UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

Department of Computer & Communication Engineering

USE OF MOBILE MUSIC SYNTHESIS TOOLKIT IN C++(MoMu-STK) TO DEVELOP A MANDOLIN MUSIC SYNTHESIS

By:
Georgia Alexopoulou

Submitted in September 2012 in partial fulfilment of the requirements for the Diploma degree in Computer & Communication Engineering.

THESIS COMMITTEE
ASSOCIATE PROFESSOR ALKIVIADIS AKRITAS (SUPERVISOR)
ASSISTANT PROFESSOR ASPASIA DASKALOPOULOU

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

Τμήμα Μηχανικών Ηλεκτρονικών Υπολογιστών Τηλεπικοινωνιών και Δικτύων

ΧΡΗΣΗ ΤΟΥ ΕΡΓΑΛΕΙΟΥ ΣΥΝΘΕΣΗΣ ΜΟΥΣΙΚΗΣ ΓΙΑ ΚΙΝΗΤΑ ΣΕ C++ (MoMu-STK) ΓΙΑ ΤΗΝ ΣΥΝΘΕΣΗ ΜΟΥΣΙΚΗΣ ΜΕ ΜΟΥΣΙΚΟ ΟΡΓΑΝΟ ΤΟ ΜΑΝΤΟΛΙΝΟ

Γεωργία Αλεξοπούλου

Υποβλήθηκε τον Σεπτέμβριο του 2012 προς μερική εκπλήρωση των υποχρεώσεων για το Δίπλωμα Μηχανικού Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ
ΑΝΑΠΛΗΡΩΤΗΣ ΚΑΘΗΓΗΤΗΣ ΑΛΚΙΒΙΑΔΗΣ ΑΚΡΙΤΑΣ (ΕΠΙΒΛΕΠΩΝ)
ΕΠΙΚΟΥΡΟΣ ΚΑΘΗΓΗΤΗΣ ΑΣΠΑΣΙΑ ΔΑΣΚΑΛΟΠΟΥΛΟΥ

Abstract

A mandolin is a musical instrument in the lute family. Lute family includes instruments which make sound by the vibration of strings.

MoMu-STK is α set of open source audio signal processing and algorithmic synthesis classes. It was designed to facilitate rapid development of music synthesis and audio processing software, with an emphasis on cross-platform functionality, real-time control, ease of use, and educational example code. The Synthesis Toolkit is extremely portable and it's completely user-extensible. STK currently runs with real-time support (audio and MIDI) on Linux, Macintosh OS X, and Windows computer platforms. Generic, non-real time support has been tested under Next Step, Sun, and other platforms and should work with any standard C++ compiler. For control, the Synthesis Toolkit uses raw MIDI (on supported platforms), and SKINI (Synthesis ToolKit Instrument Network Interface, a MIDI-like text message synthesis control format). The Synthesis Toolkit can generate simultaneous SND (AU), WAV, AIFF, and MAT-file output sound file formats (as well as real-time sound output), so you can view your results using one of a large variety of sound/signal analysis tools already available (e.g. Snd, Cool Edit, Matlab).

In this thesis we describe a method for developing a mandolin music synthesis based on the Mobile Music Synthesis Toolkit (MoMu-STK). We use a set of C++ classes of MoMu-STK to create our own program. In order to cover our specific needs, we modify some of the existing classes and write a new program. This program produces music compositions (audio files) based on the mandolin instrument.

The proposed method can be used in a variety of ways, allowing composers to generate music compositions based on different musical instruments or a combination of musical instruments that belong to the same or different families. Moreover, since MoMu offers a collection of useful utilities used for audio synthesis and processing, graphics, and threading, we can provide useful functionality for mobile phone music application development.

Περίληψη

Το μαντολίνο είναι ένα μουσικό όργανο της μεγάλης οικογένειας του Λαούτου. Η οικογένεια του Λαούτου περιλαμβάνει όργανα που παράγουν ήχο από τη δόνηση των χορδών.

Το MoMu-STK είναι ένα σύνολο από κλάσεις επεξεργασίας σήματος ήχου και αλγοριθμικής σύνθεσης. Είχε σχεδιαστεί για να διευκολύνει την ταχεία ανάπτυξη της σύνθεσης μουσικής και του λογισμικού επεξεργασίας ήχου με έμφαση στη cross-platform λειτουργικότητα, στον έλεγχο πραγματικού χρόνου, στην ευκολία στη χρήση, καθώς και στον εκπαιδευτικό κώδικα. Το STK είναι εξαιρετικά φορητό και πλήρως επεκτάσιμο. Το STK λειτουργεί επί του παρόντος με υποστήριξη σε πραγματικό χρόνο (audio και MIDI) στις Linux, Macintosh OS X και Windows πλατφόρμες υπολογιστών. Γενικά, η υποστήριξη σε μη πραγματικό χρόνο, έχει δοκιμαστεί σύμφωνα με την Next Step, την Sun, και άλλες πλατφόρμες και θα πρέπει να δουλεύει με οποιοδήποτε πρότυπο C compiler. Για τον έλεγχο, το STK χρησιμοποιεί raw MIDI (στις υποστηριζόμενες πλατφόρμες), και SKINI (Synthesis ToolKit Instrument Network Interface). Το STK μπορεί να παράγει ταυτόχρονα SND (AU), WAV, AIFF, και MAT-αρχεία ήχου ως έξοδο (καθώς και πραγματικού χρόνου αρχεία ήχου ως έξοδο), ώστε να μπορείτε να δείτε τα αποτελέσματά σας χρησιμοποιώντας ένα εργαλείο από μια μεγάλη ποικιλία εργαλείων ανάλυσης ήχου / σήματος που είναι ήδη διαθέσιμα (π.χ. Snd., Cool Edit, Matlab).

Στην εργασία αυτή περιγράφουμε μια μέθοδο για την ανάπτυξη μιας μουσικής σύνθεσης με μουσικό όργανο το μαντολίνο με βάση το εργαλείο σύνθεσης μουσικής για κινητά (STK-MoMu). Χρησιμοποιούμε ένα σύνολο από C++ κλάσεις του MoMu-STK για να δημιουργήσουμε το δικό μας πρόγραμμα. Για την κάλυψη των ειδικών αναγκών μας, τροποποιούμε κάποιες από τις υπάρχουσες κλάσεις και γράφούμε ένα νέο πρόγραμμα. Αυτό το πρόγραμμα παράγει συνθέσεις μουσικής (αρχεία ήχου) με μουσικό όργανο το μαντολίνο.

Η προτεινόμενη μέθοδος μπορεί να χρησιμοποιηθεί σε μια ποικιλία από τρόπους, επιτρέποντας στους συνθέτες να παράγουν μουσικές συνθέσεις που βασίζονται σε διαφορετικά μουσικά όργανα ή σε συνδυασμό των μουσικών οργάνων τα οποία ανήκουν στην ίδια ή σε διαφορετικές οικογένειες. Επιπλέον, δεδομένου ότι το MoMu προσφέρει μια συλλογή από χρήσιμα προγράμματα που χρησιμοποιούνται για την σύνθεση και επεξεργασία ήχου, για τα γραφικά και για threading, μπορούμε να παρέχουμε χρήσιμες λειτουργίες για ανάπτυξη εφαρμογών μουσικής σε κινητά τηλέφωνα.

Contents

Illustration Index

Index of Tables

Chapter 1 : Introduction To Music Theory

This Section will give you a basic introduction to the general rules that are the basis for music all over the world.

1.1 Solfege Syllables

Solfege is a pedagogical solmization technique for the teaching of sight-singing in which each note of the score is sung to a special syllable, called a **solfège syllable**. The seven syllables commonly used for this practice in English-speaking countries are: **do, re**, **mi**, **fa**, **sol**, **la**, and **ti**/**si.**

In the major Romance and Slavic languages, the syllables Do, Re, Mi, Fa, Sol, La, and Si are used to name notes the same way that the letters C, D, E, F, G, A, and B are used to name notes in English. This system is called **fixed do** and is used in Spain, Portugal, France, Italy, Belgium, Romania, Latin American countries and in French-speaking Canada as well as countries such as Bosnia and Herzegovina, Russia, Poland, Serbia, Ukraine, Bulgaria, Greece, Albania, Macedonia, Iran, Lebanon, Turkey and Israel where non-Romance languages are spoken.

Table 1: Traditional fixed do

Note name	Syllable
C	do
D	re
E	mi
F	fa
G	so
A	la
B	ti

1.2 The Notes Of Music

There are only 12 different notes that make up the building blocks of any song you have ever heard. Even music from cultures that were previously considered

to have a separate music systems have been studied and found to use the 12 note system. These 12 notes create what is known as a chromatic scale.

C, C# (Db), D, D# (Eb), E, F, F# (Gb), G, G# (Ab), A, A# (Bb), and B.

It is important to note that the "#" symbol is pronounced "sharp" and the "b" symbol is pronounced "flat". For example, D# is the note above D and the note below E. Eb is the same note as D# simply with a different name. In most circumstances, however, we refer to the note above C as C#, not Db, and the note above F as F#, not Gb, though either way is theoretically acceptable. In the same sense, the note above D is referred to as Eb, and the note above A is referred to as Bb. Each of these notes are (effectively) the same distance apart from one to the next. In reality, D# is 6% higher than D and E is 6% higher than D# and so on. these numbers are not exact, as they have been altered so that the octaves match up. The distance between two notes that are one fret apart is called a half step or a semitone, the distance between two notes that are two frets apart is called a whole step or a tone. I prefer to use the terms half step (HS) and whole step (WS).

An octave is a note that sounds the same as another note but twice as high. Once you go through the chromatic scale starting, for example, on G, you have the following scale. G, Ab, A, Bb, B, C, C#, D, Eb, E, F, F#... But then what? Well, you get G again, but twice as high as the G you started on. The distance between these two notes is called an octave.

1.3 Scales

If you play all twelve of these notes on an instrument in succession, it won't sound like much. That's because the song we have heard all our lives have not included all twelve notes and our brains don't like to hear it. Our brains have become accustomed to other scales. Most of the scales we hear in music from the past and present usually only have 7 different notes, sometimes only 5. The first note of a scale is called the root. The major scale, which is usually described as "happy" is played:

Root, WS, WS, ST, WS, WS, WS, ST (octave)

while the minor scale, which is usually described as "sad" is played:

Root, WS, ST, WS, WS, ST, WS, WS (octave)

The C scale is unique in the sense that when you play the C major scale, there are no sharps or flats, just C, D, E, F, G, A, B, C. Likewise, the A minor scale has no sharps or flats. Because of this, A minor is known as the relative minor of C. In the case of G major and E minor, they both have one sharp (F#), Making E minor the relative minor of G major.

1.4 Intervals

In music, an interval is the distance between two notes. We can count simple intervals by simply starting with the root and counting up to the note in question through the scale of the root. For example, the interval between a C and an A would be calculated in the following manner:

C (1), D (2), E (3), F (4), G (5), A (6)

So now we know that the interval from C to A is a 6th. However, musicians like to make things complicated, so there is more to the question. The way to calculate the more precise interval is to follow this table.

Table 2: Intervals

Distance in half steps	**Interval Name**
0	Unison
1	Minor Second
2	Major Second
3	Minor Third
4	Major Third
5	Perfect Fourth
6	Tritone
7	Perfect Fifth
8	Minor Sixth
9	Major Sixth
10	Minor Seventh

11	Major Seventh
12	Octave

So now we can count chromatically, saying C (0), C# (1), D (2), Eb (3), E (4), F (5), F# (6), G (7), Ab (8), A (9). That's a 9 half step difference, making it a Major Sixth. While it may seem like a lot now, it's not too bad once you get the hang of it.

1.5 The Circle Of Fifths

Possibly the most important structure in music theory, the circle of fifths ties all of the notes, chords, and scales together by relating them to one another.

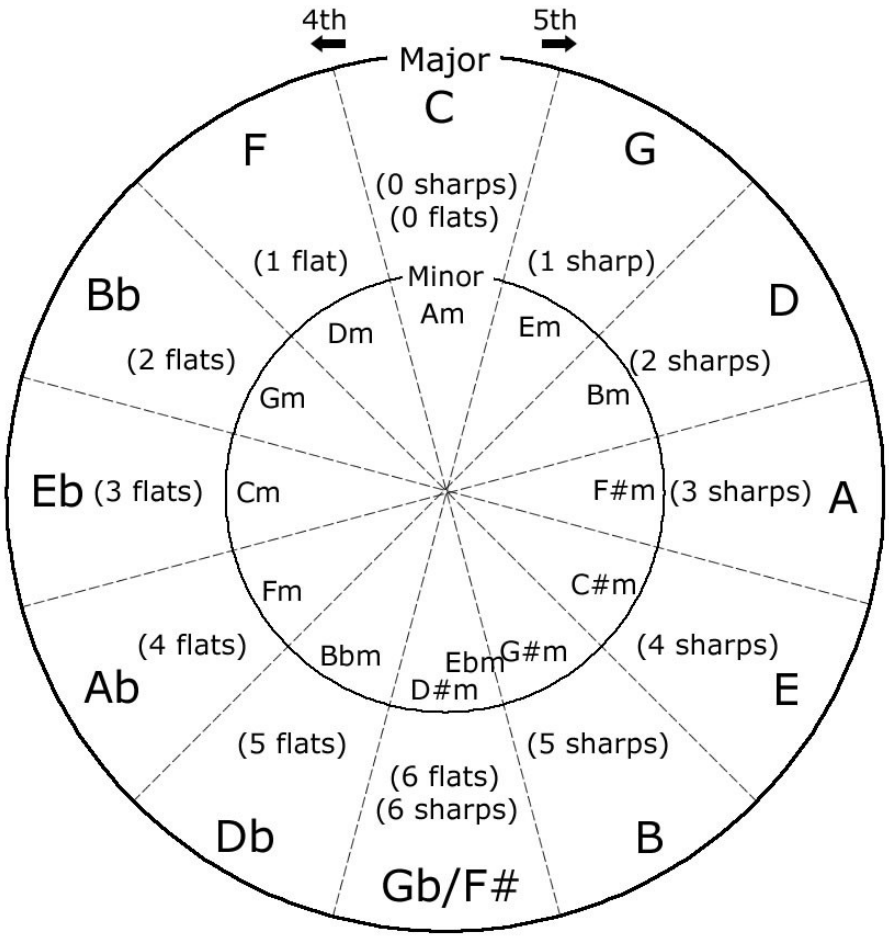

Figure 1: The Circle of Fifths

Again, this is a lot of information all at once, so don’t feel like you need to learn it all at once. Looking at this larger circular jumble of information, it is best to break it down into sections.

The first thing to notice is that, when you follow the outermost letters clockwise, they are all separated by perfect fifths, C to G, G to D, etc. Following the circle counterclockwise, the progression moves in fourths, C to F, F to Bb, etc.

As a practical tool, lets say we wanted to play a song in A. To figure out what chords to play, you already know that you want the I, IV, and V. Looking at the circle of fifths, find the A. Your fourth will be one step counterclockwise from your A, and your fifth will be one step clockwise from your A.

The next section of the circle is the inner circle. The inner circle deals with relative minors. As you have learned, each scale has a number of sharps or flats, or, in the case of C major, no sharps or flats. When it comes to minor scales, it is the same, with Am being the scale with no sharps or flats. From this information we can discover that both C and Am use the same notes. the C scale is C, D, E, F, G, A, B, C. The Am scale is A, B, C, D, E, F, G, A. There are no sharps or flats in either scale. One way of looking at it is, to play the Am scale, you play the notes of a C scale but rather than starting on the C note, you start on the A note. As another example, the key of D consists of the notes D, E, F#, G, A, B, C#, D. The relative minor of D is Bm, which has the notes B, C#, D, E, F#, G, A, B. One easy way to find the relative minor of a key is to count backwards two notes. For example, counting backwards two notes on the A major scale gives you A, G# (Ab), F#. Therefore, your relative minor of A is F#m. Understanding the theory behind relative minors simplifies music even farther by reducing the number of scales you need to know. Rather than picking around until you get all the notes of the Dm scale, play your F major scale but start on the D note of the scale, giving you D, E, F, G, A, Bb, C, D.

Chapter 2 : Sound and Music

2.1 Sound Waves

Sound is the vibration of air particles, which travels to your ears from the vibration of the object making the sound. These vibrations of sound in the air are called **sound waves**.

When a door is slammed, the door vibrates, sending sound waves through the air.

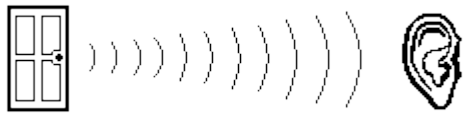

Figure 2: a slammed door

When a mandolin string is plucked, the string vibrates the soundboard, which sends sound waves through the air.

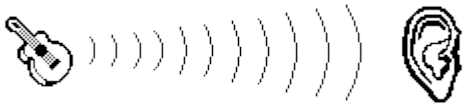

Figure 3: a plucked string

What makes one sound different from another? To answer this question, we need to look at the waveforms of the two different sounds, to see the shape of their vibrations.

The waveform of a door slamming looks something like this:

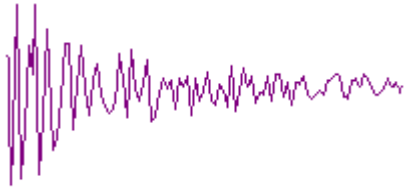

Figure 4: waveform of a door slamming

This waveform is jerky and irregular, resulting in a harsh sound. Notice how it is loud (with big waves) at the start, but then becomes soft (small waves) as it dies away.

The waveform of a mandolin string looks something like this:

Figure 5: waveform of a mandolin string

This waveform makes the same transition from loud to soft as the first, but otherwise is quite different.

The mandolin string makes a continuous, regular series of repeated cycles, which we hear as a smooth and constant musical tone.

This regularity of the vibration is the difference between a musical sound and a non-musical sound.

2.2 Musical Sounds

Musical sounds are vibrations which are strongly regular. When you hear a regular vibration, your ear detects the frequency, and you perceive this as the **pitch** of a musical tone.

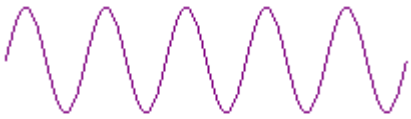

Figure 6: pitch of a musical sound

Non-musical sounds are a complex mix of different (and changing) frequencies. Your ear still follows these vibrations, but there is no strong regularity from which you can pick up a musical tone.

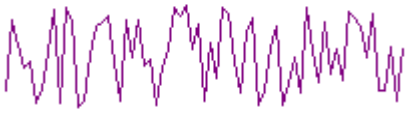

Figure 7: pitch of a non-musical sound

Many sounds are a mixture of both, such as drums and other percussion instruments. You can usually decide which of two drums has the higher pitch, even if it might be difficult to decide exactly what that pitch is.

Most sounds have some regularity in them (even a door slamming) but not enough for your ear to detect a specific pitch.

2.3 Amplitude and Frequency

There are two main properties of a regular vibration - the amplitude and the frequency - which affect the way it sounds.

Amplitude is the size of the vibration, and this determines how loud the sound is. We have already seen that larger vibrations make a louder sound.

Figure 8: Low Amplitude

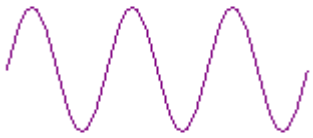

Figure 9: High Amplitude

Amplitude is important when balancing and controlling the loudness of sounds, such as with the volume control on your CD player. It is also the origin of the word **amplifier**, a device which increases the amplitude of a waveform.

Frequency is the speed of the vibration, and this determines the pitch of the sound. It is only useful or meaningful for musical sounds, where there is a strongly regular waveform.

Figure 10: Low Frequency

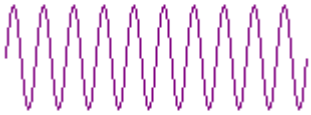

Figure 11: High Frequency

Frequency is measured as the number of wave cycles that occur in one second. The unit of frequency measurement is Hertz (Hz for short).

A frequency of 1 Hz means one wave cycle per second. A frequency of 10 Hz means ten wave cycles per second, where the cycles are much shorter and closer together.

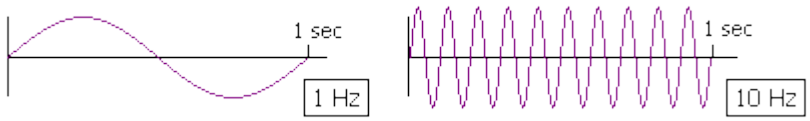

Figure 12: Frequency of 1 Hz and 10 Hz

The note A which is above Middle C (more on this later) has a frequency of 440 Hz. It is often used as a reference frequency for tuning musical instruments.

2.4 Musical Instrument Tone

There is a huge variety of musical instruments and sounds, as you would already know from your experience with music. Even two instruments playing the same note can sound very different.

This is because a musical instrument produces a sound wave which is a combination of different but related frequencies (known as **harmonics**) which all mix together to create the distinctive **tone** or **voice** of the instrument.

The lowest frequency is usually dominant, and you perceive this one as the pitch. The combination of the other harmonics provides the distinctive shape of the waveform, and thereby the distinctive tone of the instrument.

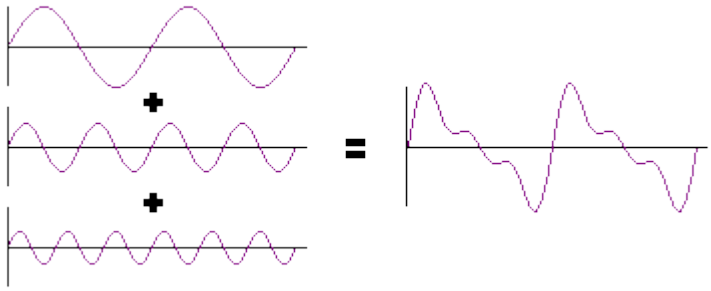

Figure 13: create the tone or voice of the instrument.

2.5 Frequency and Pitch

Frequency and pitch describe the same thing, but from different viewpoints. While frequency measures the cycle rate of the physical waveform, **pitch** is how high or low it sounds when you hear it.

This is directly related to frequency: the higher the frequency of a waveform, the higher the pitch of the sound you hear.

Think of the sound of a car or motorcycle engine accelerating. As the engines turns faster (at a higher frequency) the engine makes a higher-pitched sound.

Human ears can only hear sounds within a certain range of frequencies. As people grow older, their hearing range reduces. A young person can usually hear sounds in the range of 20 Hz to 20,000 Hz.

The Figure below is an example of a continuous pitch sweep from the lowest to the highest audible frequencies.

Figure 14: continuous pitch from the lowest to the highest audible frequencies.

At the lower end of this range are low-pitched sounds like the booming of thunder before a storm. At the upper end of this range are high-pitched sounds like the piercing whine of a mosquito.

Between these is the whole spectrum of sound and music!

2.6 Doubling Frequency

Something very interesting happens when you double the frequency of a note. The pitch of the doubled frequency sounds higher, but somehow the same as the original note, while the pitches of all frequencies in between sound quite different.

Lets use the pitch of frequency 440 Hz as an example. It is the note A, as mentioned earlier. The pitch of frequency 880 Hz is higher, but sounds like the same note.

It seems strange, but there is a logical reason for this similarity. The sound waves below show us that two cycles of the 880 Hz frequency fit exactly in the space of a single cycle of the 440 Hz frequency.

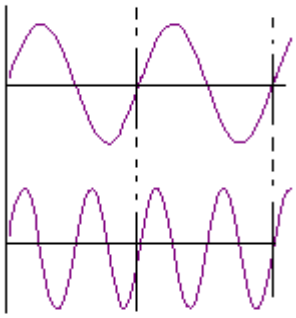

Figure 15: Sound waves of 440Hz and 880Hz

If we keep doubling this frequency, we find that all of the resulting pitches sound similar, except that each one is higher than the last. In fact, they are all the note A, just like the original, but they are all one **octave** apart from each other.

2.7 What Is String Muting & Damping?

String muting means preventing certain strings from sounding while we play other strings.

String Damping means stopping the string we are playing from vibrating with its usual freedom, and “dampening” the sound so that a muffled type of tone is

produced. When we damp, unlike when we mute a string, the actual pitch of the note is still evident.

The reason we need to mute strings at times is because otherwise the vibrating strings will interfere with the music we are making. For instance, in doing many rock licks, the bending and release of a higher string will cause lower bass strings to vibrate, either because we actually bump into them, or because they start to vibrate “in sympathy” with the ones we have played.

The reason we damp notes (usually bass notes) is because the tone produced is itself an expressive musical device. For hundreds of years, string players have done it, and in classical music it is called “pizzicato”.

2.7.1 How is it done?

The way we mute or damp strings we don’t want to hear from is by touching the string with the skin of the side of the hand. If you do a karate chop on the table, the part of your hand touching the table is the part used to mute the strings.

This is the same position we use to mute the strings, however, there is more to the story. We have to make smaller adjustments to the hand as we play and move from string to string.

2.7.2 The Difference Between Muting Position and Damping Position

The hand itself is in the same position in relation to the strings for both techniques, but the position of the hand itself is different. Since we do not need any tone from the note in muting, it does not matter where along the length of the strings we place our hand, as long as we silence the strings.

For damping, since we need a discernable pitch, we must place our hands down by the bridge and only partially cover the strings, right at the point where they meet the bridge. We have to leave enough string free to vibrate.

2.8 What is Strumming?

Strumming is way of playing a stringed instrument such as a guitar. A **strum** or **stroke** is a sweeping action where a fingernail or plectrum brushes past several strings in order to set them all into motion and thereby play a chord. Strums are executed by the dominant hand, right for a right-handed musician and left for a left handed one, while the other hand holds down notes on the fretboard. Strums are contrasted with plucking, as means of activating strings

into audible vibration, because in plucking, only one string is activated by a surface at a time. A hand-held pick or plectrum can only be used to pluck one string at a time, but multiple strings can be strummed by one. Plucking multiple strings simultaneously requires a finger style or finger pick technique.

2.9 Tuning

2.9.1 What is tuning?

Tuning is the process of adjusting the pitch of one or many tones from musical instruments until they form a desired arrangement. Pitch is the perceived fundamental frequency of a sound. Instruments basically just produce vibrations, and these vibrations produce the sound that we hear. The vibrations or soundwaves that an instrument produces are measured by hertz (symbol: Hz). One hertz simply means one cycle per second, 100 Hz means one hundred cycles per second, and so on. The average human can hear sounds between 20 Hz and 16,000 Hz. In music and acoustics, the frequency of the standard pitch A above middle C on a piano is usually defined as 440 Hz.

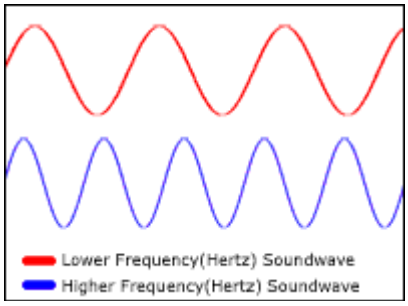

Figure 16: Lower/ Higher Frequency Soundwave

2.9.2 What does "My instrument is out of tune" mean?

When an instrument is out of tune it means that the pitch or tone of the instrument is either too high or too low. If a tone is too high, it is considered sharp, if it is to low, then it is flat.

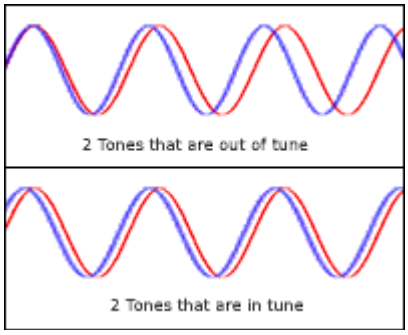

Figure 17: in and out of tune

2.9.3 Why do instruments go out of tune?

Well, there are many factors why an instrument may go out of tune. Some instruments get 'out of tune' with damage or age (warping) when they will no longer play true and have to be repaired. Also changes in temperature and humidity can affect some sensitive instruments. As temperatures fluctuate, instruments may expand or contract. This causes the instrument to go slightly out of tune. On stringed instruments, brand new strings go out of tune quickly at first and need to be "broken in" at first. Also a string obviously can get out of tune if the tuning pegs or tuning devices are bumped or adjusted.

Chapter 3 : Musical Instrument Digital Interface

3.1 What is MIDI?

MIDI (short for **Musical Instrument Digital Interface**) is an electronic musical instrument industry specification that enables a wide variety of digital musical instruments, computers and other related devices to connect and communicate with one another. It is a set of standard commands that allows electronic musical instruments, performance controllers, computers and related devices to communicate, as well as a hardware standard that guarantees compatibility between them. MIDI equipment captures note events and adjustments to controls such as knobs and buttons, encodes them as digital messages, and sends these messages to other devices where they control sound generation and other features. This data can be recorded into a hardware or software device called a sequencer, which can be used to edit the data and to play it back at a later time. MIDI carries note event messages that specify notation, pitch and velocity, control signals for parameters such as volume, vibrato, audio panning and cues, and clock signals that set and synchronize tempo between multiple devices. A single MIDI link can carry up to sixteen channels of information, each of which can be routed to a separate device. The 1983 introduction of the MIDI protocol revolutionized the music industry.

3.2 MIDI Note Numbers for Different Octaves

In this system, middle C (MIDI note number 60) is C4. A MIDI note number of 69 is used for A440 tuning, that is the A note above middle C.

Table 3: MIDI Note Numbers

Octave	Note Numbers							
	C	C#	D	D#	E	F	F#	G
-1	0	1	2	3	4	5	6	7
0	12	13	14	15	16	17	18	19
1	24	25	26	27	28	29	30	31
2	36	37	38	39	40	41	42	43
3	48	49	50	51	52	53	54	55
4	60	61	62	63	64	65	66	67
5	72	73	74	75	76	77	78	79
6	84	85	86	87	88	89	90	91
7	96	97	98	99	100	101	102	103
8	108	109	110	111	112	113	114	115
9	120	121	122	123	124	125	126	127

Octave	Note Numbers			
	G#	A	A#	B
-1	8	9	10	11
0	20	21	22	23
1	32	33	34	35
2	44	45	46	47
3	56	57	58	59
4	68	69	70	71
5	80	81	82	83
6	92	93	94	95
7	104	105	106	107
8	116	117	118	119

3.3 MIDI Note Numbers

The MIDI specification only defines note number 60 as "Middle C", and all other notes are relative. The absolute octave number designations shown here are based on Middle C = C4.

There is a discrepancy that occurs between various models of MIDI devices and software programs, and that concerns the octave numbers for note names. If your MIDI software/device considers octave 0 as being the lowest octave of the MIDI note range, then middle C's note name is C5. The lowest note name is

then C0 (note number 0), and the highest possible note name is G10 (note number 127).

Some software/devices instead consider the third octave of the MIDI note range (2 octaves below middle C) as octave 0. In that case, the first 2 octaves are referred to as -2 and -1. So, middle C's note name is C3, the lowest note name is C-2, and the highest note name is G8.

A MIDI controller can have up to 128 distinct pitches/notes. But whereas musicians name the keys using the alphabetical names, with sharps and flats, and also octave numbers, this is more difficult for MIDI devices to process, so they instead assign a unique number to each key.

The numbers used are 0 to 127. The lowest note upon a MIDI controller is a C and this is assigned note number 0. The C# above it would have a note number of 1. The D note above that would have a note number of 2. So "Middle C" is note number 60. A MIDI note number of 69 is used for A440 tuning, that is the A note above middle C.

Most keyboard controllers have a "MIDI transpose" function so that, even if you don't have the full 128 keys, you can alter the note range that your keyboard covers. For example, instead of that lowest A key being assigned to note number 21, you could transpose it down an octave so that it is assigned a note number of 9.

3.4 MIDI Note Names

Many instruments can play distinct pitches. For example, an acoustic piano has 88 keys, or 88 distinct pitches/notes.

Instruments with keyboards were among the earliest, most versatile musical instruments at around the time when musicians were devising a way to notate music. So, it's traditional to name musical pitches based upon the piano keyboard. They are visually grouped into octaves where one octave contains 12 keys.

Musicians name the musical pitches played upon the white keys by using the alphabetical names A to G. For example, "middle C" is the white key closest to the center of the keyboard. Musicians append sharps or flats to the alphabetical names to identify the black keys. For example, the black key above middle C is a C#. Also, musicians use the octave number to further identify a particular key.

3.5 MIDI Messages

3.5.1 Note On

Purpose

Indicates that a particular note should be played. Essentially, this means that the note starts sounding, but some patches might have a long VCA attack time

that needs to slowly fade the sound in. In any case, this message indicates that a particular note should start playing (unless the velocity is 0, in which case, you really have a Note Off).

Status

Status is the MIDI channel.

Data

Two data bytes follow the Status.

The first data is the note number. There are 128 possible notes on a MIDI device, numbered 0 to 127 (where Middle C is note number 60). This indicates which note should be played.

The second data byte is the velocity, a value from 0 to 127. This indicates with how much force the note should be played (where 127 is the most force). It's up to a MIDI device how it uses velocity information. Often velocity is be used to tailor the VCA attack time and/or attack level (and therefore the overall volume of the note). MIDI devices that can generate Note On messages, but don't implement velocity features, will transmit Note On messages with a preset velocity of 64.

A Note On message that has a velocity of 0 is considered to actually be a Note Off message, and the respective note is therefore released.

A device that recognizes MIDI Note On messages must be able to recognize both a real Note Off as well as a Note On with 0 velocity (as a Note Off). There are many devices that generate real Note Offs, and many other devices that use Note On with 0 velocity as a substitute.

Errata

In theory, every Note On should eventually be followed by a respective Note Off message (ie, when it's time to stop the note from sounding). Even if the note's sound fades out (due to some VCA envelope decay) before a Note Off for this note is received, at some later point a Note Off should be received. For example, if a MIDI device receives the following Note On:

Note On/chan 0, Middle C, velocity could be anything except 0

Then, a respective Note Off should subsequently be received at some time, as so:

Note Off/chan 0, Middle C, velocity could be anything

Instead of the above Note Off, a Note On with 0 velocity could be substituted as so:

Really a Note Off/chan 0, Middle C, velocity must be 0

If a device receives a Note On for a note (number) that is already playing (ie, hasn't been turned off yet), it is the device's decision whether to layer another "voice" playing the same pitch, or cut off the voice playing the preceding note of that same pitch in order to "retrigger" that note.

3.5.2 Note Off

Purpose

Indicates that a particular note should be released. Essentially, this means that the note stops sounding, but some patches might have a long VCA release time that needs to slowly fade the sound out.

Status

Status is the MIDI channel.

Data

Two data bytes follow the Status.

The first data is the note number. There are 128 possible notes on a MIDI device, numbered 0 to 127 (where Middle C is note number 60). This indicates which note should be released.

The second data byte is the velocity, a value from 0 to 127. This indicates how quickly the note should be released (where 127 is the fastest). It's up to a MIDI device how it uses velocity information. Often velocity will be used to tailor the VCA release time. MIDI devices that can generate Note Off messages, but don't implement velocity features, will transmit Note Off messages with a preset velocity of 64.

3.5.3 Control Change

This message is quite versatile; it is usually generated by a musician using knobs, sliders, footswitches, or pressure on a physical MIDI controller (or MIDI-equipped instrument). While the response to this message is generally totally up to the receiving device, it is typically used to change the tone, timbre, or volume of an instrument's sound.

3.5.4 Chord

Indicates that a particular sequence of chords should be played.

3.5.5 ChordOff

Indicates that a particular sequence of chords should be released.

3.6 How do messages work?

All MIDI compatible controllers, musical instruments, and MIDI-compatible software follow the same MIDI 1.0 specification, and thus interpret any given

MIDI message the same way, and so can communicate with and understand each other. For example, if a note is played on a MIDI controller, it will sound at the right pitch on any MIDI instrument whose MIDI In connector is connected to the controller's MIDI Out connector.

When a musical performance is played on a MIDI instrument (or controller) it transmits *MIDI channel messages* from its MIDI Out connector. A typical MIDI channel message sequence corresponding to a key being struck and released on a keyboard is:

1. The user presses the middle C key with a specific *velocity* (which is usually translated into the volume of the note but can also be used by the synthesizer to set characteristics of the timbre as well). The instrument sends one *Note-On* message.
2. The user changes the pressure applied on the key while holding it down: a technique called *Aftertouch* (can be repeated, optional). The instrument sends one or more Aftertouch messages.
3. The user releases the middle C key, again with the possibility of velocity of release controlling some parameters. The instrument sends one *Note-Off* message.

Note-On, *Aftertouch*, and *Note-Off* are all channel messages: embedded in the message is one of 16 channel IDs. This enables instruments to be set to respond to messages on specific channels while ignoring all others. (System messages, in contrast, are designed to be responded to by all connected devices.) For the Note-On and Note-Off messages, the MIDI specification defines a number (from 0–127) for every possible note pitch (C, C♯, D, etc.), and this number is included in the message along with the velocity value.

Other performance parameters can be transmitted with channel messages, too. For example, if the user turns the pitch wheel on the instrument, that gesture is transmitted over MIDI using a series of *Pitch Bend* messages (also a channel message). The musical instrument generates the messages autonomously; all the musician has to do is play the notes (or make some other gesture that produces MIDI messages). This consistent, automated abstraction of the musical gesture could be considered the core of the MIDI standard.

3.7 MIDI Note Number to Frequency Conversion

The standard tuning pitch is A4 (A above middle C) 440 Hertz(Hz). When the octave is divided into 12 equally spaced pitches, then each interval between these pitches when they are arranged within the same octave in an ascending sequence is the equal-tempered semitone:

$$2^{1/12} = 1.059463094\ldots$$

The frequencies of equal-tempered pitches are easily calculated from any reference frequency according to :

$$f = 2^{n/12} f_{ref}$$

where n is the number of semitones between the reference frequency ($f_{ref)}$, and the desired frequency, f .

For example, the frequency of the equal-tempered pitch located a major third up (+4 semitones) from A440 is $440(2^{4/12}) = 554.365...$ Hz, while middle C, a major sixth down (-9 semitones), has a frequency of $440(2^{-9/12}) = 261.6255...$Hz.

Table 4: Frequencies of MIDI Note Numbers

MIDI Note		Frequency	MIDI Note	Frequency
C	0	8.1757989156	12	16.3515978313
Db	1	8.6619572180	13	17.3239144361
D	2	9.1770239974	14	18.3540479948
Eb	3	9.7227182413	15	19.4454364826
E	4	10.3008611535	16	20.6017223071
F	5	10.9133822323	17	21.8267644646
Gb	6	11.5623257097	18	23.1246514195
G	7	12.2498573744	19	24.4997147489
Ab	8	12.9782717994	20	25.9565435987
A	9	13.7500000000	21	27.5000000000
Bb	10	14.5676175474	22	29.1352350949
B	11	15.4338531643	23	30.8677063285
C	24	32.7031956626	36	65.4063913251
Db	25	34.6478288721	37	69.2956577442
D	26	36.7080959897	38	73.4161919794
Eb	27	38.8908729653	39	77.7817459305
E	28	41.2034446141	40	82.4068892282

F	29	43.6535289291	41	87.3070578583
Gb	30	46.2493028390	42	92.4986056779
G	31	48.9994294977	43	97.9988589954
Ab	32	51.9130871975	44	103.8261743950
A	33	55.0000000000	45	110.0000000000
Bb	34	58.2704701898	46	116.5409403795
B	35	61.7354126570	47	123.4708253140
C	48	130.8127826503	60	261.6255653006
Db	49	138.5913154884	61	277.1826309769
D	50	146.8323839587	62	293.6647679174
Eb	51	155.5634918610	63	311.1269837221
E	52	164.8137784564	64	329.6275569129
F	53	174.6141157165	65	349.2282314330
Gb	54	184.9972113558	66	369.9944227116
G	55	195.9977179909	67	391.9954359817
Ab	56	207.6523487900	68	415.3046975799
A	57	220.0000000000	69	440.0000000000
Bb	58	233.0818807590	70	466.1637615181
B	59	246.9416506281	71	493.8833012561
C	72	523.2511306012	84	1046.5022612024
Db	73	554.3652619537	85	1108.7305239075
D	74	587.3295358348	86	1174.6590716696
Eb	75	622.2539674442	87	1244.5079348883
E	76	659.2551138257	88	1318.5102276515
F	77	698.4564628660	89	1396.9129257320
Gb	78	739.9888454233	90	1479.9776908465
G	79	783.9908719635	91	1567.9817439270
Ab	80	830.6093951599	92	1661.2187903198
A	81	880.0000000000	93	1760.0000000000
Bb	82	932.3275230362	94	1864.6550460724
B	83	987.7666025122	95	1975.5332050245
C	96	2093.0045224048	108	4186.009044809
Db	97	2217.4610478150	109	4434.922095630

D	98	2349.3181433393	110	4698.636286678
Eb	99	2489.0158697766	111	4978.031739553
E	100	2637.0204553030	112	5274.040910605
F	101	2793.8258514640	113	5587.651702928
Gb	102	2959.9553816931	114	5919.910763386
G	103	3135.9634878540	115	6271.926975708
Ab	104	3322.4375806396	116	6644.875161279
A	105	3520.0000000000	117	7040.000000000
Bb	106	3729.3100921447	118	7458.620234756
B	107	3951.0664100490	119	7902.132834658
C	120	8372.0180896192		
Db	121	8869.8441912599		
D	122	9397.2725733570		
Eb	123	9956.0634791066		
E	124	10548.0818212118		
F	125	11175.3034058561		
Gb	126	11839.8215267723		
G	127	12543.8539514160		

Chapter 4 : Putting All Of This Into Terms On The Mandolin

"So," you may be asking, "What does all this abstract information actually have to do with anything". Well, that is a very interesting question with an even more interesting answer. In, the long run, nothing. Ideally, you should get so comfortable with all this information that you forget you ever learned it and it becomes second nature to your playing. The fundamentals of music theory can be hard to grasp, but they will improve the way you think about and play music in ways you cannot imagine. Knowing the information above will help you connect everything you play on the mandolin with everything else you play on the mandolin and any other instrument, including singing. Knowing your scale theory will help you play in different keys, and your interval theory will help your overall ability to play on the fretboard, whether you are playing a simple scale on the first few frets or a crazy blues solo up above the 15th fret on your

highest string. It will also help you stay grounded when the basics of theory come up (as it inevitably will) in my lessons.

Because the mandolin is tuned in fifths (G to D, D to A, A to E), every pattern you learn or come across can be transposed up, down, and across strings to any key with relative ease. Unlike many guitar players, good mandolin players do not need to rely on capos to play in some funny keys like B.

4.1 What is a mandolin?

A **mandolin** is a **plucked string instrument** in the **lute** family.

Plucked string instruments are a subcategory of string instruments that are played by plucking the strings.

Plucking is a way of pulling and releasing the string in such as way as to give it an impulse that causes the string to vibrate. Plucking can be done with either a finger or a plectrum.

Most plucked string instruments belong to the lute family (such as guitar, bass guitar, mandolin, banjo, etc.), which generally consist of a resonating body, and a neck; the strings run along the neck and can be stopped at different pitches.

4.2 History of a mandolin

Mandolins evolved as part of the Lute family in Italy during the 17th -18th centuries, and the deep bowled mandolin produced particularly in Naples became a common type in the 19th century. The original instrument was the **mandola** (mandorla is almond in Italian and describes the instrument body shape) and evolved in the 15th century from the Lute.

A later, smaller mandola was developed and became known as a mandolina. The 20th century saw the rise in popularity of the mandolin for celtic, bluegrass, jazz and classical styles. Much of the development of the mandolin from neapolitan bowl back to the flat back style is thanks to Orville Gibson (1856 - 1918) and Lloyd Loar, the chief designer for the Gibson Mandolin-Guitar Manufacturing Co Ltd.

4.3 Mandolin Anatomy

We could say that this instrument is a combination of a violin, guitar, lute, and banjo rolled into one. While the lute’s body is round, the mandolin, although made in a number of variations, generally has a hollow body that includes a sounding board with a teardrop shape. Some of the instruments are designed with scrolls or similar-type projections. Mandolins have floating bridges and pin blocks or tailpieces at their edges to which the strings of the instrument are affixed. Instead of pegs, the instrument uses mechanical tuning machines for purposes of tuning. The neck on the instrument is usually flat or has a radius that is imperceptible. The nut appears at the top of the neck which includes a

fretted fingerboard. Metal strings on the instrument come in pairs or double courses of four.

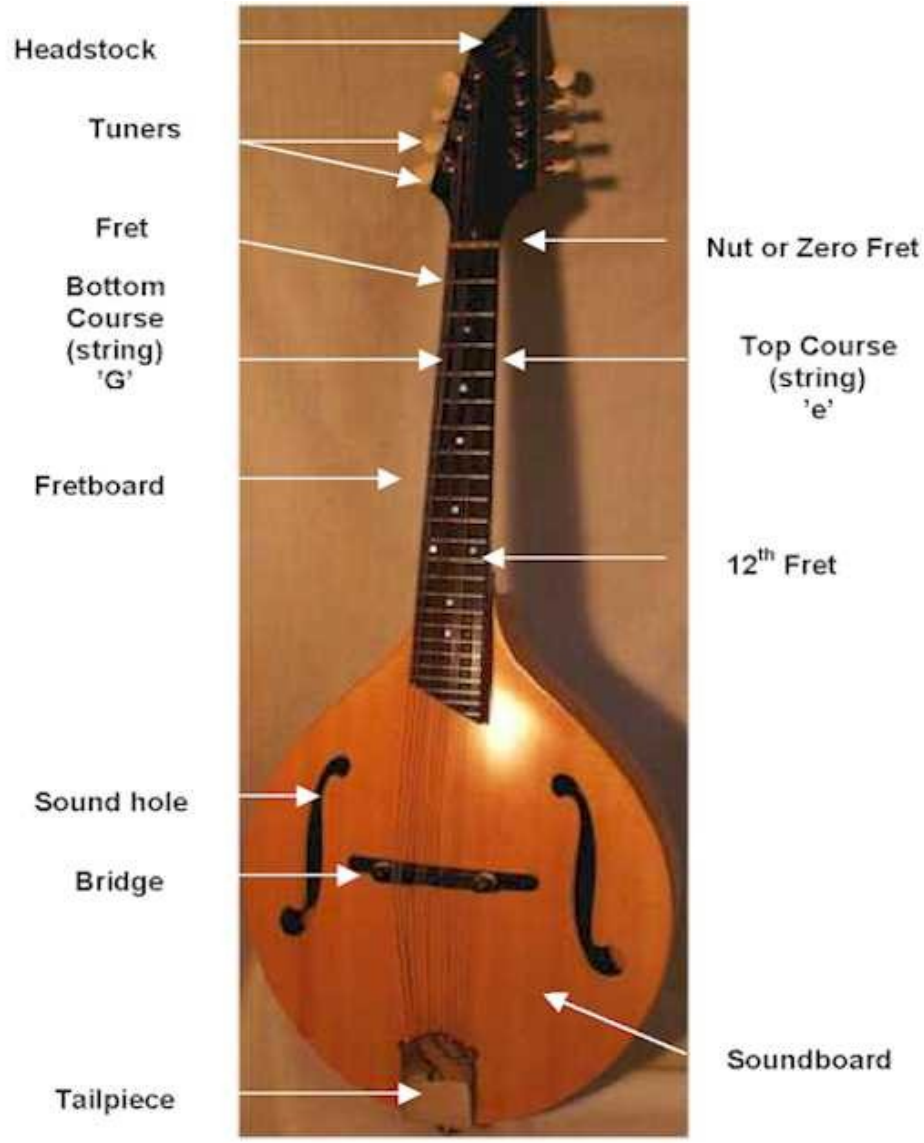

Figure 18: a mandolin instrument1

At the very top is the **headstock**(or **peg head**). The headstock supports eight tuning keys, which are used to tune the eight mandolin strings. Depending on the angle at which your tuning keys are sitting, you might be able to see a small hole in it, through which the string is threaded. The headstock is also usually where the manufacturer puts their name. So if you see someone sneaking a covert glance at your headstock, they're usually trying to find out who made the mandolin you are playing.

The **nut** (or **zero fret**) sits in between the peg head and the neck. If you look closely at the nut, you can see tiny grooves in it. These grooves hold the strings

steady, and keep them from getting pushed all over the place when you touch the strings. These grooves eventually wear down over time. If they become too wide, it allows your strings to roll while you play. If they become too deep, they can prevent your strings from vibrating and your instrument will not get a good sound. These are problems that are easily repaired by a qualified luthier.

The **neck** is the long thin piece that connects the peg head to the body.

The **fretboard** (or **fingerboard**) is on the top side of the neck. - that's the part that your fingertips will touch. The fingerboard should be flat, with no grooves or worn spots.

The fingerboard is bisected over its entire length, by wires that are embedded in it. These wires are called frets. When you play the mandolin, your fingers should go BEHIND the frets, not ON the frets. When you press your finger behind a fret, you push the string tight against the fret. That action changes the sound that the string makes, and so produces the note you're playing.

The **bridge** serves two purposes. The first is to act as a partner with the nut, and keep your strings from rolling while you play. The bridge also has tiny grooves in which your strings sit and, like the nut, can become worn with age and cause poor sound. The bridge also controls the instrument's "action". The action refers to the distance that the strings sit above the fingerboard. Some players prefer that the strings sit very high and away from the fingerboard, because they hit the strings very hard with their fingertips while they play. If the action is high, then the strings will serve as a sort of trampoline, to cushion the impact of their fingertips on the fingerboard. Another player with a gentler touch might find the same mandolin difficult to play, because they have to press extremely hard to hold the string down against the fret. There is no good or bad action – it's strictly a matter of personal preference. But if you find your mandolin physically difficult to play, you might want to consider having the action adjusted, which is accomplished by slightly raising or lowering the height of the bridge. Some bridges have little wheels on the sides – that's why they're there. Others have small holes through which a wrench is used to adjust the height.

The **tailpiece** is ornamental – it covers the ends of the strings. It is the area where the strings attach to the instrument below the bridge or at the base of the instrument. It is usually removed fairly easily – on most mandolins, it just slides (not pulls) off. Hold the body of your mandolin firmly with one hand, and, using a sliding motion, gently push the tailpiece away from the body. The tailpiece is intended to be taken on and off, so that you can put new strings on your instrument.

The **sound hole** on a mandolin is used to project musical quality more effectively. While the vibrations produced come primarily from the surface of the sounding board, the sound holes aid in permitting some of the sound that is generated within the mandolin to emanate outside of it. Sound holes come in various shapes.

4.4 GDAE Chords & Fretboard of the Mandolin

Figure 18: Fretboard of a mandolin

Each fret is a semitone. The instruments are tuned so that the next highest string sounds the same as the one below fretted at the 7th fret. (This means that they are tuned in *5ths.*

Explanation:

G and D are 5 whole notes apart [G-A-B-C-D] but in total, there are actually 7 semitones (frets) [G-G#-A-Bb-B-C-C#-D] with G counting as zero).

Table 5: Notes on the fretboard

String/ Fret	0	1st	2	3rd	4	5th	6	7th	8	9	10	11	12
1(thinnest)	E	F	F#	G	G#	A	Bb	B	C	C#	D	D#	E
2	A	Bb	B	C	C#	D	D#	E	F	F#	G	G#	A
3	D	D#	E	F	F#	G	G#	A	Bb	B	C	C#	D
4(thickest)	G	G#	A	Bb	B	C	C#	D	D#	E	F	F#	G
	nut	-	-	-	-	-	-	-	-	-	-	-	Octave

The notes from the nut to the 12th fret on a string is called an octave and has 12 semitones and 8 whole tones (hence Octave). The octave note is the same note as the open string but a whole set of notes higher in pitch. The length of the string determines the pitch (frequency) of the basic note (open string). A longer string gives a slower frequency, and hence a lower note. Holding a string down at a certain fret shortens the string and makes it sound higher in pitch. The frets are positioned at just the right interval to ensure that the notes sound correctly and have the right relationship with each other. (This is called the Well Tempered Scale - not all instruments are well tempered - in more ways than one!)

The mandolin family of instruments can be tuned easily by getting the bottom (G) string(s) in tune and then tuning each higher course to the previous string at the 7th fret - they should sound the same. You can get the G in tune from a piano, another musician, a tuning fork or an electronic tuner.

Middle C on a mandolin is 4th string, 5th fret which gives you some idea of the range of the instrument. Octave mandolins are a whole octave lower, so middle C occurs at 2nd string, 3rd fret.

4.5 Tuning of a mandolin

The most common tuning by far is GDAE. Mandolin is tuned in fifths starting with the G strings as the lowest strings:

- fourth (lowest tone) course: G3 (196.00 Hz)
- third course: D4 (293.66 Hz)
- second course: A4 (440.00 Hz; A above middle C)
- first (highest tone) course: E5 (659.25 Hz)

The thickest pair (at the top for a right-handed player) are tuned to the note G above middle C. The next pair are tuned to the D above, the next to the A above that and the thinnest pair are tuned to the E above that. The strings are tuned in unison (both in the pair are tuned to the same note). The Mandolin can be tuned to an electronic tuner, another instrument (e.g. piano), or to itself.

Here are diagrams illustrating this tuning sequence:

Figure 19: Tuning Sequence

Figure 20: tuning sequence

Chapter 5 : MoMu-STK

5.1 General Information

5.1.1 What is MoMu-STK?

MoMu-STK (short for Mobile Music Synthesis Toolkit) is a light-weight software toolkit for creating musical instruments and experiences on mobile device, and currently supports the iPhone platform (iPhone, iPad, iPod Touches). MoMu provides API's for real-time full-duplex audio, accelerometer, location, multi-touch, networking, graphics, and utilities. The MoMu Toolkit was developed as part of the Mobile Music research initiative in Music, Computing & Design group at Stanford University's CCRMA, in collaboration with Smule.

5.1.2 What is the *Synthesis ToolKit (STK)*?

The Synthesis ToolKit in C++ (STK) is a set of open source audio signal processing and algorithmic synthesis classes written in the C++ programming language. STK was designed to facilitate rapid development of music synthesis and audio processing software, with an emphasis on cross-platform functionality, real time control, ease of use, and educational example code. The Synthesis ToolKit is extremely portable (it's mostly platform-independent C and C++ code), and it's completely user-extensible (all source included, no unusual libraries, and no hidden drivers). STK currently runs with real time support (audio and MIDI) on Linux, Macintosh OS X, and Windows computer platforms. Generic, non-realtime support has been tested under NeXTStep, Sun, and other platforms and should work with any standard C++ compiler.

The Synthesis ToolKit is free. The only parts of the Synthesis ToolKit that are platform-dependent concern real-time audio and MIDI input and output, and that is taken care of with a few special classes. The interface for MIDI input and the simple Tcl/Tk graphical user interfaces (GUIs) provided is the same, so it's easy to experiment in real time using either the GUIs or MIDI. The Synthesis ToolKit can generate simultaneous SND (AU), WAV, AIFF, and MAT-file output soundfile formats (as well as real time sound output), so you

can view your results using one of a large variety of sound/signal analysis tools already available (e.g. Snd, Cool Edit, Matlab).

5.1.3 What the *Synthesis ToolKit* is not.

The Synthesis Toolkit is not one particular program. Rather, it is a set of C++ classes that you can use to create your own programs. A few example applications are provided to demonstrate some of the ways to use the classes. If you have specific needs, you will probably have to either modify the example programs or write a new program altogether. Further, the example programs don't have a fancy GUI wrapper. It is easy to embed STK classes inside a GUI environment but we have chosen to focus our energy on the audio signal processing issues. Spending hundreds of hours making platform-dependent graphical user interfaces would go against one of the fundamental design goals of the ToolKit - platform independence.

For those instances where a simple GUI with sliders and buttons is helpful, we use Tcl/Tk (that is freely distributed for all the supported ToolKit platforms). A number of Tcl/Tk GUI scripts are distributed with the ToolKit release. For control, the Synthesis Toolkit uses raw MIDI (on supported platforms), and SKINI (Synthesis ToolKit Instrument Network Interface, a MIDI-like text message synthesis control format).

5.1.4 A brief history of the *Synthesis ToolKit in C++*.

Perry Cook began developing a pre-cursor to the Synthesis ToolKit (also called STK) under NeXTStep at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University in the early-1990s. With his move to Princeton University in 1996, he ported everything to C++ on SGI hardware, added real-time capabilities, and greatly expanded the synthesis techniques available. With the help of Bill Putnam, Perry also made a port of STK to Windows95. Gary Scavone began using STK extensively in the summer of 1997 and completed a full port of STK to Linux early in 1998. He finished the fully compatible Windows port (using Direct Sound API) in June 1998. Numerous improvements and extensions have been made since then.

The Toolkit has been distributed continuously since 1996 via the Princeton Sound Kitchen, Perry Cook's home page at Princeton, Gary Scavone's home page at McGill University, and the Synthesis ToolKit home page. The ToolKit has been included in various collections of software. Much of it has also been ported to Max/MSP on Macintosh computers by Dan Trueman and Luke Dubois of Columbia University, and is distributed as PeRColate. Help on real-time sound and MIDI has been provided over the years by Tim Stilson, Bill Putnam, and Gabriel Maldonado.

5.2 Class Documentation

5.2.1. Delay Class Reference

Non-Interpolating Delay Line Class.

Inheritance Relationship for Delay: Delay->Filter->Object

Public Methods

• Delay(long max_length)

Constructor which specifies the maximum delay-line length.

• Delay()

Class destructor.

• void clear ()

Clears the internal state of the delay line.

• void setDelay (StkFloat length)

Set the delay-line length.

• StkFloat tick(StkFloat sample)

Input one sample to the delay-line and return one output.

5.2.1.1 Detailed Description

Non-interpolating delay line class.

This protected Filter subclass implements a non-interpolating delay-line.

This one uses a delay line of maximum length specified on creation.

A non-interpolating delay line is typically used in fixed delay-length applications,

such as for reverberation.

5.2.1.2 Member Function Documentation

5.2.1.2.1 void Delay::setDelay (StkFloat length)

Set the delay-line length.

The valid range for length is from 0 to the maximum delay-line length.

The documentation for this class was generated from the following file:

• Delay.h

5.2.2. DelayA Class Reference

Allpass interpolating delay line class.

Inheritance relationship DelayA: DelayA->Filter->Object

Public Methods

• DelayA (long max_length)

Constructor which specifies the maximum delay-line length.

• DelayA ()

Class destructor.

• void clear ()

Clears the internal state of the delay line.

• void setDelay (StkFloat length)

Set the delay-line length.

•StkFloat tick (StkFloat sample)

Input one sample to the delay-line and return one output.

5.2.2.1. Detailed Description

Allpass interpolating delay line class.

This one uses a delay line of maximum length specified on creation and interpolates fractional length using an all-pass filter. This version is more efficient for computing static length delay lines (alpha and coeff are

computed only when the length is set, there probably is a more efficient computational form if alpha is changed often (each sample)).

5.2.2.2 Member Function Documentation

5.2.2.2.1 void DelayA::setDelay (StkFloat length)

Set the delay-line length.

The valid range for length is from 0.5 to the maximum delay-line length.

The documentation for this class was generated from the following file:

• DelayA.h

5.2.3. DelayL Class Reference

Linearly Interpolating Delay Line Class

Inheritance relationship for DelayL:: DelayL->Filter->Object

Public Methods

• DelayL (long max_length)

Constructor which specifies the maximum delay-line length.

• DelayL ()

Class destructor.

• void clear ()

Clears the internal state of the delay line.

• void setDelay (StkFloat length)

Set the delay-line length.

• StkFloat tick (StkFloat sample)

Input one sample to the delay-line and return one output.

5.2.3.1. Detailed Description

Linearly interpolating delay line class.

This Delay subclass uses a delay line of maximum length specified on creation, and linearly interpolates fractional length. It is designed to be more efficient if the delay length is not changed very often.

5.2.3.2. Member Function Documentation

5.2.3.2.1 void DelayL::setDelay (StkFloat length)

Set the delay-line length.

The valid range for length is from 0 to the maximum delay-line length.

The documentation for this class was generated from the following file:

• DelayL.h

5.2.4. Filter Class Reference

Filter class.

Inheritance relatioship for Filter::

Delay, DelayA, DelayL, OneZero ->Filter->Object

Public Methods

• Filter ()

Default constructor creates a zero-order pass-through ”filter”.

• Filter ()

Class destructor.

• StkFloat lastOut ()

Return the last computed output value.

5.2.4.1. Detailed Description

Filter class.

This class is a base class for all filters. This filter is defined as something which has inputs(s), output(s), and gain.

The documentation for this class was generated from the following file:

• Filter.h

5.2.5. Instrmnt Class Reference

Instrument base class.

Inheritance relationship for Instrmnt::

Mandolin, PluckTwo ->Instmnt->Object

Public Methods

• Instrmnt ()

Default constructor.
•virtual void noteOn (StkFloat frequency, StkFloat amplitude)
Start a note with the given frequency and amplitude.
• virtual void noteOff (StkFloat amplitude)
Stop a note with the given amplitude.
• virtual void setFreq (StkFloat frequency)
Set instrument parameters for a particular frequency.
• Stkfloat lastOut ()
Return the last output value.
• virtual StkFloat tick ()
Compute one output sample.
• virtual void controlChange (int number, StkFloat value)
Perform the control change specified by number and value (0.0 – 128.0).

5.2.5.1. Detailed Description

Instrument base class.

This class provides an interface for the mandolin instrument.

The documentation for this class was generated from the following file:

• Instrmnt.h

5.2.6. Mandolin Class Reference

Commuted Mandolin Subclass of enhanced dual plucked-string model

Inheritance relationship for Mandolin::

Mandolin->Instrmnt->Object

Public Methods

• Mandolin (StkFloat lowestFrequency)

Class constructor, taking the lowest desired playing frequency.
• void pluck (StkFloat amplitude)
Pluck the strings with the given amplitude (0.0 - 1.0) using the current frequency.
• void pluck (StkFloat amplitude, StkFloat position)
Pluck the strings with the given amplitude (0.0 - 1.0) and position (0.0 - 1.0).
• virtual void noteOn (StkFloat frequency, StkFloat amplitude)
Start a note with the given frequency and amplitude (0.0 - 1.0).
• void setBodySize (StkFloat size)
Set the body size.
• virtual StkFloat tick ()
Compute one output sample.
• virtual void controlChange (int number, StkFloat value)
Perform the control change specified by number and value (0.0 – 128.0).

5.2.6.1 Detailed Description

Commuted Mandolin Subclass of enhanced dual plucked-string model.

This class inherits from PluckTwo and uses ”commuted synthesis” techniques

to model a mandolin instrument.

Control Change Numbers:

• Body Size = 2

• Pluck Position = 4

• String Sustain = 11

• String Detuning = 1

• Microphone Position = 128

The documentation for this class was generated from the following file:

• Mandolin.h

5.2.7. Noise Class Reference

Noise Generator Class.

Inheritance relationship for Noise:: Noise->Object

Public Methods

• Noise ()

Default constructor.

• virtual Noise ()

Class destructor.

• virtual StkFloat tick ()

Return a random number between -1.0 and 1.0 using the random() function.

• StkFloat lastOut ()

Return the last computed value.

5.2.7.1. Detailed Description

Noise Generator Class.

Generic random number generation using the random() function. The quality of

the random() function varies from one OS to another.

The documentation for this class was generated from the following file: Noise.h

5.2.8. Object Class Reference

Object Base Class.

This is mostly here for compatibility with Objective C.

5.2.8.1. Detailed Description

The documentation for this class was generated from the following file “Object.h”.

This file defines sampling rates, basic trigonometric constants, states for envelopes, MIDI definitions and generally all the necessary definitions for our program.

5.2.9. OneZero Class Reference

One Zero Filter Class.

Inheritance relationship for OneZero:: OneZero->Filter->Object

Public Methods

• OneZero ()

Default constructor creates a first-order low-pass filter.

• OneZero ()

Class destructor.

• void clear ()

Clears the internal state of the filter.

• void setCoeff (StkFloat aValue)

Set the coefficient value.

• void setGain (StkFloat aValue)

Set the filter gain.

• StkFloat tick (StkFloat sample)

Input one sample to the filter and return one output.

5.2.9.1. Detailed Description

One Zero Filter Class.

The parameter gain is an additional gain parameter applied to the filter on top of the normalization that takes place automatically. So the net max gain through the system equals the value of gain. sgain is the combination of gain and the normalization parameter, so if you set the poleCoeff to alpha, sgain is always set to gain / (1.0 - fabs(alpha)).

5.2.9.2. Member Function Documentation

5.2.9.2.1 void OneZero::setGain (StkFloat aValue) [virtual]

Set the filter gain.

The gain is applied at the filter input and does not affect the coefficient values.

The default gain value is 1.0.

Reimplemented from Filter.

The documentation for this class was generated from the following file:

• OneZero.h

5.2.10. PlayMusic Class Reference

PlayMusic Class.

This class is aimed to control commuted dual plucked-string model.

Inheritance relationship for PlayMusic::

PlayMusic->Instrmnt->Object

Public Methods

•PlayMusic()

Default constructor.
•PlayMusic()
Class destructor.
• virtual void noteOnN(short num, StkFloat amplitude)
Start a note with the given num and amplitude.
• virtual void noteOffN(short num, StkFloat amplitude)
Stop a note with the given num and amplitude.
• virtual StkFloat tick()
Compute and return one output sample.
• virtual void controlChange(int number, StkFloat value)
Perform the control change specified by *number* and *value* (0.0 - 128.0).
• virtual void Mandolin_Chords(StkFloat amplitude, char* chordString)
Choose Mandolin chords.

The documentation for this class was generated from the following file:

• PlayMusic.h

5.2.11. PluckTwo Class Reference

Karplus-Strong plucked model class.

Inheritance relationship for PluckTwo::

PluckTwo-> Instrmnt->Object

Public Methods

• PluckTwo (StkFloat lowestFrequency)

Class constructor, taking the lowest desired playing frequency.
• PluckTwo ()
Class destructor.
• void clear ()
Reset and clear all internal state.
• virtual void setFreq (StkFloat frequency)
Set instrument parameters for a particular frequency.
• void setDetune(StkFloat detune)

detune the two strings by the given factor
• void setFreqAndDetune(StkFloat frequency, StkFloat detune)
efficient combined setting of frequency and detuning
• void setPluckPos(StkFloat position)
set the pluck position along the string
• void setBaseLoopGain(StkFloat aGain)
set the base loop gain
• virtual void noteOff (StkFloat amplitude)
Stop a note with the given amplitude.

5.2.11.1 Detailed Description

Karplus-Strong plucked model class.

This class implements a plucked physical model based on the Karplus-Strong algorithm.

The documentation for this class was generated from the following file:

• PluckTwo.h

5.2.12. RawWave Class Reference

RawWave Soundfile Class

Inheritance relationship for RawWave: RawWave->Object

Public Methods

• RawWave(char *fileName)

Class Constructor taking the name of the soundfile.

• RawWave(StkFloat *someData,long aLength)

Overloaded Constructor taking the data and the length.

• RawWave()

Class Destructor

•void reset()

Clear output and reset time pointer to zero.

• void normalize()

Normalize data to a maximum of +-1.0.

• void normalize(StkFloat peak)

Normalize data to a maximum of +-peak.

• void setRate(StkFloat rate)

Set the data read rate in samples.

• void setFreq(StkFloat aFreq)

Set the data interpolation rate based on a looping frequency.

• void addTime(StkFloat time)

Increment the read pointer by time in samples.

• void addPhase(StkFloat anAngle)

Increment current read pointer by anAngle, relative to a looping frequency.

• void addPhaseOffset(StkFloat anAngle)

Add a phase offset to the current read pointer.

• void setLooping(int aLoopStatus)

Set the looping variable.

• StkFloat tick()

Compute and return one output sample.

• int informTick()

Update current time.

• StkFloat lastOut()

Return the last computed output value.

5.2.12.1 Detailed Description

RawWave Sound file Class.

This class can open a raw 16bit data (signed integers) file, and play back the data once or looping, with linear interpolation on playback.

5.2.12.2 Member Function Documentation

5.2.12.2.1 void RawWave::setRate(StkFloat rate)

Set the data read rate in samples. The rate can be negative. If the rate value is negative, the data is read in reverse order.

5.2.12.2.2 void RawWave::setFreq(StkFloat aFreq)

Set the data interpolation rate based on a looping frequency.

5.2.12.2.3 void RawWave:: addPhase(StkFloat anAngle)

Increment current read pointer by anAngle, relative to a looping frequency. In this function we add time in cycles. As one cycle we consider the length.

5.2.12.2.4 void RawWave:: addPhaseOffset(StkFloat anAngle)

Add a phase offset to the current read pointer. In this function we add a phase offset in cycles. As one cycle we consider the length.

The documentation for this class was generated from the following file:

- RawWave.h

5.2.13. RawWvOut Class Reference

Raw Wave File Output Class

Inheritance relationship for RawWvOut : RawWvOut->Object

Public Methods

• RawWvOut (char *fileName)

Constructor taking the soundfile.

• RawWvOut ()

Class destructor.

• long getCounter()

Return the number of output.

•StkFloat getTime ()

Return the number of seconds of data output.

• void tick (StkFloat sample)

Output a sample in vector.

5.2.13.1 Detailed Description

This class opens a mono NeXT .snd file 16bit data at 22KHz, and pokes buffers of samples into it.

The documentation for this class was generated from the following file:
• RawWvOut.h

5.2.14. SKINI Class Reference

SKINI Text File Reader Class

Inheritance relationship for SKINI:: SKINI->Object

Public Methods

• SKINI ()

Default constructor used for parsing messages received externally.
• SKINI (char *fileName)
Constructor taking a SKINI formatted scorefile.
• SKINI ()
Class destructor.
• long parseThis (char* aString)
Attempt to parse the given string, returning the message type.
• long nextMessage ()
Parse the next message (if a file is loaded) and return the message type.
• long getType ()
Return the current message type.
• long getChannel ()
Return the current message channel value.
• StkFloat getDelta ()
Return the current message delta time value (in seconds).
• StkFloat getByteTwo ()
Return the current message byte two value.
• StkFloat getByteThree ()

Return the current message byte three value.
• long getByteTwoInt ()
Return the current message byte two value (integer).
• long getByteThreeInt ()
Return the current message byte three value (integer).
• char* getRemainderString ()
Return remainder string after parsing.
• char* getMessageTypeString ()
Return the message type as a string.
• char* whatsThisType (long type)
Return the SKINI type string for the given type value.
• char* whatsThisController (long type)
Return the SKINI controller string for the given controller number.

5.2.14.1 Detailed Description

This class can open a SKINI File and parse it.

SKINI (Synthesis ToolKit Instrument Network Interface) is like MIDI, but allows for floating point control changes, note numbers, etc.

Example: noteOn 60.01 111.132 plays a sharp middle C with a velocity of 111.132

The documentation for this class was generated from the following file:

• SKINI.h

5.2.15. Voice Class Reference

Voice manager class.

Inheritance relationship for Voice:: Voice->Object

Public Methods

• Voice(int maxVoices, char *instrType)

Class constructor taking the maximum number of voices and the desirable instrument(in my case the mandolin).

• Voice ()
Class destructor.
• long noteOnN (StkFloat noteNum, StkFloat amplitude)
Initiate a noteOn event with the given note number and amplitude and return a unique note tag.
• long noteOn(StkFloat frequency, StkFloat amplitude)
Initiate a noteOn event with the given frequency and amplitude and return a unique note tag.
• void noteOffN (int note_num, StkFloat amplitude
Send a noteOff to all voices having the given note number and amplitude.
• void noteOffT (long tag, StkFloat amplitude)
Send a noteOff to the voice with the given note tag.
• void pitchBend (StkFloat value)
Send a pitchBend message to all voices.
• void pitchBend (long tag, StkFloat value)
Send a pitchBend message to the voice with the given note tag.
• void controlChange (int number, StkFloat value)
Send a controlChange to mandolin instrument.
• void controlChange (long tag, int number, StkFloat value)
Send a controlChange to the voice with the given note tag.
• void kill (long tag)
Send a noteOff message to the voice with the given note tag.
•long oldestVoice()
Return the tag of the sounding note.
• StkFloat tick ()
Mix the output for all sounding voices.

5.2.15.1 Detailed Description

Voice manager class.

We just need to define the maximum number of voices we want and the mandolin instrument will be mangling. We then pipe SKINI messages into it and it will return the mixed channel signal each tick. Each noteOn returns a unique tag, so we can send control changes to unique instances of mandolin within an ensemble. SKINI (Synthesis toolKit Instrument Network Interface) is like MIDI, but allows for floating point control changes, note numbers, etc.

Example: noteOn 60.01 111.132 plays a sharp middle C with a velocity of 111.132.

5.2.15.2 Member Function Documentation

5.2.15.2.1 long Voice::noteOnN (StkFloat note_num, StkFloat amplitude)

Initiate a noteOn event with the given note number and amplitude and return a unique note tag. If no voices are found, the function returns -1. The amplitude value should be in the range 0.0 - 128.0.

5.2.15.2.2 void Voice::noteOffN (StkFloat note_num, StkFloat amplitude)

Send a noteOff to all voices having the given note number and amplitude.

The amplitude value should be in the range 0.0 - 128.0.

5.2.15.2.3 void Voice::noteOffT (long tag, StkFloat amplitude)

Send a noteOff to the voice with the given note tag.

The amplitude value should be in the range 0.0 - 128.0.

The documentation for this class was generated from the following file:

• Voice.h

5.3 Usage Documentation

Directory Structure

The top level distribution contains the following directories:

- The ***src*** directory contains the source .cpp files for all the algorithm classes.
- The ***include*** directory contains the header files for all the algorithm classes.
- The ***rawwaves*** directory contains my music.raw used with the classes.

5.4 Compiling

The Synthesis ToolKit can be used in a variety of ways, depending on your particular needs. Some people choose the classes they need for a particular project and copy those to their working directory. Others create Makefiles that compile project-specific class objects from common src and include directories. And still others like to compile and link to a common library of object files. STK was not designed with one particular style of use in mind.

My approach in using STK was to simply copy the class files needed for my program into a project directory named PlayMusic. Some of the classes were

modified for the need of my program, while others remained as they were. The program was then compiled on Linux, using the GNU g++ compiler as follows:

```
g++ -g -Wall -lm *.CPP -o PlayMusic
```

5.5 Control Input

Each Synthesis ToolKit instrument, in this case the mandolin, exposes its relevant control parameters via public functions such as setFrequency() and controlChange(). Generally programmers are free to implement the control scheme of their choice in exposing those parameters to the user.

A text-based control protocol called SKINI is provided with the Synthesis ToolKit. SKINI extends the MIDI protocol in incremental ways, providing a text-based messaging scheme in human-readable format and making use of floating-point numbers wherever possible. Each SKINI message consists of a message type (e.g., NoteOn, PitchBend), a time specification (absolute or delta), a channel number (scanned as a long integer), and a maximum of two subsequent message-specific field values. Knowing this, it should be relatively clear what the following SKINI "scorefile" specifies:

```
NoteOn        0.000082 2 55.0 82.3

NoteOff       1.000000 2 55.0 64.0

NoteOn        0.000082 2 69.0 82.8

StringDetune 0.100000 2 10.0

StringDetune 0.100000 2 30.0

StringDetune 0.100000 2 50.0

StringDetune 0.100000 2 40.0

StringDetune 0.100000 2 22.0

StringDetune 0.100000 2 12.0

NoteOff          1.000000 2 69.0 64.0
```

MIDI messages are easily represented within the SKINI protocol.

Since the program is compiled as PlayMusic and we allow control via SKINI messages read from a SKINI scorefile, for example, testscor.ski the program can be run as:

```
./PlayMusic testscor.ski
```

Chapter 6 : Synthesis ToolKit Instrument Network Interface (SKINI)

6.1 MIDI Compatibility

SKINI was designed to be MIDI compatible wherever possible, and extend MIDI in incremental, then maybe profound ways.

Differences from MIDI, and motivations, include:

• Text-based messages are used, with meaningful names wherever possible. This allows any language or system capable of formatted printing to generate SKINI. Similarly, any system capable of reading in a string and turning delimited fields into strings, floats, and integers can consume SKINI for control. More importantly, humans can actually read, and even write if they want, SKINI files and streams. Use an editor and search/replace or macros to change a channel or control number. Load a SKINI score into a spread sheet to apply transformations to time, control parameters, MIDI velocities, etc.

• Floating point numbers are used wherever possible. Note Numbers, Velocities, Controller Values, and Delta and Absolute Times are all represented and scanned as ASCII double-precision floats. MIDI byte values are preserved, so that incoming MIDI bytes from an interface can be put directly into SKINI messages. 60.0 or 60 is middle C, 127.0 or 127 is maximum velocity etc. But, unlike MIDI, 60.5 can cause a 50 cent sharp middle C to be played. As with MIDI byte values like velocity, use of the integer and SKINI-added fractional parts is up to the implementor of the algorithm being controlled by SKINI messages. But the extra precision is there to be used or ignored.

6.2 Why SKINI?

SKINI was designed to be extensible and hackable for a number of applications: embedded synthesis in a game or VR simulation, scoring and mixing tasks, real- time and non-real time applications which could benefit from controllable sound synthesis, JAVA controlled synthesis, or eventually maybe JAVA synthesis, etc. SKINI is not intended to be ”the mother of score files,” but since the entire system is based on text representations of names, floats, and integers, converters from one scorefile language to SKINI, or back, should be easily created.

6.3 SKINI Messages

A basic SKINI message is a line of text. There are only three required fields, the message type (an ASCII name), the time (either delta or absolute), and the channel number. Don’t think that this is MIDI channel 0- 15 (which is supported), because the channel number is scanned as a long integer. Channels could be socket numbers, machine Ids, serial numbers, or even unique tags for

each event in a synthesis. Other fields might be used, as specified in the SKINI.tbl file. This is described in more detail later. Fields in a SKINI line are delimited by spaces, commas, or tabs. The SKINI parser only operates on a line at a time, so a newline means the message is over.

Multiple messages are NOT allowed directly on a single line (by use of the ; for example in C). Message types include standard MIDI types like NoteOn, NoteOff, ControlChange, etc. MIDI extension message types (messages which look better than MIDI but actually get turned into MIDI-like messages) include StringDamping, etc.

All fields other than type, time, and channel are optional, and the types and usage of the additional fields is defined in the file SKINI.tbl. The other important file used by SKINI is SKINI.msg, which is a set of defines to make C code more readable, and to allow reasonably quick re-mapping of control numbers, etc.. All of these defined symbols are assigned integer values.

6.4 C Files Used To Implement SKINI

SKINI.cpp is an object which can either open a SKINI file, and successively read and parse lines of text as SKINI strings, or accept strings from another object and parse them. The latter functionality would be used by a socket, pipe, or other connection receiving SKINI messages a line at a time, usually in real time, but not restricted to real time. SKINI.msg should be included by anything wanting to use the SKINI.cpp object. This is not mandatory, but use of the SK blah symbols which are defined in the .msg file will help to ensure clarity and consistency when messages are added and changed. SKINI.tbl is used only by the SKINI parser object (SKINI.cpp). In the file SKINI.tbl, an array of structures is declared and assigned values which instruct the parser as to what the message types are, and what the fields mean for those message types. This table is compiled and linked into applications using SKINI, but could be dynamically loaded and changed in a future version of SKINI.

6.5 SKINI Messages and the SKINI Parser

Here are the basic rules governing a valid SKINI message:

• If the first character in a SKINI string is ’/’ that line is treated as a comment.

• If there are no characters on a line, that line is treated as blank. Tabs and spaces are treated as non-characters.

• Spaces, commas, and tabs delimit the fields in a SKINI message line. (We might allow for multiple messages per line later using the semicolon, but probably not. A series of lines with deltaTimes of 0.0 denotes simultaneous events. For read-ability, multiple messages per line doesn’t help much, so it’s unlikely to be supported later).

• The first field must be a SKINI message name (like NoteOn). These might become case-insensitive , so don’t plan on exciting clever overloading of

names (like noTeOn being different from NoTeON). There can be a number of leading spaces or tabs, but don't exceed 32 or so.

• The second field must be a time specification in seconds. A time field can be either delta-time or absolute time. Absolute time messages have an '=' appended to the beginning of the floating point number with no space. So 0.10000 means delta time of 100 ms, while =0.10000 means absolute time of 100 ms. Absolute time messages make most sense in score files, but could also be used for (loose) synchronization in a real-time context. Real-time messages should be time-ordered AND time-correct. That is, if you've sent 100 total delta-time messages of 1.0 seconds, and then send an absolute time message of =90.0 seconds, or if you send two absolute time messages of =100.0 and =90.0 in that order, things will get really fouled up. The SKINI parser doesn't know about time, however. The RawWvOut device is the master time keeper in the Synthesis ToolKit, so it should be queried to see if absolute time messages are making sense. Absolute times are returned by the parser as negative numbers (since negative deltaTimes are not allowed).

• The third field must be an integer channel number. Don't think that this is just MIDI channel 0-15 (which is supported). The channel number is scanned as a long integer. Channels 0-15 are in general to be treated as MIDI channels. After that it's wide open. Channels could be socket numbers, machine Ids, serial numbers, or even unique tags for each event in a synthesis. A -1 channel can be used as don't care, omni, or other functions depending on your needs and taste.

• All remaining fields are specified in the SKINI.tbl file. In general, there are maximum two more fields, which are either SK INT (long), SK DBL (double float), or SK STR (string). The latter is the mechanism by which more arguments can be specified on the line, but the object using SKINI must take that string apart (retrieved by using getRemainderString()) and scan it. Any excess fields are stashed in remainderString.

6.6 A Short SKINI File

```
/*Howdy!!! Welcome to SKINI
*****************************************
  Howdy!!  ToolKit SKINI File  */

StringDamping        0.0 2 127
Chord                0.0 2 100 G
StringDamping        0.2 2 32
StringDamping       =4.0 2  0.0
ChordOff             0.0 2 100
```

Chord	0.2 2 100 G
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 C
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 C
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 G
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 G
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 D
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 D
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 G
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 G
StringDamping	0.2 2 32
StringDamping	0.0 2 127

ChordOff	0.0 2 100
Chord	0.2 2 100 C
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 C
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 G
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 D
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 G
StringDamping	0.2 2 32
StringDamping	0.0 2 127
ChordOff	0.0 2 100
Chord	0.2 2 100 G
StringDamping	0.1 2 32
ChordOff	0.1 2 100
StringDamping	0.0 2 120
Strumming	0.0 2 127
NoteOn	0.1 2 55 60
NoteOff	0.7 2 55 60
NoteOn	0.0 2 60 60
NoteOff	0.4 2 60 60
NoteOn	0.0 2 62 60
NoteOff	0.2 2 62 60
NoteOn	0.0 2 60 60
NoteOff	0.2 2 60 60
NoteOn	0.0 2 59 60

NoteOff	0.4 2 59 60
NoteOn	0.0 2 57 60
NoteOff	0.2 2 57 60
NoteOn	0.0 2 55 60
NoteOff	0.2 2 55 60
NoteOn	0.0 2 62 60
NoteOff	0.8 2 62 60
NoteOn	0.1 2 67 100
NoteOff	0.7 2 67 100
NoteOn	0.0 2 72 100
NoteOff	0.4 2 72 100
NoteOn	0.0 2 74 100
NoteOff	0.2 2 74 100
NoteOn	0.0 2 72 100
NoteOff	0.2 2 72 100
NotStrumming	0.0 2 0
NoteOn	0.0 2 71 100
NoteOff	0.1 2 71 100
NoteOn	0.0 2 76 100
NoteOff	0.1 2 76 100
NoteOn	0.0 2 74 100
NoteOff	0.1 2 74 100
NoteOn	0.0 2 70 100
NoteOff	0.1 2 70 100
NoteOn	0.0 2 69 100
NoteOff	0.1 2 69 100
NoteOn	0.0 2 67 100
NoteOff	0.1 2 67 100
NoteOn	0.0 2 64 100
NoteOff	0.1 2 64 100
NoteOn	0.0 2 62 100
NoteOff	0.1 2 62 100
Chord	0.0 2 64 G
ChordOff	2.0 2 64

6.7 The SKINI.tbl File and Message Parsing

The SKINI.tbl file contains an array of structures which are accessed by the parser object SKINI.cpp. The struct is:

```
struct SKINISpec {
                    char messageString[32];
                    long type;
                    long data2;
                    long data3;
                    };
```

so an assignment of one of these structs looks like: messageString ,type, data2, data3 where type is the message type sent back from the SKINI line parser and data<n> is either:

- NOPE : field not used, specifically, there aren't going to be any more fields on this line. So if there is is NOPE in data2, data3 won't even be checked.
- SK INT : byte (actually scanned as 32 bit signed long integer). If it's a MIDI data field which is required to be an integer, like a controller number, it's 0-127. Otherwise, get creative with SK INTs.
- SK DBL : double precision floating point. SKINI uses these in the MIDI context for note numbers with micro tuning, velocities, controller values, etc.
- SK STR : only valid in final field. This allows (nearly) arbitrary message types to be supported by simply scanning the string to EndOfLine and then passing it to a more intelligent handler. For example, MIDI SYSEX (system exclusive) messages of up to 256 bytes can be read as space-delimited integers into the 1K SK STR buffer. Longer bulk dumps, sound files, etc. should be handled as a new message type pointing to a File Name, Socket, or something else stored in the SK STR field, or as a new type of multi-line message.

Here's a couple of lines from the SKINI.tbl file :

```
{"NoteOff"          , __SK_NoteOff_          , SK_DBL, SK_DBL},
{"NoteOn"           , __SK_NoteOn_           , SK_DBL, SK_DBL},
{"ControlChange"    , __SK_ControlChange_    , SK_INT, SK_DBL},
```

{"Volume" , __SK_ControlChange_, __SK_Volume_ , SK_DBL},

{"StringDamping" , __SK_ControlChange_, __SK_StringDamping_ , SK_DBL},

{"StringDetune" , __SK_ControlChange_, __SK_StringDetune_ , SK_DBL},

The first three are basic MIDI messages. The first two would cause the parser, after recognizing a match of the string ”NoteOff” or ”NoteOn”, to set the message type to 128 or 144 (SK NoteOff and SK NoteOn are defined in the file SKINI.msg to be the MIDI byte value, without channel, of the actual MIDI messages for NoteOn and NoteOff). The parser would then set the time or delta time (this is always done and is therefore not described in the SKINI Message Struct). The next two fields would be scanned as double-precision floats and assigned to the byteTwo and byteThree variables of the SKINI parser. The remainder of the line is stashed in the remainderString variable.

The ControlChange spec is basically the same as NoteOn and NoteOff, but the second data byte is set to an integer (for checking later as to what MIDI control is being changed). The Volume spec is a MIDI Extension message, which behaves like a ControlChange message with the controller number set explicitly to the value for MIDI Volume (7). Thus the following two lines would accomplish the same changing of MIDI volume on channel 2:

ControlChange 0.000000 2 7 64.1

Volume 0.000000 2 64.1

The StringDamping and StringDetune messages behave the same as the Volume message, but use Control Numbers which aren’t specifically nailed-down in MIDI. Note that these Control Numbers are carried around as long integers, so we’re not limited to 0-127. If, however, you want to use a MIDI controller to play an instrument, using controller numbers in the 0-127 range might make sense.

Chapter 7 : Testing my program trying different scorefiles

We allow control via SKINI messages read from six different SKINI score files: CHORDS.SKI, FUNSKINI.SKI, FUNICULA.SKI, TESTSCOR.SKI, SCALES.SKI, PICKDUMP.SKI.

Running the program we notice that different score files give different computation time. Moreover, we take different computation time every time we change a set of rate variables in one scorefile.

These are the rate variables: SRATE

SRATE_OVER_TWO = SRATE/2

ONE_OVER_SRATE = 1/SRATE

RATE_NORM = 22050 / SRATE

Here's a table which shows the computation time of each skini file while changing the SRATE variable

Table 6: Computation time

	COMPUTATION TIME		
SKINI_FILE	SRATE=220.50 Hz	SRATE=441 Hz	SRATE=800 Hz
SCALES.SKI	19.000000 sec	18.500000 sec	20.756250 sec
CHORDS.SKI	14.799910 sec	14.299954 sec	16.556000 sec
FUNSKINI.SKI	17.024172 sec	16.524965 sec	18.776625 sec
FUNICULA.SKI	16.625713 sec	16.128073 sec	18.374750 sec
TESTSCOR.SKI	17.200726 sec	16.701042 sec	18.955999 sec
PICKDUMP.SKI	9.749569 sec	9.250000 sec	11.506250 sec

Simultaneously with the computation time a file test.wav.snd is created, every time I run my program. Different score files give different audio files. Furthermore, we take different audio files every time we change the set of rate variables in one scorefile. In order to open these audio files, I use audacity, a free digital audio editor and recording application, available for Linux.

Here are the waveforms of each scorefile after taking consider all the possible values of the variable SRATE

1) SRATE =22050 Hz

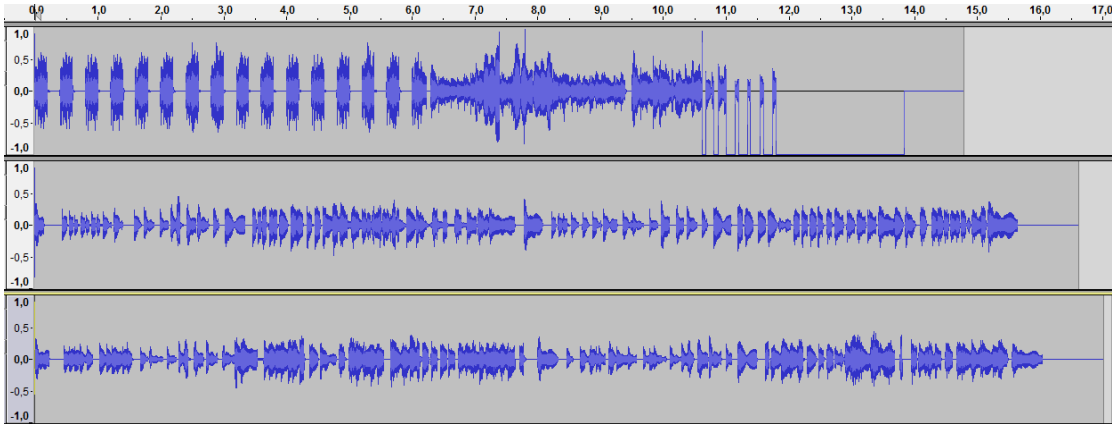

Figure 21: waveforms of CHORDS.SKI, FUNICULA.SKI, FUNSKINI.SKI

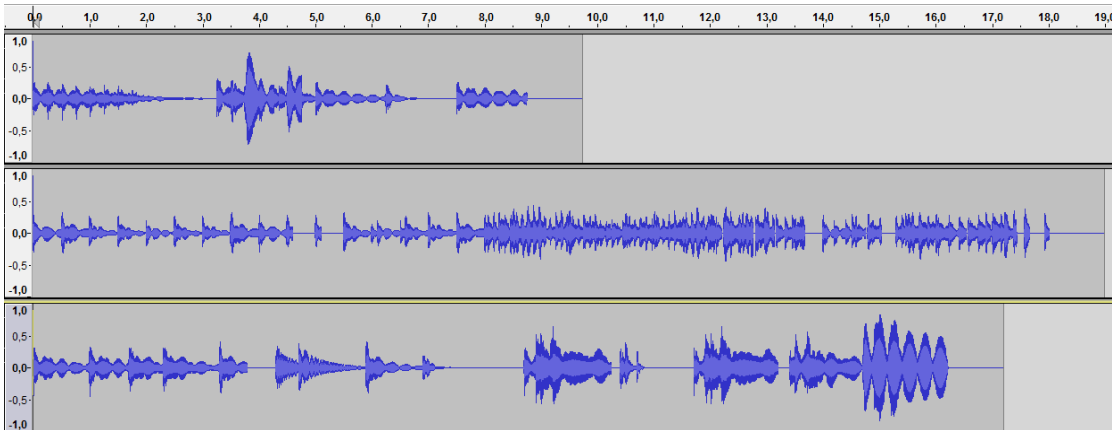

Figure 22: waveforms of PICKDUMP.SKI, SCALES.SKI, TESTSCOR.SKI

2) SRATE =44100 Hz

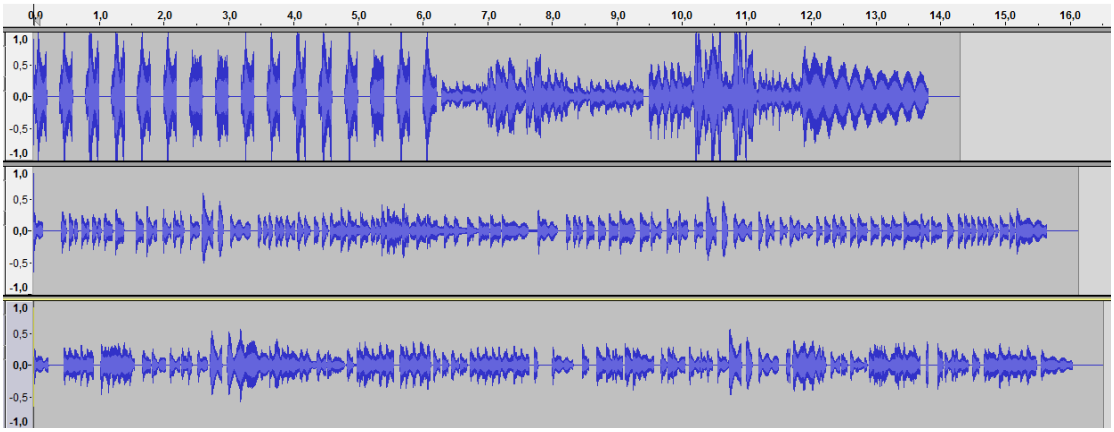

Figure 23: waveforms of CHORDS.SKI, FUNICULA.SKI, FUNSKINI.SKI

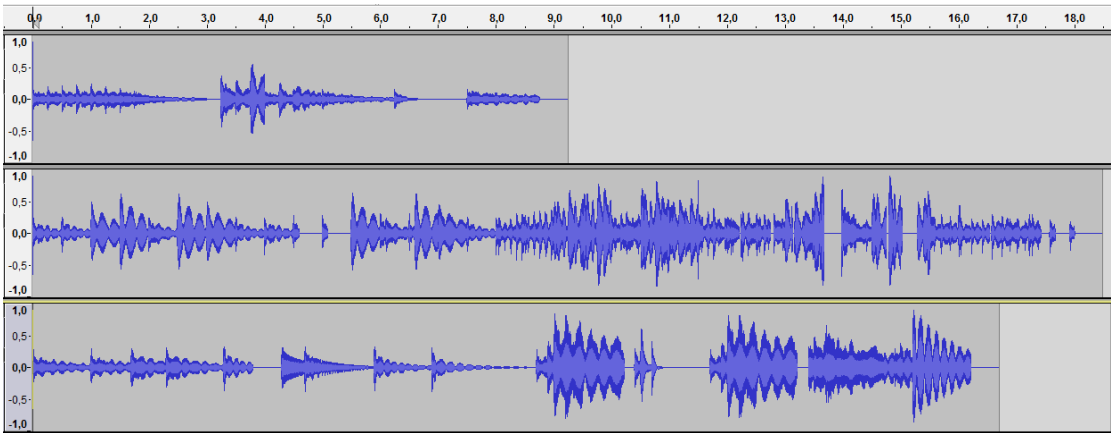

Figure 24: waveforms of PICKDUMP.SKI, SCALES.SKI, TESTSCOR.SKI

3) SRATE = 80000 Hz

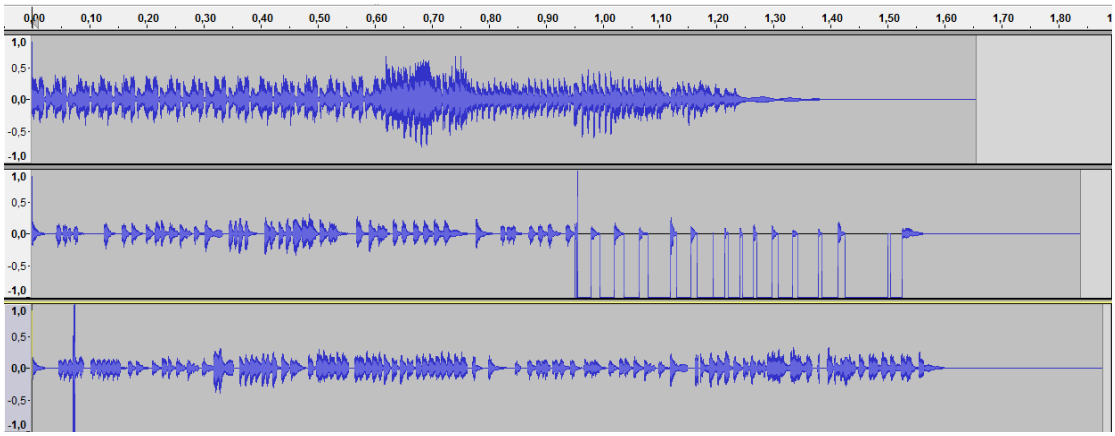

Figure 25: waveforms of CHORDS.SKI, FUNICULA.SKI, FUNSKINI.SKI

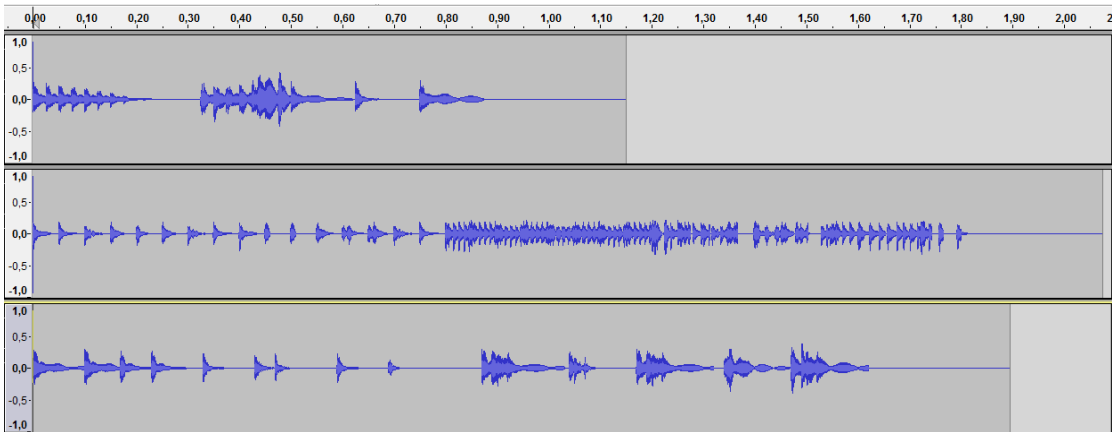

Figure 26: waveforms of PICKDUMP.SKI, SCALES.SKI, TESTSCOR.SKI

Chapter 8 : Conclusion

8.1 Limitations

The STK has been developed in the same department of Stanford University as MoMu and was already released in 1995 but is still developed until now. It offers “an array of unit generators for filtering, input/output, etc., as well as examples of new and classic synthesis and effects algorithms for research, teaching, performance, and composition purposes”. The STK has been slightly modified by the MoMu team. In terms of documentation and community support, the STK library makes a good impression: A comprehensive tutorial covers the basic features. Several demo projects that are included in the source download provide well documented examples. An API documentation offers an overview about the available classes and their methods. Al-though the library is already more than 15 years old, the mailing list is still active and a comprehensive archive exists.

All in all, this library seems to be very promising. It provides a very comprehensive set of features, is well documented and with 15 years of age a very matured library with efficient algorithms. Still, it cannot work wonders with the limited resources of mobile devices and therefore virtual instruments that are developed using the STK, should of course use as little resources as possible.

8.2 Future Work

Motivated by the newly blossoming field of mobile music, the Mobile Music (MoMu) toolkit offers a collection of application programming interfaces (API) and utilities focusing on mobile music development and design. The initial MoMu release focuses on usability and rapid prototyping for the iPhone OS with a particular emphasis toward unifying audio input/output, synthesis, and graphics with the on board sensors now available on commodity mobile phones including accelerometer, compass, location, and multi-touch. More specifically, the

fundamental design goals of MoMu include:

-Real-time audio, synthesis, and control

-Consistent conventions for external sensor access

-Unified common functionality for mobile music

-Focus on ease of use, setup, and installation

-Open source C, C++, and Objective-C code

In this thesis we used MoMu-STK to develop a mandolin music synthesis and to produce music compositions (audio files) based on the mandolin instrument. Following this music synthesis, composers can generate music compositions based on different musical instruments or a combination of musical instruments that belong to the same or different instrument families.
Moreover, since MoMu offers a collection of useful utilities used for audio synthesis and processing, graphics, threading, and general purpose filtering, we can provide useful functionality for mobile phone music application development.

The design focus enables programmers with little or no prior mobile development experience to rapidly develop interactive audio applications, while concentrating on musical and aesthetic considerations. MoMu builds upon the iPhone OS SDK as well as several open source software packages including the Synthesis ToolKit (STK) for sound synthesis and processing, OSCpack for networking via Open Sounds Control, and a Fast Fourier Transform (FFT) implementation adapted from the CARL software distribution. To maximize performance on current mobile hardware, MoMu has been implemented largely in a low-level language (C/C++). The open-source nature allows for custom modifications or additions for production level applications. As far as our experience has shown, such an approach tends to be more familiar to computer musicians and audio developers alike, easier to learn, and lends itself to greater code reuse for future platforms. It can also encourage academic researchers and commercial developers to focus on more musical and interactive applications.

Bibliography

[1] “Resource for all beginning mandolin students.”, http://www.mandolinmania.com

[2] “The mandolin page.”, http://www.banjolin.co.uk/mandolin/

[3] “The Science of Tuning Musical Instruments”, http://www.get-tuned.com/tuning_science.php

[4] “a guide on the mandolin”, http://www.get-tuned.com/mandolin-guide.php#features

[5] “Chords for traditional music”, http://www.banjolin.co.uk/chordtheory/chords.htm

[6] “Introduction to music theory”, http://www.mandolessons.com/lessons/musictheory.html

[7] “Solfege”, http://en.wikipedia.org/wiki/Solfege

[8] “How music works”, http://www.howmusicworks.org

[9] “Strum”, http://en.wikipedia.org/wiki/Strum

[10] “The Synthesis ToolKit in C++ (STK)”, http://ccrma.stanford.edu/software/stk/index.html

[11] “midi analyzer”, http://midikits.net23.net/midi_analyzer/