Enhancing a Virtual Reality Game through Sonic Interaction and Adaptive Music

Martin Maunsbach

Master's Thesis

AALBORG UNIVERSITY
STUDENT REPORT

Department of Architecture, Design, and Media Technology
Aalborg University Copenhagen
http://www.aau.dk

Title:
Enhancing a Virtual Reality Game through Sonic Interaction and Adaptive Music

Theme:
Virtual Reality Musical Instruments
Adaptive Music

Project Period:
Spring Semester 2019

Participant:
Martin Maunsbach

Supervisor:
Stefania Serafin

Page Numbers: 81

Date of Completion:
May 28, 2019

Abstract:

This thesis researches whether sonic interaction and adaptive music can enhance a Virtual Reality beat slicing game. The sonic interactions take the form of Virtual Reality Musical Instruments and are based on knowledge of existing physical instruments and synthesis techniques. Adaptive music techniques are explored and appropriate uses created. A base game of an already existing game concept is create in the Unity game engine alongside the sonic interactions and adaptive music. The features are subjected to user testing and discussed. Future improvements are suggested.

AALBORG UNIVERSITET
STUDENTERRAPPORT

Institut for Arkitektur, Design og Medieteknologi
Aalborg Universitet Koebenhavn
http://www.aau.dk

Titel:
Forbedring af et Virtual Reality spil med sonisk interaktion and adaptiv musik

Tema:
Virtual Reality Musik Instrumenter
Adaptiv Musik

Projektperiode:
Foraarssemestret 2019

Deltager:
Martin Maunsbach

Vejleder:
Stefania Serafin

Sidetal: 81

Afleveringsdato:
28. maj 2019

Abstract:

Dette projekt undersoeger hvorvidt sonisk interaktion og adaptiv musik kan forbedre et Virtual Reality beat skaerende spil. Den soniske interaktion er lavet med Virtual Reality musik instrumenter og er basered paa viden om eksisterende instrumenter og syntese tekniker. Adaptiv musik tekniker er undersoegt og passende brug er af dem foregaar. Et grundspil af et allerede eksisterende spil koncept er lavet i Unity game engine sammen med den soniske interaktion og adaptiv musik. Funktionerne er testet og diskuteret. Fremtidige forberedninger er foreslaaet.

Contents

Preface

This project perfectly sums up my work from the last four semesters. It grew from a static MatLab impact synthesis in the first semester, to a usable dynamic Unity implementation in the second, to Virtual Reality in the third semester and now combining all in this project. Also, a guitar lightsaber.

Thank you, Stefania Serafin. For guidance and support with this project, for help with and article, opportunities at conferences and, most importantly, for making a degree in Sound & Music Computing possible. Grazie mille.

Aalborg University, May 28, 2019

[illegible]

Martin Maunsbach
<mmauns17@student.aau.dk>

Chapter 1

Introduction

1.1 Project Description

This thesis combines the areas of game development and sound synthesis by exploring audio features for an interactive Virtual Reality (VR) application. The application is based on an already existing concept where players use gestures to interact with 3D cube-like objects to the rhythm of a linear song. Apart from an impact sound effect, no music is created from the gestures. The aim of this thesis is to research and test additional contributions to make the concept more musically interactive.

The first contribution is to add sonic interaction with the creation of Virtual Reality Musical Instruments (VRMIs). By designing a VRMI inside an existing concept, the instrument should conform to the aesthetics and design of the game. This creates restrictions on the visual aspect, physical controller, input modalities and soundscape. The instruments should also fit within the context of the game by enhancing the experience in a new way and not as a separate experience with the same design aesthetics.

The second contribution is adaptive music. Instead of a linear song that always progresses the same way from beginning to end, it can change depending on actions by the user. Adaptive music can be used in many ways. In this project it is used to change parts of a song depending on the actions of a user and to transition another song to create an endless experience.

To evaluate the contributions, the game is created from the ground up in the Unity game engine and is not a modified version of the original game. The mechanics and overall control of the game is coded using C# and the digital signal processing (DSP) is implemented using C++ for efficiency. JSON files are used to store the information on what gestures the user must produce. To have seamless transitions with adaptive music, the audio middleware FMOD is used. The usability of the contributions are evaluated and discussed.

1.2 Motivation and General Research

Basing the VRMI on an existing framework has led to the creation of original instruments instead of virtual counterparts to existing instruments. This VR-first approach is chosen to adopt the existing user-friendly interactions within the VR environment as opposed to creating new ones. There are three main motivations behind the VRMI. Firstly, it is to create an instrument that is rooted in VR. This is to make the instrument approachable by utilizing mechanics players already are familiar with, but also by not using custom hardware and instead relying on controllers available for most head mounted displays (HMDs). Secondly, it is done to increase the interactivity of the game and let the user affect the soundscape. The freedom to create music while playing can lead to a new experience on every play-through. Thirdly, by letting the user create music, it can be a gateway to musicianship for players . This was observed with other musical rhythm games like Guitar Hero [1, 2]. The game inspired users to "explore new ideas and concepts" [3].

For the adaptive music, the motivation is different for version with adaptive segments and the endless version. The goal of the version with adaptive segments is to observe if it increases the replayability of a song. By adding segments the user can unlock by a better in-game performance, it should be motivating by the sheer power of curiosity, as the segments otherwise will remain secret. The endless mode is not to increase replayability, but to keep the user playing, making it useful for players who use the game as a workout routine.

Chapter 2

Background

2.1 Game Concept

The game concept is based on the VR game Beat Saber by Beat Studios [4]. The concept of the game revolves around a user making gestures in Virtual Reality while a linear song is played back. By slicing small cubes as they are coming towards the player, the player accumulates points that can be used to obtain a high score in an arcade-style setting. To slice a cube, the player is equipped with sabers resembling the lightsabers from Star Wars. In the default setting, the left hand is a red saber while the right is a blue saber. How to slice the cube is varied by which hand to use as well as the direction of the slice. The cube and the saber slicing it must match in color and a directional symbol on the cube must point in the same direction of the slice. Apart from the small cubes, obstacles like walls and bombs are also present. The timing of each cube and obstacle is preset to match the composition of a song by creating a beat mapping of the objects.

2.2 Virtual Reality Music Instruments

As the name implies, Virtual Reality Musical Instruments are instruments created for use inside a VR environment. Although VRMIs emerged in some form in the beginning of the millennium [5, 6], the accessibility to VR granted by its resurgence through higher quality head mounted displays has led to the creation of new and exciting VRMIs.

2.2.1 Extended Possibilities with VR

One of the exciting aspects of VRMIs are the extended possibilities they can provide. An aspect of Virtual Musical Instruments (VMIs) is the possibility to extend sound beyond what is otherwise achievable. The same is valid for VRMIs, while

VRMIs also has the potential for a more immersive experience due to seeing everything in 3D and being inside the interactive environment with VR. Serafin et al. described the expanded possibilities as "magical" interactions as opposed to natural interactions stating [6]

> "[...] an interaction or an instrument will qualify as magical if it is not limited by real-world constraints, such as the ones imposed by the laws of physics, human anatomy, or the current state of technological development. On the other hand, interactions and instruments qualify as natural if they conform to real-world constraints."

The interactions presented with the game concept can quickly be identified as magical, as the "lightsabers" used to play the game do not adhere to the laws of physics or the current state of technological development, as it can be computed that the power of such an object would be around two orders of magnitude smaller than what can be produced by smaller nuclear generators [7], depending on the color of the saber [8]. VRMIs allows the user to experience new interactions or extensions upon existing interactions that were otherwise impossible. Interactions can be mapped in whatever way the designer can imagine and implement. The interactions can then control parameters of an unlimited number of sound processes [9].

Since VR opens up doors to new interactions and experiences, it has been discussed whether replicating real instruments is useful. Mäki-Patola et al. argued interfaces should be created to be best suited for VR and that users might not even be interested in virtual replicas, when the originals work fine [5]. Cook called copying an instrument dumb, but recommended leveraging the techniques found in existing physical instruments [10]. The use of existing skills has been reiterated by others, while familiarity of techniques from an existing instrument can help grasp the virtual one [5, 6].

Extended opportunities arise from what input devices are used as a controller. Previous VR experiences relied heavily on custom controllers as seen from Mäki-Patola [5]. Controllers from the NIME community could increase the input possibilities, as most standard VR controllers are gloves, remote controllers or joysticks [6]. Using controllers created with sound control in mind can match the complexity of some sound models [5], though this quickly can exclude many users who do not have access to these controllers.

2.2.2 Feedback in VR

Visual feedback is an important factor in VR to create a more natural interaction with the instrument [6, 11]. The visual feedback does not have to be realistic, as cartoonish representations can give a better sense of control than realistic. The same can be seen when it comes to the fidelity of interactions. Low-fidelity interactions were observed to be better for users than semi-natural interactions, that also could create an uncanny valley effect [12]. Another drawback is cybersickness, that can arise from conflicting information from the visual and the vestibular senses [6]. Visual feedback is especially needed if the instrument does not resemble any existing counterpart, as it can use any form of sound synthesis technique and any mapping between gestures and sound [11]. Fortunately, the visual feedback is not restricted in the virtual environment. It can take any form that is needed to understand or use the instrument. Compared to the static appearance of real instruments, where the feedback is limited to the physics of the object, virtual instruments can intelligently change in real-time to fit the user's needs.

While the game concept does not present a traditional performer-audience relation as the performer is the player who is trying to achieve a high score for themselves and not playing for the audience, the relation is still present when watching another person play either in the same room or through a video. A disconnect can occur when the audience can not share the experience of the performer [6, 11]. Improving the audience experience by adding stereoscopic glasses to share the spectacular aspect was previously suggested [9], but does not seem relevant anymore. VRMIs are often experienced by seeing the performer and a screen showing a 2D presentation of what the performer sees. This way the audience can share most of the visual experience, with notable exceptions being the 3D view and haptic feedback. Today, many "performances" are shared through online videos. When the audience is not physically present, it is still important to show both the performer and what the performer sees. This can be done by creating Mixed Reality videos, where a video of the performer is mixed with what they see.

2.3 Adaptive Music in Games

The interactive aspect of video games separate them from the linearity of traditional film and television shows. This is why adaptive music is ideal for many video games, as they are "inherently interactive and involve unpredictable real-time state change" [13]. The interactive, non-linear aspect of games is also one of the most complicating problems facing game composers, as direction and timing of the player can change in a branching fashion [14]. Adaptive music is heard when actions by a player is reflected in the music [15]. These actions should not stem from the intention of playing music. The changes should come from non-musical actions [16]. It is also not to be confused with interactive audio content, that arises directly from player actions, like the "swoosh" sound effect of swinging a sword [17].

Collins described adaptive music as a train on a track [17] . Linear music is on a straight track, going from point A to point B using the same route and speed every time. Non-linear, adaptive music can at any time change tracks, you might want to change the car you are in, and it can at any point speed up or slow down. On a technical level, the adaptive music is triggered by sending events from the game engine to a music engine or middleware [14]. Interactive art installations can also benefit from this adaptiveness [13].

Although it is not common today, adaptive music was used in the production stages of early Western movies, though it could not be noticed by moviegoers. This was because composers were tasked to finish their composition before the movie had been edited, meaning the music should be able to be shortened or lengthened without becoming incoherent [14]. One of the first instances of adaptive music in video games was seen with the Interactive MUsic Streaming Engine (iMUSE) [13, 14, 16, 18]. iMUSE was used to synchronize game actions with audio.

Examples of adaptive music in video games include Monkey Island 2 and Grim Fandango, which used iMUSE to make music continuous while it reacted to spontaneous player actions by transitioning between audio files or altering MIDI [14]. The Legend of Zelda: Ocarina of Time from 1998, a long-lasting adventure game, required music that would adapt to help the player complete challenges [17]. Similarly, changes in the music from the 2012 game Journey "sets the pacing of the game" [19]. In Super Mario Brothers from 1985, the tempo of songs doubled when players ran out of time [17].

2.3.1 Uses and Techniques

The composer of Super Mario Brothers, Koji Kondo, described four components for dynamic music [20][1]:

1. The ability to create music that changes with each play-through
2. The ability to create a multicolored production by transforming themes in the same composition
3. The ability to add new surprises and increase gameplay enjoyment
4. The ability to add musical elements as gameplay features

Dynamic music encompasses both interactive audio and adaptive music [17]. The components can be separated into the first and second describing adaptive music, the fourth describing interactive audio and the third can be used for both.

Brown et al. described five techniques used to create adaptive audio [13]:

1. *Templates* are used to provide a musical "backbone" for generated music. Chord progressions, melodies and more can be pre-described and music is then generated or combined following these harmonic templates.

[1]Summarized by Collins [14]

2. *Abstraction* is used to generalize information into output. This can be gestures to create musical phrases.
3. *Recombination* is when alternative tracks or patterns are used. This can be a any segment of a song that is interchanged with another, effectively recombining the whole song. A simple state variable can be used to decide whether to use an alternate track.
4. *Transformation* of musical parts like chords or a melody is used to modify music. This can be as simple as changing the rhythm of a chord progression or transposing a motif.
5. *Probability* is used to create add randomization to computer generated composition.

The primary technique to be used in for this project is recombination, as seen in sections 3.3 and 3.4, where transitions between audio files are used.

2.3.2 Transitions

A choice that the sound designer or composer must make is the timing of a transition. Music being rooted in time can an obstacle when composing to video games that are not rooted in the same time or to the same beat [15]. The transition might happen immediately, possibly using cross-fading. When making an instant transition, a short segment can be composed as a bridge between two parts. The transition can also take place after a sequence is completed. The latter is used to avoid sudden changes that mess up the beat of a song, for instance if a transition occurs at a random point in time of a bar. Beginning a transition at the end of a bar so the part that is transitioned to starts on the beat of the next bar creates a seamless transition. This type of transition is known as cue-to-cue [17] and was even patented as part of iMUSE [21].

How to compose a musical transition can be a challenge both for computers and humans. Hoffert described seven musical elements to focus on when composing a transition: volume, tempo, rhythm, key, harmony, texture, and style [14, 22]. Of those elements, among the most common to affect how the music is perceived are tempo and key [13].

In music theory, key modulation can be used to create tension or a form of resolution. A common technique is to end on a dominant seven chord to the chord that follows the next transition. This leads to a tension that is resolved by the next chord and is also useful when transitioning between songs in different keys.

As seen with Super Mario Brothers, altering the tempo can increase the intensity, but it can also be necessary when transitioning between tracks with different beats per minute (BPM). In music theory, this is known as accelerando when increasing the tempo and ritardando when decreasing. While an accelereando can lead to tension, a ritardando often has the opposite effect of leading to an ending. If this is not the intention of the transition, other elements can be used to preserve the tension and energy.

One of these elements is the rhythm. The frequency of events in a time period is known as rhythmic density [13]. Rhythmic density thinning can be achieved by removing some events, starting with those that are placed in "less dominant metric positions". In quadruple time, the most dominant position is the first down beat on beat one, then the downbeat on beat three followed by the upbeats on beat two and then beat four. While rhythmic density thinning decreases tension, the opposite effect can be achieved by increasing the density.

2.4 Physical Modelling for Real-Time Synthesis

Physical modelling for synthesis is the act of modelling sound synthesis based on the physical properties of a source. The approach is useful for synthesizing sources such as strings, rigid bodies, vocal tracts as well as like impact, friction, air pressure and more. By using physical properties to explain and synthesize sound, real world sounds can be replicated and controlled realistically. An advantage of physical modelling synthesis beyond mirroring the real world is that physical models often can be exploited to exceed what is physically possible or practical. Strings are a perfect example of this. In the digital domain there is no limit to how long a vibrating string can be. It can go around the world and farther while still being tightly enough wound to be picked at the force of a comet crashing into it. Objects can appear to magically change and the sound can change similarly. This change can come in real-time from physical parameters like mass, elasticity and more. Even complete deformations can occur with the sound following suit [23].

A heuristic approach is often taken when using physical modelling. Taking into account all the physical properties of a source can be too complicated to mathematically model and too complex for real-time computation. This can at time lead to simplistic and "cartoonish" sounding synthesis, as some physical properties are neglected. For speech synthesis, it would be too complicated to model every detail of the vocal tract and how it moves.

A common method for physical modelling is modal synthesis [24, 25]. Modal synthesis exploits that many sources have prominent peaks in its spectral content. These peaks are known as normal modes and even simple models using sinusoids or band-pass-filtered noise based on these modes can sound like a unique material. Modal synthesis has a low complexity, which makes it ideal for real-time computation.

Digital waveguide models are well-suited for strings and wind instruments [26]. By modelling how audio propagates through a medium using delay lines, digital waveguide models are computationally efficient. Bidirectional delay lines

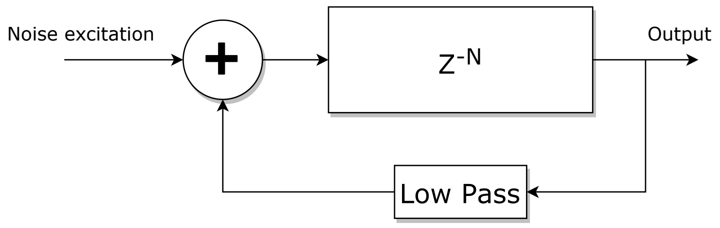


Figure 2.1: The simplest version of the Karplus-Strong algorithm with a delay loop and a low-pass filter.

are used in a one-dimensional system with losses and dispersion at points on the waveguide. The one-dimensional aspect of digital waveguide synthesis is limiting when it comes to more complex sources. To account for this, a digital waveguide mesh can be used, where digital waveguides are connected together as a two-dimensional array at a certain wave impedance [27]. Waveguide meshes are useful for 2D sources like plates, percussion instruments [28] and reverberations. Another approach in the waveguide family is banded waveguide synthesis [29]. Banded waveguide synthesis takes an approach similar to modal synthesis, where bands of digital waveguides are used to model a resonating normal mode. Some of the sources that are useful with this method are bowed friction and idiophones. Impact sounds were also achieved using banded waveguide [30]. An ideal vibrating string can be easily modelled using digital waveguides, as will be explained in section 2.4.1

2.4.1 Extended Karplus-Strong

The Karplus-Strong algorithm was invented by Karplus and Strong in 1983 [31]. The digital waveguide method is an extension of the Karplus-Strong algorithm. The algorithm models an ideal string based on a delay line with a simple loss filter.

A string is excited by a short burst of white noise that is equal in length to the desired pitch period. The simplest version of the Karplus-Strong algorithm can be seen in figure 2.1. The white noise is used as an excitation and repeated at a loop delay N determined by the pitch period, but each time it is looped, a loss occurs due to the low-pass filter. At initial conditions, the first N output samples are pure white noise. The simplest version of the low-pass filter is a one-zero filter and can be written as:

$$y[n] = 0.5x[n-N] + 0.5x[n-N-1] \tag{2.1}$$

Extension to the algorithm were published by Jaffe and Smith [32]. The extensions used for this project was later described again by Smith [33]. The algorithm works

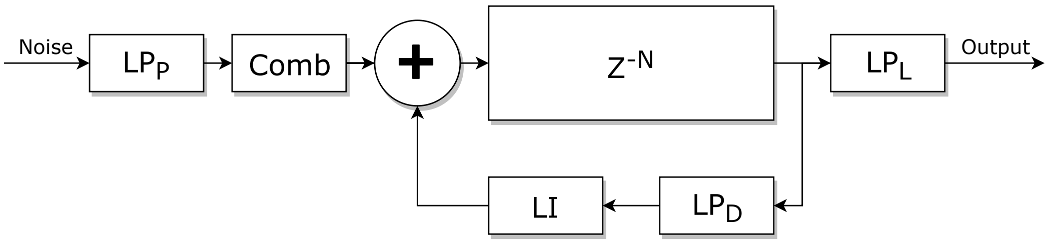


Figure 2.2: The Karplus-Strong algorithm extended with changes to the excitation burst, a dynamic level and string tuning.

similarly to the original Karplus-Strong, but with additional filters to model various aspects of strings and to alleviate shortcomings. The algorithm can be seen in figure 2.2[2].

LP_P is a low-pass filter used to model the pick direction up or down.

Comb is a comb filter used to model the pick position between the nut and the bridge. The position used is usually over the soundhole on acoustic guitars.

LP_L is a dynamic low-pass filter that models how more energetic picking of a string increases the spectral centroid.

LP_L is the low-pass damping filter used to create the loss. It is used in place of the low-pass filter of figure 2.1.

LI is a filter used for more precise string tuning. Since N is an integer if fractional delay lines aren't used, the strings can get out of tune due to the rounding up or down. The filter makes up for the lost precision by linearly interpolating between the current and previous sample..

All these extensions add to the realistic aspect of the Karplus-Strong algorithm. Especially the string-tuning is essential, since modelling an instrument would yield imprecise tones except at very high sampling rates.

Even with the additional filters, the Extended Karplus-Strong is still computationally efficient. How the filters are implemented is described in section 4.3.1

2.4.2 Low-Level Friction

The low-level friction synthesis is a system of mass-spring interactors coupled together [34]. The mass-spring system is a mechanical oscillator described by its displacement, velocity, acceleration and the force acting upon it. Assuming no other forces are acting upon it, the system can be seen in figure 2.3, where x is the

[2]One filter, the string-stiffness all-pass filter, is omitted here.

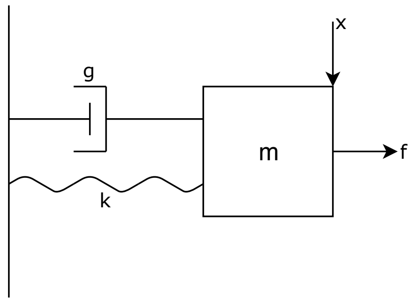


Figure 2.3: A mass-spring system with damping.

displacement, f is the force, k is the elastic constant, g is the damping and m is the mass. The vibrations lead to the displacement, which is used as the audio output. The system can be written in a continuous-time system as:

$$\ddot{x} + g\dot{x} + \omega^2 x = \frac{1}{m} f \tag{2.2}$$

where ω^2 is equivalent to the elastic constant, and $\dot{x}$ and $\ddot{x}$ are the velocity and acceleration respectively.

The oscillator is useful for modelling interactions with rigid bodies. This was seen in impact collisions with a hammer-resonator relationship and various approaches using the model to create position-dependent excitation exist using a contact force equation proposed by Hunt and Crossley [35–37]. Using friction as the force, the model can be used in bowed instruments like the cello and objects like squeaking doors or wheels grinding to a halt [34, 38].

Equation 2.2 requires a force equation. Which force equation is used can set it apart from sounding like an impact and more like friction. An equation of the steady force using the Coulomb force f_c and stiction force f_s is given by

$$f(v) = sgn(v)(f_c + (f_s - f_c)e^{-(v/v_s)^2}) \tag{2.3}$$

where v is the velocity and v_s known as the Stribeck velocity.

The equation for the friction force uses the elasto-plastic modelling approach and is given by

$$f(z, \dot{z}, v, w) = \sigma_0 z + \sigma_1 \dot{z} + \sigma_2 v + \sigma_3 w \tag{2.4}$$

where w is a random value to add noise and v is the velocity. Each component describes an aspect of the friction. $\sigma_0 z$ is an elastic term, $\sigma_1 \dot{z}$ describes the dissipation and $\sigma_2 v$ is the viscosity. The remaining equations making up the system are taken from Rocchesso [34].

2.4.3 Bilinear Transform

Since equation 2.2 is in continuous time, it must be discretized before being used of sound synthesis. Using a substitution formula, the continuous time system can become a discrete-time z-transfer of numerical approximations. Various z-transform methods exist, with some of the most common being the forward Euler, the backward Euler and the bilinear transform. The transforms are given by:

$$\begin{aligned} s &= Sr(1 - z^{-1}) && \text{(Backward Euler)} \\ s &= Sr(1 + z^{-1}) && \text{(Forward Euler)} \\ s &= 2Sr\frac{1 - z^{-1}}{1 + z^{-1}} && \text{(Bilinear Transform)} \end{aligned} \tag{2.5}$$

where Sr is the sampling rate and z^{-1} will become equivalent to a delay of one sample. The transforms are substituted with the s from the state-variable form of a continuous time system. The continuous time system should take the form

$$\begin{aligned} \dot{w} &= Aw + Bu \\ ws &= Aw + Bu \end{aligned} \tag{2.6}$$

where $\dot{w}$ is the derivative of the states and w and u are the states transformed by A and B. Substituting s with the bilinear transform and isolating w, it can be written in discrete time of sample n as:

$$\begin{aligned} w[n] = H(\alpha I + A)w[n-1] + HB(u[n] + u[n-1]) \\ H = [\alpha I - A]^{-1} \end{aligned} \tag{2.7}$$

Assuming the input and output are not mutually dependent and form a delay-free loop, this form of the state-variable system can be used. If not, the K method can be used to eliminate the delay-free loop.

2.4.4 Eliminating Delay-Free Loops

The K method solves the computation problem of delay-free loops [39]. The method adds an additional variable and state transform to equation 2.6 with

$$\dot{w} = Aw + Bu + Cy \tag{2.8}$$

where y is the mutual dependent variable. New state transforms D, E, and F are used to describe y:

$$y = f(Dw + Eu + Fy) \tag{2.9}$$

Equation 2.8 is discretized using the bilinear transform just like equation 2.7:

$$\begin{aligned} w[n] = H(\alpha I + A)w[n-1] + HB(u[n] + u[n-1]) + HC(y[n] + y[n-1]) \\ H = [\alpha I - A]^{-1} \end{aligned} \tag{2.10}$$

The inside of the function f is discretized using the found form for w:

$$\begin{aligned} DH(\alpha I + A)w[n-1] + (DHB + E)u[n] + DHBu[n-1] + DHCy[n-1] \\ + (DHC + F)y[n]) \end{aligned} \tag{2.11}$$

This is separated into a linear combinations current and past computable values in $p(n)$ and a weighting matrix K to the mutually dependent variable y

$$\begin{aligned} y = f(p(n) + Ky) \\ p(n) = DH(\alpha I + A)w[n-1] + (DHB + E)u + DHBu[n-1] + DHCy[n-1] \\ K = (DHC + F) \end{aligned} \tag{2.12}$$

An explicit solution can not be found if y is non-linear, but an approximation method can be used to estimate the value. One method that can be used is Newton-Rhapson.

2.4.5 Approximating a Solution to the Non-Linearity

The value of a system with a non-linearity can be approximated using the root-finding algorithm Newton-Rhapson. The approach here is adopted from Rocchesso [34]. The value the root-finding algorithm is looking for is denoted as h, which serves as the mutual dependent variable y from the K method. h is found by repeating the process

$$h_{k+1} = h_k - \frac{g(h_k)}{g'(h_k)} \tag{2.13}$$

where k counts each time the process is repeated until the function converges below and error margin of the absolute difference of the current and next h. h_0 starts out as an estimated guess.

Combining the process with the K method, $g(h)$ becomes

$$g(h_k) = f(p[n] + Kh_k) - h_k \tag{2.14}$$

The maximum error and a maximum amount of times the process may repeat is determined. The error is calculated as the absolute value of the difference between the current and previous h.

2.5 Tools

Implementing the VR game concept required a game engine and efficient digital signal processing. The following software and tools were used.

Unity Unity is a game engine that can be used to create 2D and 3D audio-visual applications. The game engine includes all the components necessary to implement the base game concept. This includes 3D objects, shaders, a collision detection system, audio playback and more. Programming in the game engine uses the C# language with the base class MonoBehaviour that all other Unity scripts derive from. It is possible to create real-time synthesis inside the MonoBehaviour function `OnAudioFilterRead` in C#. The function can act as a filter if audio is passed through it or create new sound. It is not as computationally efficient as C++, which is why it is not used in this project.

Native Audio Plugin SDK The Native Audio Plugin SDK for Unity adds the possibility to create native plugins in C++ that can be compiled to work as plugins in Unity's mixers. It is similar to a VST[3]. The native plugin acts as a part of an audio chain, meaning it can take in values of a plugin before it and pass it on to the next. Audio is operated on a specified buffer at a time. The buffer size and sampling rate is set in Unity. Compiling the C++ file to .dll for the specific platform in use and adding it to the Unity project makes it available as and effect for an audio mixer component.

Code is added to five parts of the framework. They are the parameters to expose to Unity, dynamic unexposed parameters, constant unexposed parameters, a callback when parameters are changed in Unity, and a process callback, which is where most DSP takes place and fills the buffer.

HTC VIVE The standard HTC VIVE VR head mounted display was used for this project. The HMD has a refresh rate of 90 Hz, a field of view of 110 degrees and supports six degrees of freedom. Most HMDs could have been used for this project, since no specific interactions except moving the controllers and pressing standard buttons were required.

OpenVR OpenVr enables VR support for building VR applications in Unity. The package requires the use of STEAMVR, which can be used to calibrate the surroundings used for the VR application.

[3]Virtual Studio Technology, an audio plugin usually running in a digital audio workstation (DAW)

FMOD FMOD is an audio middleware. It is used to control audio and effects with control options that are not available natively in Unity. Since it is a middleware, parameters in FMOD is controlled through C# code in Unity. One use of FMOD is adaptive music, where a song can be set up with transitions based on conditions, that can be altered using the control parameter.

Chapter 3

Design

3.1 Cylindrical Guitar Anatomy

The VRMI "Guitar Saber" is designed to rely on existing knowledge of musical instruments but with a new interaction. As the name suggest, the instrument is a combination of the functionality of a guitar with the visuals of sabers. The design aims to preserve many aspect of traditional guitar playing to make users instantly familiar with it. These functionalities include the layout of strings and frets, up and down picking, glissando effects and more. Keeping the game mechanic of using no buttons, the Guitar Saber relies on gestures to control the synthesis. An initial mock-up and final design can be seen in figure 3.1 It will here be explained how every musical functionality is adapted to the sabers.

Picking a String The interaction of plucking or picking a string is designed to be a simple tap of one saber on the other. The original game includes haptic feedback whenever the sabers collide, and the Guitar Saber adds a fully-fledged musical aspect to this interaction. The pick can occur from any direction, but due to how the sabers are held, it is usually a top-down or bottom-up tap. Picking is designed to be done by slashing the right blue saber onto the stable left red saber similarly to right-handed guitar players using one hand to pluck and the other to rest on frets. Due to both sabers being similar cylindrical objects, the pick can be done by any colliding interaction. In the top of figure 3.1 the pluck occurs at the intersection of the two sabers, at the C. The note C is determined by the frets and strings.

Frets The anatomy of a guitar includes a fretboard, where raised elements called "frets" are used to fixate the string length between the fret and the nut. Pressing down on the string between two frets makes it fixated on those two frets. In the standard western system, each fret represents a semitone.

The same semitonal approach can be seen on the left red saber in figure 3.1.

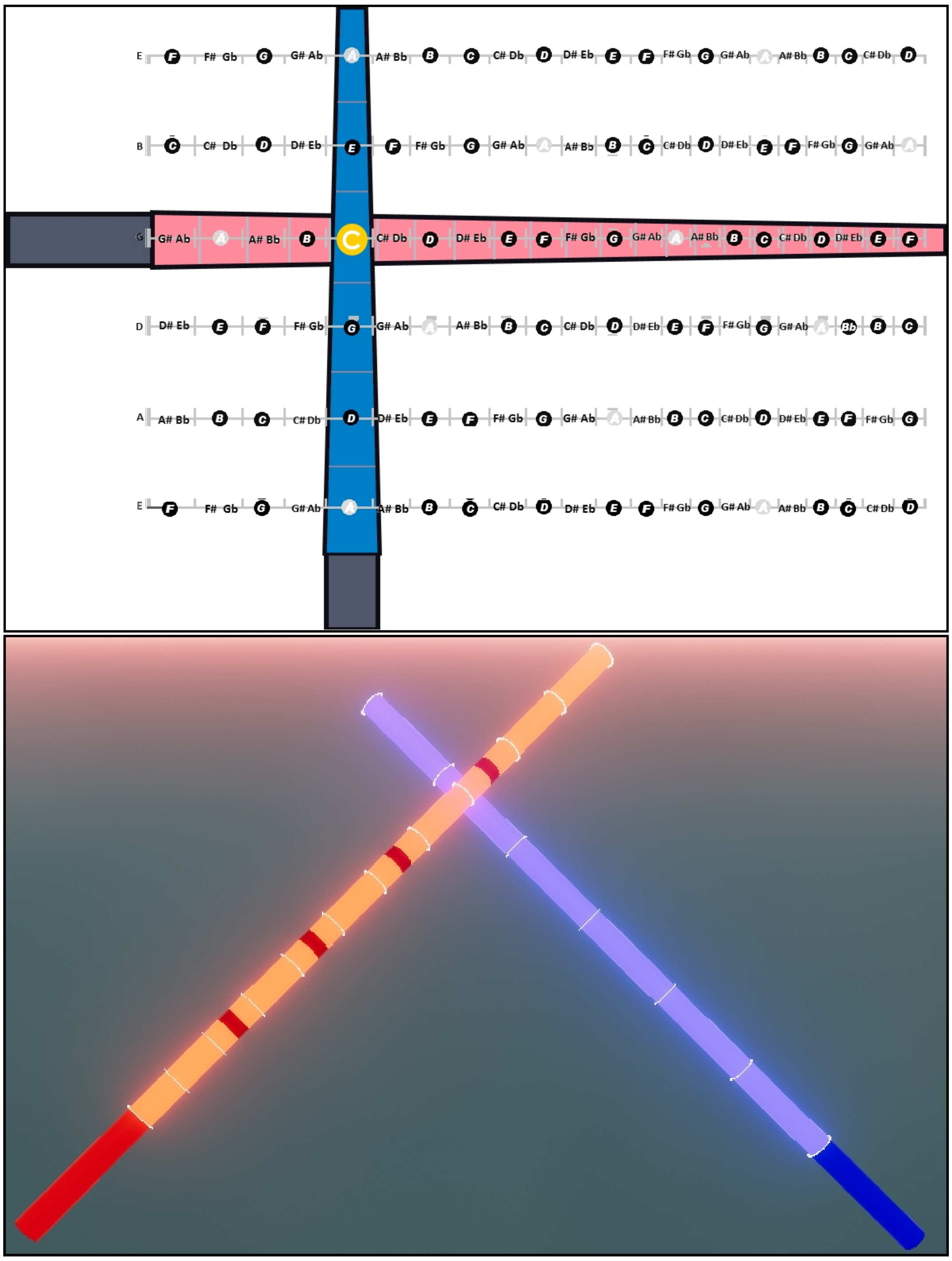

E F F# Gb G G# Ab A A# Bb B C C# Db D D# Eb E F F# Gb G G# Ab A A# Bb B C C# Db D
B C C# Db D D# Eb E F F# Gb G G# Ab A A# Bb B C C# Db D D# Eb E F F# Gb G G# Ab A
G G# Ab A A# Bb B C C# Db D D# Eb E F F# Gb G G# Ab A A# Bb B C C# Db D D# Eb E F
D D# Eb E F F# Gb G G# Ab A A# Bb B C C# Db D D# Eb E F F# Gb G G# Ab A Bb B C
A A# Bb B C C# Db D D# Eb E F F# Gb G G# Ab A A# Bb B C C# Db D D# Eb E F F# Gb G
E F F# Gb G G# Ab A A# Bb B C C# Db D D# Eb E F F# Gb G G# Ab A A# Bb B C C# Db D

Figure 3.1: The mock-up prototype (top) with note placements resembling a two-dimensional array. It is realized in the Unity (bottom) with the two sabers including frets and string indicators.

The saber represents the fretboard and frets are placed to visually distinguish the tones, since there is no string that will be fixated on it. As the frequency increases on a single string, the distance between frets decreases. It is based on the 12 tone equal temperament, which results in a ratio of $\sqrt[12]{2} \approx 1.0595$ between each fret. Using this ratio, the longest distance would be between the lowest frets near to the handle and the shortest distance would be at the nut. This was not practical for the Guitar Saber, as the angle formed by the two sabers usually gets sharper when interacting at the far edge of the sabers. A sharper angle makes it harder to precisely slash between two frets. Therefore, the sabers are designed with the ratio progressing in the opposite direction of a standard guitar. This means the distance between frets near the far edge are longer than the distances closer to the handle.

Position inlays can be placed on to help guide the player where tones lie. They are often placed at the 3rd, 5th, 7th, 9th, 12th tone position. This pattern can be repeated on subsequent octaves on the same string. The visual aids are added to the fretboard of the left red saber as seen in the bottom of figure 3.1.

By using semitonal frets, guitar tablatures can be used to play songs on the Guitar Saber. The lowest tone, represented by the handle, is the same as playing an open string.

Strings Which string to use is determined by the position on the right blue saber. For a standard western guitar, there are six strings, as seen in figure 3.1. The mock-up shows how this results in a two-dimensional map of tones. Moving from one string to the next is similar to moving from one fret to the next, except it is not a semitonal step but instead relates to a guitar tuning. In this example, it is the standard tuning EADGBE.

Discrete Glissando Glissando is the effect of sliding discretely from tone to tone. From early user testing it was observed that participants naturally tried to glide from one tone to the next it while the sabers where staying in a state of collision. The glissando effect works by sliding the left hand position on a fret up or down in one dimension on the fretboard. On a guitar it is not possible to have a glissando between strings, but the Guitar Saber is not limited by this, which allows it to become two-dimensional, both along the frets and the strings.

Overdrive The futuristic and electric look of the sabers do not fit with the natural sound of an ideal string without any additional effects. As previously discussed in section 2.2.1, the power of a theoretical light saber is enormous and the sound should follow this thought. Distortion and overdrive effects are useful for this, as they amplify the signal, often resulting in clipping. This gives off a compressed and intensified sound that is better suited for the sabers.

Wah-Wah The wah-wah effect is commonly used as pedals for electric guitars to add a spectral glide to the sound. Getting its name from the audio phenomenon of vocalizing "wah-wah", the effect focuses on a center frequency that can change according to user input. For the wah-wah pedal, the center frequency depends on the pressure on the pedal. The Guitar Saber can control this effect by angling the left red saber up and down, just like the pedal is angled when pressure is applied to it.

3.2 Carving Friction

As is the case with Guitar Saber, the carving friction sonic interaction is designed to add a musical aspect to where a collision previously only triggered haptic feedback. Instead of the two sabers colliding, it is the case of a saber colliding with the wall obstacle. In the game, players should avoid the walls, but the sabers can safely interact with them for no apparent advantage except being a gimmick. The carving friction adds the musical aspect to the interaction.

The intersection of the saber-wall interaction controls the frequency of the friction sound. The closer to the ground floor it is, the lower the frequency. Compared to the discrete separation the frets on a guitar can create, friction-based instruments like the violin and cello are continuous. The friction interaction with the wall can be set to a value linearly between a maximum and minimum or to a key.

Visually, there is not change from when the wall friction is on or off. Compared to the Guitar Saber, this means it fits directly into the game without changing what the player sees, while adding a musical aspect to the walls.

3.3 Adaptive Music Remix Mode

The primary focus of the first adaptive music mode is to observe whether it can increase replayability. This is done by changing the song that is being slashed along to from a linear song to include recombinations. Adaptive recombinations, as described in section 2.3.1, are where alternate tracks are used in certain parts. Figure 3.2 shows how this works when part of an intro can change. A corresponding implementation in FMOD can be found in figure 4.1.

The mode does not randomly select when to do a recombination, but it is instead triggered by a condition based on the actions of the player. This can be the need to achieve a high streak of slicing cubes correctly in a row or having obtained enough points or another aspect. By having it be a point-positive condition, i.e. something that encourages getting a better score, the condition is not limiting the player's possible result in points achieved. The conditions can be seen as a gate that can be unlocked. If the condition is not met, the gate is visibly locked to the

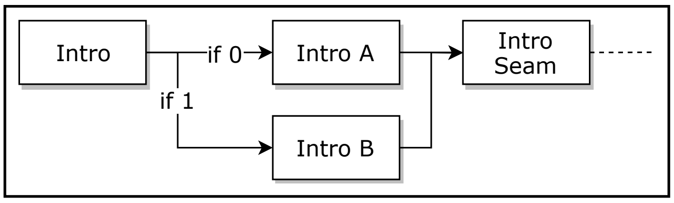


Figure 3.2: Beginning of a remix mode example, where part of the introduction can change depending on a simple boolean condition.

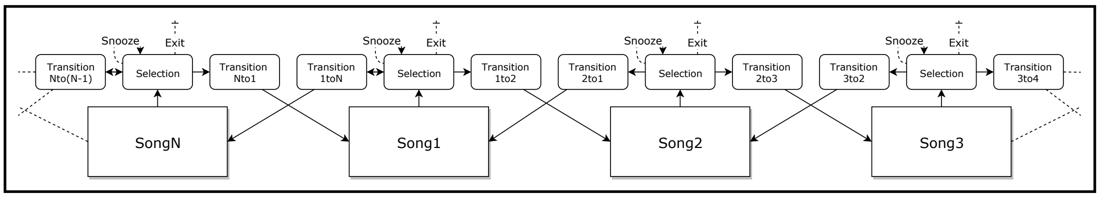


Figure 3.3: An overview of how the endless version can progress from song to song. Starting from any song number, it plays the main song before going to the selection segment. This segment can be snoozed, exited or the next song can be selected, which will trigger the transition leading to the next song, where the process is repeated.

player and the alternate track is kept secret. The hypothesis is that this sparks the players curiosity to replay the song and unlock the locked gate.

The adaptive remixing is underlined by the visuals, that changes the color of most objects when entering a remixed segment. A short time before an adaptive remix can be entered, a form of countdown appears as well as a user interface explaining what the player must do to unlock the remixed segment.

3.4 Adaptive Music Endless Mode

The endless mode is designed to avoid the need to stop playing and return to a selection screen. This would be especially useful for people who use the game as a workout exercise. The endless mode is also a version of adaptive recombination, and a diagram of the transitions can be seen in figure 3.3. Every box in the diagram requires a musical composition and a mapping. The parts are explained in the following paragraphs.

Song The song can be any song linearly composed and mapped song. Each time a now song starts, a variation in color of the environment changes to signal the new beginning.

Selection The selection is a musically composed and lets the player select the next song without the need of a menu. During the selection segment, the player can pick the next song by slashing a cube that symbolizes this song. If no song is selected, the game effectively "snoozes", and the selection segment is seamlessly repeated along with the ability to select the next song. This is useful for taking a breather between songs while working out. Instead of selecting a song, exiting the mode can also be selected. Musically, the selections in this project are designed to sound as natural extensions to their respective song by keeping the BPM, key, instruments and musical themes used. The same chord progression is used in all songs, though in different keys, to have a consistent aspect that makes the players associate it with the selection segment.

Transition The transitions are composed and mapped to seamlessly go from one song to the next. Section 2.3.2 explains how this can include altering the tempo, key and mixing themes from both songs. Each transition is composed to link a specific pair of songs together. To create a familiar aspect in all transitions, an increase in rhythm density is used at the end of the transition, leading to an increase in intensity, signalling a new song is about to start.

Chapter 4

Implementation

4.1 Beat-Slashing Game

The game concept is recreated from the ground up using the Unity game engine. Appendix A.2 includes a link to the whole project including all code. The game includes many components that will not be discussed in detail here. Some of them are listed below. They were created specifically for this project unless otherwise listed.

- Menu scene design
- Song scene design
- Song mappings[1]
- JSON importation to mapping classes
- Coordination of mapping data
- Object spawner
- Audio playback
- Scoring system
- UI for point system
- UI for selection menu
- Raycast interaction with UI
- Controller input hook
- Haptic feedback hook
- Custom 3D meshes (using Blender[2])
- Use of collision system
- Cube object controller
- Wall object controller
- Bomb object controller
- Saber interaction controller
- Saber angle calculations
- Mesh slicing[3]
- Environment animation

How a beat mapping results in a spawned object is explained here, as it will become relevant for timing of the adaptive segments.

When a beat mapping is created, it is specified when a type of object should be slashed by the player, or avoided, if it is an obstacle. The timing of this is set in quaters at a BPM matching the song. If the player should hit the first four quater notes, they are placed at time 0, 1, 2 and 3. This is then exported to a JSON file containing the timing, object type, and possibly slashing direction for cubes or duration for the length of walls.

The mapping data is imported into classes in Unity matching the data structures in the JSON files. From there, a coordinator can check when it is time to spawn an object. The timing in this coordinator matches the BPM by converting a timer in seconds to quaters. Since the timing in the mapping is when it reaches the player, there is a window of time before where it must spawn and move towards the player. If the objects move at 10 units per second, also known as the "note jump speed", and it spawns 10 units in front of the player, it takes it one second to reach the player. One unit in this Unity project is one meter. To synchronize the music and beat mapping, the music must start a second after the mapping. An additional visual spawning part is added to the cubes lasting 0.3 seconds, making the synchronization spawn window last 1.3 seconds for a note jump speed of 10. An increase in note jump speed makes the game harder.

4.2 Adaptive Music

Implementing the adaptive music is achieved by setting up a song with transitions in the audio middleware FMOD, creating a beat mapping and synchronizing it with events in Unity.

4.2.1 Adaptive Segments

The song is set up so segments smoothly can recombine to another version. In this implementation, a song is used alongside a remixed version of it. The remixed version follows the same structure and segments from it can be interchanged with the original. Figure 4.1 shows part of the composition in FMOD with the first recombination.

The gray flags are destination markers while the green are transitions. In this example, part of the intro can change to an alternate track depending on a condition. This condition is checked in the transition marker "To Intro B". If the condition

[1]Community created mapping tools were used to create the JSON files containing the mapping and info. `https://github.com/squeaksies/MediocreMapper`.

[2]Blender is an open source 3D creation suite.

[3]An open source mesh-slicing framework was used from `https://github.com/DavidArayan/ezy-slice`.

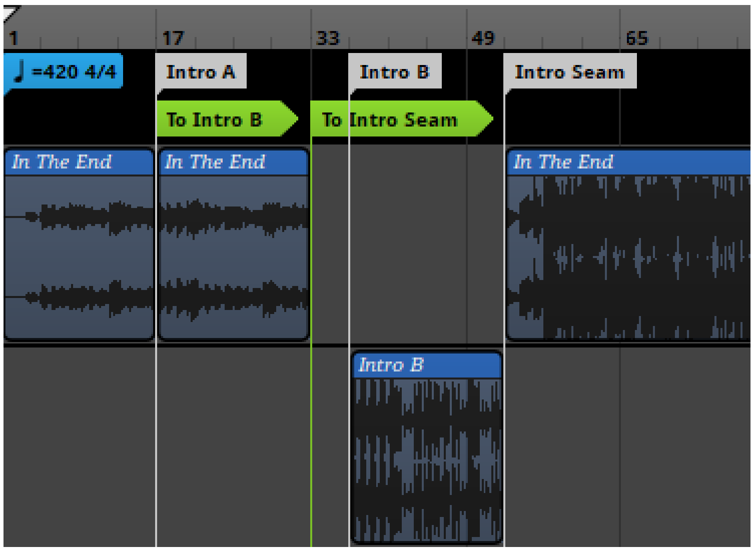


Figure 4.1: A snapshot of the FMOD setting for the adaptive remix song. The green flags are transition markers and can make the audio timeline jump to on the gray destination markers depending on a condition.

is met, the timeline jumps to the destination marker "Intro B" and carries on from there. If not, the timeline continues until it meets the transition marker "To Intro Seam". This transition has no condition, and the timeline immediately jumps to the "Intro Seam" destination marker.

To create a beat mapping of this, the audio is exported from FMOD where no transition is triggered, meaning both segments are heard, creating a recording with all the audio files that are used for the song. This song is then mapped using the beat mapping software and exported to JSON. The JSON file is manually modified by inserting when the transitions occur as seen in the JSON code below.

Listing 4.1: JSON File with Recombination Timing

```
"_secrets": [
  {
    "_time": 16,
    "_durationA": 16,
    "_durationB": 16,
    "_condition": 2,
    "_value": 10
  },
  {
    "_time": 152,
    "_durationA": 32,
    "_durationB": 32,
    "_condition": 1,
    "_value": 100000
  },
    {
      "_time": 284,
      "_durationA": 36,
      "_durationB": 36,
      "_condition": 2,
      "_value": 50
    },
    {
      "_time": 404,
      "_durationA": 36,
      "_durationB": 36,
      "_condition": 1,
      "_value": 150000
    }
```

The recombination data structures are here called "secrets". The first secret matches the FMOD of figure 4.1, as the time of 16 is when the recombination condition is checked, since FMOD's counter starts from 1 and the counter implemented in C# starts from 0. The durations explain how long either variations last, making it possible for them to differ in length. The condition determines which goal in the game the player must achieve and the value is connected to the goal. The first condition of 2 is linked to having a streak of slicing at minimum 10 cubes correctly in a row. The second, condition 1 and its value of 100000, checks whether the player has reached a score in points of 100000.

The time and duration also separates the beat mapping into regular and remix through an import function in C#. Using the first secret, it can be seen that all beat mapping between time 16 and 32 is part of the regular because of the time 16 and duration of 16. Knowing 4 quaters of silence are inserted after this, the remixed part can be calculated to be between 36 and 52. All beat mappings between 36 and 52 are then part of the remix and is moved to another data structure. The time values in the segment is also subtracted by 20 (16+4 quaters of silence) moving it into the same 16 to 32 space as the regular. All values following the remixed segment are also subtracted by 20. This continues with the next secret. In the end, the regular and remixed beat mappings are running in parallel using the same counter.

There are two reason for doing it this way. Firstly, the beat mapping can be created in a single instance using the beat mapping software and be contained in a single file. Secondly, the counter does not have to skip any segment of time, making it run from 0 and until the song is over at the same tempo.

Similarly to how the music must be started after the spawn window as described in section 4.1, the condition checking is also done before recombination takes place. This is set to the same window as the song, which is 1.3 seconds for a note jump speed of 10.

Visually, the player is notified that a change is about to happen. This is done by adding frames one at a time to resemble a countdown while a UI shows what is required to unlock the gate. Figure 4.2 shows what this looks like with a gate where the condition is not going to be met (top) and where the condition has been met and the remixed beat mapping is being spawned (bottom).

The UI includes a tick that is filled when the goal is achieved, a word to explain the goal (in this case "combo"), as well as the value and what the player currently has. On the top in the figure, the player has not correctly sliced a single cube and the combo is zero. To unlock the gate, a combo of minimum 4 had to be achieved, and the frames will disappear again. On the bottom the combo has be exceeded and the change is occurring. This is also underlined by the change in color, as the red and blue is part of the regular, while green and purple is part of the remix. A red cube is still visible in the bottom, as the change is just about to happen. At the end of the remixed segment, the frames disappear and the colors change back.

A coordinator C# script controls when to spawn regular or remixed objects, when to check a condition and when to spawn the frames. If a condition is met, it signals FMOD by sending a value to it using `setParameterValue("Trigger", f)`, where `f` is 1 for a remixed section and 0 otherwise.

Other versions were tested before settling on this version with frames several quater beat beats before the change and the condition being checked at the same window as the song.

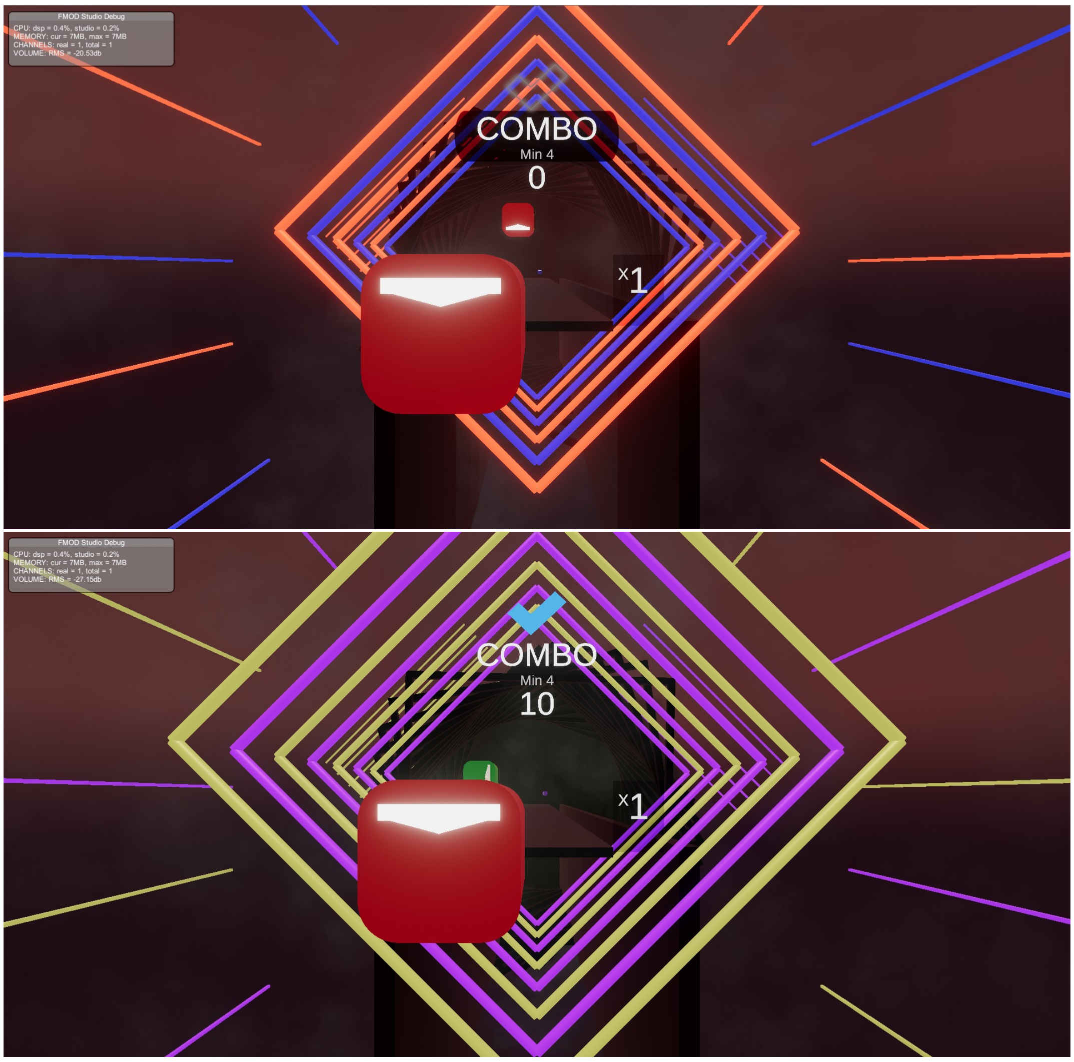
In one version, the condition was checked a single quater before, after the remixed objects were already spawned. This was possible as the remixed objects were spawned on the opposite site of the long track in front of the player. If the condition was met, the track rotated 180 degrees, resulting in the regular track being upside down. This version was better at visually depicting there was a change, but the transition time made it impractical.

4.2.2 Outro Transitions

How to synchronize the outro transitions for endless playback required a different implementation, as it needed to transition to a song with another BPM and should possibly repeat itself in the "snooze" section.

Three songs were composed to work with endless playback. Each had a transition explicitely composed to transition into one of the other based on the transition elements described in section 2.3.2. All songs were then set up in FMOD with transitions leading into each other as seen in figure 4.3

The songs include a the main song, a selection and two transitions each. Just like the recombination transitions based on a condition, the endless transitions at the end of the selection and each transition based on a condition signaled in Unity. The timings of the selections and transitions are manually obtained along with the length of the transitions. For simplicity, all transitions linearly interpolate



```
FMOD Studio Debug
CPU: dsp = 0.4%, studio = 0.2%
MEMORY: cur = 7MB, max = 7MB
CHANNELS: real = 1, total = 1
VOLUME: RMS = -20.53db
```

```
FMOD Studio Debug
CPU: dsp = 0.4%, studio = 0.2%
MEMORY: cur = 7MB, max = 7MB
CHANNELS: real = 1, total = 1
VOLUME: RMS = -27.15db
```

Figure 4.2: A view of the remix mode in action. Top shows an instance where the goal is not achieved and the gate will not be unlocked. The tick is not filled, the combo is below the minimum of 4. Bottom shows a combo above the minimum. The colors have begun to change and another colored not is visible. In FMOD, the transition is synchronized to start at the time the player slices the visible green cube.

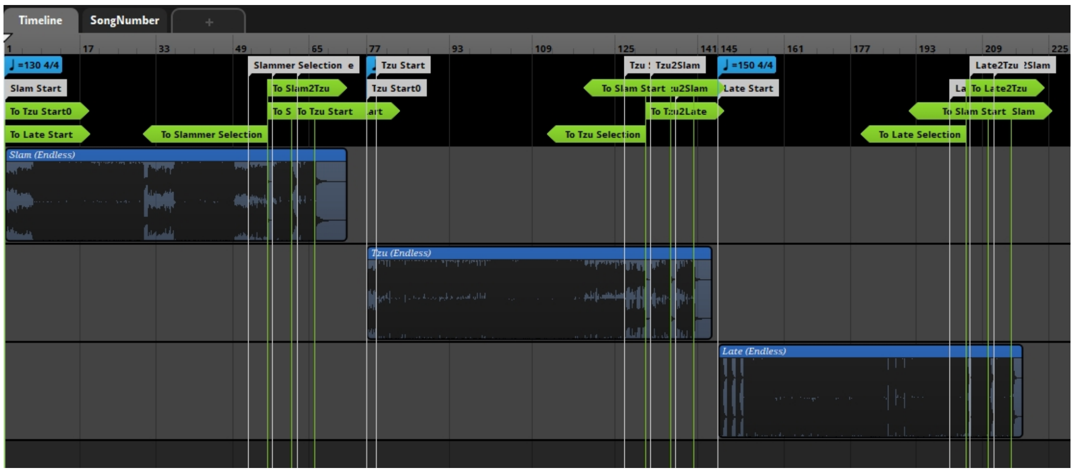


Figure 4.3: The FMOD implementation of the endless music and their transitions. Three songs are used in the example.

the BPM it is transitions to so the average BPM of the two songs can be used for synchronization.

When the song is at a selection segment, special cubes are spawned including song titles or "Quit", that the user can slice. If the user does not slice one of the cubes, the music adapts by repeating the selection segment. Slicing the "Quit" object stops the song after the selection. The following function is triggered after a song can be selected, in the middle of the selection segment.

Listing 4.2: C# Code Synchronizing the Endless Transition

```
private void Shift2Song(ref float q, float secTransitionTime, ref float
    endTime, bool isOtherColor, float timeMult, ref bool gateSpawned)
{
    musicControl.SetSongNumber(shift2SongNumber);

    float bars2Shift = 8;

    if (shift2SongNumber == 0)
    {
        Invoke("StopSong", bars2Shift/timeMult);
        return;
    }

    q = secTransitionTime*timeMult - bars2Shift + 1.3f*timeMult;

    float nextBPM = data.songBPM[shift2SongNumber];

    float avgBPM = curBPM / (0.5f*(curBPM + nextBPM));

    float transitionBarTime = bars2Shift + 16f * avgBPM;

21      endTime = secTransitionTime*timeMult + 17f * avgBPM;
22
23      float translateBPM = nextBPM / curBPM;
24
25      Invoke("ColorSwitch", (transitionBarTime-2f) / timeMult);
26
27      float newOffset = data.barStartOffset[shift2SongNumber] - (
            transitionBarTime  * nextBPM/curBPM) + (1.3f * nextBPM /60f);
28
29      BeginWithSong(shift2SongNumber, newOffset, !isOtherColor);
30  }
```

The function can be described with the following steps with the codes line number in parenthesis:

1. (3) Signal FMOD the next song number. If it is the same as the current song, the transition in FMOD repeats the selection segment.
2. (5) This happens in the middle of a selection segment of 16 bars, so the amount of bars until the shift is 8.
3. (7-11) Song number 0 is set to stopping the song. This is invoked after the selection is over by converting from quaters to seconds. The function the returns.
4. (13) The timing counter `q` affects which beat mapping objects to spawn. The counter is synchronized to the beginning of the transition subtracted by the time until the shift and the spawn window. Compared to the adaptive remix, altering the counter `q` is unavoidable when it also needs to be able to the repeat selection segment indefinitely.
5. (15) BPM of the next song is found.
6. (17) A multiplier to convert current bars to the average.
7. (19) Transitions last 16 bars if the tempo is kept, but this is altered with the multiplier.
8. (21) The time to end the use of the current song's beat mapping is set one bar after the transition is over. Beat mappings of the two songs overlap for a while in code, but not visually.
9. (23) A multiplier to translate the current bars from the current BPM to the next.
10. (25) Similarly to the color change seen in figure 4.2, the colors are changed just before the transition

11. (27) The offset in time of the next beat mapping is found. It can have an additional offset in `barStartOffset`, if the song should not start from the beginning. The previously calculated values and spawn window are subtracted, meaning the next counter usually starts as a negative value.

12. (29) The beat mapping for the next song is begun with the song number, the counter offset and which color of objects to use.

With this synchronization of timing it is possible to play endlessly. One drawback is that the timing of the song in FMOD and the counter for each beat mapping runs independently of each other. If there is latency in either instance, it will only be reflected on that part and the song and beat mapping will become out of sync.

4.3 Guitar Saber

This section will explain how the string sound synthesis was implemented and realized as a VRMI.

4.3.1 Ideal String Synthesis

The implementation is based on a FAUST version by Smith [33] and can be seen in figure 2.2. MatLab code for prototyping can be found in appendix A.3 and the C++ code using Native Audio SDK can be found in appendix A.4. The filters and their implementation are described in the following paragraphs.

Noise Excitation The noise burst is N white noise samples, where N is the sampling rate divided by the fundamental frequency. This value is rounded down to become an integer delay. The rounding error is later accounted for with the linear interpolation. Part of the filling an array with N white noise samples is seen below.

Listing 4.3: C++ Initial noise burst

```cpp
data->N = (int)floor(SAMPLERATE / data->p[P_FREQ] - 1.0f);

for (int n = 0; n < data->N; n++)
   data->noiseBurst[n] = data->random.GetFloat(-1.0f, 1.0f);
```

Variables of the type `data->p[var]` are created to be exposed to Unity. In this example, `var` is the fundamental frequency `P_FREQ`. Using `data->` without the `p` array, like the frequency, are unexposed dynamic parameter. Loop delay `N` is such a parameter, as it is used over many buffer windows and similarly will all arrays containing previous data, like `data->noiseBurst[]`.

Pick Direction Lowpass The pick direction lowpass filter is given by the transfer function

$$H_p(z) = \frac{1-p}{1-pz^{-1}} \tag{4.1}$$

and its discrete-time difference equation is

$$y[n] = (1-p)x[n] + py[n-1] \tag{4.2}$$

The pick direction is determined by the value of p, which is a value between 0 and 1.

Listing 4.4: C++ Pick Direction

```
data->outPickDir[n] = (1 - data->p[P_PICKDIR_P]) * data->noiseBurst[n] +
    data->p[P_PICKDIR_P] * data->outPickDir[(n-1) & 0x3FF];
```

The code snippet is applied to the noise burst only, which makes it part of the initial excitation. Bit shifting 1023 with `& 0x3FF` is used for modulo 1024 to keep the indexing from going out of bounds. This bit shifting trick is used for indexing throughout the implementation.

Pick Position Comb Filter The comb filter simulates the pick position from the nut to the bridge of a guitar, where 0 is the bridge and 1 is the nut. The transfer function is given by

$$H_\beta[z] = 1 - z^{-\beta N} \tag{4.3}$$

and the difference equation becomes

$$y[n] = x[n] - y[n - \beta N] \tag{4.4}$$

where N is the loop delay and β is the 0 to 1 value. Combining this with the pick direction and white noise, the initial excitation burst is seen in the code below.

Listing 4.5: C++ Complete Initial Excitation Burst

```
data->N = (int)floor(SAMPLERATE / data->p[P_FREQ] - 1.0f);
int combDel = floor(data->p[P_PICKPOS_BETA] * data->N);

   for (int n = 0; n < data->N; n++)
   {
      data->noiseBurst[n] = data->random.GetFloat(-1.0f, 1.0f);
      data->outPickDir[n] = (1 - data->p[P_PICKDIR_P]) * data->noiseBurst[
          n] + data->p[P_PICKDIR_P] * data->outPickDir[(n-1) & 0x3FF];
      data->outPickPos[n] = data->outPickDir[n] - data->outPickDir[(n -
          combDel) & 0x3FF];
   }
```

The pick position of the sabers follow the frets whereas the pick position of a guitar usually is above the soundhole for an acoustic guitar and above the pickups for an electric guitar. The value β is exposed in Unity, but does not currently change when playing.

Loop Delay In the Native Audio SDK method "ProcessCallback", which allows data to be inserted into the audio buffer, it is ensured that the noise only is used as the initial excitation.

Listing 4.6: C++ Loop Delay

```cpp
float noise = 0.0f;
for (unsigned int n = 0; n < length; n++)
   {
      noise = 0.0f;
      if (data->noiseIdx < data->N)
      {
         noise = data->outPickPos[data->noiseIdx];
         data->noiseIdx++;
      }
         data->outDel[data->n] = noise + data->outStringTuning[(data->n -
             data->N) & 0x3FF];

         // Rest of filter and buffer operations here...
   }
```

The buffer of `outStringTuning` is the output of the linear interpolation, that will be explained later. The counter `noiseIdx` is what ensures the noisy excitiation only last the cycle of the first loop delay. Since the data is delayed immediately, the `data->n` counter starts out equal to the loop delay, as the first output otherwise would be N samples of zero.

Damping Lowpass The damping filter introduces the loss. The transfer function for the second-order filter is

$$H_d[z] = b_1 + b_0 z^{-1} + b_1 z^{-2} \tag{4.5}$$

and the difference equation

$$y[n] = b_1 x[n-1] + b_0(x[n] + x[n-2]) \tag{4.6}$$

where

$$\begin{gathered} b_0 = \rho\frac{1+B}{2}; \quad b_1 = \rho\frac{1-B}{4}; \\ \rho = 0.001^{1/(f_0 t_{60})}; \end{gathered} \tag{4.7}$$

The damping filter has two control parameters. The brightness is controlled by B, a value between 0 and 1. Control parameter t_{60} is how long in seconds it takes to reach a decay of -60db. With a delay of one sample for all frequencies in the range that is used, the filter has tuning invariance. This is an important factor with the string tuning filter.

A part of the ProcessCallback method containing the damping filter operations is seen below.

Listing 4.7: Second-Order Damping Filter Code Snippet

```
float dampRho = pow(0.001f, (1.0f / (data->p[P_FREQ] * data->p[P_DAMP_SEC
    ])));
float damph0 = (1.0f + data->p[P_DAMP_B]) * 0.5f;
float damph1 = (1.0f - data->p[P_DAMP_B]) * 0.25f;

for (unsigned int n = 0; n < length; n++)
   {
    data->outDamp[data->n] = dampRho * (damph1*(data->outDel[data->n] +
        data->outDel[(data->n - 2) & 0x3FF]) + damph0 * data->outDel[(data
        ->n - 1) & 0x3FF]);
    }
```

String-Tuning Filter The string-tuning filter makes up for the lack of precision associated with the integer delay. The filter performs linear interpolation using the first-order transfer function

$$H_\eta[z] = (1-\eta) + \eta z^{-1} \tag{4.8}$$

and the difference equation

$$y[n] = (1-\eta)x[n] + \eta x[n-1] \tag{4.9}$$

where η is the rounding error in samples. This makes the filter linearly interpolate between the current and past input sample.

Listing 4.8: C++ String Tuning

```
data->nStringTuning = SAMPLERATE / data->p[P_FREQ] - data->N - 1.0f;

for (unsigned int n = 0; n < length; n++)
   {
    data->outStringTuning[data->n] = (1.0f - data->nStringTuning)*data->
        outDamp[data->n] + data->nStringTuning*data->outDamp[(data->n - 1)
         & 0x3FF];
    }
```

Dynamic Level Filter A dynamic level low-pass filter is used to make the gain equal for all fundamental frequencies. The transfer function is given by

$$H_{L,\omega_1}[z] = \frac{\tilde{\omega}_1}{1+\tilde{\omega}_1} \frac{1+z^{-1}}{1-(\frac{1-\tilde{\omega}_1}{1+\tilde{\omega}_1})z^{-1}} \tag{4.10}$$

with $\tilde{\omega}_1 = \pi f_0$. The difference equation becomes

$$\begin{aligned} y[n] = c_0(x[n] + x[n-1]) + c_1 y[n-1] \\ c_0 = \frac{\tilde{\omega}_1}{1+\tilde{\omega}_1}; \quad c_1 = \frac{1-\tilde{\omega}_1}{1+\tilde{\omega}_1} \end{aligned} \tag{4.11}$$

On top of this, the input to this filter, from the loop delay, and the output is mixed together with the formula

$$\begin{aligned} L \cdot L_0(L)x[n] + (1-L)y[n] \\ L_0(L) = L^{1/3} \end{aligned} \tag{4.12}$$

The parameter L is controlled to achieve the dynamic level. It is set between 0 and 1, where a higher value results in a more energetic pick.

A code snippet of the dynamic level filtering can be seen below, where the exposed parameter `P_DYNLEV_L` can change at any sample.

Listing 4.9: C++ Dynamic Level

```cpp
float w = (kPI*data->p[P_FREQ]) / SAMPLERATE;
float enerC0 = w / (1.0f + w);
float enerC1 = ((1.0f - w) / (1.0f + w));
float L0 = pow(data->p[P_DYNLEV_L], (1.0f / 3.0f));

for (unsigned int n = 0; n < length; n++)
    {
    data->outDynLev[data->n] = enerC0 * (data->outDel[data->n] + data->
        outDel[(data->n-1) & 0x3FF]) + enerC1 * data->outDynLev[(data->n -
        1) & 0x3FF];
    data->outDynLev[data->n] = data->p[P_DYNLEV_L] * L0*data->outDel[data->
        n] + (1.0f - data->p[P_DYNLEV_L])*data->outDynLev[data->n];
     }
```

Filling the Buffer For each channel, the value of the dynamic level filter is inserted, the `data->n` index is incremented and the bit shifting is applied. A mixing flag `wetTarget` is multiplied to final value, which is 1 if the application is running, is not muted and is not paused. The code for this part can be seen below.

Listing 4.10: C++ Buffer Filling

```
for (unsigned int n = 0; n < length; n++)
{
    for (int i = 0; i < outchannels; i++)
    {
        outbuffer[n * outchannels + i] = data->outDynLev[data->n] *
            wetTarget;
    }
}
data->n = (data->n+1) & 0x3FF;
```

4.3.2 Audio Effects

The audio effects are implemented as a separate native plugin. The exposed parameters here are the pregain of the overdrive, the center frequency of the wah wah and the mix of the wah wah.

Overdrive The overdrive (or distortion) is a simple operation that adds a gain to a single input value. The effect is adapted from Zölzer [40]. No previous values are needed. The equation is given by the function

$$f(x) = \begin{cases} 2x & \text{for } 0 \leq x \leq 1/3 \\ \frac{3-(2-3x)^2}{3} & \text{for } 1/3 \leq x \leq 2/3 \\ 1 & \text{otherwise} \end{cases} \tag{4.13}$$

This is implemented as a float function in C++ since it only needs to return a value and not save any as delayed values.

Listing 4.11: C++ Soft Clipping Overdrive

```
const float th = 1.0f / 3.0f;

float Overdrive(float x)
{
   float y = 0.0f;
   if (fabs(x) < th)
      return 2.0f * x;
   else if (fabs(x) > 2.0f * th)
   {
      if (x > 0)
         return 1.0f;
      if (x < 0)
         return -1.0f;
   }
   else
   {
      if (x > 0)
```

```
            return (3.0f - (2.0f - 3.0f*x)*(2.0f - 3.0f*x)) / 3.0f;
        else
            return -(3.0f - (2.0f - fabs(x) * 3.0f)*(2.0f - fabs(x) * 3.0f))
                / 3.0f;
    }
}
```

Before the value is used as input in the function, a pregain is applied to it. This pregain parameter is the exposed parameter in Unity and is calculated as 10^{2d} where d is the 0 to 1 pregain. The overdrive in combination with the pregain as seen in the code below.

Listing 4.12: C++ Pregain

```cpp
float preGain = pow(10, 2.0f*data->p[P_DRIVE]);

x = x*preGain;
x = Overdrive(x);
```

The initial x is taken from the input buffer. It is the value put into the buffer from the Extended Karplus-Strong, when the wah wah and drive plugin follows it in the audio chain.

Wah Wah The wah wah acts as a band-pass filter with a moving center frequency. It also includes a low-pass and high-pass component. Different wah wah effects exists and this one is adapted from Marshall [41]. The difference equations, where the subscript denotes the type of filter (low-pass, band-pass and high-pass), is given by

$$\begin{aligned} y_l[n] &= F_1 y_b[n] + y_l[n-1] \\ y_b[n] &= F_1 y_h[n] + y_b[n-1] \\ y_h[n] &= x[n] - y_l[n-1] - Q_1 y_b[n-1] \end{aligned} \tag{4.14}$$

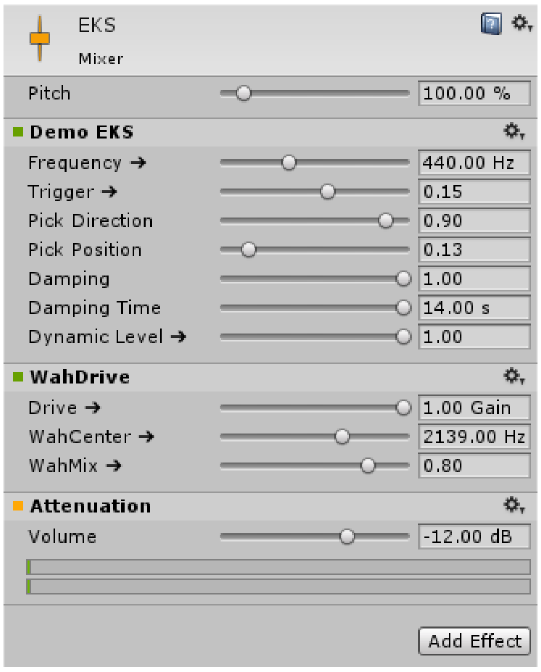
where

$$F_1 = 2sin(\pi f_c / f_s); \quad Q_1 = 2d \tag{4.15}$$

The center frequency is f_c, f_s is the sampling rate and d is a damping factor. Single float variables are used to store the delayed values.

Listing 4.13: C++ Wah Wah

```cpp
float F1 = 2 * sin((kPI*data->p[P_WAHCENTER]) / SAMPLERATE);

for (unsigned int n = 0; n < length; n++)
{
    data->delY = F1 * (inX - data->delLow - 0.4*data->delY) + data->delY;
    data->delLow = F1 * data->delY + data->delLow;
```



EKS
Mixer

Pitch 100.00 %

Demo EKS

Frequency ➔ 440.00 Hz
Trigger ➔ 0.15
Pick Direction 0.90
Pick Position 0.13
Damping 1.00
Damping Time 14.00 s
Dynamic Level ➔ 1.00

WahDrive

Drive ➔ 1.00 Gain
WahCenter ➔ 2139.00 Hz
WahMix ➔ 0.80

Attenuation

Volume -12.00 dB

Figure 4.4: The compiled native audio plugin added to a mixer in Unity. Parameters exposed to C# scripting have an arrow to the right of them. The possible values of the sliders are set in the C++ framework.

```

    x = data->p[P_WAHMIX] * data->delY + (1.0f - data->p[P_WAHMIX])*inX;
}
```

A constant of 0.4 is used for the damping Q_1 while the center frequency and mix is exposed to a mixer in Unity.

4.3.3 Controlling the Guitar Saber

Once compiled, the native plugin is added to a mixer object. Figure 4.4 shows the audio chain of the Extended Karplus-Strong (Demo EKS) followed by the audio effects (WahDrive). The parameters can be controlled by using the sliders and be exposed to scripting, which is indicated by the arrows. The possible values of the sliders are set in the C++ script in a "RegisterParameter" method. See appendix A.4 lines 72 through 78. Selected code snippets are used to explain how the synthesis is controlled and visualized here, while the whole project can be found in appendix A.2.

This section will go through how each parameter is controlled from Unity interactions and scripting.

The `Trigger` is used to signal when a new pick occurs. It refills the excitation burst with N samples depending on the frequency and resets the noiseIdx counter, so the model is excited again. Since the parameter is a floating point number, it can not trigger the refill and reset like the "bang" object of pure data would. At the same time, the Native Audio SDK only registers changes in parameters, which means sending a repeated 1 does not work. A simple workaround is achieved by switching between sending -1 and 1 from a C# script and 0 when it should be turned off. The code below summarises the essential parts of how this works.

Listing 4.14: C# Code for Guitar Pick Trigger

```csharp
private float triggerSwitch = 1f;

private void Awake()
{
       audioMixer.SetFloat("Trigger", 0.0f);
}

void OnCollisionEnter(Collision col)
{
    if (col.transform.tag == "Saber")
    {
    audioMixer.SetFloat("Trigger", triggerSwitch = -triggerSwitch);
    }
}

private void OnApplicationQuit()
{
    audioMixer.SetFloat("Trigger", 0.0f);
}
```

To ensure there is no sound when the application starts, the trigger is set to zero in `Awake`, which runs before the game starts. The trigger is also set to zero when the game ends in `OnApplicationQuit`. By changing between 1 and -1, the `triggerSwitch` variable is passed to the native plugin which registers it as a change. The code is attached to the right blue saber and the triggering happens when it collides with another objects tagged as a "saber", which is the left red saber.

Frequency As mentioned, the frequency is determined by the location of the collision along the sabers. Figure 3.1 shows this in the top as a two-dimensional array of frequencies and the bottom figure shows the implementation with the frets. The collision system determines when one saber is colliding with the other, but not which part of it. Fortunately, the collision system supplies the contact point of the impact as a 3D point (`Vector3` type in Unity). By calculating the distance between the handle and the tip of the saber along with the ratio used to place the frets, the frequency is determined. This can be a continuous value in the scale, but is for the Guitar Saber discretized to fit the tuning.

The tuning of the guitar can be changed. Using the guitar as inspiration, only the open string turnings are set and the rest are calculated from it. The amount of

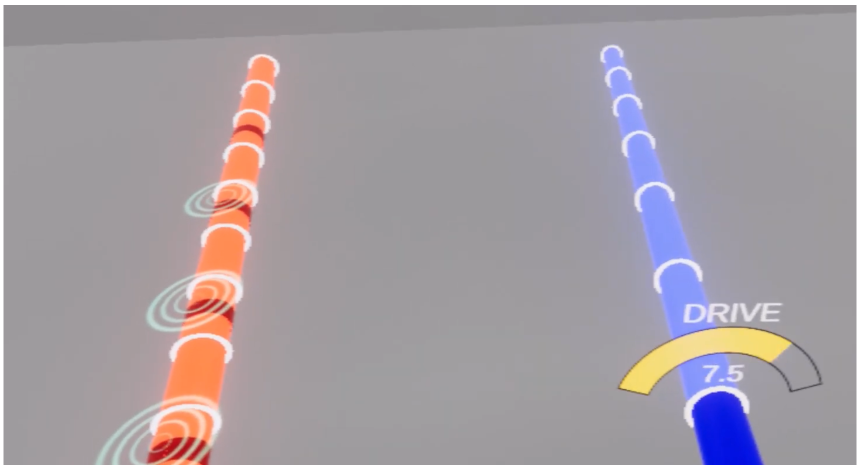


Figure 4.5: The Guitar Saber implementation where the drive (pregain) is being changed. The UI for the drive is seen by holding down a trigger.

frets and strings can also also be set, though a higher number makes it harder to correctly trigger a desired note.

Dynamic Level The dynamic level is set using the velocity of the right blue saber. Velocity of the saber is already calculated as part of the base game, so this variable is reused here. Testing it for possible values, it was observed that the largest values were around 20, so the number sent to the mixer is scaled by a division of 20 and clamped between 0 and 1.

Drive and WahMix The drive and mix of the wah wah is the only interaction that requires the use of a button. Figure 4.5 shows it in use. A button is held down, and a UI interface is seen above the handle of the saber. By twisting the saber in a positive direction the value decreases, while twisting it in a negative direction increases it. The UI shows the effect name, the current value and a color indicator for the value. While twisting the saber, the UI component remains static while the color indicator changes. Once the button is let go, the UI is not shown anymore.

Other ways to control the value was created before settling on this one. One was to change it without using the button. This was done by twisting the saber like throttling a motorbike. Unfortunately, the throttling could be accidentally triggered while using the Guitar Saber to play on.

Wah Center The wah center is controlled by the angle of the left red saber. The angle is calculate using the bottom of the saber as origo. When a pick occurs, the angle is set as the initial angle and upon angling the saber further up or down, the center frequency is changed. The complete function is seen in the code below.

Listing 4.15: C# Code for Guitar Pick Trigger

```
private void WahCenter(float angle)
{
    float angleDiff = Mathf.Clamp(angle - initialAngle, -30f, 20f);
    newWahCenter = 2000f + 50f * angleDiff;
    audioMixer.SetFloat("WahCenter", newWahCenter);
}
```

The center frequency has a minimum value of 500 Hz and a maximum of 3000 Hz. Starting with a default of 2000 Hz, the center value in Hz can increase to 3000 by with a positive difference in angle of 20 degrees, while it can decrease to 500 Hz with a negative difference in angle of 30 degrees.

Discrete Glissando The discrete glissando is also implemented using Unity's collision system. Instead of detecting the entering of a collision, the glissando is affected at each frame the sabers stay in a state of collision. The frequency is then calculated similarly to how it is upon entering a collision, but the difference is that a new excitation is not triggered. Instead, the frequency only changes and the delay loop along with it.

Visual Indicators Visual indicators are created to see which position has been struck. This is done using particle effects of a circle enlarging to resemble an outgoing wave like a drop in water as seen in figure 4.6.

It is created using Unity's particle system and the speed at which the waves move increase with higher tones to resemble an increase in frequency. On a change of strings, the position of the indicator also moves to fit in line with the string indication on the right blue saber.

Novice Version By decreasing the amount of frets and strings it can be easier to have an overview of the tone possibilities. For players unfamiliar with the layout of a guitar, the meaning of the frets and the strings might not indicate which tones to play. Because of this, a novice version of the instrument is also implemented. The novice Guitar Saber includes two strings with eight notes each, that are mapped to a scale. There is an octave in difference between the two strings. Removing the connection with the semitonal guitar layout takes away one of the visual aspects that makes it the most familiar to musicians, but makes it easier for novices to play along to a song, if the key of the saber and song matches. The change is determined by a boolean when initializing the instrument as seen in the code example below.

Listing 4.16: C# Code for Novice Guitar Saber

```
if (novice)
{
    stringCount = 2;
    fretCount = 9;
```

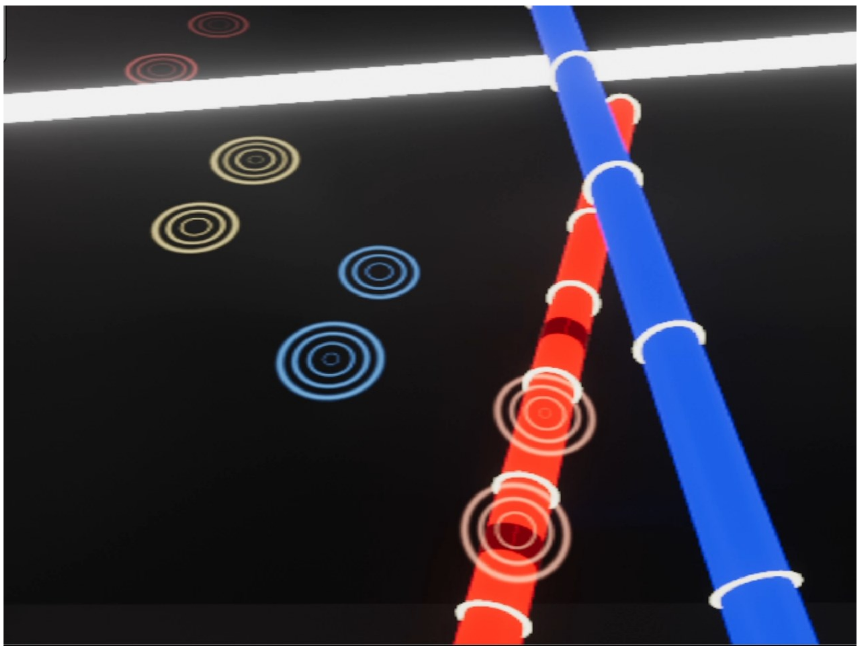


Figure 4.6: An early version of the Guitar Saber. The particle effects show which notes have been hit recently.

```
    frequencies = new float[,] { {0f, 116.54f, 130.81f, 138.59f, 155.56f,
        174.61f, 185.00f, 207.65f, 233.08f }, { 0f, 233.08f, 261.63f,
        277.18f, 311.13f, 349.23f, 369.99f, 415.30f, 466.16f } };
}
```

Making this change is simple, as it only requires bypassing the string tuning for a fixed scale (hardcoded Bb minor here) and making sure the position inlays are not instantiated.

4.4 Mechanical Oscillator for Friction

Implementing the friction synthesis follows the same procedure as the Extended Karplus-Strong. A prototype was created in MatLab (see appendix A.5) before it was coded in C++ for the native plugin (appendix A.6) and controlled through a mixer in Unity.

4.4.1 Non-Linear Friction Synthesis

The friction synthesis implementation can be split into four parts. The initialization of state-variables and constants, processing using known variables, approximation using Newton-Rhapson and updating the variables to include the value from Newton-Rhapson.

State-Variables and Constants As described in section 2.4.2 on low-level friction and the mechanical oscillator, the system can be decomposed into state space form. From the mechanical oscillator in equation 2.2 and using the bilinear transform to discretize the K method as seen in equation 2.10, the matrices become

$$\begin{aligned} A = H(\alpha I + A^{(K)}) = det_H \cdot \begin{bmatrix} \alpha^2 + \alpha g - \omega^2 & 2\alpha \\ -2\alpha\omega^2 & \alpha^2 - \alpha g - \omega^2 \end{bmatrix} \\ b = HB = det_H \cdot \begin{bmatrix} \frac{1}{m} \\ \frac{\alpha}{m} \end{bmatrix} \\ det_H = \frac{1}{\alpha^2 + \alpha g + \omega^2} \end{aligned} \tag{4.16}$$

where $A^{(K)})$ is from the state-variable as seen in equation 2.8 and the leftmost A is the reduced form. Upon triggering the friction the constants in the transformation matrices are computed once, as they do not change unless a parameter like the frequency changes. The code below shows how this works when 3 resonators are used in a slightly optimized version.

Listing 4.17: C++ Computation of Transformation Matrice Variables

```cpp
for (int i = 0; i < 3; i++)
{
    omega_r[i] = 2.0f * m_pi*freq[i];
   g_r[i] = omega_r[i] / q_r;
   detTemp = (Fs*Fs + g_r[i] * Fs / 2.0f + (omega_r[i] * omega_r[i]) / 4.0
       f);

   data->A_r[0][0][i] = 1.0f / detTemp * (detTemp - (omega_r[i] * omega_r[
       i]) / 2.0f);
   data->A_r[1][0][i] = 1.0f / detTemp * (-Fs * omega_r[i] * omega_r[i]);
   data->A_r[0][1][i] = 1.0f / detTemp * Fs;
   data->A_r[1][1][i] = 1.0f / detTemp * (2.0f * Fs*Fs - detTemp);

   data->b_r[0][i] = (1.0f / m_r)*(1.0f / (4.0f * detTemp));
   data->b_r[1][i] = (1.0f / m_r)*(1.0f / (4.0f * detTemp)) * 2.0f * Fs;

   data->bv_r += data->b_r[1][i];
}
```

A three-dimensional array is used for the resonators, so they can indexed over through a loop. The matrices are computed on a trigger or frequency change, similar to the re-excitation from the Guitar Saber. To compute the K components, `data->bv_r` is summed in this loop, which is a sum of the velocities.

For the bow, the frequency is set to zero as the sound resonating from the bow is neglected. It is used as the exciter in this system and only has a single instance of transformation matrices.

Other constants affected by a change in normal force, Stribeck velocity or frequency are also set here. The code below shows an excerpt of these.

Listing 4.18: C++ Initial Constants for Friction Synthesis

```cpp
data->f_s = mu_s * data->p[P_FN];
data->f_c = mu_d * data->p[P_FN];
data->Zba = 0.7f * data->f_c / sig0;

data->fe_b = data->f_c + (data->f_s - data->f_c)*expf(-(data->p[P_STRIBECK
    ] / v_s)*(data->p[P_STRIBECK] / v_s)) + sig2 * data->p[P_STRIBECK];

data->K2 = 1.0f / (2.0f * Fs);

data->bv_b = data->b_b[1];
data->bv = data->bv_b + data->bv_r;

data->K1 = -data->bv / (1.0f + sig2 * data->bv)*(sig0 / (2.0f * Fs) + sig1
    );
```

The values for all σ, masses and more are set as unexposed constants. The K constants are explained in the paragraph on Newton-Rhapson.

While the system can continue without resetting state variables like the compression x and velocity v, there is a risk of it becoming unstable. To limit this, the option to reset it is added to the native plugin. This can be used on the first instance of interaction. The code below shows the reset.

Listing 4.19: C++ Initial Constants for Friction Synthesis

```cpp
if (!data->stateOn)
{
   data->stateOn = true;
   data->h0 = 0.0f;
   data->zPrev = 0.0f;
   data->yPrev = 0.0f;
   data->f_tot_b = 0.0f;
   data->f_tot_r = 0.0f;

   for (int j = 0; j < 3; j++)
   {
      data->xv_r[0][j] = 0.0f;
      data->xv_r[1][j] = 0.0f;
   }
   data->xv_b[0] = 0.0f;
   data->xv_b[1] = 0.0f;
}
```

A simple boolean is used to reset the data if it is the first interaction. The boolean `stateOn` is set to false when the value of the trigger (used like in Guitar Saber) is zero. The state values are set in `data->xv_b` for the bow and `data->xv_r` for the resonators.

Computable Part In the computation of each sample, the computable parts are first applied to the state variables as temporary values, that are used in the Newton-Rhapson.

Listing 4.20: C++ Computable Parts for Each Sample

```cpp
for (unsigned int n = 0; n < length; n++)
{
   // Resonators
   v_rSum = 0.0f;
   for (int j = 0; j < 3; j++)
   {
      tempX = data->A_r[0][0][j] * data->xv_r[0][j] + data->A_r[0][1][j] *
           data->xv_r[1][j] + data->b_r[0][j] * data->f_tot_r;
      data->xv_r[1][j] = data->A_r[1][0][j] * data->xv_r[0][j] + data->A_r
          [1][1][j] * data->xv_r[1][j] + data->b_r[1][j] * data->f_tot_r;
      data->xv_r[0][j] = tempX;

      v_rSum += data->xv_r[1][j];
   }

   // Bow
   tempX = data->A_b[0][0] * data->xv_b[0] + data->A_b[0][1] * data->xv_b
       [1] + data->b_b[0] * data->f_tot_b;
   data->xv_b[1] = data->A_b[1][0] * data->xv_b[0] + data->A_b[1][1] *
       data->xv_b[1] + data->b_b[1] * data->f_tot_b;
   data->xv_b[0] = tempX;

   v_bSum = data->xv_b[1];

   // Computable parts
   float z_Ti = data->zPrev + 1.0f / (2.0f * Fs)*data->yPrev;

   float w = data->random.GetFloat(-1.0f, 1.0f) *abs(v_bSum); // * f_n

   v_Ti = 1.0f / (1.0f + sig2 * data->bv)*((v_bSum + data->bv_b * (data->
       fe_b - sig0 * z_Ti)) + (-v_rSum - data->bv_r * (sig0*z_Ti)));

   // NEWTON-RHAPSON HERE //

   // COMPLETE SYSTEM HERE //
}
```

The first parts of the resonators and bow follow equation 2.10 without the friction force. The computable part of z, `z_Ti`, is found using the trapezoidal rule as and removing the unknown part $\dot{z}[n]$

$$\tilde{z}[n] = z[n-1] + \frac{1}{\alpha}\dot{z}[n-1] \tag{4.17}$$

For the velocity, this is more complicated, as it it needs to take into account all the computable parts from the friction force equation 2.4 as well as the state-variables and transformations. Combining each resonators state-variables together simplifies the equation to

$$\tilde{v}[n] = \frac{1}{1+\sigma_2 b}((\dot{x}^{(b)} + b(2)^{(b)}(f_e^{(b)} - \sigma_0 \tilde{z}[n])) - (\dot{x}^{(r)} + b(2)^{(r)}\sigma_0 \tilde{z}[n])) \tag{4.18}$$

where $b(2)$ is value transforming the velocity in (b) of equation 2.10. It is denoted by either a b for bow, (r) for resonator or nothing, when it is the sum of both. The random noise in `w` is attenuated by the absolute value of the latter.

Newton-Rhapson approximation To approximate the value of $\dot{z}$ the values of $\tilde{z}$ and $\tilde{v}$ are used with their K component and the process described in section 2.4.5. The K components are the missing transformations of the unknown friction forces that makes $\tilde{v}$ and $\tilde{z}$ become v and z. For z this value is found by looking at equation 4.17 without removing the unknown component $\dot{z}$.

$$\begin{aligned} z[n] &= \tilde{z}[n] + \frac{1}{\alpha}\dot{z} \\ z[n] &= \tilde{z}[n] + K(2)\dot{z} \end{aligned} \tag{4.19}$$

For the velocity, this includes a couple more components.

$$\begin{aligned} v[n] &= \tilde{v}[n] + \frac{b}{1+\sigma_2 b}(\sigma_0\alpha + \sigma_1)\dot{z} \\ v[n] &= \tilde{v}[n] + K(1)\dot{z} \end{aligned} \tag{4.20}$$

Listing 4.21: C++ Newton-Rhapson v and z Approximations

```
const float errMax = 1e-13f;
float count = 1;
float err = 99;
while (err > errMax && count < 10 && wetTarget == 1.0f)
{
   float vNew = v_Ti + data->K1 * data->h0;
   float zNew = z_Ti + data->K2 * data->h0;
}
```

The new values for v and z (`vNew` and `zNew`) are updated each round of the Newton-Rhapson loop as the approximation of $\dot{z}$ changes in h_0. A maximum error is set as well as a maximum number of iterations to avoid an infinite loop. The `wetTarget` is the same as in the Guitar Saber.

To use the Newton-Rhapson approach described in section 2.4.5, the new values for v and z are computed for this iteration of the while loop. The code can be seen below.

Listing 4.22: C++ Mean Bristle Velocity

```cpp
// Find Zss
float Zss = (sign(vNew) / sig0)*(data->f_c + (data->f_s - data->f_c)*expf
    (-(vNew / v_s)*(vNew / v_s)));
float aNew = 0.0f;
// Find alpha tilde
if (sign(zNew) != sign(vNew))
   aNew = 0.0f;
else if (fabs(zNew) < data->Zba)
   aNew = 0.0f;
else if (fabs(zNew) > Zss)
   aNew = 1.0f;
else
   aNew = 0.5f*(1.0f + sin(m_pi*((zNew - 0.5f*(Zss + data->Zba)) / (Zss -
       data->Zba))));

// Compute derivatives needed for derivative of g
// Zss / v, a / z, a / v to compute dotz / v and dotz / z
float ZssvDeri = -sign(vNew) * (2.0f * vNew) / (sig0*v_s*v_s) * (data->f_s
    - data->f_c)*exp(-(vNew / v_s)*(vNew / v_s));

float azDeri = 0.0f;
float avDeri = 0.0f;

if ((data->Zba < fabs(zNew)) && (fabs(zNew) < Zss) && (sign(vNew) == sign(
    zNew)))
{
   float temp = 0.5f*m_pi * cos(m_pi*(zNew - 0.5f*(Zss + data->Zba)) / (
       Zss - data->Zba));
   azDeri = temp * (1.0f / (Zss - data->Zba));
   avDeri = temp * ((ZssvDeri*(data->Zba - zNew)) / ((Zss - data->Zba)*(
       Zss - data->Zba)));
}
```

The root-finding can then start with equation 2.13 and using the steady velocity friction force in equation 2.3. The derivative becomes

$$\frac{\partial g}{\partial h} = \frac{\partial \dot{z}}{\partial v}K(1) + \frac{\partial \dot{z}}{\partial z}K(2) - 1 \tag{4.21}$$

where the K components are previously found and the derivatives according to x and v are taken from Rocchesso [34].

Listing 4.23: C++ Newton-Rhapson Root Finding

```cpp
// Compute g
float gNom = vNew * (1.0f - aNew * zNew / Zss) - data->h0;

float derZ = -(vNew / Zss)*(zNew*azDeri + aNew);
float derV = 1.0f - zNew * (((aNew + vNew * avDeri)*Zss - aNew * vNew*
    ZssvDeri) / (Zss*Zss));
```

```
float gDeri = derV * data->K1 + derZ * data->K2 - 1.0f;

float h1 = data->h0 - gNom / gDeri;

err = fabs(h1 - data->h0);
data->h0 = h1;
```

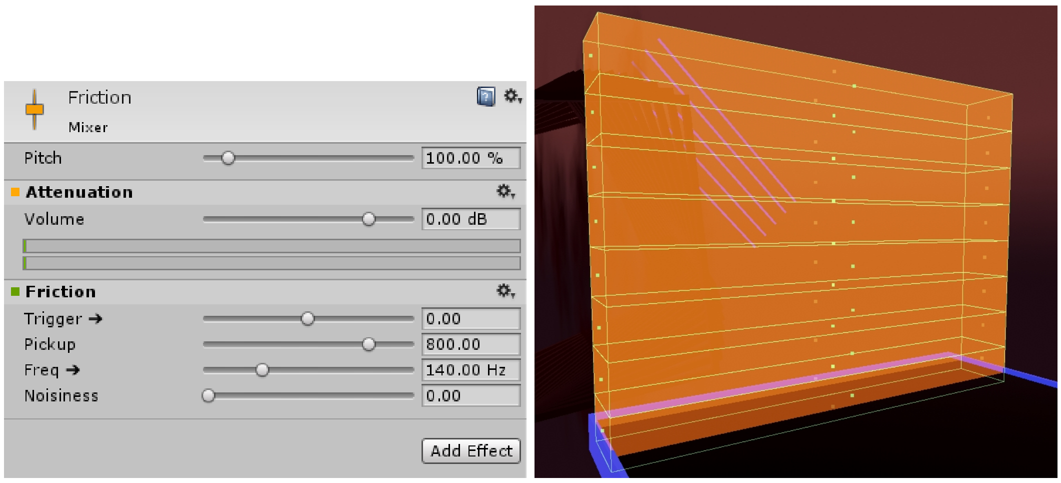
The code shows how the next value for h is found and the error is computed, converging if the error is below the maximum allowed.

Updating Variables After the Newton-Rhapson, the variables are updated to factor in the approximated value of $\dot{z}$. The actual v and z is found, which then allows for computing the friction force. If other forces are acting on the bow or resonator, they are taken into account as the total force acting on the object. The resonators and bow state-variables are then updated as seen in the code below.

Listing 4.24: C++ Updating the Variables

```
float dotz = data->h0;
float v = v_Ti + data->K1 * dotz;
float z = z_Ti + data->K2 * dotz;

float f_fr = sig0 * z + sig1 * dotz + sig2 * v + data->p[P_NOISE] * w;

data->f_tot_b = data->fe_b - f_fr;
data->f_tot_r = f_fr;

x = 0.0f;
for (int j = 0; j < 3; j++)
{
   data->xv_r[0][j] = data->xv_r[0][j] + data->b_r[0][j] * data->f_tot_r;
   data->xv_r[1][j] = data->xv_r[1][j] + data->b_r[1][j] * data->f_tot_r;
   x += data->xv_r[0][j];
}

data->xv_b[0] = data->xv_b[0] + data->b_b[0] * data->f_tot_b;
data->xv_b[1] = data->xv_b[1] + data->b_b[1] * data->f_tot_b;

data->fader = data->p[P_TRIGGER] == 0.0f ? fmax(data->fader - 0.001f, 0.0f
    ) : fmin(data->fader + 0.001f, 1.0f);

for (int i = 0; i < outchannels; i++)
   outbuffer[n * outchannels + i] = data->p[P_PICKUP] * x * wetTarget *
       data->fader;
```

An additional smoothing component with `data->fader` is added to smoothly transition when ending and starting the synthesis. The audio is filled into the buffer and multiplied by a pickup that acts as a gain. This value is exposed as a parameter in Unity.



Friction
Mixer
Pitch 100.00 %
Attenuation
Volume 0.00 dB
Friction
Trigger ➔ 0.00
Pickup 800.00
Freq ➔ 140.00 Hz
Noisiness 0.00

(a) (b)

Figure 4.7: The mixer controlling the friction and the wall with separated colliders.

4.4.2 Controlling Wall Synthesis

Several of the friction synthesis parameters are hardcoded in the native plugin leaving three exposed parameters alongside the trigger. As seen in figure 4.7a, these parameters are the frequency, a pickup and noisiness. The Stribeck velocity can also be exposed, though the value must be kept in a tight range as the system otherwise becomes unstable. In this implementation the range found to be between 0.1 and 0.3 while the frequency still can change.
The friction is mapped to a scale on a wall by dividing the wall into eight colliders fitting an octave on a scale. When a saber enters a collider, the name of the collider is converted to a frequency that is sent to the mixer. Moving the saber from one collider to the next while staying in contact with the whole wall gives a discrete glissando effect like with the Guitar Saber. Figure 4.7b shows how these colliders are placed separate in the wall. When the wall scales, the collider scale along with it. The trigger is set to 1 (or -1) upon entering the collision and 0 when exiting.

Chapter 5

Evaluation

Four tests were conducted with more to come. The tests included 9 participants (9 men) and all went through the four tests in one session lasting 15 to 20 minutes.

The tests were conducted in the following order:

1. Assessment of the adaptive remix mode
2. Observational usability of the Guitar Saber
3. Audio User Experience of the Guitar Saber
4. Assessment of using the Guitar Saber in-game

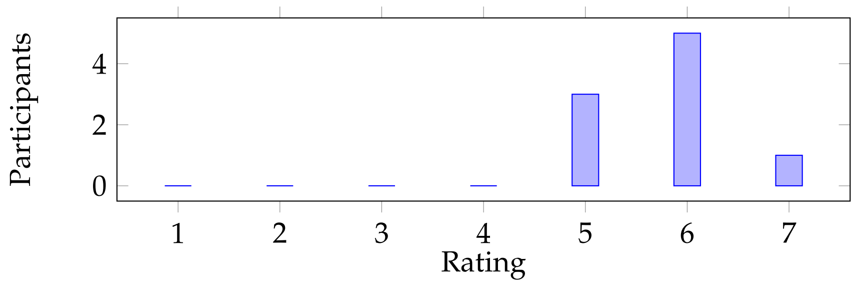
5.1 Adaptive Song Replayability

Participants played the remix mode until they had passed two possible recombination segments. All participants unlocked the first segment. Each participant was explained how to play the game, though only three had not played the original before and of those three, two had knowledge of the game. After playing the mode, they were explained that segments could change.

Only one participant did not notice that the visuals changed to fit with the remix segment. This can be due to focusing on the playing immediately in front of them. Most participants (66 %) answered that they did not know how to unlock the gate to the remixed segment. Another participant, who claimed they knew, answered that it was achieved by slicing green cubes, which is incorrect. This means that the visuals in figure 4.2 should be changed to make it clearer what is required. It can also be that participants would need to play the mode for longer than two possible segments.

The participants were asked to rate the following statement from 1 to 7, where 1 means they disagree strongly and 7 means they agrees strongly:

Playing a song where remixed segments can be unlocked will increase the chance that I will replay the song, in case I can unlock previously unheard segments.



The ratings can be seen in the bar plot above. All participants agreed to some degree, indicating that this mode could increase replayability.

5.2 VRMI: Guitar Saber

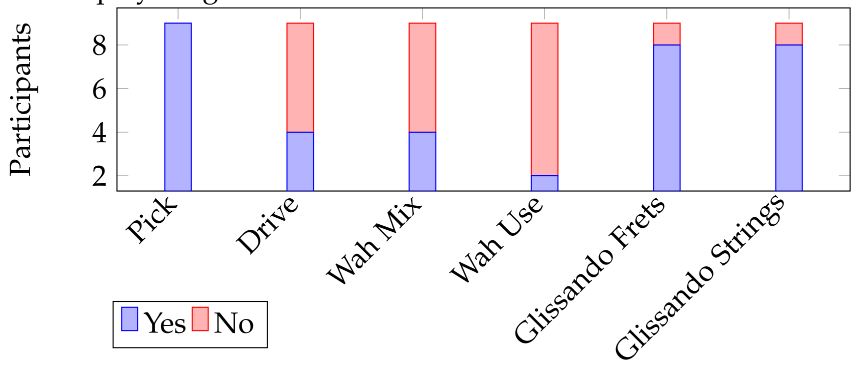
Three tests were conducted using the Guitar Saber VRMI to evaluate its stand-alone use and in-game use.

5.2.1 Observational Usability Test

The first test of the Guitar Saber was conducted as an observational test. Each participant was tasked with exploring the instrument. They were not told how to play it nor that it was based on the layout of a guitar. They were told that they were allowed to use one button, but not what it does. The button is the trigger used to control the mix of the wah wah and drive.

The stacked bar plot below shows how many participants where able to figure out the various features without instructions on how to use them.

The player figured out how to use the feature on their own.



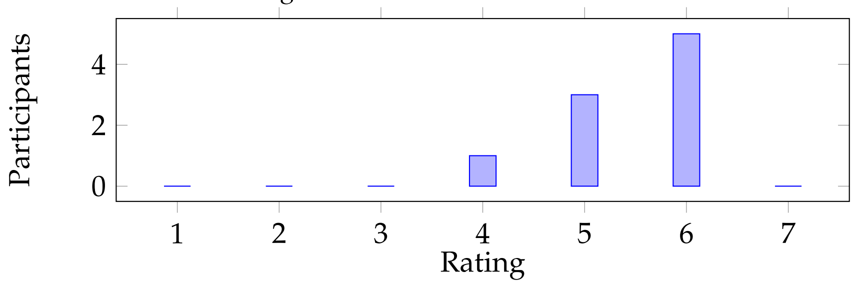
All participants quickly figured out the instrument was played by a collision of the two sabers. The glissandos were also easily found, sometimes by accident, but participants then went on to use it purposefully. Holding the trigger down and changing the mix of the drive and wah was not intuitive for most participants. They mostly held the button down and did not let it go before playing, meaning the value would change drastically on play. Some participants also pressed the trigger when colliding with the sabers, though did not have any intended effect. How to use the wah wah was only figured out by one user.

After playing for a while, the participants were presented with the novice version of the Guitar Saber while they were explained all the functionalities they had missed.

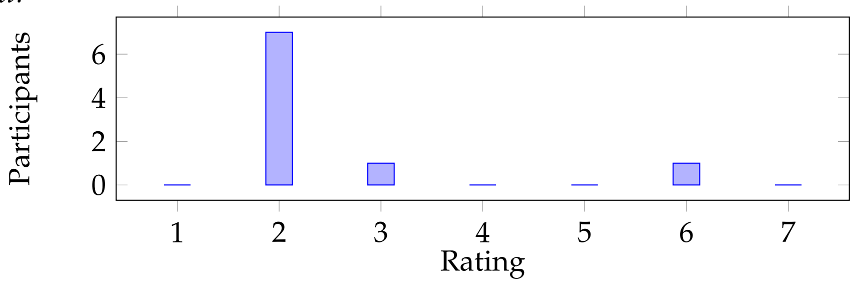
5.2.2 Audio User Experience

The second test using the Guitar Saber was based on the BUZZ: Audio User Experience (audioUX) Scale [42]. The questions are used to measure the usability of an auditory display by rating them on a scale of 1 to 7, where 1 means they disagree strongly and 7 means they agrees strongly. Five selected questions from the questionnaire were asked after participants had tried the VRMI.

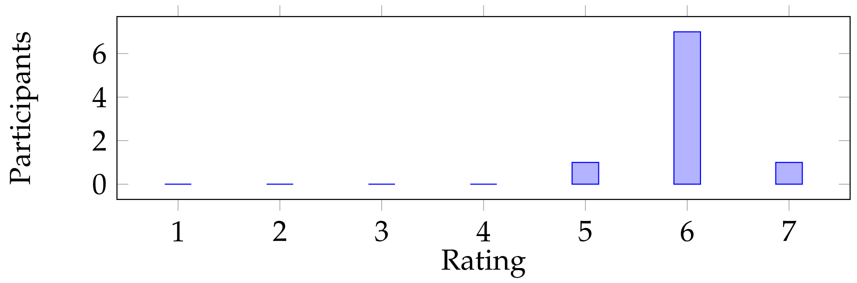
The sounds were interesting.



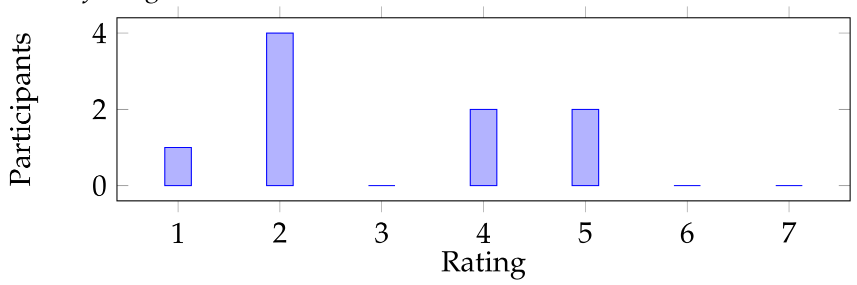
It was difficult to understand how the sounds changed from one variable to the next.



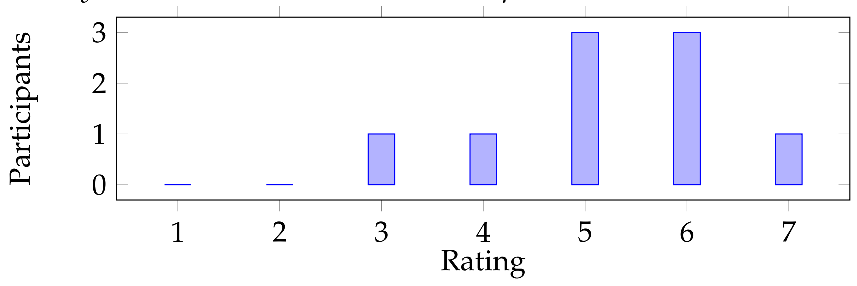
It was fun to listen to these sounds.



It was confusing to listen to these sounds.



It was easy to understand what each sound represented.

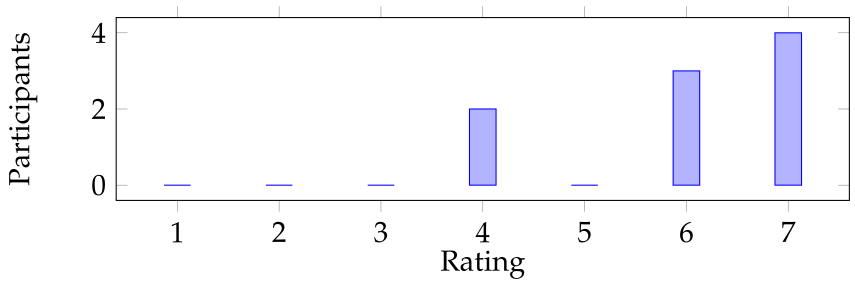


The participants generally found the sounds interesting and fun. It was more split on how easy the instrument was to understand.

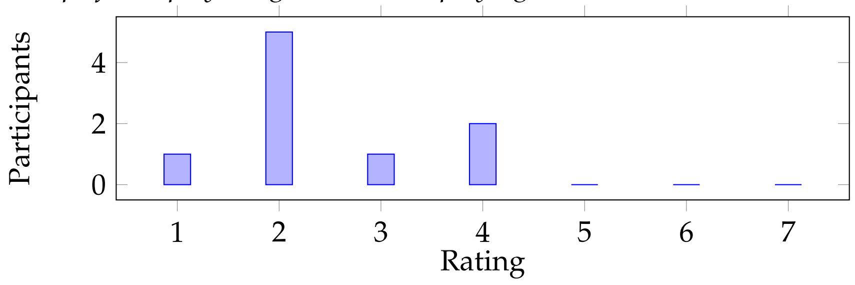
5.2.3 Assessment of In-Game Use

Each participant was asked to play a game of the original concept where it at times would be possible to use the Guitar Saber. They were then asked 4 questions in the same style as the BUZZ questionnaire.

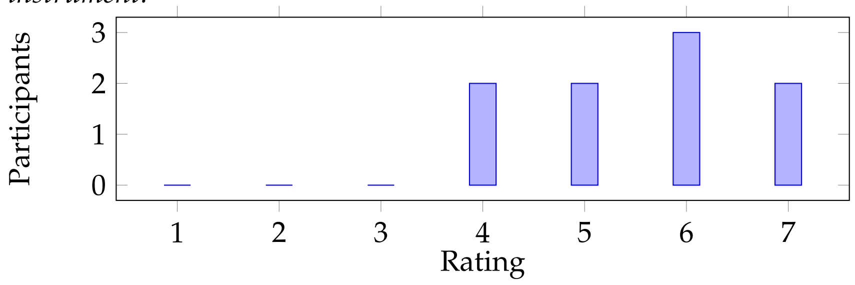
It was fun to create sounds while playing the game.



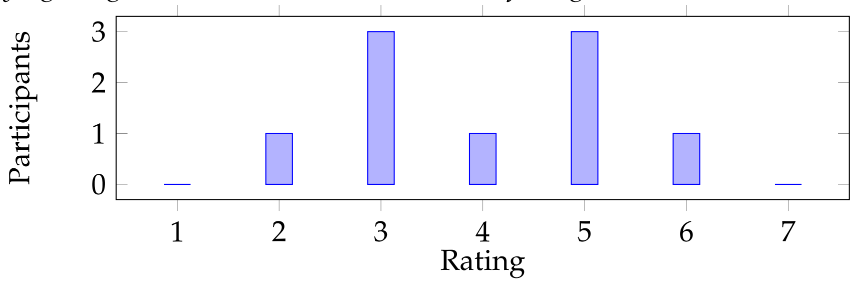
I would prefer to play the game without playing the instrument.



If I were playing the game, I would occasionally choose to play the game with the instrument.



Playing the game with an instrument was confusing.



Participants generally found it fun to use the instrument and would occasionally use it to play the game. It confusing to some to use this new instrument while in-game.

Chapter 6

Discussion

This section will look at what worked, what did not, improvements and further possibilities with each feature.

6.1 Adaptive Remix Mode

Although 9 participants are not enough to conclude anything with certainty, it was clear the participants found replayability value in the mode. This probably stems from the being curious about what other parts are possible to hear and that unlocking the remixed parts also mean that the player is performing better. As noted from Koji Kondo in section 2.3.1, the dynamic music has the ability to add new surprises and increase gameplay enjoyment.

Participants visually noticed the change, but did not understand why it happened or what they could do to unlock it. This means that either the visuals must become more user-friendly or that there is a learning curve to understanding what needs to be achieved. Since it is a simple task, it is probably the former. Several version can be implemented and tested to see what works the best.

There are many more possibilities with the adaptive remix and the gates that are used to unlock them. This version includes two condition examples; a minimum point value and a minimum streak. It can be changed to many different conditions or a combination of conditions. It can even be made into a puzzle by not showing what the condition is but only whether it has been met. This way, a player can replay the same song multiple times while trying different combinations to see what unlocks the gate. This can end up affecting the goal of a gate challenging the player to perform better, but can have other positives. If very hard puzzles are created, it might lead to communities working together to unlock the gates or if song with a hard puzzle is available to all players at the same time, there might arise a race to be the first to unlock the gate. The audio of the unlocked remix segment can then be viewed as a prize for the player.

6.2 Adaptive Endless Mode

The implementation of the endless version shows that it is possible to create a mode with endless playback. This can be useful for players who use the game as a workout routine, but requires testing from participants who has done this beforehand. The implementation can also use more songs as well as a beat mapping specifically created as a workout routine.

Additional features can be implemented to keep track of the players workout, like a calorie counter. The points obtained for each song can be stored to create highscore that can be compared to versions without the endless mode. A common feature in video games is playing against a previous version of one self. This version is often called a "ghost" mode. Since there is no multiplayer function, the player can instead play against the ghost, which can be a recording of how many points the player previously has had at times in a song.

From a technical standpoint, the counter controlling the audio and the beat mapping should be the same to avoid the risk of getting out of sync. One way of achieving this is by using Unity's function `PlayScheduled` to stich audio files together sample-accurately. This will eliminate the use of FMOD. The built-tin functions runs on a timer based on how many samples the audio system has processed and the beat mapping can use the same timer using the double `AudioSettings.dspTime` to control the counter.

6.3 Guitar Saber

The participants were mostly positive about the Guitar Saber's sound and its use. It was easy to understand how it worked and how the sound changed from variable to the next. This indicates that using the guitar as a layout is a good choice for the instrument, though more testing is required for a better assessment.

By observing each participant's use the instrument it was clear that some features were not intuitive enough. Especially controlling the mix of the effects and how to use the wah wah. A test where participants are presented with specific tasks could be conducted to get a better understanding. In this test they were not told there was a wah wah, but by tasking them to figure out how to use it, they might look for it more systematically. Participants were quick to understand how to use the features once they were explained and used them while in-game.

Using the instrument in-game was also mostly fun and something participants saw as a possibility. Multiple even strongly agreed to it being fun. This could because it allows them to be creative and improvise something unique to their play-through, that is otherwise not possible.

Some found it confusing to play the instruments in-game. This can be due to only having played it for a few minutes before using it in-game. It can also be because some players are so focused on the task of slicing cubes that they don't notice the ability to play the instrument. Few participants missed some windows of time it was possible to play the instrument, though all participants played it at some point during the song. Having an instrument "on rails" might be too restricting for an instrument that is otherwise meant to give the player freedom to improvise.

Many improvements can be added to the VRMI. There exists many refinements to the Karplus-Strong synthesis and a vast amount of audio effects can be used instead of the wah wah and overdrive. Other string synthesis techniques can also be used. Testing what sound fits with the visuals is also important, as the overdrive was selected specifically to fit with the sabers' energetic and high-powered visuals.

The visual aspect can be improved. Some users noted that they did not understand what the circular particle effects represented. This can be changed to be part of the saber instead of floating on top of it, so the effect is like waves on the surface of the sabers instead of floating in the air above.

6.4 Friction Walls

The friction of walls, which is yet to be subjected to user testing, is more of a gimmick in the game than a fully-fledged instrument. It is simple as it is fitted to a scale and the points of interaction used to control it are large compared to the Guitar Saber. The interaction lacks expanded features the Guitar Saber has. It does not have any audio effects nor control of the dynamic level. This can be changed by mapping an angle or velocity of the saber to the Stribeck velocity or the normal force, though it needs to be in a fixed range and change gradually to avoid the system becoming unstable. A simple implementation of the friction could also be used when slicing the small cubes.

Whether the sound of friction fits with carving a wall with a lightsaber should also be a subject of user testing. The BUZZ testing would be good for such a test.

Chapter 7

Conclusion

This aim of this thesis was to enhance a VR game experience through sonic interaction and adaptive music.

A VRMI "Guitar Saber" was created with the intention of rooting it in existing VR technology and interactions. This was realized by using the same collision interactions already used in the original game concept and not using standard VR controllers. The instrument was based on existing knowledge from guitars, which was translated to a new cylindrical anatomy. Users found it fun to use both for standalone and in-game use, indicating that the initial motivations mentioned in section 1.2 yielded the desired results, though it is prone for further user testing. The string synthesis technique used was computationally efficient and no audio glitches were noticed by participants in the test or afterwards. This shows the efficiency of the Karplus-Strong algorithm and the extensions that can be used. Other synthesis techniques are also possible to add other aspects and it is possible to build upon this VRMI instrument with effects and features.

It was possible to implemented the friction carving while not visibly changing any part of the original concept, which is an ideal way to keep it rooted in VR. The synthesis is more computationally complex than the Karplus-Stong, while it can be argued that it is less flexible since it is prone to becoming unstable if parameters are not carfully selected. It should be evaluated whether this synthesis technique is worth what ends up being a small part of the game, whether the interaction should be revised away from a wall, or a simpler synthesis technique could be used that still gives the desired effect.

The adaptive songs showed how moving away from a linear song can affect the player. The goal was to research whether the remix mode could increase replayability, which the participants agreed with it achieving. An endless adaptive music mode was implemented, showing the possibilities that such a version can bring and what components are useful for creating it.

The VR game was enhanced with four features, that each added a new aspect to the concept and at the same time set out to research different hypotheses on VRMIs and the effects of adaptive music.

Bibliography

[1] *Music games lead kids to real instruments*. https://www.techradar.com/news/consoles/gaming/music-games-lead-kids-to-real-instruments-491180. Retrieved 2019/05/27. 2008.

[2] Biamonte, N. *Pop-Culture Pedagogy in the Music Classroom*. 2010. Chap. 8, pp. 133–147.

[3] Hoffmann, L. "Learning Through Games". In: *Communications of the ACM* (2009).

[4] Beat Studios. *Beat Saber*. Prague, Czech Republic, 2018.

[5] Mäki-Patola, T., Laitinen, J., Kanerva, A., and Takala, T. "Experiments with Virtual Reality Instruments". In: *Proceedings of the 2005 Conference on New Interfaces for Musical Expression*. NIME '05. National University of Singapore, 2005, pp. 11–16. URL: http://dl.acm.org/citation.cfm?id=1085939.1085946.

[6] Serafin, S., Erkut, C., Kojs, J., Nilsson, N., and Nordahl, R. "Virtual Reality Musical Instruments: State of the Art, Design Principles, and Future Directions". In: *Computer Music Journal* 40 (Sept. 2016), pp. 22–40. DOI: 10.1162/COMJ_a_00372.

[7] Willcocks, L. "Calculating the Power Output of Qui-Gon Jinnâ€™s Lightsaber". In: *Journal of Interdisciplinary Science Topics* 6 (2017).

[8] Willcocks, L. "Calculating the Power Change of a Lightsaber Due to Colour". In: *Journal of Interdisciplinary Science Topics* 6 (2017).

[9] Florent Berthaut, M. D.-C. and Hachet, M. "Interacting with 3D Reactive Widgets for Musical Performance". In: *Journal of New Music Research* 40.3 (2011), pp. 253–263. DOI: 10.1080/09298215.2011.602693. eprint: https://doi.org/10.1080/09298215.2011.602693. URL: https://doi.org/10.1080/09298215.2011.602693.

[10] Cook, P. R. "Principles for Designing Computer Music Controllers". In: *NIME*. 2001.

[11] Berthaut, F., Zappi, V., and Mazzanti, D. "Scenography of immersive virtual musical instruments". In: *2014 IEEE VR Workshop: Sonic Interaction in Virtual Environments (SIVE)*. Mar. 2014, pp. 19–24. DOI: `10.1109/SIVE.2014.7006285`.

[12] McMahan, R. P., Lai, C., and Pal, S. K. "Interaction Fidelity: The Uncanny Valley of Virtual Reality Interactions". In: *Virtual, Augmented and Mixed Reality*. Ed. by S. Lackey and R. Shumaker. Cham: Springer International Publishing, 2016, pp. 59–70.

[13] Brown, A. and Kerr, T. "Adaptive music techniques". In: (Jan. 2009).

[14] Collins, K. *Game Sound; An Introduction to the History, Theory and Practice of Video Game Music and Sound*. Aug. 2008. DOI: `10.7551/mitpress/7909.001.0001`.

[15] Collins, K. *From Pac Man to Pop Music: Interactive Audio in Games and New Media*. Jan. 2008.

[16] Aska, A. *Introduction to the Study of Video Game Music*. Lulu.com, 2017. ISBN: 1387037137.

[17] Collins, K. "An Introduction to the Participatory and Non-Linear Aspects of Video Games Audio". In: (Jan. 2007).

[18] Land, M. and McConnell, P. *iMUSE*. 1994.

[19] Valerie Kobylski Ruiting Ji, T. A.-S. and Badea, M. *Symbiosis in Multiplayer: How Journey Offers a Novel Perspective on Online Cooperative Play*. `https://gamesandaslit2017.wordpress.com/2017/03/03/symbiosis-in-multiplayer-how-journey-offers-a-novel-perspective-on-online-cooperative-play/`. [Online; accessed 17-May-2019]. 2007.

[20] Kondo, K. *Painting an Interactive Musical Landscape*. Paper presented at the annual Game Developerâ€™s Conference, San Francisco, March 4â€“ 9. 2017.

[21] Land, M. Z. and McConnell, P. N. *Method and Apparatus for Dynamically Composing Music and Sound Effects using a Computer Entertainment System*. United States Patent Number 5,315,057. 1994.

[22] Hoffert, P. *Music for New Media: Composing for Video Games, Websites, Presentation, and Other Interactive Media*. Boston: Berklee Press, 2007. ISBN: 0876390645.

[23] Ren, Z., Yeh, H., and Lin, M. C. "Example-guided Physically Based Modal Sound Synthesis". In: *ACM Trans. Graph.* 32 (). DOI: `10.1145/2421636.2421637`. URL: `http://doi.acm.org/10.1145/2421636.2421637`.

[24] Adrien, J.-M. "The missing link: Modal synthesis". In: *Representations of musical signals*. 1991, pp. 269–298.

[25] Van Den Doel, K. and Pai, D. K. "Modal synthesis for vibrating objects". In: *Audio Anectodes. AK Peter, Natick, MA* (2003), pp. 1–8.

[26] Smith, J. O. "Principles of Digital Waveguide Models of Musical Instruments". In: *Applications of Digital Signal Processing to Audio and Acoustics*. Ed. by M. Kahrs and K. Brandenburg. Boston, MA: Springer US, 2002, pp. 417–466. ISBN: 978-0-306-47042-4. DOI: 10.1007/0-306-47042-X_10. URL: https://doi.org/10.1007/0-306-47042-X_10.

[27] Smith, J. O. "A New Approach to Digital Reverberation Using Closed Waveguide Networks". In: STAN-M-31. Burnaby, B.C., Canada, 1985, pp. 47–53. URL: https://ccrma.stanford.edu/files/papers/stanm31.pdf.

[28] Fontana, F. and Rocchesso, D. "A new formulation of the 2D-waveguide mesh for percussion instruments". In: (May 2019).

[29] Essl, G., Serafin, S., Cook, P. R., and Smith, J. O. "Theory of Banded Waveguides". In: *Computer Music Journal* 28.1 (2004), pp. 37–50. DOI: 10.1162/014892604322970634.

[30] Aramaki, M. and Kronland-Martinet, R. "Analysis-synthesis of impact sounds by real-time dynamic filtering". In: *IEEE Transactions on Audio, Speech, and Language Processing* 14.2 (Mar. 2006), pp. 695–705. ISSN: 1558-7916.

[31] Karplus, K. and Strong, A. "Digital Synthesis of Plucked-String and Drum Timbres". In: *Computer Music Journal* 7 (June 1983), pp. 43–55. DOI: 10.2307/3680062.

[32] A. Jaffe, D. and Smith, J. "Extensions of the Karplus-Strong Plucked-String Algorithm". In: *Computer Music Journal* 7 (June 1983), pp. 56–69. DOI: 10.2307/3680063.

[33] Smith, J. O. "Virtual electric guitars and effects using FAUST and Octave". In: *Proceedings of the 6th International Linux Audio Conference (LAC-08)* (2008).

[34] Rocchesso, D. and Fontana, F. "The Sounding Object". In: *IEEE Multimedia - IEEEMM* (Jan. 2003).

[35] Avanzini, F., Rath, M., and Rocchesso, D. "Physically-based audio rendering of contact". In: vol. 2. Feb. 2002, 445 –448 vol.2. ISBN: 0-7803-7304-9. DOI: 10.1109/ICME.2002.1035636.

[36] Hunt, K. and Crossley, E. "Coefficient of restitution interpreted as damping in vibroimpact". In: *Journal of Applied Mechanics* (1975). DOI: 10.1115/1.3423596. URL: https://hal.archives-ouvertes.fr/hal-01333795.

[37] Maunsbach, M. and Serafin, S. "Non-Linear Contact Sound Synthesis for Real-Time Audio-Visual Applications using Modal Textures". In: 2019.

[38] Serafin, S., Avanzini, F., and Rocchesso, D. "Bowed string simulation using an elasto-plastic friction model". In: 2003.

[39] Borin, G., De Poli, G., and Rocchesso, D. “Elimination of delay-free loops in discrete-time models of nonlinear acoustic systems”. In: *IEEE Transactions on Speech and Audio Processing* 8.5 (Sept. 2000), pp. 597–605. ISSN: 1063-6676. DOI: 10.1109/89.861380.

[40] Zölzer, U. *DAFX: Digital Audio Effects, Second Edition*. 2011.

[41] Marshall, D. and Sidorov, K. *Digital Audio Effects*. http://users.cs.cf.ac.uk/Dave.Marshall/CM0268/PDF/10_CM0268_Audio_FX.pdf. Accessed: 2019-05-26.

[42] J. Tomlinson, B., Noah, B., and N. Walker, B. “BUZZ: An Auditory Interface User Experience Scale”. In: Apr. 2018, pp. 1–6. DOI: 10.1145/3170427.3188659.

Appendix A

Appendix

A.1 Videos

Open "links.pdf" for external links or "Videos.zip" for files. Attached are links to 5 videos showing the features in use. Most audio and video glitches are from the screen recording, especially the Guitar Saber sounds. Only the first video is edited. The videos are:

1. *Adaptive Remix (First Recombination)* - showing the adaptive remix mode and the first recombination. This was used for testing. As well as an early prototype.
2. *Adaptive Endless* - showing that the endless mode is synchronized properly. Transitions begin at 1:45, 3:35 (with snoozing) and 5:30.
3. *Guitar Saber* - The Guitar Saber and its functionalities are showed. This instrument was used for testing.
4. *Guitar Saber Game* - The guitar saber being used in a game. This was used for testing.
5. *Friction Wall* - The carving friction being used on a wall.

A.2 Beat-Slashing Game

The Unity project is attached as "Saber Appendix.zip". All C# scripts are in the folder Saber Appendix/Assets/Scripts and C++ native code can be found in Saber Appendix/NativeCode.

A.3 MatLab: Extended Karplus-Strong

```
% Noise burst
noise = 2*rand(4000,1)-1;
% Sampling Rate, fundamental frequency and length
Fs = 48000; len = 12*Fs; f0 = 440;

% Delay
N = floor(Fs/f0-1);

% Initialize excitation and output arrays
X = [noise(1:N); zeros(len-N,1)]; Y = zeros(len,1);

% FILTER PARAMETERS %
% Delay
outDel = zeros(len,1);
% Pick Direction lowpass
p = 0.9; outPickDir = zeros(len,1);

% Pick-Position comb filter
beta = 0.13; combDel = floor(beta*N); outPickPos=zeros(len,1);

% Damping second order lowpass
B = 1.0; S = B/2; sec = 14; rho = 0.001^(1/(f0*sec));
h0 = (1+B)/2; h1 = (1-B)/4; outDamp = zeros(len,1);

% String-Tuning allpass
outStringTuning = zeros(len,1); nt = Fs/f0-N-1;;

% Energetic dynamic lowpass
w = (pi*f0)/Fs; enerC0 = w/(1+w); enerC1 = ((1-w)/(1+w)); L = 1.00; L0 = L
    ^(1/3);
outEner = zeros(len,1);

% Initial N samples
for n=1:N
    outPickDir(n) =  (1-p) * X(n) + p*del(outPickDir,n-1);
    outPickPos(n) = outPickDir(n) - del(outPickDir,n-combDel);
end

% Samples after the first N
for n=(N+1):len
    outDel(n) = del(outPickPos,n-N) + del(outStringTuning, n-N);

    outDamp(n) = rho*(h1*(del(outDel, n)+del(outDel, n-2)) + h0*del(outDel
        , n-1));
    outStringTuning(n) = (1-nt)*outDamp(n) + nt*del(outDamp, n-1);

    outEner(n) = enerC0*(del(outDel,n)+ del(outDel,n-1))+enerC1*del(
        outEner, n-1);
    outEner(n) = L*L0*del(outDel,n)+(1-L)*outEner(n);
    Y(n-N) = outEner(n);
end
```

A.4 Native Audive SDK: Extended Karplus-Strong

```
#include "AudioPluginUtil.h"
#include <algorithm>;

namespace Plugin_EKS
{
    const int MAXDEL = 1024;
    const float SAMPLERATE = 48000.0f;

    enum Param
    {
        P_FREQ,
        P_TRIGGER,
        P_PICKDIR_P,
        P_PICKPOS_BETA,
        P_DAMP_B,
        P_DAMP_SEC,
        P_DYNLEV_L,
        P_NUM
    };

    struct EffectData
    {
        struct Data
        {
            float p[P_NUM];
            int N;
            int n;
            float noiseBurst[MAXDEL];
            int noiseIdx;
            float outPickDir[MAXDEL];
            float outPickPos[MAXDEL];
            float outDel[MAXDEL];
            int combDel;
            float outDamp[MAXDEL];
            float outStringTuning[MAXDEL];
            float nStringTuning;
            float outDynLev[MAXDEL];

            float out[MAXDEL];
            Random random;
        };
        union
        {
            Data data;
            unsigned char pad[(sizeof(Data) + 15) & ~15];
        };
    };

    static void ReTrigger(EffectData::Data* data)
    {
        data->N = (int)floor(SAMPLERATE / data->p[P_FREQ] - 1.0f);
        data->nStringTuning = SAMPLERATE / data->p[P_FREQ] - data->N - 1.0f;
        int combDel = floor(data->p[P_PICKPOS_BETA] * data->N);

        ResetBuffers(data);

        for (int n = 0; n < data->N; n++)
        {
            data->noiseBurst[n] = data->random.GetFloat(-1.0f, 1.0f);
            data->outPickDir[n] = (1 - data->p[P_PICKDIR_P]) * data->
                noiseBurst[n] + data->p[P_PICKDIR_P] * data->outPickDir[n];
            data->outPickPos[n] = data->outPickDir[n] - data->outPickDir[(n -
                combDel) & 0x3FF];
        }

        data->n = data->N;
        data->noiseIdx = 0;
    }

    int InternalRegisterEffectDefinition(UnityAudioEffectDefinition&
        definition)
    {
        int numparams = P_NUM;
        definition.paramdefs = new UnityAudioParameterDefinition[numparams];
        RegisterParameter(definition, "Trigger", "", -1.0f, 1.0f, 0.0f, 1.0f
            , 1.0f, P_TRIGGER, "Trigger a signal");
        RegisterParameter(definition, "Frequency", "Hz", 60.0f, 3000.0f,
            440.0f, 1.0f, 3.0f, P_FREQ, "Frequency of sine oscillator that
            is multiplied with the input signal");
        RegisterParameter(definition, "Pick Direction", "", 0.0f, 1.0f, 0.9f
            , 1.0f, 1.0f, P_PICKDIR_P, "Pick direction. 0 is up, 0.9 is down
            .");
        RegisterParameter(definition, "Pick Position", "", 0.0f, 1.0f, 0.13f
            , 1.0f, 1.0f, P_PICKPOS_BETA, "Ratio of pick position on
            fretboard. 0 is bridge, 1 is nut.");
        RegisterParameter(definition, "Damping", "", 0.00f, 1.0f, 1.0f, 1.0f
            , 1.0f, P_DAMP_B, "Brightness 0-1");
        RegisterParameter(definition, "Damping Time", "s", 0.5f, 14.0f, 14.0
            f, 1.0f, 1.0f, P_DAMP_SEC, "Time in seconds to get to -60db.");
        RegisterParameter(definition, "Dynamic Level", "", 0.0f, 1.0f, 0.0f,
            1.0f, 1.0f, P_DYNLEV_L, "Dynamic level for more energetic pick.
            ");
        return numparams;
    }
```

```

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK CreateCallback(
    UnityAudioEffectState* state)
{
   EffectData* effectdata = new EffectData;
   memset(effectdata, 0, sizeof(EffectData));
   state->effectdata = effectdata;
   InitParametersFromDefinitions(InternalRegisterEffectDefinition,
       effectdata->data.p);
   return UNITY_AUDIODSP_OK;
}

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ReleaseCallback(
    UnityAudioEffectState* state)
{
   EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
   delete data;
   return UNITY_AUDIODSP_OK;
}

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK SetFloatParameterCallback
    (UnityAudioEffectState* state, int index, float value)
{
   EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
   if (index >= P_NUM)
      return UNITY_AUDIODSP_ERR_UNSUPPORTED;
   data->p[index] = value;

   if (index == P_TRIGGER)
   {
      if (data->p[P_TRIGGER] != 0.0f)
         ReTrigger(data);
   }
   else if (index == P_FREQ)
      data->N = (int)floor(SAMPLERATE / data->p[P_FREQ]);
   return UNITY_AUDIODSP_OK;
}

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK GetFloatParameterCallback
    (UnityAudioEffectState* state, int index, float* value, char *
    valuestr)
{
   EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
   if (index >= P_NUM)
      return UNITY_AUDIODSP_ERR_UNSUPPORTED;
   if (value != NULL)
      *value = data->p[index];
   if (valuestr != NULL)
      valuestr[0] = 0;
   return UNITY_AUDIODSP_OK;
}
```

```

int UNITY_AUDIODSP_CALLBACK GetFloatBufferCallback(UnityAudioEffectState* state, const char* name, float* buffer, int numsamples)
{
   return UNITY_AUDIODSP_OK;
}

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ProcessCallback(UnityAudioEffectState* state, float* inbuffer, float* outbuffer, unsigned int length, int inchannels, int outchannels)
{
   EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
   float wetTarget = ((state->flags & UnityAudioEffectStateFlags_IsPlaying) && !(state->flags & (UnityAudioEffectStateFlags_IsMuted | UnityAudioEffectStateFlags_IsPaused))) ? 1.0f : 0.0f;

   float dampRho = pow(0.001f, (1.0f / (data->p[P_FREQ] * data->p[P_DAMP_SEC])));
   float damph0 = (1.0f + data->p[P_DAMP_B]) * 0.5f;
   float damph1 = (1.0f - data->p[P_DAMP_B]) * 0.25f;

   float w = (kPI*data->p[P_FREQ]) / SAMPLERATE;
   float enerC0 = w / (1.0f + w);
   float enerC1 = ((1.0f - w) / (1.0f + w));
   float L0 = pow(data->p[P_DYNLEV_L], (1.0f / 3.0f));

   for (unsigned int n = 0; n < length; n++)
   {
      // Only apply noise in the beginning
      float noise = 0.0f;
      if (data->noiseIdx < data->N)
      {
         noise = data->outPickPos[data->noiseIdx];
         data->noiseIdx++;
      }

      data->outDel[data->n] = noise + data->outStringTuning[(data->n - data->N) & 0x3FF];

      data->outDynLev[data->n] = enerC0 * (data->outDel[data->n] + data->outDel[(data->n-1) & 0x3FF]) + enerC1 * data->outDynLev[(data->n - 1) & 0x3FF];
      data->outDynLev[data->n] = data->p[P_DYNLEV_L] * L0*data->outDel[data->n] + (1.0f - data->p[P_DYNLEV_L])*data->outDynLev[data->n];

      data->outDamp[data->n] = dampRho * (damph1*(data->outDel[data->n] + data->outDel[(data->n - 2) & 0x3FF]) + damph0 * data->outDel[(data->n - 1) & 0x3FF]);
```

```
            data->outStringTuning[data->n] = (1.0f - data->nStringTuning)*
                data->outDamp[data->n] + data->nStringTuning*data->outDamp[(
                data->n - 1) & 0x3FF];

            for (int i = 0; i < outchannels; i++)
            {
                float x = data->outDynLev[data->n];

                outbuffer[n * outchannels + i] = x * wetTarget;
            }

            data->n = (data->n+1) & 0x3FF;
        }

        return UNITY_AUDIODSP_OK;
    }
}
```

A.5 MatLab: Friction Synthesis

```
% Control parameters
Vb = 0.1;
freq = 100*(1:3);

% Initial values
modes = length(freq);
q_r = 500;
oneMass = 1e-3;
m = ones(1,modes)*oneMass;

Fs = 48000;
timeStep = 1/Fs;
h = 2*Fs;

omega_r = zeros(1,modes);
g_r = zeros(1,modes);

b_r = zeros(2,modes);
A_r = zeros(2,2,modes);
xv_r = zeros(2,modes);

for i=1:modes
    omega_r(i) = 2*pi*freq(i);
    g_r(i) = omega_r(i)/q_r;
    detTemp = (Fs^2+g_r(i)*Fs/2+omega_r(i)^2/4);

    A_r(1,1,i) = 1/detTemp * (detTemp-omega_r(i)^2/2);
    A_r(2,1,i) = 1/detTemp * (-Fs*omega_r(i)^2);
    A_r(1,2,i) = 1/detTemp * Fs;
    A_r(2,2,i) = 1/detTemp * (2*Fs^2-detTemp);

    b_r(1,i) = (1/m(i))*(1/(4*detTemp));
    b_r(2,i) = (1/m(i))*(1/(4*detTemp))*2*Fs;
end

% Bow
q_b = 1; m_b = 50e-3;

b_b = zeros(2,1); A_b = zeros(2,2); xv_b = zeros(2,1);

omega_b = 0;
g_b = 0;
detTemp = (Fs^2+g_b*Fs/2+omega_b^2/4);
A_b(1,1) = 1/detTemp * (detTemp-omega_b^2/2);
A_b(2,1) = 1/detTemp * (-Fs*omega_b^2);
A_b(1,2) = 1/detTemp * Fs;
A_b(2,2) = 1/detTemp * (2*Fs^2-detTemp);
b_b(1) = (1/m_b)*(1/(4*detTemp));
b_b(2) = (1/m_b)*(1/(4*detTemp))*2*Fs;


% Force
sig0 = 10000; sig1 = .1*sqrt(sig0); sig2 = 0.4; sig3 = 0.0; v_s = 0.1;
mu_d = 0.2; mu_s = 0.4;  Zss = 0; c = 0.7;
f_N = 1; f_s = mu_s*f_N; f_c = mu_d*f_N; Zba = c*f_c/sig0;

fe_b=f_c +(f_s-f_c)*exp(-(Vb/v_s)^2) +sig2*Vb; % with w=0 and sgn(Vb)=1

% K components
K2 = 1/(2*Fs);

bv_r = sum(b_r,2);
bv_r = bv_r(2);
bv_b = b_b(2);

bv = bv_r+b_b(2);
K1 = -bv/(1+sig2*bv)*(sig0/(2*Fs)+sig1);

% Initial values
smplen = 4*Fs;
sig = zeros(smplen, 1);
yPrev = 0;
zPrev = 0;
z_Ti = 0;
f_tot_b=0;                      % total force on bow
f_tot_r=0;

errMax = 10^(-13);
hs = zeros(smplen,1);
h0 = 0;
```

```
maxIt = 0;
f_fr = zeros(smplen, 1);

for t=1:smplen
    % Computable part of x, v and z
    for j=1:modes
        xv_r(:,j) = A_r(:,:,j)*xv_r(:,j) + b_r(:,j)*f_tot_r;
    end

    xv_b = A_b*xv_b + b_b*f_tot_b;
    z_Ti = zPrev + 1/(2*Fs)*yPrev;

    v_rSum = sum(xv_r,2);
    v_rSum = v_rSum(2);
    v_bSum = xv_b(2);

    w = (rand(1)*2-1)*abs(v_bSum)*f_N;

    v_Ti = 1/(1+sig2*bv)* ...
        ((v_bSum+bv_b*(fe_b-sig0*z_Ti-sig3*w))+...
        (-v_rSum - bv_r*(sig0*z_Ti+sig3*w)));

    % Newton Rhapson
    count = 1;
    err = 99;
    while err > errMax && count < 1000
        vNew = v_Ti + K1*h0;
        zNew = z_Ti + K2*h0;

        % Find Zss
        Zss = (sign(vNew)/sig0)*(f_c+(f_s-f_c)*exp(-(vNew/v_s)^2));
        if vNew==0
            zss=f_s/sig0;
        end

        % Find alpha tilde
        if sign(zNew) ~= sign(vNew)
            aNew = 0;
        elseif abs(zNew) < Zba
            aNew = 0;
        elseif abs(zNew) > Zss
            aNew = 1;
        else
            aNew = 0.5*(1+sin(pi*((zNew-0.5*(Zss+Zba))/(Zss-Zba))));
        end

        % Compute g
        gNom = vNew*(1-aNew*zNew/Zss)-h0;

        % Compute derivatives needed for derivative of g
        % Zss/v, a/z, a/v to compute dotz/v and dotz/z
        ZssvDeri = -sign(vNew) *...
            (2*vNew)/(sig0*v_s^2) * (f_s-f_c)*exp(-(vNew/v_s)^2);

        if Zba < abs(zNew) && abs(zNew) < Zss && sign(vNew) == sign(zNew)
            temp = 0.5*pi * cos(pi*(zNew-0.5*(Zss+Zba))/(Zss-Zba));
            azDeri = temp * (1/(Zss-Zba));
            avDeri = temp * ((ZssvDeri*(Zba-zNew))/(Zss-Zba)^2);
        else
            azDeri = 0;
            avDeri = 0;
        end

        derZ = -(vNew/Zss)*(zNew*azDeri+aNew);
        derV = 1-zNew*(((aNew + vNew*avDeri)*Zss-aNew*vNew*ZssvDeri)/(Zss
            ^2));
        gDeri = derV*K1 + derZ*K2 - 1;

        h1 = h0 - gNom/gDeri;

        count = count+1;
        maxIt = max(count, maxIt);

        err = abs(h1 - h0);

        h0 = h1;
    end
    hs(t) = count;


    dotz = h0;
    v = v_Ti+K1*dotz;
    z = z_Ti+K2*dotz;
    zPrev = z;
    yPrev = h0;

    % Newton Rhapson over and out
    f_fr(t) = sig0*z + sig1*dotz + sig2*v+sig3*w;

    f_tot_b=fe_b -f_fr(t);
    f_tot_r=f_fr(t);


    for j=1:modes
        xv_r(:,j) = xv_r(:,j) + b_r(:,j)*f_tot_r;
    end

    xv_b = xv_b + b_b*f_tot_b;

    sig(t) = sum(xv_r(1,:));
end
```

A.6 Native Audive SDK: Friction Synthesis

```
#include "AudioPluginUtil.h"
#include <algorithm>;

namespace Plugin_EKS
{
   const int MAXDEL = 1024;
   const float SAMPLERATE = 48000.0f;

   enum Param
   {
      P_FREQ,
      P_TRIGGER,
      P_PICKDIR_P,
      P_PICKPOS_BETA,
      P_DAMP_B,
      P_DAMP_SEC,
      P_DYNLEV_L,
      P_NUM
   };

   struct EffectData
   {
      struct Data
      {
         float p[P_NUM];
         int N;
         int n;
         float noiseBurst[MAXDEL];
         int noiseIdx;
         float outPickDir[MAXDEL];
         float outPickPos[MAXDEL];
         float outDel[MAXDEL];
         int combDel;
         float outDamp[MAXDEL];
         float outStringTuning[MAXDEL];
         float nStringTuning;
         float outDynLev[MAXDEL];

         float out[MAXDEL];
         Random random;
      };
      union
      {
         Data data;
         unsigned char pad[(sizeof(Data) + 15) & ~15];
      };
   };

```

```
static void ReTrigger(EffectData::Data* data)
{
    data->N = (int)floor(SAMPLERATE / data->p[P_FREQ] - 1.0f);
    data->nStringTuning = SAMPLERATE / data->p[P_FREQ] - data->N - 1.0f;
    int combDel = floor(data->p[P_PICKPOS_BETA] * data->N);

    ResetBuffers(data);

    for (int n = 0; n < data->N; n++)
    {
        data->noiseBurst[n] = data->random.GetFloat(-1.0f, 1.0f);
        data->outPickDir[n] = (1 - data->p[P_PICKDIR_P]) * data->
            noiseBurst[n] + data->p[P_PICKDIR_P] * data->outPickDir[n];
        data->outPickPos[n] = data->outPickDir[n] - data->outPickDir[(n -
            combDel) & 0x3FF];
    }

    data->n = data->N;
    data->noiseIdx = 0;
}

int InternalRegisterEffectDefinition(UnityAudioEffectDefinition&
    definition)
{
    int numparams = P_NUM;
    definition.paramdefs = new UnityAudioParameterDefinition[numparams];
    RegisterParameter(definition, "Trigger", "", -1.0f, 1.0f, 0.0f, 1.0f
        , 1.0f, P_TRIGGER, "Trigger a signal");
    RegisterParameter(definition, "Frequency", "Hz", 60.0f, 3000.0f,
        440.0f, 1.0f, 3.0f, P_FREQ, "Frequency of sine oscillator that
        is multiplied with the input signal");
    RegisterParameter(definition, "Pick Direction", "", 0.0f, 1.0f, 0.9f
        , 1.0f, 1.0f, P_PICKDIR_P, "Pick direction. 0 is up, 0.9 is down
        .");
    RegisterParameter(definition, "Pick Position", "", 0.0f, 1.0f, 0.13f
        , 1.0f, 1.0f, P_PICKPOS_BETA, "Ratio of pick position on
        fretboard. 0 is bridge, 1 is nut.");
    RegisterParameter(definition, "Damping", "", 0.00f, 1.0f, 1.0f, 1.0f
        , 1.0f, P_DAMP_B, "Brightness 0-1");
    RegisterParameter(definition, "Damping Time", "s", 0.5f, 14.0f, 14.0
        f, 1.0f, 1.0f, P_DAMP_SEC, "Time in seconds to get to -60db.");
    RegisterParameter(definition, "Dynamic Level", "", 0.0f, 1.0f, 0.0f,
         1.0f, 1.0f, P_DYNLEV_L, "Dynamic level for more energetic pick.
        ");
    return numparams;
}

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK CreateCallback(
    UnityAudioEffectState* state)
{
    EffectData* effectdata = new EffectData;
    memset(effectdata, 0, sizeof(EffectData));
    state->effectdata = effectdata;
    InitParametersFromDefinitions(InternalRegisterEffectDefinition,
        effectdata->data.p);
    return UNITY_AUDIODSP_OK;
}

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ReleaseCallback(
    UnityAudioEffectState* state)
{
    EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
    delete data;
    return UNITY_AUDIODSP_OK;
}

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK SetFloatParameterCallback
    (UnityAudioEffectState* state, int index, float value)
{
    EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
    if (index >= P_NUM)
        return UNITY_AUDIODSP_ERR_UNSUPPORTED;
    data->p[index] = value;

    if (index == P_TRIGGER)
    {
        if (data->p[P_TRIGGER] != 0.0f)
            ReTrigger(data);
    }
    else if (index == P_FREQ)
        data->N = (int)floor(SAMPLERATE / data->p[P_FREQ]);
    return UNITY_AUDIODSP_OK;
}

UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK GetFloatParameterCallback
    (UnityAudioEffectState* state, int index, float* value, char *
    valuestr)
{
    EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
    if (index >= P_NUM)
        return UNITY_AUDIODSP_ERR_UNSUPPORTED;
    if (value != NULL)
        *value = data->p[index];
    if (valuestr != NULL)
        valuestr[0] = 0;
    return UNITY_AUDIODSP_OK;
}

int UNITY_AUDIODSP_CALLBACK GetFloatBufferCallback(
    UnityAudioEffectState* state, const char* name, float* buffer, int
    numsamples)
{
        return UNITY_AUDIODSP_OK;
    }

    UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ProcessCallback(
        UnityAudioEffectState* state, float* inbuffer, float* outbuffer,
        unsigned int length, int inchannels, int outchannels)
    {
        EffectData::Data* data = &state->GetEffectData<EffectData>()->data;
        float wetTarget = ((state->flags &
            UnityAudioEffectStateFlags_IsPlaying) && !(state->flags & (
            UnityAudioEffectStateFlags_IsMuted |
            UnityAudioEffectStateFlags_IsPaused))) ? 1.0f : 0.0f;

        float dampRho = pow(0.001f, (1.0f / (data->p[P_FREQ] * data->p[
            P_DAMP_SEC])));
        float damph0 = (1.0f + data->p[P_DAMP_B]) * 0.5f;
        float damph1 = (1.0f - data->p[P_DAMP_B]) * 0.25f;

        float w = (kPI*data->p[P_FREQ]) / SAMPLERATE;
        float enerC0 = w / (1.0f + w);
        float enerC1 = ((1.0f - w) / (1.0f + w));
        float L0 = pow(data->p[P_DYNLEV_L], (1.0f / 3.0f));

        for (unsigned int n = 0; n < length; n++)
        {
            // Only apply noise in the beginning
            float noise = 0.0f;
            if (data->noiseIdx < data->N)
            {
                noise = data->outPickPos[data->noiseIdx];
                data->noiseIdx++;
            }

            data->outDel[data->n] = noise + data->outStringTuning[(data->n -
                data->N) & 0x3FF];

            data->outDynLev[data->n] = enerC0 * (data->outDel[data->n] + data
                ->outDel[(data->n-1) & 0x3FF]) + enerC1 * data->outDynLev[(
                data->n - 1) & 0x3FF];
            data->outDynLev[data->n] = data->p[P_DYNLEV_L] * L0*data->outDel[
                data->n] + (1.0f - data->p[P_DYNLEV_L])*data->outDynLev[data
                ->n];

            data->outDamp[data->n] = dampRho * (damph1*(data->outDel[data->n]
                 + data->outDel[(data->n - 2) & 0x3FF]) + damph0 * data->
                outDel[(data->n - 1) & 0x3FF]);
            data->outStringTuning[data->n] = (1.0f - data->nStringTuning)*
                data->outDamp[data->n] + data->nStringTuning*data->outDamp[(
                data->n - 1) & 0x3FF];

            for (int i = 0; i < outchannels; i++)
```

```
                {
                    float x = data->outDynLev[data->n];

                    outbuffer[n * outchannels + i] = x * wetTarget;
                }

                data->n = (data->n+1) & 0x3FF;
            }

            return UNITY_AUDIODSP_OK;
        }
}
```