CS 370 Final Report

Mark Kedzierski

Table of Contents

1 Introduction

In this report I will describe the theory, design, and development of a software-based synthisizer driver meant to act the same manner as the Roland GK-2JH guitar synthisizer driver. The GK-2JK is a hardware unit which mounts on the guitar body, below the strings. It picks up the string vibrations, and converts them into messages which can be understood by a synthesizer. The synthesizer then, in turn, plays the messages using the user-selected instrument. This empowers the guitar player to utilize the sounds of thousands of different instruments just by playing guitar. The applications of this capability include music education, composition, and recording. In general, the driver brings the musician and his computer closer together by enabling communication. Any device (software or hardware) which acts in this manner is considered a synthesizer driver.

The role of the driver is to transcribe monophonic musical melodies in real-time. This problem can be divided into instantaneous pitch recognition, note onset/offset detection, and the dispatch of messages which a synthesizer can understand. The detection of pitch in an audio signal is a problem which has long been studied because of its wide range of applications. It is a difficult one however, because the pitch an audio signal is a subjective attribute which cannot be directly measured. Pitch is defined as the characteristic of a sound which places it at a specific location in the musical scale. The ability to detect it varies among different people. Some are born with “perfect pitch”, which means that they can accurately classify the pitch of a sample sound without any reference. Most people however, can only detect intervals, or the difference in pitch between different sounds. Some suggest that pitch detection might be a learned ability, as different cultures often have different musical scales, each with different intervals. In any case, it is difficult to detect any attribute which is, by definition, subjective.

The first approach I took to solving this problem was to utilize a known pattern recognition technique using neural networks[1]. The basis for this approach was the theory that there existed a common pattern in a sound’s frequency spectrum[2] which could uniquely identify pitch. However, after doing research and performing some experiments I opted for a different method of pitch detection. The new method would determine the fundamental frequency[3] of the audio sample and use it to classify its pitch. This change in approach resulted in an effective real-time algorithm to detect the pitch of digital audio.

The final product of my research is the of a software product entitled Pcm2Midi. It was original developed for the Linux operating system, and was later ported to Windows a little more user-friendliness. The program is a synthesizer driver which converts any input of digitally sampled audio data to MIDI[4] messages which can be relayed to any device. The remainder of this document consists of several

[1] Neural Network -
[2] Frequency Spectrum -
[3] Fundamental Frequency -
[4] Musical Instrument Device Interface -

sections. I have attempted to order the document in such a way that all necessary background theory is presented before its application. This is not always possible so I include footnotes where necessary. The glossary and appendices are also useful references. In the first section I describe those physical properties of sound waves which are relevant to music. Second, I provide a primer in the music theory necessary to understanding the implementation. Third, I discuss the DSP[5] techniques, terms and algorithms I have utilized. In the fourth section I explain the MIDI standard and how it enables digital instruments to communicate. Finally I outline the initial neural-network solution, as well as the algorithms utilized in the final implementation. To close the document I propose future enhancements the product.

In the appendices I include useful references, documented source code, glossary and all my cited references.

2 Physical Properties of Sound Waves

In the section I discuss the physical nature of sound as relevant in the discussion of musical tones and/or melodies. We define *pitch* as any the attribute of sound which defines it's placement on the musical scale. We define a *tone* as any sound which is associated with a pitch. First I offer a concise explanation by Brian C.J. Moore: "Sound originates from the motion or vibration of an object. This motion is impressed upon the surrounding medium (usually air) as a pattern of changes in pressure. What actually happens is that the atmospheric particles, or molecules, are squeezed closer together than normal (called condensation) and then pulled farther apart than normal (called rarefaction)." (Moore, 1999) We obtain a graphical representation of a sound be plotting the amplitude of pressure variation over time. We call this plot a *waveform*. I provide an example below. Note because waveforms are a plot against time, they are said to exist in the *time-domain*.

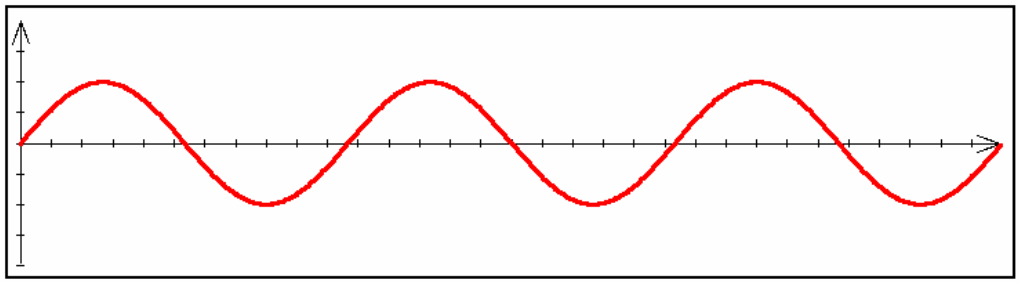

Figure 1: Waveform of a pure tone. Plotted as pressure variation over time.

This waveform is known as a *pure-tone*. Remember that a *tone* is defined as any sound which can be classified as having a pitch. A gunshot is not a tone. A pure tone is named for its very "clean" musical sound, very much similar to that of a tuning fork. A pure tone's waveform is simply a sine wave, whose

[5] Digital Signal Processing -

frequency determines where the note lies on the musical scale. For example, the commonly used A-440 tuning fork produces a wave very similar to a 440 Hz sine wave. We will discuss this in greater detail in the next section.

While pure tones are important in defining pitch and useful in theoretical discussion, we do not encounter them in nature. Instead, we see *complex-tones*. Like pure-tones, all complex tones are assigned a unique pitch value. But instead of consisting of a single sine wave at a given frequency, they contain several superimposed sine waves, all located at *harmonics*. More accurately, a complex tone consists of a *fundamental frequency* and several *overtones.*

We define the i^{th} harmonic of a given frequency as the i^{th} integral multiple of said frequency. For example, the 0^{th}, 1^{st}, and 2^{nd} harmonics of a 440Hz tone would exist at 440Hz, 880Hz, and 1320Hz, respectively. We define the fundamental frequency of a complex tone to be the 0^{th} harmonic. We similarly define any tones which occur at subsequent harmonics as overtones of that fundamental frequency. The next figure depicts sine waves of frequencies at 3^{rd}, 4^{th}, and 5^{th} harmonics combined to form a complex tone.

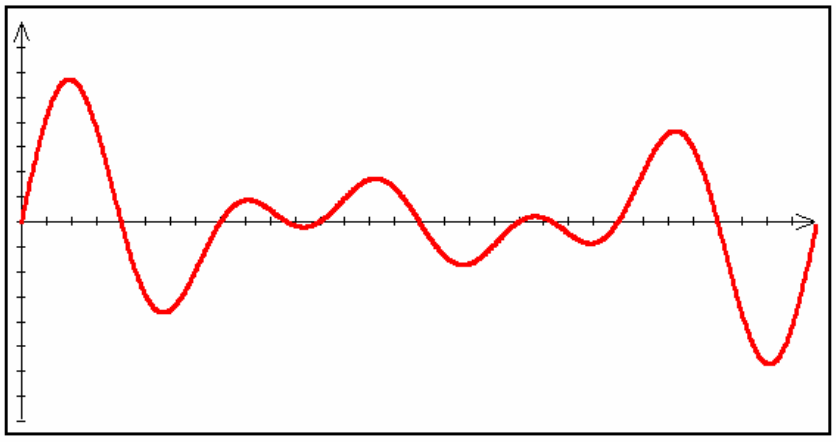

Figure 2: Complex tone containing 3 overtones.

The sound generated from a plucked guitar string is an example of a complex tone. The waveform is shown below. Note how it resembles a damped sin wave, with the amplitude decreasing over time.

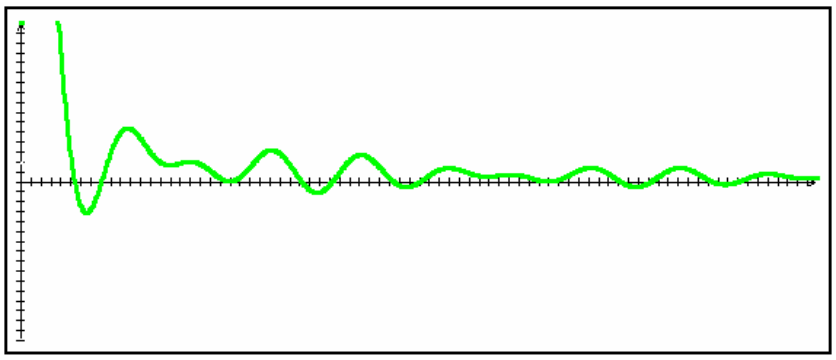

Figure 3: Plucked string waveform. Example of a complex tone.

3 Music Theory Primer

To begin our primer on music theory, we define a *note* as any named tone. The ordered collection of all possible notes is known as the *musical scale*. We define an *interval* as the spatial distance, on the musical scale, between successive notes. Commonly occurring intervals are given names. The smallest interval is the ½-step (or *semitone*), which is used to represent the distance between two adjacent notes on the scale. The Whole-step is the equivalent of two ½-steps. Some other commonly occurring intervals are the Fifth, Major Third and Minor Third. We can now define a *melody* as a series of notes played at varying intervals.

The musical scale is logically divided into groups of twelve notes known as *octaves*. Every octave consists of twelve semitones which are labeled in the following manner: **C, $C^{\#}/D^{b}$, D, $D^{\#}/E^{b}$, E, F, $F^{\#}/G^{b}$, G, $G^{\#}/A^{b}$, A, $A^{\#}/B^{b}$, B.** We use this scheme because each of these labels represents a unique "sound", only twelve of which exists. This means that a **C** note sounds the "same" as a **C** note in any other octave. Each octave in the musical scale is labeled by an index, starting at 0. In turn, every note on the musical scale can be uniquely identified by a note label and octave index. Examples are: C_0, D^{b}_5, and G_2. Middle **C** is represented as C_4. So, if we know the pitch of a tone, we know where it belongs on the musical scale. But since real instruments don't make pure tones, we need a way to assign pitch. "Since pitch is a subjective attribute, it cannot be measured directly. Assigning a pitch value to a sound is generally understood to mean specifying the frequency of a pure tone having the same subjective pitch as a sound." (Moore, 1999). We define the *relative frequency* $f(\eta)$ of any note η as the frequency of said pure tone. For example, $f(C_4) = 261.4Hz$. Next we discuss how the musical scale maps to the human audible frequency range.

The human audible frequency range is approximately 20-20,000Hz. However, we are not concerned with anything above 5Khz, as we are unable to discern the pitch of frequencies in the that range. (Moore, 1999) The range of 0-5kHz is partitioned into eight octaves, each beginning at **C**, where $f(C_0)=16.4Hz$, $f(C_1)=32.7Hz$, $f(C_8)=4186Hz$. It may be clear now why C_4 is known as middle **C**. You may notice that $f(C_1) = 2f(C_0)$. This leads us to the real definition of the octave as the interval between two tones when their frequencies are in a ratio 2:1 (Moore, 1997) We can deduce from this that the eight octaves are not the same size, instead they increase exponentially. Every interval, in fact, is actually a relationship between two tones. The following table depicts common intervals and their associated ratios.

Interval	Ratio
½ step	1:1.05946
Fifth	3:2
Maj Third	5:4
Min Third	6:5

4 Digital Signal Processing

Digital Signal Processing refers to the techniques and algorithms used to process and or transform digitally sampled data. In our case, we are referring to a stream if digital audio samples. A digital audio sample is the amplitude of pressure variation in a single instant of time. We can obtain a digital waveform by taking digital audio samples at fixed time intervals. We define *sampling rate* as samples/sec. Below are examples of digitally sampled waveform of a pure tone and a plucked guitar string.

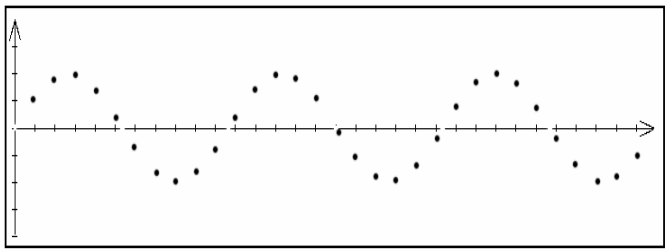

Figure 4: Digitally sampled waveform of pure tone.

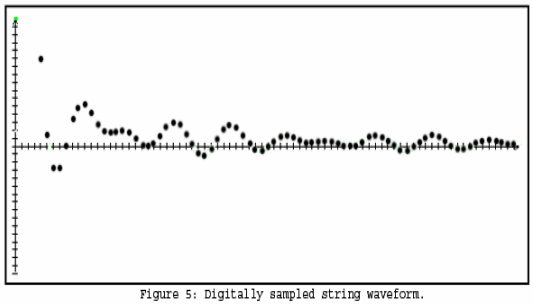

Figure 5: Digitally sampled string waveform.

To this point all the data we have discussed has existed in the time-domain. In DSP we often find it useful to view data in the *frequency domain*. In the frequency domain, we use frequency as the independent variable. Instead of a waveform, in the frequency domain we use a *frequency-spectrum* to visualize data. Below is a frequency spectrum of a plucked guitar string taken from the Pcm2Midi software.

Figure 6: Frequency spectrum. Blue line is the fundamental frequency.

In order to obtain the frequency sprectrum of a particular waveform we use a technique known as the *Fourier transform*. The technique is based on the fact that any periodic signal can be decomposed into sinusoidal waves of varying frequencies, amplitudes. There is actually a family of Fourier transforms, all for different kinds of input. The one we are interested in is the *Discrete Fourier Transform*, which operates on discrete, non-periodic signals. (Smith, 1997) The DFT takes as input an **N** point signal **X**. This array represents **N** digital audio samples. As output, there are **2(N-1)** point signals **Re** and **Im**. These represent the amplitudes of the the component cosine and sin waves, also known as “real” and “imaginary”

parts of output. These component waves are known as *basis functions* and **k**th functions are defined as follows:

$$C_k[i] = \cos(2\pi * ki/N)$$

$$S_k[i] = \sin(2\pi * ki/N)$$

The following depicts some basis sinusoids to help you visualize.

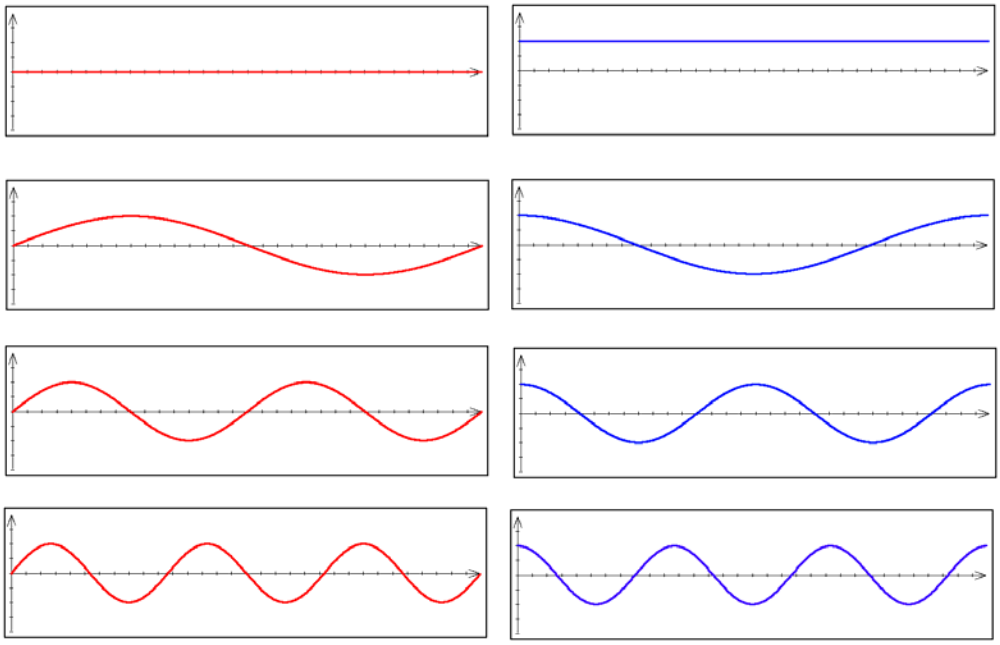

Figure 7: Basis sinusoids from k=0 to k=3.

We can re-obtain the waveform from the frequency spectrum by applying the *Inverse Fourier Transform*. The IFT is implemented by simply synthesizing the basis functions:

$$X[i] = \sum Re[k] \cos(2\pi ki/N) + \sum Im[k] \sin(2\pi ki/N)$$

Once we use the DFT to obtain the real and imaginary parts of the input, we still need one more step to obtain a frequency spectrum like the one presented above. We need to convert from rectangular to polar coordinates. Polar coordinates consist of the Magnitude and Phase of **X**. We only care about the magnitude and use the following equation to obtain it.

$$Mag[k]=(Re[k]^2 + Im[k]^2)^{1/2}$$

Plotting the magnitude against the frequency we can now obtain the frequency spectrum. To initially obtain the real and imaginary parts of the input, we use the Fast Fourier Transform, which is an efficient algorithm easily implemented by computer programs.

If we are using this method to analyze real-time incoming data, we will get a quite noisy result. This is because there is never enough data to produce a clean spectrum. To solve this, we isolate the middle sections of the signal by multiplying the signal by a *Hamming Window* or *Blackman Window*. This amplifies the peaks and clarifies the frequency spectrum. A Blackman window is depicted below.

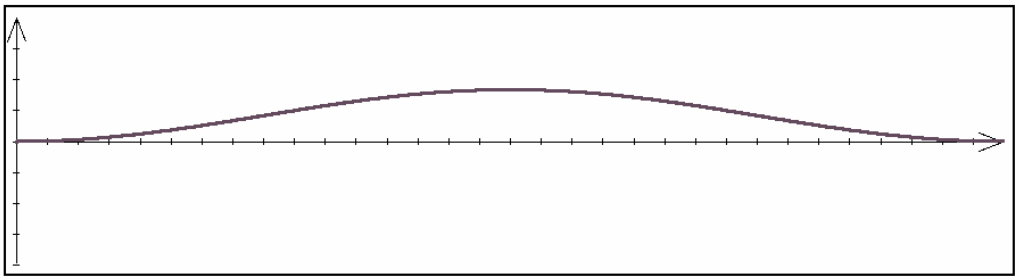

Figure 8: Blackman window.

To implement this kind of processing in real-time, we need to process the incoming wave data as several arrays of **N** samples. We need to make sure that we use enough samples to obtain the desired spatial and temporal resolution in the frequency spectrum. For example, if we use a 4096 sample input signal, we obtain 2048 band frequency spectrum. If we are using a 48,000 Hz sampling rate, we obtain a frequency bandwidth of 23.4Hz. This means that the smallest interval we can detect is 23.4Hz. This is adequate for our application. We also need to make sure that we have a good enough temporal resolution. For the same 48,000Hz sampling rate, can obtain 11.7 frequency spectra per second. This poses a problem! We need to be able to detect changes in frequency faster than that. We can improve this performance by overlapping the windows. For example, we calculate a FFT every 2048 samples, and still use 4096 samples for each calculation. In conclusion, with a 48,000Hz sampling rate, using these methods we can obtain the frequency peaks of a real-time signal with a spatial resolution of 23.4Hz and temporal resolution of 22.7 Hz. This is adequate for the transcription of a monophonic musical melody.

5 MIDI

The Musical Instrument Device Interface is widely used by musicians to communicate between digital musical instruments. It is a standard which has been agreed upon by major manufacturers in the industry. It connects not only musical instruments, but all devices such as recorders and lighting systems.

The communication in a MIDI system is done by sending and receiving messages. Examples of some messages related to melody composition are Note-On, Note-Off, Tempo Change, Key-Signature Change, and Time-Signature change. The meanings behind these events are self-explanatory. Each message contains variables specific to the message. For example, a Note-On event would contain information about which note should be played, and the volume at which it should be played. Each note is assigned a specific index. These indices are standard and are listed in Appendix A. The Key-Signature message would contain the new desired key signature. A summary of MIDI events is listed in Appendix B at the end of this document.

In practice a synthesizer driver would create and dispatch a series of MIDI messages to a synthesizer. The synth would then playback the desired notes using pre-recorded instrument audio samples.

6 Neural Network Pattern Recognition

The initial approach I took was based the following theory: “Pitch is related to some weighted average of all the components … the way in which this average is taken is complex” (Whitefield, 1970) Since neural-networks are used to find such complex relationships, I set out to utilize them to recognize pitch in digital audio signals. I created a neural network very similar to one commonly used to visually recognize handwritten numerical digits. The input layer would consist of **N** nodes for **N** frequency components. There would then be a hidden layer of some variable size. The output layer would contain one node for each possible note value. The network, of course, would have to be trained to recognize every possible note. This posed an initial problem, so I limited my work to a single octave. After training, I obtained an accuracy of approximately 70%. This was definitely not good enough, so I opted for the more direct approach of finding the fundamental frequency component of the audio data.

7 Implementation

Application Window

Below is a screenshot of the main application window running under KDE with important parts labelled.

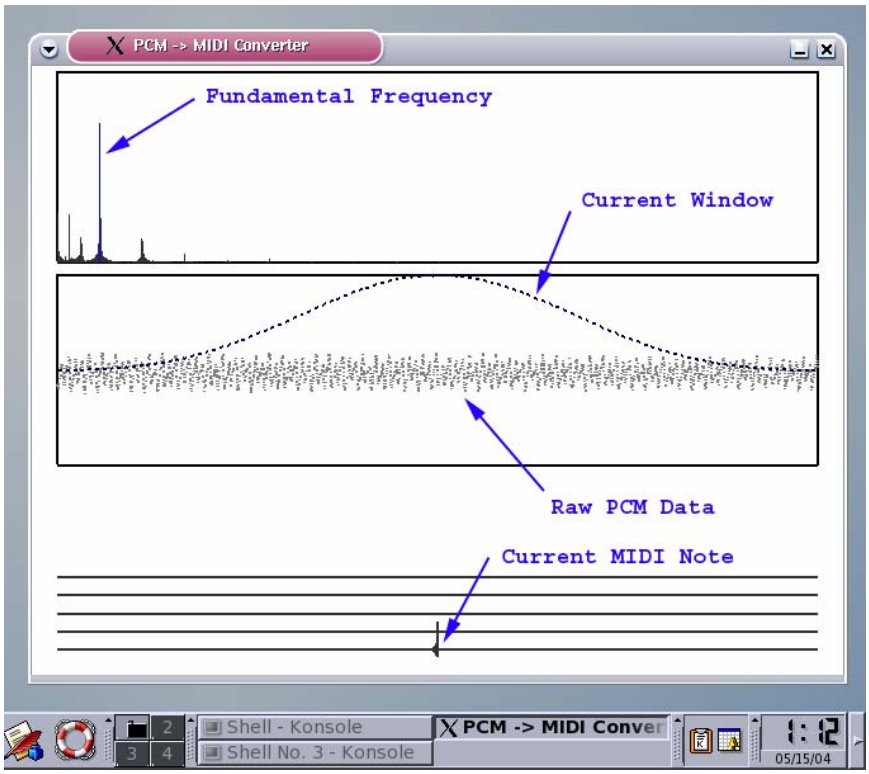

Figure 9: Main Application window.

Call-back Structure

We implement the processing of digital audio using a call-back function and a worker thread. As the audio input is received by the sound card, it is sent to the operating system and handled by an audio server. In the case of Linux, the audio server is JACK In any case, the audio server caches the audio data in a buffer of 4096 points. Every time such a buffer is filled, it invokes the call-back functions of any registered applications and passes them the array of audio samples. The application, upon entering the call-back, stores the raw audio data in a synchronized data structure for processing by the worker thread.

The worker thread constantly checks the synchronized data structure. The thread takes action every time it receives 2048 new audio samples. When this occurs, the thread shifts over the data from the last call-back and inserts the new samples. This effectively implements two overlapping windows. Next it calculates the fundamental frequency and checks for any new note events; a note event is either a new note,

or the end of a note. Upon the occurrence of a note event, it dispatches the appropriate MIDI messages back to the audio server. From there they can be relayed to any source.

Calculation of Fundamental Frequency

Given a 2048 point frequency spectrum, we calculate the fundamental frequency using the following algorithm, which is devised based on guidance I received from Russell Pinkston, University of Texas. First, we locate peaks in the frequency spectrum by locating areas of zero slope. Next we increase the amplitude of each peak by adding to it the amplitude of its harmonics. The largest resultant peak is located at the fundamental frequency. This works on the assumption that the fundamental frequency is the one which has the most harmonics present. The index of the fundamental in the frequency spectrum is multiplied by the frequency bandwidth to obtain a value in Hertz.

Detection of Note Events

Once we obtain a valid fundamental frequency in Hertz, we can proceed to detect the onset or offset of musical notes in the incoming data. At all times a monophonic melody there are two possible states. A single note is either currently ringing, or there is silence. The following is a pseudo-code description of the algorithm. Let f represent the current fundamental frequency, f_{old} the previous iteration's fundamental frequency, and *threshold* as some predefined constant.

begin_loop

if note_is_ringing

if $f = f_{old}$ and the amplitude(f) < *threshold* send note-off(f)
//The note has faded away

if $f = f_{old}$ and the amplitude(f) > amplitude(f_{old}) then send note-off(f_{old}) and note-on(f_{old}) //A same note re-occurred

if f != f_{old} then send note-off(f_{old}) and note-on(f)
//A different note occured

else

if f is valid then send note-on(f)
//A new note occured

end_loop

Responding to Note Events

After we have deciphered exactly what has happened, we respond by sending the appropriate MIDI message. But first, we need to translate the fundamental frequency to a MIDI note index. We do this by hashing the frequency in to a pre-filled array of MIDI note values. See Appendix A for the MIDI Note → Hertz translation table. We send can send the amplitude at the fundamental along with the message for more accurate translation.

8 The Future…

This project shows the basis for a simple, yet effective program which transcribes monophonic melodies in real-time. Improvements would include more detailed note events, which contain information such as vibrato and effects. The detection of bends and slides could also easily be implemented. The final goal would be the transcription of polyphonic melodies. I suggest that such algorithms would be based on locating intervals in frequency spectra.

Glossary

Basis Functions Component sinusoids which compose any complex tone. See Fourier Transforms.

Complex Tone A tone composed of a number of sinusoids of different frequencies (Moore, 1997)

Rarefaction The expansion of molecules as a sound wave travels through a medium.

Digital Signal Processing (DSP) Techniques/Algorithms used to process and/or transform digitally sampled data. In our case, we are referring to a stream if digital audio samples.

Fourier Transform (FT) A transform from the time domain into the frequency domain.

Discrete Fourier Transform (DFT) Implementation of Fourier transform which acts upon discrete, non-periodic signals

Frequency For a sine wave the frequency is the number of periods occurring in one second. The unit is cycles per second, or hertz (Hz). (Moore, 1997)

Frequency Domain Domain in which data is plotted against frequency

Frequency Spectrum A plot of amplitudes of various frequency bands

Fundamental Frequency Fundamental frequency of a periodic sound is the frequency of that sinusoidal component of the sound that has the same period as the periodic sound. (Moore, 1997)

Harmonic Component of a complex tone whose frequency is an integral multiple of the fundamental frequency of the complex tone.(Moore, 1997)

Interval Referring to musical interval, or the spatial relationship between two notes. Examples of common intervals are an octave, ½ step, full step, a fifth, major third, and minor third.

MIDI Musical Instrument Digital Interface. Provides a standard digital representation of various musical events. Examples of events are note onset, note offset, tempo change, key signature change, and time signature change. Individual events are encoded and sent as messages. Messages can include several variables, specific to the message type. For example, a note onset message contains information about the note and its velocity, tremolo, and reverb. MIDI is used as a composition tool, or to communicate between digital instruments such as synthesizers.

Octave Interval between two tones when their frequencies are in a ratio 2:1 (Moore, 1997)

Overtone Let $f(\eta)$ represent the frequency of pure tone η. A pure tone ν is an overtone of η if $f(\nu)$ is an integral multiple of $f(\eta)$.

Periodic Sound A sound whose waveform repeats itself regularly as a function of time. (Moore, 1997)

Pitch Attribute of auditory sensation in terms of which sounds may be ordered on a musical scale. (Moore, 1997)

Pure Tone A sound wave whose instantaneous pressure variation as a function of time is a sinusoidal function. Also called a simple tone. (Moore, 1997)

Rarefaction The expansion of molecules as a sound wave travels through a medium.

Relative Frequency The relative frequency of a periodic sound refers to the frequency of the pure tone with the same pitch.

Soundfont© A file containing a database of instruments, each having audio samples associated with MIDI notes. Can be loaded by a synthesizer for playback.

Synthesizer Instrument used to play or record MIDI compositions. This can be a digital unit, or a piece of software. To play a MIDI composition, pre-recorded audio samples are associated with MIDI notes.

Synth Driver Sends MIDI messages to a synthesizer for recording or playback. Can be digital unit, such as an electronic keyboard, or Software.

Time Domain Domain in which the data is plotted over time.

Tone A tone is a sound wave capable of evoking an auditory sensation having pitch. (Moore, 1997)

Waveform Term used to describe the form or shape of a wave. It may be represented graphically by plotting instantaneous amplitude of pressure as a function of time. (Moore, 1997)

Appendix A:

MIDI to Equal Temperament Semitone Conversion Table

12	C	16.4	36	C	65.4	60	C	261.6	84	C	1046.5	108	C	4186.0	na	C	16744.0
13	C#	17.3	37	C#	69.3	61	C#	277.2	85	C#	1108.7	109	C#	4434.9	na	C#	17739.7
14	D	18.4	38	D	73.4	62	D	293.7	86	D	1174.7	110	D	4698.6	na	D	18794.5
15	D#	19.4	39	D#	77.8	63	D#	311.1	87	D#	1244.5	111	D#	4978.0	na	D#	19912.1
16	E	20.6	40	E	82.4	64	E	329.6	88	E	1318.5	112	E	5274.0	na	E	21096.2
17	F	21.8	41	F	87.3	65	F	349.2	89	F	1396.9	113	F	5587.7	na	F	22350.6
18	F#	23.1	42	F#	92.5	66	F#	370.0	90	F#	1480.0	114	F#	5919.9	na	F#	23679.6
19	G	24.5	43	G	98.0	67	G	392.0	91	G	1568.0	115	G	6271.9	na	G	25087.7
20	G#	26.0	44	G#	103.8	68	G#	415.3	92	G#	1661.2	116	G#	6644.9	na	G#	26579.5
21	A	27.5	45	A	110.0	69	A	440.0	93	A	1760.0	117	A	7040.0	na	A	28160.0
22	A#	29.1	46	A#	116.5	70	A#	466.2	94	A#	1864.7	118	A#	7458.6	na	A#	29834.5
23	B	30.9	47	B	123.5	71	B	493.9	95	B	1975.5	119	B	7902.1	na	B	31608.5
24	C	32.7	48	C	130.8	72	C	523.3	96	C	2093.0	120	C	8372.0	na	C	33488.1
25	C#	34.6	49	C#	138.6	73	C#	554.4	97	C#	2217.5	121	C#	8869.8	na	C#	35479.4
26	D	36.7	50	D	146.8	74	D	587.3	98	D	2349.3	122	D	9397.3	na	D	37589.1
27	D#	38.9	51	D#	155.6	75	D#	622.3	99	D#	2489.0	123	D#	9956.1	na	D#	39824.3
28	E	41.2	52	E	164.8	76	E	659.3	100	E	2637.0	134	E	10548.1	na	E	42192.3
29	F	43.7	53	F	174.6	77	F	698.5	101	F	2793.8	125	F	11175.3	na	F	44701.2
30	F#	46.2	54	F#	185.0	78	F#	740.0	102	F#	2960.0	126	F#	11839.8	na	F#	47359.3

31	G	49.0	55	G	196.0	79	G	784.0	103	G	3136.0	127	G	12543.9	na	G	50175.4
32	G#	51.9	56	G#	207.7	80	G#	830.6	104	G#	3322.4	na	G#	13289.8	na	G#	53159.0
33	A	55.0	57	A	220.0	81	A	880.0	105	A	3520.0	na	A	14080.0	na	A	56320.0
34	A#	58.3	58	A#	233.1	82	A#	932.3	106	A#	3729.3	na	A#	14917.2	na	A#	59669.0
35	B	61.7	59	B	246.9	83	B	987.8	107	B	3951.1	na	B	15804.3	na	B	63217.1

Appendix B:

Summary of MIDI Messages

MIDI 1.0 Specification Message Summary

Updated 1995 By the MIDI Manufacturers Association

WARNING: The details of this implementation could dramatically affect compatibility with other products. It is recommended that you consult the official MMA detailed specification for any additional information.

Status D7----D0	Data Byte(s) D7----D0	Description
Channel Voice Messages		
1000cccc	0nnnnnnn 0vvvvvvv	Note Off event. This message is sent when a note is released (ended). (nnnnnnn) is the note number. (vvvvvvv) is the velocity.
1001cccc	0nnnnnnn 0vvvvvvv	Note On event. This message is sent when a note is depressed (start). (nnnnnnn) is the note number.
1010cccc	0nnnnnnn 0vvvvvvv	Polyphonic Key Pressure (Aftertouch). This message is sent when the pressure (velocity) of a previously triggered note changes. (nnnnnnn) is the note number. (vvvvvvv) is the new velocity.
1011cccc	0nnnnnnn 0vvvvvvv	Control Change. This message is sent when a controller value changes. Controllers include devices such as pedals and levers. Certain controller numbers are reserved for specific purposes. See Channel Mode Messages. (ccccccc) is the controller number. (vvvvvvv) is the new value.
1100cccc	0ppppppp	Program Change. This message sent when the patch number changes.

		(ppppppp) is the new program number.
1101nnnn	0ccccccc	Channel Pressure (After-touch). This message is sent when the channel pressure changes. Some velocity-sensing keyboards do not support polyphonic after-touch. Use this message to send the single greatest velocity (of all the current depressed keys). (ccccccc) is the channel number.
1110nnnn	0lllllll 0mmmmmmm	Pitch Wheel Change. This message is sent to indicate a change in the pitch wheel. The pitch wheel is measured by a fourteen bit value. Center (no pitch change) is 2000H. Sensitivity is a function of the transmitter. (llllll) are the least significant 7 bits. (mmmmmm) are the most significant 7 bits.
Channel Mode Messages (See also Control Change, above)		
1011nnnn	0ccccccc 0vvvvvvv	Channel Mode Messages. This the same code as the Control Change (above), but implements Mode control by using reserved controller numbers. The numbers are: Local Control. When Local Control is Off, all devices on a given channel will respond only to data received over MIDI. Played data, etc. will be ignored. Local Control On restores the functions of the normal controllers. c = 122, v = 0: Local Control Off c = 122, v = 127: Local Control On All Notes Off. When an All Notes Off is received, all oscillators will turn off. c = 123, v = 0: All Notes Off (See text for description of actual mode commands.) c = 124, v = 0: Omni Mode Off c = 125, v = 0: Omni Mode On c = 126, v = M: Mono Mode On (Poly Off) where M is the number of channels (Omni Off) or 0 (Omni On) c = 127, v = 0: Poly Mode On (Mono Off) (Note: These four messages also cause All Notes Off)

System Common Messages		
11110000	0iiiiiii 0ddddddd 0ddddddd 11110111	System Exclusive. This message makes up for all that MIDI doesn't support. (iiiiiii) is a seven bit Manufacturer's I.D. code. If the synthesizer recognizes the I.D. code as its own, it will listen to the rest of the message (ddddddd). Otherwise, the message will be ignored. System Exclusive is used to send bulk dumps such as patch parameters and other non-spec data. (Note: Real-Time messages ONLY may be interleaved with a System Exclusive.)
11110001		Undefined.
11110010	0lllllll 0mmmmmmm	Song Position Pointer. This is an internal 14 bit register that holds the number of MIDI beats (1 beat= six MIDI clocks) since the start of the song. l is the LSB, m the MSB.
11110011	0sssssss	Song Select. The Song Select specifies which sequence or song is to be played.
11110100		Undefined.
11110101		Undefined.
11110110		Tune Request. Upon receiving a Tune Request, all analog synthesizers should tune their oscillators.
11110111		End of Exclusive Used to terminate a System Exclusive dump (see above).
System Real-Time Messages		
11111000		Timing Clock. Sent 24 times per quarter note when synchronization is required (see text).
11111001		Undefined.
11111010		Start. Start the current sequence playing. (This message will be followed with Timing Clocks).
11111011		Continue. Continue at the point the sequence was Stopped.
11111100		Stop. Stop the current sequence.
11111101		Undefined.
11111110		Active Sensing. Use of this message is optional.

		When initially sent, the receiver will expect to receive another Active Sensing message each 300ms (max), or it will be assume that the connection has been terminated. At termination, the receiver will turn off all voices and return to normal (non-active sensing) operation.
11111111		Reset. Reset all receivers in the system to power-up status. This should be used sparingly, preferably under manual control. In particular, it should not be sent on power-up.

Appendix C:

Complete Documented Source:

Source Files	Contents
fft.c	- Digital Signal Processing algorithms.
midi.c	- Dispatching MIDI messages
Pcm2Midi.c	- Main program initialiazation and loop

fft.c :

```
#include <math.h>
#include "fft.h"

/* fft.c
 * Implementation of Fast Fourier Transform, Inverse Fourier
Transforms,
 * Rectangular to Polar coordinate conversion, and fundamental
frequency
 * locater algorithm.
 */

/*
 * Convert n-length signal from Rectangular coordinates to polar
coordinates.
 * n -> number of points in input
 * x -> "real" part of input
 * y -> "imaginary" part of input
 * z -> polar magnitude of ouput.
 */
void Rec_To_Polar( int n, float *x, float *y, float *z ) {
  int i;
  for( i=0; i<n; i++ )
    z[i] = powf( x[i]*x[i] + y[i]*y[i], 0.5 );
}

/*
 * Calculate Fast Fourier Transform
 * re -> "real" part of input
 * im -> "imaginary" part of input
 */
void fft( int m, float *re, float *im ) {
  int i,ip,n, nm1, nd2, j, jm1, k, l, le, le2;
  float tr, ti, ur, ui, si, sr;
```

```
  n = 1<<m;
  nm1 = n-1;
  nd2 = n/2;
  j = n/2;

  for( i=1; i<=n-2; i++ )
  {
    if( i<j )
    {
      tr = re[j];
      ti = im[j];
      re[j] = re[i];
      im[j] = im[i];
      re[i] = tr;
      im[i] = ti;
    }
    k = nd2;
    while( k<=j )
    {
      j-=k;
      k/=2;
    }
    j+=k;
  }

  for( l=1; l<=m; l++ )
  {
    le = 1<<l;
    le2 = le/2;
    ur = 1;
    ui = 0;
    sr = cosf(M_PI/le2);
    si = -sinf(M_PI/le2);
    for( j=1; j<=le2; j++ )
    {
      jm1 = j-1;
      for( i=jm1; i<=nm1; i+=le )
      {
        ip = i+le2;
        tr = re[ip]*ur - im[ip]*ui;
        ti = re[ip]*ui + im[ip]*ur;
        re[ip] = re[i]-tr;
        im[ip] = im[i]-ti;
        re[i] += tr;
        im[i] += ti;
      }
      tr = ur;
      ur = tr*sr - ui*si;
      ui = tr*si + ui*sr;
    }
  }
}

/* Inverse FFT
 * re -> "real" part of input
 * im -> "imaginary" part of input
 */
```

```
void inverse_fft( int m, float *re, float *im ) {
  int i, n;
  n = 1<<m;

  for( i=0; i<n; i++ )  //change sign of im[]
      im[i] = -im[i];
  fft( m, re, im );  //calculate forward fft
  for( i=0; i<n; i++ ) //divide the time domain by n and change
  {               //sign of im[]
      re[i] = re[i]/n;
      im[i] = -im[i]/n;
  }
}

/* Print out contents of array */
void dump_array( int size, float *x ) {
  int i;
  for( i=0;i<size;i++ )
      printf("x[%d]= %f\n",i,x[i]);
}

/*
 * Return Fundamental Frequency of data
 */
int find_peak( int size, float *data ) {
  int i,k=0;
  float x=-1;
  for( i=1; i<size; i++ )
  {
     if( data[i-1] < data[i] && data[i] > data[i+1] )
     {
      data[i]+=data[i+i];
      if( data[i] > x ) { x = data[i]; k = i; }
     }
  }
  return k;
}
```

midi.c :

```
#include "midi.h"

/*
 * Defines methods for sending MIDI messages
 */

/*
 * Opens ALSA MIDI Sequencer
 */
int open_seq( snd_seq_t **seq_handle)
{
      int portid;
```

```
      if( snd_seq_open( seq_handle, "default", SND_SEQ_OPEN_OUTPUT, 0 )
< 0) {
            fprintf( stderr, "Error opening ALSA seqeuncer.\n" );
            exit(1);
      }
      snd_seq_set_client_name( *seq_handle, "PCM -> MIDI" );
      if ((portid = snd_seq_create_simple_port(*seq_handle, "Output",
            SND_SEQ_PORT_CAP_READ|SND_SEQ_PORT_CAP_SUBS_READ,
            SND_SEQ_PORT_TYPE_APPLICATION)) < 0) {
      fprintf(stderr, "Error creating sequencer port.\n");
      exit(1);
      }
      return portid;
}

/*
 * Sends NOTEOFF Message
 * midiNum -> MIDI note index
 * seq_handle -> ALSA Sequencer
 * port -> MIDI port to use
 * channels -> array of channels to use
 */
void endMidiNote( int midiNum, snd_seq_t *seq_handle, int port, char
*channels )
{
      snd_seq_event_t ev;
      int i;
      for( i=0; i<16; i++ )
        if( channels[i]==1 )
        {
            snd_seq_ev_set_noteoff( &ev, i, midiNum, 127 );
            snd_seq_ev_set_source( &ev, port );
            snd_seq_ev_set_subs( &ev );
            snd_seq_ev_set_direct( &ev );
            snd_seq_event_output_direct( seq_handle, &ev );
        }
}

/*
 * Sends NOTEON Message
 * midiNum -> MIDI note index
 * seq_handle -> ALSA Sequencer
 * port -> MIDI port to use
 * channels -> array of channels to use
 */
void sendMidiNote( int midiNum, snd_seq_t *seq_handle, int port, char
*channels )
{
      snd_seq_event_t ev;
      int i;
      for( i=0; i<16; i++ )
      {
          if( channels[i]==1 )
          {
            snd_seq_ev_set_noteon( &ev, i, midiNum, 127 );
            snd_seq_ev_set_source( &ev, port );
            snd_seq_ev_set_subs( &ev );
```

```
            snd_seq_ev_set_direct( &ev );
            snd_seq_event_output_direct( seq_handle, &ev );
          }
      }
}
```

Pcm2Midi.c :

```
/** @file Pcm2Midi.c
 *
 * @brief Real Time PCM -> MIDI converter
 * @author Mark Kedzierski
 * Some code is copied from JACK example source code & nobugs.org.
Thanks.
 */

#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
#include <GL/gl.h>
//#include <GL/glu.h>
#include <jack/jack.h>
#include <jack/ringbuffer.h>
#include <alsa/asoundlib.h>
#include "fft.h"
#include "midi.h"

/* JACK-related definitions */
jack_nframes_t sr;            // Current sample rate
jack_nframes_t bs;            // Current buffer size
typedef jack_default_audio_sample_t sample_t;
jack_port_t *input_port;      // Audio Input Port
jack_ringbuffer_t *rb;        //Ring Buffer for Synchronized DATA
access

/* Application specific definitions */
const char *mainCaption = "PCM -> MIDI Converter";
short graphics = 0;           //Graphics Display Toggle
short displayWindow = 0;      //Displaying Windowed PCM Data
const int width = 640;        //Window Width
const int height = 480;       //Window Height
const int rb_size = 32768;    //# of Frames Inside Ringbuffer
size_t dump_size;             //Size of Each Ringbuffer Dump.
sample_t *raw_pcm_data;       //Stores Raw PCM data
sample_t *win_pcm_data;       //Stores windowed PCM data
sample_t *freq_data;          //Stores frequency data
sample_t *re;                 //used for FFT
sample_t *im;                 //used for FFT
sample_t freq_band;           //Bandwith of Frequency Range
```

```
sample_t gain = 1.0;          //Master Gain
sample_t midi_notes[128];     //Store MIDI note hertz values
enum Window { Hamming, Blackman };  //Available Windows
sample_t *currentWindow;      //Pointer to current window
sample_t *hammingWindow;      //Hamming window
sample_t *blackmanWindow;     //Blackman window

/* SDL-related definitions */
SDL_Thread *event_thread;     //Event-Handler
SDL_Thread *worker_thread;    //Main thread
void eventHandler(void);
void worker_func(void);
int appIsRunning = 1;

/* ALSA-related definitions */
snd_seq_t *seq_handle;        //MIDI Sequencer
int alsa_port;                //MIDI Output port
char channels[16];            //MIDI Output Channels [boolean array]
int current_ch = 0;           //Current MIDI Channel

/* OpenGL-related definitions */
float desired_framerate = 70.0;  //Desired Graphics Refresh Rate

/* Midi-related defintions */
short hertz_to_midi( float hz);     //Converts Hz to MIDI note
short midi_to_line(short midi); //Returns which Treble Clef line the
note is on
short midi_to_sharp(short midi); //Determines whether note is Sharp or
Flat

/**
 * The process callback for this JACK application.
 * It is called by JACK at the appropriate times.
 */
int
process (jack_nframes_t nframes, void *arg) {
   sample_t *in = (sample_t *) jack_port_get_buffer (input_port,
nframes);

   if( jack_ringbuffer_write_space(rb) > dump_size )
   {
      jack_ringbuffer_write( rb, (char*)in, dump_size );
   }
  return 0;
}

/**
 * This is the shutdown callback for this JACK application.
 * It is called by JACK if the server ever shuts down or
 * decides to disconnect the client.
 */
void
jack_shutdown (void *arg) {
      appIsRunning = 0;
      exit (1);
}
```

```
int
main (int argc, char *argv[]) {
      jack_client_t *client;
      const char **ports;
      const char *name = "PCM->MIDI";
      short connect_ports = 0;

      /* try to become a client of the JACK server */

      if ((client = jack_client_new (name)) == 0) {
            fprintf (stderr, "jack server not running?\n");
            return 1;
      }

      printf( "\nPCM -> MIDI Converter\n--------------------\n\n" );

      /* Process command-line Arguments */
      int t=0;
      for( t=0; t<argc; t++ )
      {
            char *c = argv[t];
            switch( *c )
            {
                  case 'h':
                        printf("\nUsage: Pcm2Midi [h] [g] [c]
[w]\n\nOptions:\ng  -  Graphics Enabled\nc  -  Connect Capture Port\nw
-  Display Windowed PCM Data\n");
                        exit(0);
                  case 'g':
                        printf("Graphics are Enabled\n");
                        graphics = 1;
                        break;
                  case 'c':
                        printf("Connecting to Capture Port\n");
                        connect_ports = 1;
                        break;
                  case 'w':
                        printf("Displaying Windowed PCM Data.");
                        displayWindow = 1;
                        break;
            }
      }
      printf("Initializing...\n");
      /* tell the JACK server to call `process()' whenever
         there is work to be done.
      */
      printf("Setting Jack server process callback...");
      jack_set_process_callback (client, process, 0);
      printf("OK\n");
      /* tell the JACK server to call `jack_shutdown()' if
         it ever shuts down, either entirely, or if it
         just decides to stop calling us.
      */
      printf("Setting Jack server on_shutdown...");
      jack_on_shutdown (client, jack_shutdown, 0);
      printf("OK\n");
```

```
	printf("Jack server details:\n");
	/* Determine statistics and print them */
	sr=jack_get_sample_rate (client);
	bs=jack_get_buffer_size (client);
	dump_size=bs*sizeof(sample_t);

	/* Create the ringbuffer for synchronized data access */
	printf("Creating ring buffer...\n");
	rb = jack_ringbuffer_create( sizeof( sample_t ) * rb_size );

	printf ("Sample Rate: %" PRIu32 "\n",
		sr);
	printf ("Buffer Size: %" PRIu32 "\n",
		bs);
	printf ("Ringbuffer Size: %" PRIu32 "\n",
		jack_ringbuffer_write_space(rb));
	printf ("Ringbuffer Dump Size: %" PRIu32 "\n",
		dump_size);

	/* allocate memory for pcm data */
	printf("\nAllocating memory for raw pcm data...\t");
	raw_pcm_data = (sample_t*) malloc( dump_size * 2 );
	memset( raw_pcm_data, 0, dump_size * 2);
	win_pcm_data = (sample_t*) malloc( dump_size * 2 );
	memset( win_pcm_data, 0, dump_size * 2);
	printf("OK\n");

	/* allocate memory for frequency data */
	printf("Allocating memory for frequency data...\t");
	freq_data = (sample_t*) malloc( dump_size );
	memset( freq_data, 0, dump_size );
	re = (sample_t*) malloc( dump_size * 2);
	im = (sample_t*) malloc( dump_size * 2);
	printf("OK\n");

	/* Create windows */
	printf("Allocating memory for windows...\t");
	hammingWindow = (sample_t *) malloc( dump_size * 2);
	blackmanWindow = (sample_t *) malloc( dump_size * 2);
	printf("OK\n");
	printf("Initializing windows...\t");
	int i;
	for( i=0; i<(bs*2); i++ )
	{
		hammingWindow[i] = 0.54 - 0.46*cos(2*M_PI*i/(bs*2));
		blackmanWindow[i] = 0.42 -
0.5*cos(2*M_PI*i/(bs*2))+0.08*cos(4*M_PI*i/(bs*2));
	}
	/* Set current window to Blackman as default */
	currentWindow = blackmanWindow;
	printf("OK\n");

	/* Calculate Frequency Bandwidth */
	freq_band = (float)sr/(bs*2);

	printf("Generating MIDI notes...");
	/* Generate MIDI notes */
```

```
      int k;
      sample_t ratio = powf( 2, 1.0/12.0 );
      memset( midi_notes, 0, 12 );
      midi_notes[12] = 16.4;  //MIDI note # 12, C
      for( k=13; k<128; k++ )
            midi_notes[k] = midi_notes[k-1]*ratio;
      printf("OK\n");

      /* create one input & one output port */
      printf("Registering Jack input port...");
      input_port = jack_port_register (client,
                  "Input", JACK_DEFAULT_AUDIO_TYPE, JackPortIsInput,
0);
      printf("OK\n");

      if( graphics == 1 )
      {
      // Initialize SDL
        printf("Initializing SDL...");
        if (SDL_Init (SDL_INIT_VIDEO) < 0) {
            printf ("Failed to initialize SDL\n");
            return;
        }
        printf("OK\n");
      }

      //Initialize ALSA sequencer
      printf("Initializing ALSA Sequencer...");
      alsa_port = open_seq( &seq_handle );
      printf("OK\n");

      /* Midi Output channels */
      printf("Allocating MIDI output channels...");
      memset( channels, 0, 16 );
      channels[0] = 1;
      printf("OK\n");

      /* tell the JACK server that we are ready to roll */
      printf("Activating JACk Client...");
      if (jack_activate (client)) {
            fprintf (stderr, "cannot activate client");
            return 1;
      }
      printf("OK\n");

      /* connect the ports. Note: you can't do this before
            the client is activated, because we can't allow
           connections to be made to clients that aren't
          running.
        */
      if( connect_ports == 1 )
      {
      printf("Connecting input port...");
        if ((ports = jack_get_ports (client, NULL, NULL,
JackPortIsPhysical|JackPortIsOutput)) == NULL) {
                  fprintf(stderr, "Cannot find any physical capture
ports\n");
```

```
                exit(1);
        }

         if (jack_connect (client, ports[0], jack_port_name
(input_port))) {
                fprintf (stderr, "cannot connect input ports\n");
         }
        free (ports);
        printf("OK\n");
      }
      appIsRunning = 1;

      //start threads
      printf("Starting SDL Threads...");
      event_thread      = SDL_CreateThread ((void *) eventHandler,
NULL);
      worker_thread = SDL_CreateThread ((void *) worker_func, NULL);
      printf("OK\n");

      SDL_WaitThread(worker_thread, NULL); //wait for worker to
complete.
      SDL_Quit();

      printf("Shutting down...");
      jack_ringbuffer_free( rb );
      free( raw_pcm_data );
      free( win_pcm_data );
      free( freq_data );
      free( re );
      free( im );
      free( hammingWindow );
      free( blackmanWindow );
      jack_client_close (client);
      printf("OK\n");
      exit (0);
}

/* Handles Keyboard events
      This includes graphics and non-graphical modes
      */
void eventHandler(void) {
      int i;
      while (appIsRunning == 1) {

      if( graphics == 1 )                 //If graphics enabled
      {
      SDL_Event event;
      while(SDL_PollEvent(&event)) {
              switch(event.type) {
                  case SDL_KEYDOWN:
                        if( event.key.keysym.sym == SDLK_q )
                        {
                              printf("Quitting...\n");
                              appIsRunning = 0;
                        }

                        if( event.key.keysym.sym == SDLK_z )
```

```
                        {
                              printf("Removing Channel %d\n",
current_ch);
                              channels[current_ch--]=0;
                              if( current_ch == -1 ) current_ch = 15;
                              printf("Adding Channel %d\n",
current_ch);
                              channels[current_ch]=1;
                        }

                        if( event.key.keysym.sym == SDLK_c )
                        {
                              printf("Removing Channel %d\n",
current_ch);
                              channels[current_ch++]=0;
                              if( current_ch == 16 ) current_ch = 0;
                              printf("Adding Channel %d\n",
current_ch);
                              channels[current_ch]=1;
                        }

                        if( event.key.keysym.sym == SDLK_x )
                        {
                              printf("Adding Channel %d\n",
current_ch+1);
                              if( current_ch == 15 ) current_ch = -1;
                              channels[++current_ch]=1;
                        }
                        if( event.key.keysym.sym == SDLK_w )
                        {
                              if( currentWindow == hammingWindow )
                              {
                                    currentWindow = blackmanWindow;
                                    printf("Window type: Blackman\n");
                              }
                              else
                              {
                                    currentWindow = hammingWindow;
                                    printf("Window type: Hamming\n");
                              }
                        }
                  default:
                        break;
                    }
            }
            }
            else                              //If no Graphics mode
            {
                  char command;
                  printf("\nEnter Command:\n>: ");
                  command = getchar();
                              //get the newline character
                  switch( command )
                  {
                        case 'q':
                              printf("Quitting...\n");
                              appIsRunning = 0;
```

```
                        break;

                case 'z':
                        printf("Removing Channel %d\n", current_ch);
                        channels[current_ch--]=0;
                        if( current_ch == -1 ) current_ch = 15;
                        printf("Adding Channel %d\n", current_ch);
                        channels[current_ch]=1;
                        break;

                case 'c':
                        printf("Removing Channel %d\n", current_ch);
                        channels[current_ch++]=0;
                        if( current_ch == 16 ) current_ch = 0;
                        printf("Adding Channel %d\n", current_ch);
                        channels[current_ch]=1;
                        break;

                case 'x':
                        printf("Adding Channel %d\n", current_ch+1);
                        if( current_ch == 15 ) current_ch = -1;
                        channels[++current_ch]=1;
                        break;

                case 'w':
                        if( currentWindow == hammingWindow )
                        {
                                currentWindow = blackmanWindow;
                                printf("Window type: Blackman\n");
                        }
                        else
                        {
                                currentWindow = hammingWindow;
                                printf("Window type: Hamming\n");
                        }
                        break;
                case 'h':
                        printf("\nAvailable Commands:\nq - quit\nz -
previous MIDI channel\nc - next MIDI channel\nx -add next MIDI
channel\nw - change window type {Blackman, Hamming}\n");
                        break;
            }getchar();
        }
        }
}

void worker_func(void) {
        int i,t;
        if( graphics == 1 )
        {
        // Create window for OpenGL output
          if (!SDL_SetVideoMode(width, height, 32, SDL_OPENGL)) {
              printf ("Failed to create window\n");
              return;
          }

        SDL_WM_SetCaption( mainCaption, NULL);
```

```
	glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
	glViewport(0,0,width,height);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glOrtho(-1 * width/2,width/2,-1 * height/2,height/2,-1,1);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	//gluLookAt(0, 0, 1, 0, 0, 0, 0, 1, 0);

	}
	float sampleSep = (float)(width-40.0) / (bs*2);

	int current_note = -1;
	int current_note_amp = 0.0;
	int insideNote = 0;

	while(appIsRunning==1) {

		/* Shift old PCM data over to the left */
		for( i=0; i<bs; i++ )
			raw_pcm_data[i] = raw_pcm_data[i+bs];

		/* Read new PCM data from shared memory */
		if( jack_ringbuffer_read_space(rb) > dump_size)
			jack_ringbuffer_read( rb,
(char*)raw_pcm_data+dump_size, dump_size );

		/* Multiply PCM data by window */
		for( i=0; i<(bs*2); i++ )
			win_pcm_data[i] = raw_pcm_data[i] * currentWindow[i];
		/* Calculate FFT */
		memcpy( re, win_pcm_data, dump_size*2 );
		memset( im, 0, dump_size*2 );
		fft( (int)(logf((bs*2))/logf(2)),re,im );
		Rec_To_Polar( bs,re,im,freq_data);

		/* Find the peak frequencies */
		int fund_freq = find_peak( bs/4, freq_data );
		int note = hertz_to_midi( fund_freq*freq_band );

		/* Note Detection
		TODO: pitch shifts */
		if( insideNote == 1 )
		{
			if( freq_data[fund_freq]<15.0 )
			{

endMidiNote(current_note,seq_handle,alsa_port,channels);
			  current_note = -1;
			  insideNote = 0;
			}
			else if( note != current_note )
			{

endMidiNote(current_note,seq_handle,alsa_port,channels);
```

```
                    sendMidiNote(note,seq_handle,alsa_port,channels);
                        current_note=note;
                }
                else if( freq_data[fund_freq]>(current_note_amp+60) )
                {
                  endMidiNote(
current_note,seq_handle,alsa_port,channels);

sendMidiNote(current_note,seq_handle,alsa_port,channels);
                }
            }
            else if( (freq_data[fund_freq]>50.0) && (note > 0) )
            {
                    sendMidiNote(note,seq_handle,alsa_port,channels);
                    current_note = note;
                    insideNote = 1;
            }
            current_note_amp = freq_data[fund_freq];

        /* Enter Fancy Graphics... */
        if( graphics == 1 )
        {
            glClear(GL_COLOR_BUFFER_BIT);

            glColor3f( 0.0f, 0.0f, 0.0f );
            for( i=0; i<2; i++ )
            {
              glBegin(GL_LINE_LOOP);
              glVertex3f( -1*width/2+20, height/6-5+i*(height/3), 0 );
              glVertex3f( -1*width/2+20, -1*height/6+5+i*(height/3), 0
);
              glVertex3f( width/2-20, -1*height/6+5+i*(height/3), 0 );
              glVertex3f( width/2-20, height/6-5+i*(height/3), 0 );
              glEnd();
            }

            /* Draw the Waveform. */
            glColor3f(0.5,0.5,0.5);
            glLineWidth(2.0);
            glBegin( GL_LINES );
            for( i=0; i<(bs*2); i+=2 )
            {
              if(  displayWindow == 1 )
              {
                  glVertex3f(-1*width/2+20+i*sampleSep,
                        (height/6-5)*win_pcm_data[i],   0);
                  glVertex3f(-1*width/2+20+(i+1)*sampleSep,
                        (height/6-5)*win_pcm_data[i+1],   0);
              } else {
                  glVertex3f(-1*width/2+20+i*sampleSep,
                        (height/6-5)*raw_pcm_data[i],   0);
                  glVertex3f(-1*width/2+20+(i+1)*sampleSep,
                        (height/6-5)*raw_pcm_data[i+1],   0);
              }
            }
            /* Display Blue shadow of window in background */
            if(  displayWindow == 0 )
```

```
            {
              glColor3f(0.0,0.0,0.3);
              for( i=0; i<(bs*2); i+=2 )
              {
                  glVertex3f(-1*width/2+20+i*sampleSep,
                        (height/6-5)*currentWindow[i],   0);
                  glVertex3f(-1*width/2+20+(i+1)*sampleSep,
                        (height/6-5)*currentWindow[i+1],   0);
              }
            }
            glEnd();

            /* draw freq data */
            sample_t barWidth = (sample_t)(width-40)/(bs/4);
            glBegin( GL_QUADS );
            for( i=0; i<bs/4; i++ )
            {
            glColor3f(0.2,0.2,0.2);
            if( i == fund_freq )
                  glColor3f(0.0, 0.0, 1.0 ); //accent fundamental GREEN
             glVertex3f(-1*width/2+20 + barWidth*i,
                  (height/6-5)*freq_data[i]/100.0-
height/6+height/3+5,0);
              glVertex3f(-1*width/2+20 + barWidth*i,
                  -(height/6)+height/3+5,0);
              glVertex3f(-1*width/2+20 + barWidth + barWidth*i,
                  -(height/6)+height/3+5,0);
              glVertex3f(-1*width/2+20 + barWidth + barWidth*i,
                  (height/6-5)*freq_data[i]/100.0-
height/6+height/3+5,0);
            }
            glEnd();

            /* Draw Musical Staff */
            glColor3f( 1.0, 1.0, 1.0 );
            glBegin(GL_QUADS);
              glVertex3f( -1*width/2+20, height/10-5-2*(height/5), 0 );
              glVertex3f( -1*width/2+20, -1*height/10+5-2*(height/5), 0
);
              glVertex3f( width/2-20, -1*height/10+5-2*(height/5), 0 );
              glVertex3f( width/2-20, height/10-5-2*(height/5), 0 );
            glEnd();
            glColor3f( 0.2, 0.2, 0.2 );

            glBegin(GL_LINES);
            for( i=1;i<6;++i)
            {
              glVertex3f( -1*width/2+20,
                  height/10-5-2*height/5 - i*(height/5-10)/6, 0 );
              glVertex3f(    width/2-20,
                  height/10-5-2*height/5 - i*(height/5-10)/6, 0 );
            }
            glEnd();

            /* Draw Note */
            if( insideNote == 1 )
            {
```

```
            int line_num = midi_to_line( current_note );
            if( midi_to_sharp( note ) == 1 )
                  glColor3f(1.0, 0.0, 0.0 );
            glBegin(GL_TRIANGLES);
            glVertex3f( 0,
            height/10-5-2*height/5 - line_num*(height/5-10)/12, 0 );
            glVertex3f( -5,
            height/10-5-2*height/5 - (height/5-10)/12-
line_num*(height/5-10)/12, 0 );
            glVertex3f( 0,
            height/10-5-2*height/5 - (height/5-10)/6-
line_num*(height/5-10)/12, 0 );
            glEnd();
            glBegin(GL_LINES);
            //stem
            glVertex3f( 0,
            height/10-5-2*height/5 - (line_num-2)*(height/5-10)/12, 0
);
            glVertex3f( 0,
            height/10-5-2*height/5 - (height/5-10)/6-
line_num*(height/5-10)/12, 0 );
            glEnd();
            }

            glFinish();
            SDL_GL_SwapBuffers();
      }
            //sleep(1.0/desired_framerate);
      }
}

short hertz_to_midi( float hz) {
      if( hz < 0.0 ) return -1;
      int i, bestGuess, rVal;
      float difference = 1000.0;
      for( i=0; i<128; i++ )
      {
            if( fabs(midi_notes[i] - hz) < difference )
            {
                  bestGuess = i;
                  difference = fabs(midi_notes[i] - hz);
            }
      }
      if( bestGuess == 0 ) return -1;
      return bestGuess;
}
short midi_to_line(short midi) {
      if( midi==-1 ) return 100;

      int line;
      int octave = midi/12;
      switch(midi%12)
        {
            case 0:  line =  11; break;//c
            case 1:  line =  11; break;//c#
            case 2:  line =  10; break;//d
            case 3:  line =  10; break;//d#
```

```
            case 4:  line =  9; break;//e
            case 5:  line =  8; break;//f
            case 6:  line =  8; break;//f#
            case 7:  line =  7; break;//g
            case 8:  line =  7; break;//g#
            case 9:  line =  6; break;//a
            case 10:  line =  6; break;//a#
            case 11:  line =  5; break;//b
        }

      return line - 7*(octave-5);
}
short midi_to_sharp(short midi) {
      if( midi==-1 ) return -1;
      switch(midi%12)
        {
            case 1:  return 1; //c#
            case 3:  return 1; //d#
            case 6:  return 1; //f#
            case 8:  return 1; //g#
            case 10:  return 1; //a#
            default: return 0;
        }
}
```

References

Byrd, Don. (2003). *MIDI note number to equal temperament semitone to hertz conversion table.* Retrieved April 3, 2003, from Indiana University School of Music Center for Electronic and Computer Music Web site: http://www.indiana.edu/~emusic/hertz.htm

Moore, Brian C.J. (1997). *An Introduction to the Psychology of Hearing 4th Edition*, (Academic Press, London).

Smith, Steven W. (1999). *A Scientists and Engineer's Guide to Digital Signal Processing 2nd Edition*, (Technical Publishing, San Diego).

Whitefield, I. C. (1970). Central nervous processing in relation to spatiotemporal discrimination of auditory patterns, in *Frequency Analysis and Periodicity Detection in Hearing,* edited by R. Plomp and G. F. Smoorenburg (Sijthoff, Leiden).

MIDI 1.0 Specification Message Summary. Retrieved April 3, 2003. From: http://www.midi.org/about-midi/table1.shtml