux * 1.5)
5. prunedSpectralFlux = Mathf.max(0f, spectralFluxSamples[i].spectralFlux - spectralFluxSamples[i].threshold)
6. SpectralFlux[i].prunedSpectralFlux > SpectralFlux[i + 1].prunedSpectralFlux & SpectralFlux[i].prunedSpectralFlux > SpectralFlux[i - 1].prunedSpectralFlux

Final step: Peak filtering

```
Pseudo Code:
  For each peak sample i
    if i is the first sample or the last sample
        add i to the list of peaks
    else
        add the sample with the highest spectral flux i with i-1 and i+1 to the list of peaks
```

JSON Generation

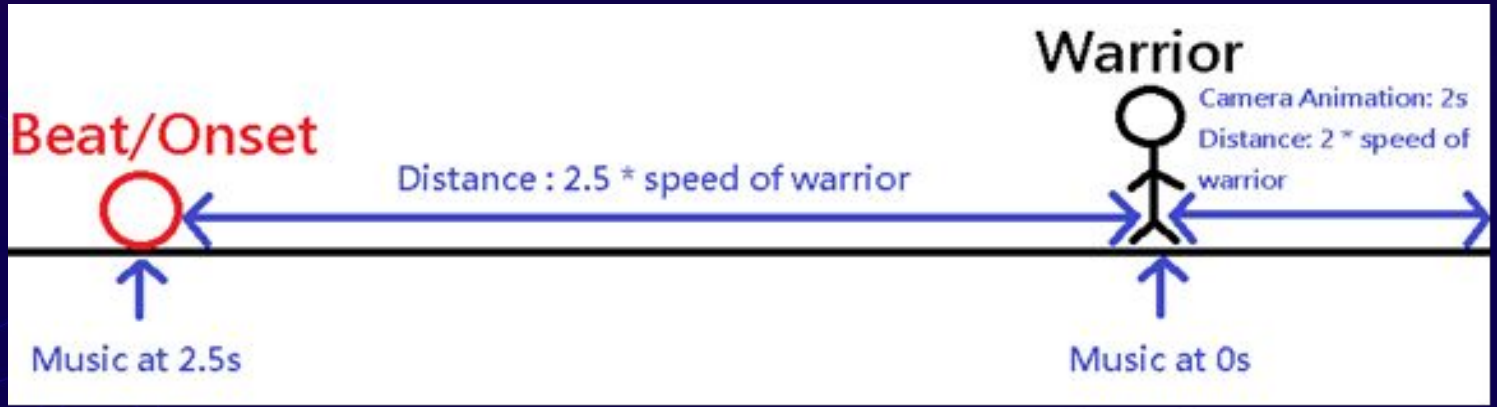
1. FFT - O(Nlog(N)), N is FFT size

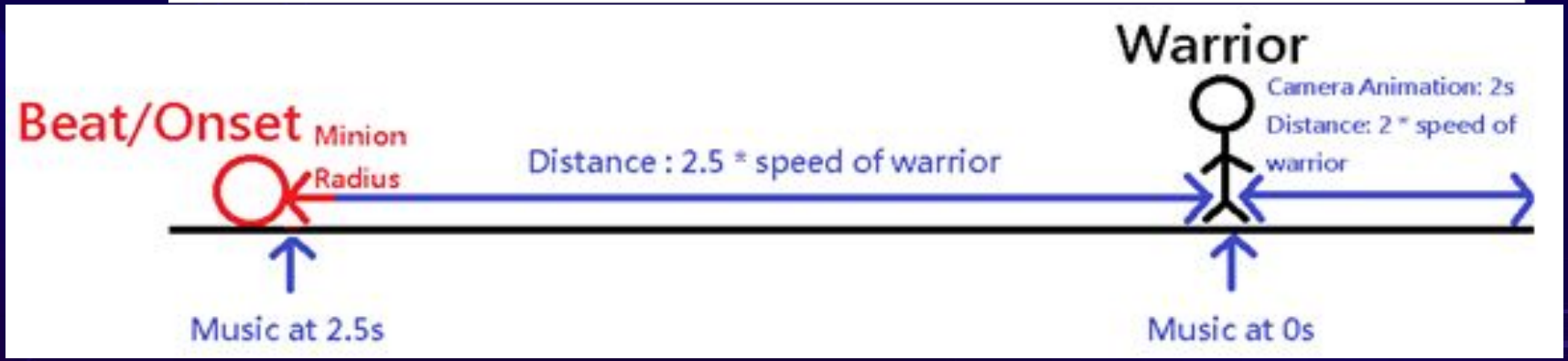
2. For example, Faded (44.1 kHz sampling rate, stereo channels, 3:32, MP3 format) has 9,373,824 normalized samples → (9,373,824/1024) * (1024log(1024)) = 28,218,022 iterations

3. Execute the algorithm on the soundtrack in Unity for once and output the extracted spectral flux information (including the timestamps) as .json formatted data and store the file in Resource/Json folder

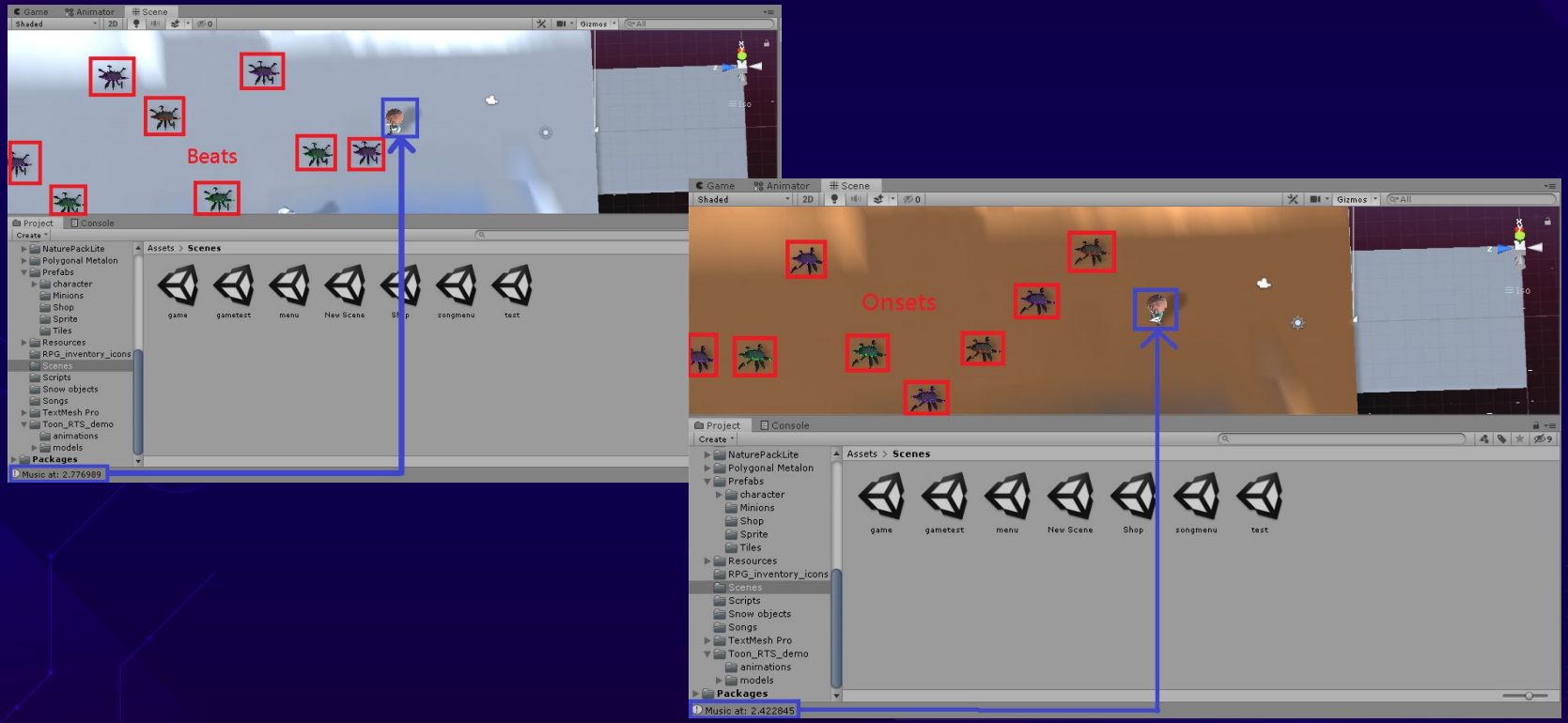
4. In onset mode, directly read the calculated timestamps from spectral flux information .json file of the selected song and print the minions according to the timestamps

Implementation: Audio Visualization

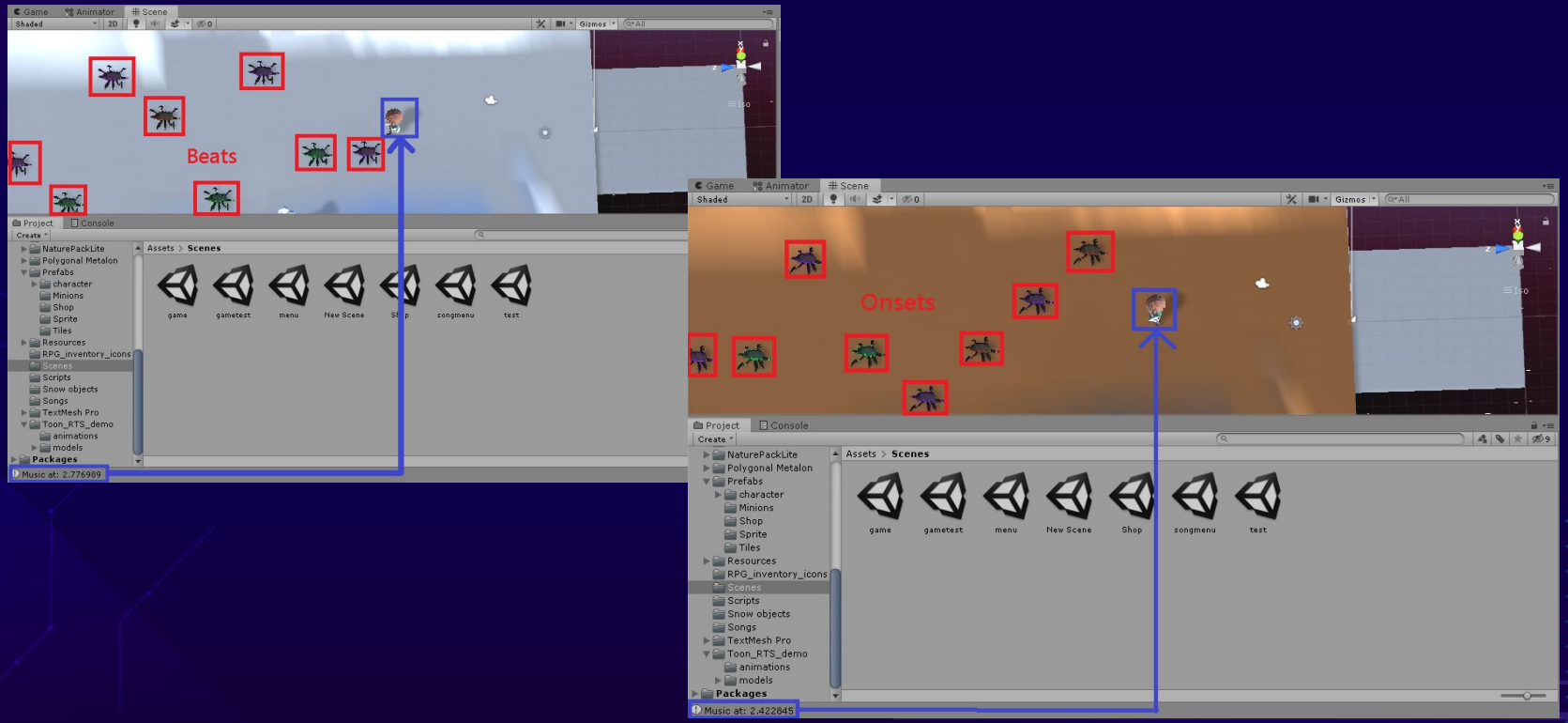




Implementation: Audio Visualization



Audio Visualization




TESTING

Testing

System Testing

- Gameplay
- UI elements

Logging

- timestamps
- scores
- character position

Subjective Testing

- user friendliness
- storytelling
- smooth gameplay
- graphic design
- difficulty

EVALUATION

Evaluation

Recalling our objectives

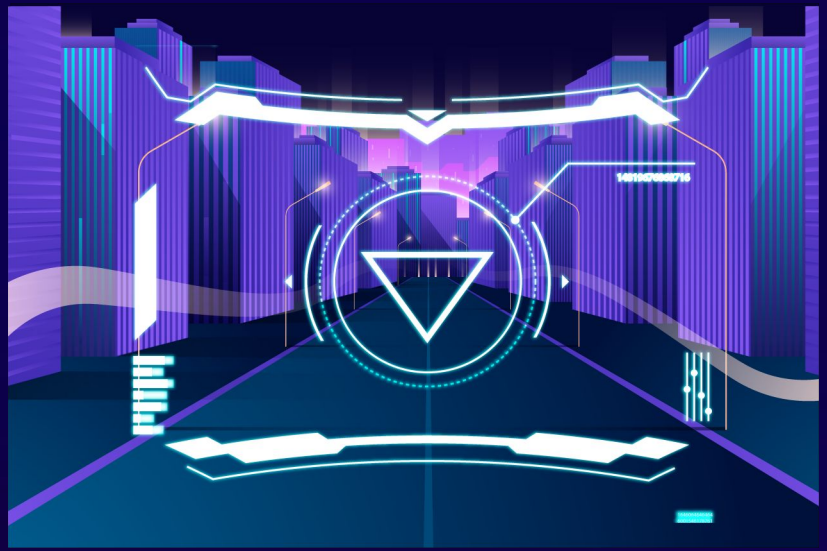
1. Combine RPG elements with music rhythm game
2. Develop a user-friendly user interface for players
3. Integrate a ranking system into the game
4. Convert MIDI files into usable format for rhythm processing

THANK YOU

Ho Wai Kin (Sam, 2044 7589)
Ho Tsz Kiu (Sarah, 20424236)
Fung Hing Lun (Victor, 20448193)

HKUST CSE FYP 2019-2020 HO2

DEMO



https://johnnyn2.github.io/MusicRhythmGameBuild/index.html

Q&A