Opus: A Music Education Game

Submitted by:
Justin May
Salman Omer
Yash Shah
Jonathan Hong
Skyler Lee

Team Project Number: S20-15

Advisor:
Waheed Bajwa

May 11, 2020

Submitted in partial fulfillment of the requirements for senior design project

Electrical and Computer Engineering Department
Rutgers University, Piscataway, NJ 08854

1. Introduction

Music education is a critically important part of a child's development and is known to improve long-term memory and leads to better brain development [1]. One important aspect of music is practice; one problem with practicing is that it is boring and children hate being bored. Popular play-along music games, such as "Guitar Hero" [2] and "Rock Band" [3], currently allow users to play along in real time to a song using a controller designed to mimic an actual instrument. For example, the guitar controller has five colored buttons that are to be pressed in different combinations using the left hand, and a strum bar to be pressed by your right hand to imitate strumming. The players are then given a visual aid of five vertical lines, each line corresponding to a button for the left hand. Bubbles automatically scroll down these lines and the player must press the correct buttons with their left hand while hitting the strum bar with their right. While users may learn and gain a stronger intuition of some musical concepts such as rhythm, the skills required to play these games will not translate into playing the actual corresponding instrument because of how simplified the controller is. On the opposite spectrum of musical games are "Simply Piano" [4] and "Yousician" [5]. These are phone apps that teach the user how to play real instruments. Such apps can process environmental sound in real time to tell if the user is playing an instrument correctly. "Simply Piano" and "Yousician" seek to replace real music teachers by assessing the user's ability to play off sheet music. Opus aims to cover all the weaknesses of competing products while truly maintaining the essence of a game, where fun comes first. Opus is a musical game where the controller is a real, physical piano instead of an electronic-instrument analog and as a result, the player would learn more skills necessary for playing real instruments than they would learn from playing traditional music games.

2. Methods

2.1 Conceptual design

The Game

Initially there were a few considerations for frameworks to build the game with. Frameworks include: Unity in C#, iOS Spritekit in Swift, and native iOS.

	Unity	Spritekit	Native
Pros	-unparalleled support and community surrounding unity dev. -cross platform (iOS, android, macOS, and windows) -can be built on mac and windows	-common dev. language with integration and DSP team -best graphical performance in iOS, spritekit is an apple library optimized to run on the mobile chips	-common dev. language with integration and DSP team -fastest implementation (no bridge or libraries) -access to low level audio streams and iOS DSP libraries
Cons	-different dev languages: C#, Rust, js (limited support) -potentially largest performance issues with iOS / Unity bridge and C# compilation. potential issues accessing iOS native audio libraries	-no cross platform support -requires macbook	-highest dev time, requires implementing own physics engine and/or rendering system -no cross platform support -requires macbook

One limiting factor was that only two group members initially had Macbooks to develop the application on and iOS applications can only be built on XCode on Macs. This led credence to choosing Unity. Additionally, the fact that Unity was extensible to Android and desktop operating systems was attractive, as it allowed the project to be ported to other operating systems. But, the potential performance issues could not be ignored; the final solution was a two framework solution. The audio digital signals processing algorithms would live in the native iOS application giving us access to vDSP and Apple's Accelerate audio frameworks while the game would live on a unity game.

Despite the benefits, the integration of this two-framework solution posed a challenge in itself as it created a requirement of an interprocess communication layer which passes along the pitch detection information to the Unity game. The initial benefit of Unity being able to compile the game to various platforms quickly became an engineering problem as the iOS application it outputted used archaic Apple frameworks and the Objective-C language, which was incompatible with Opus' implementation of the pitch detection algorithm. Through some extensive research of the Unity framework, the system was reconfigured to use the Unity game as a third-party plugin to a native iOS application developed from scratch. This gave the team greater control of the iOS project and prevented unnecessary bugs caused by the auto-generated project from the Unity compiler.

The Pitch Detection Problem

The difficulty of pitch detection is due to the structure of a note. Each note on the piano is uniquely identified by its fundamental frequency. For example, a Middle C (C4) on the piano corresponds to 261Hz, which is the fundamental frequency. However, the frequency spectrum of a single note is a combination of the fundamental and its harmonics. Harmonics are integer multiples of the fundamental frequency, so if one took the Fast Fourier Transform (FFT) of Middle C, the peak would be seen at 261 Hz, 522 Hz, 783 Hz, and so on. Additionally, if a note is played on both a piano and a violin, the frequency spectrums contain a different number of harmonics with varying magnitudes. This phenomenon is described by the term "timbre", which is essentially how the human ear can distinguish instruments apart. The overall goal of pitch detection is to identify the fundamental frequency out of the rest of its frequency components.

2.2 Detailed Design

Single Pitch (Monophonic) Detection

While there are many good single pitch detection methods like the YIN algorithm[20], which has a wider range of detection, we approached this problem with the Harmonic product spectrum (HPS)[7] for its extensibility to polyphonic detection. This algorithm is also computationally light, which guarantees an acceptable latency in our game. The HPS measures the coincidence of each fundamental frequency with its harmonics. To begin the HPS, take a window of 8192 samples and perform an FFT on it. Note that a bigger sized window will provide a better frequency resolution and applying a Hamming Window[1] to the samples before performing the FFT will help reduce spectral leakage. Take the result of the FFT and create multiple down samples by factors of integer multiples (e.g. 2, 3, 4, etc.). The fundamental frequency of the original spectrum lines up with the next harmonic each time a down sample is taken (denoted by the rectangle in Figure 1.). When multiplying the downsampled FFT together, the biggest peak occurs at the fundamental because it has the largest coincidence with its harmonics. Essentially, the location of the maximum peak (maximum coincidence) in the HPS (frequency domain) is identified as the frequency of the real pitch being played.

[1] A Hamming window is a window function that is multiplied to a segment of a discrete signal that tapers the tail ends of the signal

Harmonic Product Spectrum

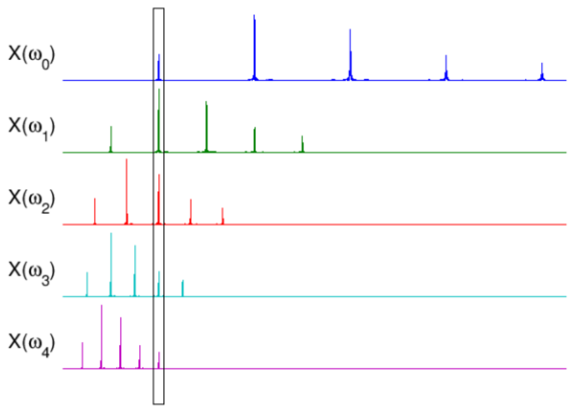


Figure 1. Tamara Smyth, “Music 270a: Signal Analysis”, UCSD [7]

Multipitch (Polyphonic) Detection

With single pitch detection, it is possible to pick the maximum peak of the HPS and call it the fundamental frequency because only one pitch is being sought. However, with multi pitch detection, that certainty is lost because the number of pitches to be detected at any given time varies (≥ 1). After exploring options, the time domain approach (e.g. autocorrelation, YIN algorithm) was found to not be feasible for multi pitch detection. Instead, a detection algorithm in the frequency domain would have to be used. Given the time constraint, an extended/modified version of the Harmonic Product Spectrum (MHPS)[6] was used as the approach. This algorithm has two prerequisites: (1) a set of prominent harmonics and (2) minimal noise. The approach is as follows:

First, take a window of 8192 samples from the microphone:

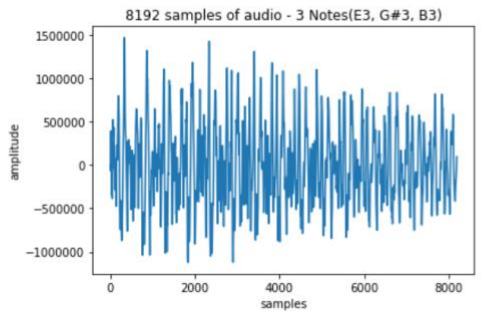


Figure 2. Window of 8192 audio samples

Apply a Hamming window to the signal to reduce spectral leakage[2]:

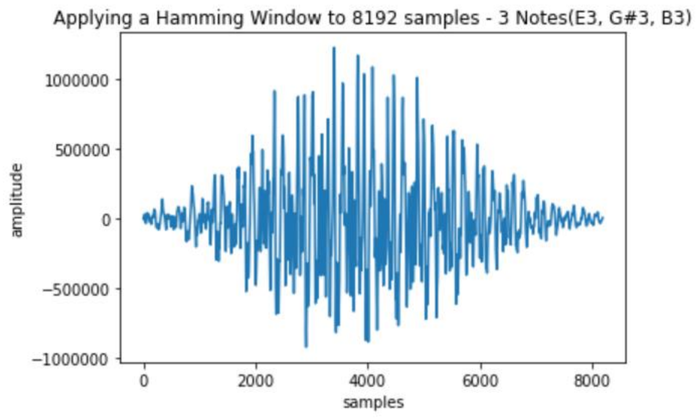


Figure 3. Audio samples multiplied by Hamming Window

Pad zeroes to make the signal length twice as long (to increase frequency bin resolution):

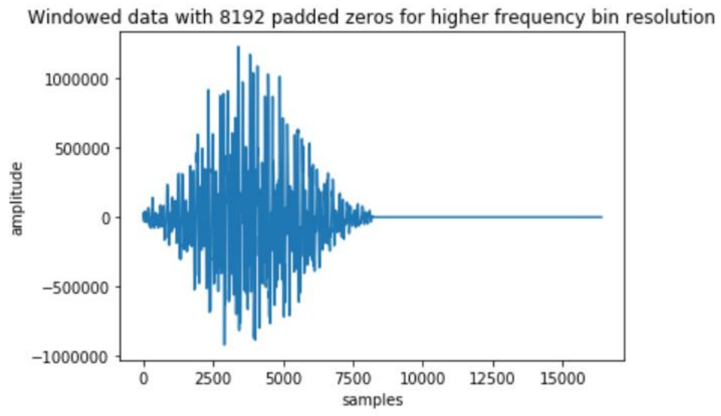
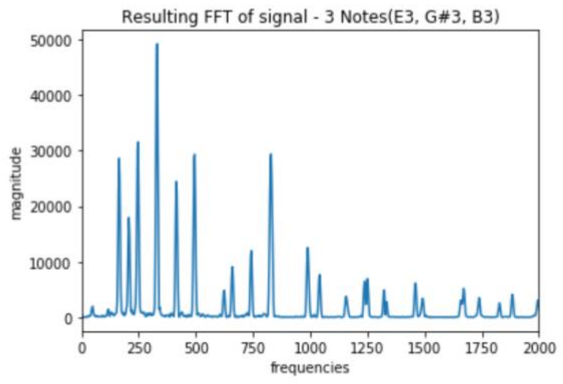


Figure 4. Windowed data padded with zeros

Take the Fast Fourier Transform of Figure 3:



[2] If a segment of samples is taken from a signal, there may be discontinuities at the boundaries of the segment. This causes the FFT to detect more frequencies present than there really are.

Figure 5. Frequency Spectrum after taking the FFT of zero padded audio sample

Downsample to see the coincidence among the next 3 harmonics:

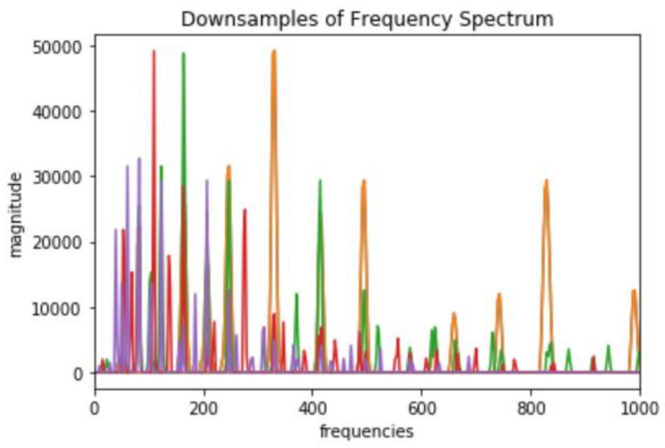


Figure 6. Plot of multiple downsampled FFTs

Multiply down samples together to create the Harmonic Product Spectrum:

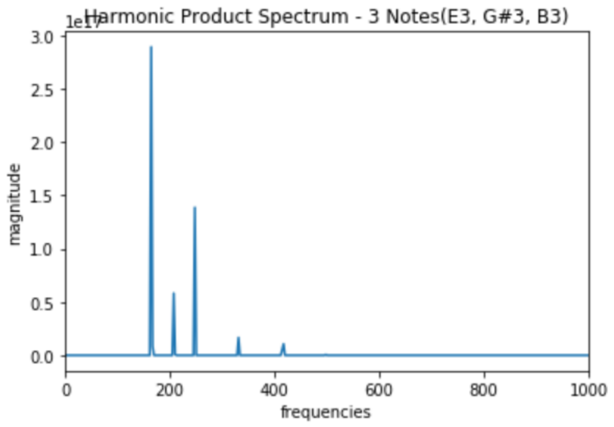


Figure 7. The Harmonic Product Spectrum (product of multiplying the downsampled FFTs together)

Apply a threshold to the HPS. Any peaks above the threshold are considered real pitches (a note currently being played), while the others are classified as uncertain pitches:

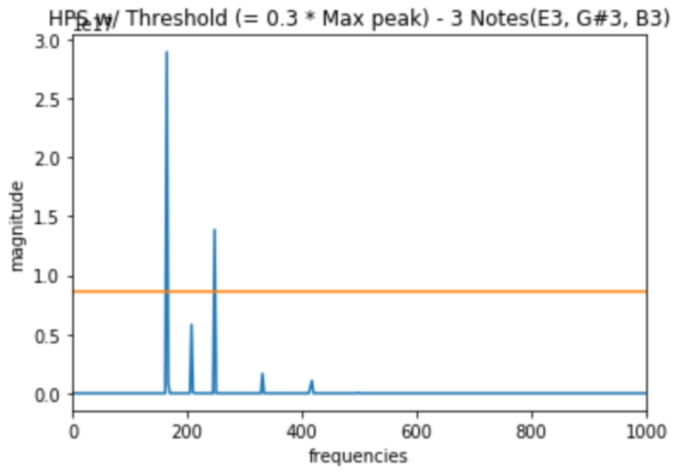


Figure 8. Apply threshold (orange line) to HPS to determine real pitches from uncertain pitches

The remainder of the algorithm explores the possibility of an uncertain pitch to be a real pitch. First, the next four harmonics in the original FFT spectrum of each uncertain pitch are checked. If all of the harmonics are nonzero, it is certain that it is a real pitch (because a musical pitch's energy is distributed among the harmonics [6]). The rest of the uncertain pitches that do not qualify are then tested by one last set of thresholds.

Because a musical note's energy is distributed among its harmonics, the magnitudes of the fundamental frequency and its next three harmonics in the FFT spectrum are summed together for each uncertain pitch [6]. The same calculations are done for all the real pitches already determined, after which they are compared with those of the uncertain pitches. These comparisons check if the energy of the uncertain pitch is large enough relative to the energy of the real pitches. If they are large enough, they can be reclassified from uncertain pitches to real pitches. Lastly, all of the real pitch frequencies are sent to the Pitchy library, which maps each frequency to its closest letter note. Due to frequency bin resolution, the detected frequencies can sometimes be about ~2 Hz off from the actual frequency, but this is already considered accurate enough for Pitchy to label the frequency with the right note.

iOS Software Engineering

The iOS application in Opus serves as the layer which fetches and processes a stream of raw audio signals from the environment in real time. This component was engineered with extensive performance and latency considerations since it serves a critical role as the interface between the user and the game. Keeping that in mind, a set of relevant metrics was devised to help determine if the implementation successfully achieved the abstract goals. These metrics include the latency in sample processing, audio sample size, and, most importantly, precision in frequency detection. The definition of the measure of success for each metric was established through research and consideration of the problem at hand.

The first step was identifying the maximum acceptable latency in sample processing by researching various response times in human perception and hardware limitations that exist on various platforms and implementations. On average, humans have a response time to visual stimulus between 150ms to 300ms [8], and this can be used to define the maximum acceptable latency from when the user plays a note to when the user receives feedback from the system on their mobile phone. In order to ensure that this was feasible, the limitations posed by the iPhone audio interface were identified, where a case study from Superpowered, Inc. concluded that all iPhones released after 2016 had a latency in audio signals between 10 and 20 milliseconds [9]. With this information, a maximum latency of 300ms was chosen as the success threshold for the latency metric.

The next step was identifying the minimum buffer size required by the pitch detection algorithm to detect all 108 keys on a standard piano. The Harmonic Product Spectrum algorithm used for pitch detection requires the system to transform the audio data from the time domain into the frequency domain, and the NyQuist-Shannon Sampling Theorem provides a mathematical proof relating the sample size and the derived frequency. The theorem establishes the condition that in order to sample a frequency X, the sampling rate must be at least 2X. Therefore, since the 108 keys span a frequency range from 16Hz to 7902Hz, the system must sample at a rate of at least 15,804 samples per second. This sampling rate helps derive the success metric for a buffer size of at least 15,804 samples, provided that the audio buffer is sampled at least once every second and matches the minimum sampling rate.

The final step involved identifying the acceptable level of precision in frequency detection for Opus. Note that this system focuses on precision as its metric over accuracy as this component is only responsible for fetching accurate real-time data for the pitch detection algorithm. The pitch detection algorithm further processes this data to achieve a reasonable accuracy in frequency detection. A standard piano has 12 different semitones within each octave, where the frequency between each semitone increases exponentially as you move up in octaves. Due to this fact, the precise detection of frequencies in the third octave was selected as the minimum requirement, as the frequency difference between adjacent keys in this octave is low enough to produce a system with robust frequency detection. Specifically, the success requirement for this metric states that the frequency detection must not have an error boundary outside of +/- 8 Hz of the fundamental frequency of the note being played [10].

Due to a general lack of experience of iOS development across the team, the first few weeks of development for this component were spent learning the Swift programming language and the Apple application frameworks through the available documentation and the relatively small developer community. The challenges presented by the lack of prior experience led the team to the initial approach of using third party libraries which implemented the audio interfaces exposed by Apple. The first iteration of this component used AudioKit, which is an open-sourced audio framework with an easy to use API to access the iOS microphone. AudioKit even presented the ability to automatically perform an FFT across the realtime signal. Unfortunately, the simplicity of this framework posed grave limitations for the Opus project through its inability to control the sampling rate and the buffer size provided by the iPhone microphone, and its FFT capabilities only produced a singular output with the maximum energy found, which immediately posed a problem for multiple pitch detection. Additional research on competing libraries yielded similar results, with new issues rising during the project's integration phase due to conflicting dependencies.

Eventually, the use of external audio processing libraries was scrapped in favor of Apple's AVFoundation framework [7]. The AVFoundation framework has more precise control over the iPhone microphone and therefore is able to set the highest audio resolution possible and tap the microphone at the required intervals to fetch the correct size of audio samples: a sampling rate of 48kHZ with a sample size of 16,000. Prior discussion of the Nyquist-Shannon sampling theorem reveals that this sample size enables the pitch detection algorithm to successfully retrieve the entire breadth of frequencies up to the eight octave on a standard piano.

With a recurring source of real-time audio samples, the iOS component needed a means to effectively communicate this audio event to the Unity framework. Rather than using an inefficient timing approach, this component implements an event based architecture which sends a data message to the Unity framework once the audio processing component successfully gathers and processes the audio data through the use of callback functions. This architecture successfully isolates the three major components of the Opus project—the game running within the Unity framework, the audio signal processing running as part of the native iOS program, and finally the application framework which integrates and ties the whole project together.

During the development of this event stream, the observed latency of audio processing was greater than the initial goal of 300ms. Investigation of this latency proved that this delay was not due to heavy computation, but rather from a hardware limitation. Since the iPhone is only capable of an audio resolution of up to 48 kHZ, and since the system requires a buffer size of 16,000 samples, the minimum sampling interval that could be set was 333 ms. This low sampling interval restricts the iOS component to receiving audio samples every 333 ms, which exceeds the success metric before even accounting for the added latency due to data processing and computation. In order to solve this problem, the team devised a sliding window approach, where the iOS component taps the microphone for a smaller set of audio samples at a more frequent rate. The smaller sets of samples are appended to a collection of samples, while sliding out a part of the previous set of samples.

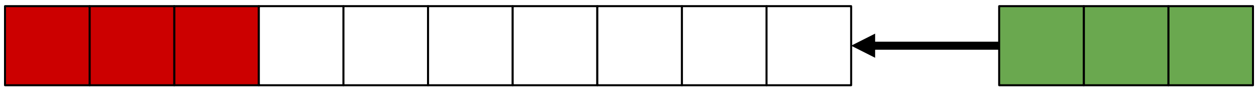


Figure 11. Visual Representation of Sliding Sampling Window

This approach enables the audio processing component to retrieve and process audio data every tenth of a second, while maintaining a sample set of 2^{14} samples, or 16,384 samples. Further testing of this method yielded that this approach inadvertently helped to solve another issue relating to pitch detection. It turns out that the prior approach of using discrete sample sets occasionally causes the data representing the playback of a piano note to be split between two sampling intervals, ultimately causing a loss of data. Fortunately, the sliding window approach ensured that the frequency detection phase received a smooth, continuous flow of audio samples and prevented this data loss.

The final stage of implementation involved using the Accelerate Framework to implement the algorithm yielded from researching various pitch detection approaches. The Accelerate Framework provides a library of performant digital signal processing and vector manipulation functions which were used in the implementation of the Harmonic Product Spectrum. This framework helped this component achieve the latency requirement defined at the start of this project.

Unfortunately, the results found during testing indicated that the precision in note detection was extremely low, which contradicts the initial research that was conducted on the pitch detection algorithms using Python and the recordings of individual piano notes. Investigation of this problem led to the discovery that this low precision value stemmed from random frequencies being detected from background noise that exists in the real world environment. Using a third party application, SPLnFFT, the following snapshot of a real time FFT was taken in a quiet room on the testing device.

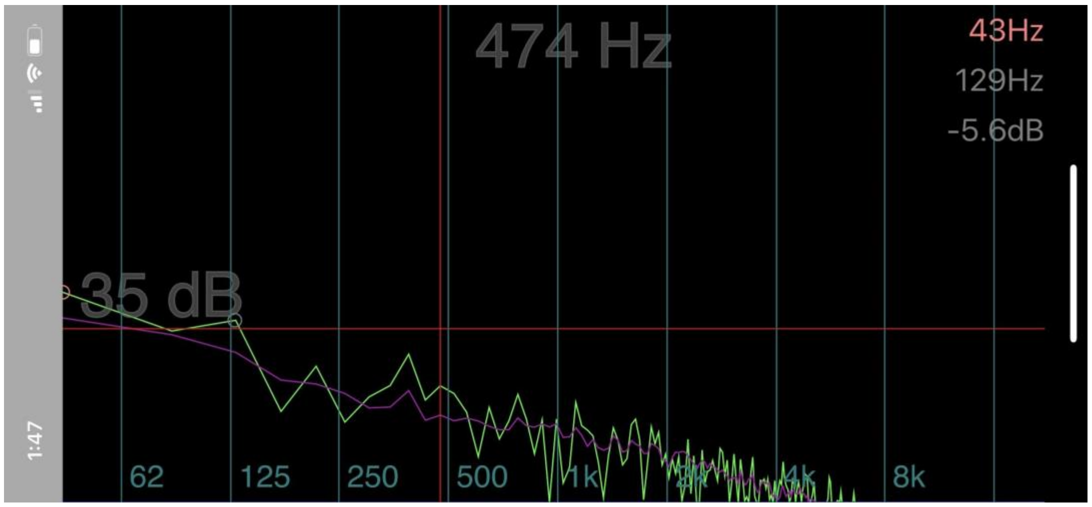
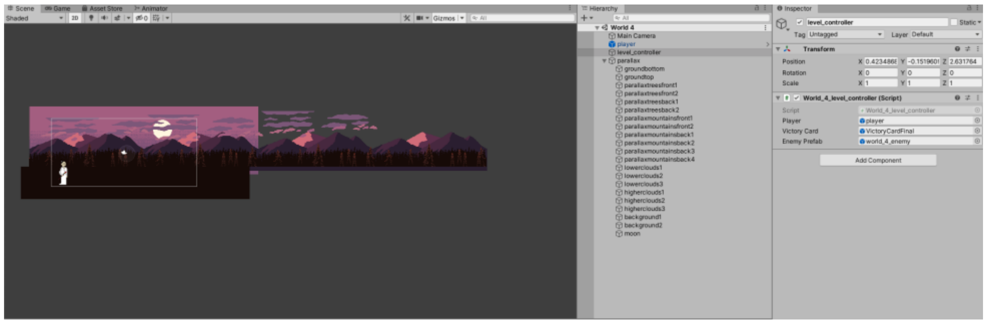


Figure 12. Snapshot of a real time FFT of a quiet room taken using SPLnFFT

The snapshot in Figure 12 demonstrates the frequencies that exist in the environment as background noise in a quiet room. This data exhibits the need for an additional audio processing step where the component performs noise level filtering to ultimately increase the precision in frequency detection. This extended the use of an event-based architecture for the process communication layer as it prevented the Unity framework from receiving any unneeded messages about the microphone readings when the user is not playing any piano notes. The implementation for this noise level filtering involves taking the average power level reading over the amplitudes of the audio data. Then, the component uses a predetermined threshold value, which testing yielded to be -35 dB, to determine if a specific frame of audio data is significant.

Unity Software Engineering

The primary directive of the game was performance; because the audio DSP would eat up a large amount of the phone’s resources, the game had to be incredibly performant. The game itself is written in C# inside the Unity Framework. Each world is a self-contained scene, a Unity framework notion of a separate memory space for different groups of assets, scripts, and processes. Much like other front-end development frameworks such as Xcode, the Unity Engine editor allows developers to drag and drop elements on the screen, organizing the GameObjects in a hierarchical view with their properties exposed from an inspector panel. The initial levels in World One were constructed this way and each level functioned off one event loop, a thread that updated once per frame. This was not very performant as multiple checks occurred every frame that were waiting on various conditions to update or timers were simply constructed from incrementing counters per frame. Furthermore, the drag and drop method is limited as all assets that would be needed in the beginning of the level and exist during the lifetime of the scene.



```
level_controller                          Static
Tag Untagged        Layer Default
Position   X 0.4234868  Y -0.151960  Z 2.631764
Rotation   X 0          Y 0          Z 0
Scale      X 1          Y 1          Z 1
Script        World_4_level_controller
Player        player
Victory Card  VictoryCardFinal
Enemy Prefab  world_4_enemy
```

Figure 13. Unity Engine Editor

A limitation of the drag and drop method is that it is difficult to manage enemy states, parameters and variables specific to each enemy such as animation state, health, text, current notes, etc.. World 2 introduced the concept of "forest_enemies" objects that housed all the necessary parameters for each enemy and this class referenced the enemy GameObject (a unity-specific object type) passed to the level_controller script when it initializes the enemy. The enemy object exposed callbacks for hurting the enemy and accessing public variables, as well as handling animating enemy objects, managing the death state, health state, animating the associated text objects, and destroying itself during the death sequence.

One issue with World 2's implementation was the lack of code abstraction and huge code blocks. Unity's underlying advantage is the ability to abstract the logic into different scripts attached to GameObjects. Later worlds moved enemy states onto scripts attached to each enemy GameObject. The advantage is that the scripts can directly reference GameObjects they are attached to on initialization instead of making expensive "FindObject" calls.

The natural next step was to begin multithreading specific event drive logic away from the update thread and avoid adding logic to the update thread all together. For example, in World 3 when an enemy is spawned a specific series of events needed to occur in order: 1. the enemy needs to be referenced and spawned from memory, 2. the enemy needs to fall, 3. the enemy's box collider needs to detect that it has hit the ground, 4. the enemy's collider triggers the animation's controller endpoint to begin the "hit ground" animation, 5. the health bar needs to load on the enemy, and 6. the enemy needs to begin accepting inputs from the player. Each piece of event driven logic was taken out of the update thread and turned into "coroutines" or unity's notion of a thread.

The final UML Diagram is shown in Figure 14 and highlights how far the different worlds have come. Each world references the same static Settings and Scales classes to keep user configuration data consistent across worlds. The main menu calls a Unity change scene method and passes level and world data to the static settings class before it changes scene. Each world includes micro-optimizations to squeeze out the last bits of performance such as adding frame-specific event-triggers to animations to avoid using the "update" thread.

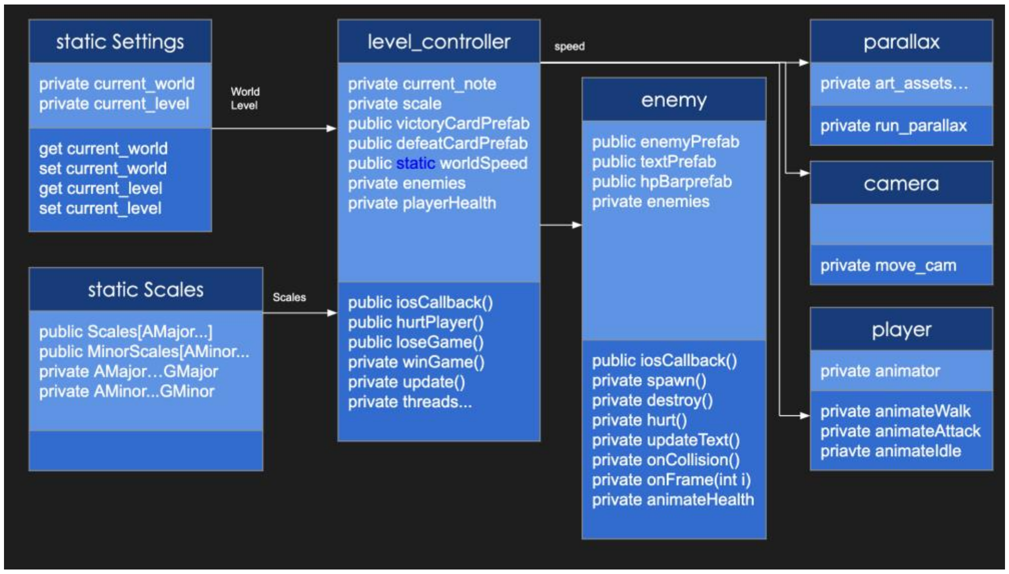


Figure 17. Final UML Diagram

As each member of the game team was relatively new to Unity, drag-and-drop elements were first used to break into Unity. Once integration to mobile devices began, it was clear that much optimization was needed: The processing power of a personal computer is magnitudes higher than that of a mobile device, so one of the avenues explored in the pursuit of optimization was programmatically spawning our enemies. Some of the game levels were reaching dozens of assets and behaviors that needed to be created and updated sixty times a second. Paired with the tedious work that would have followed if drag-and-drop were to be continued being used, programmatically spawning our enemies with C# was proved to be a correct choice in tackling these two issues.

Animating the Game

Most of the in game assets are completely original, except for the backgrounds on the Main Menu and Worlds 2 and 4, all of which are open source. Making all the assets was a huge undertaking and ate up a lot of time. One design choice that had to be made early was how the project would graphically look.

Considering how a majority of the team has little to no experience in digital graphic design, there were two popular options: Rigging or sprite sheets.

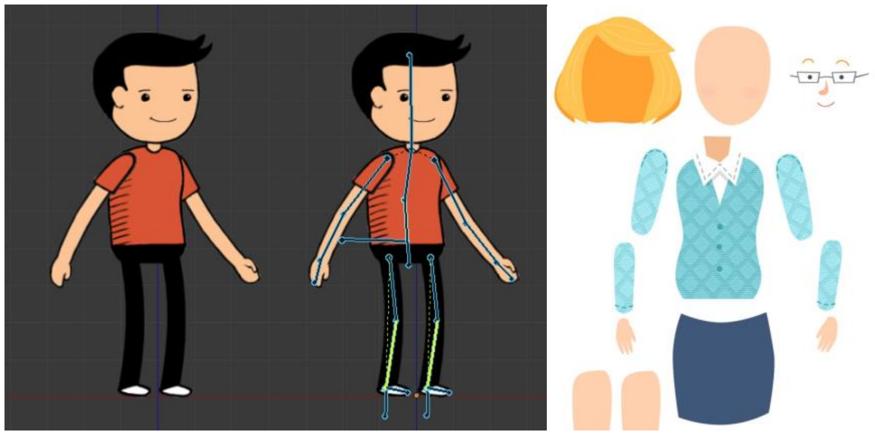


Figure 18: An example of rigging a 2D animated character

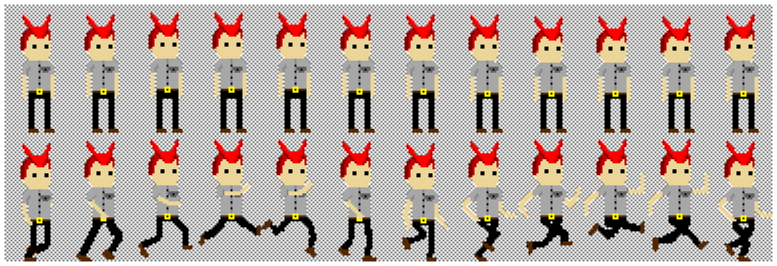


Figure 19: An old sprite sheet of Opus' main character

Both methods of 2D animation bring their own benefits and drawbacks: Rigged characters make it easy to reuse assets and not have to create separate sprites for each frame of action, but what the characters can do is defined, and therefore restricted, by the program (in this case, Unity). While using sprite sheets was time consuming, using sprite sheets made it so that controlling what was being shown on screen every frame was easier, and did not pose any restrictions on what could and could not be done.

Digging a little bit deeper into the actual animation, because C# scripts were used to control sets of assets per level, signals were sent to and from different game objects that had scripts assigned to them.

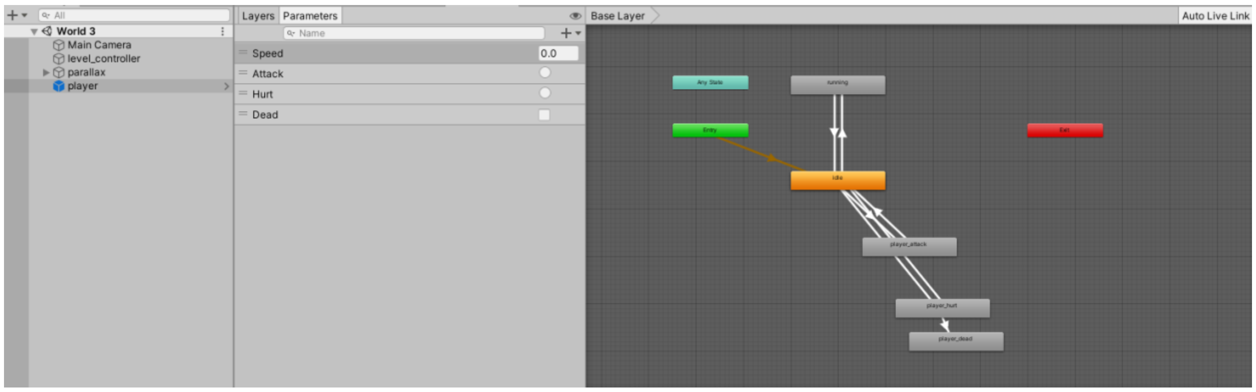
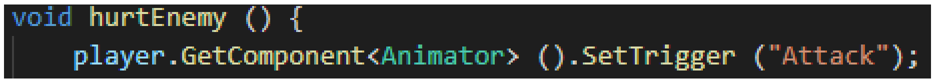


Figure 20: Unity's parameter system for Animator

With the scripts and Unity's parameter system, assets in the game could communicate with each other with said parameter system. For example, whenever the player inputs the correct note, the level controller's controller script would send the trigger parameter of "attack" to the main character asset, letting Unity know to play the attack animation of the main character on the next frame.



```
void hurtEnemy () {
    player.GetComponent<Animator> ().SetTrigger ("Attack");
```

Figure 21: Code from World 3's level controller script

iOS-Unity integration

Because our conceptual design required the game to be developed in Unity while accomplishing the audio digital signals processing on the iOS application, there existed integration challenges. Unity's framework trivializes creating stand alone Unity apps, but running iOS code at the same time presents integration challenges. The solution is to export the game as a library which could then be imported in the iOS application. In Xcode, one can import the iOS project that Unity generates as well as a standard single-view iOS app project into the same workspace. This shared workspace gives the iOS part of the app full control of the life cycle of the game and makes it possible to implement communication between the two.

Unity-iOS communication is accomplished with the Swift function sendUnityMessage(). This function takes three inputs: the game object ID to send a message to, the name of the function within the game object to send the message to, and the message itself as a string. Once some audio data is processed in iOS to find out what notes are being played, that data can be sent to Unity through this function as a parseable string. IOS sends three different kinds of messages to Unity: when a note is starting to be played, when a note is finished playing, and the current volume of sound in the environment. For example, assume that for seven consecutive audio samples the sampler returns the note arrays [A], [A], [A,B], [A], [B], [], [] for each sample respectively. In the first sample, a “start note(A)” message is sent to Unity. The second sample shows that “A” was recognized which is the same as the previous sample, so no message is sent. In the third sample, a “B” appears for the first time, so a “start note(B)” is sent to Unity. In the fourth sample, only an “A” is present. However, a “stop note(B)” is not sent to Unity even though there is no longer a “B” present. Since the polyphonic note detection algorithm sometimes fails to recognize simultaneous notes, the user is given the benefit of the doubt, and the program does not make a decision on whether or not “B” has stopped playing yet. Once the fifth sample containing a “B” is processed, it is assumed that the user was also playing a “B” in the fourth sample. In reality, the program samples at ten samples a second. This means that during the fourth sample, for 100 ms, the program is unsure whether or not a “B” is playing but withholds judgement until later. The program waits for two consecutive sampling windows to lack a note before it decides to send a “stop note()” to Unity. In the sixth sample, both “A” and “B” are missing, but only “A” has been missing for two consecutive samples. So, only a “stop note(A)” is sent to Unity. In the seventh sample, a “B” is missing, so a “stop note(B)” is sent to Unity. During each sampling period, one “volume(X)” is sent to Unity, where “X” is the decibel volume of the environmental noise during that sampling period.

On the Unity side, each level is constructed with a “level_controller” object as the highest level object. This object receives messages from iOS and executes certain behavior in the game based on the level. For example, World One only cares about when a note is started since the duration that the user plays a note does not change the behavior of the level. Once the correct “start note()” message is received, the enemy is immediately damaged. On the other hand, World Five tests how long the user holds particular notes, so it has a different implementation for how it processes“start note()” and “end note()” messages from iOS.

3. Results

The main motivation for developing the app with an iOS layer is to leverage the low latency audio processing capabilities of iPhones. But in doing so, an element of inefficiency is introduced in the design by having the two iOS and Unity software layers communicate with each other. To ensure that the communication aspect of the app did not hinder its efficiency, it was necessary to test how long it takes for Unity to receive a message from iOS. In practice, intra-layer communication took roughly 17 ms on average and reached 33 ms in the worst case. The spread of the latency is consistent across multiple types of iPhones as seen in the graph below. The technical requirement imposed by the game is that communication is faster than the time it takes for a game object to update to its next frame of animation. Since the fastest game object updates at sixteen frames per second or once every 62.5 ms, this communication delay is well within the requirements given by our game implementation.

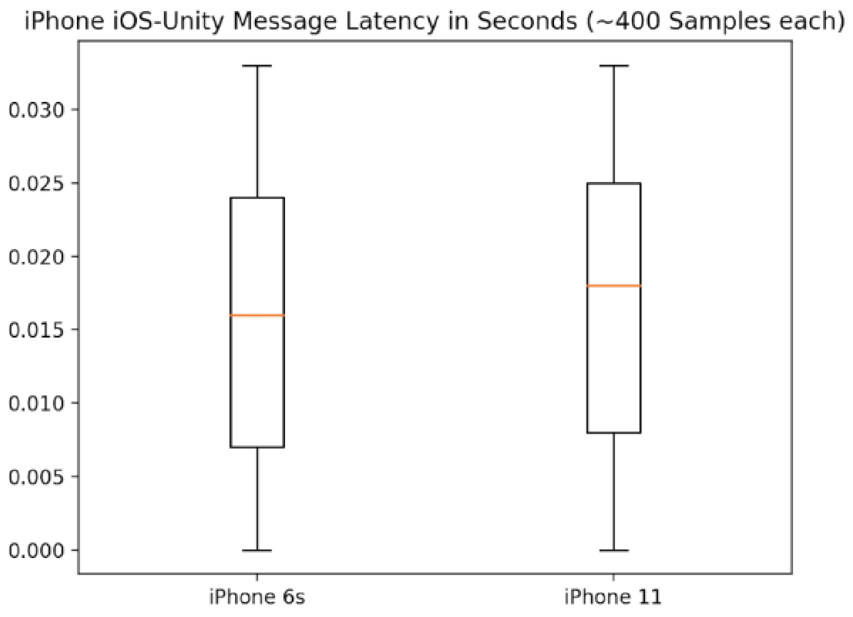


Figure 22. Communication latency comparison for iPhone 6s and 11

The next point of latency measurement involves measuring the time it takes for the system to process the sliding window implementation and perform pitch detection on the sampling window once it receives a batch of samples from the iPhone microphone. The results in Figure 23 indicate that on average, it takes the system anywhere between 15ms to 30ms to process data at an idle state and up to 60 ms when it needs to perform heavy computation to fetch multiple pitches from a set of samples.

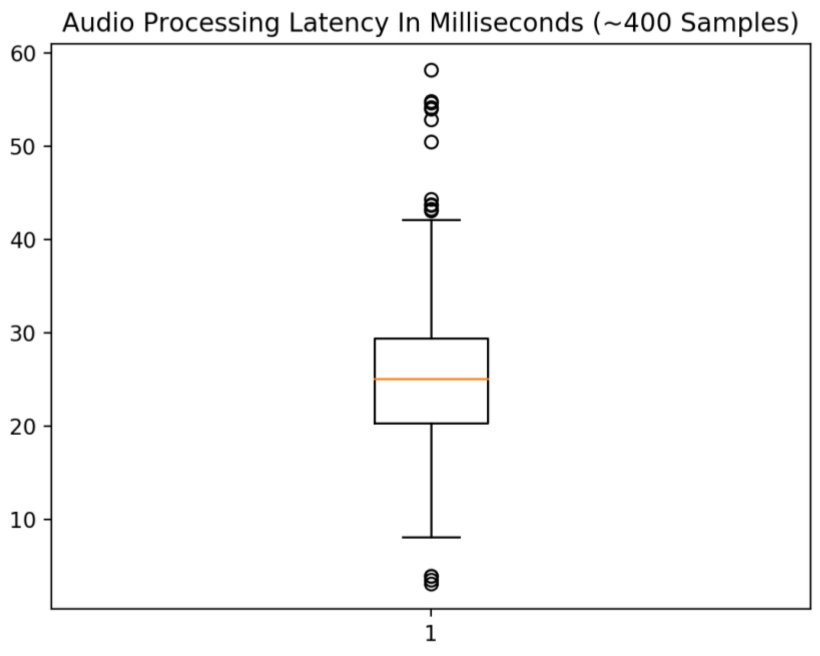


Figure 23. Audio Processing Latency Measurements

These two latency measurements help to identify the maximum time it takes the system to provide the user with feedback once the user plays a note on their instrument. Taking the initial worst-case scenario of 100 ms delay due to the prescribed sampling rate, the worst-case sampling processing latency of 60 ms, and the worst-case latency in inter-process communication of 33 ms, it can be concluded that the system meets the success metrics for both latency as well as sample size which was defined for the audio processing component.

Single Pitch Detection Results

As discussed in the detailed design section, the Opus note detection system must have a high precision in detecting notes in order to successfully achieve its goals. To be more specific, this metric required that the error boundary in frequency detection must be no more than +/- 8 HZ from the frequency of the note being played. In order to evaluate the implementation, a single note was played fifty times, and the estimated frequency was recorded. This frequency value was then compared with the original note's frequency by taking the difference and the values were then plotted onto a box plot to visualize the error boundary across all samples.

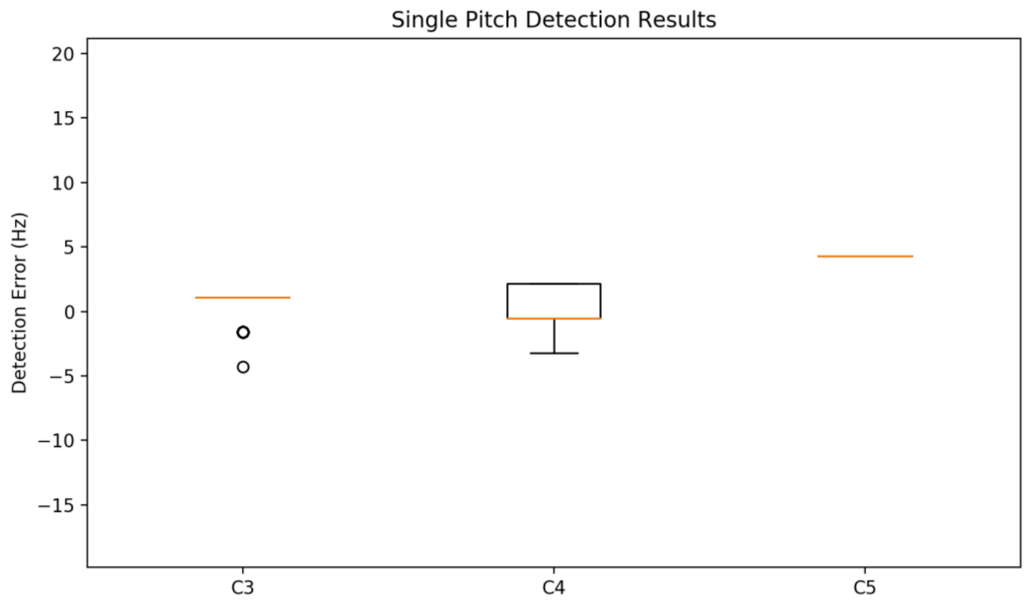


Figure 24. Error Boundary in Single Pitch Detection

Figure 24 displays the results yielded from this evaluation procedure across a single note on three of the primary octaves. Note that this figure has been zoomed in and does not display all outliers which occur due to the overtones which exist at the beginning and end of a note's audio signature. However, the box plot clearly identifies the inliers of this dataset, and highlights the error boundary of our implementation. The results indicate an extremely high precision value through the slim box, where the estimated value of a note differs only between +/- 2 Hz in the worst case between repeated playbacks of the same note, which is well within the success metric defined earlier. Furthermore, the figure also points to a relatively high accuracy from the frequency estimator, as the notes in the third and fourth octave are within 2 Hz of the note's actual frequency, and within 5 Hz for the fifth octave. These results are well within the initial goals for the Opus project and prove its capability for achieving the original goal.

Multiple Pitch Detection Results

Unlike single pitch detection, the evaluation procedure for multiple pitches involves measuring the system's ability to detect the notes being played. The evaluation procedure involves taking the four most common piano chords and repeating them fifty times while collecting data on all of the notes that were detected by the system. The resulting data can be traversed through to perform binary classifications of successful detections and count false positives. From this classification, a "True Positive" result or "TP", is identified as the scenario where the pitch detection algorithm detected all three notes from the chord successfully. A "False Positive" classification, or "FP", identifies the cases where the pitch detection algorithm found additional notes that were not being played. The following figure displays the results of this binary classification from testing the multiple pitch detection algorithm across all four chords.

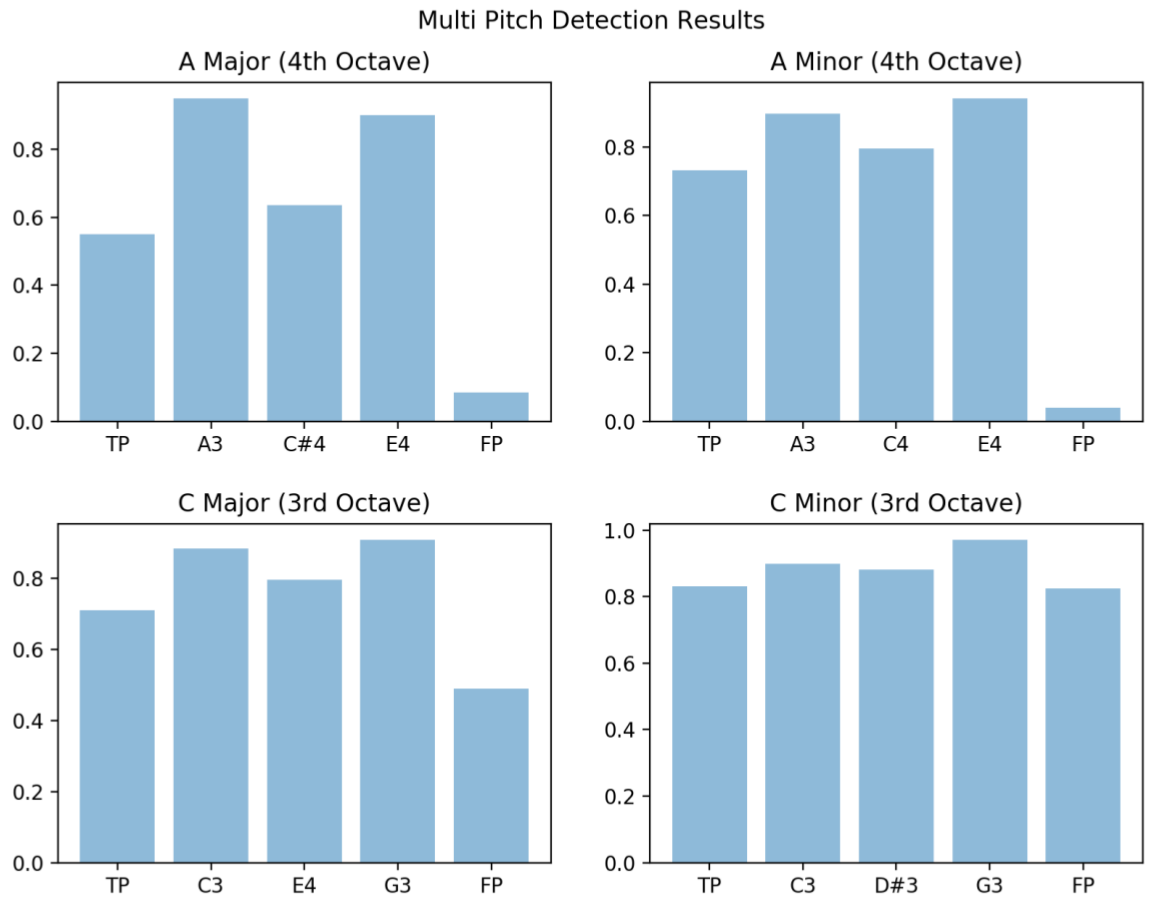


Figure 25. Probability of Success in Multiple Pitch Detection Using Common Piano Chords

The charts in Figure 25 portray the probability of success in multiple pitch detection through the repeated testing procedure. While the results convey an imperfect solution, they indicate a promising start to this complex problem, especially since the probability of successfully detecting the individual notes in each chord are relatively high. Through further optimizations, the system is capable of increasing it's accuracy in detecting multiple notes and providing a user with a system that is capable of teaching advanced musical concepts.

4. Future Improvements

Deep Learning

The primary goal for audio processing is to achieve polyphonic note detection for a given input audio sample. With traditional DSP techniques, it is possible to detect the characteristics of a single note being played, but accurately detecting multiple notes played simultaneously through DSP techniques is an open research problem. This is because the timbre and resonant frequencies of the different notes create complications. A possible solution is to use deep learning [12] to achieve polyphonic note detection. Since Simply Piano, an app that achieves polyphonic note detection for up to five simultaneous notes, uses deep learning, it is evident that a deep learning approach has merit in Opus [13].

Data Collection

Since deep learning solutions are data driven, it is necessary to have ample data to start. However, it would be difficult to manually collect data from a real world environment since it would take a huge amount of time and effort to collect and label it meaningfully. While not ideal, the best alternative is to slightly modify a publicly available dataset. For this project, we consider two promising datasets: Nsynth [14] and Maestro [15].

Nsynth is a dataset that contains separated audio files of individual notes from various instruments. The Nsynth dataset contains ~300,000 audio samples from ~1000 instruments made available through commercial sampling libraries recorded at 16kHz. For Opus, the instrument of interest is the electronic keyboard, which has 42,645 samples in the dataset. Each sample is a four second, monophonic audio recording ranging over the pitches of a standard MIDI piano (21-108) [14]. This corresponds to all the available notes on an 88 key piano from A0 to C8. In each sample, the note is held for three seconds and allowed to decay for the final second.

This data is a good starting point but would still require further preprocessing. Opus uses the common sampling frequency of 48kHz, while the Nsynth data is sampled at 16kHz. If the game used a pure deep learning solution for note detection, then it would be feasible to change the game's sampling frequency to

16kHz. But, if that is not possible, then the training data would have to be upsampled to a frequency of 48kHz. This would result in a three times increase in file size and would have similar implications in training and processing time. For the rest of this analysis, assume that the game will continue to sample sound at 48kHz and that the data is modified to accommodate this requirement.

Another shortcoming of this data is that all the sound files are monophonic, meaning there is only one note being played at a time. Since one of the main reasons for using this deep learning approach is to allow for polyphonic note detection, it is necessary to create polyphonic data to train on. To create this data, samples of different notes could be added together and given labels of both notes. If the goal is to have the same number of each two-note sample as each single one-note sample and have all possible combinations of two-note samples, the dataset would increase by 100 times. Achieving three-note samples or four-note samples would result in similar increases in the size of the dataset. Considering that the upsampled dataset of 42,645 samples is roughly 9 GB in size, this method for building polyphonic samples would quickly get out of hand, giving thousands of gigabytes of data to work it. Processing that volume of data would require more computing resources than would be available. It is possible that the network could be trained accurately with significantly less data, but the only way of knowing is to test it.

The last modification to make to the data would be to add some noise to each sample. Since this data is recorded electronically, there is no noise similar to the noise one would experience in a real world environment. Since the game is meant to use a microphone in a noisy environment, it is possible that a network trained on pure data would not be able to generalize for noisy data. This noise could be generated by recording the environment, adding random gaussian noise, or adding sounds from other datasets. One possible dataset that could be used for this purpose is the FreeSound dataset [16].

Another interesting dataset that could be promising for Opus is Maestro. Maestro is a dataset that contains over 200 hours of virtuosic piano performances recorded from concert quality acoustic pianos. The piano used for these recordings is able to produce accompanying MIDI files, giving information on what note is played at exactly what time with a ~3 ms error. This data is recorded at 44.1kHz in an impure environment. This dataset contains upwards of 7 million labeled notes, providing sufficient data for an Opus application.

This data avoids many of the problems that Nsynth has. This data is polyphonic, not recorded in a completely pure environment, and has more room for variation in sound between different playings of the same note. This kind of data would allow the network to better generalize on what notes are being played at a particular time. However, one shortcoming of this data is that all possible note combinations that may appear in other genres of music may not be properly represented. As a result, the network may be unable to identify all combinations of notes being played at all times, even if those combinations may be common in music overall.

Preprocessing and Training

Currently, the game processes sampled sound at 48kHz. This sampled sound is read into a window of size 16384, corresponding to that many samples. The data in this window shares the tail-end of the data from the previous window, giving it some overlap. The data in this window is processed and the output is then prepared for use in the game logic. A deep learning solution would adapt this sliding-window sampling design to get an input of data. To do so, the input data would have to be modified respectively. For example, divide a 4 second audio clip sampled at 44.1kHz from the Nsynth dataset into segments of size 16384 with some amount of overlap. A second's worth of data would be split into 10 samples of size 16384 to allow for some overlap. So, a single 4 second audio clip would produce 40 sample inputs in this sound processing design. This process would be done with every audio file used. If this project is carried out, various input sizes would be tested to see which would work best for a deep learning approach.

Opus' music processing requirement is not one that can be easily satisfied by using transfer learning on an already existing neural network since few networks exist for similar use cases. Instead, different networks would be constructed from scratch and tested to see what works. For music processing, there are two main approaches. One option is to convert the audio to a 2D format such as a spectrogram. After conversion, one would process that data using 2D convolutional neural nets (CNNs) whose architectures are based on networks used for image processing. The second approach is to keep the audio in a 1D format such as a time series sample and use a 1D CNN instead. Abdoli et al. (2019) suggest that a 1D CNN can achieve a 89% accuracy in classifying types of environmental sound [17]. This section will explore how their approach can be adapted to Opus' music classification problem.

Abdoli et al. tested 1D convolutional networks where the inputs pass through multiple alternating convolutional and pooling layers before being consolidated into multiple fully connected layers. Every layer except the last layer uses a ReLU activation function. The last layer uses a softmax activation function for classification. To reduce overfitting, batch normalization is applied after the activation function of every convolution layer. After the final pooling layer, there are two fully connected layers with 128 and 64 neurons respectively on which a dropout of probability 0.25 is applied. The last layer is a fully connected layer with 10 neurons used to classify the input into 10 classes. Mean squared logarithmic error is the final error function used. In their deepest network, there are 5 convolutional and pooling layers. If they were to make their network deeper, the network would be susceptible to overfitting since they do not have as much training data to work with (~8000 samples).

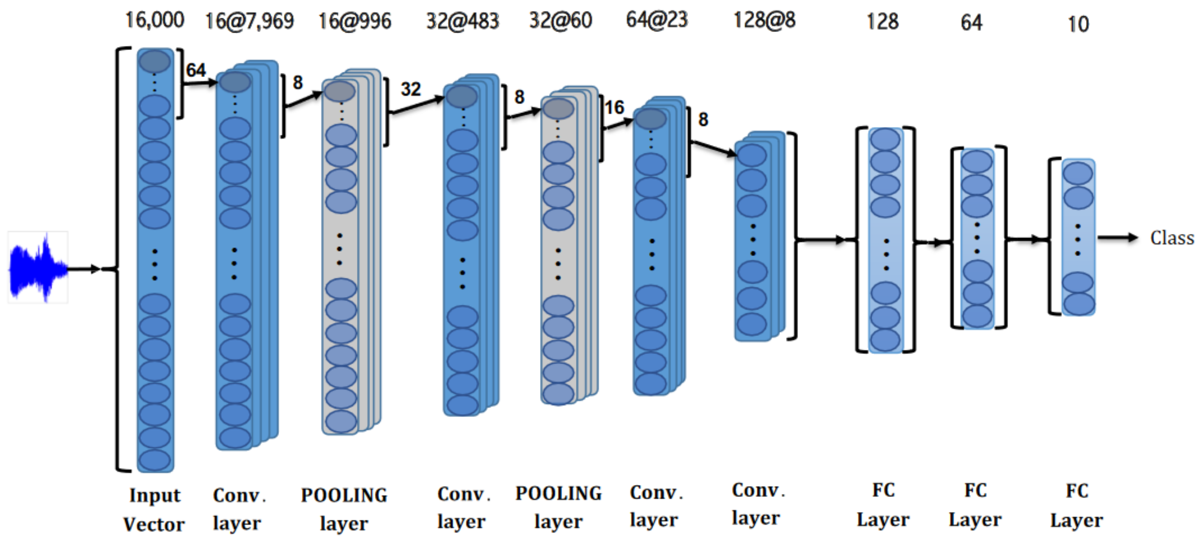


Figure 26. Abdoli et al.'s network architecture for an input of size 16000 with convolutional (Conv) and fully connected (FC) layers [17]

Abdoli et al. tested multiple networks with multiple sizes. The most applicable to Opus are their architectures with input sizes 16K, 8K, and 1.6K. While an Opus implementation would not have exactly the same size input, Opus can use their architectures as a starting point. An important adjustment they make in their networks is with the initialization of their first layer's filters. Instead of starting with random filters, Abdoli et al. experimented with using a Gammatone filter bank. This is a set of 64 band pass Gammatone filters with central frequencies ranging from 100 Hz to 8 kHz. When they used this filter bank as their filter initialization, they saw a 4% increase in classification accuracy compared to when they did not initialize the filters.

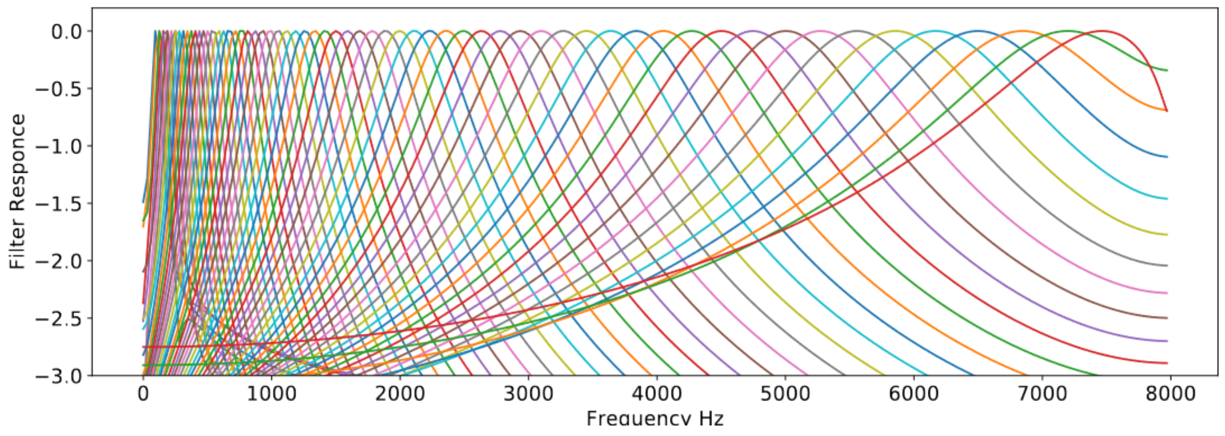


Figure 27. Frequency response of 64 filters of Gammatone Filter Bank [17]

The biggest difference in a neural network for Opus would be in the final layer. Abdoli et al have a network that can only classify 10 classes, while Opus would have to classify for 87 classes. As a result, the final fully connected layers would have to be modified to accommodate for this. With the large amount of data available in relevant data sets, it is possible to classify an input for all these classes. Another important difference in an Opus implementation would be in the final activation function, changing it from a softmax to a sigmoid function to allow for multi-class classification. Some threshold value would be set on the sigmoid function to classify if a note is being played or not. This is necessary to achieve the goal of polyphonic note detection.

5. Use of Standards

Standardized software development tools and software environments implemented in Opus:

a. Unity 2019.3.2f1
b. Xcode 11.4
c. iOS versions >13.4 or <=12.4.4 (13.3.1 broke dependencies)

Tested Hardware

a. iPhone 11
b. iPhone X
c. iPhone 6s (worked prior to sliding window implementation; processor cannot keep up with the new sampling speed)

iOS 3rd party libraries

a. Pitchy

6. Cost and Sustainability Analysis

Opus is a purely software solution to the problem of making musical education more accessible. As a software solution, Opus can be developed without any environmental considerations, and it does not require traditional manufacturing costs that a physical product might require. Furthermore, Opus can be downloaded in its entirety to a phone and does not conduct any network communication. This means that there would be no server costs to maintain the app. The only "manufacturing" costs would be paying to have the app appear on the App Store as well as paying Apple a cut of any proceeds that the app generates. Thus, costs would be primarily developmental costs to program the app.

Currently, there is no monetization implemented within Opus, but there are many monetization approaches that could be explored in the future. Some possible approaches are charging the user a one-time fee to download the app, charging a subscription fee for continued use of the app, having in-app advertising implemented through Unity and iOS frameworks, or any combination of those methods. The competing apps "Simply Piano" and "Youscian" operate on a subscription model, charging $10 a month. While Opus needs to be considerably improved before it could reasonably charge any amount of money, it is easy to see how the app could be similarly monetized.

Social Impact

One of the goals of Opus is for musical education to be easily accessible to anyone without the need for a private teacher. By doing so, people have more agency when pursuing musical education, removing barriers such as finance, finding music tutors, and possibly the commute to and from said music tutor. Opus is not trying to replace other forms of musical education, but rather, Opus is making musical education more readily available, and can be used in tandem with private musical education. Concerning consumption patterns, hopefully families who cannot afford music lessons or do not have music lessons accessible to them can come to Opus as a comprehensive alternative, and those who can will see Opus as a great tool for keeping people engaged in music outside of the music lessons.

According to studies, musical education has a clear positive impact on the development of the brain: People who begin musical training early in their lives will develop areas of the brain related to language, spatial intelligence, and reasoning, as well as better SAT scores [18]. Another study conducted by the Sam Houston State University cites multiple other studies for the already measured effects of private music lessons, affecting a variety of factors such as intellectual development using IQ as a metric, learning behaviors, and socialization [19]. Looking at a specific 3 year study by Piro and Ortiz published in 2009, students who took piano-based music lessons twice a week saw a significant increase in vocabulary and verbal sequencing scores versus the control group [19]. The psychological and intellectual effects of music education for both children and adults are well documented, and more can be done with musical education as a parent for their children, an example being how Opus can also be used as a vehicle for teaching discipline. The adage of “practice makes perfect” implies that work must be done in order for there to be progress, and without motivation, many children lose focus and interest. With Opus’ goal of being a game first, motivation for practicing the piano comes naturally.

7. Conclusion

While Opus is primarily an educational tool, it attempts to take a different approach from simply having sheet music scroll across the screen. Opus seeks to create a more engaging experience by incorporating creative elements such as unique artwork, game design, and the beginnings of a story. In doing so, this project is a platform to more easily confer the benefits of musical education on anyone who wants to get started. To achieve this outcome, a number of development challenges had to be overcome. In Unity, strong software engineering fundamentals were employed to make the game computationally efficient and scalable. Furthermore, a front-end filled with unique artwork was implemented to make the game fun to play. On the iOS side, musical notes are identified in real-time from sound in the environment using algorithmic techniques such as the Modified Harmonic Product Spectrum (MHPS) and a sliding window sampler. Lastly, these two layers were integrated to allow anyone with an iPhone and an instrument to start learning musical fundamentals such as majors, timings, minors, and triads.

8. Acknowledgments

We would like to extend our deepest gratitude to Professor Waheed U. Bajwa for guiding us through the challenges we faced along the way. Thank you to the Electrical and Computer Engineering department at Rutgers University for all their help and guidance over the past four years

9. REFERENCES

[1] Rampton, John. “The Benefits of Playing Music Help Your Brain More Than Any Other Activity.” *Inc.com*, Inc., 21 Aug. 2017, www.inc.com/john-rampton/the-benefits-of-playing-music-help-your-brain-more.html.

[2] “Guitar Hero Live The Game.” *Official Site of Guitar Hero*, www.guitarhero.com/game.

[3] “Rock Band.” *Rock Band Rivals*, Harmonix Music Systems, Inc., www.rockband4.com/.

[4] “Learn Piano in a Fun and Easy Way - Piano Apps to Learn How to Play Piano.” *Simply Piano*, JoyTunes, www.joytunes.com/simply-piano.

[5] “How to Play Piano: Learn Piano: Piano Lessons.” *Yousician*, Yousician, yousician.com/piano.

[6] Chen, Xuemei & Liu, Ruolun, “Multiple Pitch Estimation Based on Modified Harmonic Product Spectrum”, ResearchGate, Nov 2013.

[7] Smyth, Tamara, “Music 270a: Signal Analysis”, Department of Music, University of California San Diego, December 2019.

[8] Yuhas, Daisy. “Speedy Science: How Fast Can You React?” *Scientific American*, Scientific American, 24 May 2012, www.scientificamerican.com/article/bring-science-home-reaction-time/.

[9] “Test IOS and Android Audio Latency with Superpowered Latency Test App.” *Superpowered*, Superpowered, superpowered.com/latency.

[10] Sengpiel, Eberhard. *Note Names of Musical Notes* , www.sengpielaudio.com/calculator-notenames.htm.

[11] Apple Inc. “AVFoundation.” *AVFoundation - Apple Developer*, developer.apple.com/av-foundation/.

[12] Goodfellow, Ian et al. *Deep Learning*.. The MIT Press, 2016.

[13] Tsafir, Yoni. “A Look under-the-Hood of Simply Piano (Part 2).” *Medium*, JoyTunes, 20 Dec. 2018, medium.com/joytunes/a-look-under-the-hood-of-simply-piano-part-2-3ba3cafa1bbf.

[14] Jesse H. Engel and Cinjon Resnick and Adam Roberts and Sander Dieleman and Douglas Eck and Karen Simonyan and Mohammad Norouzi, . "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders". *CoRR* abs/1704.01279. (2017).

[15] Curtis Hawthorne and Andriy Stasyuk and Adam Roberts and Ian Simon and Cheng-Zhi Anna Huang and Sander Dieleman and Erich Elsen and Jesse H. Engel and Douglas Eck, . "Enabling

Factorized Piano Music Modeling and Generation with the MAESTRO Dataset". *CoRR* abs/1810.12247. (2018).

[16] Fonseca, Eduardo et al. "Freesound Datasets: a platform for the creation of open audio datasets." *Proceedings of the 18th International Society for Music Information Retrieval Conference (ISMIR 2017)*.

[17] Sajjad Abdoli and Patrick Cardinal and Alessandro Lameiras Koerich, . "End-to-End Environmental Sound Classification using a 1D Convolutional Neural Network". *CoRR* abs/1904.08990. (2019).

[18] "20 Important Benefits of Music In Our Schools." *National Association for Music Education*, 25 Apr. 2018, nafme.org/20-important-benefits-of-music-in-our-schools/.

[19] Kloss, Thomas et al. "The Relationship Between Private Education and Financial Reward – A Pilot Study Case of Private Music Lessons and College Scholarship". *Praxis, the electronic journal of the Sam Houston State University Center for Music Education* 2. (2016).

[20] de Cheveigne, Alain, and Hideki Kawahara. "YIN, a Fundamental Frequency Estimator for Speech and Music". Acoustical Society of America, Apr. 2002.