MUSIC IN AIR – DIGITAL THEREMIN

Submitted by

Wong Wee Meng (U1923221F)

School of Computer Science and Engineering

A Final Year Project presented to the Nanyang Technological University

in partial fulfilment of the requirements for the

Degree of Bachelor of Engineering

2021/22

Contents

Contents 1

Table of Figure 3

Abstract 4

Acknowledgment 5

1. Introduction 6

  1.1 Background 6

    1.1.1 Theremin 6

    1.1.2 Leap Motion 6

  1.2 Problem 7

2. Existing Approach (Literature Review) 8

3. Research Hypothesis 9

  3.1 Purpose 9

  3.2 Scope 9

4. Research and development plan 10

  4.1 Gemini 10

  4.2 Unity 10

    4.2.1 Unity's Audio Function 11

  4.3 Csound Unity 12

    4.3.1 Csound Syntax 12

  4.3.2 Implement Csound with Unity 13

5. Design and Implementation 14

  5.1 Using Mouse to generate sound 14

    5.1.1 Mouse Input 14

    5.1.2 Frequency Control 14

    5.1.3 Amplitude Control 15

    5.1.4 Generate Sound using OnAudioFilterRead 15

  5.2 Integrating the Leap Motion 16

    5.2.1 Leap Motion Input 16

    5.2.2 Mapping different frequencies based on hand position 16

    5.2.3 Mapping amplitude based on hand position 17

  5.3 Playing different wave functions 18

  5.4 User Interface 19

  5.5 Csound Implementation 20

    5.5.1. Basic Oscillation 20

    5.5.2 Sound Font Implementation 21

    5.5.3. FM (frequency modulated) synthesis with CSound 22

    5.5.4. WG (Wave Guided) synthesis with CSound 23

5.5.5. Update User Interface 24
5.6 Customised Starting Frequency 24
6. Implementation Issues 25
6.1 Lag Issues 25
6.2 Hardware Issue 25
6.3 Heat Issue 25
7. Performance Measurement 26
7.1 Procedures for measuring performance 26
7.2 Results of experiment on measuring performance 26
7.2.1 Experiment 1 26
7.2.2 Experiment 2 26
7.2.3 Experiment 3 27
8. User Study 27
9. Conclusion 27
10. References 28

Table of Figure

Figure 1 Theremin appearance 6
Figure 2. Leap Motion Controller 6
Figure 3. Ultraleap hand tracking interaction zones 7
Figure 4. Software implemented by former students 8
Figure 5. Physics representations of hands and VR controllers with interaction in Unity 10
Figure 6. An example of a OnAudioFilterRead function from Unity documentation 11
Figure 7. An example of csd file 12
Figure 8. Calculating size of each note in playing area 14
Figure 9. Computing fixed and continuous frequency 14
Figure 10. Computing amplitude based on mouse position 15
Figure 11. Using OnAudioFilterRead() function to generate sound 15
Figure 12. A software implementation of using mouse controller input to produce sound in Unity 15
Figure 13. Computing the position of both hands in playing space 16
Figure 14. Computing Frequency based on right hand position 16
Figure 15. Computing Amplitude based on left hand position 17
Figure 16. Smoothing the transition of amplitude to prevent pop sound in the buffer. 17
Figure 17. Selecting between 6 different wave functions 18
Figure 18. Interaction between main script and user interface 19
Figure 19. Overall Software's User Interface 19
Figure 20. Interaction between main game unity script and CsoundUnity script 20
Figure 21. Using the 'Oscil' Opcode to generate sound 20
Figure 22. Passing data between LibrarySound.cs and csd file 20
Figure 23. Reading Sound Font into csd file 21
Figure 24. A csd file that generate sound similar to a percussion flute 22
Figure 25. A csd file that generate a sound similar to a clarinet 23
Figure 26. Additional User Interface to toggle between Unity default library and CSound library. 24

Abstract

Past FYP students had successfully implemented a digital theremin using the Leap Motion controller. Tools such as Microsoft Visual Studio, .NET Framework, NAudio library were used to simulate playing the Theremin. However, undesired static noises are also introduced as well. The reason of the occurrence of static noise is unknown, it could be a combination of NAudio and old drivers of Leap Motion.

Here, the same project is re-created using Unity software to determine if the static sounds persist. If there is no more audio flaw, the Csound library will then be investigated on whether it can be used to generate a more variety and beautiful sound.

Lastly, experiments is conducted to measure the latency of the sound libraries used and determine if it is suitable to simulate playing the theremin.

Acknowledgment

I would like to express my gratitude and appreciation to my project supervisor, Associate Professor Alexei Sourin for his encouragement and guidance throughout the project.

1. Introduction

1.1 Background

This section discusses the background information of the Theremin, and the importance of simulating the Theremin with the Leap Motion.

1.1.1 Theremin

The Theremin is a unique musical instrument that is played without physical contact. Moving one's hands near a vertical antenna causes the theremin to produce a different pitch, and a separate horizontal antenna controls the theremin's volume. However, precise control involves the feedback of hearing the theremin's tone and moving one's hands in accordance. Experienced players can roughly estimate the positions needed to produce various notes [1].

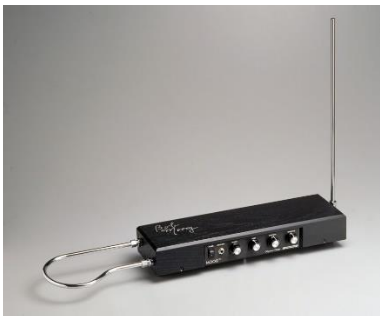


Figure 1 Theremin appearance

1.1.2 Leap Motion

The Leap Motion Controller is an optical hand tracking module that captures hand movements with exceptional accuracy. The Leap Motion Controller features an interaction zone that spans from the device in a 140x120° typical field of vision and extends from 10cm to 60cm or more [2] .

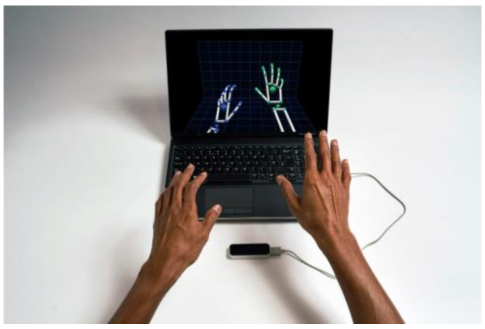


Figure 2. Leap Motion Controller

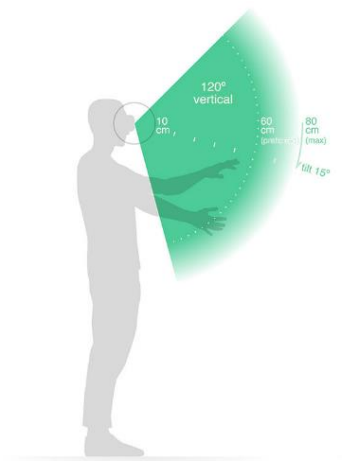


Figure 3. Ultraleap hand tracking interaction zones

1.2 Problem

The three main negative factors affecting usage of Theremins are affordability, portability, and user experience.

Theremins typically cost between $200 and $899 [3], which is quite pricey when compared to the Leap Motion's pricing of about $80 [4]. Furthermore, the Theremin is larger in shape compared to a small Leap motion controller. It becomes less portable because of this.

Lastly, playing the Theremin may results a poor user experience because it is difficult to play, and most users would give up after a while. The user would not be able to distinguish the note that is sounded without having perfect pitch, which is rare ability to recognize specific musical note without the need of a reference tone. However, with the use of Leap Motion, a simple software can be developed that displays the note being played by the user. This may improve user experience as the user knows the note which he is currently playing.

Therefore, it is important to simulate the Theremin digitally with a Leap motion as it would benefit the users by providing an affordable and portable option, as well as improving the user experience.

2. Existing Approach (Literature Review)

Theremin simulation software had been implemented by former students using .NET Framework using C# programming language [5]. Depending on the user's hand location in relation to the Leap Motion controller, the software can generate different frequencies using the NAudio sound libraries. For various audio effects, users can also choose from several wave functions like sine, triangle, and sawtooth. Additionally, there is a simple user interface that shows an image of a piano keyboard for reference and the note that is currently being played.

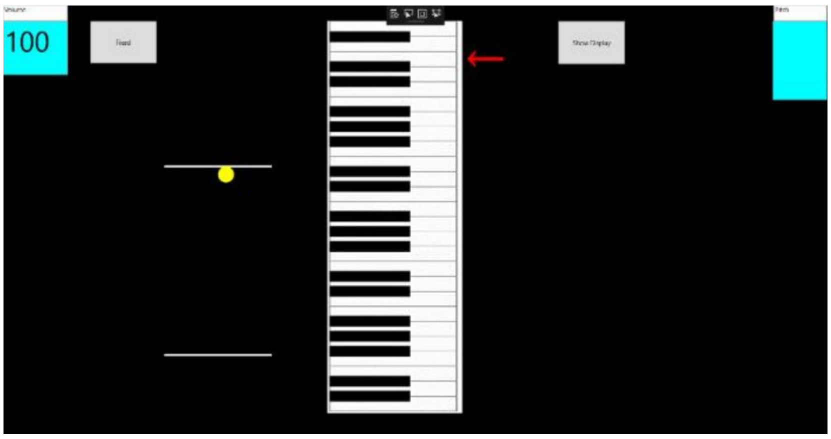


Figure 4. Software implemented by former students

However, there are some problems with the software. Static noise and popping sounds are produced on around every 2 seconds while playing the software which results to a bad user experience. The origin of the static noise is not known, and it could be due to the NAudio sound library used or the old drivers of the leap motion.

Thus, there is a need to explore an alternative way to simulate playing the Theremin and create high quality sound.

3. Research Hypothesis

3.1 Purpose

The purpose of this project is to explore how well Unity Software can simulate playing the Theremin and to create a performance workbench exhibiting the best velocity achievable without any audio flaws. Then, we investigate if CSound can be utilized to enhance the sound quality and variety at an expense of latency.

3.2 Scope

In this project, instead of developing a program that generates sound from scratch, we will utilise existing sound library to ease the generation of sound. Should the popping sound persist even when using a new sound library, we will proceed to explore other hardware alternative like Kinect as we would not be investigating the hardware issue of the Leap Motion Controller.

4. Research and development plan

This section discussed the necessary research and knowledge required for the development of the software.

4.1 Gemini

There is a recent released of Gemini, the fifth version of UltraLeap hand tracking software. With a newer version, there is an improved in two-handed interaction, initialization and allows more robust and accurate hand tracking [6]. This would also allow a more stable and reliable hardware to work with.

4.2 Unity

In this project, the Unity software is being explored. Unity is a cross-platform game engine that support a variety of desktop, mobile, console and virtual reality platforms. The ultra-leap plugin for unity provides the tools and utilities to connect Unity applications to hand tracking data makes it possible to use with the leap motion.

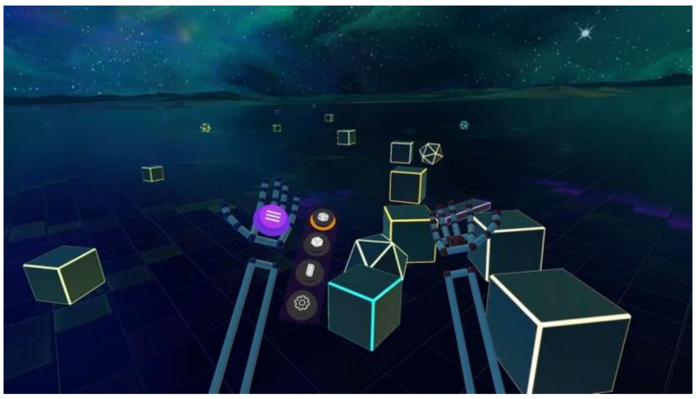
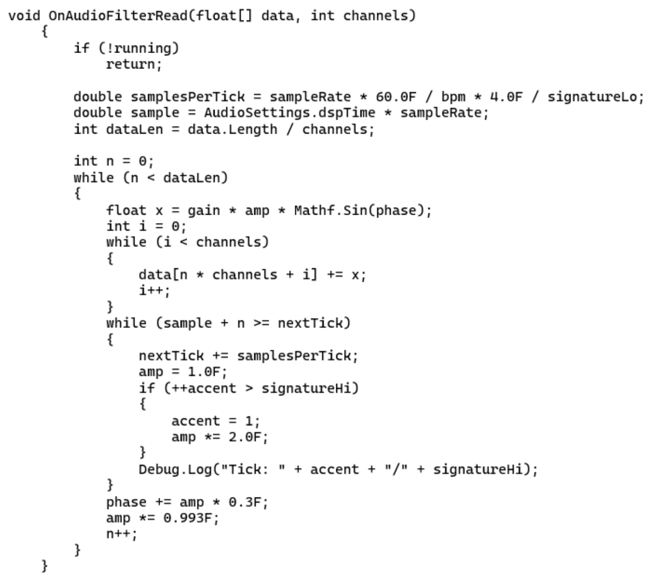


Figure 5. Physics representations of hands and VR controllers with interaction in Unity

Before implementing leap motion in unity, the ultra-leap hand tracking software must be installed first, followed by the unity modules package which includes Core, Interaction Engine, and the Hands Module.

4.2.1 Unity's Audio Function

Unity has a main default audio function which is known as OnAudioFilterRead. Every time a chunk of audio is supplied to the filter, the OnAudioFilterRead function is invoked. The audio data is a float array ranging from -1 to 1 that contains audio from the preceding filter in the chain or the Audio Clip on the Audio Source [7]. This function is necessary for sound generation.



```
void OnAudioFilterRead(float[] data, int channels)
    {
        if (!running)
            return;

        double samplesPerTick = sampleRate * 60.0F / bpm * 4.0F / signatureLo;
        double sample = AudioSettings.dspTime * sampleRate;
        int dataLen = data.Length / channels;

        int n = 0;
        while (n < dataLen)
        {
            float x = gain * amp * Mathf.Sin(phase);
            int i = 0;
            while (i < channels)
            {
                data[n * channels + i] += x;
                i++;
            }
            while (sample + n >= nextTick)
            {
                nextTick += samplesPerTick;
                amp = 1.0F;
                if (++accent > signatureHi)
                {
                    accent = 1;
                    amp *= 2.0F;
                }
                Debug.Log("Tick: " + accent + "/" + signatureHi);
            }
            phase += amp * 0.3F;
            amp *= 0.993F;
            n++;
        }
    }
```

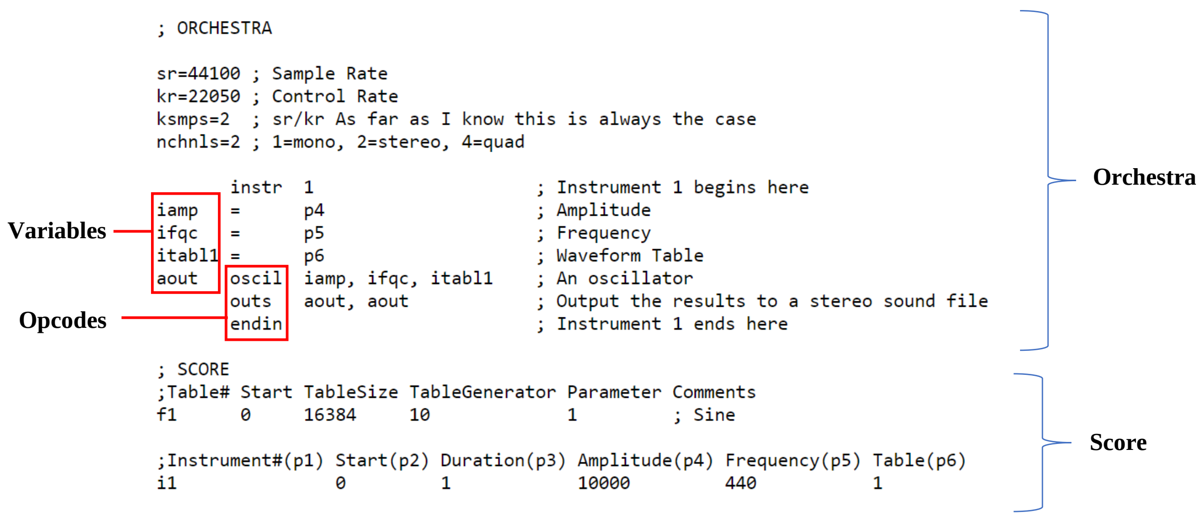
Figure 6. An example of a OnAudioFilterRead function from Unity documentation

4.3 Csound Unity

Csound is a programming language (based on C) that was created and optimized for sound rendering and signal processing. Currently, there is a custom Csound package for Unity known as CsoundUnity. It provides Unity users to access the Csound's core API within their Unity C# scripts and provides a Unity bridge to a set of pInvoke signatures that are used to call Csound's native API functions from C# code [8]. Once the CsoundUnity packages is downloaded, it needs to be imported directly into Unity.

4.3.1 Csound Syntax

Csound consists of 2 different files, an orchestra file and a score file. The figure below is an example of a Csound .csd script.



```
; ORCHESTRA

sr=44100 ; Sample Rate
kr=22050 ; Control Rate
ksmps=2  ; sr/kr As far as I know this is always the case
nchnls=2 ; 1=mono, 2=stereo, 4=quad

       instr  1                      ; Instrument 1 begins here
iamp   =      p4                     ; Amplitude
ifqc   =      p5                     ; Frequency
itabl1 =      p6                     ; Waveform Table
aout   oscil  iamp, ifqc, itabl1     ; An oscillator
       outs   aout, aout             ; Output the results to a stereo sound file
       endin                         ; Instrument 1 ends here

; SCORE
;Table# Start TableSize TableGenerator Parameter Comments
f1      0     16384     10             1         ; Sine

;Instrument#(p1) Start(p2) Duration(p3) Amplitude(p4) Frequency(p5) Table(p6)
i1               0         1            10000         440           1
```

Figure 7. An example of csd file

The Orchestra is defined above the score. The orchestra section starts with a header and defined the list of instruments below, and the score section contains tables events and instrument events. Anything appear at the right side of a semicolons are comments.

There are three types of variables, and they usually start with letter “a”, “k” and “i”. The letter ‘a’ means audio rate and “k” means control rate. They can be modified during runtime, except for variables that start with “i” which are initialized to a value at the start and do no change.

There are many opcodes available in Csound. Every opcode may have variables on the left or on the right of the opcode. In the figure above, the opcode “oscil” takes in 3 variables, ‘iamp’, ‘ifqc’ and ‘itabl1’, and output the variable ‘aout’.

4.3.2 Implement Csound with Unity

To implement Csound with Unity, two important scripts are commonly used.

First is CsoundUnity.cs which is the main Csound script which is imported to the project. It creates an instance of CsoundUnityBridge for accessing most Csound's methods. CsoundUnity focus on common methods such as compiling and channel communication and does not expose all of Csound's native API functions. This script is attached to a GameObject and should be instantiated when a game first starts.

Second is the csd file which had been discussed above. This file provides some simple instrument definition and contains the orchestral and score information. Each CsoundUnity.cs will take in one csd file and the variables inside the csd file can be accessed mainly using SendScoreEvent() or the setChannel() method.

5. Design and Implementation

5.1 Using Mouse to generate sound

To begin with, we implemented a sound generation feature using a simple mouse input. A simple scene is set up with minimum interface.

5.1.1 Mouse Input

The position of the mouse will determine the amplitude and frequencies being played. Moving in the horizontal direction will controls the amplitude while moving in the vertical direction will controls the frequency of the sound. Input.mousePosition is called to retrieve the position of the Mouse.

5.1.2 Frequency Control

The next step is to map the mouse position to play different amplitude and frequency of sound. Firstly, we compute the total notes by multiplying total octave by 12. Example, if there are 3 octaves, then total notes would be 36. The size of each note determines the location of the note in the 3D playing area. It can be computed by dividing the height of the screen by the total notes.

```
void CalculateTotalNote()
{
    int totalNotes = numOfOctave * 12;
    sizeOfNote = Screen.height / totalNotes;
}
```

Figure 8. Calculating size of each note in playing area

Next, we can compute the frequency by obtaining the note index. The note index determine what note is being played. For a 3-octave range, the note index will be between 0 to 36, example from A3 to G6 We then compute the power raised by dividing the note Index by 12, to get a value between 0 to 3. Assume the starting frequency is 220, a note Index of 0, 1, 2, 3 would result in a frequency of 220, 440, 880 and 1760 respectively. For a non-continuously frequency function, the noteIndex will be round down to an integer.

```
void UnFixedFrequency()
{
    mousePos = Input.mousePosition;
    noteIndex = mousePos.y/ sizeOfNote; //0 to 36
    powerRaised = noteIndex / 12; //0 to 3
    frequency = 220 * Mathf.Pow(2, powerRaised);
    noteKey = (int)noteIndex % 12;
    DisplayNote(noteKey);
}

void FixedFrequency()
{
    mousePos = Input.mousePosition;
    noteIndex = Mathf.RoundToInt(mousePos.y/sizeOfNote);  //0 to 36 but no decimal
    fixedPowerRaised = noteIndex / 12;
    frequency = 220 * Mathf.Pow(2, fixedPowerRaised);
    noteKey = (int)noteIndex % 12;
    DisplayNote(noteKey);
}
```

Figure 9. Computing fixed and continuous frequency

5.1.3 Amplitude Control

The amplitude of the sound is determined by the x position of the mouse in respective to the screen width.

```
void UpdateVolume()
    {
        mousePos = Input.mousePosition;
        amplitude = (mousePos.x / Screen.width) * (maxAmp-0.5) +0.5;
    }
```

Figure 10. Computing amplitude based on mouse position

5.1.4 Generate Sound using OnAudioFilterRead

With the computed amplitude and frequency, we can pass the values to the OnAudioFilterRead () function to generate the sine wave.

```
void OnAudioFilterRead(float[] data, int channels)

    {
       if (playSound)
        {
            increment = frequency * 2 * Math.PI / sampling_frequency;
            for (var i = 0; i < data.Length; i += channels)
            {
                phase += increment;
                data[i] = (float)(amplitude * Math.Sin(phase));
                if (channels == 2)
                {
                    data[i + 1] = data[i];
                }

                if (phase >= 2 * Math.PI)
                {
                    phase -= 2*Math.PI;
                }
            }
        }
    }
```

Figure 11. Using OnAudioFilterRead() function to generate sound

Since this feature created a smooth generation of sound, the next step is to substitute the mouse input with a Leap Motion Controller Input.

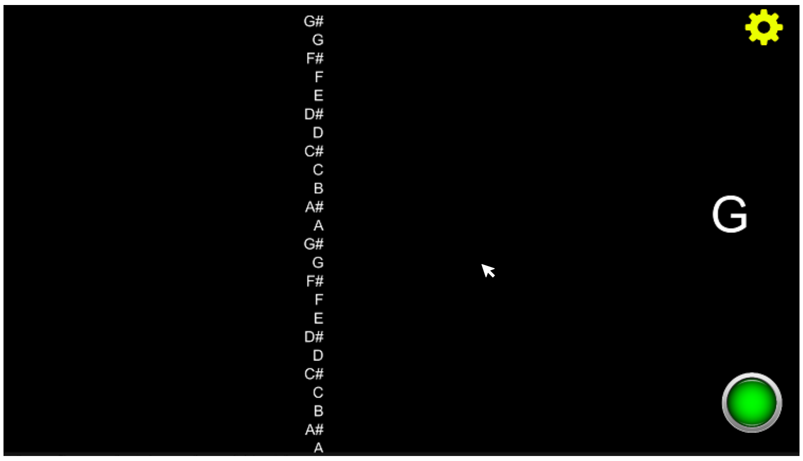


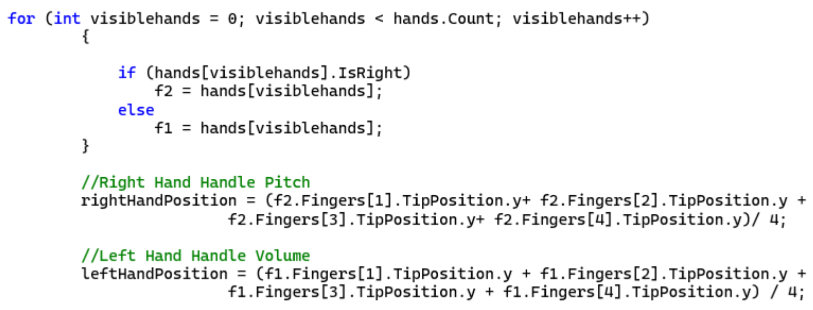
Figure 12. A software implementation of using mouse controller input to produce sound in Unity

5.2 Integrating the Leap Motion

5.2.1 Leap Motion Input

Accessing the Leap Motion library is possible after importing the Leap package into Unity. Once imported, the 3D hand model can be view in the Game View. Simulating playing the Theremin with 2 hands would means that we must compute the position of both hands. We must first deduce the position of the hands in the 3D leap motion space. Since the right hand controls the pitch, the average y axis position of all the fingers is computed. The position of the y position in the 3D leap motion space will determine the frequency being produced. The thumb is excluded since the mobility of the thumb is limited and would not greatly affect the average position.

Similarly for the left hand, the average y axis position of all the fingers is computed, and it will determine the amplitude of the sound.

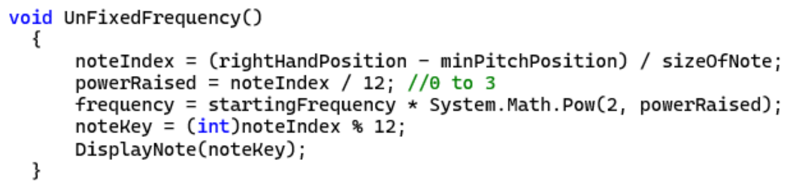


```
for (int visiblehands = 0; visiblehands < hands.Count; visiblehands++)
        {

            if (hands[visiblehands].IsRight)
                f2 = hands[visiblehands];
            else
                f1 = hands[visiblehands];
        }

        //Right Hand Handle Pitch
        rightHandPosition = (f2.Fingers[1].TipPosition.y+ f2.Fingers[2].TipPosition.y +
                        f2.Fingers[3].TipPosition.y+ f2.Fingers[4].TipPosition.y)/ 4;

        //Left Hand Handle Volume
        leftHandPosition = (f1.Fingers[1].TipPosition.y + f1.Fingers[2].TipPosition.y +
                        f1.Fingers[3].TipPosition.y + f1.Fingers[4].TipPosition.y) / 4;
```

Figure 13. Computing the position of both hands in playing space

5.2.2 Mapping different frequencies based on hand position

The boundary of the playing space is set by initialising the minimum and maximum y position of the leap motion space. We then map the available range of the 3D space to the range of frequency. For example, we set the minimum y position to be 185 and maximum y position to be 293. If 3 octaves are selected, then the hand position at 185 would be note A2 and the hand position at 293 would be note G#5. Any position lies in the range will be from A2-G#7.



```
void UnFixedFrequency()
  {
      noteIndex = (rightHandPosition - minPitchPosition) / sizeOfNote;
      powerRaised = noteIndex / 12; //0 to 3
      frequency = startingFrequency * System.Math.Pow(2, powerRaised);
      noteKey = (int)noteIndex % 12;
      DisplayNote(noteKey);
  }
```

Figure 14. Computing Frequency based on right hand position

5.2.3 Mapping amplitude based on hand position

Instead of using mousePosition.x, we use the left-hand position in space to calculate the amplitude.

```
void UpdateVolume()
    {
        amplitude = ((leftHandPosition - minVolumePosition) / leftHandPositionRange) * (maxAmp) ;
        amplitude = Mathf.Clamp((float)amplitude, 0, (float)maxAmp);
    }
```

Figure 15. Computing Amplitude based on left hand position

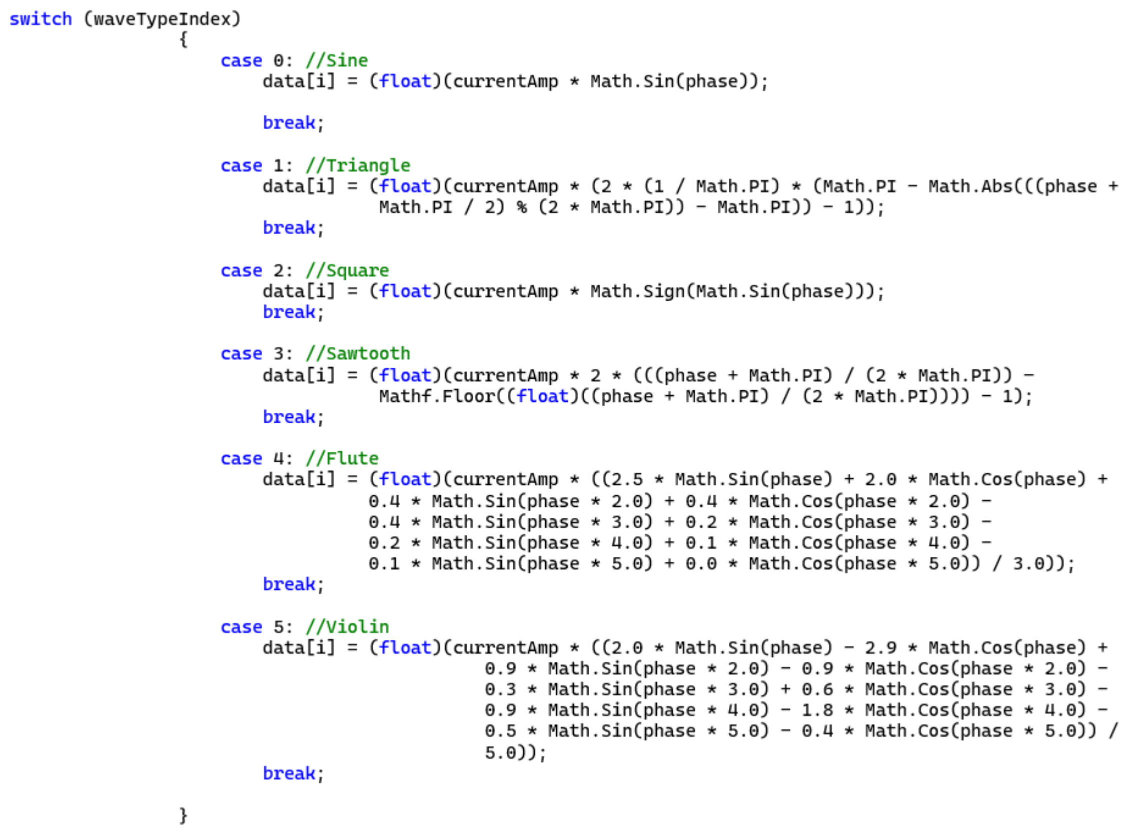
There is a slight problem when generating sound. When the user removes their left hand in the play area, the sudden change of the amplitude value to zero will cause a pop sound. To tackle this issue, a lerp is added for a smooth transition of the amplitude value to zero.

```
void UpdateCurrentAmp()
    {
        if (amplitude <= 0)
        {
            if (currentAmp >= amplitude)
            {
                currentAmp -= Time.deltaTime * speedOfAmpChange;
            }
        }
        else
        {
            if (Math.Abs(currentAmp - amplitude) > 0.0f)
            {
                if (currentAmp < amplitude)
                    currentAmp += Time.deltaTime * speedOfAmpChange;
                else if (currentAmp > amplitude)
                    currentAmp -= Time.deltaTime * speedOfAmpChange;
            }
        }
    }
```

Figure 16. Smoothing the transition of amplitude to prevent pop sound in the buffer.

5.3 Playing different wave functions

Six different wave formulas are implemented in this software. User can select different wave options from the dropdown menu in the game scene to be played by the OnAudioFilterRead() function.



```
switch (waveTypeIndex)
                {
                    case 0: //Sine
                        data[i] = (float)(currentAmp * Math.Sin(phase));

                        break;

                    case 1: //Triangle
                        data[i] = (float)(currentAmp * (2 * (1 / Math.PI) * (Math.PI - Math.Abs(((phase +
                                   Math.PI / 2) % (2 * Math.PI)) - Math.PI)) - 1));
                        break;

                    case 2: //Square
                        data[i] = (float)(currentAmp * Math.Sign(Math.Sin(phase)));
                        break;

                    case 3: //Sawtooth
                        data[i] = (float)(currentAmp * 2 * (((phase + Math.PI) / (2 * Math.PI)) -
                                   Mathf.Floor((float)((phase + Math.PI) / (2 * Math.PI)))) - 1);
                        break;

                    case 4: //Flute
                        data[i] = (float)(currentAmp * ((2.5 * Math.Sin(phase) + 2.0 * Math.Cos(phase) +
                                  0.4 * Math.Sin(phase * 2.0) + 0.4 * Math.Cos(phase * 2.0) -
                                  0.4 * Math.Sin(phase * 3.0) + 0.2 * Math.Cos(phase * 3.0) -
                                  0.2 * Math.Sin(phase * 4.0) + 0.1 * Math.Cos(phase * 4.0) -
                                  0.1 * Math.Sin(phase * 5.0) + 0.0 * Math.Cos(phase * 5.0)) / 3.0));
                        break;

                    case 5: //Violin
                        data[i] = (float)(currentAmp * ((2.0 * Math.Sin(phase) - 2.9 * Math.Cos(phase) +
                                              0.9 * Math.Sin(phase * 2.0) - 0.9 * Math.Cos(phase * 2.0) -
                                              0.3 * Math.Sin(phase * 3.0) + 0.6 * Math.Cos(phase * 3.0) -
                                              0.9 * Math.Sin(phase * 4.0) - 1.8 * Math.Cos(phase * 4.0) -
                                              0.5 * Math.Sin(phase * 5.0) - 0.4 * Math.Cos(phase * 5.0)) /
                                              5.0));
                        break;

                }
```

Figure 17. Selecting between 6 different wave functions

5.4 User Interface

Up till now, all the features are inside the UnitySoundLeap.cs script which manage the leap motion input, computing frequencies and amplitude, and output the sound. Therefore, to avoid increasing the complexity of this script, another script called UIHand.cs is created. The aim of this script is to manage the user input and update the screen accordingly and passing new data to the UnitySoundLeap.cs.

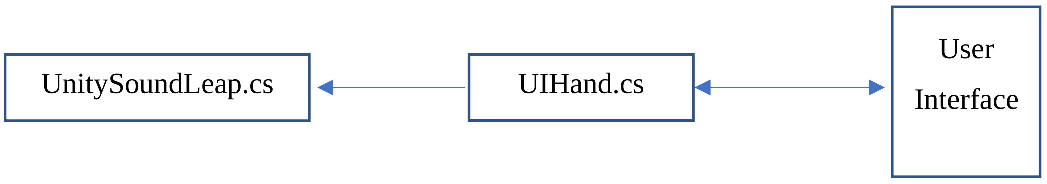


Figure 18. Interaction between main script and user interface

The user interface or this software is shown below. Users can:

1. Select a range of octave from 1 to 3.
2. Toggle between fixed and continuous frequency.
3. Toggle the display of keyboard guide and alphabet notes guide.
4. Select from a variety of wave function.
5. Close and open setting menu.

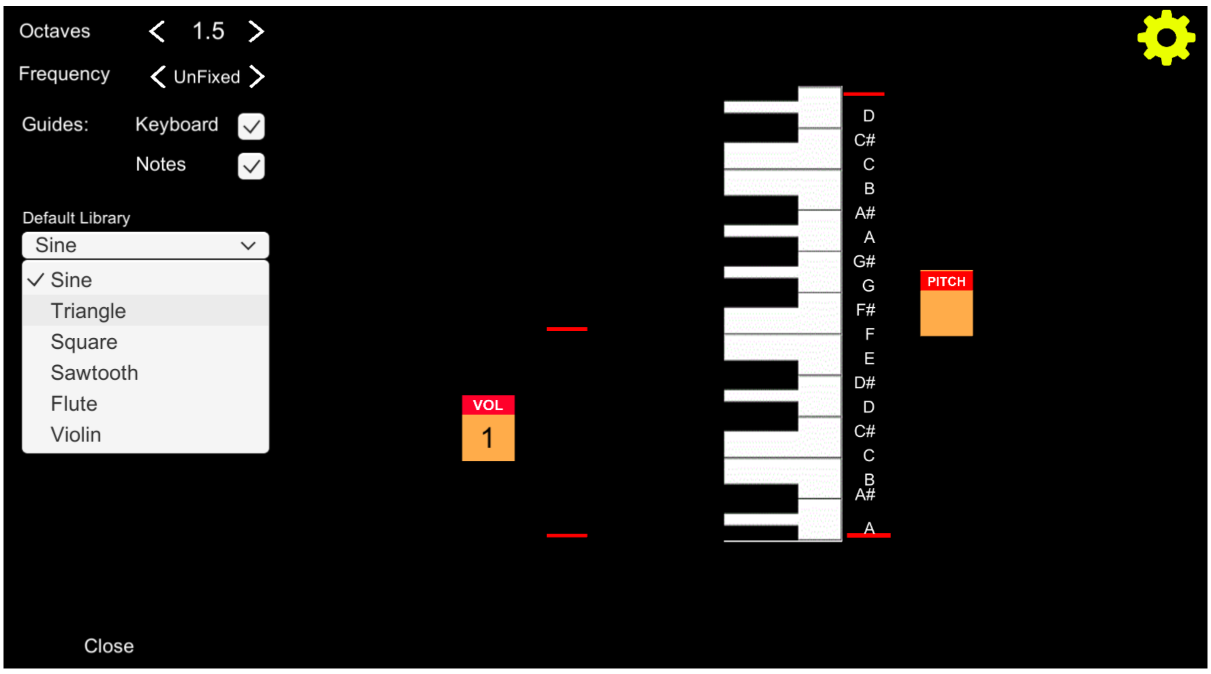


Figure 19. Overall Software's User Interface

5.5 Csound Implementation

The Csound library is explored to have access to a much more variety of instrumental sound. Instead of using the OnAudioFilterRead () function to create sound, the csd file is responsible for creating sound. A LibrarySound.cs script is created to manage everything that is related to using Csound library. The main script UnitySoundLeap.cs will pass the frequency and amplitude data to the LibrarySound.cs during runtime for updating the csd file from CSoundUnity.cs. The data flow is shown below.

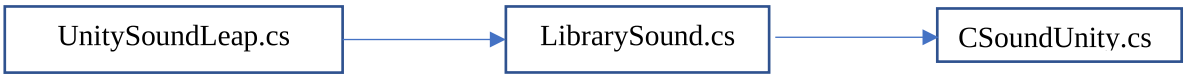
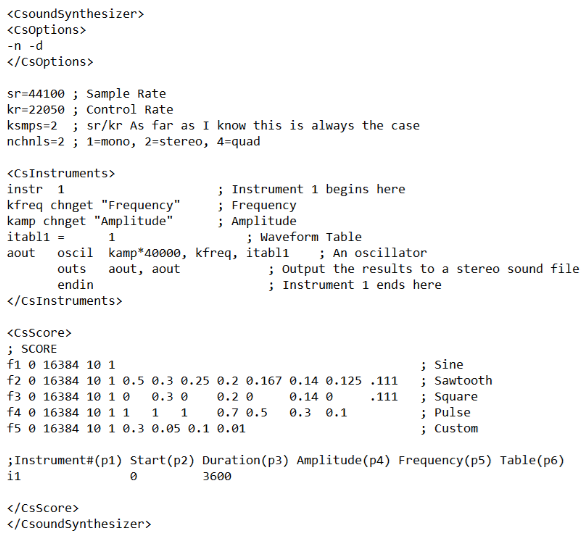


Figure 20. Interaction between main game unity script and CsoundUnity script

5.5.1. Basic Oscillation

A basic instrument that output a sine wave function is defined in the .csd file shown in figure below.



```
<CsoundSynthesizer>
<CsOptions>
-n -d
</CsOptions>

sr=44100 ; Sample Rate
kr=22050 ; Control Rate
ksmps=2  ; sr/kr As far as I know this is always the case
nchnls=2 ; 1=mono, 2=stereo, 4=quad

<CsInstruments>
instr  1                    ; Instrument 1 begins here
kfreq chnget "Frequency"    ; Frequency
kamp chnget "Amplitude"     ; Amplitude
itabl1 =      1                 ; Waveform Table
aout   oscil  kamp*40000, kfreq, itabl1    ; An oscillator
       outs   aout, aout           ; Output the results to a stereo sound file
       endin                       ; Instrument 1 ends here
</CsInstruments>

<CsScore>
; SCORE
f1 0 16384 10 1                                          ; Sine
f2 0 16384 10 1 0.5 0.3 0.25 0.2 0.167 0.14 0.125 .111   ; Sawtooth
f3 0 16384 10 1 0   0.3 0    0.2 0     0.14 0     .111   ; Square
f4 0 16384 10 1 1   1   1    0.7 0.5   0.3  0.1          ; Pulse
f5 0 16384 10 1 0.3 0.05 0.1 0.01                        ; Custom

;Instrument#(p1) Start(p2) Duration(p3) Amplitude(p4) Frequency(p5) Table(p6)
i1               0         3600

</CsScore>
</CsoundSynthesizer>
```

Figure 21. Using the 'Oscil' Opcode to generate sound

The variables kfreq and kamp controls the frequency and amplitude of the sound. Since the values will change during runtime, the LibrarySound.cs must be able to access the two variables in the csd file. This can be done using the SetChannel() method in Unity script and chnget in csd file. Below is an example on how to pass data from LibrarySound.cs to the csd file.

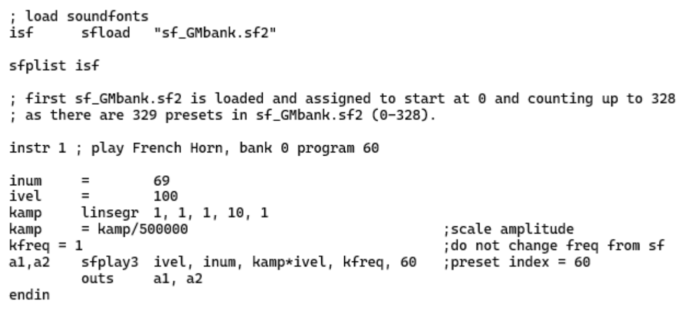
LibarySound.cs file	csd file

`csoundUnity.SetChannel("Frequency", frequency);`
`csoundUnity.SetChannel("Amplitude", currentAmp);`
	`kfreq chnget “Frequency”`
	`kamp chnget “Amplitude”`

Figure 22. Passing data between LibrarySound.cs and csd file

5.5.2 Sound Font Implementation

To have access to different instruments, the sf_GMbank.sf2 is being explored. Sf_GMbank.sf2 is a sound font that contains 329 presets instrument. There is a sample scene in the CsoundUnity package that illustrate playing the French horn instrument using the 60th preset index. The ‘inum’ variable in the csd file determines the midi number. As shown below, the variable ‘inum’ is initialised to 69, therefore playing the note “A” with 440Hz.



```
; load soundfonts
isf      sfload   "sf_GMbank.sf2"

sfplist isf

; first sf_GMbank.sf2 is loaded and assigned to start at 0 and counting up to 328
; as there are 329 presets in sf_GMbank.sf2 (0-328).

instr 1 ; play French Horn, bank 0 program 60

inum     =        69
ivel     =        100
kamp     linsegr  1, 1, 1, 10, 1
kamp     = kamp/500000                          ;scale amplitude
kfreq = 1                                       ;do not change freq from sf
a1,a2    sfplay3  ivel, inum, kamp*ivel, kfreq, 60   ;preset index = 60
         outs     a1, a2
endin
```

Figure 23. Reading Sound Font into csd file

The sound produced is beautiful however there is a limitation. In this project, the frequency of sound produced must be continuous. This csd file is unable to play all the frequency between inum 69 to 70, (A to A#). Thus, other approach is explored.

Instead of changing the inum, we can change the frequency instead. However, the variable frequency in this csd file is slightly different. It does not take in the hertz value. Instead, it is set as preset of 1. Therefore a inum of 69 and frequency of 1 means 440Hz, and a frequency of 2 would means 880Hz.

With this new approach, we can play continuous frequency, however the produced sound remains strange. Additionally, the author of this csd sample had commented in the script that we should not change the frequency parameter, and it should remain as 1. The only parmeter that can be changed is the inum variable which determine the midi number. Therefore, sound font is mainly for playing fixed frequency. It is more suitable for games that simulate playing a piano with each note already preset to different inum value.

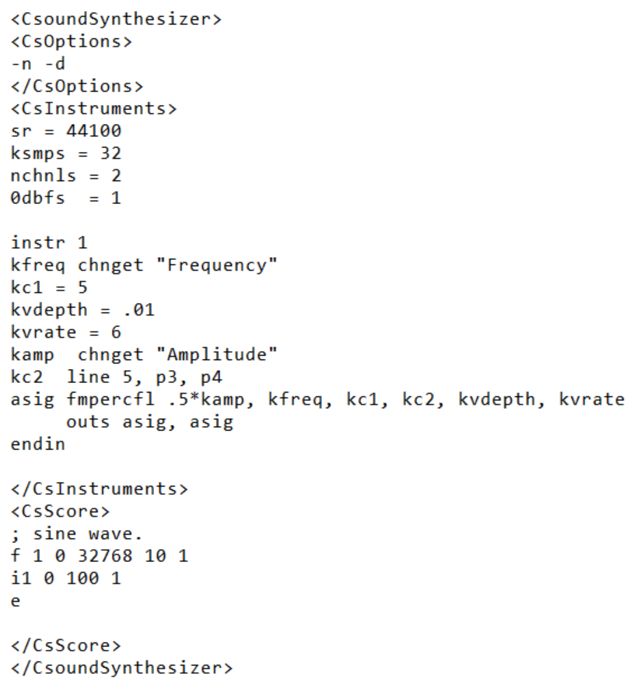
5.5.3. FM (frequency modulated) synthesis with CSound

Since using sound font is not possible, the FM opcode is being explored. In frequency modulated synthesis, the instrument plays on a continuous frequency and is being changed overtime. One example of such opcode is fmpercfl, which uses FM synthesis to create a percussive flute sound. The syntax for this opcode is

```
ares fmpercfl kamp, kfreq, kc1, kc2, kvdepth, kvrate[, ifn1, ifn2, \ ifn3, ifn4, ivfn]
```

The kamp and kfreq represent the amplitude and frequency which we can be accessed from LibrarySound.cs. The remaining parameters can be adjusted to affect instrument sound such as vibrato depth and vibrator rate.

We had implemented a few instruments such as Percussion flute, Tubular bell, Hammond organ and Trumpet. The outcome is successful, and the sound produced sounded much more unique than a normal sine wave. There is a much more detailed manual on FM for Csound which can be found at https://flossmanual.csound.com/sound-synthesis/frequency-modulation. The percussion flute csd file is shown below.

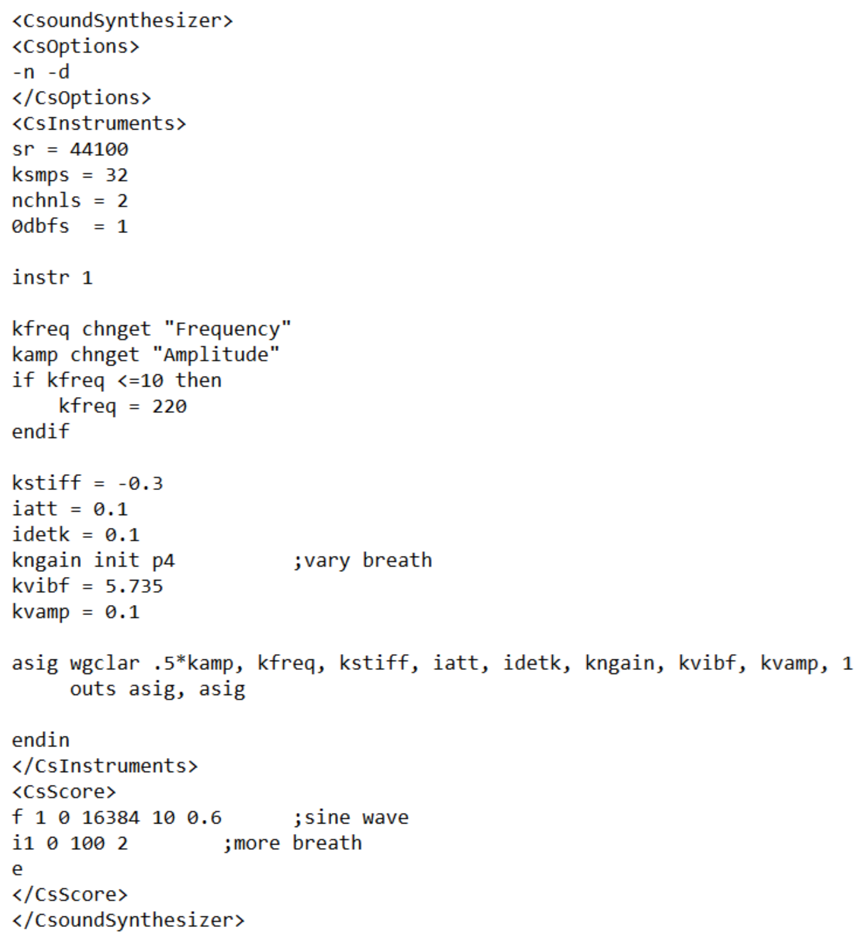


```
<CsoundSynthesizer>
<CsOptions>
-n -d
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs  = 1

instr 1
kfreq chnget "Frequency"
kc1 = 5
kvdepth = .01
kvrate = 6
kamp  chnget "Amplitude"
kc2  line 5, p3, p4
asig fmpercfl .5*kamp, kfreq, kc1, kc2, kvdepth, kvrate
     outs asig, asig
endin

</CsInstruments>
<CsScore>
; sine wave.
f 1 0 32768 10 1
i1 0 100 1
e

</CsScore>
</CsoundSynthesizer>
```

Figure 24. A csd file that generate sound similar to a percussion flute

5.5.4. WG (Wave Guided) synthesis with CSound

Apart from using FM synthesis, there are other methods explored to produce sound using wave guided synthesis. The wgclar opcode is explored here which allows us to create a tone like a clarinet, using a physical model developed from Perry Cook. The implemented csd file is shown below.



```
<CsoundSynthesizer>
<CsOptions>
-n -d
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs  = 1

instr 1

kfreq chnget "Frequency"
kamp chnget "Amplitude"
if kfreq <=10 then
    kfreq = 220
endif

kstiff = -0.3
iatt = 0.1
idetk = 0.1
kngain init p4          ;vary breath
kvibf = 5.735
kvamp = 0.1

asig wgclar .5*kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, 1
     outs asig, asig

endin
</CsInstruments>
<CsScore>
f 1 0 16384 10 0.6      ;sine wave
i1 0 100 2        ;more breath
e
</CsScore>
</CsoundSynthesizer>
```

Figure 25. A csd file that generate a sound similar to a clarinet

The kfreq and kamp variables will be passed from the LibrarySound.cs to control frequency and amplitude and the remaining parameter will determine how would the clarinet sound.

5.5.5. Update User Interface

With the implementation of Csound Library, more interfaces had been added. A new drop-down option to select between the different Csound instruments is implemented. Additionally, the option to toggle between Unity default library and Csound Library is also implemented.

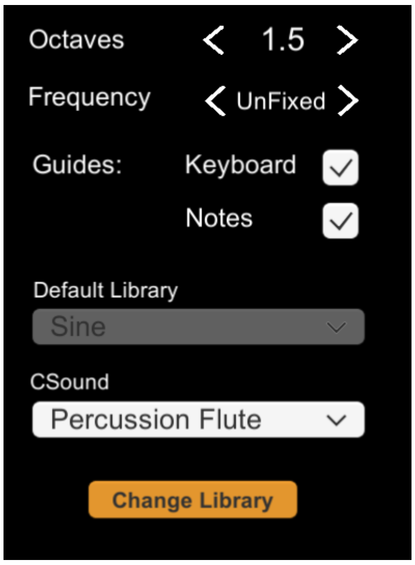


Figure 26. Additional User Interface to toggle between Unity default library and CSound library.

5.6 Customised Starting Frequency

Currently, all instruments can be played from a minimum frequency of 440Hz. This caused some instrument to sound strange. For example, a Tubular bell would sound strange if the frequency played is below 880Hz.

Therefore, additional code is added into the LibrarySound.cs to be able to change the starting frequency variable in UnitySoundLeap.cs, depending on the instrument that the user had selected.

6. Implementation Issues

6.1 Lag Issues

Certain issues such as lags occurs when using the software, and the frame rate dropped to 10-20 frame per seconds (FPS). This usually happens when the user computer did not shutdown or restart for a few days. This is quite common even when developing games in Unity, and the fps is extremely low when the computer had operated for a long time. This issue can be solved by restarting the user computer and start open the software again. This allows the software to be able to run at a normal speed of 200-300 fps again.

Another lag issues happened when headphone is used when testing the software. It will cause a high latency when generating sound. Therefore, no external speaker or headphone should be plugged in and only internal speaker should be used,

6.2 Hardware Issue

Occasionally, the leap motion will stop working when the software is developing and testing with Unity. The leap motion controller is not able to track anything and lose its connection. Plugging out and plugging in the leap motion controller would not solve the problem. The only solution is to restart Unity again and the leap motion controller can then be connected once again.

6.3 Heat Issue

Using the Leap Motion for a long period of time would cause the hardware to feel hot. This may introduce some lag in the software. A good practice would be unplugging the Leap Motion for a while after using it for around 30 minutes.

7. Performance Measurement

Csound library proved to be successfully to be able to generate a more variety of sound in Unity. However, using an external library would introduce additional latency, therefore we aimed to investigate how much additional latency is introduced.

7.1 Procedures for measuring performance

An experiment is conducted to assess the performance with BPM (beat per minute) as the primary determinant. A simple soundtrack arrangement is composed using Garage Band with an iPad. It is important to be familiarized with the song so that the only latency is from the software itself, and not from the delayed human reaction.

The soundtrack can be looped in Garage Band at the desired tempo. The tempo is initially set to 150. We used the Unity default OnAudioFilterRead() function to play along with the soundtrack from the garage band. Each time we can play along successfully with the garage band soundtrack, the tempo will increase by ten.

This continues until our software's latency causes us to be unable to keep up with the pace. The highest tempo will be recorded, and the experiment will then be repeated using the CSound Library.

7.2 Results of experiment on measuring performance

7.2.1 Experiment 1

A simple “Twinkle Twinkle Little Star” song is composed in the Garage Band and we are able to play along with it using the Leap Motion Controller. Both sound libraries can play at a BPM of 240, which is the maximum tempo allowed in the Garage Band. The reason may be because the note transition between the notes is usually 1-2 away. A noticeable delay only happened at the part whereby the note changes from C to G. Thus, a more detailed experiment must be conducted.

7.2.2 Experiment 2

Based on experiment 1, We composed a track whereby there are only two notes, C and G. This track will be looping between note C and note G continuously with increasing tempo until we are unable to play along with the pace on the leap motion controller. This experiment focus on changing between two notes that are almost half an octave apart.

The result is that the default library can play up to 220 BPM, while the Csound library is also able to play up to 220 BPM. However, when the clarinet instrument is selected on the Csound library, there is a noticeable delay and the playable BPM dropped to 200. It may be because the clarinet csd file contains a much more complex formula, compared to a normal sine wave.

7.2.3 Experiment 3

For the final experiment, we played the song “Red River Valley” on the Garage Band and loop it with a BPM of 140. Most of modern songs are within the range of 100-140, therefore if the Csound Library can play within a BPM of 140, it would be able to play most songs.

The result is that both libraries can play along with the soundtrack with a BPM of 140 without much noticeable delay, even with the clarinet instrument. This shows that Csound Library can play most songs without any issue.

8. User Study

A simple user study is conducted on 3 people that have musical background and 3 people without musical background. Those that have musical background are given 10 minutes to play any song that they desired, while those without musical background are given the same amount of time to play anything they desired. All of them are informed to compare the latency between both sound libraries. The result is that all of them felt that the delay is unnoticeable.

9. Conclusion

In conclusion, it is proven that we can simulate playing the Theremin with Unity successfully without producing any audio flaws such as static noise and popping sound. We had also successfully integrated Csound library and introduced more variety of interesting sound to be played. Even though implementing Csound had cause additional delay, it is still barely noticeable. The team will continue to explore more opcodes in Csound to generate a more variety of interesting and beautiful sound.

10. References

[1] K. Mathew, "Strange Vibrations: The Evolution of the Theremin," 2019.

[2] UltraLeap. "How Hand Tracking Works." https://www.ultraleap.com/company/news/blog/how-hand-tracking-works/ (accessed 22 August 2022, 2022).

[3] T. World. "Theremins and Theremin Kits." http://www.thereminworld.com/theremin-store (accessed.

[4] D. Heaney. "Ultraleap Gemini Impressions: Great Hand Tracking Is Coming Soon." https://uploadvr.com/ces-ultraleap-gemini-hand-tracking/#:~:text=By%202019%20even%20AltSpace%20dropped,egg%20problem%20of%20input%20accessories. (accessed 21 August 2022, 2022).

[5] C. Z. Cai, "Playing digital music by waving hands in the air," 2018.

[6] UltraLeap. "The fifth generation of the world’s best hand tracking." https://www.ultraleap.com/tracking/gemini-hand-tracking-platform/ (accessed.

[7] U. Technologies. "Unity Documentation." https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnAudioFilterRead.html (accessed 16-10-2022.

[8] R. Walsh, "Csound and Unity3D " 2015.