Magisterabschlussarbeit im Studiengang

Kommunikationswissenschaft

Technische Universität Berlin

A fast convolution engine for the Virtual Electronic Poem project

Stefan Kersten (186183)

Betreuer: Prof. Dr. Stefan Weinzierl

15. September 2006

Contents

Chapter 1

Introduction

The Virtual Electronic Poem (VEP) project is a joint effort of several European institutions aiming at the reconstruction of the *Poème électronique* – one of the first and most influential pieces of media art and electroacoustic music – by means of virtual reality presentation techniques.

Funded by the EU commission under the *Culture 2000* programme, the project was carried out as a collaboration between five institutions and individuals, each responsible for a different aspect of the reconstruction: *VRMM Park and Università di Torino Italy* (visual reconstruction), *Department of Computer Science University of Bath* (reconstruction of the score), *Instytut Informatyki Politechnika Slaska Gliwice* (web presentation), Kees Tazelar of *Institute of Sonology The Hague* (musical and historical expertise) and *Fachgebiet Kommunikationswissenschaft Technische Universität Berlin* (acoustical reconstruction).

The project started in September 2004 and was presented at the ICMC in Barcelona in October 2005.

This thesis presents a binaural rendering engine developed for the VEP project and a method for dealing with large impulse sets by means of an impulse response cache. The engine was presented together with a visual rendering for head-mounted display as an integrated, immersive installation, providing spectators with an experience as closely to the original performance as possible with today's technology.

Chapter 2 introduces the original *Poème électronique* and documents the methods used in the reconstruction project.

Chapter 3 provides an overview of binaural room impulse responses and their use in virtual room simulation, describes various convolution algorithms in the time and frequency domains and examines existing impulse response partitioning schemes for latency reduction.

Chapter 4 describes the architecture of SuperCollider, the software environment used for realization of the binaural rendering engine.

Chapter 5 presents the components of the rendering application, describes the low-latency convolution algorithm used and introduces the impulse response cache.

Chapter 6 finally discusses possible enhancements to the rendering engine indicated by performance measurements carried out in the preceding chapter.

Chapter 2

The Virtual Electronic Poem Project

The *Virtual Electronic Poem Project* is an attempt to reconstruct the peculiar impression of a multimedia presentation in the Philips pavilion at the world exposition of 1958 by means of virtual reality techniques. The following sections introduce the conception of the historic pavilion and present the reconstruction efforts carried out by the team of researchers taking part in the VEP project.

2.1 The Historic Pavilion

The Philips pavilion at the 1958 world exposition was dedicated not so much to the presentation of commercial products, but to the impressive use of Philips technology in a piece of multimedia art [16].

Louis Kalff, then art director at Philips, proposed the idea of designing a pavilion to renown architect Le Corbusier, who immediately accepted and developed the notion of an "electronic poem" inside a building without obvious facades. The piece would be comprised of elements in different media, such as sound, visual projections and color and be of about ten minutes in

length, including an interlude of two minutes composed by Iannis Xenakis, then Le Corbusier's assistant.

Le Corbusier directed the project and also provided the images for the visual projections, while the sound composition was developed by composer Edgard Varèse and the building itself was designed by architect and composer Iannis Xenakis (Figure 2.1).

Figure 2.1: The Philips pavilion at the Brussels World Exposition in 1958

In order to accommodate the large number of visitors that were expected at the exhibition each day, the pavilion had to provide a separate entrance and exit, located at opposite sides of the building. Xenakis' design resembles a stomach, that digests the spectators and lets them out transformed by the performance in the inside. Two almost vertical walls facing each other were required for visual projections; Xenakis' solution involved the use of curved

surfaces with varying radius, that also helped with acoustical inhibition of echoes and standing waves.

According to an original sketch by Xenakis in [26], about ten loudspeaker "routes" can be identified, comprising about 350 loudspeakers in total. The use of sound routes, either linear stretches or clustered segments, were essential to the realization of Varèse's objective of "sound masses" that should not blend into one another, but form disparate entities, identified by different timbres and sound intensities [30].

The *Poème électronique* was a unique and unprecedented synthesis of diverse media, which attracted around two million visitors during its existence. Despite the visionary nature of the project, the pavilion was destructed at the end of the exposition and only fragments of the original design remain, such as photographs, handwritten drafts, videos and tape recordings.

2.2 The Reconstruction

Although there have been other attempts to reconstruct the experience of the *Poème électronique* either in software or physically, the VEP follows a novel approach by use of immersive technologies (Figure 2.2).

Visually, the reconstruction provides a complete view on the scenery inside the pavilion, as well as a realistic and historically accurate virtual rendering of the original performance through a head-mounted stereoscopic display.

The aural reconstruction is based on a binaural rendering model for a predefined listener position, that combines the accurate computation of room impulse responses for 350 loudspeakers with the reconstructed score and the original sound material.

The room modeling – starting from a CAD model for the visual reconstruction – and impulse response rendering based on the reconstructed loudspeaker positions was realized by Sebastian Benser in the commercial software EASE [1] (Figure 2.3). The 350 individual loudspeaker binaural impulse responses could, after a thorough investigation of the reconstructed control score, be combined into groups of simultaneously sounding loudspeakers, thereby reducing the number of required BRIR sets to 204. Each set is comprised of 9720 separate binaural impulse responses for 360 horizontal degrees and 27 vertical degrees ($-40°$ to $90°$ in $5°$ steps) of resolution (Fig. 2.4).

Mainly due to the necessity of reducing the overall computation time for the BRIR sets, only the first 0.13s of early reflections were computed with a

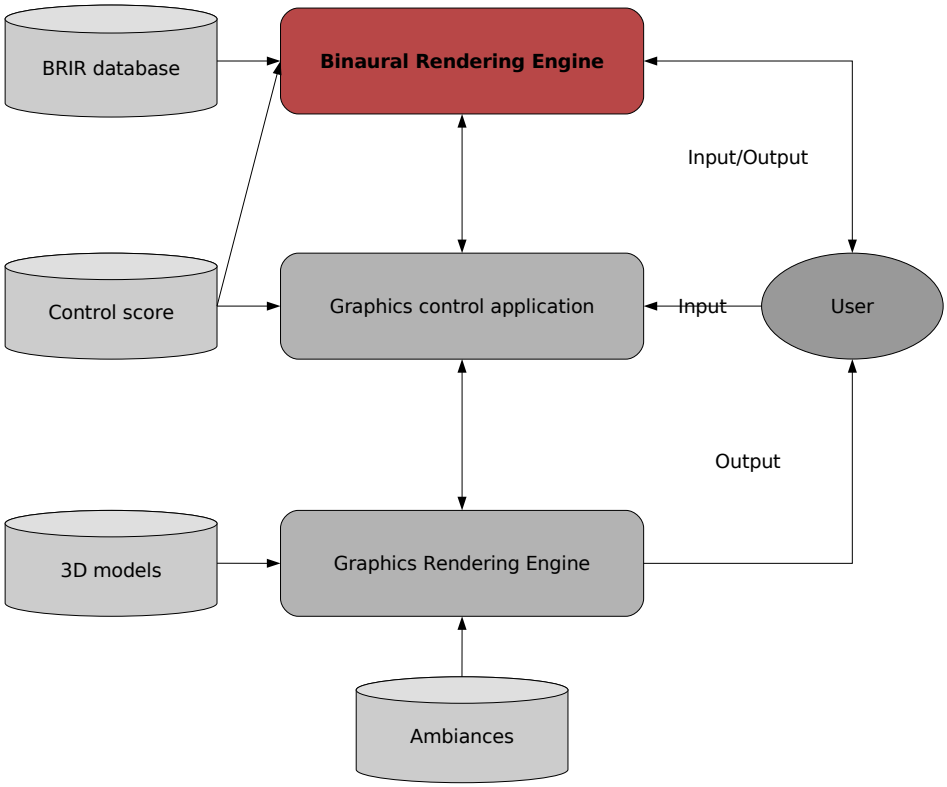

Figure 2.2: VEP virtual reality rendering architecture

sampling rate of 44100 Hz and 16 bit resolution. The reverb tail is comprised of a single binaural impulse response of about 1.6s.

The control score reconstruction was a joint effort of Kees Tazelaar, Richard Dobson and John Fitch [6], based on 30 seconds of original score material by Varèse.

2.3 Convolution engine requirements

Owing to the physical structure of the piece in the pavilion, the binaural rendering subsystem, in the following referred to as the VEP application, must meet several requirements.

Three tracks of audio corresponding to the individual tape tracks should be convolved with the computed BRIRS of the corresponding loudspeaker

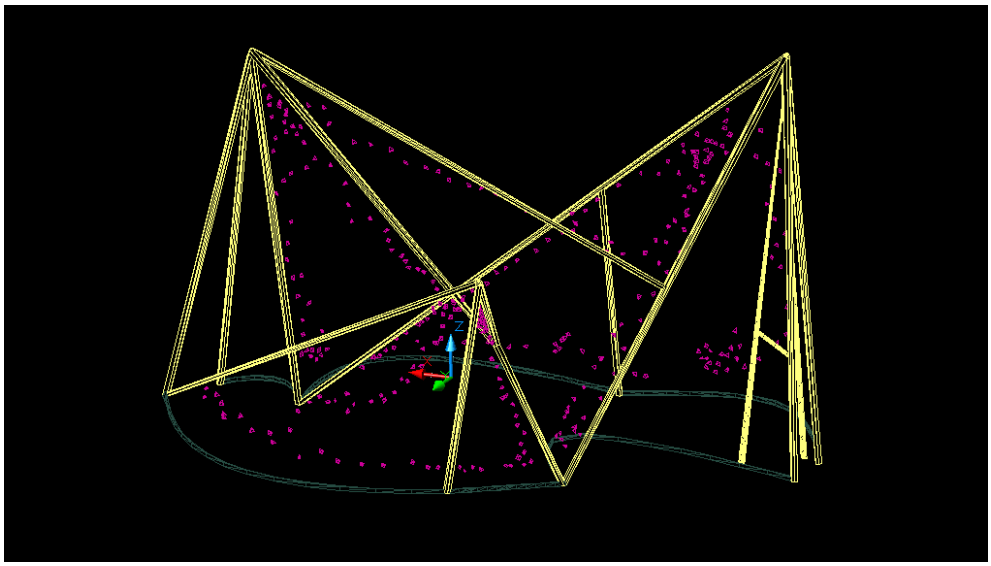

Figure 2.3: CAD model of the pavilion with reconstructed loudspeaker positions

route for two channel headphone realization. The convolution process for each track should respond to head tracker input induced by listener head movements and score events triggering the transition to a new loudspeaker.

Due to the large number of loudspeaker groups and the corresponding amount of BRIR data (about 75 GB), the engine should be able to load impulse responses from disk on demand, depending on score and head tracker input.

Score input for switching loudspeakers should be possible either from the graphical rendering application, for visualization of currently sounding loudspeaker groups, or separately from a text file.

Finally, the tracker interface should be reasonably general in order to interface either with the head tracker of the head-mounted display or another standalone tracking device.

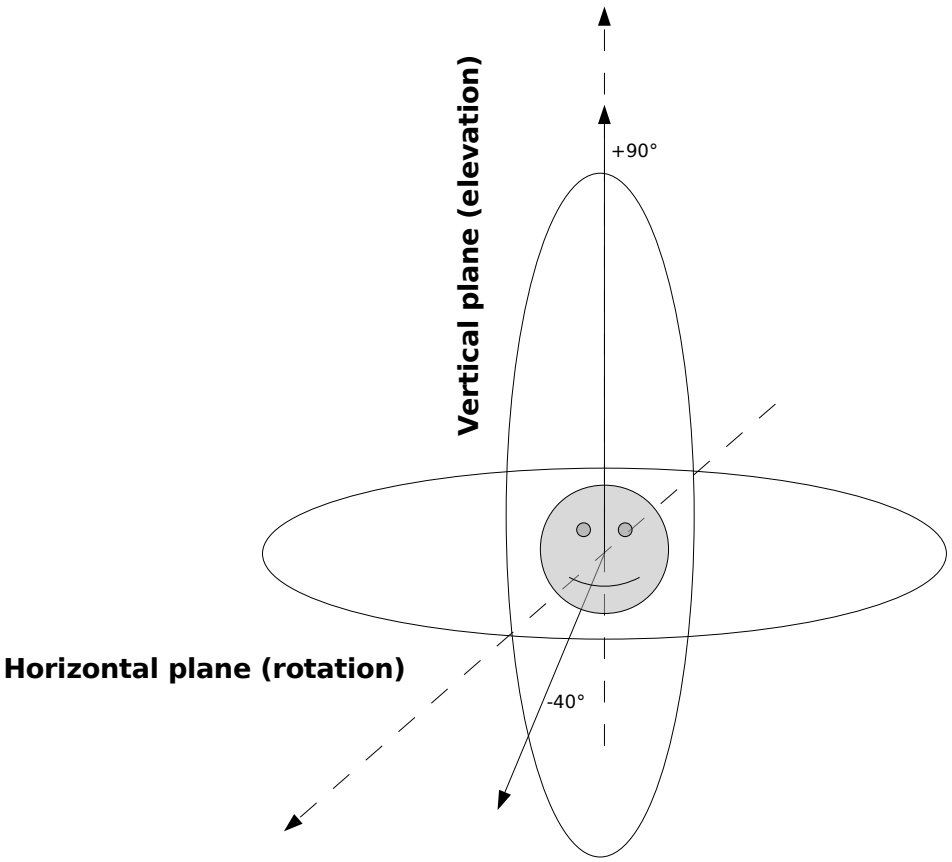

Figure 2.4: Horizontal and vertical planes of listener head orientation

Chapter 3

Fast Convolution

Convolution is a method of imposing the "quality" of a linear system onto an arbitrary input signal, thereby creating a filtered output. In the following a review of the use of impulse responses in binaural rendering applications shall be made and an overview given over different methods for convolution.

3.1 Binaural room impulse response and convolution

Localization in the human auditory tract in the horizontal and median plane is based on the effects of interaural time differences and interaural level differences as well as diffraction phenomena in the proximity of the head, that are evaluated by the human brain for clues about the three dimensional angle of incoming sound events [32, p.26].

These frequency dependent linear distortions can be modeled as a linear time-invariant (LTI) system, where input signals are filtered with a different FIR filter response depending on their direction. The filter response is called head related transfer function (HRTF) and is represented in the time domain by a head related impulse response (HRIR). HRIRs can be measured for different directions by means of a dummy head and corpus in an anechoic room; it

turned out to be beneficial to measure a set of impulse responses with fine horizontal resolution of at least 1° [21].

According to the principle of superposition, LTI systems can be fully described by delayed and appropriately scaled unit impulses [24, p.24]. The impulse response denotes the system response to a single unit impulse input. Discrete time filtering is then the superposition of one scaled version of the impulse response for each input sample.

Due to the linearity of convolution, room response information can be combined with HRTFs into binaural room impulse responses (BRIRs). This process can involve measuring HRTFs in a room that is non-anechoic in order to directly obtain BRIRs for convolution [15]. The method employed in the VEP project – as carried out by the EASE modelling software – is convolving discretized response histograms with the HRTF corresponding to the direction of each modeled mirror sound source [32, p.66]. Both methods generate sets of impulse responses for each possible listener position - sound source combination with one impulse response for each listener head orientation in the horizontal and the vertical plane.

Room impulse responses can generally be divided into three parts: direct sound, early reflections and reverb tail (Fig. 3.1). Since the reverb tail exhibits only statistical features of the energy distribution in the room impulse response, and does not contribute to binaural localization, it can be processed separately from the early reflections.

3.2 Time Domain Convolution

Convolution is the process of "combining" the impulse response of a linear system with an input signal to obtain the filtered output of the system and is denoted by the $*$ operator.

The discrete time operation of convolving input signal $x[n]$ with the impulse response $h[n]$ is defined as follows [24, p.24]:

$$x[n] * h[n] = \sum_{k=-\inf}^{\inf} x[k]h[n-k] \tag{3.1}$$

When processing an infinite signal on a block by block basis, 3.1 can be rewritten as

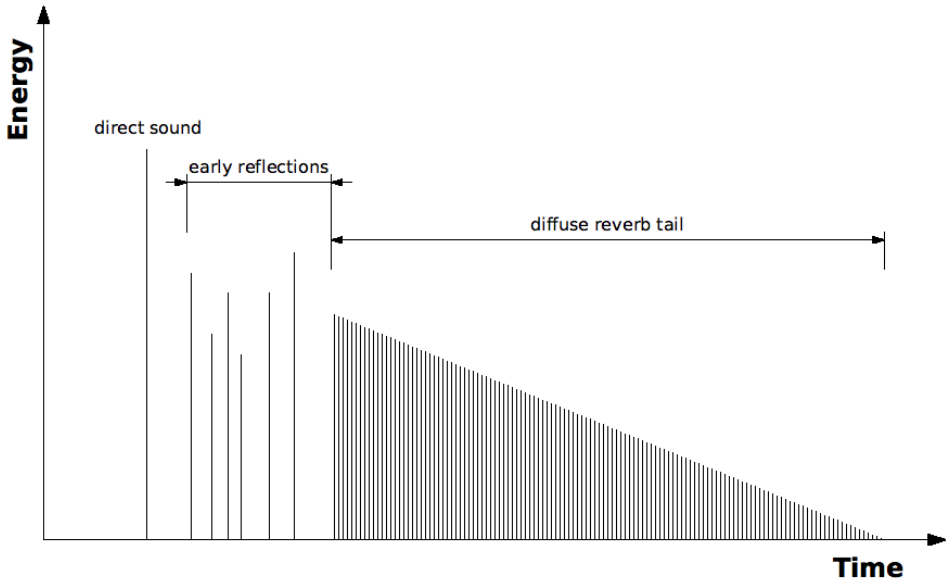

Figure 3.1: Schematic impulse response diagram

$$x[n] * h[n] = \sum_{k=0}^{L-1} x[k]h[n-k] \tag{3.2}$$

where L is the length of the impulse response. When the input signal block size is N, then the convolution result has the length $N + L - 1$, or equivalently $N + M$, where $M = L - 1$ is the FIR filter order [25, p.108]. The filter decay tail of length $L - 1$ must be handled specially in block-based time domain convolution, in what is usually called the overlap-add block convolution method [25, p.146]. Each tail M_i resulting from the convolution operation in block i is saved in memory until processing of block $j = i + 1$ and added at the beginning of the convolution result y_j of length N (Fig. 3.2).

The complexity of time domain convolution in O notation – counting the number of multiply-adds per output sample – is $O(N)$, where $N = L$ is the length of the impulse response and block size.

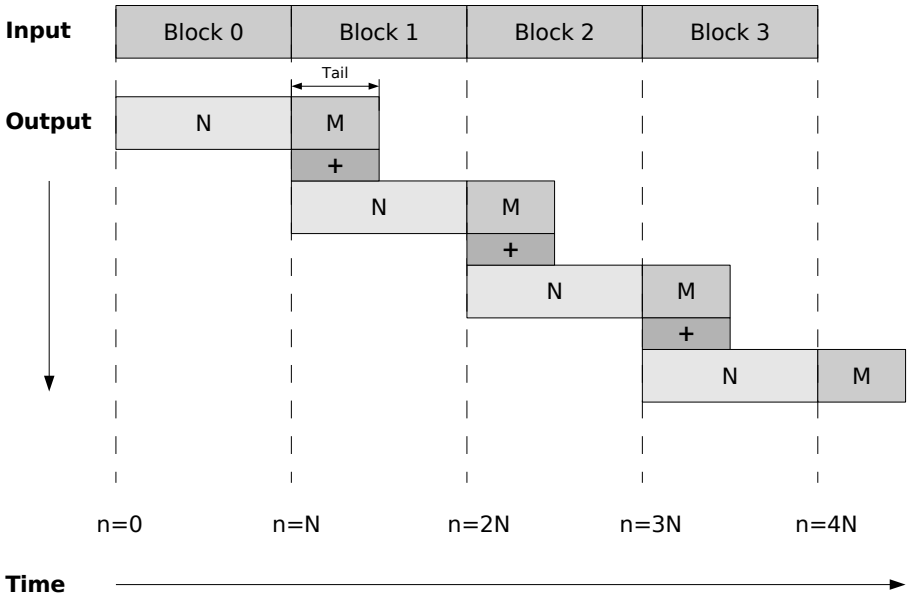

Figure 3.2: Overlap-add block convolution

3.3 Frequency Domain Convolution

It can be shown that the convolution of two signals in the time domain corresponds to a multiplication of their DFT coefficients in the frequency domain [24, p.632]

$$y[n] = h[n] * x[n] \Leftrightarrow Y[k] = H[k]X[k] \tag{3.3}$$

where the DFT is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn} \tag{3.4}$$

The convolution result can be recovered by transforming $Y(k)$ back into the time domain with the inverse DFT (DFT^{-1}) operation

$$x[n] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{j\frac{2\pi}{N}kn} \tag{3.5}$$

Thus, frequency domain convolution can be expressed symbolically as

$$y[n] = \mathrm{DFT}^{-1}(\mathrm{DFT}(h[n])\mathrm{DFT}(x[n])) \tag{3.6}$$

Care must be taken when interpreting the result, because due to the periodicity of the Fourier transform in the frequency domain and discretization of the spectrum by the DFT, the inverse DFT computes the modulo-N wrapped output signal, a property that is referred to as circular or cyclic convolution [24, p.656] [25, pp.524-529]. When undersampling in the frequency domain, part of the output signal appears to be wrapped back into the beginning of the current output block.

As mentioned in 3.2 the convolution of an input sequence block of length N and an impulse response of length L produces a result of length $N + L - 1$ and thus leads to aliasing after conversion of the convolution result to the time domain, unless constraining the size N_{DFT} of the transform to

$$N_{DFT} \geq N + L - 1 \tag{3.7}$$

3.4 Fast Convolution

Convolving two input sequences in the frequency domain was not feasible until 1965, when the Fast Fourier Transform (FFT) was developed by Cooley and Tukey [3]. Replacing the DFT by the FFT in Eq. 3.6 yields a form of frequency domain convolution usually referred to as fast convolution.

The FFT is an efficient implementation of the DFT, based on a divide and conquer approach that decomposes the problem into smaller subproblems and constructs the final DFT from the sub-DFT results. Subdivision of the initial DFT into smaller DFTs works well when N_{DFT} is highly factorisable and many efficient implementations require the size of the FFT N_{FFT} be a power of two, i.e.

$$N_{FFT} = 2^B \tag{3.8}$$

where $B \geq 0$.

In order to satisfy Eq. 3.8 in frequency domain convolution, the input block and the impulse response is usually padded with zeros to the next power of two. The resulting padded blocks of length N are then padded with zeros to obtain the size of the FFT $N_{FFT} = 2N$ to satisfy Eq. 3.7. The aliased

components in the block of length N_{FFT} are zero and thus not audible after the inverse transformation.

The cost of single block frequency domain convolution is composed of the cost of the FFT and FFT^{-1}

$$O(N_{FFT}) = O(N_{FFT} \log_2(N_{FFT})) \tag{3.9}$$

and the complex multiplication in the frequency domain with $4N$ real multiply-adds per block and can be derived from 3.9 by substituting $N_{FFT} = 2N$ and normalizing with $\frac{1}{N}$ to obtain the complexity per output sample:

$$\begin{aligned} O(N) &= O(2\frac{2N \log_2(2N)}{N} + \frac{4N}{N}) \\ &= O(4(\log_2(N) + 2)) \end{aligned} \tag{3.10}$$

When the overlap-add scheme as in 3.2 is used, another N adds and stores have to be taken into account for mixing the previous convolution tail of length N into the current output block.

3.5 Partitioned Convolution

Using convolution techniques in interactive systems usually imposes upper bounds on the acceptable output latency of the system, where latency in this context means the time passing between a change of state in the input – e.g. head tracker data – and the instant this change is reflected in the output.

Time domain convolution has the desirable property of providing zero latency when realized in a DSP system but it doesn't scale well with increasing impulse response lengths. Single block frequency domain convolution, on the other hand, scales extremely well with larger problem sizes, but it's latency is equal to the length of the impulse response padded to the next power of two, which is unacceptable for room impulse responses.

Following the linearity property of the convolution however, it is possible to partition the time-domain impulse response in separate blocks, convolve these blocks with the input signal and delay the convolvers' outputs according to the start of the respective partition in the impulse response.

3.5.1 Uniform partition sizes

First proposed by Stockham in 1966, partitioning the impulse response into blocks of uniform length is one possibility of reducing the I/O delay of the convolution process [29].

Each block is first transformed into the frequency domain, multiplied with the frequency domain representation of the corresponding input block and accumulated into a frequency domain buffer, that finally is converted to the time domain with the FFT^{-1} operation.

Because all blocks have the same lengths, instead of delaying the time domain outputs, the delay line can operate on frequency domain input blocks, and only one FFT^{-1} is needed per output block. The time domain output buffer is then overlap-added into the output stream, as described in 3.4. Fig. 3.3 shows the process of convolution with uniform partition sizes the spectral domain input delay line and accumulator and the overlap add procedure at the output.

According to Torger [29], using uniform partition sizes has the benefit of shifting the load of the algorithm from the FFT to the complex multiply-add operation, which is more easily optimized by using SIMD vector instructions. Partitioning the impulse response not only makes sense when intending to reduce I/O latency, but also to improve caching of data and instructions on the processor.

Given a target latency of $D = \frac{N}{f_s}$ and an impulse response of length L, the partition size needs to be N and the cost of the algorithm is comprised of the costs for the FFT operations (see Eq. 3.10) and $4N$ complex multiply-add steps for each of the $\lceil \frac{L}{N} \rceil$ partitions

$$\begin{aligned} O(N) &= O(2(\frac{2N \log_2(2N)}{N}) + \frac{4N}{N}\frac{L}{N}) \\ &= O(4(\log_2(N) + \frac{L}{N} + 1)) \end{aligned} \tag{3.11}$$

García [10] shows by setting the derivative of the cost function to zero and solving for N, that the optimal blocksize can be calculated directly:

$$N_{opt} = \frac{L \ln(2)}{k} \tag{3.12}$$

where k is a constant derived from the efficiency of the particular FFT imple-

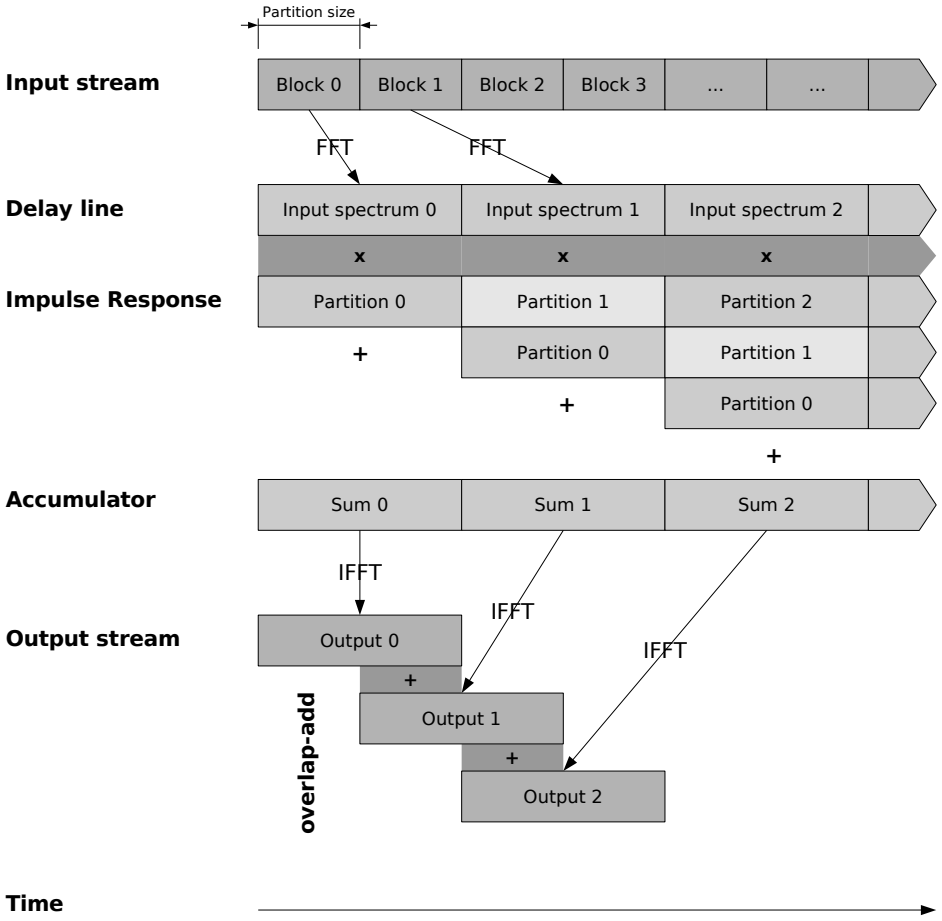

Figure 3.3: Partitioned convolution with uniform partition sizes

mentation. Conservatively assuming $k = 1$, the optimal block size for large impulse responses is too large for realistic latencies in interactive use – the algorithmic complexity increases linearly with L when N drops below the optimal block size, and thus doesn't scale well with combinations of small N and large L.

3.5.2 Non-uniform partition sizes

Gardner [11] proposes a scheme that partitions the impulse response into a number of non-uniform blocks and provides zero I/O delay at a reasonable

cost.

Fulfilling the constraints of providing latency N and ensuring that each partition of size N_i must start at least N_i samples into the impulse response in order to be able to collect enough input data and reconstruct the output without disruptions, a minimum cost partitioning scheme is starting with two partitions of size $N_0 = N_1 = N$ and doubling the size of each following partition, i.e. $N_2 = 2N$, $N_3 = 4N$ and so forth (Fig. 3.4). By convolving the first input block in the time-domain, zero latency can be achieved in a dedicated DSP environment, where interrupts are serviced at the sample level.

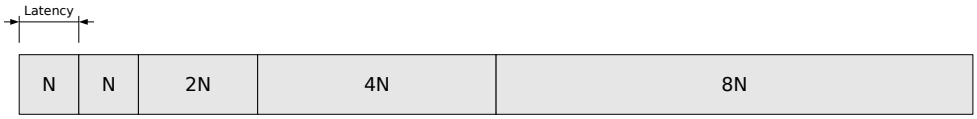

Figure 3.4: Non-uniform *minimum cost* partitioning scheme

Since the individual convolution processes have a processing period of N_i and each partition starts only N_i into the impulse response, in order to satisfy the constraint of uninterrupted reproduction at the output, the convolution results for each process must be made available instantly, leading to a non-uniform load on the processor.

A uniform load partitioning scheme doubles the partition sizes only every other partition (Fig. 3.5), i.e. each convolution process is allowed as much time for computation as it takes to collect input data, thus making the scheduling of the individual processes more flexible. Direct process scheduling, as described in [11], is not immediately applicable to general purpose operating systems, and shall not be repeated here. A detailed description of the algorithm used in the VEP application is given in 5.1.

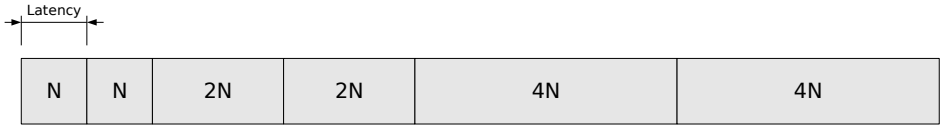

Figure 3.5: Non-uniform *uniform load* partitioning scheme

With Eq. 3.10 the cost for the uniform load algorithm is the sum of the cost for I processing modules, each processing a single partition size:

$$\begin{aligned} O(N) &= O(\sum_{i=0}^{I-1} 4(\log_2(2^i N) + 2)) \\ &= O(\sum_{i=0}^{I-1} 4(i + \log_2(N) + 2)) \\ &= O(\frac{4I(I+1)}{2} + 4I\log_2(N) + 8I) \\ &= O(2I^2 + 2I + 4I\log_2(N) + 8I) \\ &= O(2I^2 + I(4\log_2(N) + 10)) \end{aligned}$$

The length of the impulse response is $L = 2\sum_{i=0}^{I-1} 2^i N$ and thus $I = \log_2(\frac{L}{2N})$, which yields

$$O(N) = O(2[\log_2(\frac{L}{2N})]^2 + \log_2(\frac{L}{2N})[4\log_2(N) + 10]) \tag{3.13}$$

Chapter 4

The SuperCollider Architecture

The VEP audio rendering application is realized within the comprehensive SuperCollider synthesis and composition environment. SuperCollider provides a general purpose framework for generating and processing sound, as well as a general purpose programming language tailored to realtime applications.

4.1 SuperCollider Server

SuperCollider was originally developed by James McCartney on MacOS 9. Its predecessors *Synth-O-Matic*, a standalone synthesis application, and *pyrite*, a *MAX/MSP* object were the basis for the release of SuperCollider 1 in 1996 [17] and its successors SuperCollider 2 in 1998 and SuperCollider 3d5 in 2000 [18].
Up to this point SuperCollider was proprietary software, but in 2002 McCartney decided to port SuperCollider 3, or SuperCollider Server, to MacOS X and release the program under the GPL, an open source license that permitted further development of the application in a community of users and developers [19]. Between 2002 and 2003 the author of this thesis ported the SuperCollider environment to the Linux operating system [13] – today SuperCollider can be used on all three major platforms, MacOS X, Linux and Windows.

In the following, an overview of the SuperCollider Server architecture will be provided, as far as it is related to the VEP application.

4.2 Client-Server Architecture

Previous versions of SuperCollider were based on a single process model, where the SuperCollider language was executed in the same interrupt that is responsible for providing the audio hardware with new sample data [17]. This had the distinct possibility of providing the synthesis with control data from a higher level language synchronously and, more importantly, to be able to create new synthesis patches on the fly.

With version 3, SuperCollider Server, the architecture was changed to a client-server model (Fig. 4.1).

A small, efficient and portable synthesis engine – `scsynth` – is responsible for managing synthesis nodes and other resources such as audio and control signal communication buses and data buffers. At the same time it functions as a network server, exporting all the necessary commands for resource allocation and management, synthesis node control and state information through a specific protocol, OpenSoundControl (OSC, see 4.3) – the most important server commands are listed in Tab. 4.1. Currently, transport protocols supported by the synthesis engine are UDP and TCP.

The SuperCollider language `sclang` runs in a separate address space from the server and controls its operation by the use of OSC. The object oriented language is based on an efficient byte-coded interpreter with realtime execution semantics. It is close to Smalltalk, but also borrows from Lisp, J, Ruby and other programming languages, featuring a single-inheritance object model and introspection facilities.

Although `sclang` code has a byte-code representation interpreted by a virtual machine, many important methods are implemented as so called primitives, written in `C`. The virtual machine provides constant-time method lookup and a realtime gargabe-collector, allowing the use of the language environment in contexts with soft-realtime requirements, such as realtime musical composition, analysis and control.

The language core is comprised of classes for character strings, symbols, numbers and a comprehensive container data structure hierarchy inheriting from `Collection`. High level scheduling facilities are provided by instances of `TempoClock` and subclasses of `Stream`, allowing detailed control over the evolvement of processes in time on a high level of abstraction. A large hi-

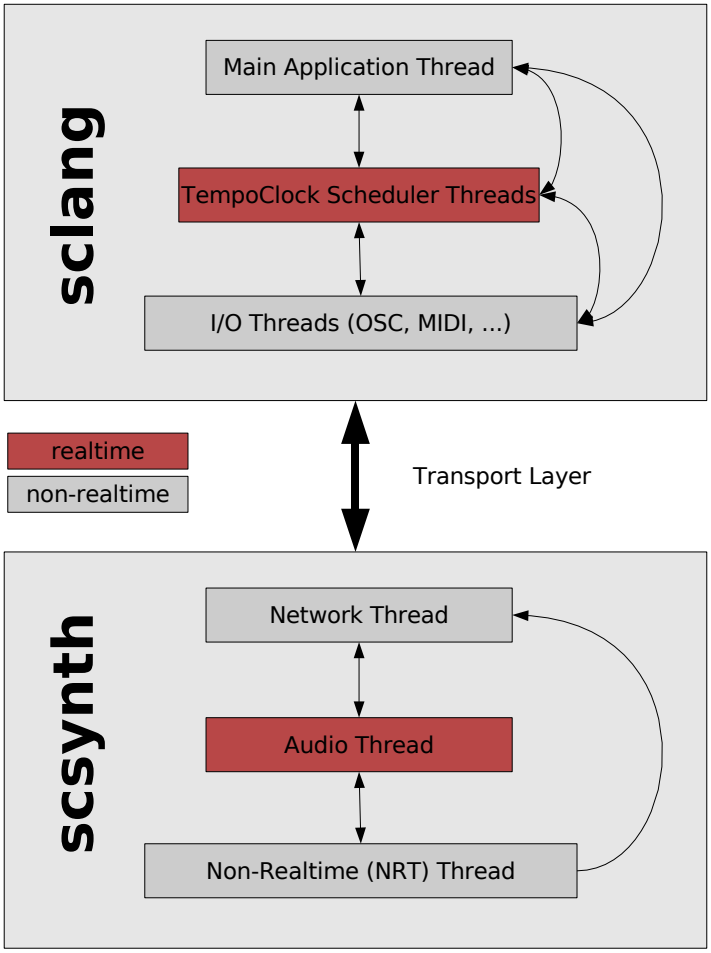

Figure 4.1: SuperCollider Server architecture

erarchy of classes inheriting from `Pattern` allows the expression of temporal behaviors in a functional and referentially transparent manner.

4.3 OpenSoundControl

OpenSoundControl is a binary protocol developed at the *Center for New Music and Audio Technologies* (CNMAT) in Berkeley for control of sound synthesis applications [33].

The protocol is packet based and transport independent, i.e. OSC packets can be transmitted through a variety of transport protocols, as long as both

Command	Description
`/s_new`	Creates a new synth from a synth definition and adds it to the tree of nodes.
`/g_new`	Creates a new group and adds it to the tree of nodes.
`/n_free`	Stops a node abruptly, removes it from its group and frees its memory.
`/g_freeAll`	Frees all nodes in the group.
`/n_set`	Takes a list of pairs of control indices or keys and values and sets the controls to those values. If the node is a group, then it sets the controls of every node in the group.
`/b_alloc`	Allocates a zero filled buffer with the specified number of channels and samples.
`/b_free`	Frees memory allocated for the specified buffer ID.
`/cmd`	Takes the name of a plugin defined command and passes any number of arguments to the appropriate command handler.

Table 4.1: Important OSC commands for `scsynth`

the transmitting and the receiving applications are able to use the same transport layer.

The most prominent transports used by OSC-enabled applications are UDP, TCP and in some cases serial protocols such as RS232. The VEP application uses UDP throughout, because packet loss turned out to be not an issue in a dedicated local network and UDP has far less bookkeeping overhead than TCP.

OSC, unlike MIDI, doesn't have a fixed namespace for commands supported by applications. Each application, or set of applications, must define their own command set tailored to the problem at hand. OSC commands are strings adhering to a virtual hierarchy similar to the Unix file system tree, where components are separated by forward slashes:

```
/synth/oscil/freq
```

OSC messages begin with a command string, followed by an argument tag string denoting the types of the arguments supplied with the message and the corresponding arguments encoded in binary form. Supported argument types are strings, 32 bit integer and floating point numbers and containers for arbitrary binary data, so called *blobs* (Tab. 4.2). For reasons of backwards

compatibility, the tag string starts with the character ",". The protocol specification makes sure that any numerical data is aligned to 4-byte boundaries, making it efficiently readable from memory on most architectures.

Type tag	Type description
`i`	32 bit integer (network byte order)
`f`	32 bit IEEE float (network byte order)
`s`	null-terminated character string, padded to 4-byte boundary
`b`	32 bit integer byte count followed by 4-byte padded binary data (*blob*)

Table 4.2: OSC argument type tag encodings

Messages can be grouped into bundles, starting with a 64 bit integer time tag and followed by any number of messages, where each message is preceded by a 32 bit integer byte count denoting the message length (Fig. 4.2).

Time tags are in standard *Network Time Protocol* (NTP) format [20], with the high 32 bits counting seconds from January 1st 1900 and the low 32 bits denoting the fractional part with more than picosecond precision.

4.4 Synthesis Graph

Signal processing tasks on the server are organized in a tree, that defines the order of execution of synthesis modules, while the vertices define containment relationships (Fig. 4.3).

Groups are nodes in the tree that can contain other nodes. They are mainly used for resource management, e.g. freeing all nodes contained in a group with the `/g_freeAll` OSC command, or controlling a group of nodes as an entity with the `/n_set` OSC command.

Synths, on the other hand, form the leaves of the tree and are the actual sound production entities. A synth is an instantiation of a *SynthDef*, a compiled binary representation of unit generator (UGen) synthesis graphs, quite similar to a Csound [31] instrument. A synth definition abstractly defines in the manner of a template which unit generators will be used and how they are to be interconnected. When a definition is instantiated, the actual processing units are initialized and connected through internally allocated buffer space that is used for communication of time domain sample data between

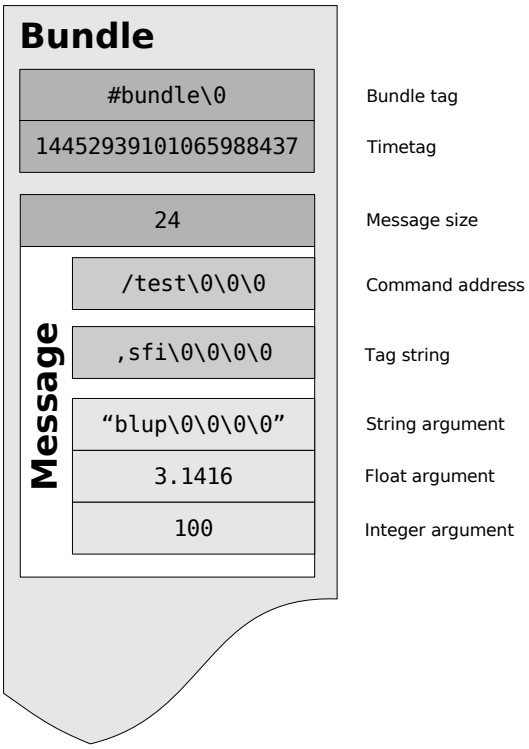

Figure 4.2: Anatomy of an OpenSoundControl bundle

units[1].

Nodes in `scsynth` are referenced by unique 32 bit integer identifiers that are passed to the server during node creation. Node IDs are mainly used to control the behavior of nodes during their lifetime e.g. with the `/n_set` message or to free them with the `/n_free` OSC command. The special node ID -1 is used to create “anonymous” nodes that can be referenced in subsequent commands only until -1 is used as a node ID again. The lifetime of anonymous nodes has to be managed either internally by special UGens like `FreeSelf` or externally via their enclosing group.

Synthesis nodes – as opposed to units embedded in a synth – are not connected directly by “wires” on the server but communicate through globally shared “buses”, such that synthesis nodes do not have to be aware of each other’s existence in order to exchange data (Fig. 4.3). Buses are simply containers for sample data that can be written and read by special UGens, most notably `In` and `Out`. Buses come in two varieties, audio rate for exchanging

[1]Exchange of frequency domain data is handled differently, but is irrelevant in the context of the VEP application.

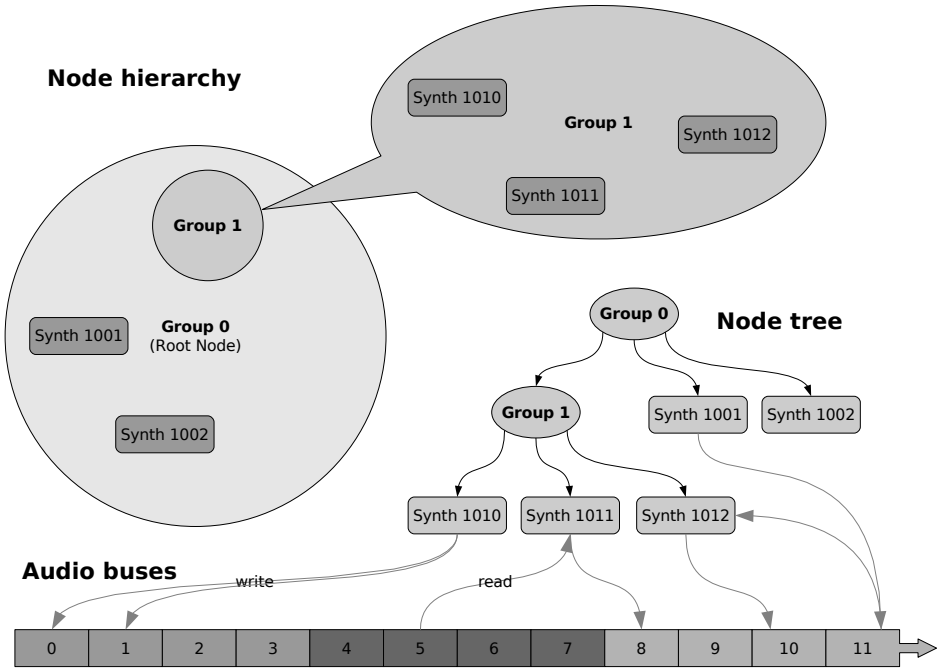

Figure 4.3: SuperCollider synthesis node tree and bus communication

sample data and control rate for transmitting control values. Audio rate buses hold one block of samples at a time while control rate buses hold a single floating point value per block.

Audio and control buses are layed out in a single contiguous region in memory; a fixed number of audio buses at the start of the bus memory are reserved for communication with the audio driver backend, i.e. for writing sample data to and reading from the audio hardware (Fig. 4.4).

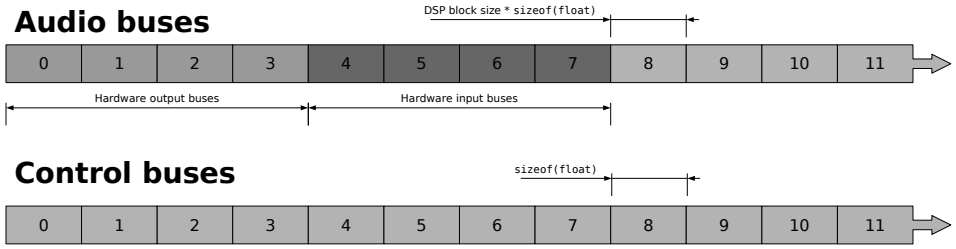

Figure 4.4: SuperCollider bus architecture

For audio signals, order of execution of synthesis nodes matters, because in

order to avoid feedback the `In` unit checks the bus write timestamp and zeros any data remaining from the previous cycle. Thus, a node n_0 reading from a bus written to by node n_1 must be placed later in the node execution hierarchy. This can either be achieved by an argument to the `/s_new` command, explicitly specifiying the placement of the new node in the containing group, or by allocating n_0 and n_1 in different groups, that already are in the correct order of execution. If audio feedback through buses is desired, the unit `InFeedback` can be used to read data of the current or the last cycle from an audio bus.

4.5 Unit Generator Plugins

The actual sound synthesis and signal processing routines in `scsynth` are written in the `C++` programming language and are loaded from compiled plugin modules.
Plugin modules export a single initialization function with the signature

```
void load(InterfaceTable *inTable);
```

that is called when the plugin is loaded and is used to define new unit generators and OSC plugin commands. Plugin code can in general not directloy call functions defined in `scsynth`, but must use the function pointers exported in the `InterfaceTable` structure in `headers/plugin_interface/SC_InterfaceTable.h` that is passed to the `load` function and must be stored in the static variable `static InterfaceTable *ft` during plugin initialization. In the course of the thesis, interface table functions will be given with their full signature as well as the signature of their corresponding preprocessor wrapper macro, that depends on the previous declaration of `ft`.
Each plugin defines one or more classes derived from the common base class `Unit` that mainly holds pointers to input and output buffers and to a sample calculation function with the signature

```
void (*UnitCalcFunc)(struct Unit *unit, int n);
```

that is called to produce one block of `n` samples[2].

[2]The DSP graph block size in `scsynth` is adjustable through the commandline argument `-z`. Typically it is 64 samples, while the actual hardware buffer size can be larger and depends on driver backend and host environment settings.

New unit definitions have to be registered with the synthesis engine by a call to one of

```
// Interface table function pointer

bool (*fDefineUnit)(char *className,

                    size_t allocSize,

                    UnitCtorFunc ctor,

                    UnitDtorFunc dtor,

                    uint32 flags);


// Wrapper macros

#define DefineSimpleUnit(name)

#define DefineDtorUnit(name)

#define DefineSimpleCantAliasUnit(name)

#define DefineDtorCantAliasUnit(name)
```

While the interface table function provides full control over the addresses of the constructor and destructor functions, the convenience macros rely on the naming convention `<UnitName>_Ctor` and `<UnitName>_Dtor`, respectively.
Unit constructors and destructors are declared to have the signatures

```
typedef void (*UnitCtorFunc)(struct Unit* unit);

typedef void (*UnitDtorFunc)(struct Unit* unit);
```

and are needed to initialize a unit after instance creation and to release any resources (e.g. memory) before the unit is freed.
One architectural property that `scsynth` shares with e.g. Csound is to not implement a fixed size synthesis "voice" structure and rely on preallocated re-

sources, but to allow new synthesis node instances to be created dynamically and in response to realtime data input. For this reason, unit constructors and destructors are called from the realtime audio processing thread and are required to adhere to strict realtime requirements, i.e. their complexity should – if at all possible – not be a function of a variable but be constant. This means for instance that only specialized memory allocation functions – provided by the `scsynth` runtime environment – should be used, no potentially blocking operating system calls should be utilized and that data initialization costs should be amortized across several callback cycles.

In order to produce audio samples written to the output buses, `scsynth` traverses the graph of nodes present at the instant of a specific audio hardware callback or interrupt and calls each node's calculation function. Container nodes continue by traversing their child nodes, whereas leaf nodes in turn call the sample calculation function of each unit contained in their body of execution.

The sample production callback must obey the same realtime constraints as the unit constructor and destructor functions and eventually writes audio samples to the output buffers of the owning unit.

4.6 Asynchronous Commands

OSC commands that don't meet the realtime requirements, such as those allocating memory from the operating system or reading data from disk, are executed asynchronously by a separate thread provided by the `scsynth` runtime environment.

Asynchronous commands are initiated from the realtime audio thread (Fig. 4.1) by calling the interface table function (and corresponding macro)

```
int DoAsynchronousCommand

(

    World *world,

    void* replyAddr,

    const char* cmdName,
```

```
    void *cmdData,
    AsyncStageFn stage2,
    AsyncStageFn stage3,
    AsyncStageFn stage4,
    AsyncFreeFn cleanup,
    int completionMsgSize,
    void* completionMsgData
);
```

where the stage callbacks have the signatures

```
bool (*AsyncStageFn)(World *inWorld, void* cmdData);

void (*AsyncFreeFn)(World *inWorld, void* cmdData);
```

After having passed a command data structure – through a lockfree ringbuffer – to the non-realtime helper thread, the *stage2* function is executed and may do any work that is not realtime-safe. When this function returns, the command structure is passed back to the realtime thread, where the *stage3* function is executed. When *stage3* returns true, an artbitrary OSC message usually passed by the network client that initiated the operation, is executed. Finally, *stage4* is executed in the non-realtime thread and sends a `/done` OSC message back to the requesting client (Fig. 4.5). The optional *cleanup* callback is executed as a last stage in the realtime thread in order to release resources allocated from the realtime memory manager.

4.7 Sample data buffers

Another important resource on the server, apart from synthesis nodes, are sample data buffers. Buffers are allocated by the `/b_alloc` and `/b_allocRead` OSC commands and can be freed with the `/b_free` OSC commands. They

are, just as nodes, indentified by globally unique integers.

The sample data held by buffers is defined to have a 32 bit floating point representation, but the exact interpretation is largely up to the unit generator accessing the buffer. The data structure includes fields for sample rate and number of channels, in order to accomodate the most common use for time domain audio data.

Fig. 4.6 shows the structural sharing of buffers between the realtime thread and the non-realtime thread (see Fig. 4.1). The buffer data strutures accessed from the realtime thread are held in a copy in the non-realtime thread. Operations involving asynchronous updates to a buffer structure first modify the mirror copy in the non-realtime thread and afterwards synchronously update the original in the realtime thread from the *stage3* callback function executed in the same thread (see 4.6). The assumptions in this update strategy are that

- asynchronous commands are executed and completed sequentially, i.e. no concurring updates of the same buffer from different commands take place,
- the respective buffer mirror image is only modified from the associated thread.

4.8 Plugin Commands

In addition to the set of OSC commands built into `scsynth` plugins may define their own custom commands. This is done by a call to the interface function

```
bool DefinePlugInCmd

(

    char *cmdName,

    PlugInCmdFunc func,

    void* userData
```

```
);
```

Plugin command functions are executed in the realtime thread, but they may spawn an asynchronous command (see 4.6) for non-realtime processing.

Plugin commands can be executed by sending the `/cmd` OSC command to `scsynth` (see Tab. 4.1), passing the plugin command name and any number of arguments.

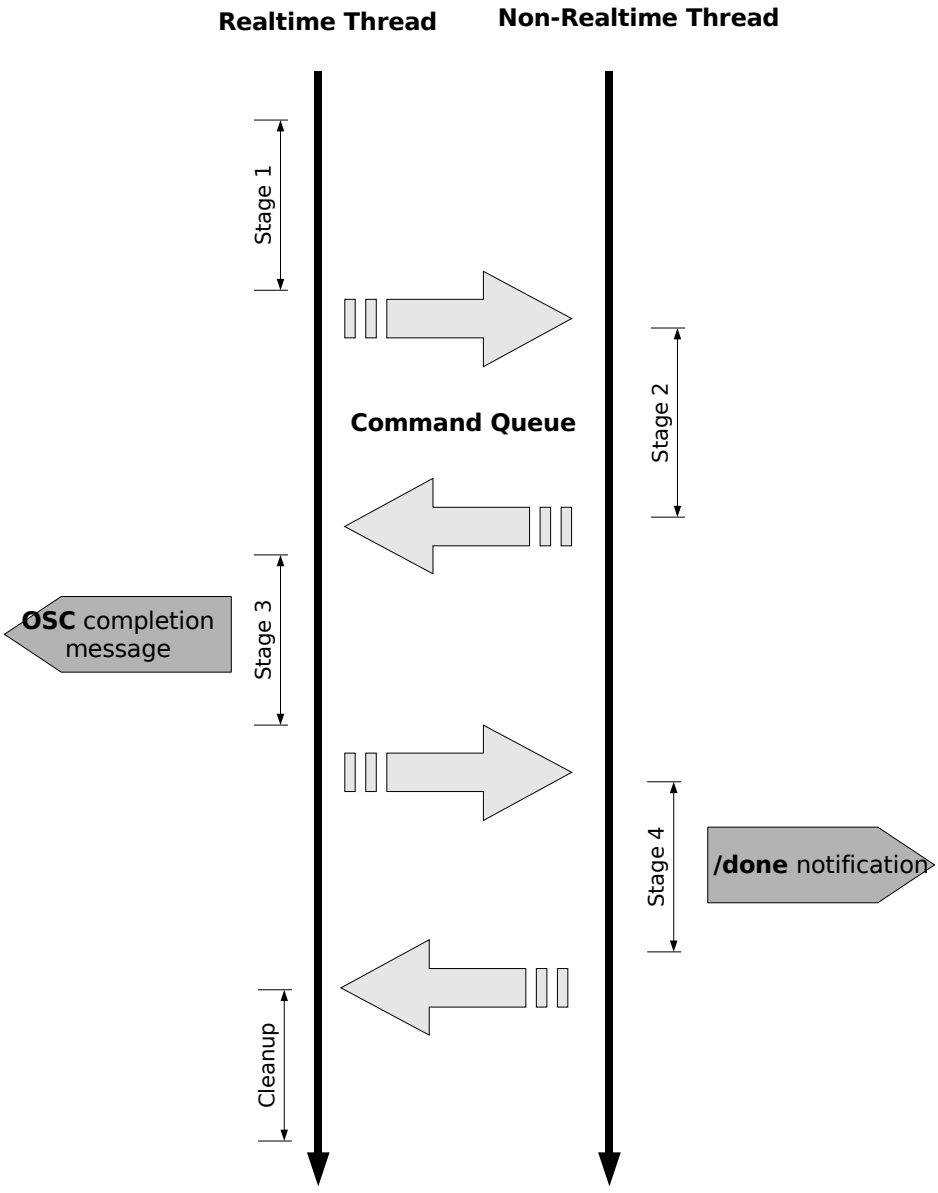

Figure 4.5: Asynchronous command execution flow

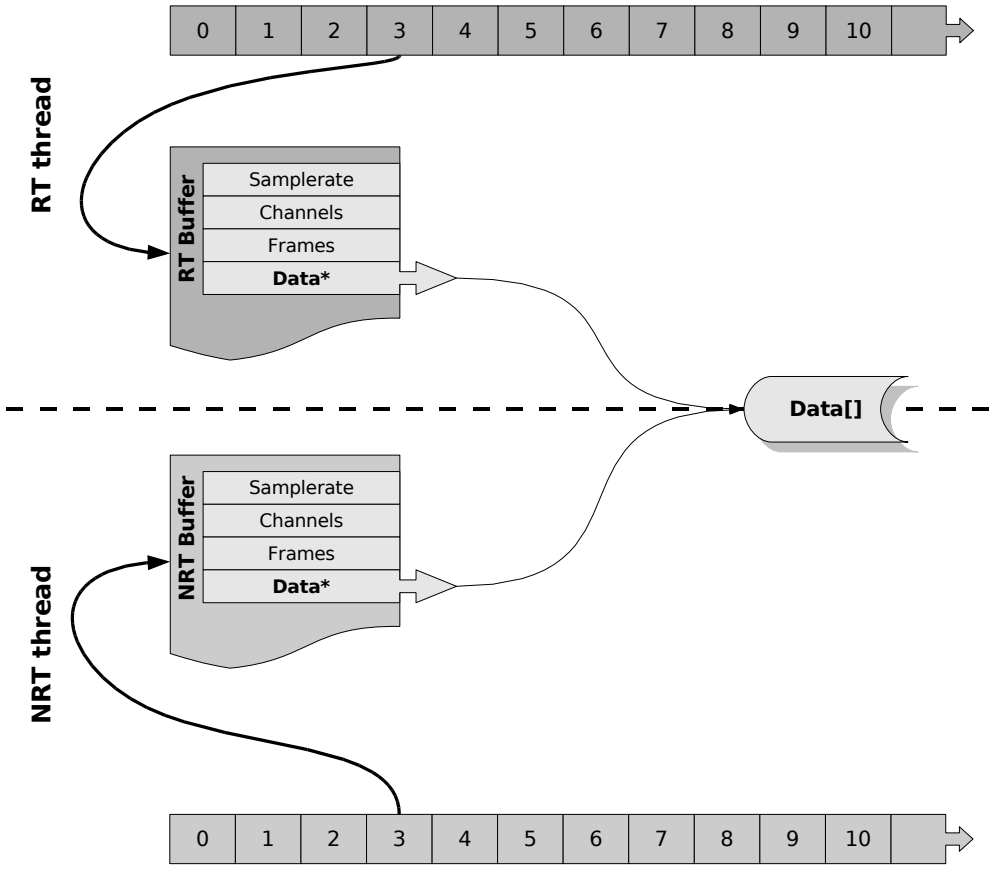

Figure 4.6: RT and NRT buffer mirror structure

Chapter 5

The VEP Application

In the following sections a description of the VEP application will be provided, along with implementation details for the respective subsystems.

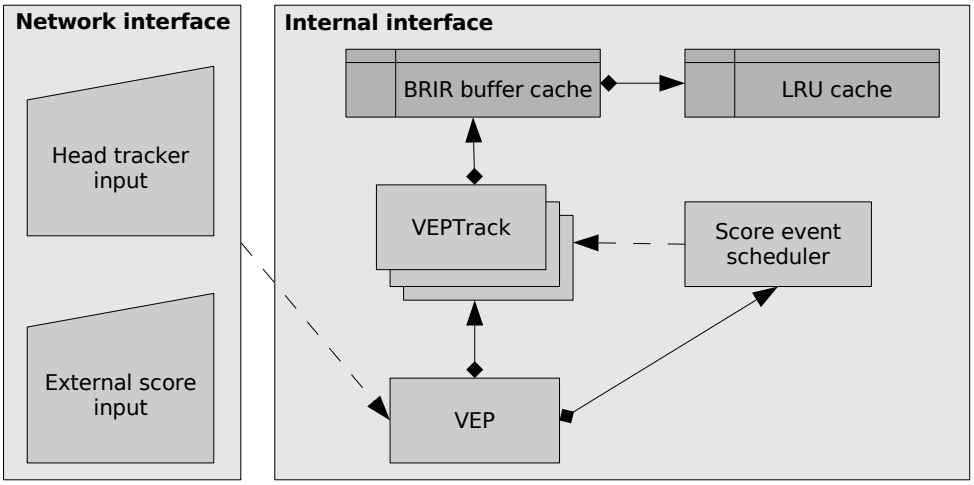

Figure 5.1: Control flow and aggregation structure of the VEP binaural renderer

Figure 5.1 gives an overview of the subsystems that constitute the audio rendering application, while Fig. 5.2 depicts the audio signal routing structure of the application.
The convolution is realized within a `C++` plugin for `scsynth` and provides

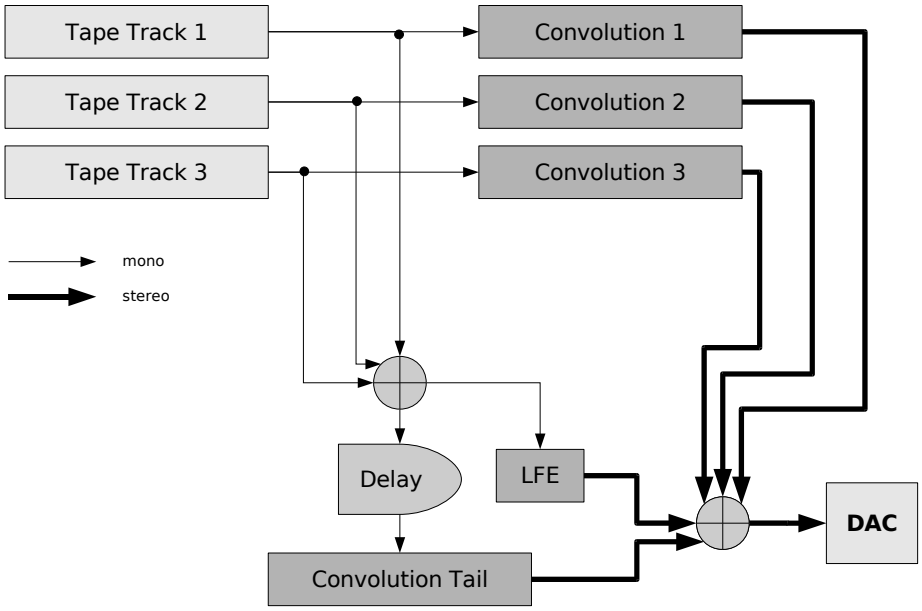

Figure 5.2: Audio signal flow of the VEP binaural renderer

uniform as well as non-uniform partitioning methods. As mentioned in 2.2, the impulse response head of about 0.13s is processed separately from the tail of about 1.6s. The convolution processes for the individual tracks are provided with dynamic updates from head tracker and control score input, and need to exchange the head parts of the binaural impulse response dynamically. The statistical tail convolver is fed a mono sum of input signals and produces a stereo signal, that is mixed into the main outputs.

The disk access plugin is implemented as an asynchronous plugin command for `scsynth` and is responsible for loading binaural, i.e. two-channel impulse responses from disk and partitioning them into a server buffer object for use by the convolution plugin.

Playback of the three tape channels converted to digital from the original analog tapes is the responsibility of the tape player.

The score player reads score commands for switching between loudspeakers in each track and passes the appropriate information.

5.1 Convolution Unit Generator

The convolution unit generator `VEPConvolution1` utilizes a non-uniform partitioning scheme as described in 3.5.2 and is implemented as a subclass of `Unit` in `src/VEP.cpp`.

5.1.1 Partitioning scheme

The partitioning scheme employed is based on the uniform load scheme proposed in [11], with two notable differences. First, since samples coming from the analog to digital converter (ADC) and moving to the digital to analog converter (DAC) are buffered by the hardware and the underlying driver, interrupts are not delivered per sample but per blocks of samples, implying a non-zero latency inherent to the buffering – thus the first partition is not convolved by time domain convolution but in the frequency domain. Second, in order to reduce the amount of internal buffering logic, the first four partitions are of length N_0, making the output delay line a multiple of the partition size N_i for each partition. Partition sizes start with N_0 equal to the SuperCollider DSP block size, which is an integral subdivision of the hardware buffer size and denotes the lowest achievable latency. Apart from the first four partitions, partition sizes are doubled every other partition, until a maximum partition size N_{max} is reached (Fig. 5.3).

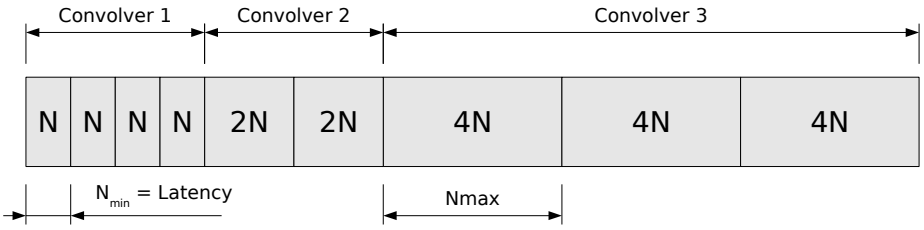

Figure 5.3: Impulse response partitioning scheme of the VEP application

5.1.2 Partition modules

Each partition size as shown in Fig. 5.3 is processed by an instance of `VEPConvolver`, that implements one binaural convolution operation for two or more partitions in the impulse response as described in 3.5.1.

Instances of `VEPConvolver` are initialized by a call to the method `VEPConvolver::init`:

```
bool init(
  // Interface function table
  InterfaceTable *ft,
  // scsynth handle
  World *world,
  // Smallest partition size N0
  size_t binSize,
  // Number of bins in this partition size
  size_t numBins,
  // Number of partitions in this convolver
  size_t numPartitions,
  // Offset in partitions from beginning of IR
  size_t partitionOffset,
  // Offset in frames from beginning of IR
  size_t irOffset,
  // Crossfade mode
  CrossfadeMode cfMode=CrossfadePartition);
```

Input samples are are passed to a convolver instance by a call to the method

```
void VEPConvolver::pushInput(float *src, size_t size);
```

`size` is assumed to be smaller or equal to the smallest partition size N_0, in this context also called "bin size". The input block is written to a circular buffer implemented by the template class `VEPRingBuffer<T>` in `src/VEPRingBuffer.h`. In order to avoid unnecessary copying of data, the ringbuffer is of size $4N_i$ where N_i is the partition size handled by the convolver. When the ringbuffer write pointer reaches an integer multiple of the partition size N_i, another N_i zeros are appended to the buffer for in-place consumption by the forward FFT routine.
The method

```
void VEPConvolver::compute(

    bool crossfade,

    fftwf_complex *ir,

    fftwf_complex *ir1,

    size_t irSize,

    size_t binIndex);
```

is performing the actual convolution operation for the partition size of the convolver. Depending on whether the convolver is currently scheduled for execution (see 5.1.4, `VEPConvolver::computeOneStage` is called to execute one of the two convolution stages.
The first stage includes computing the forward FFT with the current input buffer contents and performing the multiply-accumulate operation for the first half of the convolver's partitions.
The FFT transform used is provided by the highly efficient yet portable FFTW library [9] and expects frequency domain data to be in "half-complex" format. Spectra of real-valued input signals exhibit the Hermitian symmetry, making half of the spectrum redundant. Moreover, the coefficients at $k = 0$ and $k = \frac{N}{2}$ can be shown to be purely real, so that the spectrum can be stored in the same amount of space as the real input buffer. The half-complex format stores the real and imaginary parts of the spectrum separately

$$X[k] = r_0, r_1, r_2, ..., r_{\frac{N}{2}}, i_{\frac{N+1}{2}-1}, ..., i_2, i_1$$

such that the real part r_k of a coefficient is in $X[k]$ and the imaginary part

in $X[N-k]$, with the exception of the purely real coefficients at DC $X[0]$ and Nyquist $X[\frac{N}{2}]$. Input and output FFTs in `VEPConvolver` are performed in-place, overwriting the input array.

In order to increase cache efficiency when computing the complex multiply-accumulate and to make it easier to vectorize the MAC operation with SIMD instructions of the target CPU, the half-complex formatted data is shuffled into something more suitable by the routine `VEPFFT::shufflehc` in `src/VEPFFT.h`, storing sequences of four real values followed by the four corresponding imaginary values such that

$$\begin{aligned} X_{SIMD}[k] =& r_0, r_1, r_2, r_3, r_{\frac{N}{2}}, i_1, i_2, i_3, ..., \\ & r_{\frac{N}{2}-4}, r_{\frac{N}{2}-3}, r_{\frac{N}{2}-2}, r_{\frac{N}{2}-1}, i_{\frac{N}{2}-4}, i_{\frac{N}{2}-3}, i_{\frac{N}{2}-2}, i_{\frac{N}{2}-1} \end{aligned} \tag{5.1}$$

The converted input spectrum is directly written to the spectral delay line ringbuffer and the position of the current input spectrum is stored for later use by the complex multiplication.

After computing the forward FFT on the input block and clearing the accumulation buffer, the complex multiply-accumulate operations are performed for each partition handled by the convolver instance. The first partition is multiplied with the most recent input spectrum and later partitions are multiplied with the correspondingly delayed input spectra in the delay line. The complex multiplication itself is optimized by using SIMD vector instructions on platforms that support it, i.e. AltiVec and SSE2 [8] [12] in the routine `vep_vcmac_hc` in `src/VEPDSP.h`. The reorganized spectral data maps well to current vector unit layouts, and the implementation looks much like scalar code without unnecessary vector permutations. The multiplication is done for both binaural channels of the impulse response and accumulated into one buffer for each channel.

The last step in performing the convolution in `VEPConvolver::compute` involves execution of the inverse FFT on the spectral accumulation buffer. First, `VEPFFT::unshufflehc` in `src/VEPFFT.h` converts the accumulation buffer contents from SIMD to half-complex format into a temporary buffer, then the FFT is performed in-place. According to the overlap-add scheme in Fig. 3.2, the first half of the temporary buffer is mixed with the previously saved overlap tail into the output ringbuffer; the second half is then saved into the overlap buffer.

The overlap-save method for frequency-domain convolution would be slightly

more efficient, since the steps of saving the new overlap tail and mixing the old one into the currently process block are not needed [25, p.531], but overlap-save doubles the latency to $2N_0$, because the first N_0 samples of the convolution result have to be discarded due to aliasing effects and the input has to be delayed accordingly [25, pp.532]. The overlap-add method has the additional benefit with regard to non-uniform partitioning of not being encumbered by software patents [14].

As mentioned above, computation of input spectra, complex multiply-accumulate and output transform is split into two stages in `VEPConvolver`. Only convolvers with $N_{partition} = N_{bin}$, i.e. those with a computation period equal to the DSP processing block size, are handled specially in that all the steps above are performed in a single method call.

Finally, the method

```
void pullOutput(float* dst1, float *dst2, size_t size);
```

is called to transfer samples from the output ringbuffer to the calling unit. Convolvers with $N_{partition} = N_{bin}$ simply copy the output buffer data to the destination buffers, while convolvers with later partitions mix their output into previously stored content.

5.1.3 Impulse response buffer format

Impulse responses used by the convolution plugin are stored in `scsynth` buffer objects (see 4.7) and are expected to be pre-converted to the frequency domain according to the partitioning scheme employed. Partition spectra are stored in the same order as in the impulse response; each spectrum for partition size N_i takes N_i complex or $2N_i$ real values and is in the SIMD format specified in Eq. 5.1. The two channels of the binaural impulse response are stored non-interleaved, i.e. the partitions of the left channel are followed by those of the right channel (Fig. 5.4).

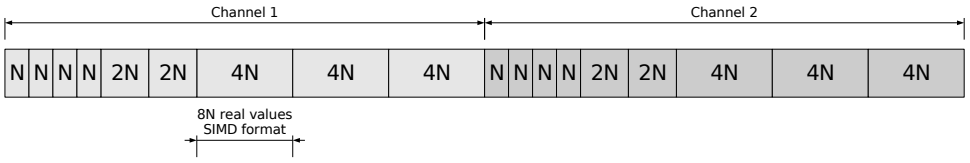

Figure 5.4: Impulse response buffer layout of transformed partitions

5.1.4 Partition scheduling

The uniform load algorithm described in [11] and the related process scheduler for a DSP implementation are not immediately mappable to a general purpose operating system context. None of the mainstream modern operating systems in use provide means of directly scheduling processes, but rely on higher-level abstractions. Müller-Tomfelde describes a portable implementation of non-uniform partitioned convolution based on multiple processes; unfortunately such a scheme doesn't map well to the SuperCollider process model, where sample buffers in particular are designed to be only ever accessed from one of two threads: the realtime audio thread and the non-realtime thread (Fig. 4.1). Instead of implementing a buffer management scheme on top of the SuperCollider API, it was decided to implement the convolution in a single process model.

Finding an optimal schedule online with a greedy algorithm is in general not possible. The process scheduler described by Gardner could theoretically be mapped to block based, single process convolution, but the main problem is apparent: every convolution operation is executed "in one go", i.e. it must complete within the time slot of one audio processing interrupt or callback. Priority scheduling based on the partition sizes and the corresponding deadlines produces a schedule that effectively needs to compute convolutions for many partition sizes in one processing period, because the convolution period boundaries are integral multiples of the smallest partition size N_0.

Instead it is desirable to interleave the computations in such a way, that the processing load is spread over periods as much as possible, while still meeting the deadlines of each convolution process.

Since the number of FFTs per convolution process is fixed – one forward and one backward transform is needed – and the number of complex MAC operations is limited to two spectra in the simplest case, each convolver processes its input in two stages (see 5.1.2). Since each convolver i needs to provide output every $N_i = B_i N_0$ samples, where N_i is the partition size, and B_i the number of bins in partition i, its `computeOneStage` method needs to be called twice every N_i samples.

The experimentally determined scheduling rule used in the VEP convolver is

$$(j - \frac{\tilde{B}_i}{2}) \bmod \tilde{B}_i = 0 \tag{5.2}$$

where $\tilde{B}_i = \frac{B}{2}$ is the scheduling frequency of the convolver i and $j \in [0, N_{max} - 1]$ is the index of the currently processed input block. The modulo

operation with argument $\tilde{B}_i$ ensures that each convolver is scheduled twice within its respective output period of N_i. The subtractive term $\frac{\tilde{B}_i}{2}$ spreads the computation of larger partitions apart from each other in such a way, that the maximum number of partition sizes processed in one period is three.

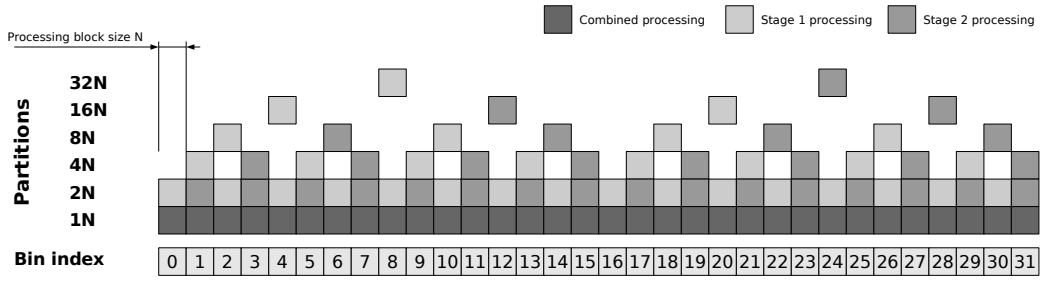

Figure 5.5: Convolver schedule for $N_{max} = 32N_0$

A schedule generated by Eq. 5.2 for a maximum partition size of $N_{max} = 32N_0$ is shown in Fig. 5.5. As can be seen, the convolver with partition size $N_i = N_0$ is handled specially in that it is scheduled in every block – the other partition size convolvers are scheduled twice within their respective processing period, once for each stage of the computation. Within one processing block, at most three convolver stages have to be executed, two for the partition sizes N_0 and $N_1 = 2N_0$ and one for another partition size N_j with $j > 1$. Because larger partition sizes are integral multiples of smaller ones, the whole schedule has a periodicity of $32N_0$.

5.1.5 Coefficient exchange and crossfade

Binaural rendering in a virtual reality environment usually involves dynamic update of FIR coefficients in response to user or other external input, transforming the LTI convolution system into a linear time variant system.

In order to avoid artifacts when switching between FIR coefficient sets or impulse responses, one possible approach is to compute two LTI systems corresponding to the two impulse responses that should be exchanged, and crossfade in the time domain between the filter outputs.

The computational cost of the convolution is doubled, but the increase in load can be reduced to just the time of the crossfade [23].

Due to the block based processing in frequency domain convolution, filter coefficients for a particular convolver i that handles partitions of size N_i can only be exchanged with a minimum delay d_i of

$$N_i < d_i < 2N_i \tag{5.3}$$

because a change in coefficients in block j only takes effect in the block $j+1$ after the triggering of the crossfade, and block $j+1$ still has an output latency of N_i.
Since the different convolvers have different partition sizes and thus different minimum and maximum exchange delays, a trade off has to be made between a simultaneous crossfade at the output of all modules and minimization of response time. [23] proposes two exchange strategies:

- trigger at the same time, crossfade at different times
- trigger at different times, crossfade at the same time

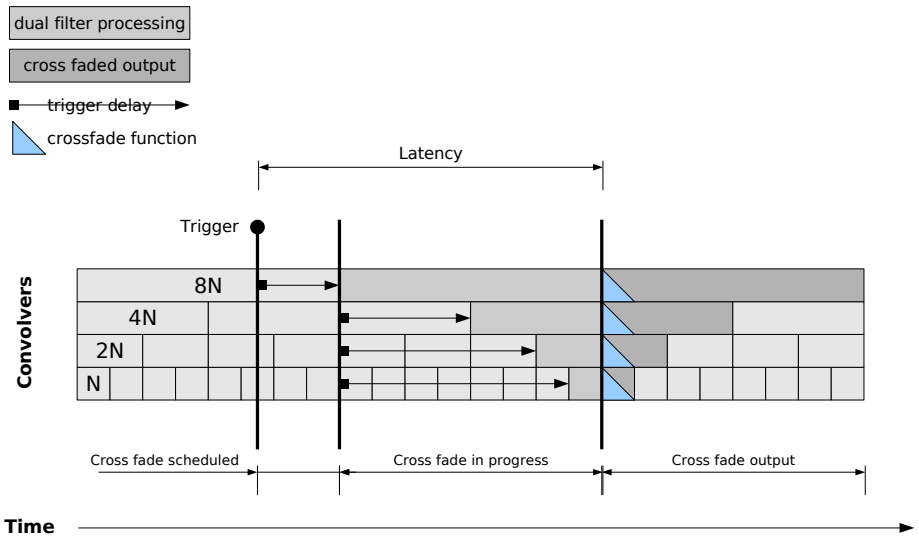

Figure 5.6: Crossfade scheduling logic for dynamic filter coefficient exchange

`VEPConvolution1` employs the second approach, shown in Fig. 5.6. Triggers are scheduled until the next block boundary that coincides with the largest output block boundary. Crossfade time is minimized with this approach, but the exchange delay can be up to $2N_{max}$.

The crossfade logic is encapsulated in the utility class `VEPCrossfade` in `src/VEPConv.cpp`, so that different exchange strategies can be integrated relatively easily.

During the time of the crossfade in `VEPConvolver`, two LTI systems[1] have to be computed, but the input spectra can be shared by both filters, so that only the costs of the output FFT and the complex MAC operations are doubled. The overall increase in load is automatically distributed in time across processing blocks by the partition scheduling algorithm (5.1.4).

`VEPConvolver` implements two linear crossfade functions, termed *full fade* and *primary block fade* in [23], so that different triggering and cross fading strategies can be combined.

5.1.6 Computational complexity

The cost of the partitioning scheme employed is that of the uniform load scheme described in 3.5.2, with the additional cost of two partition sizes of N (see Eq. 3.11):

$$\begin{aligned} O(N) = O(&8(\log_2(N)+3)+ \\ &2[\log_2(\frac{L}{2N})]^2 + \log_2(\frac{L}{2N})[4\log_2(N)+10]) \end{aligned} \tag{5.4}$$

Research carried out by García in [10] indicates that the modified uniform load scheme does not necessarily lead to the most efficient partitioning possible, because the cost function can have several minima in between, that can be found with dynamic programming techniques, when brute-force search is not feasible.

In the context of the VEP application, however, with a maximum impulse response size of 1.6s for the reverb tail, the system load was found to be sufficiently low even at small block latencies of 64 samples at 44.1kHz.

Tab. 5.1 shows the measured runtime and CPU load obtained on the installation system, a dual Xeon 3.2GHz, for different impulse response sizes and a single convolution process. The runtime of the convolution was measured in the processing function of the VEP plugin and includes the costs for data copying, FFT and frequency-domain multiplication. The CPU load denotes

[1] Four, in fact, counting two convolutions per binaural channel.

IR length	$N_{max} = 64$		$N_{max} = 8192$	
	time μs	load %	time μs	load %
8192	46	5.3	47	5.2
32768	227	17.9	72	7.3
65536	607	45.5	78	7.6
131072	1211	85.0	81	7.9

Table 5.1: Convolution runtime measurements for $N_0 = 64$

the overall system load as displayed by the `top` command and should only be regarded as a qualitative approximation.

Uniform partitioning with $N_0 = N_{max} = 64$ is compared with the non-uniform partitioning scheme from 5.1.1 with $N_0 = 64$ and $N_{max} = 8192$. In both cases the latency is set to $\frac{2N_0}{f_s} = \frac{128}{44100}s = 2.9s$. Impulse response lengths are varied between 8192 frames (0.186s) and 131072 frames (2.97s) and are constrained to powers of two to avoid processing overhead arising from partitioning[2].

For small impulse response lengths the two partitioning schemes perform almost equivalently, but for increasing impulse response lengths, non-uniform partitioning performs significantly better. The measured runtime and load of uniformly partitioned convolution increases almost linearly with impulse response length L, which coincides with Eq. 3.11, where the dominating term is $\frac{L}{N}$ for large L. In the case of non-uniform partition sizes, the runtime cost increases approximately logarithmically with L, because $O(N)$ in Eq. 5.4 is dominated by $[\log_2(\frac{L}{2N})]^2$ for large L.

A near optimal partitioning for large impulse responses can be found through experimentation, by varying the two constraining parameters latency $d = N_0$ and maximum partition size N_{max}. Specifically by changing N_{max} the trade off between the cost for the FFTs, that largely depends on the implementation used, and that of the complex multiply-add can be adjusted.

5.1.7 Impulse response tail

Due to restrictions in computing time for the simulated impulse responses of a few weeks, only the first 0.13s of each binaural impulse response were actually rendered, while the statistical reverb tail was computed only once.

[2]Impulse responses are zero-padded to fit the sum of partition sizes dictated by the partitioning scheme.

The impulse response head is faded out at the end within 20ms using a half-cosine window, the reverb tail is faded in appropriately (Fig. 5.7).

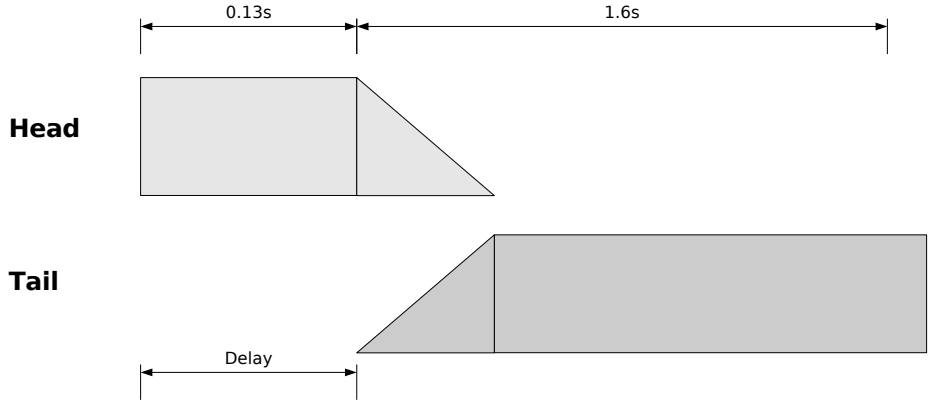

Figure 5.7: Separation of impulse response head and tail

A monophonic sum signal of all three input tracks is appropriately delayed (according to the start of the tail in the impulse response) and fed to the reverb tail convolver synth. The binaural stereo output is then mixed into the final output stream with an adjustable amplitude level.

5.2 Disk access plugin

The disk access plugin is implemented as an asynchronous server command that loads binaural impulse responses from sound files on disk and has the following arguments:

```
/cmd "vep" "read"

     BUFFER_INDEX        int32

     FILE_NAME           string

     FILE_OFFSET         int32

     NUM_FRAMES          int32
```

```
CROSSFADE          int32
```

Sound files are accessed using the ubiquitous *libsndfile* library [5]. Impulse response files are expected to be stereo file in any of the formats libsndfile can read, e.g. 16 bit PCM files or 32 bit floating point files. No resampling is provided – the files should match the sampling rate of the audio hardware.

Asynchronous commands (see 4.6) start in the realtime audio thread, are then transferred to the non-realtime thread to do their work and deliver a notification of their completion back to the realtime thread.

The class `VEPReader` encapsulates the process of reading the sound file, deinterleaving and partitioning its contents according to the global parameters $N_{min} = N_0$ (derived from the `scsynth` block size) and N_{max} (passed to the `init`) command) and transforming the individual partitions into the frequency domain, according to the scheme shown in Fig. 5.4.

To maximize throughput and disk utilization the relatively small impulse response files – of about 40kb in size – are read into memory in a single call to `sf_readf_float`. The optimal reading block size remains to be determined for bigger files, although it is to be expected, that in the case of non-competing access, the whole-file approach is the most efficient in relation to operating system buffering and scheduling strategies.

The two SATA harddisks used on the installation machine are attached to a hardware RAID controller, that is configured for RAID-0 where disk blocks are stored interleaved ("striped") on alternating disks. The file system used is XFS [28], which is optimized for streaming multimedia data, but the performance was not compared to other filesystems, such as ReiserFS.

In order to evaluate impulse response load times, impulse responses of 0.13s were loaded randomly from a single speaker BRIR set and the time between initiating the load request and receiving the request notification (overall load time), as well as the times to load the impulse response from disk and to perform the frequency domain transformation were measured. The measurements were averaged over a 50 runs of 1000 iterations and for the transformation non-uniform partitioning was used with $N_0 = 64$ and $N_{max} = 8192$.

The average worst case overall load time determined is 0.036s, while the average overall load time is 0.0058s. The average amount of time spent in reading the file from disk (`sf_readf_float`) is $60 \cdot 10^{-6}$s and the average time needed for converting the impulse response to the frequency domain is $520 \cdot 10^{-6}$s.

The measurements indicate that during impulse response loading the vast amount of time is spent in network and inter-thread communication as de-

scribed in 4.6 necessary for maintaining cache consistency in the SuperCollider language client, i.e. for ensuring that buffers are only added to the impulse response cache when they are already filled with valid data. Actual disk load times are significantly lower for the impulse response length used.

5.3 Impulse Response Cache

Since the amount of impulse response data used by the VEP application – about 75GB – is too large to fit into the main memory of current mainstream computers, impulse response files need to be loaded from disk on demand. Disk loads are both relatively slow and also non deterministic in their completion time, so it is desirable to keep as many impulse responses in main memory as possible, based on the listener's current head position and possible future head movements.

For this purpose the VEP application maintains a *least-recently-used-cache* that caches impulse response buffer contents for future use.

The `sclang` class `VEPBufferCache` maintains a pool of integer indexes of buffer objects that are preallocated on the synthesis server. The cache is indexed by position keys, 32 bit integers composed from the horizontal and vertical head orientation angles φ_x and φ_y and quantized to 1° steps:

```
(int(x) % 360) << 16) | (int(y) % 360)
```

`VEPBufferCache` delegates the cache logic to an instance of `VEPLRUCache`, that maintains an identity-hashed associative array (`IdentityDictionary`) and a doubly linked list of cache nodes. When a key-value association is inserted into the cache, the dictionary holds to the mapping of key to cache node while the cache node itself holds the actual object to be cached. When inserting a key or accessing the cache, the cache node corresponding to the key is unlinked and transferred to the beginning of the queue – due to the doubly linked list a constant time operation. The `pop` method removes the last element from the cache node queue and removes its associated key from the cache.

Because of the asynchronous communication with `scsynth` a buffer id in `VEPBufferCache` can be in one of three states: *free*, *reserved* and *used*. When a particular impulse response, uniquely identified by φ_x and φ_y, is not cached, a new buffer id is allocated from the pool or popped from the cache and enters the *reserved* state. As soon as the corresponding sound file has been read by

the disk access plugin, the buffer id is entered into the cache and assigned the *used* state.

Instances of the class `VEPTrack`, an abstraction of one individual tape track, hold to an instance of `VEPBufferCache`, which is updated regularly in response to head tracker input.

Apart from the current user input comprising head rotation and elevation, `VEPTrack` asynchronously spawns new impulse response read requests in order to increase the cache hit rate for head positions in the proximity of the current position.

Currently, a simple algorithm is used: based on the current head orientation $p = (\varphi_x, \varphi_y)$, the quantization step size $d = (d_x, d_y)$ and a range parameter N_r the state space around p is explored to yield a set of prospective positions to load into the cache:

$$\tilde{p} = \{(\tilde{\varphi}_x, \tilde{\varphi}_y) \leftarrow \tilde{\varphi}_x \in [-N_r d_x, N_r d_x], \tilde{\varphi}_y \in [-N_r d_y, N_r d_y]\} \tag{5.5}$$

By disregarding those positions already present in the cache, only positions in the direction of the listener's head movement are explored (Fig. 5.8).

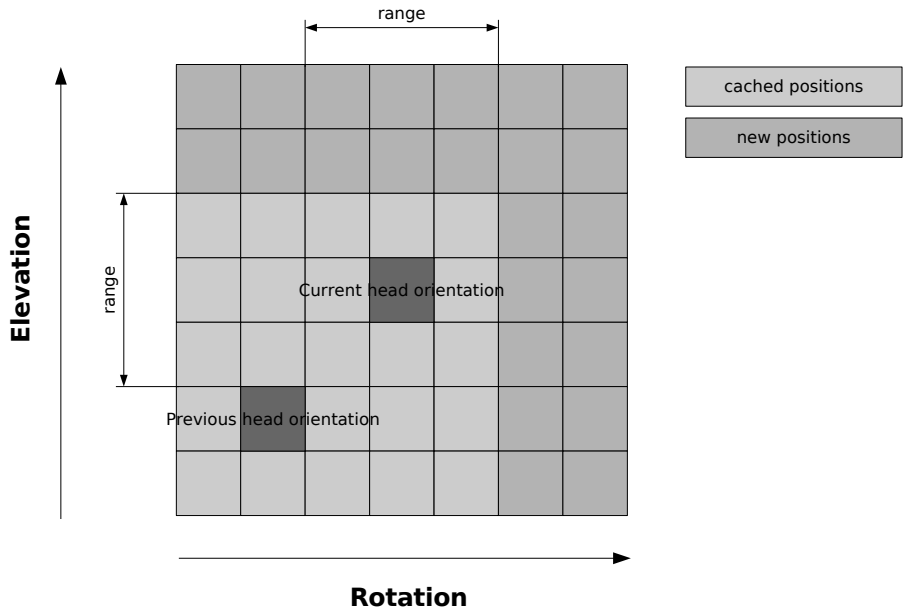

Figure 5.8: Asynchronous exploration of listener head orientations

Experimentation shows that the disk load process is congested with rather low values of N_r, which can be attributed to the time of almost 6ms it takes from initiating an impulse response load and getting the completion notification (see 5.2). New requests are queued sequentially in `scsynth`'s asynchronous command queue and cannot be processed before completion of earlier requests. Since every head movement potentially triggers new asynchronous requests, user induced head movements face an increasing and non-deterministic latency until they are actually performed.

5.4 Tape player and LFE signal

`VEPPlayer` encapsulates the three tracks of the original reconstructed tape transferred to digital audio files. The tape tracks are loaded into memory and played back simultaneously to three adjacent audio buses on the server. The buses are read by the convolution synth (see 5.1) and the synth node responsible for generating the low frequency effects (LFE) signal, that is derived from the mono sum of all tracks fed into a lowpass filter with configurable cutoff frequency.

5.5 Score Player

The control score in the VEP application specifies the time instants each track should be rendered on another speaker or speaker group.

The score file is a file containing `sclang` statements, and consists of five parts: one control score for each track and one LFE score.

Score lines are arrays of the form

onset time	*onset duration*	*speaker category*	*speaker group*

Currently the parameters *onset duration* and *speaker category* are ignored for tape tracks, because each track is assumed to be audible on exactly one speaker or speaker group at a time.

In the LFE score however, the duration parameter is interpreted; the LFE signal is audible only at the times and for the duration specified in the score.

Upon application startup the score file is read, interpreted by `sclang` and compiled into a schedule, an instance of the library class `Routine`, that can be played back directly on a `TempoClock`.

Tape track score entries are passed at the appropriate times to the respective `VEPTrack` instance, which clears the buffer cache and subsequently uses impulse responses from the set of the new speaker group.

5.6 Tracker Interface

The tracker interface in the VEP application is based on UDP/IP networking and the OpenSoundControl protocol (see 4.3). The message used is `/tracker` and expects two arguments, φ_x and φ_y in degrees.

The application is adaptable to different tracking devices, by assigning the class variable `trackerMessageHandler`, that gets passed the OSC message as an argument, and returns a two-element array with the appropriately converted head orientation coordinates.

One tracker device used in the installation is an *InterSense Inertia Cube3* with an update frequency of 60Hz. The example program contained in the distribution was used to send the tracker data to one or more network addresses passed to the program on the command line, with a configurable maximum update rate.

Chapter 6

Discussion

In the preceding chapters a binaural convolution engine has been presented that addresses particular constraints encountered in the VEP reconstruction project, such as handling sets of binaural room impulse responses for a large number of sources.

The partitioned convolution algorithm used is suited in performance to the impulse response lengths used in the VEP application, as indicated by the measurement results determined in 5.1.6. Still, in order to accommodate for a larger number of simultaneously sounding sources or longer impulse responses, the convolution process can be improved in various ways.

In [10] an algorithm for finding an optimal non-uniform impulse response is presented, that can reduce the number of multiply-adds of the convolution to one third in comparison to Gardner's uniform load scheme (which is very similar to the one used in the VEP application), depending on the particular FFT implementation used. The computational complexity of the Viterbi search algorithm used in [10] might be too high to be useful in a realtime context, but in the VEP application an optimal partitioning could be computed once during initialization, because the impulse responses are all of the same length. The impact of a different partitioning scheme on the two-stage load distribution model described in 5.1.4 is unclear, however.

A further improvement would be to abandon the single-process partition scheduling strategy and distribute the different convolvers among several processes. Meeting the scheduling deadlines needs to be enforced by careful assignment of thread priorities to the different convolver tasks according to

their computation period, as described in [2] and [23], a method that can be regarded as a mapping of Gardner's process scheduler for DSP platforms presented in [11] to general purpose operating systems.

Improving buffer cache performance is crucial when the complete set of impulse responses does not fit into main memory. The measurements obtained in 5.2 indicate, that the main source of overhead in overall impulse response load time is in the round-trip communication needed in order to ensure cache coherence in the SuperCollider client application.

Putting the buffer cache in the same address space as the convolution and disk access processes should help minimize delays induced by network communication. Optionally, impulse response load time can be further reduced by transferring the frequency domain transformation into the realtime audio thread, thereby trading parallelism in execution for lower disk thread response times. By using a lossless compression format such as FLAC [7] disk load times could be reduced especially for larger impulse responses – a single set of impulse responses can be compressed to 40 percent of the original size with the most aggressive setting.

It became evident in 5.3, that the fine granularity of asynchronous cache updates based on a resolution of $(\Delta_{\varphi_x}, \Delta_{\varphi_y}) = (1°, 5°)$ in combination with the sequential handling of asynchronous requests by the `scsynth` engine poses a restraint on background prefilling of the buffer cache with nearby head positions.

Asynchronous cache updates need to be prioritized according to their relevance to the current listener head orientation and must be removable from the command queue to be able to discard obsolete requests.

The fine granularity of cache prefilling is unsuitable for very fast head movements to diverse positions. In [27] various possibilities for interpolation of binaural room impulse responses are examined, and it seems beneficial to combine the impulse response cache with a method of interpolating between cached positions. The asynchronous prefill granularity could be coarser for fast head movements by only preloading some nearby locations as grid points for interpolation and refining the cache resolution as time permits.

In summary, the work presented is an integral part of the VEP reconstruction effort, with a new approach to handling large sets of binaural impulse responses in virtual acoustics. The impulse response cache in particular helps reducing the memory requirements of binaural room simulations with many sources and provides a promising foundation for future developments.

Bibliography

[1] Acoustic Design Ahnert, *EASE room modeling software.*
`http://www.ada-acousticdesign.de/german/ease/index_ease.html`

[2] Adriaensen, F. "Design of a Convolution Engine optimised for Reverb", *LAC2006 Proceedings*, 4th International Linux Audio Conference, Karlsruhe, 2006, 49-53.

[3] Cooley, J.W.; Tukey, J.W. "An Algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computation vol. 19*, 297-301, 1965.

[4] Davis, P. et al. *JACK Audio Connection Kit.*
`http://www.jackaudio.org/`

[5] de Castro Lopo, E. *C library for reading and writing files containing sampled sounds.*
`http://www.mega-nerd.com/libsndfile/`

[6] Dobson, R.; Fitch, J.; Tazelaar, K; Valle, A; Lombardo, V. "Varèse's Poème Electronique regained: Evidence from the VEP project", *Proceedings of the ICMC*, Barcelona, 2005.

[7] *Free Lossless Audio Codec.*
`http://flac.sourceforge.net/`

[8] Freescale Semiconductor, *AltiVec Technology Programming Interface Manual*, Rev. 0, 1999.
`http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf`

[9] Frigo, M.; Johnson, S.G. "The Design and Implementation of FFTW3", *Proceedings of the IEEE 93 (2)*, 2005, 216-231.

[10] Garcia, G. "Optimal filter partition for efficient convolution with short input/output delay", *Audio Engineering Society Paper 5660*, 113th AES Convention, Lose Angeles, 2002.

[11] Gardner, W. "Efficient convolution without input-output delay", *Journal of the Audio Engineering Society*, 43(3):127-136, 1995.

[12] Intel Corporation, *Intel Architecture Optimization Reference Manual*, 1999.
`http://www.intel.com/design/PentiumII/manuals/245127.htm`

[13] Kersten, S. "SuperCollider 3 Server on Linux — A Status Report", *Linux Audio Developer Conference*, Karlsruhe, 2004.
`http://space.k-hornz.de:8888/space/uploads/lac04_sc3_slides.pdf`

[14] *Digital filter having high accuracy and efficiency*, European Patent Specification, EP 0 649 578 B1, 2003.

[15] Lindau, A. *Ein Instrument zur softwaregestützten Messung binauraler Raumimpulsantworten in mehreren Freiheitsgraden*, Master Thesis, Institut für Kommunikationswissenschaft, Technische Universität Berlin, 2006.

[16] Lombardo, V.; Fitch, J.; Weinzierl, S.; Starosolski, R. et al. "The Virtual Electronic Poem (VEP) Project", *Proceedings of the ICMC*, Barcelona, 2005.

[17] McCartney, J. "SuperCollider: A New Real-time Synthesis Language", *Proceedings of the ICMC*, Hong Kong, 1996.

[18] McCartney, J. "A New, Flexible Framework for Audio and Image Synthesis.", *Proceedings of the ICMC*, Berlin, 2000

[19] McCartney, J. "Rethinking the Computer Music Language SuperCollider", *Computer Music Journal*, Vol. 26, 2002, 61–68

[20] Mills, D.L. "Internet time synchronization: the Network Time Protocol.", *IEEE Trans. Communications COM-39, 10*, 1991, 1482-1493.

[21] Moldrzyk, C.; Ahnert, W.; Feistel, S; Lentz, T; Weinzierl, S. *Head-Tracked Auralization of Acoustical Simulation*, 117th AES Convention Paper, San Francisco, 2004.

[22] Ingo Molnar, *Realtime preemption Linux kernel patch.* `http://people.redhat.com/~mingo/realtime-preempt/`

[23] Müller-Tomfelde, C. "Time varying filter in non-uniform block convolution", *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, Limerick, 2001.

[24] Oppenheim, A.V.; Schafer, R.W. *Zeitdiskrete Signalverarbeitung*, 3. Auflage, R. Oldenbourg Verlag, München, 1999.

[25] Orfanidis, S.J. *Introduction to Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[26] Petit, J. *Le Corbusier. Le Poème Electronique.*, Edition Le Minuit, Paris, 1958.

[27] Pfundt, T. *Überblendung von Impulsantworten in der virtuellen Akustik*, Diploma Thesis, Institut für Kommunikationswissenschaft, Technische Universität Berlin, 2006.

[28] Silicon Graphics, Inc. *XFS: A high-performance journaling filesystem.* `http://oss.sgi.com/projects/xfs/`

[29] Torger, A.; Farina, A. "Real-time partitioned convolution for ambiophonics surround sound", *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, New York, 2001.

[30] Varèse, E. "The Liberation of Sound", in Borety, Benjamin e Cone, Edward T. (eds.) *Perspectives on American Composers*, NY, Norton, 1971, 25-33.

[31] Vercoe, B. *Csound: a manual for the audio processing system.*, Massachusetts Institute of Technology Media Lab, Cambridge, MA, 1996.

[32] Vorländer, M. *Virtuelle Akustische Realität*, Institut für Technische Akustik, RWTH Aachen, 2003.

[33] Wright, M.; Freed, A.; Momeni A. "OpenSound Control: State of the Art 2003", *Proceedings of the Conference on New Interfaces for Musical Expression (NIME-03)*, Montreal, 2003, 153-159.

Appendix A

Glossary

ADC	Analog to Digital Converter
API	Application Programming Interface
BRIR	Binaural Room Impulse Response
DAC	Digital to Analog Converter
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
GNU	GNU is Not Unix
GPL	GNU General Public License
HRIR	Head Related Impulse Response
HRTF	Head Related Transfer Function
ICMC	International Computer Music Conference
LFE	Low Frequency Effects
LRU	Least Recently Used
LTI	Linear Time-Invariant
MAC	Multiply ACCumulate
MADD	Multiply ADD
MIDI	Musical Instrument Digital Interface
NTP	Network Time Protocol
OSC	OpenSoundControl
RAID	Redundant Array of Independent Disks
RS232	Recommended Standard 232 (serial data interface)
SIMD	Single Instruction Multiple Data
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UGen	Unit Generator
VEP	Virtual Electronic Poem

Appendix B

Program Structure

B.1 Hardware and Software requirements

The VEP application currently runs only under Linux, but should be easiliy portable to MacOS X, since the synthesis engine and the SuperCollider language already run on both platforms. As soon as SuperCollider matures on Windows, there will also be the option of porting the application to that platform.
On Linux, SuperCollider requires the following programs and libraries for operation:

- the *libsndfile* library [5]
- the JACK audio connection kit as the main audio driver backend [4]

Additionally, the VEP plugin requires the FFTW fast fourier transform library [9].
The VEP application was developed on a dual Intel Xeon 3.2 GHz machine, with 4GB of RAM and a hardware RAID-0 configuration with two 400 GB SATA disks. Processor speed was found to be a minor issue with the relatively short impulse responses used, but disk subsystem throughput is crucial for the operation of the impulse response cache.

The operating system used is a 64 bit Debian Linux distribution; due to the workings of the `sclang` interpreter, however, the SuperCollider language application has to be built in a separate 32 bit subsystem (*chroot*).

Although Linux kernel support for realtime applications has improved quite a bit since the 2.6 series, the situation is still not optimal, especially with regard to concurrent use of realtime threads and disk access. On the VEP development machine, Ingo Molnar's *realtime-preempt* kernel patch [22] has been applied; it modifies certain kernel code paths to be preemptable by realtime processes and thus reduces interrupt scheduling latencies considerably.

B.2 Components

Tab. B.1, Tab. B.2 and Tab. B.3 list the various files encompassed in the VEP distribution.

B.3 Configuration file format

The configuration file is comprised of standard `sclang` statements, and shall be exemplified in the following.

B.3.1 Literals

Character strings are enclosed in double quotes:

```
"Example␣string"
```

Numbers can be written in any of the common formats:

```
1

1.5

1e3
```

```
1e-9
```

Literal lists or arrays are enclosed by square brackets, where the elements are separated by commas:

```
[1, "string␣element", 2]
```

B.3.2 Configuration variable assignment

Configuration variables are class variables in the class VEP, and are accessed with the *dot* notation. Assignment to a left hand assignable expression is done with the assignment operator `=`. Statements, including assignment statements, are terminated by a semicolon:

```
VEP.configVariable = "string␣value";
```

B.3.3 Expressions

Since the configuration file contains plain SuperCollider code, more advanced uses of the configuration file involve more complex expressions. Although the language is quite transparent and very similar in syntax to Java or `C`, the user is referred at this point to the extensive documentation provided in the SuperCollider distribution, particularly the help files `Literals.rtf`, `Method-Calls.rtf` and `Classes.rtf`.

B.4 Configuration variables

The following table provides a list of configuration variables that control the behavior of the VEP application.

Variable	Type	Description
`binSize`	`Integer`	Minimum partition size, determines latency

Variable	Type	Description
maxPartSize	Integer	Maximum partition size
headReponseFrames	Integer	Number of frames to read from the beginning of impulse response head
tailResponseFrames	Integer	Number of frames of the impulse response tail
tailSoundFile	String	Path to the impulse response tail sound file
soundFileDir	String	Path to the base directory containing impulse response files
soundFilePrefix	String	String prefix for sound files (concatenated with the encoded position)
sampleFiles	Array	Array of paths to the input sound files
scoreFile	String	Path to the control score file
xSpec	ControlSpec	Specification of horizontal head orientation range and quantization
ySpec	ControlSpec	Specification of vertical head orientation range and quantization
numBuffers	Integer	Size of the impulse response cache for each track
explore	Boolean	Wether cache should be filled asynchronously
explorationRange	Integer	Number of horizontal and vertical steps to explore asynchronously
trackerMessageHandler	Function	Filter function for converting head tracker input
recordTrackerEvents	Boolean	Wether tracker events should be recorded and written to disk
speakerMap	Dictionary	Dictionary for optional speaker name remapping
debug	Dictionary	Dictionary mapping debugging symbols to boolean flags, valid symbols are \osc, \tracker, \score

src/VEPConv.h	Declaration and implementation of VEPResponse, VEPConvolver, VEPCrossfade and VEPReader
src/VEPConv.cpp	
src/VEP.h	Utility template and inline function implementations
src/VEP.cpp	Unit and plugin command implementations
src/VEPDSP.h	Scalar and vectorized implementations of the complex multiply-add operation
src/VEPFFT.h	Declaration and implementation of FFTW interface functions and data shuffling routines
src/VEPFFT.cpp	
src/VEPRingBuffer.h	Ringbuffer template implementation
tracker/main.c	Serial-to-OSC tracker interface
lib/VEPBufferCache.sc	Buffer cache implementation
lib/VEPConvolution.s	Unit generator interface classes
lib/VEPLFE.sc	LFE signal synth abstraction
lib/VEPLRUCache.sc	LRU-cache implementation
lib/VEPPlayer.sc	Tape playback synth abstraction
lib/VEPProgressDisplay.sc	Console progress display
lib/VEPResponse.sc	Impulse response and partitioning abstraction
lib/VEP.sc	Main entry point and application logic
lib/VEPTail.sc	Reverb tail convolution synth abstraction
lib/VEPTrackerEventRecorder.sc	Recorder for OSC tracker events
lib/VEPTrack.sc	Tape track convolution abstraction

Table B.1: Source file index

`bin/scsynth.sh`	`scsynth` wrapper script
`bin/vep`	Main VEP application script
`bin/vep-init`	VEP initialization script (needs superuser priviledges)
`etc/hdsp.mix`	Configuration file for the HDSP soundcard mixer application
`scripts/app.sc`	Main VEP configuration file

Table B.2: Script and configuration file index

`scores/score_full.txt`	Raw reconstructed text input score
`scores/score.sc`	Compiled input score for the VEP application
`scores/split_score.rb`	Compiler script for score preprocessing
`sounds/pe_track1.wav`	Soundfiles of digitized original tape recordings
`sounds/pe_track2.wav`	
`sounds/pe_track3.wav`	

Table B.3: Sound file and control score index