

Bachelor Thesis

Sound Programming in the Apple iPhone OS Environment

Credits	15 ECTS
Student	Benjamin Eriksson benjamin.eriksson@codemill.se
Company	CodeMill AB
External Supervisor	Emanuel Dohi emanuel@codemill.se
Internal Supervisor	Fredrik Georgsson fredrikg@cs.umu.se
Examiner	Per Lindström perl@cs.umu.se

Abstract

Over the last years an increasing number of music applications and games have been released. The iPhone platform has good variety in this field. However, most often the audio quality is not a prioritised feature, having bad impact on the perceived product quality.

This bachelor thesis describes the different sound programming interfaces available on the iPhone platform and how to use them to generate a high fidelity realtime audio stream. This is achieved using techniques for variations in sounds and multi channel mixing while keeping a low latency and high sound quality.

This thesis shows that real time audio synthesizing is possible on the iPhone platform through the library framework provided by Apple.

After reading this work, one will have a broader knowledge about the available sound API on the Apple iPhone and sound usage in general.

Acknowledgements

I would like to acknowledge the support of my supervisors Fredrik Georgsson at the department of Computing Science, and Emanuel Dohi at CodeMill AB for their support and valuable feedback.

I would also like to thank Terese von Ahn for always supporting me and being there for me when needed.

Contents

1	Introduction	1
1.1	Background	1
1.2	Chapter overview	2
2	Problem Description	3
3	Sound Programming on iPhone	5
3.1	iPhone Development Setup	5
3.1.1	iPhone Developer Program Portal	5
3.1.2	Audacity	6
3.1.3	App Store Limitations	6
3.2	Apple iPhone OS Limitations	6
3.2.1	Sandbox	6
3.2.1.1	Filesystem	6
3.2.1.2	Hardware Access	7
3.2.1.3	Dynamic Execution	7
3.2.2	Resources	7
3.3	Audio APIs	7
3.3.1	OpenAL	8
3.3.2	Audio Queue	8
3.4	Audio Mixing	8
3.4.1	Gain	8
3.4.2	Signal Limiting	9
3.4.2.1	Hard Clipping	9
3.4.2.2	Signal Compression	10
3.4.2.3	Soft limiting	11
3.5	Sound Variations	12
3.5.1	The Wavetable Approach	14
3.5.2	Samples Based Variations	15
3.5.2.1	Audio Data Compression	15
3.5.2.2	Asynchronous sound loading	15
4	Application	17
4.1	User Input	17
4.2	Audio mixing	18
4.3	Sound Variations	18

5 Tests	21
5.1 Latency	21
5.2 More than four voices playing at the same time	22
5.3 Bitrate and precision	22
5.4 Limiting algorithm	22
5.5 Variations	22
6 Analysis	23
7 Conclusion	25
7.1 Variation improvement	25
7.2 Sound Mixing	25
References	27
A Fourier Transformation	29

List of Figures

3.1	Sine wave within the limits	9
3.2	Overdriven sine wave resulting in a two's complement overflow	10
3.3	FT of an overdriven sine wave, compared to a non overdriven	10
3.4	Overview of how different filters effects the timbre of a sine wave	11
3.5	An overdriven sine wave with a hardclipping function applied	12
3.6	FT of a hard clipped sine wave, compared to a non-clipped	12
3.7	Sigmoid function in this case a $\frac{2}{1+e^{-2x}} - 1$	13
3.8	Output signal from a sine function through a sigmoid filter	13
3.9	FT of a sine wave through a sigmoid filter compared to a non-altered	14
4.1	The different API layers used	17
4.2	Flow graph of the system design	18
4.3	Flow cart of how the sound variation library works	19

Chapter 1

Introduction

As technology develop further and further devices become both smaller and more powerful[18]. With small portable devices such as Apple's iPhone the only interface is the touch screen, therefore we have in this project investigated the use of alternative input using the microphone. This has been done by developing a real time instrument simulator. To make the instrument sound more natural, some techniques such as varying sounds has been investigated.

This thesis is one of two in the project, my (Benjamin Eriksson) part describes how to handle high quality audio in and output in the Apple iPhone environment with it's limited resources and how it is implemented in the iPhone project. Tomas Härdin's part is about classifying sounds on a real time audio stream. The project has been developed at a software developing company called CodeMill AB.

1.1 Background

In late december 2008 CodeMill AB wanted to participate in a project with another local company specialized in audio software. The partner wanted to expand to the iPhone market and therefore choose to work with CodeMill.

An idea of a real-time drum application which would get the input from the device' microphone was invented. The idea was inspired by two papers on scratch input[5] and acoustic emanations of computer keyboards[25].

One of the criteria for the application was to have high fidelity audio with a latency lower than 25 ms¹. This was necessary as the partnership company's reputation lies in giving the end user the most high quality sound available.

¹Upper limit suggested by professional drummers

1.2 Chapter overview

Chapter 1 Introduction:

This chapter, an introduction to this thesis.

Chapter 2 Problem Description:

An description of the underlying problem resulting in the idea of writing this writing.

Chapter 3 Sound Programming on iPhone:

An walkthrough of different APIs and sound technics available and suitable for the target platform.

Chapter 4 Application:

This chapter is an description of how the system is implemented and which algorithms that are used.

Chapter 5 Future Features:

Here is a discussion of what can be made to improve the application.

Chapter 6 Tests:

Describes how the system has been tested.

Chapter 7 Analysis:

An overview of how well the work stands up to the expectations from the problem description.

Chapter 2

Problem Description

The objective of this thesis is to study how sound quality in a real time music instrument application can be achieved in the Apple iPhone environment. The application should also be easy to port to other similar platforms and therefore not locked to some specific libraries or computer languages.

If the following is achieved, the product can be called a success:

- sound output without buffer underrun¹
- sound samples at 44100 Hz with 16 bit precision
- more than four voices playing at the same time
- limiting algorithm that reduces distortion during otherwise overflow intervals
- sample variation for different instruments
- latency² under 23 ms

The goal is to create a user intractable high fidelity real time instrument application. The application should be fun to use for both the general public but also for music enthusiasts.

The purpose is to see how high quality real time applications can be implemented in an embedded environment as the Apple iPhone and still kept on a such high level that the project can be portable to similar platforms if demand.

¹ When a buffer is drained at a faster rate than it has been filled

²The time it takes for signal to be recorded, processed and to be presented

Chapter 3

Sound Programming in the Apple iPhone OS Environment

To be able to develop applications to Apple's iPhone environment one must use Apple's own development suite. This since the iPhone environment is a closed platform.

The development environment is free for everyone to download and install. Applications can be installed and debugged on the iPhone Simulator in the development suite. But in order to debug on the target device, such as the iPhone and iPod touch, one must apply for the iPhone Developer Program[10].

The iPhone Simulator is however not an emulator, and does only simulate high level APIs on the host native architecture and can therefore not be used for low level API's and performance tests.

The Apple App Store is where the end user buys iPhone applications. The App Store can either be accessed via the iTunes Store in iTunes on a PC or Mac, or directly via an application on the device. The price for an application is set by the developer. Apple takes a 30% cut of the revenue for providing the distribution channel.

3.1 iPhone Development Setup

The iPhone development setup includes the Xcode development environment, which includes an integrated text editor as well as a graphical debugger[10]. Interface Builder is a design tool for creating graphical user interface for the CocoaToch framework. Instruments is used for measuring processor and memory in real-time[10].

Xcode is build on top of open source tools such as GNU Compiler Collection and Subversion[12] and can also be used via a terminal as a standard UNIX command line interface.

3.1.1 iPhone Developer Program Portal

The iPhone developer program portal¹ is a web portal for managing team members, certificates and provisioning profiles used for signing applications. All applications must be signed to be able to executed on a real device[11].

¹<http://developer.apple.com/iphone/manage/overview/index.action>

3.1.2 Audacity

Audacity is a free software for recording and editing audio, it is available on the Mac OS X operation system and can export to iPhone friendly formats such as AIFF, WAV and MP3[23, 7].

3.1.3 App Store Limitations

When an application is ready for distribution it gets compressed and uploaded to Apple's review service. When the application has been accepted to be distributed on Apple's App Store it will be re-compressed by Apple with the PKZIP format with the suffix .ipa². If this .ipa file is more than 10 MB it will not be distributed through 3G/EDGE on iPhone and must therefore be downloaded via iTunes. This should be taken into consideration to avoid limiting the application to a smaller market. It is possible to decrease the application size with compression of the data files. For instance images can be decreased in size with tools such as pngquant³ and other data files can be compressed with zlib which is included in the standard SDK. More on audio compression can be found in section 3.5.2.1 on page 15.

3.2 Apple iPhone OS Limitations

Since Apple chose to make the iPhone a closed platform a number of limitations has been introduced. Even though some of these limitations can be worked around it should be avoided, due to the iPhone SDK Agreement, if the application is intended to be distributed through Apple's App Store[13].

3.2.1 Sandbox

When a user starts an application the system creates a so called sandbox around the process. This sandbox both protects the user integrity with limiting which files a third party application can reach and protects the iPhone operation system from crashing due to malfunctioning code. All applications has its own virtual memory address space and can therefore not accidentally bring another process down[16].

3.2.1.1 Filesystem

Files bundled with the application cannot be modified due to all applications need to be signed. To be able to modify files, they need to be copied to one of the following folders:

- <Application Home>/Documents
- <Application Home>/Library/Preferences
- <Application Home>/Library/Caches
- <Application Home>/tmp

Only the first two folders are automatically saved during backup[9].

².ipa = iPhoneApplication

³<http://www.libpng.org/pub/png/apps/pngquant.html>

3.2.1.2 Hardware Access

All hardware access has to be done through Apple’s APIs, this to avoid the program from going outside the sandbox. This command tunneling through different API layers introduces an overhead which adds to the latency.

Another problem introduced with the usage of a sandbox is that it is impossible to use all features the hardware offers because of limitations in the generalization of the API that Apple is offering. For example, it is not possible to set the size of an audio buffer, it is only possible to request a preferred audio delay[8].

The iPhone (2g) uses a `Wolfson WM8758` chip for audio processing[22] which is equivalent to the `Wolfson WM8978`[2], this audio processor includes support for *32bits* signals at a sample rate of *48kHz*[21] which is not configurable via the Apple provided API. Only up to *41kHz* is.

3.2.1.3 Dynamic Execution

Apple does not allow dynamic linking, plugin usage or interpreted code[14]. This may also restrict iPhone applications to not be able to use LGPL as static linking may be considered as the application is a derived work of the static linked LGPL part.

3.2.2 Resources

Apple does not have a specified limit for how much memory an application can use, what is said in the documentation is that a notification will be sent whenever memory runs low. If such a notification is received it is up to the application to clean up caches. It is not only the active application that will receive this notification, background processes will receive it too. User tests has shown that some available applications uses up to *60MB* however other have found that they get a “low memory” notification at *40MB*[1].

As for OpenGL ES⁴, there is a hardware *24MB* limit in how much the GART⁵ can address at a frame. Performance hits will occur if this limitation is ignored[1].

3.3 Audio APIs

The music integration is an important part of the iPhone product since it shares the operation system and much of the architecture design and applications with the iPod Touch music player. This audio specialty does not show in the designed API:s and documentation for those. Developers are more or less left to figure out the API:s for themselves or ask other developers at the iPhone development forum⁶.

As for the audio data format; even though some of the internal digital analog converter supports a wide range of audio formats natively[21] only linear PCM at *16bits* is considered canonical on iPhone[7].

⁴Used for 3D graphics on embedded systems

⁵Graphics Address Remapping Table

⁶<https://devforums.apple.com/community/iphone>

3.3.1 OpenAL

The iPhone SDK includes the OpenAL⁷ framework as default. OpenAL a cross-platform 3D sound positional API[17] original designed for 3D games by Loki Entertainment Software for desktop computers[4]. OpenAL on iPhone has a fairly low latency, since it deploys internally an IO Remote Audio Unit. OpenAL supports multiple sounds simultaneous and level control on each source. However, this adds a latency.

The iPhone implementations does not feature the effect extension, thus does not support roger beep, distortion, reverb, obstruction, and occlusion. Audio capture is also not available in the current iPhone OS implementation of OpenAL[15].

3.3.2 Audio Queue

The Audio Queue API is designed to provide the highest control of the audio content, including synchronization[17].Audio Queue does handle mixing of multiple channels and it can be requested to use the end nodes which makes it possible to minimize latency using the Audio Session API.

Audio Queue is built like a graph where each node is either a source/destination or effect. Because of this setup all nodes are independently rendered. An overhead is added for each node due to memory access times in tight loops. This makes one big node which does many operation better to use than many small nodes with on operation each.

To minimize latency all mixing and effects should be done in the audio render callback. The motivation for this is to minimize loops and to avoid reading and writing to buffers between nodes. More about Software mixing can be read in section 3.4.

3.4 Audio Mixing

An important step in how to achieve minimum audio latency with multiple samples is to use Remote IO Audio Queue with an inline software mixer in the audio render callback.

Due to the physical properties of audio waves, two or more sounds can be mixed into a new audio signal by only summing the signals.

Gain needs to be added to the implementation if the amplitude for the in and out signals need to be configurable. More about gain can be found in section 3.4.1.

Some signal distortions called overflow artifacts can be introduced in this mixing. How to handle these distortions will be discussed in section 3.4.2 on the next page.

3.4.1 Gain

The gain value is multiplied to an audio channel to adjust the amplitude. Gain is measured in how much to reduce the signal, $0dB$ is 100% of the original volume and $-5dB$ is 56.234%. The output signal can be calculated with following equation:

$$out = in \cdot 10^{\frac{gain}{20}} \quad (3.1)$$

⁷Open Audio Library

If possible, positive gain should be avoided due to the discrete property of the amplitude in digital audio. When a digital signal is amplified so will the error.

3.4.2 Signal Limiting

There is always a risk of getting overflow problems when two or more digital values are summed, or when multiplying two values. This is also a problem with digital audio signals since they are vectors quantified of samples. Compare a normalized sine wave (Figure 3.1) to an overdriven sound wave (Figure 3.2 on the next page). When a signal is overdriven several overtones are added to the timber. This overtones are shown in figure 3.3 on the following page, notice how the wanted frequency is reduced to less than a fifth of the original amplitude and that the first overtone has become three times the amplitude of the reduced frequency.

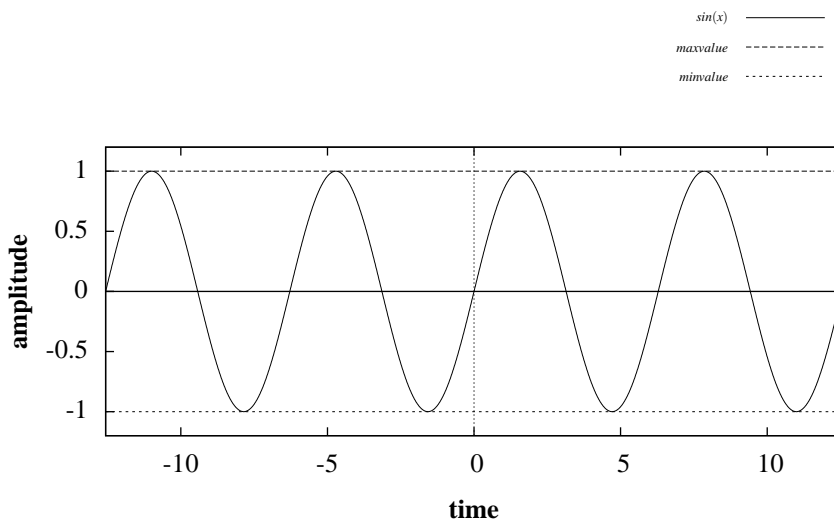


Figure 3.1: Sine wave within the limits

One way to detect overflow errors is to add the samples to a larger data type and when all samples in the different voices are summed, compare the resulted value to the maximum value for the source data type. When problem areas are known, different filters can be applied to minimize distortions. An overview of the result for the filters described in this section can be found in figure 3.4 on page 11.

3.4.2.1 Hard Clipping

Hard clipping is one of the easiest algorithm to implement in a software mixer, one way to perform hard clipping is as shown in algorithm 3.4.1 on page 11. This algorithm will truncate the signal so no overflow occurs as in figure 3.5 on page 12. Only a conditional store operation is needed per sample.

A problem with the hard clipping algorithm is that high frequency signals are introduced in the output audio as shown in figure 3.6 on page 12.

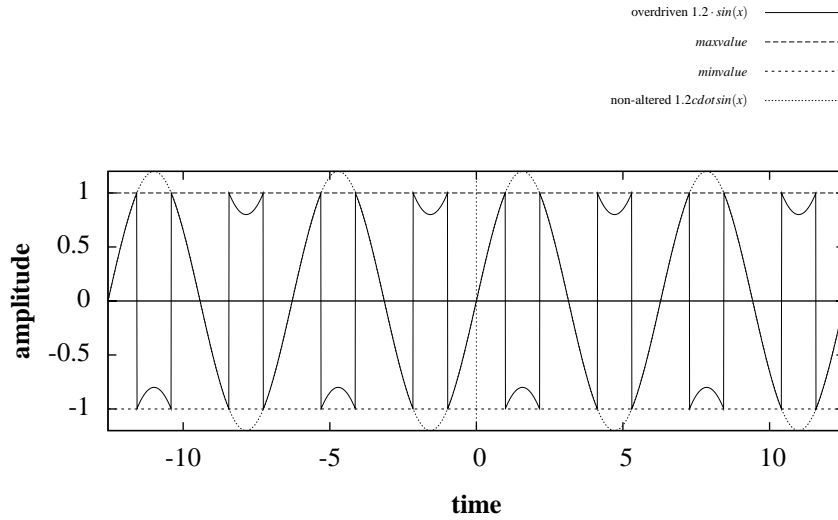


Figure 3.2: Overdriven sine wave resulting in a two's complement overflow

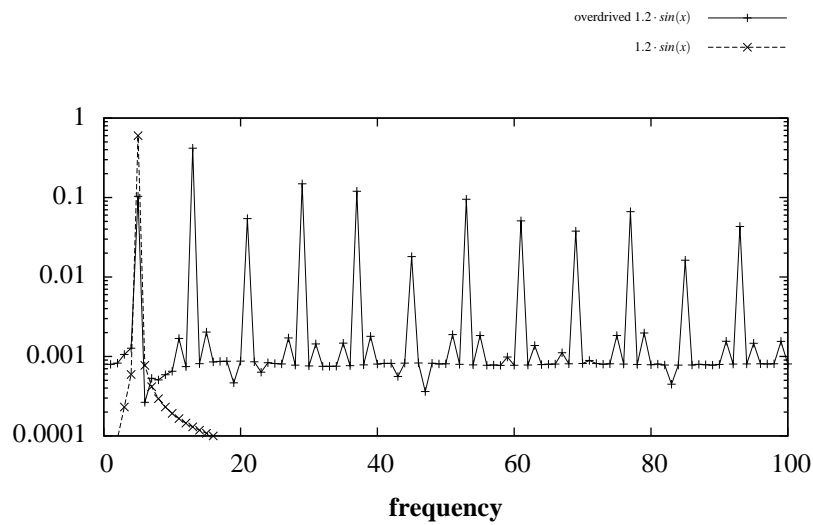


Figure 3.3: Fourier Transformation of an overdriven sine wave, compared to a non overdriven, more about FTs see appendix A on page 29

3.4.2.2 Signal Compression

Signal compression can be achieved by letting the output signal be a nonlinear function of the input signal, see figure 3.7 on page 13. The pseudo code for this type of overflow prevention is shown in the algorithm 3.4.2 on page 13. The output signal for this function is shown in figure 3.8 on page 13.

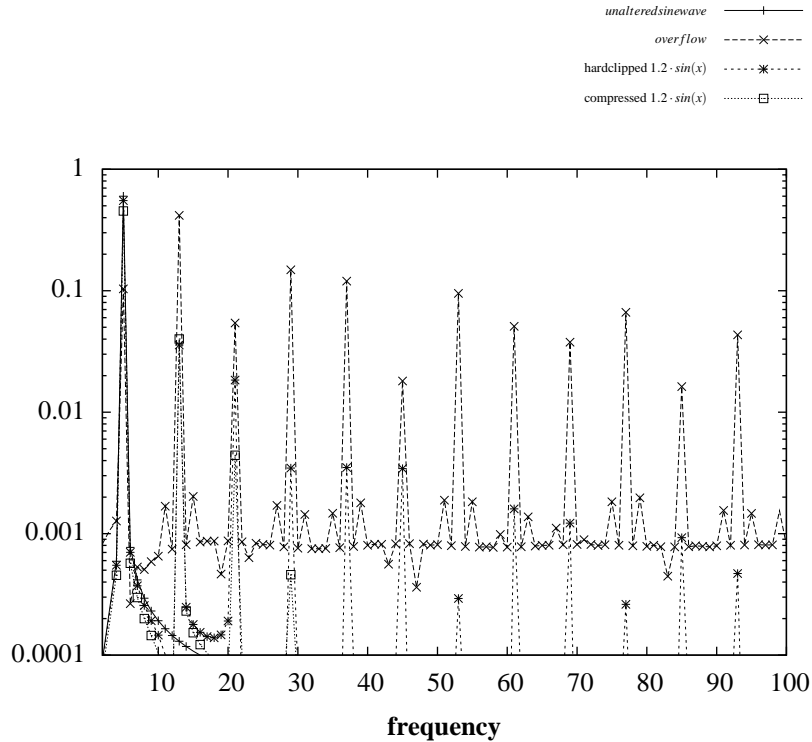


Figure 3.4: Overview of how different filters effects the timbre of a sine wave

Algorithm 3.4.1: HARD CLIPPING(*samples, index*)

```

extern smallDataTypeMax
local bigDataTypeVariable ← 0, output
for each voice ∈ voices
  do bigDataTypeVariable ← bigDataTypeVariable + voice.frame[index].sample
if bigDataTypeVariable ≥ smallDataTypeMax or bigDataTypeVariable ≤ smallDataTypeMax
  then output ← smallDataTypeMax
  else output ← bigDataTypeVariable
return output

```

3.4.2.3 Soft limiting

As shown earlier in this section there will always be some kind of distortion when a signal needs to be limited. Signal compression with an sigmoid function yields a near approximation to the original signal, compared to other discussed methods. A problem though is that the signal will always be compressed even when it is not overdriven. This creates additional distortions compared to hard clipping. To minimize the distortion, compression can be added to signals in problem areas only. But this would however add some latency depending on how it is implemented since the method needs data from all channels before any computation of problem areas can be found. To reduce the mentioned problems above the compression process can be done as described in

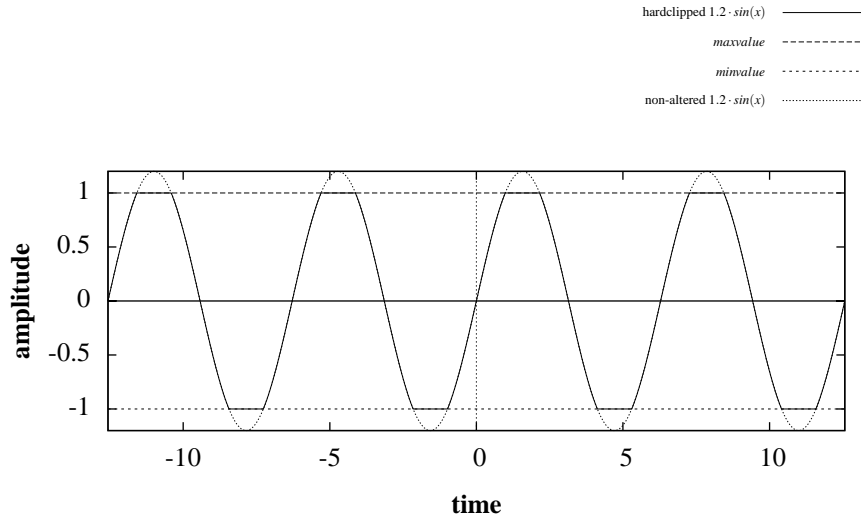


Figure 3.5: An overdriven sine wave with a hard clipping function applied to avoid overflow distortions.

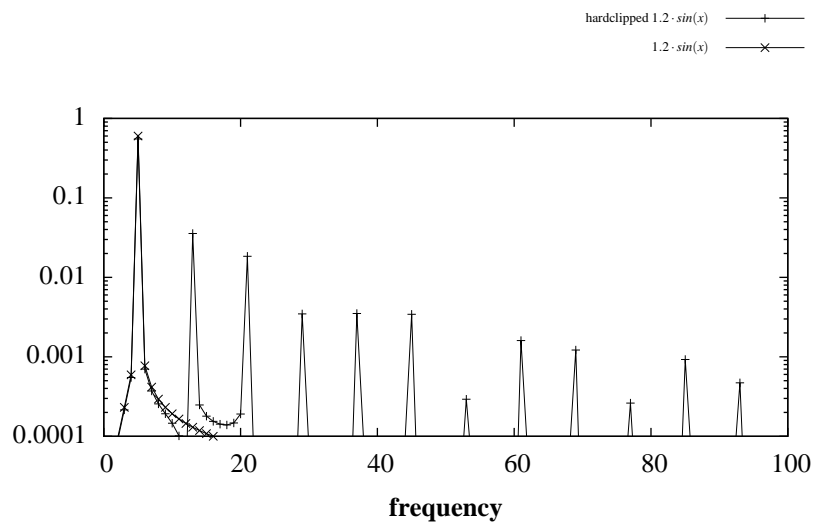


Figure 3.6: Fourier transformation of a hard clipped sine wave, compared to non-clipped.

algorithm 3.4.3 on page 14.

3.5 Sound Variations

Sound variations between each time a sound effect is played is preferable to make the sounds sound more natural, since it is often very hard to reproduce the exact sounds twice in real life. Sound variation can be in amplitude, pitch and timbre.

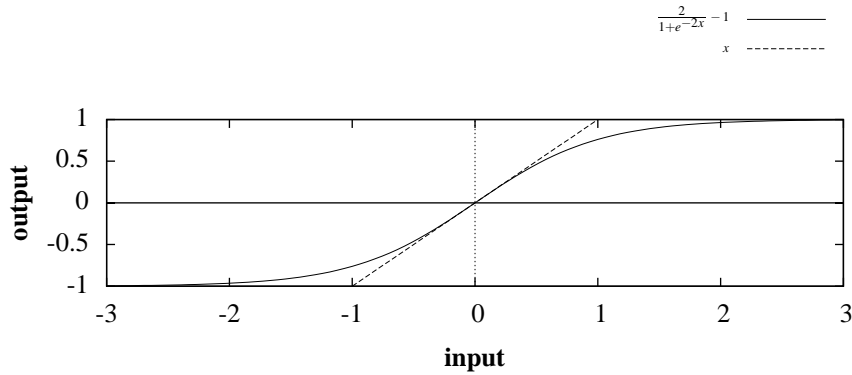


Figure 3.7: Sigmoid function in this case a $\frac{2}{1+e^{-2x}} - 1$.

Algorithm 3.4.2: SIGNAL COMPRESSION(*samples*, *index*)

```

extern smallDataTypeMax
local bigDataTypeVariable  $\leftarrow$  0, output
for each voice  $\in$  voices
  do bigDataTypeVariable  $\leftarrow$  bigDataTypeVariable + voice.frame[index].sample
return smallDataTypeMax  $\cdot$   $\left(\frac{2}{1+e^{-2bigDataTypeVariable}} - 1\right)$ 

```

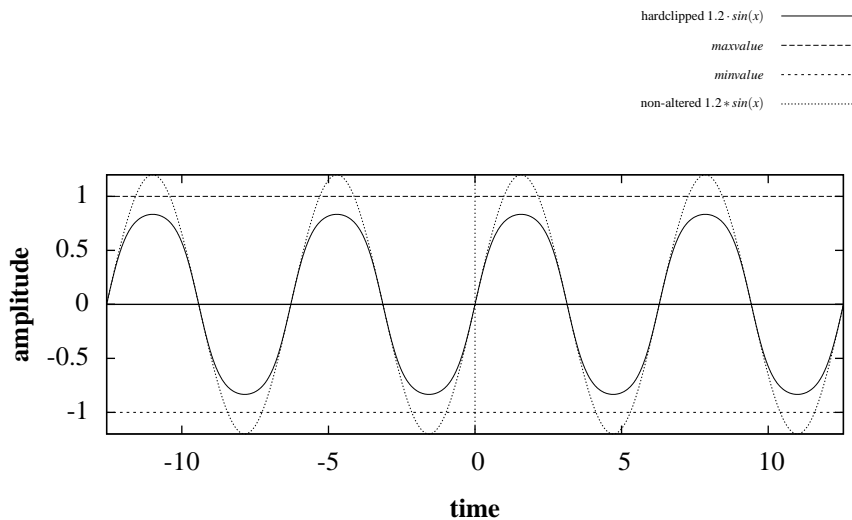


Figure 3.8: Output signal from a sine function through a sigmoid filter

These variations can be achieved by using different sound sources or by applying filters. This works for both sample based sounds and for wavetables.

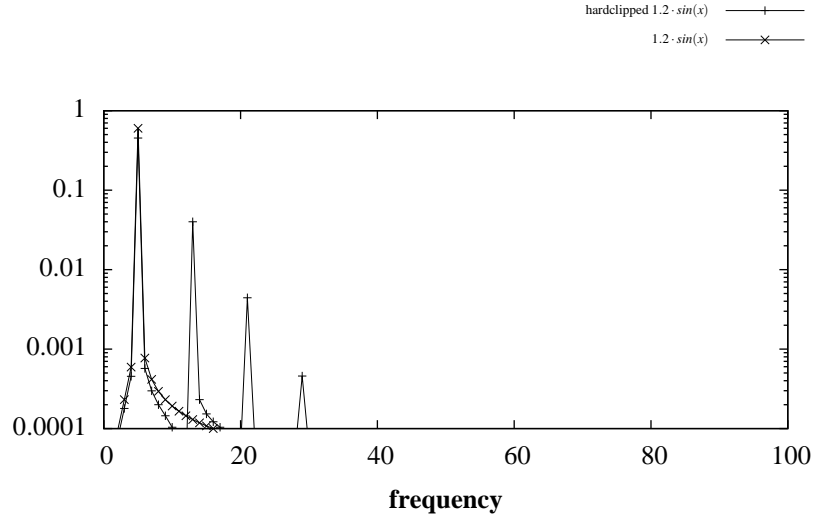


Figure 3.9: FT of a sine wave through a sigmoid filter compared to a non-altered

Algorithm 3.4.3: SMARTCOMPRESSION(*sounds*, *bufferSize*, *lookAhead*)

```

extern roof, lastCompression
local problemPeaks, i, mixed ← MIXTOBIGTYPE(sounds, bufferSize)
problemPeaks ← ABOVE(GETPEAKS(|mixed|), roof)
roofCrossings ← CROSSINGS(|mixed|, roof)
for each i ∈ [0 : bufferSize - 1]
  do { if INSIDEPROBLEMAREA(roofCrossings, problemPeaks, i)
    then COMPRESSSOUND(GETCOMPRESSIONRATE(roofCrossings, problemPeaks, i), mixed)
  }
return TOSMALLTYPE(mixed)
    
```

3.5.1 The Wavetable Approach

The wavetable is a technique that has been around since the early 80's designed by Wolfgang Palm [24].

A wavetable synthesizer uses a lookup table to store several mathematical functions. These functions are then mixed together to create the requested instrument sound. Another key feature for wavetables is to not only change the amplitude over time, but also change the timber dynamically [3].

Because of the nearly periodic nature of instrument tones, the tones can be approximated in a much smaller data set than a real PCM⁸ file [3].

This sound synthesization can also be used to generate different timbre, intensity and pitch depending on how the user strikes a key, if multiple sound data sets are stored[20]. But can also randomly variety to get a more natural sound on binary user input.

⁸Pulse-code modulation

3.5.2 Samples Based Variations

Sample based sound variations should be considered if high fidelity sound is needed. This approach uses two or more sounds to alternate between. This approach do however require more memory than the wave table approach for the same number of variations. To encounter the large memory footprint that is needed for natural variations in the sound, different methods can be used, for instance; loading sound data asynchronous and by using data compression algorithms.

3.5.2.1 Audio Data Compression

There are two kinds of compression available for audio, lossless compression and lossy compression. They both have advantages and disadvantages depending on implementation context.

lossless encoding: Lossless encoding has high sound quality, when no signal data is lost in the compression and decompression stage. However lossless files are generally larger in size compared to lossy formats[19]. Lossless compression tends to need more CPU time at higher compression rates.

lossy encoding: Lossy encoding such as MP3 uses psychoacoustic models to remove low priority sound data. Lossy encoding is often variable in sound quality, audio files can be compressed to smaller files with the loss of sound quality[19].

3.5.2.2 Asynchronous loading of sound variations

As the internal RAM is limited on embedded system such as the iPhone not all sounds can be kept in there. To be able to have a larger sound data set than what fits in the RAM, sound data can be exchanged asynchronously during runtime. This can be done by alternating a few sound files for each variety sound and to load new sounds from disk in the background. Once a new sound is loaded an old can be unloaded. This way there is only need for $n + 1$ sounds in RAM per sound source, where n is the number of sounds to randomly choose from.

To avoid RAM fragmentations, sounds should be around the same size in memory and be statically allocated with a size determined as in the algorithm 3.5.1. If the memory is allocated that way there will not be no need for dynamical allocating and deallocating memory in which causes fragmentation problems over time.

Algorithm 3.5.1: DETERMINBLOCKSIZE(*sounds*, *minFilesInMemory*)

```

local largestSound ← GETLARGEST(sounds)
return largestSound.size · (MIN(minFilesInMemory, sounds.length) + 1)

```

Chapter 4

Application

The application created for this thesis is a drum simulator application made for people interested in music. The user is playing drum beats using acoustic input via the built in microphone and the realtime generated sound is played in the device' speaker or in connected earphones.

The application uses the API Cocoa Touch for receiving data from the touchscreen and Audio Unit Remote IO for audio in and output. The layer structure can be found in figure 4.1.

Graphical User Interface Interactions	Acoustic Input Recognition Procedure	Sound Signal Generation Output Procedure
Cocoa Touch	Audio Unit Remote IO	
Apple iPhone Hardware		

Figure 4.1: The different API layers used

An overview flow graph of the system design can be found in figure 4.2 on the next page.

4.1 User Input

A library has been created to determine which instrument should be played when a certain sound is recorded by the iPhone. The library does only support sounds that have a short audio envelope and fast decay, this while only audio peaks are processed due to timber changes over time in sounds[6].

When a sound has been classified it is inserted as a note to the internal music sheet. The music sheet is a node data set abstraction and currently only contains a fixed number of notes, but this can be changed in a later version of the library implementation to include other functions without changing the API.

A non acoustic interface is also implemented to make the application usable in a noisy environment. Due to restrictions in the provided iPhone API the interface introduces an extra latency.

is a relative bad random algorithm but only if the data set is accessed faster than new sounds are loaded. This procedure is shown in figure 4.3.

Timber changes on volume is not implemented at all. A much larger in memory data set would be needed to be able to use the same algorithm. Solutions for this is further discussed in section 7.1.

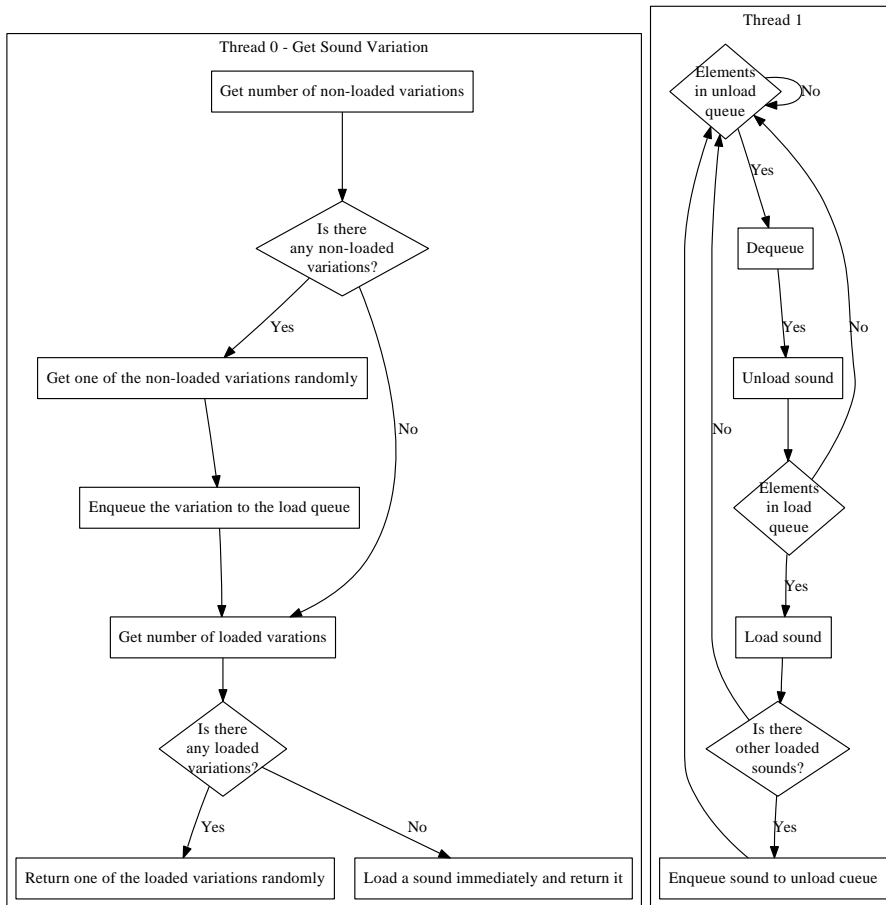


Figure 4.3: Flow cart of how the sound variation library works

Chapter 5

Tests

In this chapter we will go through a couple of tests made that determines if the criteria specified in chapter 2 on page 3 is met or not.

5.1 Latency

A test has been made that involves a user to make a sound and measure how long time it takes to generate a new sound.

This is how the test has been performed

1. a computer with Audacity¹ start recording sounds in the environment
2. a user makes a sound that is heard by both the computer's microphone and by the iPhone software
3. the sound is recorded on the computer and processed by the iPhone software
4. when the iPhone software is finished processing the sound it will output the resulting sound on its speaker
5. this sound is also recorded by the computer
6. the recording session is ended
7. the time between the first sound's start point and the second sound's start point is measured.

The measured time is around 12 ms² depending on if the request to configure the audio buffers are accepted by the system or not. It appears to fail more frequently if another program made audio API calls in a short period of time before this request is made.

¹audio software

²most often 2 · 0.005805s

5.2 More than four voices playing at the same time

A buffer underrun will occur if the audio mixer cannot mix all voices together at the same time.

And since we have not been able to get any underrun for 10 simultaneously playing voices it is safe to say that the system can handle more than four.

5.3 Bitrate and precision

A part from that the API provided by Apple can return how the audio device is currently configured, one can also hear and measure if the bitrate and precision is correctly configured.

If a sound is being played at the wrong bitrate, then the sound will either be played faster or slower depending on if the bitrate is lower or higher than the original bitrate. This yields that one can easily distinguish if the correct bitrate is being used simply by either comparing frequency or how long time a sound takes to play.

If the precision is wrong an audio stream will sound distorted independently of the number of channels and if they are interlaced or not.

In the tests we³ have done we have not found any disturbances in the output for both mono and stereo signal. This and the fact that the device is giving the wanted configuration parameters back when asked for the current configuration, points at both the bitrate and the precision is correctly configured.

5.4 Limiting algorithm

The clipping algorithm has been tested by increasing the voices gain.

A clear difference can be heard between not using any limiting algorithm and by using the hard clipping algorithm.

5.5 Variations

Recordings of the numbers between 0 and 9 has been inputted as different variations of the same instruments. The algorithm works if it alternates between the different numbers. Otherwise it does not work.

During the test the algorithm seems to work to some degree, however the underlying memory management.

³at Codemill

Chapter 6

Analysis

As stated in chapter 2 on page 3, a few criteria was set to determine if the project was a success or not.

Latency under 23 ms: As with using Remote IO (discussed in section 3.3.2 on page 8) the audio buffers have been successfully reduced to 12 ms. However the size can not be guaranteed and must sometimes be requested several times.

However since the program is handling when the system does not accept the buffer configurations, the conclusion is that it does meet the criteria of a latency under 23 ms.

Sound output without buffer underrun: Several sound sources of both stereo and mono audio has been successfully mixed in realtime. No tests has result in buffer underrun, including test involving loading of other parallel threads such as the UI thread and disk loading thread.

More than four voices playing at the same time: A limit is set to a maximum of 8 parallel voices at the same time. The oldest voice is replaced with the new one if the limit is reached.

Sound samples at 44100 Hz with 16bit precession: Sounds are being played at 44100 Hz with 16 bit precession without problem.

Limiting algorithm that reduces distortion: Most of the methods mentioned in 3.4.2 on page 9 has been tested. Currently hardclipping is being used and voices are held at a low input volume to reduce distortions. A soft limiting is recommended for best result though.

Sample variation for different instruments: The variation algorithm described in 4.3 on page 18 has been implemented at a early stage but still needs some caring before it is stable enough as a release candidate.

Four out of six items are completely fulfilled, and the others are partly fulfilled. Most items still have room for improvements, but I feel that the result does satisfy the stated criteria.

Chapter 7

Conclusion

7.1 Variation improvement

Systems with more RAM could be expanded with more instruments with variations for each tone. The algorithm used would however not be suitable on a range of keys when it does not support random access without a heavy time penalty. A solution that might be considered is to use filters to make the sounds vary, this would also make it possible to changing the timbre on volume.

7.2 Sound Mixing

The usage of floating point should be investigated if it can be done in realtime. The VFP¹ might be able to handle it. This would both increase audio quality and make it easier to add filter effects.

¹Vector Floating Point coprocessor

References

- [1] Minimum memory (i.e. empty app)? <https://devforums.apple.com/message/33650>. (visited 2009-03-06).
- [2] Products at Just Listen Audio. <http://justlistenaudio.com/products/>. (revision Aug. 16, 08).
- [3] Robert Bristow-Johnson. Wavetable Synthesis 101, A Fundamental Perspective. Technical report.
- [4] Linux Journal David Penn. Loki Brings 3D Sound to 3D Vision. <http://www.linuxjournal.com/article/5394>. (published March 10th, 2000).
- [5] Chris Harrison and Scott E. Hudson. Scratch input: creating large, inexpensive, unpowered and mobile finger input surfaces. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 205–208, New York, NY, USA, 2008. ACM.
- [6] Tomas Härdin. Realtime Percussive Audio Classification Using Reduced Spectral And Temporal Features On Embedded Systems. (On going thesis work at the department of computing science, Umeå University).
- [7] Apple Inc. https://developer.apple.com/iphone/library/documentation/MusicAudio/Conceptual/CoreAudioOverview/CoreAudioEssentials/chapter_3_section_7.html#//apple_ref/doc/uid/TP40003577-CH10-SW7. (visited 2009-03-07).
- [8] Apple Inc. Audio Session Services Reference. <http://developer.apple.com/iphone/library/documentation/AudioToolbox/Reference/AudioSessionServicesReference/Reference/reference.html>. (visited 2009-03-06).
- [9] Apple Inc. File and Data Management. https://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/FilesandNetworking/chapter_8_section_2.html. (visited 2009-03-06).
- [10] Apple Inc. iPhone Developer Connection - iPhone SDK. <http://developer.apple.com/iphone/program/download.html>. (visited 2009-03-05).
- [11] Apple Inc. iPhone Developer Program. http://adcdownload.apple.com/iphone/iphone_developer_program_user_guide/iphone_developer_program_user_guide__standard_program_v2.4.pdf. (version 2.4).

- [12] Apple Inc. iPhone Development Guide: Building and Running Your Application. http://developer.apple.com/iphone/library/documentation/Xcode/Conceptual/iphone_development/100-The_Development_Process/chapter_2_section_7.html#//apple_ref/doc/uid/TP40007959-CH3-SW23. (visited 2009-03-05).
- [13] Apple Inc. iPhone SDK Agreement. http://adcdownload.apple.com/iphone/iphone_sdk_3.0__final/iphonesdk3.0agreement031709.pdf. (visited 2009-08-20).
- [14] Apple Inc. iPhone SDK Agreement. http://adcdownload.apple.com/iphone/iphone_sdk_for_iphone_os_2.2.1__9m2621a__final/iphone_sdk_agt_ea0495.pdf. (visited 2009-03-06).
- [15] Apple Inc. Technical Note TN2199: OpenAL FAQ for iPhone OS. <http://developer.apple.com/iphone/library/technotes/tn2008/tn2199.html>. (visited 2009-03-07).
- [16] Apple Inc. The Application Runtime Environment. http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/ApplicationEnvironment/chapter_3_section_3.html. (visited 2009-03-06).
- [17] Apple Inc. Using Sound in iPhone OS. http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/AudioandVideoTechnologies/chapter_9_section_2.html. (visited 2009-03-10).
- [18] Intel. Excerpts from A Conversation with Gordon Moore: Moore's Law. ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf. (visited 2009-09-09).
- [19] Clarence Song Lars Ahlzen. *The Sound Blaster Live! Book: A Complete Guide to the World's Most Popular Sound Card*. illustrated edition, 2003.
- [20] Nagai, Yohei (Hamamatsu, JP), Okamoto, Shimaji (Hamamatsu, JP). Electronic musical instrument producing tones by variably mixing different waveshapes. <http://www.freepatentsonline.com/4138915.html>, February 1979. Patent number US:4138915.
- [21] Wolfson Microelectronics plc. Stereo CODEC with Speaker Driver. <http://www.wolfsonmicro.com/uploads/documents/en/WM8978.pdf>. (version June 2008, Rev 4.2).
- [22] Semiconductor insights inc. <http://www.techonline.com/product/underthehood/209000013?pgno=2>. (visited 2009-03-06).
- [23] Unknown. Audacity: About Audacity. <http://audacity.sourceforge.net/about/>. (visited 2009-05-06).
- [24] Mark Vail. *Vintage Synthesizers*. Published by Backbeat Books, 2 edition, 2000.
- [25] Li Zhuang, Feng Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 373–382, New York, NY, USA, 2005. ACM.

Appendix A

Fourier Transformation

A Fourier Transformation is a transformation used to transform a time domain function into a frequency domain representation. This can, for instance, be used to split an audio signal to its base tone and overtones. More about FT's can be read at:

http://en.wikipedia.org/wiki/Fourier_transform

and:

http://en.wikipedia.org/wiki/Fast_Fourier_transform