UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FERNANDO TREBIEN

A GPU-based Real-Time Modular Audio Processing System

Undergraduate Thesis presented in partial fulfillment of the requirements for the degree of Bachelor of Computer Science

Prof. Manuel Menezes de Oliveira Neto
Advisor

Porto Alegre, June 2006

CIP – CATALOGING-IN-PUBLICATION

Trebien, Fernando

A GPU-based Real-Time Modular Audio Processing System / Fernando Trebien. – Porto Alegre: CIC da UFRGS, 2006.

71 f.: il.

Undergraduate Thesis – Universidade Federal do Rio Grande do Sul. Curso de Ciência da Computação, Porto Alegre, BR–RS, 2006. Advisor: Manuel Menezes de Oliveira Neto.

1. Computer music. 2. Electronic music. 3. Signal processing. 4. Sound synthesis. 5. Sound effects. 6. GPU. 7. GPGPU. 8. Realtime systems. 9. Modular systems. I. Neto, Manuel Menezes de Oliveira. II. Title.

ACKNOWLEDGMENT

I'm very thankful to my advisor, Prof. Manuel Menezes de Oliveira Neto, for his support, good will, encouragement, comprehension and trust throughout all semesters he has instructed me. I also thank my previous teachers for their dedication in teaching me much more than knowledge—in fact, proper ways of thinking. Among them, I thank specially Prof. Marcelo de Oliveira Johann, for providing me a consistent introduction to computer music and encouragement for innovation.

I acknowledge many of my colleagues for their help on the development of this work. I acknowledge specially:

- Carlos A. Dietrich, for directions on building early GPGPU prototypes of the application;
- Marcos P. B. Slomp, for providing his monograph as a model to build this text; and
- Marcus A. C. Farias, for helping with issues regarding Microsoft COM.

At last, I thank my parents for providing me the necessary infrastructure for study and research on this subject, which is a challenging and considerable step toward a dream in my life, and my friends, who have inspired me to pursue my dreams and helped me through hard times.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ADC	Analog-to-Digital Converter (hardware component)
AGP	Accelerated Graphics Port (bus interface)
ALSA	Advanced Linux Sound Architecture (software interface)
AM	Amplitude Modulation (sound synthesis method)
API	Application Programming Interface (design concept)
ARB	Architecture Review Board (organization)
ASIO	Audio Stream Input Output[1] (software interface)
Cg	C for Graphics[2] (shading language)
COM	Component Object Model[3] (software interface)
CPU	Central Processing Unit (hardware component)
DAC	Digital-to-Analog Converter (hardware component)
DFT	Discrete Fourier Transform (mathematical concept)
DLL	Dynamic-Link Library[3] (design concept)
DSD	Direct Stream Digital[4,5] (digital sound format)
DSP	Digital Signal Processing
DSSI	DSSI Soft Synth Instrument (software interface)
FFT	Fast Fourier Transform (mathematical concept)
FIR	Finite Impulse Response (signal filter type)
FM	Frequency Modulation (sound synthesis method)
FBO	Framebuffer Object (OpenGL extension, software object)
GUI	Graphical User Interface (design concept)
GmbH	Gesellschaft mit beschränkter Haftung (from German, meaning "company with limited liability")
GLSL	OpenGL Shading Language[6]
GSIF	GigaStudio InterFace[7] or GigaSampler InterFace[8] (software interface)

[1] Steinberg Media Technologies GmbH. [2] NVIDIA Corporation. [3] Microsoft Corporation. [4] Sony Corporation. [5] Koninklijke Philips Electronics N.V. [6] OpenGL Architecture Review Board. [7] TASCAM. [8] Formerly from NemeSys.

GPGPU	General Purpose GPU Programming (design concept)
GPU	Graphics Processing Unit (hardware component)
IIR	Infinte Impulse Response (signal filter type)
LFO	Low-Frequency Oscillator (sound synthesis method)
MIDI	Musical Instrument Digital Interface
MME	MultiMedia Extensions[3] (software interface)
MP3	MPEG-1 Audio Layer 3 (digital sound format)
OpenAL	Open Audio Library[9] (software interface)
OpenGL	Open Graphics Library[6] (software interface)
PCI	Peripheral Component Interconnect (bus interface)
RMS	Root Mean Square (mathematical concept)
SDK	Software Development Kit (design concept)
SNR	Signal-to-Noise Ratio (mathematical concept)
SQNR	Signal-Quantization-Error-Noise Ratio (mathematical concept)
THD	Total Harmonic Distortion (mathematical concept)
VST	Virtual Studio Technology[1] (software interface)

[9] Creative Technology Limited.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ABSTRACT

The impressive growth in computational power experienced by GPUs in recent years has attracted the attention of many researchers and the use of GPUs for applications other than graphics ones is becoming increasingly popular. While GPUs have been successfully used for the solution of linear algebra and partial differential equation problems, very little attention has been given to some specific areas such as 1D signal processing.

This work presents a method for processing digital audio signals using the GPU. This approach exploits the parallelism of fragment processors to achieve better performance than previous CPU-based implementations. The method allows real-time generation and transformation of multichannel sound signals in a flexible way, allowing easier and less restricted development, inspired by current virtual modular synthesizers. As such, it should be of interest for both audio professionals and performance enthusiasts. The processing model computed on the GPU is customizable and controllable by the user. The effectiveness of our approach is demonstrated with adapted versions of some classic algorithms such as generation of primitive waveforms and a feedback delay, which are implemented as fragment programs and combined on the fly to perform live music and audio effects. This work also presents a discussion of some design issues, such as signal representation and total system latency, and compares our results with similar optimized CPU versions.

Keywords: Computer music, electronic music, signal processing, sound synthesis, sound effects, GPU, GPGPU, realtime systems, modular systems.

Um Sistema Modular de Processamento de Áudio em Tempo Real Baseado em GPUs

RESUMO

O impressionante crescimento em capacidade computacional de GPUs nos últimos anos tem atraído a atenção de muitos pesquisadores e o uso de GPUs para outras aplicações além das gráficas está se tornando cada vez mais popular. Enquanto as GPUs têm sido usadas com sucesso para a solução de problemas de álgebra e de equações diferenciais parciais, pouca atenção tem sido dada a áreas específicas como o processamento de sinais unidimensionais.

Este trabalho apresenta um método para processar sinais de áudio digital usando a GPU. Esta abordagem explora o paralelismo de processadores de fragmento para alcançar maior desempenho do que implementações anteriores baseadas na CPU. O método permite geração e transformação de sinais de áudio multicanal em tempo real de forma flexível, permitindo o desenvolvimento de extensões de forma simplificada, inspirada nos atuais sintetizadores modulares virtuais. Dessa forma, ele deve ser interessante tanto para profissionais de áudio quanto para músicos. O modelo de processamento computado na GPU é personalizável e controlável pelo usuário. A efetividade dessa abordagem é demonstrada usando versões adaptadas de algoritmos clássicos tais como geração de formas de onda primitivas e um efeito de atraso realimentado, os quais são implementados como programas de fragmento e combinados dinamicamente para produzir música e efeitos de áudio ao vivo. Este trabalho também apresenta uma discussão sobre problemas de projeto, tais como a representação do sinal e a latência total do sistema, e compara os resultados obtidos de alguns processos com versões similares programadas para a CPU.

Palavras-chave: Música computacional, música eletrônica, processamento de sinais, síntese de som, efeitos sonoros, GPU, GPGPU, sistemas de tempo real, sistemas modulares..

1 INTRODUCTION

In recent years, music producers have experienced a transition from hardware to software synthesizers and effects processors. This is mostly because software versions of hardware synthesizers present significant advantages, such as greater time accuracy and greater flexibility for interconnection of independent processor modules. Software components are also generally cheaper than hardware equipment. At last, a computer with multiple software components is much more portable than a set of hardware equipments. An ideal production machine for a professional musician would be small and powerful, such that the machine itself would suffice for all his needs.

However, software synthesizers, as any piece of software, are all limited by the CPU's computational capacity, while hardware solutions can be designed to meet performance goals. Even though current CPUs are able to handle most common sound processing tasks, they lack the power for combining many computation-intensive digital signal processing tasks and producing the net result in real-time. This is often a limiting factor when using a simple setup—*e.g.,* one MIDI controller, such as a keyboard, and one computer—for real-time performances.

There have been speculations about a technological singularity for processor computational capacity unlimited growth (WIKIPEDIA, 2006a; ZHIRNOV et al., 2003). Even if there are other points of view (KURO5HIN, 2006), it seems Moore's law doubling time has been assigned increasing values along history, being initially 1 year (MOORE, 1965) and currently around 3 years. Meanwhile, graphics processors have been offering more power at a much faster rate (doubling the density of transistors every six months, according to nVidia). For most streaming applications, current GPUs outperform CPUs considerably (BUCK et al., 2004).

Limited by the CPU power, the user would naturally look for specialized audio hardware. However, most sound cards are either directed to professionals and work only with specific software or support only a basic set of algorithms in a fixed-function pipeline (GALLO; TSINGOS, 2004). This way, the user cannot freely associate different audio processes as he would otherwise be able to using modular software synthesizers.

So, in order to provide access to the computational power of the GPU, we have designed a prototype of a modular audio system, which can be extended by programming new processor modules. This adds new possibilities to the average and professional musician, by making more complex audio computations realizable in real-time. This work describes the design and implementation details of our system prototype and provides information on how to extend the system.

In the context of this work, Figure 1.1 presents an abstract representation of data flow in a real-time audio application according to a particular arrangement. Sound waves are captured from the environment along time (step 1 on the right) and converted to a string

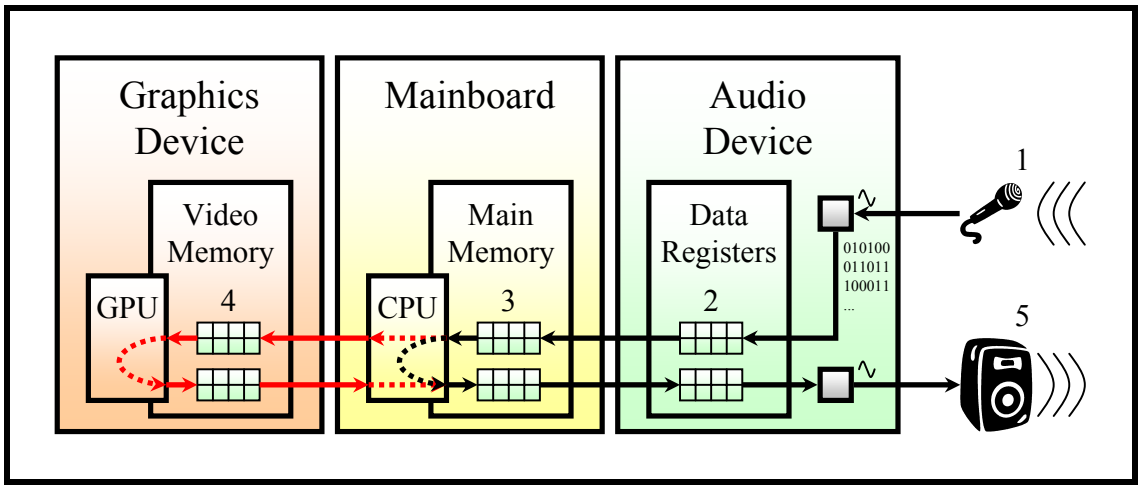


Figure 1.1: Overview of the proposed desktop-based audio processing system.

of numbers, which is stored on the audio device (step 2). Periodically, new audio data is passed to the CPU (step 3) which may process it directly or send it for processing on the GPU (step 4). On the next step, the audio device collects the data from the main memory and converts it to continuous electrical signals, which are ultimately converted into air pressure waves (step 5). Up to the moment, the paths between CPU and GPU remain quite unexplored for audio processing.

In this work, we have not devised a new DSP algorithm, neither have we designed an application for offline audio processing[1]. We present a platform for audio processing on the GPU with the following characteristics:

- Real-time results;
- Ease of extension and development of new modules;
- Advantage in terms of capabilities (*e.g.,* polyphony, effect realism) when compared to current CPU-based systems and possibly even hardware systems specifically designed for music;
- Flexible support for audio formats (*i.e.,* any sampling rate, any number of channels); and
- Ease of integration with other pieces of software (*e.g.,* GUIs).

We analyze some project decisions and their impact on system performance. Low-quality algorithms (*i.e.,* those subject to noise or aliasing) have been avoided. Additionally, several algorithms considered basic for audio processing are implemented. We show that the GPU performs well for certain algorithms, achieving speedups of as much as $50\times$ or more, depending on the GPU used for comparison. The implementation is playable by a musician, which can confirm the real-time properties of the system and the quality of the signal, as we also discuss.

1.1 Text Structure

Chapter 2 discusses other works implementing audio algorithms on graphics hardware. In each case, the differences to this work are carefully addressed.

[1] See the difference between *online* and *offline* processing in Section 3.2.

Chapter 3 presents several concepts of physical sound that are relevant when processing audio. The basic aspects of an audio system running on a computer are presented. Topics in this chapter were chosen carefully because the digital audio processing subject is too vast to be covered in this work. Therefore, only some methods of synthesis and effects are explained in detail.

Next, on Chapter 4, the processes presented on the previous chapter are adapted to the GPU and presented in full detail. We first discuss details of the operation of graphics systems which are necessary to compute audio on them. The relationship between entities in the contexts of graphics and audio processing is established, and part of the source code is presented as well.

Finally, performance comparison tests are presented and discussed on Chapter 5. Similar CPU implementations are used to establish the speedup obtained by using the GPU. The quality of the signal is also discussed, with regard to particular differences between arithmetic computation on CPU and GPU. Finally, the limitations of the system are exposed.

2 RELATED WORK

This chapter presents a description of some related works on using the power of the GPU for audio processing purposes. It constitutes a detailed critical review of current work on the subject, attempting to distinctively characterize our project from others. We review each work with the same concerns with which we evaluate the results of our own work on Chapter 5. The first section will cover 5 works more directly related to the project. Each of them is carefully inspected under the goals established in Chapter 1. At the end, a short revision of the chapter is presented.

2.1 Audio Processing Using the GPU

On the short paper entitled *Efficient 3D Audio Processing with the GPU*, Gallo and Tsingos (2004) presented a feasibility study for audio-rendering acceleration on the GPU. They focused on audio rendering for virtual environments, which requires considering sound propagation through the medium, blocking by occluders, binaurality and Doppler effect. In their study, they processed sound at 44.1 kHz in blocks of 1,024 samples (almost 23.22 ms of audio per block) in 4 channels (each a sub-band of a mono signal) using 32-bit floating-point format. Block slicing suggests that they processed audio in real-time, but the block size is large enough to cause audible delays[1]. They compared the performance of implementations of their algorithm running on a 3.0 GHz Pentium 4 CPU and on an nVidia GeForce FX 5950 on AGP 8x bus. For their application, the GPU implementation was 17% slower than the CPU implementation, but they suggested that, if texture resampling were supported by the hardware, the GPU implementation could have been 50% faster.

Gallo and Tsingos concluded that GPUs are adequate for audio processing and that probably future GPUs would present greater advantage over CPU for audio processing. They highlighted one important problem which we faced regarding IIR filters: they cannot be implemented efficiently due to data dependency[2]. Unfortunately, the authors did not provide enough implementation detail that could allow us to repeat their experiments.

Whalen (2005) discussed the use of GPU for offline audio processing of several DSP algorithms: chorus, compression[3], delay, low and highpass filters, noise gate and volume normalization. Working with a 16-bit mono sample format, he compared the performance of processing an audio block of 105,000 samples on a 3.0 GHz Pentium 4 CPU against an nVidia GeForce FX 5200 through AGP.

[1] See Section 3.2 for more information about latency perception. [2] See Section 4.4.5 for more information on filter implementation using the GPU. [3] In audio processing, *compression* refers to mapping sample amplitude values according to a shape function. Do not confuse with data compression such as in the gzip algorithm or audio compression such as in MP3 encoding.

Even with such a limited setup, Whalen found up to 4 times speedups for a few algorithms such as delay and filtering. He also found reduced performance for other algorithms, and pointed out that this is due to inefficient access to textures. Since the block size was much bigger than the maximum size a texture may have on a single dimension, the block needed to be mapped to a 2D texture, and a slightly complicated index translation scheme was necessary. Another performance factor pointed out by Whalen is that texels were RGBA values and only the red channel was being used. This not only wastes computation but also uses caches more inefficiently due to reduced locality of reference.

Whalen did not implement any synthesis algorithms, such as additive synthesis or frequency modulation. Performance was evaluated for individual algorithms in a single pass, which does not consider the impact of having multiple render passes, or changing the active program frequently. It is not clear from the text, but probably Whalen timed each execution including data transfer times between the CPU and the GPU. As explained in Chapter 4, transfer times should not be counted, because transferring can occur while the GPU processes. Finally, Whalen's study performs only offline processing.

Jędrzejewski and Marasek (2004) used a ray-tracing algorithm to compute an impulse response pattern from one sound source on highly occluded virtual environments. Each wall of a room is assigned an absorption coefficient. Rays are propagated from the point of the sound source up to the 10th reflection. This computation resulted in a speedup of almost 16 times over the CPU version. At the end, ray data was transferred to main memory, leaving to the CPU the task of computing the impulse response and the reverberation effect.

Jędrzejewski and Marasek's work is probably very useful in some contexts—*e.g.,* game programming—, but compared to the goals of this work, it has some relevant drawbacks. First, the authors themselves declared that the CPU version of the tracing process was not highly optimized. Second, there was no reference to the constitution of the machine used for performance comparison. Third, and most importantly, all processing besides raytracing is implemented on the CPU. Finally, calculation of an impulse response is a specific detail of spatialization sound effects' implementation.

BionicFX (2005) is the first and currently only commercial organization to announce GPU-based audio components. BionicFX is developing a DSP engine named RAVEX and a reverb processor named BionicReverb, which should run on the RAVEX engine. Although in the official home page it is claimed that those components will be released as soon as possible, the website has not been updated since at least September, 2005 (when we first reached it). We have tried to contact the company but received no reply. As such, we cannot evaluate any progress BionicFX has achieved up to date.

Following a more distant line, several authors have described implementations of the FFT algorithm on GPUs (ANSARI, 2003; SPITZER, 2003; MORELAND; ANGEL, 2003; SUMANAWEERA; LIU, 2005). 1D FFT is required in some more elaborated audio algorithms. Recently, GPUFFTW (2006), a high performance FFT library using the GPU, was released. Its developers claim that it provides a speedup factor of 4 when compared to single-precision optimized FFT implementations on current high-end CPUs.

The reader shall note that most of the aforementioned works were released at a time when GPUs presented more limited capacity. That may partially justify some of the low performance results.

There has been unpublished scientific development on audio processing, oriented exclusively toward implementation on commercial systems. Since those products are an important part of what constitutes the state of the art, you may refer to Appendix A for an

overview of some important commercial products and the technology they apply.

In contrast to the described techniques, our method solves a different problem: the mapping from a network model of virtually interconnected software modules to the graphics pipeline processing model. Our primary concern is how the data is passed from one module to another using only GPU operations and the management of each module's internal data[4]. This allows much greater flexibility to program new modules and effectively turns GPUs into general music production machines.

2.2 Summary

This chapter discussed the related work on the domain of GPU-based audio processing. None of the mentioned works is explicitly a real-time application, and the ones performing primitive audio algorithms report little advantage of the GPU over the CPU. Most of them constitute test applications to examine GPU's performance for audio processing, and the most recent one dates from more than one year ago. Therefore, the subject needs an up-to-date in-depth study.

The next chapter discusses fundamental concepts of audio and graphics systems. The parts of the graphics processing pipeline that can be customized and rendering settings which must be considered to perform general purpose processing on the GPU are presented. Concepts of sound and the structure of audio systems also are covered. At last, we present the algorithms we implemented in this application.

[4] See Section 3.2.2 for information on architecture of audio applications.

3 AUDIO PROCESSES

This chapter presents basic concepts that will be necessary for understanding the description of our method for processing sound on graphics hardware. The first section presents definitions related with the physical and perceptual concepts of sound. The next sections discusses audio processes from a computational perspective. These sections receive more attention, since an understanding of audio processing is fundamental to understand what we have built on top of the graphics system and why.

3.1 Concepts of Sound, Acoustics and Music

Sound is a mechanical perturbation that propagates on a medium (typically, air) along time. The study of sound and its behavior is called *acoustics*. A physical sound field is characterized by the pressure level on each point of space at each instant in time. Sound generally propagates as waves, causing local regions of compression and rarefaction. Air particles are, then, displaced and oscillate. This way, sound manifests as continuous *waves*. Once reaching the human ear, sound waves induce movement and subsequent nervous stimulation on a very complex biological apparatus called *cochlea*. Stimuli are brought by nerves to brain, which is responsible for the subjective interpretation given to sound (GUMMER, 2002). Moore (1990, p. 18) defines the basic elements of hearing as

$$\text{sound waves} \rightarrow \text{auditory perception} \rightarrow \text{cognition}$$

It is known since ancient human history that objects in our surroundings can produce sound when they interact (normally by collision, but often also by attrition). By transferring any amount of mechanical energy to an object, molecules on its surface move to a different position and, as soon as the source of energy is removed, accumulated elastic tension induces the object into movement. Until the system reaches stability, it oscillates in *harmonic motion*, disturbing air in its surroundings, thereby transforming the accumulated energy into sound waves. Properties of the object such as size and material alter the nature of elastic tension forces, causing the object's oscillation pattern to change, leading to different kinds of sound.

Because of that, humans have experimented with many object shapes and materials to produce sound. Every culture developed a set of instruments with which it produces music according to its standards. Before we begin defining characteristics of sound which humans consider interesting, we need to understand more about the nature of sound waves.

3.1.1 Sound Waves

When working with sound, normally, we are interested on the pressure state at one single point in space along time[1]. This disregards the remaining dimensions, thus, sound at a point is a function only of time. Let $w : \mathbb{R} \to \mathbb{R}$ be a function representing a wave. $w(t)$ represents the amplitude of the wave at time t. Being oscillatory phenomena, waves can be divided in *cycles*, which are time intervals that contain exactly one oscillation. A cycle is often defined over a time range where $w(t)$, as t increases, starts at zero, increases into a positive value, then changes direction, assumes negative values, and finally returns to zero[2]. The *period* of a cycle is the difference in time between the beginning and the end of a single cycle, *i.e.,* if the cycle starts at t_0 and ends at t_1, its period T is simply defined as $T = t_1 - t_0$. The *frequency* of a wave is the number of cycles it presents per time unit. The frequency f of a wave whose period is T is defined as $f = \frac{N}{t} = \frac{1}{T}$, where N represents the number of cycles during time t. The amplitude of a cycle A is defined as the highest deviation from the average level that w achieves during the cycle. The average level is mapped to amplitude zero, so the amplitude can be defined as the maximum value for $|w(t)|$ with $t_0 \leq t \leq t_1$. *Power* is a useful measure of perceived intensity of the wave. Power P and amplitude A are related by $A^2 \propto P$. The *root-mean-square* (RMS) power of wave is an useful measure in determining the wave's perceived intensity. The RMS power P_{RMS} of wave f with $T_0 \leq t \leq T_1$ is defined as

$$P_{\text{RMS}} = \sqrt{\frac{1}{T_1 - T_0} \int_{T_0}^{T_1} w(t)^2 \, dt} \tag{3.1}$$

To compare the relative power of two waves, the *decibel* scale is often used. Given two waves with powers P_0 and P_1 respectively, the ratio $\frac{P_1}{P_0}$ can be expressed in decibels (dB) as

$$P_{dB} = 10 \, log_{10} \left(\frac{P_1}{P_0} \right) \tag{3.2}$$

Remark When the time unit is the *second* (s), frequency (cycles per second) is measured in *Hertz* (Hz). Hertz and seconds are reciprocals, such that $\text{Hz} = \frac{1}{s}$.

A *periodic* wave w is such that a single oscillation pattern repeats infinitely in the image of w throughout its domain, *i.e.,* there is $T \in \mathbb{R}$ such that $w(t) = w(t + T)$ for any $t \in \mathbb{R}$. The minimum value of T for which this equation holds defines precisely to the period of the wave. No real world waves are periodic, but some can achieve a seemingly periodic behavior during limited time intervals. In such cases, they are called *quasi-periodic*. Notice, then, that amplitude, period and frequency of a wave are characteristics that can change along time, except for the theoretical abstraction of periodic waves[3].

As all waves, sound exhibits *reflection* when hitting a surface, *interference* when multiple waves "crossing" the same point in space overlap, and *rectilinear propagation*.

[1] This is a simplification that derives from the fact that microphones, loudspeakers and even the human ear interact with sound at a very limited region in space. The sound phenomenon requires a three-dimensional function to be represented exactly. [2] This definition works for periodic waves, but it is very inaccurate for most wave signals and cannot be applied in practical applications. [3] One should note that, differently from quasi-periodic signals, periodic signals present well-defined characteristics such as period, frequency and amplitude. In this case, these characteristics are constants through all the domain.

Sound also experiences *refraction*, *diffraction* and *dispersion*, but those effects are generally not interesting to audio processing because they mainly affect only sound spatialization, which can be innacurately approximated considering only reflections.

The most fundamental periodic waveform is the sinusoidal wave, defined by

$$w\left(t\right) = A\,sin\left(\omega t + \phi\right) \tag{3.3a}$$

in which A is the amplitude, $\omega = 2\pi f$ where f is the frequency, and ϕ is the phase offset of w. This equation can be rewritten as

$$w\left(t\right) = a\,cos\;\omega t + b\,sin\;\omega t \tag{3.3b}$$

where

$$A = \sqrt{a^2 + b^2} \quad \text{and} \quad \phi = tan^{-1}\,\frac{b}{a}$$

or, equivalently,

$$a = A\,cos\;\phi \quad \text{and} \quad b = A\,sin\;\phi$$

An example of a sinusoid with $A = 1$, $T = 1$ and $\phi = 0.25$ is illustrated on Figure 3.1(a). The solid segment represents one complete cycle of the wave.

Due to interference, waves can come in many different wave shapes. The amplitude of a wave $w_r\left(t\right)$ resulting from interference of other two sound waves $w_1\left(t\right)$ and $w_2\left(t\right)$ at the same point in space at the instant x is simply the sum of them, *i.e.,* $w_r\left(t\right) = w_1\left(t\right) + w_2\left(t\right)$. When both $w_1\left(t\right)$ and $w_2\left(t\right)$ have the same sign, $w_r\left(t\right)$ assumes an absolute value greater than that of its components; therefore, the interference of w_1 and w_1 at t is called *constructive interference*. If $w_1\left(t\right)$ and $w_2\left(t\right)$ have different signals, the resulting absolute amplitude value is lower than that of one of its components, and this interaction is called *destructive interference*.

Interference suggests that more elaborated periodic waveforms can be obtained by summing simple sinusoids. If defined by coefficients $\langle a_k, b_k\rangle$ as in Equation (3.3b), a composite waveform formed by N components can be defined as

$$w\left(t\right) = \sum_{k=0}^{N-1}\left[a_k\,cos\;\omega_k t + b_k\,sin\;\omega_k t\right] \tag{3.4}$$

For example, one can define a composite wave by setting

$$\begin{aligned} \omega_k &= 2\pi f_k & a_k &= 0 \\ f_k &= 2k+1 & b_k &= \frac{1}{2k+1} \end{aligned}$$

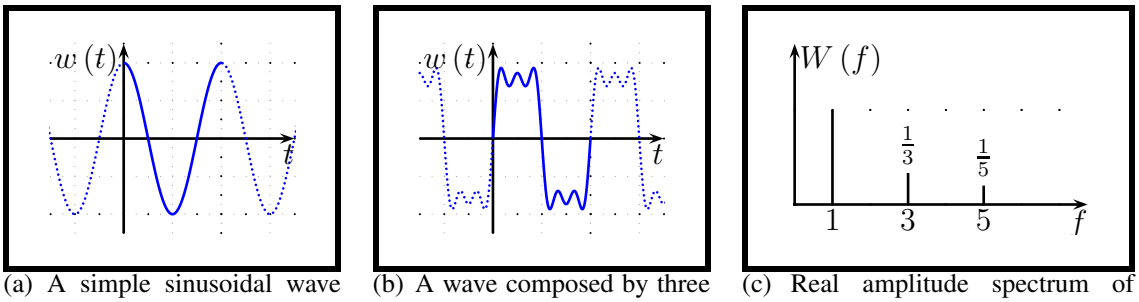
yielding

$$w\left(t\right) = sin\;2\pi t + \frac{1}{3}\,sin\;6\pi t + \frac{1}{5}\,sin\;10\pi t + \ldots + \frac{1}{2N-1}\,sin\left(2N-1\right)\pi t$$

By taking N = 3, we obtain the waveform depicted on Figure 3.1(b). Again, the solid trace represents one complete cycle of the wave.

Equation (3.4) is called the *Fourier series* of wave w. A more general instance of this arrangement, defined for the complex domain, is the Inverse Discrete Fourier Transform (IDFT), which is given by

$$w_s\left(n\right) = \sum_{k=0}^{N-1} W_s\left(k\right) e^{\imath\omega_k n} \tag{3.5a}$$



(a) A simple sinusoidal wave with $\phi = 0.25$.

(b) A wave composed by three sinusoids with $\phi = 0$.

(c) Real amplitude spectrum of waveform on item (b).

Figure 3.1: Examples of waveform composition.

where $w_s, W_s : \mathbb{Z} \rightarrow \mathbb{C}$. To finally obtain values for $W(k)$ from any periodic wave w, we can apply the Discrete Fourier Transform (DFT), given by

$$W_s(k) = \sum_{n=0}^{N-1} w_s(n)\, e^{-\imath \omega_k n} \tag{3.5b}$$

W_s is called the *spectrum* of waveform w_s. $W_s(k)$ is a vector on the complex plane representing the k-th component of w_s. Similarly to Equation (3.3b), if $W_s(k) = a_k + b_k \imath$, where $\imath = \sqrt{-1}$, we can obtain the amplitude $A(k)$ and phase $\phi(k)$ of this component by

$$A(k) = \sqrt{a_k^2 + b_k^2} \quad \text{and} \quad \phi(k) = tan^{-1}\, \frac{b_k}{a_k} \tag{3.5c}$$

Given the amplitude values and the fact that $A^2 \propto P$, one can calculate the *power spectrum* of w_s, which consists of power assigned to each frequency component.

Notice that n represents the discrete time (corresponding to the continuous variable t), and k, the discrete frequency (corresponding to f). Both DFT and IDFT can be generalized to a continuous (real or complex) frequency domain, but this is not necessary for this work. Furthermore, both the DFT and the IDFT have an optimized implementation called the Fast Fourier Transform (FFT), for when N is a power of 2. The discrete transforms can be extended into *time-varying* versions to support the concept of quasi-periodicity. This is done by applying the DFT to small portions of the signal instead of the full domain.

3.1.2 Sound Generation and Propagation

Recall from the beginning of this section that sound can be generated when two objects touch each other. This is a particular case of *resonance*, in which the harmonic movement of molecules of the object is induced when the object receives external energy, more often in the form of waves. An object can, thus, produce sound when hit, rubbed, bowed, and also when it receives sound waves. All objects, when excited, have the natural tendency to oscillate more at certain frequencies, producing sound with more energy at them. Some objects produce sound where the energy distribution has clearly distinct modes. Another way of looking at resonance is that, actually, the object *absorbs* energy from certain frequencies.

As sound propagates, it interacts with many objects in space. Air, for example, absorbs some energy from it. Sound waves also interact with obstacles, suffering reflection, refraction, diffraction and dispersion. In each of those phenomena, energy from certain frequencies is absorbed, and the phase of each component can be changed. The result

of this is a complex cascade effect. It is by analyzing the characteristics of the resulting sound that animals are able to locate the position of sound sources in space.

The speed at which sound propagates is sometimes important to describe certain sound phenomena. This speed determines how much delay exists between generation and perception of the sound. It affects, for example, the time that each reflection of the sound on the surrounding environment takes to arrive at the ear. If listener and source are moving with respect to each other, the sound wave received by the listener is compressed or dilated in time domain, causing the frequency of all components to be changed. This is called *Doppler effect* and is important in the simulation of virtual environments.

3.1.3 Sound in Music

Recall from Equation (3.5c) that amplitude and phase offset can be calculated from the output values of the DFT for each frequency in the domain. Recall from the beginning of this section that the cochlea is the component of the human ear responsible for converting sound vibration into nervous impulses. Inside the cochlea, we find the inner *hair cells*, which transform vibrations in the fluid (caused by sound arriving at the ear) into electric potential. Each hair cell has an appropriate shape to resonate at a specific frequency. The potential generated by a hair cell represents the amount of energy on a specific frequency component. However not mathematically equivalent, the information about energy distribution produced by the hair cells is akin to that obtained by extracting the amplitude component from the result of a time-varying Fourier transform of the sound signal[4]. Once at the brain, the electric signals are processed by a complex neural network. It is believed that this network extracts information by calculating correlationships involving present and past inputs of each frequency band. Exactly how this network operates is still not well understood. It is, though, probably organized in layers of abstraction, in which each layer identifies some specific characteristics of the sound being heard, being what we call "attractiveness" processed at the most abstract levels—thus, the difficulty of defining objectively what music is.

The ear uses the spectrum information to classify sounds as *harmonic* or *inharmonic*. A harmonic sound presents high-energy components whose frequencies f_k are all integer multiples of a certain number f_1, which is called the *fundamental frequency*. In this case, components are called *harmonics*. The component with frequency f_1 is simply referred to as the *fundamental*. Normally, the fundamental is the highest energy component, though this is not necessary. An inharmonic sound can either present high-energy components at different frequencies, called *partials* or, in the case of *noise*, present no modes in the energy distribution on the sound's spectrum. In nature, no sound is purely harmonic, since noise is always present. When classifying sounds, then, the brain often abstracts from some details of the sound. For example, the sound of a flute is often characterized as harmonic, but flutes produce a considerably noisy spectrum, simply with a few stronger harmonic components. Another common example is the sound of a piano, which contains partials and is still regarded as a harmonic sound.

In music, a *note* denotes an articulation of an instrument to produce sound. Each note has three basic qualities:

- *Pitch*, which is the cognitive perception of the fundamental frequency;
- *Loudness*, which is the cognitive perception of acoustic power; and

[4] There is still debate on the effective impact of phase offset to human sound perception. For now, we assume that the ear cannot sense phase information.

- *Timbre* or *tone quality*, which is the relative energy distribution of the components. Timbre is what actually distinguishes an instrument from another. Instruments can also be played in different expressive ways (*e.g.,* piano, fortissimo, pizzicatto), which affects the resulting timbre as well.

The term *tone* may be used to refer to the pitch of the fundamental. Any component with higher pitch than the fundamental is named an *overtone*. The current more commonly used definition for *music* is that "it is a form of expression through structuring of tones and silence over time" (WIKIPEDIA, 2006b). All music is constructed by placing sound events (denoted as notes) in time. Notes can be played individually or together, and there are cultural rules to determining if a combination of pitches sound well when played together or not.

Two notes are said to be *consonant* if they sound stable and harmonized when played together. Pitch is the most important factor in determining the level of consonance. Because of that, most cultures have chosen a select set of pitches to use in music. A set of predefined pitches is called a *scale*.

Suppose two notes with no components besides the fundamental frequency being played together. If those tones have the same frequency, they achieve maximum consonance. If they have very close frequencies, but not equal, the result is a wave whose amplitude varies in an oscillatory pattern; this effect is called *beating* and is often undesirable. If one of the tones has exactly the double of the frequency of the other, this generates the second most consonant combination. This fact is so important that it affects all current scale systems. In this case, the highest tone is perceived simply as a "higher version" of the lowest tone; they are perceived as essentially the same tone. If now the higher tone has its frequency doubled (four times that of the lower tone), we obtain another highly consonant combination. Therefore, a scale is built by selecting a set of frequencies in a range $[f,\ 2f)$ and then by duplicating them to lower and higher ranges, *i.e.,* by multiplying them by powers of two. In other words, if S is the set of frequencies that belong to the scale, $p \in S$ and $f \leq p < 2f$, then $\left\{p \times 2^k \mid k \in \mathbb{Z}\right\} \subset S$.

In the western musical system, the scale is built from 12 tones. To refer to those tones, we need to define a notation. An *interval* is a relationship, on the numbering of the scale, between two tones. Most of the time, the musician uses only 7 of these 12 tones, and intervals are measured with respect to those 7 tones. The most basic set of tones if formed by several notes (named do, re, mi, fa, sol, la, si), which are assigned short names (C, D, E, F, G, A, B, respectively). Because generally only 7 notes are used together, a tone repeats at the 8th note in the sequence; thus, the interval between a tone and its next occurrence is called an *octave*. Notes are, then, identified by both their name and a number for their octave. It is defined that A4 is the note whose fundamental frequency is 440 Hz. The 12th tone above A4—*i.e.,* an octave above—is A5, whose frequency is 880 Hz. Similarly, the 12th tone below A4 is A3, whose frequency is 220 Hz.

In music terminology, the interval of two adjacent tones is called a *semitone*. If the tones have a difference of two semitones, the interval is called a *whole tone*[5]. The other 5 tones missing in this scale are named using the symbols ♯ (sharp) and ♭ (flat). ♯ indicates a positive shift of one semitone, and ♭ indicates a negative shift of one semitone. For example, C♯4 is the successor of C4. The established order of tones is: C, C♯, D, D♯, E, F, F♯, G, G♯, A, A♯, and B. C♯, for example, is considered equivalent to D♭. Other equivalents include D♯ and E♭, G♯ and A♭, E♯ and F, C♭ and B. One can assign frequencies to these

[5] A whole tone is referred to in common music practice by the term "tone", but we avoid this for clearness.

Table 3.1: The western musical scale. Note that frequency values have been rounded.

Note Names												
C4	C♯4	D4	D♯4	E4	F4	F♯4	G4	G♯4	A4	A♯4	B4	C5
262	277	294	311	330	349	370	392	415	440	466	494	523
Note Frequencies (in Hz)												

notes, as suggested in Table 3.1. Notice that the frequency for C5 is the double of that of C4 (except for the rouding error), and that the ratio between the frequency of a note and the frequency of its predecessor is $\sqrt[12]{2} \approx 1.059463$. In fact, the rest of the scale is built considering that the tone sequence repeats and that the ratio between adjacent tones is exactly $\sqrt[12]{2}^6$.

To avoid dealing with a complicated naming scheme and to simplify implementation, we can assign an index to each note on the scale. The MIDI standard, for example, defines the index of C4 as 60, C♯4 as 61, D as 62, etc. A translation between a note's MIDI index p and its corresponding frequency f is given by

$$f = 440 \times 2^{\frac{1}{12}(p-69)} \quad \text{and} \quad p = 69 + 12\, log_2 \left(\frac{f}{440} \right) \tag{3.6}$$

The scale system just described is called *equal tempered tuning*. Historically, tuning refers to manually adjusting the *tune* of each note of an instrument, *i.e.,* its perceived frequency. There are other tunings which are not discussed here. A conventional music theory course would now proceed to the study the consonance of combinations of notes of different pitches on the scale and placed on time, but this is not a part of this work's discussion and should be left for the performer only.

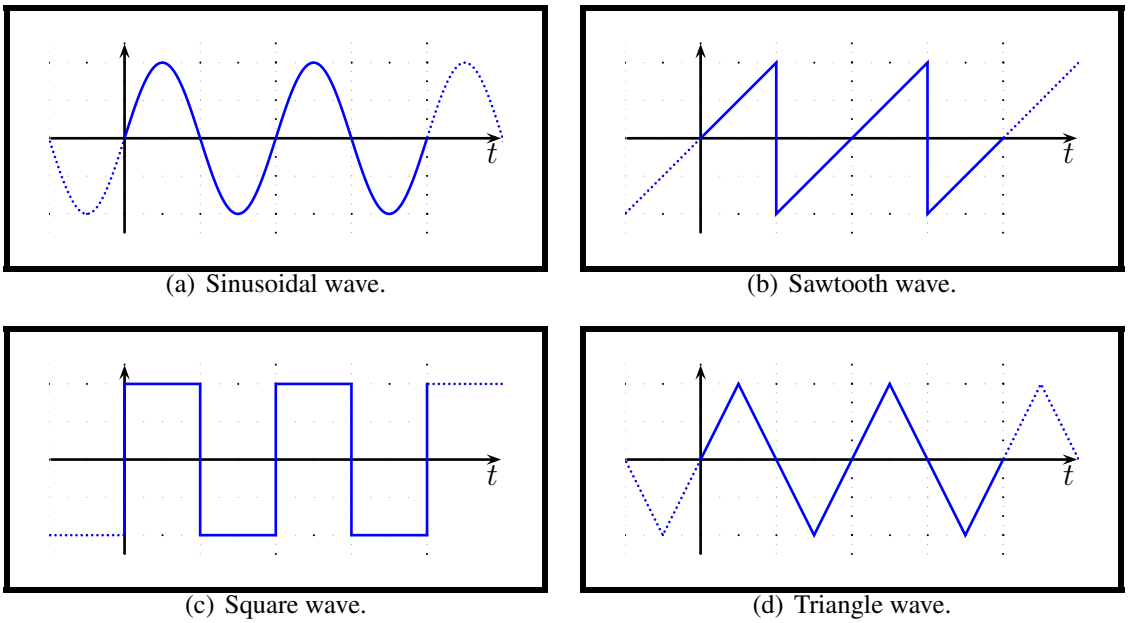
3.2 Introduction to Audio Systems

The term *audio* may refer to audible sound, *i.e.,* to the components of sound signals within the approximate range of 20 Hz to 20 kHz, which are perceivable to the human. It has been used also to refer to sound transmission and to high-fidelity sound reproduction. In this text, this term is used when referring to digitally manipulated sound information for the purpose of listening.

To work with digital audio, one needs a representation for sound. The most usual way of representing a sound signal in a computer is by storing amplitude values on an unidimensional array. This is called a *digital signal*, since it represents *discrete* amplitude values assigned to a *discrete* time domain. These values can be computed using a formula, such as Equation (3.4) or one of the formulas on Table 3.2. They can also be obtained by *sampling* the voltage level generated by an external *transducer*, such as a microphone. Another transducer can be used to convert back digital signals into continuous signals. Figure 1.1 illustrates both kinds of conversion. Alternatively, samples can also be recorded on digital media and loaded when needed.

Sounds can be sampled and played back from one or more points in space at the same time. Therefore, a digital signal has three main attributes: *sampling rate*, *sample format* and *number of channels*. The samples can be organized in the array in different arrangements. Normally, samples from the same channel are kept in consecutive positions. Sometimes, each channel can be represented by an individual array. However, it is

[6] This scale is called *equal tempered* scale, and it attempts to approximate the classical Ptolemaeus scale built using fractional ratios between tones.



(a) Sinusoidal wave. (b) Sawtooth wave.

(c) Square wave. (d) Triangle wave.

Figure 3.2: Primitive waveforms for digital audio synthesis.

Table 3.2: Formulas for primitive waveforms. The simplified formula of each wave can be evaluated with less operations, which may be useful in some situations.

Waveform	**General Formula**	**Simplified Formula**
Sinusoidal	$\text{SIN}(t) = sin\ 2\pi t$	
Sawtooth	$\text{SAW}(t) = 2\left(t - \left\lfloor t + \frac{1}{2} \right\rfloor\right)$	$\frac{1}{2}\,\text{SAW}\left(t + \frac{1}{2}\right) = t - \lfloor t \rfloor - \frac{1}{2}$
Square	$\text{SQR}(t) = 2\left(\lfloor t \rfloor - \left\lfloor t - \frac{1}{2} \right\rfloor\right) - 1$	$\frac{1}{2}\,\text{SQR}(t) = \lfloor t \rfloor - \left\lfloor t - \frac{1}{2} \right\rfloor - \frac{1}{2}$
Triangle	$\text{TRI}(t) = 4\left\lvert t - \left\lfloor t - \frac{1}{4} \right\rfloor - \frac{3}{4} \right\rvert - 1$	$\frac{1}{4}\,\text{TRI}\left(t + \frac{1}{4}\right) = \left\lvert t - \lfloor t \rfloor - \frac{1}{2} \right\rvert - \frac{1}{4}$

possible to merge samples from all channels in a single array by *interleaving* the samples of each channel.

Any arithmetic operation can be performed on a signal stored on an array. More elaborate operations are generally devised to work with the data without any feedback to the user until completion. This is called *offline* audio processing, and it is easy to work with. In *online* processing, the results of computation are supposed to be heard immediately. But in real applications, there is a slight time delay, since any computation takes time and the data need to be routed through components of the system until they are finally transduced into sound waves. A more adequate term to characterize such a system is *real time*, which means that the results of computation have a limited amount of time to be completed.

To perform audio processing in real time, though, instead of working with the full array of samples, we need to work with parts of it. Ideally, a sample value should be transduced as soon as it becomes available, but this leads to high manufacturing costs of audio devices and imposes some restrictions to software implementation (since computation time may vary depending on the program being executed, it may require executing a high amount of instructions). This is, though, a problem that affects any real-time audio system. If the required sample value is not available at the time it should be transduced, another value will need to be used instead of it, producing a sound different from what was originally intended. The value used for filling gaps is usually zero. What happens when a smooth waveform abruptly changes to a constant zero level is the insertion of many high-frequency components into the signal. The resulting sound has generally a noisy and undesired "click"[7]. This occurrence is referred to as an audio *glitch*. In order to prevent glitches from happening, samples are stored on a temporary buffer before being played. Normally, buffers are divided in blocks, and a scheme of buffer swapping is implemented. While one block is being played, one or more blocks are being computed and queued to be played later.

The *latency* of a system is the time interval between an event in the inputs of the system and the corresponding event in the outputs. For some systems (such as an audio system), latency is a constant determined by the sum of latencies of each system component through which data is processed. From the point of view of the listener, latency is the time between a controller change (such as key being touched by the performer) and the corresponding perceived change (such as the wave of a piano string reaching the performer's ears). Perceived latency also includes the time of sound propagation from the speaker to the listener and the time for a control signal to propagate through the circuitry to the inputs of the audio system. On a multi-threaded system, latency is variably affected by race conditions and thread scheduling delays, which are unpredictable but can be minimized by changing thread and process priorities.

It is widely accepted that real-time audio systems should present latencies of around 10 ms or less. The latency L introduced by a buffered audio application is given by

$$L = \text{number of buffers} \times \frac{\text{samples per buffer block}}{\text{samples per time unit}} \tag{3.7}$$

Many physical devices, named *controllers*, have been developed to "play" an electronic instrument. The most common device is a *musical keyboard*, in which keys trigger the generation of *events*. On a software application, events are normally processed

[7] Note, however, that the square wave on Figure 3.2(c) is formed basically of abrupt changes and still is harmonic—this is formally explained because the abrupt changes occur in a periodic manner. Its spectrum, though, presents very intense high-frequency components.

through a message-driven mechanism. The attributes given to each type of event are also application-dependent and vary across different controllers and audio processing systems. The maximum number of simultaneous notes that a real-time audio system can process without generating audio glitches (or without any other design limitation) is called *polyphony*. In software applications, polyphony can be logically unrestricted, limited only by the the processor's computational capacity. Vintage analog instruments[8] often had a polyphony of a single note (also called *monophony*), although some of them, such as the Hammond organ, have full polyphony.

3.2.1 Signal Processing

3.2.1.1 Sampling and Aliasing

In a digital signal, samples represent amplitude values of an underlying continuous wave. Let $n \in \mathbb{N}$ be the index of a sample of amplitude $w_s(n)$. The time difference between samples $n+1$ and n is a constant and is called *sample period*. The number of samples per time unit is the *sample frequency* or *sampling rate*.

The sampling theorem states that a digital signal sampled at a rate of R can only contain components whose frequencies are at most $\frac{R}{2}$. This comes from the fact that a component of frequency exactly $\frac{R}{2}$ requires two samples per cycle to represent both the positive and the negative oscillation. An attempt of representing a component of higher frequencies results in a reflected component of frequency below $\frac{R}{2}$. The theorem is also known as the *Nyquist theorem* and $\frac{R}{2}$ is called the *Nyquist rate*. The substitution of a component by another at lower frequency is called *aliasing*. Aliasing may be generated by any sampling process, regardless of the kind of source audio data (analog or digital). Substituted components constitute what is called *artifacts*, *i.e.,* audible and undesired effects (insertion or removal of components) resulting from processing audio digitally.

Therefore, any sampled sound must be first processed to remove energy from components above the Nyquist rate. This is performed using an analog low-pass filter. This theorem also determines the sampling rate at which real sound signals must be sampled. Given that humans can hear up to about 20 kHz, the sampling rate must be of at least 40 kHz to represent the highest perceivable frequency. However, frequencies close to 20 kHz are not represented well enough. It is observed that those frequencies present an undesired beating pattern. For that reason, most sampling is performed with sampling rates slightly above 40 kHz. The sampling rate of audio stored on a CD, for example, is 44.1 kHz. Professional audio devices usually employ a sampling rate of 48 kHz. It is possible, though, to use any sampling rate, and some professionals have worked with sampling rates as high as 192 kHz or more, because, with that, some of the computation performed on audio signals is more accurate.

When the value of a sample $w_s(n)$ is obtained for an underlying continuous wave w at time t, it is rounded to the nearest digital representation of $w(t)$. The difference between $w_s(n)$ and $w(t)$ is called the *quantization error*, since the values of $w(t)$ are actually *quantized* to the nearest values. Therefore, $w_s(n) = w(t(n)) + \xi(n)$, where ξ is the *error signal*. This error signal is characterized by many abrupt transitions and thus, as we have seen, is rich in high-frequency components, which are easy to perceive. When designing an audio system, it is important to choose an adequate sample format to reduce the quantization error to an imperceptible level. The RMS of ξ is calculated using Equation (3.1). The ratio between the power of the loudest representable signal and

[8] An analog instrument is an electric device that processes sound using components that transform continuous electric current instead of digital microprocessors.

the power of ξ, expressed in decibels according to Equation (3.2), is called the *signal-to-quantization-error-noise ratio* (SQNR). Usual sampling rates with a high SQNR are 16, 24 or 32-bit integer, 32 or 64-bit floating point. The sample format of audio stored on a CD, for example, is 16-bit integer. There are alternative quantization schemes, such as logarithmic quantization (the sample values are actually the logarithm of their original values), but the floating point representation usually presents the same set of features.

Analog-to-Digital Converters (ADC) and *Digital-to-Analog Converters* (DAC) are transducers[9] used to convert between continuous and discrete sound signals. These components actually convert between electric representations, and the final transduction into sound is performed by another device, such as a loudspeaker. Both DACs and ADCs must implement analog filters to prevent the aliasing effects caused by sampling.

3.2.1.2 Signal Format

Common possible representations of a signal are classified as:

- *Pulse coded* modulation (PCM), which corresponds to the definition of audio we have presented. The sample values of a PCM signal can represent
  - *linear* amplitude;
  - *non-linear* amplitude, in which amplitude values are mapped to a different scale (*e.g.,* in logarithmic quantization); and
  - *differential* of amplitude, in which, instead of the actual amplitude, the difference between two consecutive sample's amplitude is stored.
- *Pulse density* modulation (PDM), in which the local density of a train of pulses (values of 0 and 1 only) determines the actual amplitude of the wave, which is obtained after the signal is passed through an analog low-pass filter;
- *Lossy* compression, in which part of the sound information (generally components which are believed not to be perceptible) is removed before data compression; and
- *Lossless* compression, in which all the original information is preserved after data compression.

Most audio applications use the PCM linear format because digital signal processing theory—which is based on continuous wave representations in time—generally can be applied directly without needing to adapt formulas from the theory. To work with other representations, one would often need, while processing, to convert sample values to a PCM linear format, perform the operation, and then convert the values back to the working format.

Common PCM formats used to store a single sound are:

- *CD audio*: 44.1 kHz, 16-bit integer, 2 channels (stereo);
- *DVD audio*: 48–96 kHz, 24-bit integer, 6 channels (5.1 surround); and
- *High-end studios*: 192–768 kHz, 32–64-bit floating point, 2–8 channels.

[9] A transducer is a device used to convert between different energy types.

The bandwidth (the number of bytes per time unit) expresses the computational transfer speed requirements for audio processing according to the sample format. A signal being processed continuously or transferred is a type of data *stream*. Considering uncompressed formats only, the bandwidth B of a sound stream of sampling rate R, sample size S and number of channels C is obtained by $B = RSC$. For example, streaming CD audio takes a bandwidth of 176.4 kBps, and a sound stream with sampling rate of 192 kHz, sample format of 64-bit and 8 channels has a bandwidth of 12,288 kBps (12 MBps).

3.2.2 Audio Applications

The structure of major audio applications varies widely, depending on their purpose. Among others, the main categories include:

- *Media players*, used simply to route audio data from its source (a hard disk, for example) to the audio device. Most media players also offer basic DSPs (digital signal processors), such as equalizers, presented to the user on a simplified graphical interface that does not permit customizing the processing model (*e.g.,* the order in which DSPs are applied to the audio);

- *Scorewriters*, used to typeset song scores on a computer. Most scorewriters can convert a score to MIDI and play the MIDI events. Done this way, the final MIDI file generally does not include much expressivity (though there has been work to generate expressivity automatically (ISHIKAWA et al., 2000)) and is normally not suitable for professional results;

- *Trackers* and sequencers, used to store and organize events and samples that compose a song. This class of application is more flexible than scorewriters, since they allow full control over the generation of sound; and

- *Plug-in-based applications*, in which computation is divided in software pieces, named *modules* or *units*, that can be integrated on the main application, responsible for routing signals between modules.

Some audio applications present characteristics belonging to more than one category (*e.g., digital audio workstations* include a tracker and/or a sequencer and are based on a plug-in architecture). Plug-in based applications are usually the most interesting for a professional music producer, since it can be extended with modules from any external source.

3.2.2.1 Modular Architecture

The idea behind a modular architecture is to model in software the interconnection of processing units, similar to the interconnection of audio equipment with physical cables. A *module* is composed of inputs, outputs and a processing routine. The main advantages of a modular architecture are that computation is encapsulated by modules and that their results can be easily combined simply by defining the interconnection between modules, which can be done graphically by the user.

The processing model is a directed acyclic graph which models an unidirectional network of communication between modules. Figure 3.3 presents an example of a processing model on a modular architecture that a user can generate graphically. Each module has a set of input and output slots, depicted as small numbered boxes. Modules A and B are named *generators*, since they only generate audio signals, having no audio inputs. C, D

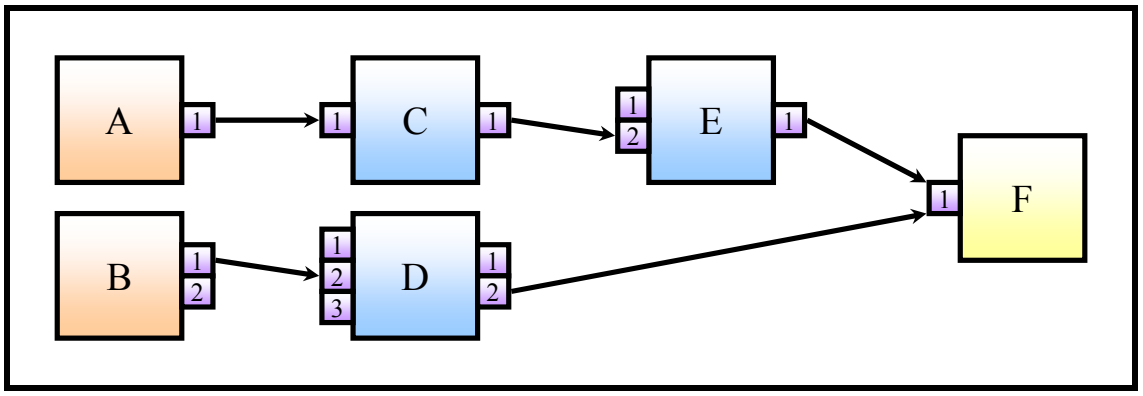


Figure 3.3: Graphical representation of a processing model on a modular architecture. A, B, C, D, E and F are modules. Numbered small boxes represent the input and output slots of each module. The arrows represent how inputs and outputs of each module are connected, indicating how sound samples are passed among module programs.

and E are named *effects*, since they operate on inputs and produce output signals with the intent of changing some characteristic of the input sound—*e.g.,* energy distribution of components, in the case of an equalizer effect. F is a representation of the audio device used to play the resulting sound wave. Each module can receive controller events, which are used to change the value of parameters used for computation. A generator also uses events to determine when a note begins and stops playing.

On Figure 3.3, the signals arriving at module F are actually implicitly *mixed* before being assigned to the single input slot. Recall from Section 3.1.1 that the interference of multiple waves at time t produces a wave resulting from the sum of the amplitudes of all interacting waves at time t at the same point. For that reason, mixing consists of summing corresponding samples from each signal. Other operations could be made implicit as well, such as, for example, signal format conversion. Another important operation (not present on the figure) is *gain* which scales the amplitude of the signal by a constant, thus altering its perceived loudness. One may define that each input implicitly adds a gain parameter to the module parameter set. Together, gain and mixing are the most fundamental operations in *multi-track recording*, in which individual tracks of a piece are recorded and then mixed, each with a different amount of gain applied to. This allows balancing the volume across all tracks.

Remark On a more general case, the processing model shown in Figure 3.3 could also include other kinds of streams and modules, such as video and subtitle processors. However, the implementation of the modular system becomes significantly more complex with the addition of certain features.

3.2.3 Digital Audio Processes

In this section, several simple but frequently used audio processes are presented. These algorithms were selected to discuss how they can be implemented on the GPU[10]. More information about audio processes can be found at TOLONEN; VÄLIMÄKI; KARJALAINEN (1998). In the following subsections, n represents the index of the "current" sample being evaluated, x is the input vector and y is the output vector. $x\,(n)$ refers to the sample at position n in x and is equivalently denoted as x_n on the following figures, for clarity.

[10] See Section 4.4.

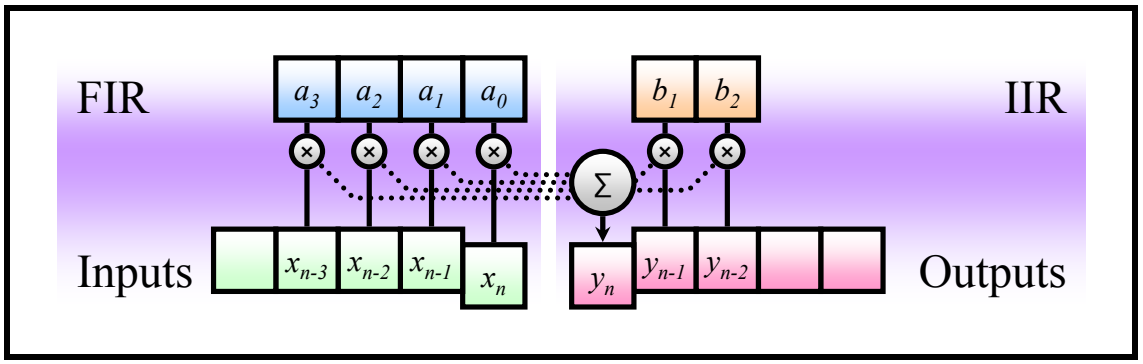


Figure 3.4: Processing model of a filter in time domain.

3.2.3.1 Filtering

A filter on time domain can be expressed as

$$\begin{aligned} y(n) = a_0\, x(n) + a_1\, x(n-1) + a_2\, x(n-2) + \ldots + a_k\, x(n-k) \\ + b_1\, y(n-1) + b_2\, y(n-2) + \ldots + b_j\, y(n-l) \end{aligned} \tag{3.8}$$

If $b_i = 0$ for any i, the filter is called a *finite impulse response* (FIR) filter. Otherwise, it is called a *infinite impulse response* (IIR) filter. Figure 3.4 illustrates how a filter is calculated.

The theory of digital filters (ANTONIOU, 1980; HAMMING, 1989) studies how the components of x are mapped into components of y according to the values assigned for each coefficient a_i, b_j. Here, we only discuss how a filter can be implemented given its coefficients. We also do not discuss *time-varying filters*, due to the complexity of the theory.

When processed on the CPU, a digital filter is obtained simply by evaluating Equation (3.8) for each output sample. The program, though, needs to save the past $k+1$ input and $l+1$ output samples. This can be done using auxiliary arrays.

3.2.3.2 Resampling

Resampling refers to the act of sampling an already sampled signal. Resampling is necessary when one needs to place a sampled signal over a time range whose length is different from the original length, or, equivalently, if one wants to convert the sampling rate of that signal. Therefore, to resample a wave, one needs to generate sample values for an arbitrary intermediary position t that may not fall exactly on a sample. The most usual methods to do this, in ascending order of quality, are:

- *Rouding* t to the nearest integer and returning the corresponding sample value;
- *Linear interpolation*, in which the two adjacent samples are interpolated using the fractional part of t. This is illustrated on Figure 3.6;
- *Cubic interpolation*, which works similarly to linear interpolation, taking 4 adjacent samples and fitting a cubic polynomial to these values before obtaining the value for the sample; and
- *Filtering*, in which a FIR filter with coefficients from a windowed $sinc$ function is used to approximate a band-limited representation of the signal[11].

[11] A simple explanation on how resampling can be performed using filters can be found at AUDIO DSP TEAM (2006). For more information, see TURKOWSKI (1990).

Table 3.3: Fourier series of primitive waveforms $w(t)$, where w corresponds to the functions SAW, SQU and TRI presented in Table 3.2. The simplified version of TRI generates a wave that approximates TRI $\left(t + \frac{1}{4}\right)$.

Waveform	**Fourier Series**	**Simplified Series**
Sawtooth	$\frac{2}{\pi}\sum_{k=1}^{\infty}\frac{sin\ 2\pi kt}{k}$	
Square	$\frac{4}{\pi}\sum_{k=1}^{\infty}\frac{sin\ 2\pi\,(2k-1)\,t}{(2k-1)}$	
Triangle	$\frac{8}{\pi^2}\sum_{k=1}^{\infty}(-1)^{k+1}\frac{sin\ 2\pi\,(2k-1)\,t}{(2k-1)^2}$	$\frac{8}{\pi^2}\sum_{k=1}^{\infty}\frac{cos\ 2\pi\,(2k-1)\,t}{(2k-1)^2}$

Resampling causes noise to be added to the original sampled wave. Among the cited methods, filtering achieves the highest SNR ratio and is, therefore, the most desirable. Rouding is generally avoided, since it introduces significant amounts of noise[12]. A last method, calculating the sample values through an IFFT from the FFT of the input signal, is generally not applied, since computing those transformations is expensive.

3.2.3.3 Synthesis

Synthesis refers to the generation of sound samples. Normally, synthesis is responsible for generating the signal for note events. You should recall from Section 3.1.3 that a note has three main attributes: pitch, intensity and timbre. On a method of digital synthesis, pitch and intensity generally are parameters, and timbre depends on the method being implemented and eventually additional parameter values.

The most basic type of synthesis is based on the aforementioned primitive waveforms. It consists simply of evaluating the formulas on Table 3.2 for the parameter $u = ft$ (*i.e.,* replacing t by u in these formulas), where f is the frequency assigned to the note's pitch according to Equation (3.6) and t represents the time. On the digital domain, $t = \frac{n}{R}$, where n is the sample's index on the array used to store the signal and R is the sampling rate.

Since all primitives, except for the sinusoidal wave, have many high-frequency components, direct sampling, due to the Nyquist theorem, causes aliasing, and thus are subject to the introduction of artifacts. Several methods have been suggested to generate *band-limited* versions of those waveforms (STILSON; SMITH, 1996). One of the simplest ways is to evaluate the respective summation on Table 3.3 for a finite number of terms. This method, however, has high computational requirements and is often used only for offline processing.

All notes have a beginning and an end. On real instruments, notes are characterized by stages in which the local intensity varies smoothly. If, for example, we stopped processing a sinusoidal wave when the amplitude value was close to one of its peaks, the result would be an abrupt change to zero that, as mentioned on Section 3.2, results in an

[12] However, some effects, such as *sample and hold*, intentionally apply the rouding method for musical aesthetics.

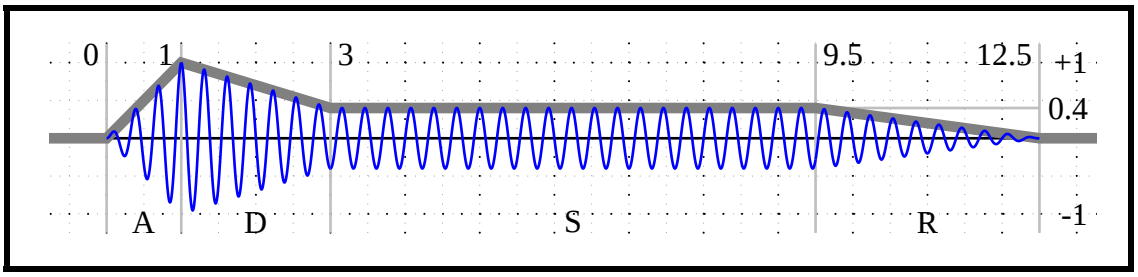


Figure 3.5: An ADSR envelope (thick line) with $a = 1$, $d = 2$, $s = 0.4$, $e = 9.5$ and $r = 3$ applied to a sinusoidal wave with $f = 3.25$.

undesirable click sound. We would like the sinusoid to gradually fade to zero amplitude along time, just as the sound of a natural instrument would do. One way to do this is to apply an *envelope* to the signal. The most famous envelope is comprised of four stages: attack, decay, sustain and release (ADSR). More complex envelopes could be designed as well, but the ADSR is widely used due to its simple implementation and easy user parametrization. Given the attack time a, the decay time d, the sustain level s, the end of the note time e and the release time r, an ADSR envelope can be defined by the stepwise function

$$f(t) = \begin{cases} 0 & \text{for } t < 0 \\ map\left(t, \langle 0, a\rangle, \langle 0, 1\rangle\right) & \text{for } 0 \leq t < a \\ map\left(t, \langle a, a+d\rangle, \langle 1, s\rangle\right) & \text{for } a \leq t < a+d \\ s & \text{for } a+d \leq t < e \\ map\left(t, \langle e, e+r\rangle, \langle s, 0\rangle\right) & \text{for } e \leq t < e+r \\ 0 & \text{for } e+r \leq t \end{cases} \tag{3.9}$$

where $map\left(t, \langle t_0, t_1\rangle, \langle a_0, a_1\rangle\right)$ maps values in the range $[t_0,\ t_1]$ into values in the range $[a_0,\ a_1]$ ensuring that $map\left(a_0, \langle t_0, t_1\rangle, \langle a_0, a_1\rangle\right) = t_0$ and $map\left(a_1, \langle t_0, t_1\rangle, \langle a_0, a_1\rangle\right) = t_1$. A linear interpolation can be defined as

$$map_{lin}\left(t, \langle t_0, t_1\rangle, \langle a_0, a_1\rangle\right) = a_0 + (a_1 - a_0)\frac{t - t_0}{t_1 - t_0} \tag{3.10}$$

The resulting signal $h(t)$ after an envelope $f(t)$ to an input signal $g(t)$ is simply $h(t) = f(t)\, g(t)$. An illustration of the ADSR envelope defined with the function map_{lin} are discussed by Figure 3.5.

Another synthesis technique is *wavetable synthesis*, which consists of storing a sample of one or more cycles of a waveform on a table and then resampling that table at specific positions. This is depicted on Figure 3.6. One can also have multiple tables and sample and combine them in many different ways. Wavetable synthesis is similar to the most widely used technique, *sample playback*, in which a table containing more than a few cycles is used and a *loop range* is defined. Along time, the beginning of the table is sampled first and then only the loop range is repetitively used. Both techniques (which are essentially the same) require a translation from the parameter t into the corresponding "index" on the table, which now can be a non-integer number (thus requiring resampling techniques). Given a sample with a loop region containing n cycles defined from l_0 to l_1 with a number of samples $N = l_1 - l_0$, the sampling index $I(t)$ corresponding to time t is

$$I(t) = \frac{t}{nN}$$

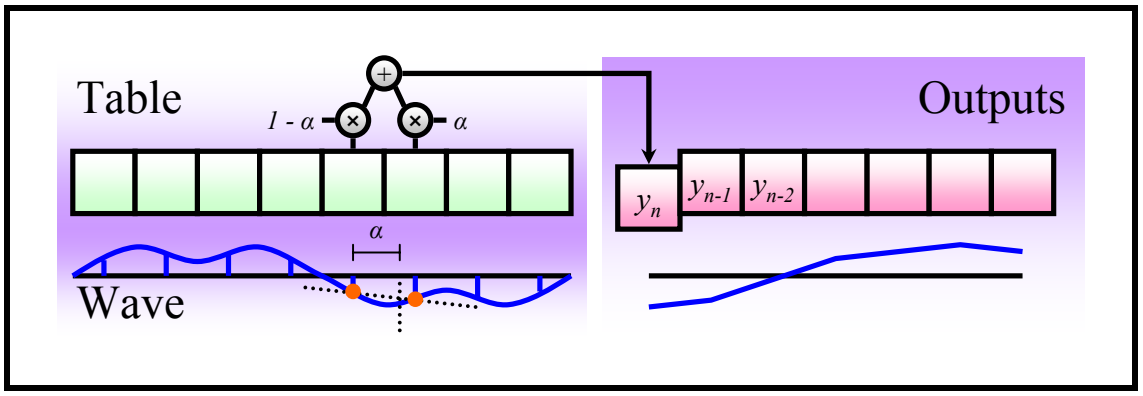


Figure 3.6: Illustration of linear interpolation on wavetable synthesis. f represents the fractional part of the access coordinate.

Before accessing a sample, an integer index i must be mapped to fall within a valid range into the sample, according to the access function

$$S(i) = \begin{cases} i & \text{for } i < l_1 \\ l_0 + [(i - l_1) \textbf{ mod } N] & \text{for } i \geq l_1 \end{cases}$$

which, in the case of wavetable synthesis, simplifies to $S(i) = i$ **mod** N. Performing wavetable synthesis, then, requires the following steps:

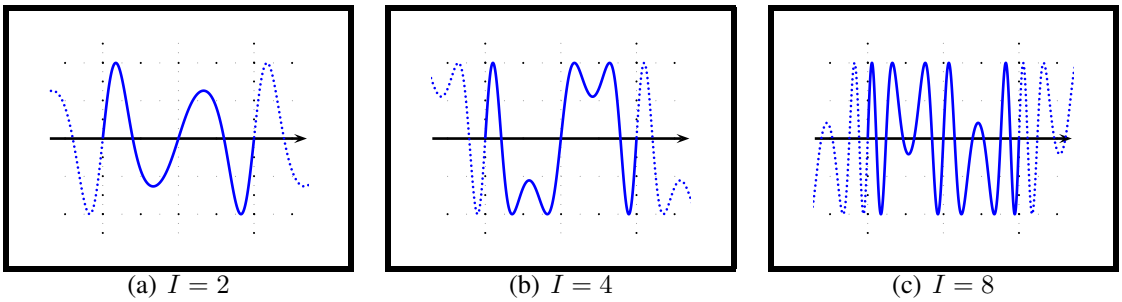
- Calculate the sampling index $I(u)$ for the input parameter u. If f is a constant, then $u = ft$ and f can be obtained from Equation (3.6);
- For each sample i adjacent to $I(u)$, calculate the access index $S(i)$; and
- Interpolate using the values from positions in the neighborhood of $S(i)$ in the array.

After resampling, an envelope may be applied to the sample values in the same way it was applied to the primitive waveforms, as previously discussed. Advanced wavetable techniques can be found in BRISTOW-JOHNSON (1996).

The most general synthesis technique is *additive synthesis*, which consists of summing sinusoidal components, similar to computing the Fourier series of a signal. Most of the time, chosen components are harmonics of the frequency of the playing note, but this is not required. In the simplest case, when the amplitude of components is not modulated by any envelope, additive synthesis is equivalent to wavetable synthesis (BRISTOW-JOHNSON, 1996, p. 4). In practice, though, an individual envelope is applied to each component. Since many components may be present, it is usual to extract parameters for each individual envelope using data reduction techniques.

The idea of producing complex sounds from an array of simpler sounds applies not only to sinusoidal waves. Components can be generated with any synthesis method and summed together by applying different gains to each signal. For example, additive synthesis is often imitated with wavetable synthesis by summing the output of several wavetables and applying individual envelopes. When dealing with a set of components, one can form groups and control each group individually—for example, instead of applying one envelope to each component, one may apply an envelope to groups of components.

Instead of summing components, one can also start with a complex waveform—*i.e.,* a sawtooth wave—and remove components from it, usually by filtering. This is called



(a) $I = 2$ (b) $I = 4$ (c) $I = 8$

Figure 3.7: Examples of FM waveforms with $f_m = f_c = 1$.

subtractive synthesis and it is the main method used for *physical modelling*, the simulation of sound produced by real-world sound sources and its interaction with nearby objects.

The previously mentioned parameter u can be understood as a variable that is incremented from sample to sample. Though the time parameter t varies, by definition, as a constant between any two adjacent samples, the frequency parameter may change. This way, u can be redefined as $u(t) = \int f(t)\,dt$, where $f(t)$ is the frequency at time t. This is called *frequency modulation* (FM). FM synthesis is usually applied to synthesize bell and brass-like sounds. $f(t)$ is the *modulator*, which is usually a sinusoidal function multiplied by an amplitude parameter. The function to which u is applied is the *carrier*, which is also usually a sinusoid. In the case where sinusoids are used for both carrier and modulator, an FM wave can defined as

$$w(t) = A\,sin\left(2\pi f_c t + I\,sin\left(2\pi f_m t\right)\right) \tag{3.11}$$

The index of modulation I is defined as $I = \frac{\Delta f}{f_m}$, where Δf is the amount of frequency deviation. Higher values of I cause the distribution of energy on high-frequency components to increase. The index of harmonicity H is defined as $H = \frac{f_m}{f_c}$. Whenever $H = \frac{1}{N}$ or $H = N$ for any $N \in \mathbb{N}$ and $N \geq 1$, the resulting waveform is harmonic with its fundamental at $\min\{f_c, f_m\}$. If I and H are both small but non-null, the result is a *vibratto* effect. However, as N grows close to 1, the harmonic content of the sinusoid starts to change noticeably, as well as its waveform. Figure 3.7 presents several waveforms generated by frequency modulation. This technique can be expanded by defining $f(t)$ as an FM function itself. This way, one may recursively generate any level of modulation. The harmonic content of an FM wave is the subject of a complicated theory whose discussion is beyond the scope of this work.

Sometimes it is necessary to synthesize *noise* too. There are many kinds of noises, but most can be obtained from *white noise*, which is characterized by an uniform distribution of energy along the spectrum. For example, *pink noise* (energy distributed uniformly on a *logarithmic* domain) can be approximated using well designed low-pass filters over a white noise input or, as done by the Csound language, using *sample and hold*. An ideal pseudo-random number generator, such as those implemented in most programming libraries provided with compilers, generates white noise.

At last, *granular synthesis* consists of mixing along time many small sound units named *granule*. A granule is simply a signal which often, but not necessarily, has small length (around 50 ms) and smooth amplitude variation, fading to zero at the edges of the granule. The number of different granules, size, frequency of resampling and the density of placement along time are variable parameters. Given that the number of granules is normally too large to be handled by a human, the study of granular synthesis focuses on

methods of controlling the choice of granules and their parameters more automatically.

There are numerous variations and combinations of the methods described on this subsection, and other methods, such as formant synthesis, phase distortion, etc. There are also specific usages of the described methods for generating percussive sounds.

3.2.3.4 Effects

In this section, two simple and similar effects used frequently for music production are described. A *tap delay* or simply a *delay* is such that of a repetition (tap) of some input sound is produced on the output after its first occurrence. A delay effect may have multiple taps, such that several repetitions of an input sound will be produced on future outputs. It can be defined as

$$y(n) = g_w\, x(n) + g_0\, x(n - k_0) + g_1\, x(n - k_1) + \ldots + g_N\, x(n - k_N)$$

where g_w represents the *wet gain*—the amount of current input signal that will be immediately present on the output—, g_0, $g_1 \ldots$, g_n are the gains respective to each tap, and k_0, $k_1 \ldots$, k_N are the sample-positional delays of each tap. In general, the delays are of more than 100 ms for the delayed repetition to be perceived as a repetition instead of part of the original sound.

A similar effect is an *echo*, which consists of an infinite repetition of any sound that comes into the input. An echo can be defined as

$$y(n) = g_w\, x(n) + g_f\, y(n - k)$$

where g_f represents the *feedback gain*—the change applied to the volume of a past *output* signal at its next repetition—and k is the time delay between two successive echoes. Generally, successive repetitions fade out as they repeat. This behavior can be obtained by enforcing $|f| < 1$. Setting either of the constants g_w or g_f to a negative value causes an inversion of the signal being multiplied with the respective constant, which generally does not produce any audible difference[13]. Notice that a delay and an echo effect are actually a FIR and an IIR filter, respectively. A 1-tap delay and an echo are depicted on Figure 3.8. More complicated delay schemes can be designed, such as cross-delays (between channels) and multi-tap feedback delays. On a more general case, the delay times k may be substituted by non-integer values, thus requiring resampling of the signal to generate taps.

There are many other effects that are not described in this work. For instance, an important effect is *reverberation*, in which multiple echoes with varying delay are applied; filtering is also often applied to simulate energy absorption by objects of an environment. Chorus, flanging, sample and hold, ring modulation, resynthesis, gapping, noise gating, etc. are other interesting audio processes.

3.3 Audio Streaming and Audio Device Setup

As discussed in Section 3.2, processing audio in real time requires the audio data to be divided in blocks. Generally, for simplicity, all blocks have a fixed amount of samples. After properly loaded, the driver periodically requests a block of audio from the

[13] An inversion of amplitude corresponds to a shift of π on each component. When two components of same frequency but phases with a distance of π are summed, they cancel each other (this is a destructive interference). Thus, in some cases, inverting the signal may produce audible effects.

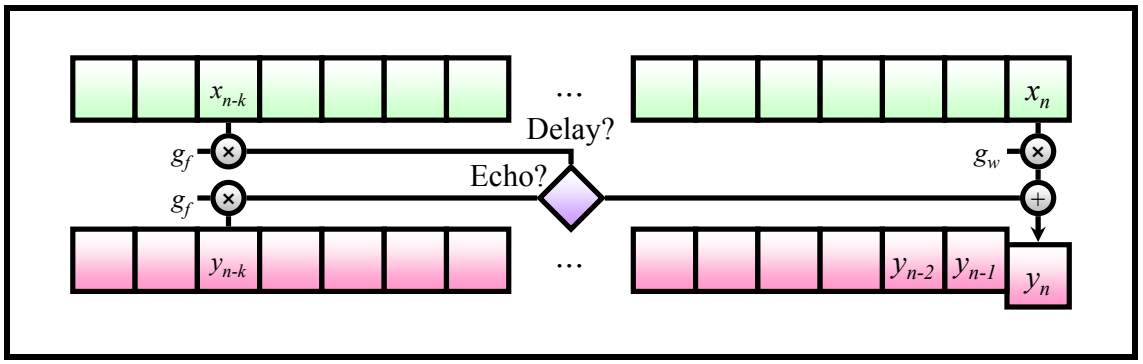


Figure 3.8: Combined illustration of a 1-delay and an echo effect. The input for the sum changes according to the effect one wants to implement: for the delay, it is $g_f\, x_{n-k}$; for the echo effect, it is $g_f\, y_{n-k}$.

application, providing or not an input block with samples that have been recorded from the device's input channels in the previous step. For a continuous output, the application is given a certain amount of time to compute samples and provide the requested block to the audio device.

If this computation takes longer than the given time, the behavior of the audio device is unspecified[14], but if one wishes to define what is played by the sound card if this happens, a possible solution is by doing the following:

- Create an intermediary buffer containing at least two blocks of samples;
- Create a thread to fill this buffer;
- Create a semaphore for the CPU to determine when the thread should write a block with the data being generated, *i.e.,* after being copied into an output block during one execution of the callback procedure; and
- Access this buffer from the callback procedure to retrieve one completely computed block, if necessary.

If more than two blocks are stored in the intermediary buffer, this procedure will buffer data ahead and increase the tolerance of the system against the occurrence of audio glitches due to heavy computation. This is a typical producer-consumer problem, where the created thread produces blocks of audio data, which are consumed by the audio driver through the callback procedure. In our implementation, if a block is not ready, the audio driver receives a block filled with silence.

As explained in Section 3.2, the format of audio contained in a block is defined by its sampling rate, sample format, number of channels and whether the data is interlieved or not. The available combinations of those characteristics vary depending on the audio device, driver and the API used to access the driver. A problem arises when one tries to run an application using an unsupported audio format. One may implement a program using many different formats and then choose the one supported by the device, or he/she may also convert the audio format prior to playback. Though not all, several libraries provide automatic audio format conversion to a certain extent. We have chosen to

[14] It seems acceptable that sound cards will play again the last completely computed block or simply silence. In both cases, though, there is a high chance that an abrupt change in amplitude will happen at the transition following the last sample of that block, causing an undesired glitch.

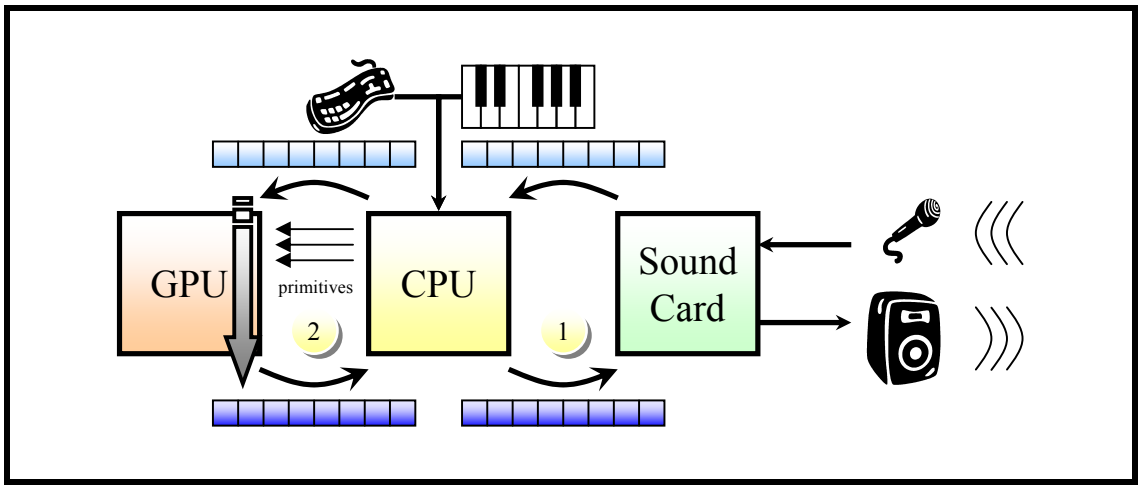


Figure 3.9: Illustration of the audio "pipeline". Audio streaming is performed on step 1, and processing on the GPU is performed on step 2.

work with RtAudio (SCAVONE, 2005), which offers interesting programming options—*e.g.,* providing audio through blocking calls instead of a callback procedure—, is portable and supports, on Microsoft Windows, the ASIO (STEINBERG, 2006) and DirectSound (MICROSOFT, 2006a) interfaces. RtAudio can convert the sample format and the interlieving of data as necessary, but sampling rate and number of channels still must be determined properly by the application.

Figure 3.9 the whole process of real-time audio processing. Audio streaming takes place on step 1, while processing on the CPU or the GPU takes place on step 2. The square blocks between each device represent the exchanged sample buffers. As we discuss in Section 4.3, the CPU must also handle input from control devices such as keyboards, which are used by the musician to play a music piece.

Before we define the internal operations performed by the producer thread, we need to explain how graphics systems work and define how to prepare them for audio processing.

3.4 Summary

This chapter presented many concepts involved in audio processing. It discussed the underlying physical concepts of sound, mentioned how sound behaves in environments which humans are used to, and presented how sound is organized in music and how a computer system can be used to produce music. At last, we presented some of the most fundamental sound processing algorithms.

Now all the theoretical tools necessary to start building computer systems for music production have been provided. The reader should be able to visualize (at least abstractly) how to implement those algorithms. The next chapter discusses the implementation of a modular system that facilitates the development of new modules that take advantage of the graphics hardware.

4 AUDIO PROCESSING ON THE GPU

In the previous chapter, we discussed the basic concepts of physical sound and of audio processing on a computer. The definition of several audio processing algorithms were given, and guidelines for the interaction with an audio device driver were given. In this chapter, a study of how graphics processing systems can be used to implement the given algorithms is presented. We present a real-time application in which sample values are obtained using fragment programs.

The first section of this chapter presents an "audio pipeline", describing which computation needs to be executed at each step of audio processing. Then, graphics processing is described as it is usually performed today with high-end graphics hardware. The relationship between entities in the contexts of graphics and audio processing is established. The state configuration required by the graphics and audio subsystems to support this application is presented. Finally, the implemented system is discussed in detail.

4.1 Introduction to Graphics Systems

An *image* is an artifact that reproduces some characteristic of a subject. Images may have a multidimensional source and target. A digital image refers to a finite discrete two-dimensional function, represented as a 2D array. In the case of a color image, the codomain of such a function usually has 3 dimensions, one for each of the red, green and blue color channels.

When the color values are generated by computation, the process is called *rendering*. Generally, "rendering" means generating an image from 3D models made of smaller geometric primitives such as points, lines and triangles, each of these represented as one, two or three *vertices*, respectively. A vertex is the representation of a point in 2D or 3D space corresponding to a corner of a flat polygonal shape. Models can be based on more complex geometrical representations, which need to be decomposed into these simple primitives for rendering.

Graphics accelerator cards are designed to speed up the rendering of these primitives, offering support to simulate many characteristics present in real-world images, such as highlights and shadows, translucency and texturing. To facilitate programming and allow optimizations and portability, several graphics libraries have been written. Currently, OpenGL (SEGAL; AKELEY, 2004) is the standard free specification that has the most extensive support. Microsoft Direct3D (THORN, 2005) is a commercial competitor, though it can only be used under Microsoft Windows. OpenGL and Direct3D have a similar design pattern and offer almost the same funcionality. Our choice is for OpenGL, because it is supported in multiple platforms, because it provides better hardware abstraction and because its performance is, in practice the same as that of Direct3D (WIKIPEDIA, 2006c).

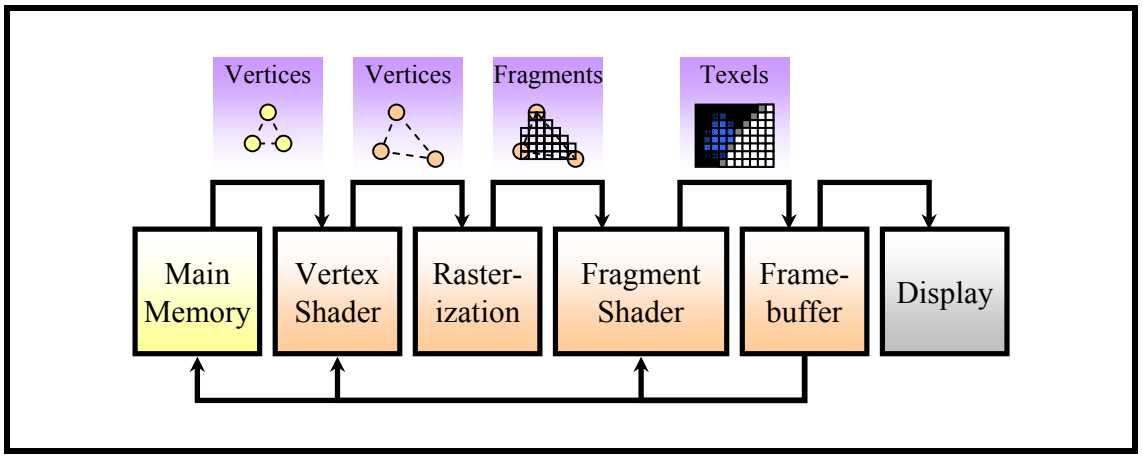


Figure 4.1: Data produced by each stage of the graphics pipeline.

4.1.1 Rendering and the Graphics Pipeline

It is important to know every component involved in the rendering process, so that we can set the rendering mode properly. A flow diagram summarizing the most used rendering process is presented in Figure 4.1 (ROST, 2005). As depicted, components of this process are:

- *Main memory*, which contains vertex information provided by an application. While the program is informing vertices (which usually are many), they are kept in memory buffers before being transferred to the graphics card.
- *Vertex shader*, in which vertices are processed by running a vertex program, producing transformed vertex coordinates.
- *Primitive assembly*, in which vertices are grouped according to primitive type;
- *Rasterization*, in which fragments (temporary pixels with associated data) are generated for a set of vertices; and
- *Fragment shader*, in which fragment colors are obtained by running a fragment program on each fragment produced by the previous step.

Once customized, the vertex program should (if desired) take care of modeling transformation, normal transformation, lighting, texture coordinate generation and coloring. Similarly, when replacing a fragment program, one should handle texture access and application, fog, etc. Shader programs can be written using *GLSL* with OpenGL (KESSENICH; BALDWIN; ROST, 2004), *Cg* with either OpenGL or DirectX (NVIDIA, 2006) or *HLSL* with DirectX (MICROSOFT, 2005). The three languages are almost identical, therefore choosing one is almost always more a matter of commodity than of prerequisites.

4.1.2 GPGPU Techniques

The programmable vertex and fragment processors available in modern graphics processing units possess important abilities that allow them to perform general computations. First, the ability to perform conditional branching and, consequently, looping. This has been well explored both for graphical (POLICARPO; NETO; COMBA, 2005) and

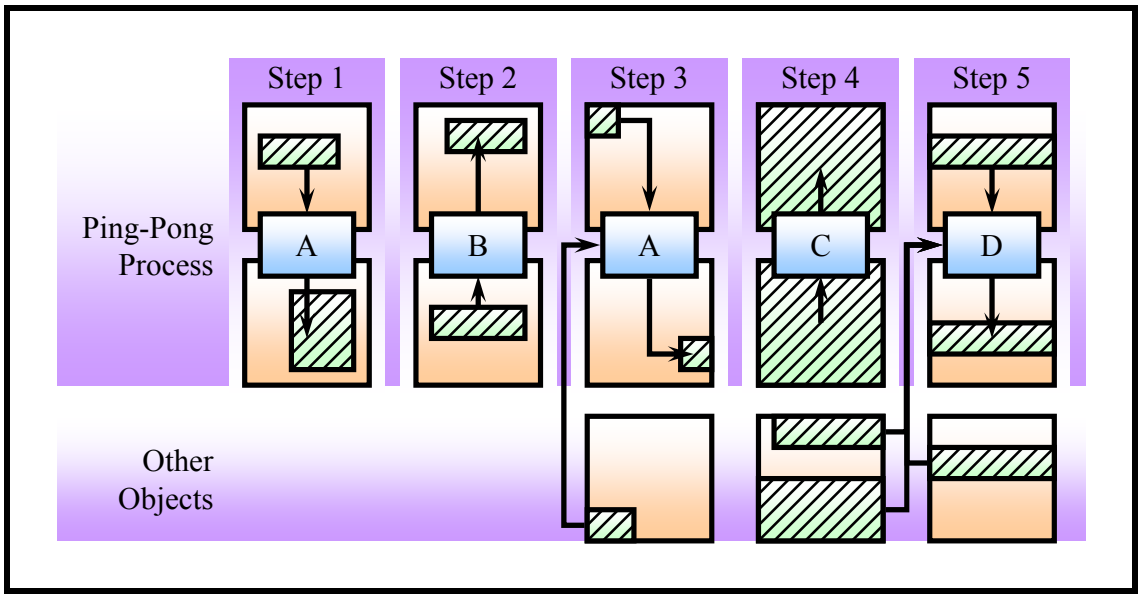


Figure 4.2: Illustration of a ping-ponging computation. A, B, C and D are shader programs. Rectangular dashed regions inside squares are source and target locations within textures used on processing. Notice the change of arrow direction, indicating the alternating reading and writing roles of each texture on each step. At each step, there is only one target location, but there may be more than one source. This is observed on steps 3, 4 and 5, where additional source textures are used by the shader.

general-purpose (GOVINDARAJU et al., 2006) applications. Second, the ability to read and write from memory locations, which allows processing data in multiple steps. And third, a set of highly efficient vector and trigonometric operations, which are suitable for many applications.

An important technique in general purpose computation on the GPU (GPGPU) is *ping-pong* rendering. Data is first uploaded as texture data. Textures are read by fragment programs and the intermediate results of the computation are then written to *another* texture. To perform rendering in multiple passes, it is possible to use just two textures and alternate source and target textures between each step. The fragment program can be changed for each polygon submitted for rendering. By structuring a process as a set of fragment programs and using ping-ponging, it is possible to perform computation on the GPU. The efficiency of such process has to be studied for each particular case. For exemplification an illustration of the process using additional textures and only rectangular primitives can be found on Figure 4.2.

To simplify programming using this process, several GPGPU-specific systems have been developed. The most widely used are *BrookGPU* (BUCK et al., 2006), *Sh* (RAPIDMIND, 2006) and *Shallows* (SHALLOWS, 2006). BrookGPU is a programming language for stream processing, while the other two are C++ libraries. None of them is an established standard, and all of them are still under development. In any case, using any of those systems is limiting to some extent and generally more inefficient than direct programming of the graphics system.

Table 4.1: Mapping audio concepts to graphic concepts.

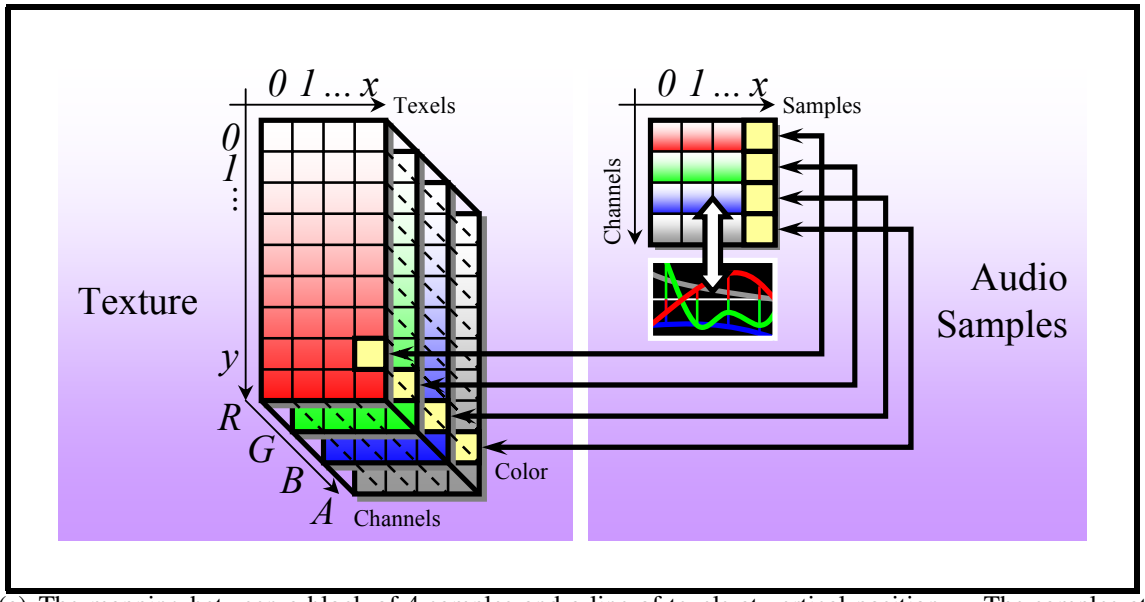
Audio Concept	Graphics Concept
Sample	Fragment
Amplitude	Color (Luminance)
Processor	Shader
Computation	Shading
Block	Texture Row
Pointers	Texture Coordinates

4.2 Using the GPU for Audio Processing

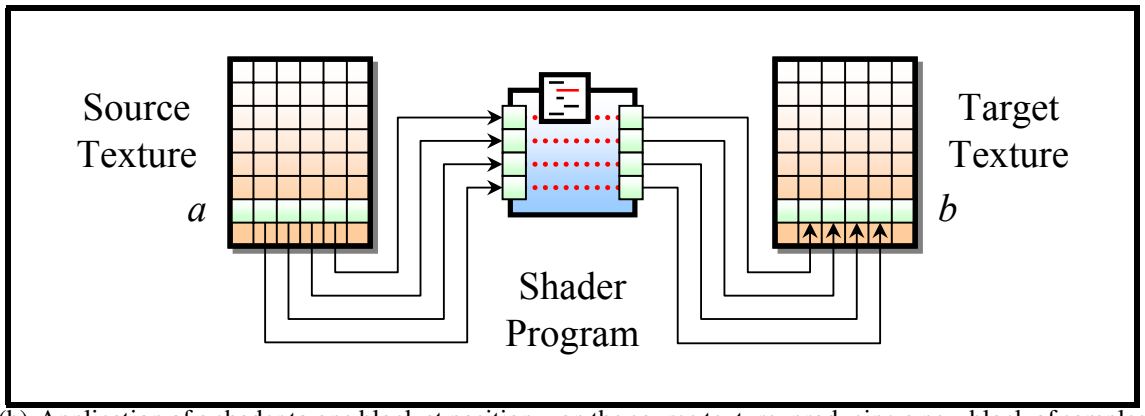
Before using the GPU, one needs to define a mapping between audio and graphics concepts. This is given explicitly on Table 4.1 and illustrated on Figure 4.3. First, *texels* and *samples* are just memory variables containing values, so samples can be transformed into texels. Second, a texel is built of several *color components* (for instance, red, green, blue and transparency levels), which actually makes a texel a small sequence of values; similarly, audio is encoded in *channels*, and one common representation of audio is by cyclically *interleaving* one sample from each channel on each position of an array. Therefore, a group of samples referring to the same instant and different audio channels can be trivially mapped into the color components of a single texel. In an audio application, blocks of n samples are processed in each step. Treating a texture as a bidimensional matrix, the contents of blocks can be stored in many different ways. One could see a bidimensional matrix as an array of same-sized arrays, which can be either lines or columns of the matrix. Then, an audio block can be mapped into one of these arrays, and, at each step, the CPU can issue the drawing of a *line primitive* providing the coordinates of the leftmost and rightmost texels of the line where the block was stored. Processing of the block, then, is performed by the fragment shader. From the rasterization stage, fragments are generated for each texel holding a sample. The parameters specified for the line primitive's vertices are interpolated according to the position of the fragment in the line and are used in the computation of the output color (a new sample value), which is stored in the corresponding position on another texture. The graphics hardware can be instructed to change the textures it reads and writes from at each step, so that other shaders can run taking as input the output of a preceding step. At last, when processing a single audio block, the computation may require many steps on the GPU.

Specifying vertices of a primitive can be viewed as *posting* a task for computation on the GPU. In the case of this work, tasks are always horizontal lines or, eventually, quadrilaterals, if more than 4 channels are needed—then, a single signal block takes more than a single line, and each line (except for the last one) holds 4 blocks of audio, one for each audio channel.

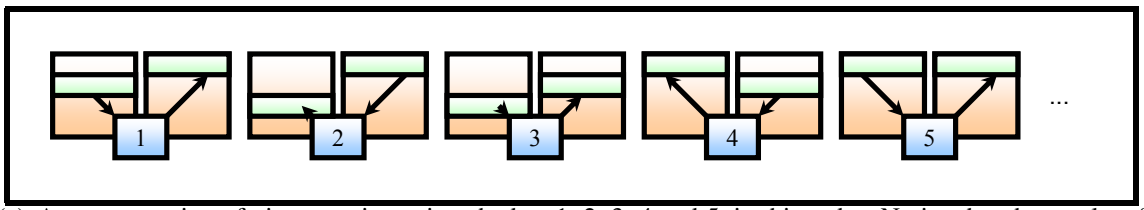
GPUs are able to perform many arithmetic operations that are useful when processing audio. Besides addition and multiplication, shading languages provide operations such as linear interpolation (which may be used for resampling or to interpolate variable input parameters), trigonometric functions (which are often used for synthesis), clamping (which may be a post-processing in cases where the sound card requires values within a specific range), step functions (which could be used to implement stepwise functions, such as envelopes), swizzling (which could be used to route signals between channels) and a random functions (which can be used to implement noise).



(a) The mapping between a block of 4 samples and a line of texels at vertical position y. The samples at position x in the block are mapped into coordinate $\langle x, y \rangle$ in the texture. Audio channels (1, 2, 3, 4) are mapped into color components (R, G, B, A, respectively).



(b) Application of a shader to one block at position a on the source texture, producing a new block of samples at position b on the target texture. In this case, the fragment processor has 4 individual pipelines (solid lines), so 4 samples, each with 4 channels, are processed simultaneously.



(c) A representation of ping-ponging using shaders 1, 2, 3, 4 and 5, in this order. Notice that the results of one step are used in the subsequent step.

Figure 4.3: Full illustration of audio processing on the GPU.

Before using the graphics hardware for effective computation of audio, the OpenGL state must be defined. For more information on this, see Appendix C.

Remark The reason for using the ping-pong technique is that hardware implementation of texture access considers them read-only or write-only but not read-write. We cannot set a texture as source and target at the same time because doing this will produce race conditions in most cases. GPUs implement separate read and write caches, and writing to a texture does not cause the read caches to be updated.

At this point, we have several tools to process signals on the GPU already. To render a sinusoidal wave, for example, one would write a shader to evaluate the sinusoidal wave, load it, set the target texture, post a line as suggested in Figure 4.3, and then read the results back to the main memory. Clearly, this data can be then passed to the audio callback procedure for playing. Therefore, the next section describes how a modular architecture can be implemented based on the GPU and how control information can be managed on a modular system.

4.3 Module System

Recall from Section 3.2.2.1 that an useful way to program audio systems is by defining a processing model graphically, interconnecting inputs and outputs of modules. Besides inputs and outputs, modules also have parameters that define their actual effect on the input data, and they also receive events, which are used to trigger the generation of notes and to change the module parameters.

Each module may have fixed or variable polyphony. In the second case, a list of playing notes can be created to represent a set of notes being played together. Every time a certain event occurs (*e.g.,* a MIDI "note on" or a keyboard "key down"), an instance of an object representing the internal state of the note is added to this list. The implementation of such object varies according to the needs of the module. For example, a simple drum machine needs only a pointer to a percussion instrument. A wavetable synthesizer, though, needs the pitch and a pointer to a wavetable. Generally, since notes may extend for more than one block, one needs to represent the elapsed time since the note started to play. Thus, elapsed time is normally a member of a note's internal state.

In our implementation, we actually replace time by a *phase offset*. Assuming that periodic waveforms have a period of T and given the periodicity of the primitive waves and of those obtained by wavetable synthesis, the time parameter can be mapped into the range $[0,\ 1)$ $-kT$, resulting in a value into the range $[0,\ 1)$. This value is called the phase offset.

In our implementation, event processing is performed at the block generation cycle inside the producer thread. How a module behaves when it receives a control event is implementation-dependent: one may prefer, for example, to implement smooth changes of a module's parameters to ensure that the wave is kept smooth along time.

The next step is to post the necessary geometric primitives to achieve the desired processing on the GPU. The order of operations is also customizable. One could devise an algorithm to position blocks of audio generated by each module in the textures along the steps of ping-ponging[1].

[1] See Section 6.1.

Listing 4.1: Sinusoidal wave shader.

```
1  float4 main(
2      float2 Params : TEXCOORD0) : COLOR
3  {
4      float t = Params.x;
5      float a = Params.y;
6      float w = a*(0.5*sin(6.28318530717958*t));
7      return float4(w, w, w, w);
8  };
```

Listing 4.2: Sinusoidal wave generator using the CPU.

```
1  void SinusoidalWave(
2      float nPhase1, float nPhase2,
3      float *pBuffer, long nSamples)
4  {
5      float nPhaseStart = 2.0 * M_PI * nPhase1;
6      float nRate = 2.0 * M_PI * (nPhase2 - nPhase1) / nSamples;
7      for (int i = 0; i < nSamples; i++)
8      {
9          float u = nPhaseStart + nIndex * nRate;
10         pBuffer[i] = sin(u);
11     }
12 }
```

4.4 Implementation

Our application was written in the C++ language using OpenGL and Cg for graphics processing and RtAudio for audio streaming. Source code was compiled with optimizations using Microsoft Visual Studio .NET 2003 (MICROSOFT, 2006b).

The following sections present the implementation of several audio processing tasks using shaders. We have written the shaders to point out clearly how the formulas we presented are applied.

4.4.1 Primitive Waveforms

Rendering primitive waveforms consists of evaluating an equation, given the parameter to the formula. Listing 4.1, Listing 4.3, Listing 4.4 and Listing 4.5 implement the rendering of primitive waveforms according to the equations presented on Table 3.2, and Listing 4.2 shows how a sinusoidal wave can be rendered using the CPU. The shaders include an amplitude parameter that is used to perform envelope shaping.

4.4.2 Mixing

A mixing shader consists simply of summing selected lines in textures. The position of these lines is determined by the module. When a signal has more than 4 channels, it is represented as a set of consecutive lines, each holding up to 4 channels. Therefore, the mixing shader needs to support this format and sum corresponding channels. Figure 4.4 illustrates this, and Listing 4.6 presents the implementation of the shader.

Listing 4.3: Sawtooth wave shader.

```
float4 main(
    float2 Params : TEXCOORD0) : COLOR
{
    float t = Params.x;
    float a = Params.y;
    float w = a * (0.5 + floor(t) - t);
    return float4(w, w, w, w);
};
```

Listing 4.4: Square wave shader.

```
float4 main(
    float2 Params : TEXCOORD0) : COLOR
{
    float t = Params.x;
    float a = Params.y;
    float w = a*(floor(t) - 0.5 - floor(t-0.5));
    return float4(w, w, w, w);
};
```

Listing 4.5: Triangle wave shader.

```
float4 main(
    float2 Params : TEXCOORD0) : COLOR
{
    float t = Params.x;
    float a = Params.y;
    float w = a*(2.0*abs(floor(t)+0.5-t)-0.5);
    return float4(w, w, w, w);
};
```

Listing 4.6: Signal mixing shader.

```
float4 main(
    float2 P : TEXCOORD0,
    uniform float Lines,
    uniform float LinesPerSignal,
    uniform samplerRECT Texture) : COLOR
{
    float4 sum = (0, 0, 0, 0);
    for (int i = 0; i < Lines; i++)
    {
        float2 T;
        T.x = P.x;
        T.y = P.y + (i * LinesPerSignal);
        sum += texRECT(Texture, T);
    }
    return sum;
};
```

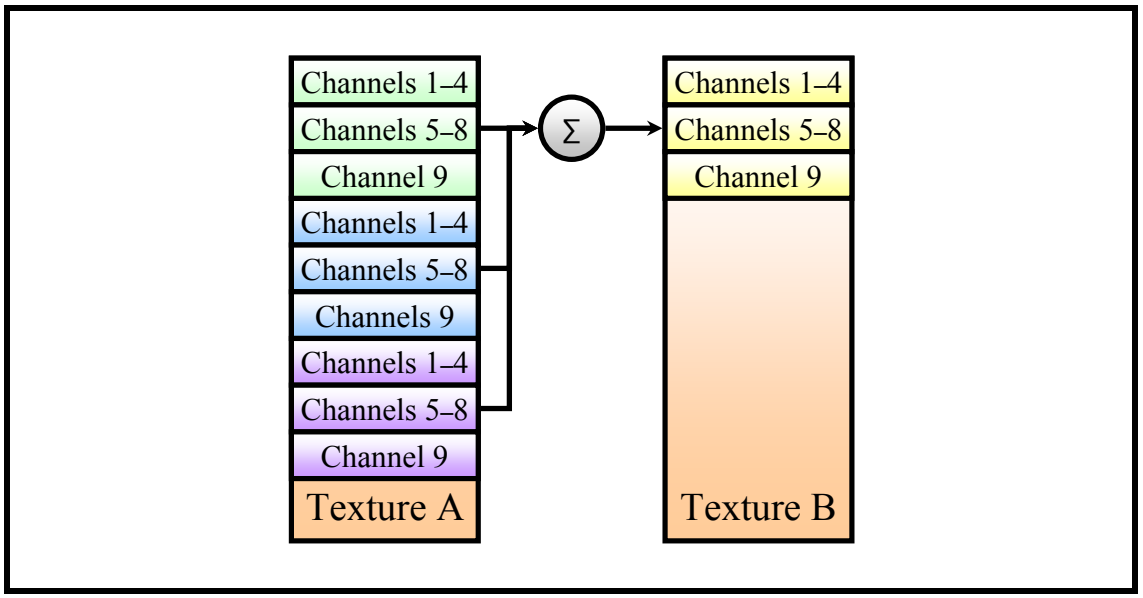


Figure 4.4: Illustration of usage of mixing shaders with 10-channel signals. Each smaller rectangle represents a line in the respective texture.

4.4.3 Wavetable Resampling

Our implementation of wavetable resampling extends the definition given on Section 3.2.3.3 by supporting a crossfade between two wavetables. *Crossfading* consists of fading from one into another. In this case, that means reducing gradually the amplitude of one wave while the amplitude of the other one is increased. The shader receives the name of the two textures holding the tables, the size of those textures, and an interpolated value for the amplitude and for the crossfading. Listing 4.7 presents the source code for the shader. Listing 4.8 shows the parameters assigned to vertices when the line is posted for rendering. Listing 4.9 presents an abstract algorithm for managing a note's state using an ADSR envelope and the wavetable shader. In this case, 5 wave tables are used and crossfaded in each of the A, D, S and R steps.

4.4.4 Echo Effect

As discussed on Section 3.2.3.4, an echo effect requires the value of the current input sample and a past output sample, offset by k samples from the current sample. This value must be accessible, so a set of past input blocks must be saved using some mechanism. For that, we can create an extra texture with the same dimensions as those of the primary textures used for ping-ponging. On that texture, blocks are stored and updated as in a circular buffer: when the pointer to the back of the buffer reaches the highest possible index, it is wrapped down to the first index instead of being incremented. Usually, echoes present more than 100 ms of delay, so we can restrict an echo's delay time to the time duration of the block size. The value of input and output samples are both scaled by constants. To simplify this operation, we create an useful shader to perform addition and multiplication by a constant in a single step: the MADD shader (Listing 4.11 and Listing 4.12). At last, a copy shader (Listing 4.10) is required because we cannot read and write from the extra texture at the same time. The operation of the echo module using MADD shaders and the copy shader is given textually on Listing 4.13 and is illustrated on Figure 4.5.

Listing 4.7: Wavetable shader with crossfading between two tables.

```
float4 main(
    float4 Params : TEXCOORD0,
    uniform float2 S,
    uniform samplerRECT Texture1,
    uniform samplerRECT Texture2) : COLOR
{
    // Table offsets
    float o1 = Params.x;
    float o2 = Params.y;

    // Interpolators
    float i = Params.z;
    float a = Params.w;

    // Linear sampling
    float s11 = texRECT(Texture1,
        float2(floor(fmod(o1, S.x)), 0.0));
    float s12 = texRECT(Texture1,
        float2(floor(fmod(o1+1, S.x)), 0.0));
    float s1 = lerp(s11, s12, frac(o1));
    float s21 = texRECT(Texture2,
        float2(floor(fmod(o2, S.y)), 0.0));
    float s22 = texRECT(Texture2,
        float2(floor(fmod(o2+1, S.y)), 0.0));
    float s2 = lerp(s21, s22, frac(o2));

    // Mixing
    float w = a * lerp(s1, s2, i);
    return float4(w, w, w, w);
};
```

Listing 4.8: Primitive posting for the wavetable shader.

```
cgGLSetTextureParameter(ParamTexture1, Texture1);
cgGLSetTextureParameter(ParamTexture2, Texture2);
cgGLEnableTextureParameter(ParamTexture1);
cgGLEnableTextureParameter(ParamTexture2);
cgGLSetParameter2f(ParamS, Texture1_Size, Texture2_Size);

glBegin(GL_LINES);
    glMultiTexCoord4f(GL_TEXTURE0,
        Offset_Start * Texture1_Size, Offset_Start * Texture2_Size,
        Interpolator_Start, Amplitude_Start);
    glVertex2i(First_Sample, Target_Row);
    glMultiTexCoord4f(GL_TEXTURE0,
        Offset_End * Texture1_Size, Offset_End * Texture2_Size,
        Interpolator_End, Amplitude_End);
    glVertex2i(Last_Sample, Target_Row);
glEnd();
```

Listing 4.9: Note state update and primitive posting.

```
do
    calculate time to next stage change
    if stage is either A, D or R
        if next stage change occurs in current block
            update state variables to next stage change
        else
            update state variables to current block's end
    else
        update state variables to current block's end

evaluate amplitude and interpolator values
evaluate current phase

post line primitive using the wavetable shader
{
    set o1 and o2 to the current sample offsets within each table
    set i to the interpolator value
    set a to the amplitude value
    set Texture1 and Texture2 to current stage's wavetables
}
update stage control variables
repeat until all block is processed
```

Listing 4.10: Copy shader.

```
float4 main(
    float2 P : TEXCOORD0,
    uniform samplerRECT Texture) : COLOR
{
    return texRECT(Texture, P);
};
```

Listing 4.11: Multiply and add shader.

```
float4 main(
    float2 PositionX : TEXCOORD0,
    float2 PositionY : TEXCOORD1,
    uniform float Alpha,
    uniform samplerRECT TextureX,
    uniform samplerRECT TextureY) : COLOR
{
    float4 X = texRECT(TextureX, PositionX);
    float4 Y = texRECT(TextureY, PositionY);
    return Alpha * X + Y;
};
```

Listing 4.12: Primitive posting for the multiply and add shader.

```
cgGLSetParameter1f(ParamAlpha, Alpha);
cgGLSetTextureParameter(ParamTextureX, TextureX);
cgGLSetTextureParameter(ParamTextureY, TextureY);
cgGLEnableTextureParameter(ParamTextureX);
cgGLEnableTextureParameter(ParamTextureY);

glBegin(GL_LINES);
    glMultiTexCoord2i(GL_TEXTURE0, 0, Row_in_TextureX);
    glMultiTexCoord2i(GL_TEXTURE1, 0, Row_in_TextureY);
    glVertex2i(0, Draw_Row);
    glMultiTexCoord2i(GL_TEXTURE0, n, Row_in_TextureX);
    glMultiTexCoord2i(GL_TEXTURE1, n, Row_in_TextureY);
    glVertex2i(n, Draw_Row);
glEnd();
```

Listing 4.13: Echo processor shader call sequence.

```
post line primitive using the multiply and add shader
{
    set a to feedback gain value
    set TextureX to Texture 4
    set PositionX to the delay line's read position
    set TextureY to Texture 1
    set PositionY to row 0
    render to Texture 3
}
post line primitive using the multiply and add shader
{
    set a to wet mix gain value
    set TextureX to Texture 4
    set PositionX to the delay line's read position
    set TextureY to Texture 1
    set PositionY to row 0
    render to Texture 2
}
copy generated block from texture 3 to texture 4 at the delay line's write position
update delay line read and write positions
```

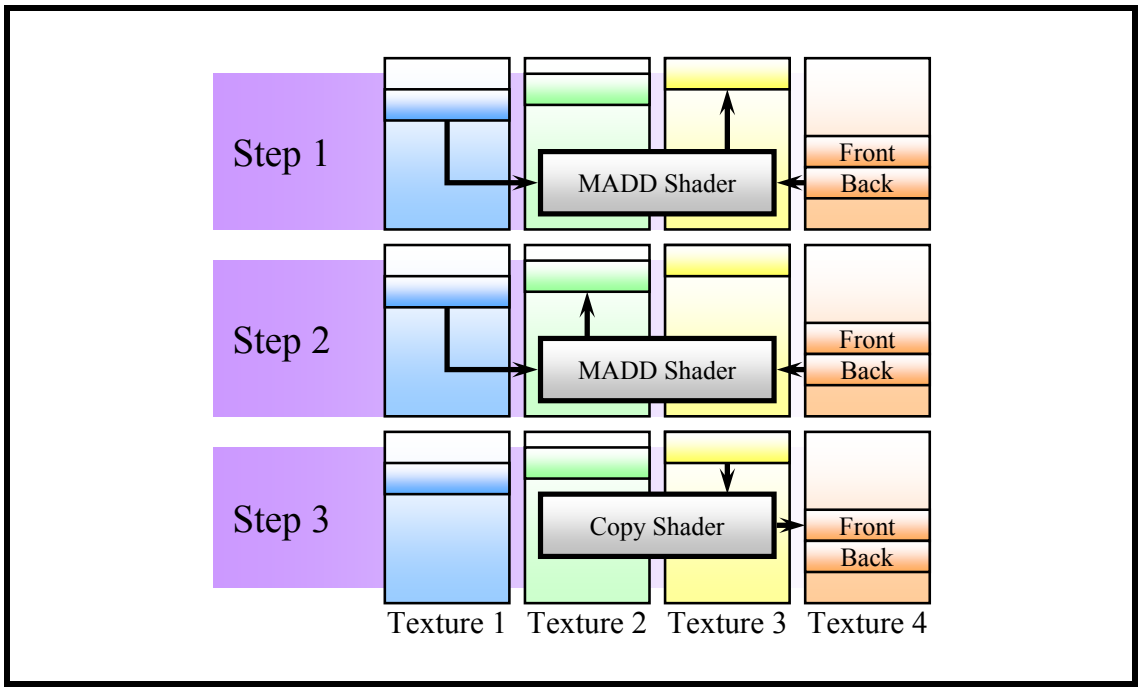


Figure 4.5: Illustration of steps to compute an echo effect on the GPU using MADD shaders and one copy operation. Notice that texture 3 is used to store a temporary result that ends up copied into texture 4 at the "front" position of a circular buffer. The actual output of the module is written on texture 2.

4.4.5 Filters

The implementation of a FIR filter follows the same idea of the past shaders: directly translate the formula into a shader and use it by rendering a line primitive. For filters, several samples of the past input block will be necessary; they can be saved on the auxiliary texture following the idea presented for the echo effect.

An IIR filter presents an implementation problem: it requires the value of output samples that are being generated in the same step. As a result, the part of Equation (3.8) using output values cannot be implemented trivially. IIR filters are still implementable, but the speed up obtained by a GPU implementation will not be as significant as the remaining shaders. For this reason, we have not implemented IIR filters.

4.5 Summary

This chapter described the structure of an audio application. We discussed how audio is passed on to the audio device and the reasons to implement an intermediary buffer between the application and the audio interface. We presented details about graphics rendering and how graphics APIs such as OpenGL can be used to instruct the GPU about how to process audio data. At last, we discussed the implementation of modular systems and presented shaders to perform audio processing on the graphics device.

In the following chapter, the performance of this system is measured. We discuss the quality of the generated signal and the limitations imposed by the GPU for audio processing.

5 RESULTS

In this chapter, we discuss the results obtained with the implementation described on the last chapter. In the first section, we present a comparison of total polyphony achieved with both a CPU and a GPU version of algorithms for primitive waveform generation. The next section evaluates the quality of audio data produced by the GPU. Then, we present the limitations posed by the graphics hardware on audio processing.

5.1 Performance Measurements

The following measurements were obtained by running the application on an AMD Athlon64 3000+ (1.81 GHz) with 1 GB of memory with an ASUS A8N-SLI Deluxe motherboard, a PCI Express nVidia GeForce 6600 with 256 MB, and a Cretive Audigy Platinum eX sound card. These devices allowed a texture size of 4096 texels in each dimension and audio blocks of 96 to 2400 samples. Although we have used a 64-bit processor, we were running the application on Windows XP, which does not support 64-bit instructions and, therefore, the system was running in 32-bit emulation mode.

We used blocks of 256 samples and set the ASIO audio device to play at a sampling rate of 48 kHz using 2 channels. At this rate, 256 samples represents a little more than 5 ms. The intermediary buffer between the callback and the producer thread was able to hold 2 blocks. Holding the block on the buffer causes a total delay of about 11 ms, which is acceptable for real-time output.

To compare the performance of the GPU with similar CPU implementations, we rendered the basic waveforms many times on subsequent rows of a texture. We wrote similar rendering algorithms on the CPU. For example, we present the sine wave generator on Listing 4.2 with code that runs on the CPU. Implementations for other shaders have been developed similarly. One may argue that there are more efficient implementations of a sine wave oscillator, but we note that other methods have drawbacks (sometimes acceptable), such as introducing noise (by sampling a wavetable) or becoming innacurate along time (by using only addition and multiplication to resolve a differential equation). Additionally, the implementation by directly evaluating a formula constitutes a more precise overall performance comparison, since both machines would be performing essentially the same computation.

We approximated the maximum number of times the rendering programs for the GPU and CPU could be invoked in real time without causing audio glitches. To do that, we gradually increased the number of line primitives being posted and programmed the code in the callback to update a counter whenever no block was present for output. By doing this rather than simply calculating the time that the GPU takes to process a certain amount of lines, one gets results that express how the application behaves on practical usage by

Table 5.1: Performance comparison on rendering primitive waveforms. The columns CPU and GPU indicate the number of times a primitive was rendered on a block in one second.

Waveform	CPU	GPU	Speedup
triangle	210	5,896	28×
sinusoid	400	11,892	30×
sawtooth	220	7,896	36×
square	100	5,996	59×

taking in account the overhead introduced by thread synchronization.

Table 5.1 presents the performance test results. Note that the results are significantly better than those reported by other studies (WHALEN, 2005; GALLO; TSINGOS, 2004), even though we were using a GeForce 6600, a medium-range GPU with only 8 fragment processors. Higher speedups should be obtained with the use of faster GPUs, such as the nVidia GeForce 7900 GTX (with 24 fragment processors) and the ATI Radeon X1900 XTX (with 48 fragment processors).

We have not designed comparison tests for other kinds of audio effects, such as feedback delay. We expect, though, that other effects will achieve similar speedups, since they are often based on simple shaders and execute entirely on the GPU.

5.2 Quality Evaluation

Sound is traditionally processed using 16-bit or 24-bit integer. On the GPU, sound is represented as 32-bit floating point values. This presents more accuracy than 16-bit integer values, presenting an increased SQNR level. A representation in floating-point also presents the advantage to remain precise regardless of the dynamic level of the sound, *i.e.,* its amplitude. Integer values suffer from the problem that very silent sounds are represented using fewer amplitude levels, which greatly increases the quantization noise. However, the 32-bit floating point implementation on the GPU includes a mantissa of 23-bits (not including the hidden bit), ensuring that 32-bit floating points have at least as much accuracy as 24-bit integers.

However, the floating-point implementation on the GPU is not exactly the same as that on the CPU. It has been shown that the rounding error is about two times greater (HILLESLAND; LASTRA, 2004) than that of operations on the GPU—except for the division, where the rouding error is almost four times greater. One would obtain similar results by removing 2 bits from the mantissa, for example. But even with a 22-bit precision, the SQNR level is much higher than that with the traditional 16-bit integer representation, which is sufficient for most needs.

The methods we implemented are musically questionable, though. With the exception of the sinusoidal wave shader, the other primitives are known to cause aliasing if rendered using formulas on Table 3.2. A better implementation could be obtained by evaluating a limited number of terms from formulas on Table 3.3 (which is actually a case of additive synthesis) or by using other methods such as proposed by STILSON; SMITH (1996).

5.3 Limitations

Several processes that do not present local data independence do not map well to the GPU. In our work, the most prominent example of that is the IIR filter. To implement an IIR filter efficiently, one would need to access the values computed for fragments to the left of the current fragment. Given that a texture cannot be read and written in the same step, this would require evaluating fragment by fragment and exchanging source and target textures between the evaluation of every fragment. Naturally, this represents a bottleneck, since one needs to submit many geometric primitives and force cache updates for each fragment. Even with the latest GPUs, the CPU implementation should probably present better performance. This problem is still under study, but it seems there are at least a few special cases for which a performance gain can be achieved. Note that this problem would be solved if the ability to access the values computed for adjacent fragments in a single step were available at the graphics hardware.

5.4 Summary

This chapter presented the results we obtained by inspecting the application. We have shown that a GPU-based implementation of several algorithms may present a speedup of $59\times$. We have explained that the quality of signals generated on the GPU is adequate and very closely related to professional 24-bit quality, except that the usage of floating-point introduces less quantization error on low-amplitude samples. The problem of data dependency, which directly affects the implementation of IIR filters, was also discussed.

6 FINAL REMARKS

This work has presented the implementation of an audio processing system using the GPU that produces results in real-time, achieves higher performance and is easily extensible. The fundamental concepts of sound processing were presented, prior to a discussion of the structure of an audio processing system and the implication of every design choice. We have presented how to use fragment shaders to perform computation of audio samples, and we also designed performance benchmarks for comparison and discussed the current limitations of the system. This work ultimately demonstrates that the GPU can be used for efficient audio processing, achieving speedups of almost $60\times$ in some cases, enabling users to take advantage of more computational power with the flexibility of software implementations at costs lower than professional audio equipments.

6.1 Future Work

This system is a prototype including several modules for demonstration. For professional music, many other modules need to be implemented, including an IIR filter, which, as discussed, could not be mapped efficiently using a direct approach.

We also presented the concept of modular processing, yet we have not developed a complete module system that would allow user customization of a processing model. To achieve this, one needs to implement an auxiliary system to distribute the usage of texture lines in each step of ping-ponging to each of the modules, as necessary according to the processing model. This is a required abstraction for efficient development, since the actual implementation of a module does not depend on which line of a texture it is reading from, but requires that this line matches the line where the results of another module were written. This auxiliary system would also be required to insert implicit operations, such as mixing two output signals that are connected to the input of a module before performing that module's computation. We have planned an algorithm to do that, but we do not present it here due to lack of time for testing.

An interesting addition would be Sony DSD, a recent sound format commonly used in processing high quality audio. This format uses delta-sigma modulation to represent the signal, and there are algorithms for several audio processes already (REISS; SANDLER, 2004). Whether this format can be efficiently processed by the GPU is the most important question. If it can, implementing DSD support could be useful, since the amount of data to be processed is relatively close to that necessary to process PCM audio[1]. Another

[1] A common DSD format is 1-bit samples at 2822,4 kHz ($64 * 44.1$ kHz), which results in approximately 2 to 4 times as much as the usual CD PCM format. A proposed alternative format would be to represent each 1 bit sample by 1 byte words, taking 64 times more storage space. Advantages of each alternative remain to be studied.

interesting addition would be the support to time-varying frequency-domain signals, in order to evaluate the efficiency of the GPU for this kind of computation. This would introduce more implicit operations to be handled.

A commercial solution would also require the addition of a graphics interface, a sequencer and the inclusion of sample banks that can be used by the musician to create music.

REFERENCES

ANSARI, M. Y. **Video Image Processing Using Shaders**. Game Developers Conference, Presentation.

DIRECTOR, S. W. (Ed.). **Digital Filters**: analysis and design. TMH ed. New Delhi, India: Tata McGraw-Hill, 1980.

AUDIO DSP TEAM. **Resampling**. Available at: <http://www.dspteam.com/resample.html>. Visited in: June 2006.

BIONICFX. 2005. Available at: <http://www.bionicfx.com>. Visited in: June 2006.

BRISTOW-JOHNSON, R. Wavetable Synthesis 101, A Fundamental Perspective. In: AES CONVENTION, 101., 1996, Los Angeles, California. **Proceedings...** [S.l.: s.n.], 1996. Available at: <http://www.musicdsp.org/files/Wavetable-101.pdf>.

BUCK, I.; FOLEY, T.; HORN, D.; SUGERMAN, J.; FATAHALIAN, K.; HOUSTON, M.; HANRAHAN, P. Brook for GPUs: stream computing on graphics hardware. **ACM Transactions on Graphics**, [S.l.], v.23, n.3, p.777–786, August 2004. Available at: <http://graphics.stanford.edu/papers/brookgpu/brookgpu.pdf>. Visited in: June 2006.

BUCK, I.; FOLEY, T.; HORN, D.; SUGERMAN, J.; HANRAHAN, P.; HOUSTON, M.; FATAHALIAN, K. **BrookGPU**. 2006. Available at: <http://graphics.stanford.edu/projects/brookgpu>. Visited in: June 2006.

GALLO, E.; TSINGOS, N. Efficient 3D Audio Processing with the GPU. **Proceedings of GP2: Workshop on General Purpose Computing on Graphics Processors**, Los Angeles, California, August 2004. Poster. Available at: <http://www-sop.inria.fr/reves/Nicolas.Tsingos/publis/posterfinal.pdf>. Visited in: June 2006.

GOVINDARAJU, N. K.; GRAY, J.; KUMAR, R.; MANOCHA, D. GPUTeraSort: high performance graphics coprocessor sorting for large database management. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2003., 2006, Chicago, IL. **Proceedings...** [S.l.: s.n.], 2006. Available at: <http://gamma.cs.unc.edu/GPUTERASORT/gputerasort_sigmod06.pdf>.

GPUFFTW Library. Sponsored by Army Modeling and Simulation Office, Army Research Office, Defense Advanced Research Projects Agency, RDECOM and NVIDIA Corporation. Available at: <http://gamma.cs.unc.edu/GPUFFTW>.

GUMMER, A. W. (Ed.). **Biophysics of the Cochlea**. Titisee, Germany: [s.n.], 2002. Proceedings of the International Symposium.

OPPENHEIM, A. V. (Ed.). **Digital Filters**. 3rd ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1989. (Signal Processing Series).

HILLESLAND, K. E.; LASTRA, A. GPU Floating-Point Paranoia. In: GP2: WORKSHOP ON GENERAL PURPOSE COMPUTING ON GRAPHICS PROCESSORS, 2004, Chapel Hill, NC, USA. **Proceedings...** ACM SIGGRAPH, 2004. Available at: <http://www.cs.unc.edu/~ibr/projects/paranoia/gpu_paranoia.pdf>. Visited in: June 2006.

ISHIKAWA, O.; AONO, Y.; KATAYOSE, H.; INOKUCHI, S. Extraction of Musical Performance Rules Using a Modified Algorithm of Multiple Regression Analysis. In: INTERNATIONAL COMPUTER MUSIC CONFERENCE, 2000, Berlin, Germany, August 2000. **Proceedings...** [S.l.: s.n.], 2000. p.348–351. Available at: <http://www-inolab.sys.es.osaka-u.ac.jp/users/ishikawa/kenkyu/study/ICMC2000.pdf>. Visited in: June 2006.

JĘDRZEJEWSKI, M.; MARASEK, K. Computation of Room Acoustics Using Programmable Video Hardware. In: INTERNATIONAL CONFERENCE ON COMPUTER VISION AND GRAPHICS, 2004, Warsaw, Poland. **Anais...** [S.l.: s.n.], 2004. Available at: <http://www.pjwstk.edu.pl/~s1525/prg/mgr/gpuarticle.pdf>. Visited in: June 2006.

JULIANO, J.; SANDMEL, J.; AKELEY, K.; ALLEN, J.; BERETTA, B.; BROWN, P.; CRAIGHEAD, M.; EDDY, A.; EVERITT, C.; GALVAN, M.; GOLD, M.; HART, E.; KILGARD, M.; KIRKLAND, D.; LEECH, J.; LICEA-KANE, B.; LICHTENBELT, B.; LIN, K.; MACE, R.; MORRISON, T.; NIEDERAUER, C.; PAUL, B.; PUEY, P.; ROMANICK, I.; ROSASCO, J.; SAMS, R. J.; SEGAL, M.; SEETHARAMAIAH, A.; SCHAMEL, F.; VOGEL, D.; WERNESS, E.; WOOLLEY, C. **Framebuffer Object**. Revision 118. Available at: <http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt>. Visited in: June 2006.

KESSENICH, J.; BALDWIN, D.; ROST, R. **The OpenGL Shading Language**. 1.10, Revision 59 ed. Madison, AL, USA: 3Dlabs, Inc. Ltd., 2004. Available at: <http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf>. Visited in: June 2006.

KURO5HIN. **The End of Moore's Law**. Available at: <http://www.kuro5hin.org/story/2005/4/19/202244/053>. Visited in: June 2006.

MICROSOFT CORPORATION. **HLSL Shaders**. 2005. Available at: <http://msdn.microsoft.com/library/en-us/directx9_c/Writing_HLSL_Shaders.asp>. Visited in: June 2006.

MICROSOFT CORPORATION. **DirectSound C/C++ Reference**. Available at: <http://msdn.microsoft.com/library/en-us/directx9_c/dx9_directsound_reference.asp>. Visited in: June 2006.

MICROSOFT CORPORATION. **Visual Studio .NET 2003**. Available at: <http://msdn.microsoft.com/vstudio/previous/2003>. Visited in: May 2006.

ROTTINO, M. P. (Ed.). **Elements of Computer Music**. Upper Saddle River, NJ, USA: PTR Prentice Hall, 1990.

MOORE, G. E. Cramming More Components onto Integrated Circuits. **Electronics**, [S.l.], v.38, n.8, p.114–117, April 1965. Available at: <ftp://download.intel.com/research/silicon/moorespaper.pdf>.

MORELAND, K.; ANGEL, E. The FFT on a GPU. In: SIGGRAPH/EUROGRAPHICS WORKSHOP ON GRAPHICS HARDWARE, 2003, San Diego, CA. **Anais...** Eurographics Association, 2003. p.112–119. Available at: <http://www.eg.org/EG/DL/WS/EGGH03/112-119-moreland.pdf>. Visited in: June 2006.

NVIDIA CORPORATION. **Cg Toolkit User's Manual**. 1.4.1 ed. Santa Clara, CA, USA: NVIDIA Corporation, 2006. Available at: <http://download.nvidia.com/developer/cg/Cg_1.4/1.4.1/Cg-1.4.1_UsersManual.pdf>. Visited in: June 2006.

POLICARPO, F.; NETO, M. M. D. O.; COMBA, J. L. D. Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. In: SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, 2005., 2005, New York, NY, USA. **Proceedings...** ACM Press, 2005. v.24, n.3, p.155–162. Available at: <http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/RTM.pdf>. Visited in: June 2006.

RAPIDMIND. **Sh**: a high-level metaprogramming language for modern GPUs. Available at: <http://www.libsh.org>. Visited in: June 2006.

REISS, J.; SANDLER, M. Digital Audio Effects Applied Directly on a DSD Bitstream. In: INTERNATIONAL CONFERENCE ON DIGITAL AUDIO EFFECTS, 7., 2004, Naples, Italy. **Proceedings...** [S.l.: s.n.], 2004. p.1–6. Available at: <http://dafx04.na.infn.it/WebProc/Proc/P_372.pdf>. Visited in: May 2006.

ROST, R. **Introduction to the OpenGL Shading Language**. OpenGL MasterClass, Presentation. Available at: <http://developer.3dlabs.com/documents/presentations/GLSLOverview2005.zip>. Visited in: June 2006.

SCAVONE, G. P. **The RtAudio Tutorial**. 2005. Available at: <http://www.music.mcgill.ca/~gary/rtaudio>. Visited in: June 2006.

SEGAL, M.; AKELEY, K. **The OpenGL Graphics System**: A specification. 2.0 ed. Mountain View, CA, USA: Silicon Graphics, Inc., 2004. Available at: <http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf>. Visited in: June 2006.

SHALLOWS. **Making GPGPU Programming Fast and Easy**. Available at: <http://shallows.sourceforge.net>. Visited in: June 2006.

SPITZER, J. **Implementing a GPU-Efficient FFT**. SIGGRAPH, Presentation. Available at: <http://gamma.cs.unc.edu/SIG03_COURSE/2003ImplementingFFTonGPU.pdf>. Visited in: June 2006.

STEINBERG MEDIA TECHNOLOGIES GMBH. **3rd Party Developers Page**. Available at: <http://www.steinberg.net/324_1_.html>. Visited in: May 2006.

STILSON, T.; SMITH, J. Alias-Free Digital Synthesis of Classic Analog Waveforms. In: INTERNATIONAL COMPUTER MUSIC CONFERENCE, 1996, Hong Kong, China. **Proceedings...** [S.l.: s.n.], 1996. Available at: <http://ccrma.stanford.edu/~stilti/papers/blit.pdf>. Visited in: June 2006.

SUMANAWEERA, T.; LIU, D. Medical Image Reconstruction with the FFT. In: PHARR, M. (Ed.). **GPU Gems 2**. [S.l.]: Addison Wesley, 2005. p.765–784. Available at: <http://download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/OpenGL/src/gpgpu_fft/docs/Gems2_ch48_SDK.pdf>. Visited in: June 2006.

THORN, A. **DirectX 9 Graphics**: the definitive guide to Direct 3D. USA: Wordware Publishing, Inc., 2005. (Worldware Applications Library).

TOLONEN, T.; VÄLIMÄKI, V.; KARJALAINEN, M. **Evaluation of Modern Sound Synthesis Methods**. [S.l.]: Helsinki University of Technology, Department of Electrical and Communications Engineering, Laboratory of Acoustics and Audio Signal Processing, 1998. Report. (48). Available at: <http://www.acoustics.hut.fi/publications/reports/sound_synth_report.pdf>. Visited in: June 2006.

TURKOWSKI, K. **Filters for Common Resampling Tasks**. [S.l.]: Apple Computer, 1990. Available at: <http://www.worldserver.com/turk/computergraphics/ResamplingFilters.pdf>.

WHALEN, S. Audio and the Graphics Processing Unit. **IEEE Visualization 2004 GPGPU Course**, [S.l.], March 2005. Tutorial. Available at: <http://www.node99.org/projects/gpuaudio/gpuaudio.pdf>. Visited in: June 2006.

WIKIPEDIA, THE FREE ENCYCLOPEDIA. **Moore's law**. Available at: <http://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=58194474>. Visited in: June 2006.

WIKIPEDIA, THE FREE ENCYCLOPEDIA. **Definition of music**. Available at: <http://en.wikipedia.org/wiki/Definition_of_music>. Visited in: June 2006.

WIKIPEDIA, THE FREE ENCYCLOPEDIA. **Comparison of Direct3D and OpenGL**. Available at: <http://en.wikipedia.org/wiki/Direct3D_vs._OpenGL>. Visited in: June 2006.

ZHIRNOV, V. V.; III, R. K. C.; HUTCHBY, J. A.; BOURIANOFF, G. I. Limits to Binary Logic Switch Scaling—A Gedanken Model. In: IEEE, 2003, Research Triangle Park, NC, USA 27709-2053. **Proceedings...** [S.l.: s.n.], 2003. v.91, n.11, p.1934–1939. Available at: <http://www.intel.com/research/documents/Bourianoff-Proc-IEEE-Limits.pdf>.

APPENDIX A COMMERCIAL AUDIO SYSTEMS

There is a vast set of commercial music production systems. There are also many kinds of products to perform all sorts of sound processing tasks. The multiplicity of products may imply that there are no universally "correct" and well-known guidelines for building sound modules in general. A detail in the design of an audio system can enable musical possibilities that another system cannot achieve. For that reason, one may need to purchase many products in order to satisfy all music production needs.

Here, I mention some of the most popular products. I cite many featured music workstations and software audio and some of the most relevant of their features. Note that it is impossible to cover the full range of music equipment, and that I have chosen products that seemed compatible with this work's goals. For example, I do not mention drum machines or studio equipment such as amplifiers.

A.1 Audio Equipment

Popular audio equipment companies include:

- Alesis

  The company produces many kinds of audio equipment and also software synthesizers. Its top product is *Fusion 8HD*, an 88-key keyboard synthesizer supporting sample playback (272 voices), FM synthesis (240 voices), analog synthesis simulations (140 voices) and physical modelling (48 or 60 voices), along with reverbs, choruses, equalizers and others sound effects. The company also produces effects modules such as reverbs and equalizers and many recording accessories.

- Creative Technology Limited

  Creative is the most important company producing sound cards for desktop computer, though the company works with many multimedia-related product types. The current top of line sound card is *Sound Blaster X-Fi Elite Pro*, which can record at 96 kHz and playback at 96 kHz (for surround sound) or 192 kHz for stereo, presenting very little distortion (SNR around 116 dB and THD around 0.0008%). The X-Fi card processes audio using an *EMU20K1* processor, which is able to achieve 10,000 MIPS, supporting up to 128 3D voices with 4 effects applied to each. The chip is also known to be programmable, though information necessary for implementation is not provided by manufacturers.

- Korg

  Korg is famous by its music workstations, but they also manufacture many kinds of audio accessories and equipment. The main product of the company is *Triton*

Extreme, a 88-key keyboard supporting a polyphony of 120 voices produced using Korg's Hyper Integrated synthesis engine, a complex synthesis scheme that includes filtering and envelope shaping to the output of each oscillator. Triton Extreme also comes with 102 effect types running up to 8 simultaneously, 2,072 instrument definitions (programs and combinations) and a set of sequencer patterns preloaded which can be used in live performances.

- Kurzweil Music Systems

  The company works with 3 kinds of audio equipment: keyboards, digital pianos and signal processors. Top products in these categories are:

  - *PC2X*, an 88-key keyboard supporting up to 128 voices, 660 instrument definitions and 163 different effects running on two processors; and
  - *Mark 152i*, a digital piano that mimics a real piano by placing seven speakers inside a wooden piano-shaped cabinet and using surround techniques to add realism to the sound. It supports a 32 voice polyphony and includes 325 instrument definitions and 880 effects.

- Moog Music Inc.

  The firm is mostly know for the production of *Minimoog Voyager*, an extension of the original and popular Model D Minimoog. Voyager is a completely analog synthesizer, using digital circuitry only to store instrument definitions and quickly restore the instrument's parameters.

- New England Digital Corp.

  Though the company was closed early in the 1990's, its most famous product, the *Synclavier*, released late in the 1970's, is still used in the recording industry in the sound design process for major movies and in music composition and performance. The Synclavier supports a sampling rate of 100 kHz and includes facilities for FM synthesis, digital sampling, hard-disk recording and sophisticated digital sound editing. It is generally regarded for its depth of sound, versatility at sound creation and production, and speed of use. In 1998, employees of the company developed an emulator to run Synclavier software on Apple Macintosh systems.

- Peavey Electronics

  Peavey produces a wide range of audio electronic equipment, but almost none involving synthesis and effects processing. They provide subharmonic and stereo enhancement, compressor/expander/limiter, crossover and equalizer modules of varying feature level.

- Roland Corporation

  Roland's top workstation is *Fantom-X8*, an 88-key keyboard with a polyphony of 128 voices, including 1,408 predefined instruments and 78 types of effects. The product, though, works only with 16-bit samples ate 44 kHz. Roland also provides drum machines and digital piano simulations, among other audio equipment.

- Yamaha Corporation

  The company is famous for its audio products, providing a wide range of audio accessories. Top workstations are:

  - *MO6 / MO8*, an 88-key keyboard 64 voices of polyphony, 705 instrument definitions (plus memory for additional ones) and 222 effects; and
  - *Tyros2*, a home-oriented 61-key keyboard with a wide collection of instruments and effects, together with accompaniment functionality.

Yamaha also produces several kinds of digital pianos, also called “clavinovas”.

A.2 Software Solutions

Popular companies working with audio software solutions include:

- Arturia

  This company develops and markets the *Moog Modular V*, a software emulator of the Moog modular synthesizer, the *CS-80 V*, an emulation of Yamaha CS-80, the *Arp 2600 V*, and emulation of the ARP 2600 synthesizer, among others.

- Cycling’74

  This company is known mainly for maintaining *Max*, a highly modular graphical development environment for music and multimedia. A well-known open-source successor to Max is Miller Puckette’s *Pure Data*, which is very similar in design and functionality.

- Digidesign

  This firm is very well know by its flagship software *Pro Tools*, known as the *de facto* industry standard professional recording application.

- Emagic

  This company is know for *Logic Pro*, a an application for the Mac OS X platform that provides software instruments, synthesizers, audio effects and recording tools for music synthesis.

- Image-Line Software

  The company is the producer of *FL Studio*, a pattern-based music sequencer widely used for electronic music.

- Mark of the Unicorn (MOTU)

  MOTU produces *Digital Performer* for Mac OS X, a recording and editing software designed for composers of large works such as film scores. The latest version bundles with a sampler, software synthesizers a drum module. It includes productivity enhancements and special features for film scoring.

- Native Instruments

  The firm sells a wide collection of software synthesizers and audio mastering software. Among them, the most relevant ones are:

  - *Absynth*, a software synthesizer combining subtractive synthesis with granular and sample-based techniques, which can be used as stand-alone or as a plug-in;

  - *FM7*, an Yamaha DX7 software simulator; and
  - *Reaktor*, a graphical modular software music studio.

- Novation Electronic Music Systems

  This company develops virtual analog synthesizers, digital simulators of analog synthesizers. Its main product is *V-Station*, a VSTi module that uses simulations of analog primitive waveforms combined with filters, effects and other functionality.

- Propellerhead Software

  The company is a competitor of other studio software and is known for defining the ReWire protocol. The most significant applications produced by the company are:

  - *Reason*, a recording studio package that emulates a rack of hardware modules;
  - *ReBirth*, an emulator of the classic Roland modules TB-303, TR-808 and TR-909; and
  - *ReCycle*, a sound loop editor.

- Steinberg Media Technologies GmbH

  Steinberg is a german firm that produces audio applications and plug-ins. The firm is also famous for the creation of ASIO and VST standards. Their main products are:

  - *Cubase*, a music sequencer with MIDI support and audio editing capabilities; and
  - *Nuendo*, similar to Cubase, but offering more functionality for video production and audio mastering and being significantly more expensive.

A.3 Plug-in Architectures

Many software solutions provide extension support to new modules through plug-ins. The most widespread plug-in interfaces are:

- Virtual Studio Technology (VST)

  Created by Steinberg, VST is a cross-platform protocol in which modules are classified as instruments (VSTi) or effects processors, and are implemented as a function callback, whose responsibility is to fill a fixed-size array with sample values.

- ReWire

  Jointly defined by Propellerhead and Steinberg, ReWire is a protocol that allows remote control and data transfer among audio software. The protocol allows transferring up to 256 audio tracks at any sample rate and 4080 MIDI data channels. Such flexibility has allowed it to become an industry standard.

- DirectSound

  Microsoft's DirectSound is part of DirectX, which runs only on Windows. DirectSound processor modules are also called filters, and they are primarily effects processors only.

- DSSI Soft Synth Instrument

  Based on LADSPA, DSSI runs on Linux and is comparable to the VSTi specification. It can also wrap VSTi modules for Linux hosts.

Finally, there have been numerous sound programming languages, such as *Csound*, *Cmusic*, *Nyquist*, *SuperCollider* and *ChucK*. Programming languages, though, are unusable by the musician, who is generally not literate on computer science.

APPENDIX B REPORT ON ASIO ISSUES

This appendix briefly discusses problems faced when programming with the ASIO interface. Its primary intention is to document the problems we have encountered during development. Hopefully, it will help other researchers to avoid some of the difficulties I faced. I will focus on the Windows implementation of ASIO, since that is the platform used for developing this project. In the following sections, ASIO is characterized from a developer's point of view. A serious flaw is described, followed by a discussion of bypassing attempts.

B.1 An Overview of ASIO

ASIO[1] is a widely used protocol for real-time (low-latency) audio streaming between a central computer system (*e.g.,* CPU and memory) and an audio device (*e.g.,* a sound card). It was defined by Steinberg and implemented by sound card manufacturers. On Windows, competitor interfaces with similar functionality are GSIF (which is mostly used by old applications) and Kernel Streaming (a low-level, counterintuitive streaming system). On other systems, popular interfaces include Core Audio on Mac OS X and ALSA on Linux. Several free and open audio libraries use ASIO. For instance, widely used libraries include:

- PortAudio, which is cross-platform and supports other audio interfaces as well;
- RtAudio, similar to PortAudio; and
- JUCE, which is a general library supporting many other features besides audio.

An application using ASIO is called an ASIO host, while an implementation of ASIO is called an ASIO driver. ASIO is multi-platform and supports a wide range of formats, including the recent Sony DSD format. ASIO is, therefore, a very general means of interaction with modern sound cards and other audio devices, making it suitable for professional audio applications.

Figure B.1 presents an abstract view of ASIO's operation. The four boxes identify the four ASIO states. Arrows indicate state changes caused by API calls. The *load* and *remove* calls are depicted in gray to indicate that they are not formally described in the documentation.

An application can use an ASIO-enabled sound card using the ASIO SDK provided by Steinberg. I have worked with SDK version 2.1, obtained by request at http://www.steinberg.net/324_1.html.

[1] One should not confuse Steinberg's ASIO protocol with the Boost Library's asio C++ library, a library for network programming.

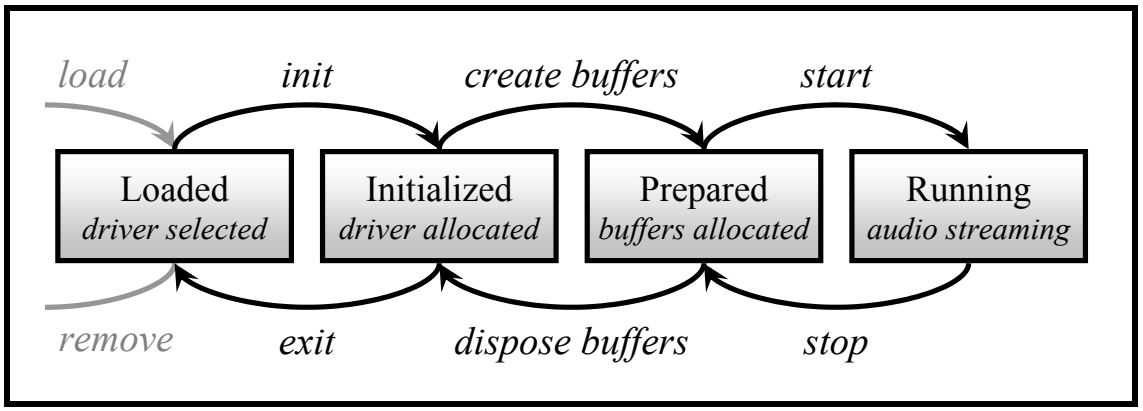


Figure B.1: ASIO operation summary.

An initial analysis of the SDK's contents clearly points out that:

- There is no guide on how to build an ASIO host. However, there is a reference manual presenting and describing the API. This reference, though, lacks some details of implementation—*e.g.,* it does not describe the loadAsioDriver call, which is fundamental to use ASIO; and
- The source code is not clearly organized. Maybe due to lack of description on how ASIO host systems are structured, finding out the actual intent of each module was not trivial. Subjectively judging, the code also does not seem very readable in some sections, which increases maintenance difficulty.

Since the code is not self-descriptive enough to cover up drawbacks in the documentation, the reliability of ASIO hosts is somehow reduced, increasing the chance of unexpected bugs. Since ASIO is generally only used by corporations, there is also very little informal support available on the Internet. In the following section, I report a flaw with which I had to deal while working with ASIO.

B.2 The Process Crash and Lock Problem

This problem arose whenever the ASIO host process was terminated for some reason. It could be terminated by Visual Studio (when stopping debugging), by the operating system (through the Task Manager) or by either of those if it crashed for any reason (an unexpected exception, an invalid pointer access, etc.), which is normal and expected during a development process. In Windows, an ASIO host communicates with an ASIO driver using a COM interface[2]. When the host process crashed, it was terminated without releasing certain resources, probably the COM object instance used by ASIO. For a reason I and others have not discovered (but I believe it is related to COM), Windows did not eliminate the process from the process pool; the process remained apparently inactive. As a result, the executable file was in use and could not be modified and, consequently, the code could not be recompiled. Forceful termination such as by the Task Manager had no effect. To continue development and modify the code, the machine would need to be rebooted each time this problem appeared. If one just wished to run another copy of the process, the ASIO driver would still be in use by the last process instance, therefore also making ASIO unavailable until rebooting.

[2] One should not confuse an interface for a Microsoft COM object with the COM port interface.

The reader may feel tempted to argue that it is the programmer's responsibility to ensure that the host does not crash, which is quite valid. Keep in mind, though, that several errors are not directly identifiable, specially when designing new algorithms, and that correction must hold only after the test stage.

To avoid this problem, I considered temporarily using the Windows MME audio interface and only used ASIO when most programming errors were already corrected. Then, I installed the ASIO4ALL driver, which can be obtained at http://www.asio4all.com. After installation, I noticed that this problem would no longer occur, either using Creative's ASIO driver or the ASIO4ALL driver. I thoroughly tested all situations that previously led to this problem and found both drivers now worked properly. At this point, I stopped looking for an explanation, as this is probably a local problem that does not appear on other systems, and it is not my goal to determine exact causes for such failures, though a more formal examination should be performed in the future.

B.3 Summary

I have invested a considerable amount of time trying to fix the described issues, mainly because ASIO is the widespread default system for real-time audio streaming, and because my most fundamental motivation is real-time audio output.

ASIO is much more difficult to use than the readily available MME interface. The SDK is not "developer-friendly" as well. I invested a long time trying to resolve issues I just discussed, since my application depends entirely on the stability of the audio subsystem. In any case, it is not my responsibility, according to the goals of this work, to enter systems' internals to fix their errors. I could only develop my code as stable as possible such that situations that cause such instabilities in ASIO do not occur.

At last, after the "crash and lock" problem was resolved, I decided to use RtAudio 3.0.3[3], since is is more likely to be stable, due to testing, than my own code accessing the ASIO calls directly.

[3] Any of the libraries listed on Section B.1 could have been used.

APPENDIX C IMPLEMENTATION REFERENCE

C.1 OpenGL State Configuration

OpenGL needs to be configured as follows to allow general purpose computation on the GPU:

- Create an OpenGL rendering context. On Windows, this is performed by calling *wglCreateContext()* and *wglMakeCurrent()*, generally right after having created a window where graphics would be presented. In our application, window creation is not completed, since it is not necessary—the framebuffer is not used, only textures;

- Check hardware support for extensions. Before we proceed, we need to know: the maximum texture size, whether the *framebuffer object* extension (JULIANO et al., 2006) is supported and how many attachments are supported per framebuffer object (FBO);

- Generate one FBO and two or more same-sized textures and attach them to the FBO;

- Load the shaders used for audio processing; and

- Ensure that the following features are *disabled*: LIGHTING, COLOR_MATERIAL, MULTISAMPLE, POINT_SMOOTH, LINE_SMOOTH, POLYGON_SMOOTH, CULL_FACE, POLYGON_STIPPLE, POLYGON_OFFSET_POINT, POLYGON_OFFSET_LINE, POLYGON_OFFSET_FILL, POST_CONVOLUTION_COLOR_TABLE, POST_COLOR_MATRIX_COLOR_TABLE, HISTOGRAM, MINMAX, COLOR_SUM, FOG, SCISSOR_TEST, ALPHA_TEST, STENCIL_TEST, DEPTH_TEST, BLEND, DITHER, LOGIC_OP, COLOR_LOGIC_OP, AUTO_NORMAL;

- Ensure that the following features are *enabled*: TEXTURE_1D, TEXTURE_2D;

- Set the shading model to SMOOTH and set the polygon mode to FILL for faces FRONT_AND_BACK;

- If clearing is desired, set the clearing color to $\langle 0, 0, 0, 0 \rangle$; and

- Provide *identity mapping*, such that texture coordinates indicate exactly the matrix positions on texture bitmaps. This is achieved by doing the following:

- Set up a two-dimensional orthographic viewing region from $\langle 0, 0 \rangle$ to $\langle T_w, T_h \rangle$, where T_w and T_h refer to the width and height of the textures assigned to the FBO;
- Set up an identity modelling transformation and an identity texture transformation and apply on both a translation on each axis (except by z) of 0.5—this is done because memory positions are assigned, by definition, to the center of pixel cells; and
- Set up the viewport transformation with the position parameters referring to $\langle 0, 0 \rangle$ and the size parameters referring to $\langle$texture width, texture height$\rangle$.

Blending is disabled because OpenGL clamps luminance values into the [0.0,1.0] range if it is enabled. At this point, the application has at least 2 texture names and 1 FBO with those textures attached to it. OpenGL provides a call to change the target texture, *glDrawBuffer()*, that takes as arguments not the texture names, but identifiers representing textures bound to an FBO. Source textures can be defined passing *uniform* parameters—*i.e.,* not subject to interpolation—to fragment shaders. Performing these operations many times is error-prone due to the different mechanisms of reference to textures. One can write a procedure to perform swapping of source and target textures automatically.