PETER VASIL

GRAPHICAL DESIGN OF PHYSICAL MODELS FOR REAL-TIME SOUND SYNTHESIS

Masterarbeit Audiokommunikation und -technologie

GRAPHICAL DESIGN OF PHYSICAL MODELS FOR REAL-TIME SOUND SYNTHESIS

Implementation of a Graphical User Interface for the Synth-A-Modeler compiler

vorgelegt von PETER VASIL

Matr.-Nr.: 328493

Technische Universität Berlin

Fakultät 1: Fachgebiet Audiokommunikation

Abgabe: 25. Juli 2013

Peter Vasil: *Graphical Design of Physical Models for Real-Time Sound Synthesis,* Implementation of a Graphical User Interface for the Synth-A-Modeler compiler, © July 2013

SUPERVISORS:
Prof. Dr. Stefan Weinzierl
Dr. Edgar Berdahl

ABSTRACT

The goal of this Master of Science thesis in Audio Communication and Technology at Technical University Berlin, is to develop a Graphical User Interface (GUI) for the *Synth-A-Modeler compiler*, a text-based tool for converting physical model specification files into DSP external modules. The GUI should enable composers, artists and students to use physical modeling intuitively, without having to employ complex mathematical equations normally necessary for physical modeling. The GUI that will be developed in the course of this thesis allows the creation of physical models with graphical objects for physical masses, links, resonators, waveguides, terminations, and audioout objects. In addition, this tool will be used to create a model for an Arabic oud to demonstrate its functionality.

ZUSAMMENFASSUNG

Das Ziel dieser Master Arbeit am Fachgebiet Audiokommunikation der TU Berlin, ist die Entwicklung einer graphischen Umgebung für die textbasierte Software, *Synth-A-Modeler compiler*, welche es erlaubt, ein speziell dafür entwickeltes Datei Format für physikalische Modelle, in externe DSP Module umzuwandeln. Die Software macht es Komponisten, Künstlern und Studenten möglich, physikalische Modellierung intuitiv zu anzuwenden, ohne die komplexen mathematischen Formeln, welche normalerweise für physikalische Modellierung nötig sind, anwenden zu müssen. Die graphische Umgebung, die im Zuge dieser Arbeit entwickelt wird, erlaubt es dem Benutzer physikalische Modelle mithilfe von graphischen Bausteinen, welche physikalische Elemente, wie z. B. Masse, Federn, Resonatoren oder Wellenleiter representieren, selbst zu erstellen. Im Rahmen der Arbeit wird die graphische Umgebung ausserdem dazu verwendet, eine arabische Laute zu modellieren, um die Funktionalität der Software zu demonstrieren.

ACKNOWLEDGMENTS

I would like to thank my partner Kathleen Reinhardt and my parents Ilona and Stefan Vasil. I would also like to thank Tobias Preuss for his valuable input and discussions regarding the implementation of the application for this thesis.

Last but not least I would like to thank my supervisors Stefan Weinzierl and especially Edgar Behrdahl, for his feedback and support during my work on the thesis.

CONTENTS

i INTRODUCTION 1

1 INTRODUCTION 2

1.1 Goals/Objectives . . . 2

1.2 Motivation . . . 3

1.3 Thesis outline . . . 3

ii FUNDAMENTALS 5

2 FUNDAMENTALS 6

2.1 Physical Modeling . . . 6

2.1.1 Digital Waveguide Synthesis . . . 6

2.1.2 Mass-Interaction Synthesis . . . 8

2.1.3 Modal Synthesis . . . 8

2.2 Prior work . . . 9

2.2.1 GENESIS . . . 9

2.2.2 Modalys . . . 10

3 SYNTH-A-MODELER COMPILER 12

3.1 Technical background . . . 12

3.1.1 Requirements . . . 12

3.1.2 Overview . . . 13

3.1.3 Faust . . . 13

3.1.4 Dataflow . . . 14

3.2 Modeling . . . 16

3.2.1 The model specification file . . . 16

3.2.2 Abstractions . . . 17

3.3 Compilation result . . . 19

3.4 Creating externals . . . 19

3.4.1 Pure Data . . . 19

3.4.2 SuperCollider . . . 21

3.4.3 Qt/GTK . . . 23

iii SYNTH-A-MODELER DESIGNER 24

4 DESIGN 25

4.1 Specification . . . 25

4.1.1 Visual specification . . . 25

4.1.2 Technical specification . . . 29

4.2 Software Architecture . . . 30

5 TECHNOLOGY SELECTED 32

5.1 JUCE C++ Library . . . 32

5.2 Regular Expressions . . . 35

5.3 Git . . . 36

5.4 Github . . . 37

6 IMPLEMENTATION 38

6.1 Application components 39
6.1.1 Core components 39
6.1.2 Internal MDL representation 43
6.1.3 MDL parsing and writing 46
6.1.4 Views 48
6.1.5 Controllers 54
6.1.6 Executing external commands 59
6.1.7 Command-line support 61
6.1.8 Unit testing 62
6.2 Automatic positioning of objects 63
6.2.1 Force-directed algorithm 64
6.2.2 Graph drawing components 66
6.2.3 Graph drawing examples 67
6.3 Extending Synth-A-Modeler Designer with a new object . 71
6.4 Problems and limitations 81
7 MODELING THE OUD 83
7.1 The Oud 83
7.1.1 History 83
7.1.2 Principle of operation 84
7.2 Modeling the oud in Synth-A-Modeler 85
7.2.1 The modeling 85
7.2.2 Commuted Synthesis 87
8 CONCLUSIONS 89
8.1 Conclusion 89
8.2 Future work 90

iv APPENDIX 91
A APPENDIX 92
A.1 Example *.mdl* file 92
A.2 Oud model 93
A.3 Emacs support 97
A.4 Vim support 98
A.5 Building *Synth-A-Modeler* on a *Beagle-Board* 100
A.6 Class hierarchy 101

BIBLIOGRAPHY 103

LIST OF FIGURES

Figure 1 Schematic relationship between physical modeling synthesis paradigms in *Synth-A-Modeler* . 6
Figure 2 Digital waveguide model of a simple string . . 7
Figure 3 *FAUST* dataflow diagram 14
Figure 4 Block diagram of a spring generated by *FAUST* 15
Figure 5 Synth-A-Modeler internal dataflow diagram . 15
Figure 6 Synth-A-Modeler dataflow diagram 16
Figure 7 "touch a resonator" PureData patch 21
Figure 8 Standalone applications 23
Figure 9 Mockup of the main *Synth-A-Modeler Designer* interface 26
Figure 10 Mockup of *Synth-A-Modeler* pull-down menus 27
Figure 11 Mass-like object icons 28
Figure 12 Link-like object icons 28
Figure 13 Waveguide and audioout object icons 28
Figure 14 Comparison, unsegmented and segmented connections 29
Figure 15 Collaboration of Model View Controller (MVC) components 31
Figure 16 *Synth-A-Modeler Designer* user interface 48
Figure 17 Object component inheritance 50
Figure 18 Window for editing object parameters 51
Figure 19 Setting font color of a `CommentComponent` . . . 53
Figure 20 *FAUST* code input and edit window 53
Figure 21 Misc tab in preferences window 54
Figure 22 Exporter tab in preferences window 55
Figure 23 Window for setting redraw options 67
Figure 24 An unordered `guiro` model 67
Figure 25 The `guiro` model after applying `CircleFlowAlgorithm` 68
Figure 26 The `guiro` model after applying `ForceBasedAlgorithm` 68
Figure 27 Cube model layout comparison 69
Figure 28 Big cube model layout comparison 69
Figure 29 Percussion model layout comparison 69
Figure 30 Random mass interaction model layout comparison 70
Figure 31 Another random mass interaction model layout comparison 70
Figure 32 Random model layout comparison 70
Figure 33 Icon of the pulsetouch object 71
Figure 34 Oud front and rear view 84
Figure 35 Simple schematic of a guitar 85

Figure 36 A string in *Synth-A-Modeler* 86
Figure 37 The oud model in *Synth-A-Modeler* 86
Figure 38 Class hierarchy diagram 102

LIST OF TABLES

Table 1 *MDL* object types and their parameters 18

LISTINGS

Listing 1 A simple spring represented in *FAUST* 14
Listing 2 A simple model specification 16
Listing 3 "touch a resonator" *FAUST* file 19
Listing 4 "touch a resonator" SuperCollider class 22
Listing 5 Example `CommandIDs` 41
Listing 6 Example usage of `CommandIDs` 41
Listing 7 *Synth-A-Modeler Designer* user settings file . . . 42
Listing 8 Pseudo code for an `UndoableAction` to add an object 43
Listing 9 Internal representation of an *MDL* file 44
Listing 10 Example assembling an *MDL* file structure . . 45
Listing 11 Class for parsing an *MDL* file 47
Listing 12 Class for writing an *MDL* file to disk 47
Listing 13 Function for fixing parameter value if not formatted properly 51
Listing 14 Function for adding a new object 56
Listing 15 Representation of a mass object when copied to the clipboard 58
Listing 16 `SAMCmd` class 59
Listing 17 Default exporter commands 61
Listing 18 Help text of the *Synth-A-Modeler Designer* command-line interface 62
Listing 19 Pseudo code of the force-based graph drawing algorithm 65
Listing 20 New code in `ObjectFactory.cpp` 71
Listing 21 New code in `ObjectsHolder.cpp` 73
Listing 22 New code in `LinkComponent.cpp` 74
Listing 23 New code in `ResourceLoader.cpp` 75
Listing 24 New code in `Application.cpp` 76

Listing 25 New code in CommandIDs.cpp 76
Listing 26 New code in ContentComponent.cpp 76
Listing 27 New code in ObjectPropertiesPanel.cpp . . . 77
Listing 28 New code in ObjectComponent.cpp 79
Listing 29 New code in MDLParser.cpp 79
Listing 30 New code in SAMRegex.cpp 80
Listing 31 New code in MiscUtilities.cpp 80
Listing 32 New default values 80
Listing 33 New values in SAMLookAndFeel.cpp 81
Listing 34 A complete mdl file 92
Listing 35 The model of an oud 93
Listing 36 Source of sam-mode for Emacs 97
Listing 37 Vim support source 98

ACRONYMS

API	Application Programming Interface
DSP	Digital Signal Processing
GUI	Graphical User Interface
RE	Regular Expression
SCM	source code management
MVC	Model View Controller

Part I

INTRODUCTION

1 INTRODUCTION

Physical modeling is a well-known and extensively developed technique to virtually create instruments as well as sounds. It is using mathematical models consisting of a set of equations and algorithms to simulate a physical source of sound. The parameters can be constants, which describe physical material and dimension of a real instrument, as well as time-dependent parameters, which describe the interaction of a player with the instrument. It has been an established research topic for decades, and many scientific texts have been published on the topic. However, creating physical models of instruments is a difficult process involving the creation of highly complex equations. Its challenge is to create a model, which sounds exactly like the instrument or sound it is imitating. Especially for non-western instruments this is a highly difficult process [1]. However, there are no widely accessible tools for composers or students to use this technique of physical modeling in order to create physical models without having to know or to use its mathematical context. Especially for real-time control it is a rarely employed technique.

1.1 GOALS/OBJECTIVES

This thesis will be part of creating the open-source *Synth-A-Modeler* modular design tool, which is being developed by Dr. Edgar Berdahl [2]. The *Synth-A-Modeler compiler* will not produce any sound directly. Instead, it will generate *DSP external modules* for the Digital Signal Processing (DSP) programming language, *FAUST*[1] which then can be used to produce modules for different environments, like SuperCollider, Max/MSP or PureData.

The goal of this thesis is to create a Graphical User Interface (GUI) for the *Synth-A-Modeler compiler*, the *Synth-A-Modeler Designer*. This will enable composers or artists to create physical models visually with the use of representations of real-world physical objects like masses, links, resonators, waveguides, terminations, and audioout objects. Furthermore, the *Synth-A-Modeler Designer* will be used to create a physical model for a non-western instrument, the Arabic oud, to demonstrate its functionality.

1 Faust is a programming language for real-time audio signal processing. http://faust.grame.fr/

1.2 MOTIVATION

Several tools exist that implement physical modeling in a modular way, so that the user does not have to program the equations. Instead, the user specifies physical systems, which consist of mechanical, acoustical or electrical elements connected together. This way it is much easier and more accessible for musicians to create new types of instruments and sounds without having to know and understand the mathematical details behind the complex physical systems.

Due to their high costs for purchasing, however, these tools are not very accessible to musicians or students. In order to work with the *Modalys* modal synthesis environment for example, the user needs to buy *Max/MSP* and the *IRCAM Forum Recherche* [2] at a high price. Similarly, *GENESIS* is a modeling environment, which provides a GUI for the creation of mass-interaction systems and can be purchased from the *Association pour la Création et la Recherche sur les Outils d'Expression* [2].

There are also non-commercial open-source software packages available for physical modeling. The BlockCompiler by Matti Karjaleinen is a very complex software. To create models, the user has to be able to program Lisp [3]. Another software package is the Synthesis Tool Kit (STK), which is written in C++.[2] However, to create new physical models the user has to write difference equations directly in C++ [2].

Thus, the interested user had to either invest financially in expensive commercial programs, or he had to be proficient in programming languages like Lisp or C++. These unnecessary barriers for using physical modeling were motivations that led to the creation of the *Synth-Modeler compiler*, in order to provide an accessible tool for sound synthesis with physical models.

This thesis contributes to the *Synth-A-Modeler* project through the development of a GUI for the *Synth-A-Modeler compiler*, which will enable the user to create physical models visually.

1.3 THESIS OUTLINE

This thesis is composed of two parts. Part one describes the fundamentals for the development of the *Synth-A-Modeler Designer* application. In part two, the development process will be described in detail.

Chapter 2 lays out the fundamentals for this thesis and for the development of *Synth-A-Modeler Designer*. The first section presents three physical modeling paradigms that are implemented in *Synth-A-Modeler*, and outlines their capabilities. The second part of the section presents an overview of existing software with functionalities similar to the ones in *Synth-A-Modeler*. It gives a brief technical overview and describes their user interfaces. Chapter 3 covers the *Synth-A-Modeler*

2 https://ccrma.stanford.edu/software/stk/

compiler. The section describes the compiler's technical details, such as the software structure, how it uses the *FAUST* compiler and its dataflow. The next section covers the modeling task, in particular the model specification file and physical objects. The following section describes the compilation output of the *Synth-A-Modeler compiler*. The chapter ends with the description on how to create externals with the compilation output and the *FAUST* compiler.

The second major part starts with Chapter 4, which describes the software design of the *Synth-A-Modeler Designer*, followed by an outline of the selected technology for the development in Chapter 5. Chapter 6 goes into detail on the implementation of the *Synth-A-Modeler Designer*. In particular, it describes the implementation of the application's main components and the automatic graph redrawing. The following section goes step by step through the process of extending the *Synth-A-Modeler Designer* with a new object. The chapter ends with a section on problems and limitations of the *Synth-A-Modeler Designer*. Chapter 7 gives a brief historical and technical overview on the Arabic oud and provides a description of the modeling process of such an instrument in *Synth-A-Modeler Designer*.

The last chapter on page 89 provides a conclusion and desirable extensions for the application in the future.

Part II

FUNDAMENTALS

2 FUNDAMENTALS

2.1 PHYSICAL MODELING

Physical Modeling is a sound synthesis technique for generating waveforms and sounds using mathematical models of virtual acoustical systems. The aim is to simulate a physical sound source virtually. Such models incorporate descriptions of physical laws to simulate the physical properties of a sound. The properties can include several parameters such as the material or size information of instrument components as well as interaction parameters describing how an instrument is played by a musician, i. e. plucking a string or hitting a membrane.

This section will describe three physical modeling synthesis paradigms that are implemented in *Synth-A-Modeler*. Figure 1 shows the relation of these three physical modeling synthesis paradigms in *Synth-A-Modeler*. The parts that intersect represent new modeling paradigms implemented by *Synth-A-Modeler*.

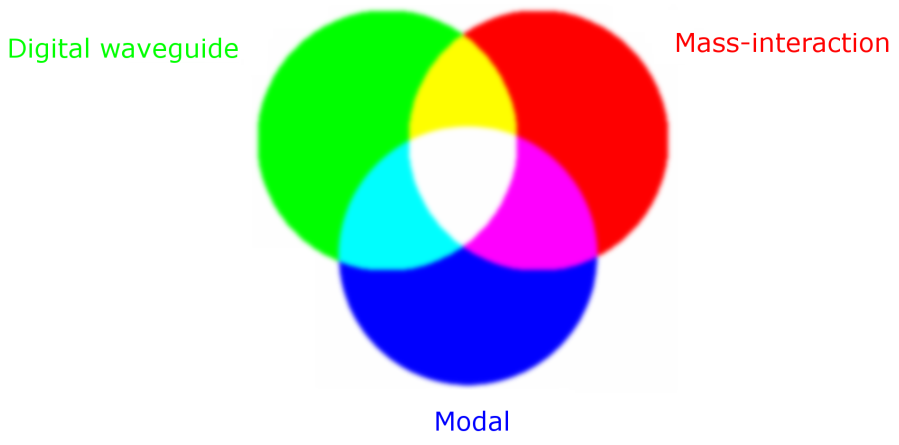


Figure 1: Schematic relationship between physical modeling synthesis paradigms in *Synth-A-Modeler*

2.1.1 *Digital Waveguide Synthesis*

Smith III [4] writes, "Digital waveguide synthesis models are computational physical models for certain classes of musical instruments (string, winds, brasses, etc.) which are made up of delay lines, digital filters, and often nonlinear elements." Furthermore he states, that digital waveguide models have the following characteristics in common:

"Sampled acoustic traveling waves, follow geometry and physical properties of a desired acoustic system, efficient for nearly lossless distributed wave media, and losses and dispersion are consolidated at sparse points along each waveguide. [4]"

A digital waveguide is represented by an bidirectional delay line as show in fig. 2, which consists of a sampled traveling wave element and has a characteristic wave impedance R. The wave impedance can

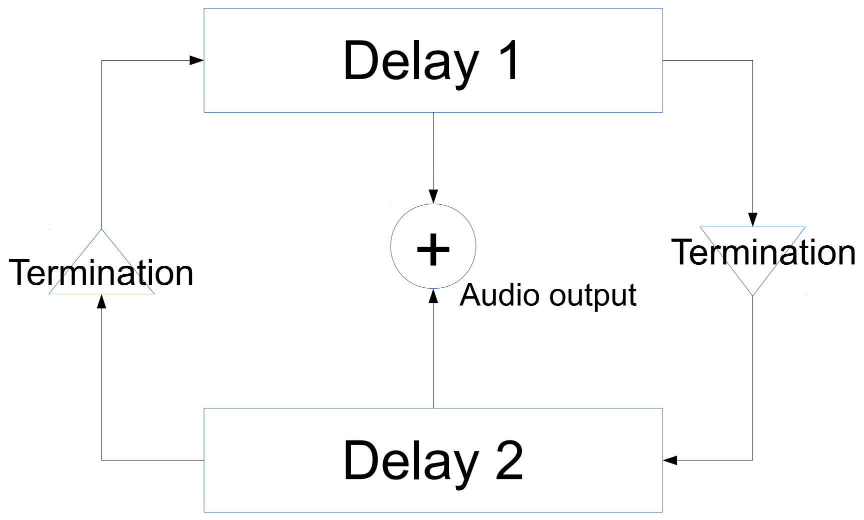


Figure 2: Digital waveguide model of a simple string

be determined via the following relationship:

$$R \hat{=} \sqrt{K\epsilon} = \frac{K}{c} = \epsilon c \tag{1}$$

and can be seen as the geometric mean of the two physical impediments to displacement: the string tension K and the linear mass density ϵ [1]. To be able to measure traveling waves and get physical values out of it, such as force, pressure and velocity, their components have to be summed (see fig. 2). The connection of physical media with different characteristic wave impedance cause scattering at the border/junction between them. For instance, when two one-dimensional waveguides with wave impedance R_1 and R_2 are connected together, an incoming wave from the first waveguide reflects back with coefficient

$$k_1 = \frac{R_2 - R_1}{R_2 + R_1} \tag{2}$$

A string for example, has boundaries, and the wave is not traveling infinitely, but has a constraint that is called termination. The simplest case is a rigid termination, which means that the wave cannot move at the termination. Rigid terminations reflect displacement, velocity and acceleration waves with sign inversion, which can also be calculated with (2), by setting the terminating impedance to infinity [4].

Synth-A-Modeler supports digital waveguide synthesis and defines the objects `waveguide`, `termination` and `junction` and will be described later in section 3.2. Only velocity waves are simulated directly in *Synth-A-Modeler* using the waveguide element.

2.1.2 *Mass-Interaction Synthesis*

According Castagne and Cadoz [5], mass-interaction synthesis is an aproach to physical modeling, which enables the composition of models with elementary modules, such as masses and springs. The modeling process is very modular and the different elementary parts are easy to understand because they are based on simple physical behaviors. Mass-interaction modeling is not specifically dedicated to for generating sounds, but it is well-suited for sound synthesis. Furthermore, the mass-interaction paradigm represents physical models in a more understandable way in contrast of using mathematical formulas to describe the physical models [5]. The latter makes the physical modeling process easier to understand, also for non-technical and non-mathemtical users. According to Castagne and Cadoz [5], mass-interaction modeling allows the representation of physical objects in a more general way, not only focusing on the the physical cause that generates a sound.

Synth-A-Modeler supports mass-interaction synthesis by providing elementary physical objects and the possibility to combine them to create physical models. More information on these objects will be provided in section 3.2.

2.1.3 *Modal Synthesis*

The third pyhsical modeling paradigm that is supported in *Synth-A-Modeler* is modal synthesis. Van den Doel and Pai [6] write, that when a solid object is hit, struck or involved in other interactions it get deformed by forces that are acting at the contact point, which causes the object to vibrate and to emit sound waves. Musical instruments that work with this principle are bells, marimba or vibraphone. Van den Doel and Pai [6] go on to state, that these musical instruments can be modeled with modal synthesis, which utilizes a bank of damped harmonic sine oscillators that are activated by some stimulus. The number of oscillators and their frequencis are determined by the object itself and its material properties. With modal synthesis in general, it also becomes possible to model any linear object that is vibrating, such as engines or other virtual music instruments.

Synth-A-Modeler provides a resonator object, which can have a variable number of resonace frequencies to perform modal synthesis and wil be discussed later in section 3.2.

2.2 PRIOR WORK

This section focuses on physical modeling software environments, allowing a user to model physical systems in order to create virtual instruments or physical objects. In particular, two systems are described, which are commercial and proprietary. The fact that the two systems are, as mentioned before, proprietary and commercial, creates a barrier to accessibility by the community and are therefore not well-suited for research, educational or artistic applications in the maker scene. Of course, the commercial background allows these systems to be developed using greater resources.

2.2.1 *GENESIS*

GENESIS is a software environment designed to be used in a musical context for musicians. The software is created by the company *ACROE-ICA*, and it lets the user model masses and interactions graphically. The first version of the software was released in 1995, in form of a beta[3] software and in 2000 a final version was published [8]. In 2009, the third version of the *GENESIS* software was released, with the version name G^3. The software uses the *CORDIS-ANIMA* language [9] which allows to build physical networks. A network is made of

> "elementary material modules `<MAT>` connected with physical interactions `<LIA>`" [8].

The modules stand for physical parameters, like mass, stiffness, viscosity and other types of interaction. G^3 has three main interfaces: the graphical modeling window, a textual modeling window and the simulation window. As the name suggests, the modeling window is the place where the user designs and creates a physical model. In the simulation window the user evaluates a model and obtains the synthesized sound. G^3 offers in comparison to G1 a more sophisticated 2D graphical representation of models, which let the user directly change parameters of elements and the ability to zoom into the model. With the simulation interface the user can visualize the model, see the waveform of the sound file, that has been generated from the physical model or change the simulation engine between an offline and real-time engine. Furthermore, Castagné et al. [8] write that not

3 Beta describes the state of an application in software development, when the software is almost ready to launch but is still being tested [7],

only graphical editing is from interest. That is the reason for implementing also the possibility to edit models textually. G^3 provides a scripting language called *Physics Network Scripting Language*, which is based on *Tcl*[4]. PNSL does not represent the state of a physical model, but it allows users to do the modeling activity by "programming" [8]. G^3 is a very mature modeling environment due to its predecessor G1 and more 10 years work in this field. The aim of G^3 is also to stabilize the *GENESIS* principles and to improve the capabilities and usability. In order to purchase *GENESIS* which is priced at approximately 500 Euro per workstation, it is necessary to have a personal relationship with employees of at *ACROE-ICA*.

2.2.2 *Modalys*

Modalys is a software system, which brings sound synthesis design into the scope of musical composition and uses modal synthesis for physical modeling. Modal synthesis "consists of solving the vibratory equations of the involved physical structures on a modal coordinate basis. A mode of vibration is an eigenvalue (frequency and loss) and an eigenvector (modeshape) of the characteristic equation of a physical system " [11]. The basic user interaction in *Modalys* is done in a Lisp-like language, in particular, an extension of the Scheme[5] language, which consists of primitive structures. These structures are objects, connections and controllers (in other words: physical structures) the interaction between these structures and time-varying parameters. There are two steps needed to create sound with *Modalys*. Creating an instrument, which is the assembly of structures and connections, and "executing" and instrument, that is sending controller data to these connection to make the instrument vibrate [11].

While *Modalys* has a textual user interface to the modeling part, there is *Modalys-ER*, a graphical environment for creating physical models and generating sound. But this environment is limited in terms of mapping controller data to the physical model's parameters. Thus, *Modalys-ER* is not well suited for performance, MIDI interaction or Standard Western Notation [12]. For this reason *Modalys-ER* has been partially ported to *OpenMusic*. *OpenMusic* is a visual programming language based on CommonLisp[6]. Programs are created by assembling and connecting icons, which represent functions and data structures [13]. The ported library for *OpenMusic* is called *MfOM* and has been developed at *IRCAM*.[7] The aim of *MfOM* is to provide an interface for Modalys to be used in a musical context. A sim-

4 Tcl, an abbreviation for "Tool Command Language", is a simple, multiparadigm open source programming language. See [10],

5 `http://groups.csail.mit.edu/mac/projects/scheme/`

6 CommonLisp is a dialect of the Lisp programming language `http://common-lisp.net/`

7 `http://www.ircam.fr/`

ple percussion instrument consist of a mass, which strikes a tuned plate. A second rectangular plate is included to limit the distance of the mass. This limitation makes sure that the distance of the striking mass is kept within a certain size, even if there is too high force applied to the mass. *MfOM* provides also a feature, called *parametrization*, which enables the user to create a simple instrument and duplicate it and assign automatically generated parameter values to. Also, it is possible to use MIDI files to play the generated instruments. The *parametrization* further makes it easier to create complex models, because modeling by hand can be very exhausting, especially when the user needs to change parameters of multiple objects. The original instrument acts like a template, and if the user changes some values, they will be assigned to the duplicates. The generated parameter values could also be accomplished algorithmically. In addition to the *parametrization* feature, there is also a function called *build-instrument*, which makes the creation of instruments even easier, by generating them algorithmically. The user only has set a few initial parameter settings, consisting of the resonator type, interaction type, list of pitches that it should be able to play, material and the number of modes in the resonator [12].

3 SYNTH-A-MODELER COMPILER

The Synth-A-Modeler compiler is a collection of short scripts to generate DSP modules from *MDL* files, which will be later discussed in section 3.2.1. It is the central unit in the Synth-A-Modeler Designer, the GUI part of Synth-A-Modeler and main topic of the thesis. The main goal of the Synth-A-Modeler compiler is to provide a toolchain for artists to create DSP modules for different sound synthesis environments with mechanical analog model specifications. It was also important to make possible to target different environments and not depend on a particular system. For this reason Synth-A-Modeler is using the FAUST programming language to generate efficient DSP code and use its ability to output its code to many external modules, i. e. Max/MSP, SuperCollider, PD, VST and many more. Synth-A-Modeler is the first software environment to facilitate the design of physical models that contain components from the paradigms of digital waveguide (Stanford University), mass-interaction (ACROE), and modal modeling techniques (IRCAM).

3.1 TECHNICAL BACKGROUND

3.1.1 *Requirements*

According to Berdahl and Smith III, the design of the Synth-A-Modeler compiler was led and influenced by several requirements.
"Synth-A-Modeler should be

- capable of efficient real-time synthesis for different host applications
- free and open-source
- modular
- easy to modify and extend
- a platform for education of the physics of mechanically vibrating systems
- accessible for artists, who have little or no experience in programming, DSP or physics
- enable development of MIDI synthesizers
- compatible with haptic force-feedback systems [2]."

3.1.2 *Overview*

To provide as much modularity as possible, the Synth-A-Modeler compiler is divided into several scripts with different functionality. The compiler basically parses an *MDL* file and generates *FAUST* code. The package consists of the following files:

- Synth-A-Modeler
- SAM-preprocessor
- SAM-regex
- physicalmodeling.lib
- SAM-fx.lib

All scripts except the file `physicalmodeling.lib` and `SAM-fx.lib` are written in the programming language *Perl*.[8] To make the parsing more efficient, `SAM-preprocessor` pre-processes the input *MDL* file, tidies its contents and removes clutter from it by eliminating unnecessary text. `SAM-regex` provides all Regular Expressions,[9] which are used for extracting the contents from an *MDL* file. `Synth-A-Modeler` is the main script and does all the parsing of an *MDL* file and the generation of the output *FAUST* file. `physicalmodeling.lib` contains all physical modeling primitives written in *FAUST* code, and `SAM-fx.lib` provides DSP code for past-processing the audio outputs from the models.

3.1.3 *Faust*

FAUST (***F****unctional* ***AU****dio* ***ST****ream*) is a functional programming language, which is designed for real-time signal processing and synthesis [15]. *FAUST* is a specification language, providing a notation for describing signal processors. Programs are compiled, not interpreted. The code is compiled into C++ source code, aiming to produce very efficient code. The main-line version works at sample level in the time domain, which makes it straightforward to implement low-level DSP functions. It is self-contained and does not depend on external code, which makes it easy to embed it in hardware. The *FAUST* language is block-diagram oriented, and it essentially implements functional programming using algebraic block diagrams. *FAUST* is designed to make it easy to deployed on a large variety of audio platforms and to describe signal processors with inputs and outputs. Because most audio processors are signal processors with inputs, outputs and controller parameters, they can be modeled easily with *FAUST* [15].

---

8 http://www.perl.org/

9 Regular Expressions provide a mechanism to select specific strings from a set of character strings [14]

Figure 3 shows a schematic diagram of *FAUST*'s dataflow from the *FAUST* code to an external. The usage of *FAUST* in Synth-A-

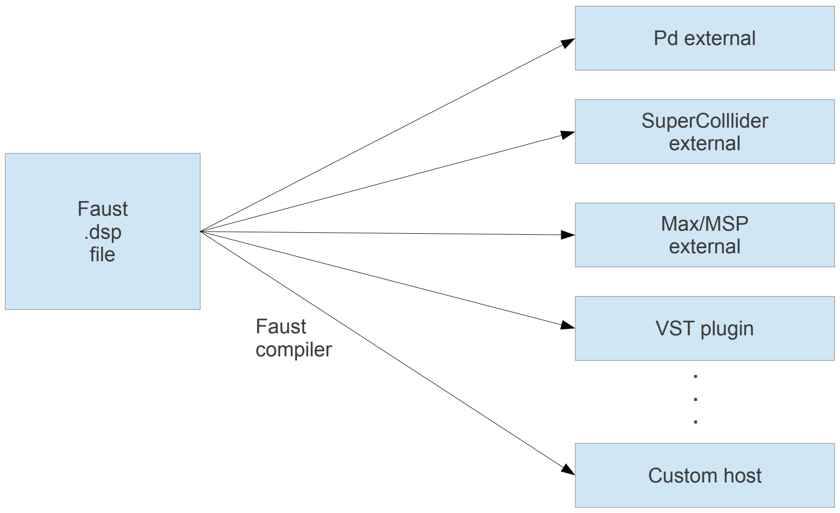


Figure 3: *FAUST* dataflow diagram

Modeler enables the generation of efficient real-time audio synthesis code. According to Berdahl and Smith III [2], there have been already physical models implemented in *FAUST* in the past before *Synth-A-Modeler*.

Listing 1: A simple spring represented in *FAUST*

```
spring(k) = (_,_) : - : *(k) : _ <: (*(-1.0),_);
process = spring(100.0);
```

Listing 1 shows a spring, modeled in *FAUST*. It takes to values as input, represented with _, subtracts them and multiplies the result with the spring constant k. In this example $k = 100$ (see listing 1). Because physical forces are always bidirectional and *FAUST*'s signal flow is from left to right, the output has to be delayed and fed back to represent the force's operation from both sides. This delay is represented in the *FAUST* code by a multiplication of the output with -1. Figure 4 shows the block-diagram generated by *FAUST*.

3.1.4 *Dataflow*

As described in section 3.1.2, the Synth-A-Modeler compiler consists of five files. Figure 5 shows the individual stages when compiling a model specification file (.mdl) into a *FAUST* file (.dsp). The first step

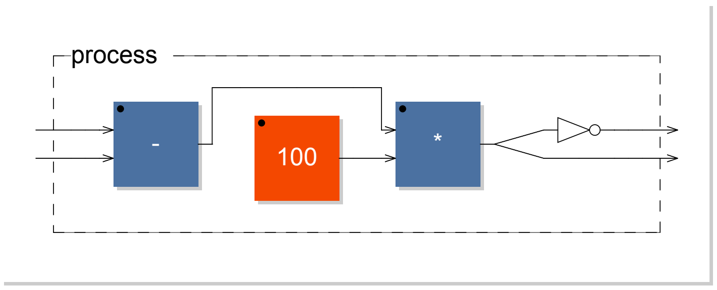


Figure 4: Block diagram of a spring generated by *FAUST*

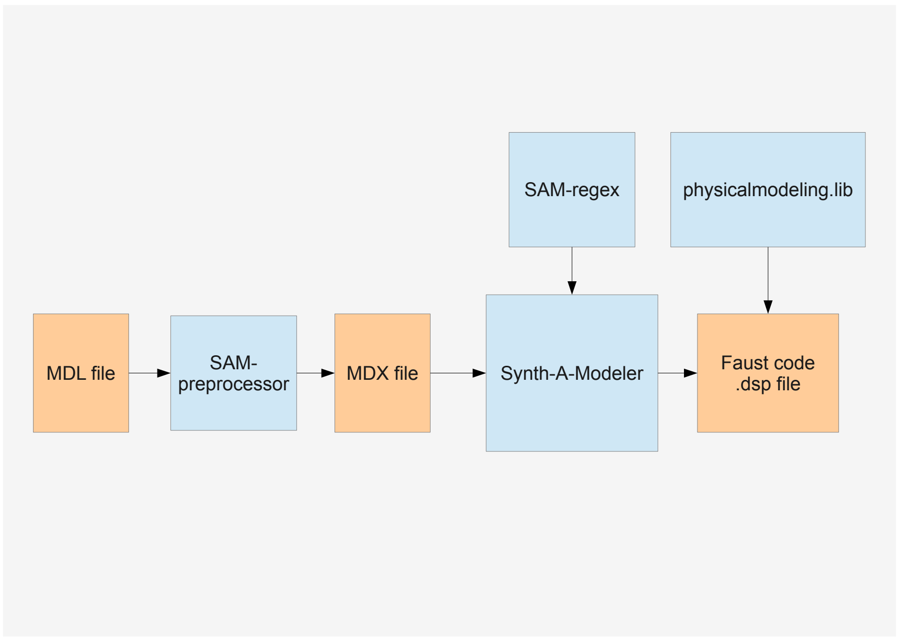


Figure 5: Synth-A-Modeler internal dataflow diagram

is to execute the `SAM-preprocessor` on the *MDL* file to remove all unnecessary content and tidies all lines to prepare it for efficient post-processing. The output is a intermediate file with the extension *.mdx*. Then the `Synth-A-Modeler` script receives the *mdx* file as input. When running the *Synth-A-Modeler* script, the `SAM-regex` file gets included to access its content. The *mdx* file gets parsed and an output *dsp* file is generated. When `Synth-A-Modeler` is finished, the output *dsp* file can be used with *FAUST* to generate an external for a specific audio host. Figure 6 shows the dataflow for creating an external module from a model specification file. The blue boxes represent files from the Synth-A-Modeler package and the orange boxes represent input, output or intermediate files. The user is shielded from these complications by a series of Makefiles that automate this process.

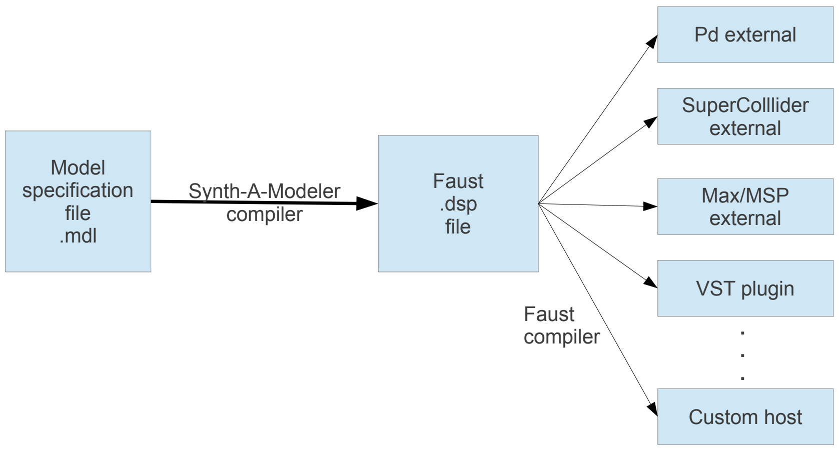


Figure 6: Synth-A-Modeler dataflow diagram

3.2 MODELING

3.2.1 *The model specification file*

Synth-A-Modeler uses a very simple structured and human-readable format for specifying models. It is basically a list of objects and its design is influenced by the *netlist* format, which can be found in many contexts. The most popular use is in circuit design and serves as a universal exchange format [16]. It describes the connectivity of entities for a design. The Synth-A-Modeler specification file lists all the model's mechanical objects and their connections. Listing 2, adopted from Berdahl and Smith III [2], shows a very basic model definition:

Listing 2: A simple model specification

```
link(4200.0,0.0),ll,m1,g;
touch(1000.0,0.03,0.0),tt,m1,dev1;
mass(0.001,0.0,0.0),m1;
ground(0.0),g;
port( ),dev1;
audioout,a1,m1,1000.0;
```

The model represents a very simple synthesizer with one resonance frequency. It consist of mechanical elements that connect to the user's finger allowing the user to touch a virtual mechanical resonator. The values require SI units and English names. A user would read the model as follows: The `mass` *m1* with 0.001 kg is connected to the `ground` *g1* with the position 0 m via the linear `link` *ll*, which has a combination of a spring with a stiffness of 400 N m^{-1} and a damping value of 0.001 N m^{-1} s^{-1}. This connection makes the mass resonate. The `port` *dev1*, which represents to connection to the outside "world" is connected to the mass *m1* via the `touch` link *tt*. The touch link is similar to the linear link, except a force is only exported if one object pushes "inside" the other. The last object, the `audioout` *a1* represents an audio output, which outputs the position of a connected object, in the example in listing 2, it output the position of the `mass` object *m1*. A full *MDL* file with all possible entities, can be found in the appendix. The general syntax of an object is the following:

```
objectname(parameters),unique identifiers,optional other objects;
```

The first word is the object name followed by a parameter list between parenthesis. The number of parameters depends on the object. Following the parameter list, the next value is a unique identifier, followed by a variable number of other values. In our simple example, the link object has two other values after the identifier, which name the other objects it links together. Table 1 shows all possible objects that can be contained in an *MDL* file. The specific parameters are specific to an object, like for a mass object its mass in kg. The generic parameters are values like the unique identifier, which are needed for almost all objects. In the case of link-like objects, the generic parameters can be also the connecting elements at the left and right side. Last but not least, an *MDL* file can have text or comments which is ignored when processing the file. These lines can be recognized by the hash or pound sign (#).

3.2.2 *Abstractions*

Synth-A-Modeler is designed to make it easy to implement abstraction for models that are often used. For example there is already one abstraction implemented. The `resonator` abstraction. It could have been implemented in Synth-A-Modeler by using a mass object and a ground object, connected with a linear link. It is often used and it is handy to have a shortcut for it. The implementation lets us specify multiple resonators in one object by putting additional values into the parameter list. A definition would look like the following code snippet:

```
resonators(400.0, 0.2, 0.01, 300.0, 0.2, 0.02),res1;
```

To specify more than one resonance frequency the three needed parameters are repeated in the list. For this reason the amount of parameters has always to be a multiple of 3.

Table 1: *MDL* object types and their parameters

Object type	**object parameters**	**generic parameters**
mass	mass in kg, initial position in m, initial velocity in $\mathrm{m\,s^{-1}}$	unique identifier
port	—	unique identifier
ground	initial position in m	unique identifier
resonators	frequency in Hz, Decay time in s, Equivalent mass in kg, ...	unique identifier
link	stiffness in $\mathrm{N\,m^{-1}}$, damping in $\mathrm{N\,m^{-1}\,s^{-1}}$, center position offset in m	unique identifier, left connection, right connection
touch	stiffness in $\mathrm{N\,m^{-1}}$, damping in $\mathrm{N\,m^{-1}\,s^{-1}}$, offset for engagement in m	unique identifier, left connection, right connection
pluck	stiffness in $\mathrm{N\,m^{-1}}$, damping in $\mathrm{N\,m^{-1}\,s^{-1}}$, minimum displacement difference for contact in m, offset for engagement in m	unique identifier, left connection, right connection
junction	offset displacement in m	unique identifier
termination	type of termination (reflection coefficient in range of -1.0 to 1.0, low pass strength in non-negative integer)	unique identifier
waveguide	characteristic wave impedance in $\mathrm{N\,m^{-1}\,s^{-1}}$, type of string (maximum time delay in s, current time delay in s)	unique identifier, left connection, right connection
audioout	—	unique identifier, linear combination of source identifier names specifies the mix
faustcode	—	raw *FAUST* code

3.3 COMPILATION RESULT

When compiling a *MDL* file with Synth-A-Modeler the result is a *FAUST .dsp* file as shown in listing 3.

Listing 3: "touch a resonator" *FAUST* file

```
import("physicalmodeling.lib");

bigBlock(m1p,gp,dev1p) = (m1,g,dev1,a1) with {
    // Link-like objects:
    ll = (m1p - gp) : link(4200.0,0.001,0.0);
    tt = (m1p - dev1p) : touch(1000.0,0.03,0.0);
    // Mass-like objects:
    m1 = (0.0-ll-tt) : mass(0.001,0.0,0.0);
    g = (0.0+ll) : ground(0.0);
    dev1 = (0.0+tt);
    // Additional audio output
    a1 = 0.0+m1*(1000.0);
};

process = (bigBlock)~(_,_):(!,!,_,_);
```

The first line consists of the physical modeling library import and also other imports, if present, would be located in the beginning of the file. The next part is the `bigBlock`. It defines all the paths and feedback paths. In listing 3 the objects *m1*, *g1* and *dev1* are fed back. The letter "p" is added to all fed back objects and means "previous". All variables, like *m1*, *g1*, *dev1*, *ll*, *tt*, *a1* are defined within the `bigBlock` as output variables and are only accessible there.

3.4 CREATING EXTERNALS

One important feature of the combination of Synth-A-Modeler and *FAUST* is the possibility to generate externals in many formats and for many different audio hosts. The combination of *FAUST* and the *gcc*[10] compiler makes it possible to generate these targets. Based on Smith III [18], this section will describe the process of creating externals for *PureData*, *SuperCollider* and *Qt/GTK*. The example *MDL* file will be the `touch_a_resonator.mdl` and its compiled *FAUST* file `touch_a_resonator.dsp`.

3.4.1 *Pure Data*

This section will describe how to generate a PureData [19] plugin using the *FAUST* compiler, its architecture file `puredata.cpp` and the script `faust2pd`. The latter was implemented by Albert Gräf [20] and

10 GCC is the compiler suite of the GNU project. It consist compiler frontends for many languages like C, C++, Objective-C etc. [17]

uses the *Pure* programming language.[11] The section will not go into detail with PureData and it is assumed that the reader is familiar with it. The following commands will be automatically issued if the user has installed the required components and uses the command `make puredata`.

3.4.1.1 *Generating a PureData plugin*

A plugin for PureData can be compiled on Linux or MacOSX with following commands on the command-line:

```
$ faust -a puredata.cpp -o touch_a_resonator.cpp touch_a_resonator.dsp
$ g++ -DPD -Wall -g -shared -Dmydsp=touch_a_resonator \
      -I/usr/include/pdextended \
      -o touch_a_resonator~.pd_linux  touch_a_resonator.cpp
```

The command on the first line generates a C++ file, which encapsulates the code for a PureData compatible plugin. PureData has an Application Programming Interface (API) which lets the user program a plugin, an externally compiled loadable module. The second line creates the actual PureData plugin. Its a dynamic loadable binary object file with the name `touch_a_resonator~.pd_linux`. For this to work, PureData needs to be installed and as in the command visible, the path `/usr/include/pdextended` has to be present on the machine, consisting of the PureData C header files, with the main file `m_pd.h`. The last could be installed in some other place and it is advised to look up its location before compiling. On Mac OS X the PureData include files are usually in the application's directory, in `/Applications/Pd-extended.app/Contents/Resources/include/` and the second command has to be changed accordingly.

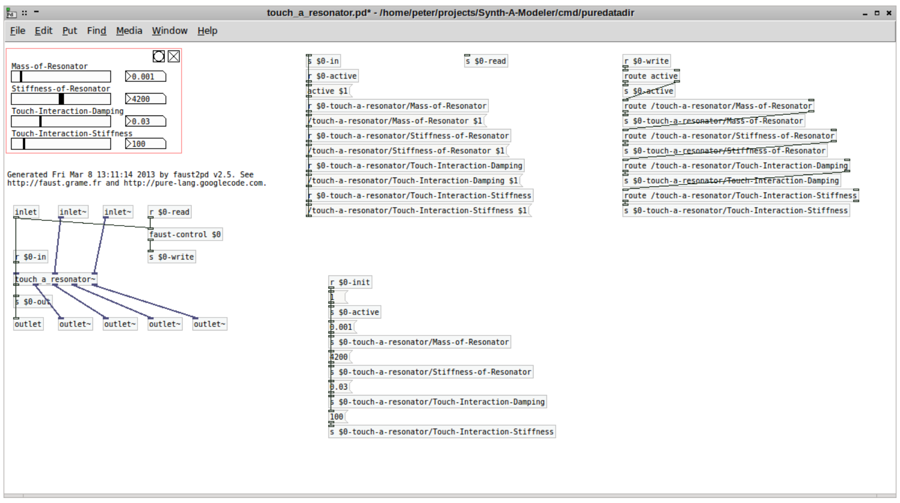
3.4.1.2 *Using `faust2pd` to generate a PureData patch*

Although the PureData plugin generated in section 3.4.1 is fully functional, it is only the "raw" PureData object. To use it in PureData the user has to create a patch. The *FAUST* install includes the `faust2pd` script, which generates a PureData patch automatically. It has the advantage that it creates also sliders and other elements, which were specified in the *MDL* file as raw *FAUST* code. As already mentioned the script is written in the *Pure* programming language and it is of course necessary to install it before using the script. Install instruction can be found on the *Pure* website[12]. To generate the patch file following commands have to be executed on the command-line:

```
$ faust -xml -a puredata.cpp -o touch_a_resonator.cpp \
     touch_a_resonator.dsp
$ faust2pd touch_a_resonator.dsp.xml
```

11 https://code.google.com/p/pure-lang/
12 https://code.google.com/p/pure-lang/

`faust2pd` uses the `touch_a_resonator.dsp.xml` file, which is generated by faust when using the option `-xml`. The generated patch can be loaded in PureData and could look like the patch shown in fig. 7.


touch_a_resonator.pd* - /home/peter/projects/Synth-A-Modeler/cmd/puredatadir
File
Edit
Put
Find
Media
Window
Help
Mass-of-Resonator
0.001
Stiffness-of-Resonator
4200
Touch-Interaction-Damping
0.03
Touch-Interaction-Stiffness
100

```
Generated Fri Mar 8 13:11:14 2013 by faust2pd v2.5. See
http://faust.grame.fr and http://pure-lang.googlecode.com.

inlet    inlet~    inlet~    r $0-read
                             faust-control $0
r $0-in                      s $0-write
touch_a_resonator~
s $0-out
outlet   outlet~   outlet~   outlet~   outlet~
```

```
s $0-in                                                s $0-read
r $0-active
active $1
r $0-touch-a-resonator/Mass-of-Resonator
/touch-a-resonator/Mass-of-Resonator $1
r $0-touch-a-resonator/Stiffness-of-Resonator
/touch-a-resonator/Stiffness-of-Resonator $1
r $0-touch-a-resonator/Touch-Interaction-Damping
/touch-a-resonator/Touch-Interaction-Damping $1
r $0-touch-a-resonator/Touch-Interaction-Stiffness
/touch-a-resonator/Touch-Interaction-Stiffness $1
```

```
r $0-write
route active
s $0-active
route /touch-a-resonator/Mass-of-Resonator
s $0-touch-a-resonator/Mass-of-Resonator
route /touch-a-resonator/Stiffness-of-Resonator
s $0-touch-a-resonator/Stiffness-of-Resonator
route /touch-a-resonator/Touch-Interaction-Damping
s $0-touch-a-resonator/Touch-Interaction-Damping
route /touch-a-resonator/Touch-Interaction-Stiffness
s $0-touch-a-resonator/Touch-Interaction-Stiffness
```

```
r $0-init
1
s $0-active
0.001
s $0-touch-a-resonator/Mass-of-Resonator
4200
s $0-touch-a-resonator/Stiffness-of-Resonator
0.03
s $0-touch-a-resonator/Touch-Interaction-Damping
100
s $0-touch-a-resonator/Touch-Interaction-Stiffness
```

Figure 7: "touch a resonator" PureData patch

3.4.2 *SuperCollider*

SuperCollider is a real-time programming environment for sound synthesis and algorithmic composition [21]. It is assumed that the reader is familiar with SuperCollider. For further reading and in-depth information it is suggested to read Wilson et al. [22]. With *FAUST* it is also possible to create an external from a *dsp* file for SuperCollider, because it has also a plugin API similar to the one, PureData provides. The resulting plugin is a dynamic loadable binary object. *FAUST* provides the script `faust2supercollider` to generate a SuperCollider plugin:

```
$ faust2supercollider touch_a_resonator.dsp
```

On Mac OS X additional steps are needed to be able to compile the dynamic library. Usually the SuperCollider ditribution does not include the header files, which are needed for the compilation process, you also have to download the SuperCollider source code and set the environment variable `SUPERCOLLIDER_HEADERS` to point to the source folder. You have to set this variable in `~/.bash_profile` (if you use `bash`) by adding the following line to it

```
SUPERCOLLIDER_HEADERS=pathtoSCsource/common/Headers
export SUPERCOLLIDER_HEADERS
```

where `pathtoSCsource` is the root of the SuperCollider source directory. The output is a SuperCollider class file, with the extension *.sc*, as shown in listing 4 and a dynamic load library with the extension *.so* on Linux and *.scx* on Mac OS X.

Listing 4: "touch a resonator" SuperCollider class

```
FaustTouchAResonator : MultiOutUGen
{
  *ar { | in1, in2, mass_of_resonator(0.001), stiffness_of_
      resonator(4200.0), touch_interaction_damping(0.03), touch_
      interaction_stiffness(100.0) |
      ^this.multiNew('audio', in1, in2, mass_of_resonator,
          stiffness_of_resonator, touch_interaction_damping,
          touch_interaction_stiffness)
  }

  *kr { | in1, in2, mass_of_resonator(0.001), stiffness_of_
      resonator(4200.0), touch_interaction_damping(0.03), touch_
      interaction_stiffness(100.0) |
      ^this.multiNew('control', in1, in2, mass_of_resonator,
          stiffness_of_resonator, touch_interaction_damping,
          touch_interaction_stiffness)
  }

  checkInputs {
    if (rate == 'audio', {
      2.do({|i|
        if (inputs.at(i).rate != 'audio', {
          ^(" input at index " + i + "(" + inputs.at(i) +
            ") is not audio rate");
        });
      });
    });
    ^this.checkValidInputs
  }

  init { | ... theInputs |
      inputs = theInputs
      ^this.initOutputs(4, rate)
  }

  name { ^"FaustTouchAResonator" }
}
```

These files has to be placed into SuperCollider's extension folder to make it availabel to SuperCollider. The extension folder differs on different operating systems. On Mac OS X it is

`~/Library/Application Support/SuperCollider/Extensions`

on Linux it is usually

`~/.local/share/SuperCollider/Extensions`

3.4.3 *Qt/GTK*

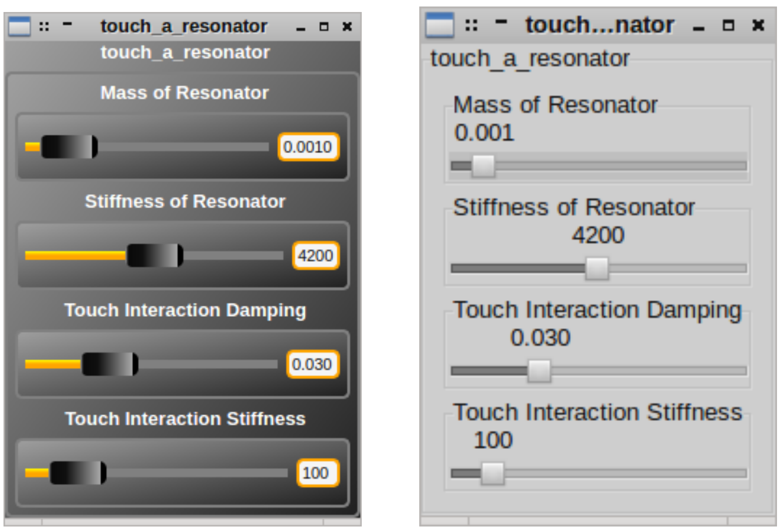
With *FAUST* it is possible to generate standalone applications, which are compatible with the *JACK Sound Server*.[13] The user can choose between a Qt[14] and a GTK[15] standalone application. The difference is that the GUI framework that is used to create the application. There are two scripts provided by *FAUST*, `faust2jaqt` and `faust2jack`. To generate a standalone application following commands have to be run on a *dsp* file:

```
$ faust2jaqt touch_a_resonator.dsp
```

for a Qt application or

```
$ faust2jack touch_a_resonator.dsp
```

for a GTK application. The output of these scripts is an executable called `touch_a_resonator` on Linux and an executable `touch_a_resonator.app` on Mac OS X. The resulting applications are shown in fig. 8.



touch_a_resonator
touch_a_resonator
Mass of Resonator 0.0010
Stiffness of Resonator 4200
Touch Interaction Damping 0.030
Touch Interaction Stiffness 100

touch...nator
touch_a_resonator
Mass of Resonator 0.001
Stiffness of Resonator 4200
Touch Interaction Damping 0.030
Touch Interaction Stiffness 100

(a) Standalone Qt application (b) Standalone GTK application

Figure 8: Standalone applications

13 http://jackaudio.org/
14 https://qt-project.org/
15 http://www.gtk.org/

Part III

SYNTH-A-MODELER DESIGNER

4 DESIGN

Synth-A-Modeler Designer, the application developed in the course of this thesis, is the front-end for the Synth-A-Modeler compiler [2] discussed in chapter 3. Its main purpose is, as the applications' name already suggests, to design and create physical models graphically, without having to edit the textual form of the model. Additionally it should give easy access to physical modeling for composers, artist and students, who prefer visual thinking. A GUI would improve some of Synth-A-Modeler's usability requirements, which were discussed in section 3.1.1. It would be an even more descriptive educational platform for the exploration of vibrating mechanical systems. Furthermore the graphical representation and graphical design could be also included in the creative process of physical modeling, but of course this idea has to be evaluated more intensively, by observing the usage of the GUI over a longer range of time following the completion of this thesis.

4.1 SPECIFICATION

The design of *Synth-A-Modeler Designer* can be divided into two categories. The first category describes the visual functionalities, for instance how the application's appearance should be and what actions can be performed by the user. This also would describe each component of the application, i. e. which detail information can be seen on them. Additionally it will outline which user interface concepts have been adopted. The second category will describe technical parameters, which have been formulated to create an usable application. To provide a good usability experience, both, the visual and the technical specification, have been derived from established concepts seen or experienced in other applications.

4.1.1 *Visual specification*

The main component of the application, the model editing window, will show the physical models loaded from a model specification file and on a canvas-like area where the objects will have a variable position. A quite similar concept can be seen in visual programming environments such as Max/MSP[16] and PureData.[17] The objects are shown on a canvas and their position can be changed by dragging them with

16 http://cycling74.com/products/max/
17 http://puredata.info/

the mouse to the desired location. This concept is very suitable for *Synth-A-Modeler Designer*, because it has to visualize physical models and make it possible to change their positions and arrangement and interact with them by clicking or dragging. The application will also have a menu bar with pull-down menus, consisting of all actions that can be performed on the *MDL* file. Because *Synth-A-Modeler Designer* will utilize the *Synth-A-Modeler Compiler*, an area will be necessary which shows the text that the compiler generates when executed. Fig-

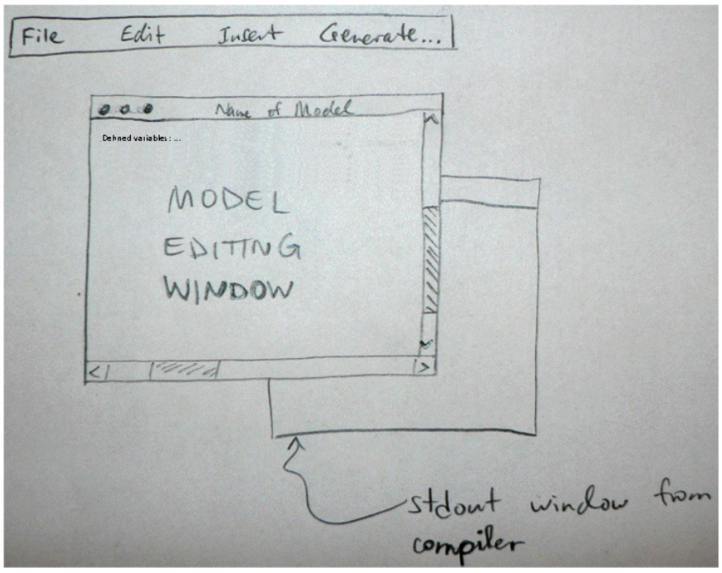


Figure 9: Mockup of the main *Synth-A-Modeler Designer* interface (by Edgar Berdahl)

ure 9 shows a mockup of the main interface, with the model editing window, the compiler output window and the menubar. The windows are floating and resizeable and the content of the model editing window is also zoomable and has scroll bars to allow to adjust the view. If there are a lot of objects on the canvas it can be helpful to restrict the viewing area to a smaller part of the model. In this case the scroll bars can be used to visit other parts of the model. The top bar of the window should show the model name to help to navigate through windows in the case when having many models open. There will always be only one compiler output window, which will show the compiler output from all models. The menu bar will have the standard entries, such as “File” and “Edit” but also entries which are specific to *Synth-A-Modeler Designer*. The location of the menubar will be managed by the operating system, for example on Mac OS it will be in the Mac menu bar, and on other operating systems like Windows and Linux it will be located in the window itself. The mockup in fig. 9 shows also an “Insert” entry, which will provide the necessary actions to insert physical objects and a “Generate” entry with the commands to run the *Synth-A-Modeler Compiler* and the *FAUST* compiler to generate externals. The “File” menu entry (fig. 10a) provides

File	
New	*Cmd-N*
Open...	*Cmd-O*
Open Recent	>
Close	*Cmd-W*
Save	*Cmd-S*
Save As...	
Quit	*Cmd-Q*

(a) "File" pull-down menu

Edit	
Undo	*Cmd-Z*
Cut	*Cmd-X*
Copy	*Cmd-C*
Paste	*Cmd-V*
Select All	*Cmd-A*
Define Variables	*Cmd-D*
√ Segmented connectors	*Cmd-T*
Zoom In	*Cmd-=*
Zoom Out	*Cmd- -*
Reverse direction	*Cmd-R*

(b) "Edit" pull-down menu

Insert	
Mass	*Cmd-1*
Ground	*Cmd-2*
Resonator	*Cmd-3*
Port	*Cmd-4*
Linear Link	*Cmd-5*
Touch Link	*Cmd-6*
Pluck Link	*Cmd-7*
Audio Output	*Cmd-8*
Waveguide	*Cmd-9*
Termination	*Cmd-0*

(c) "Insert" pull-down menu

Generate...	
Generic Faust Code	*Cmd-G*
External Object	*Cmd-E*

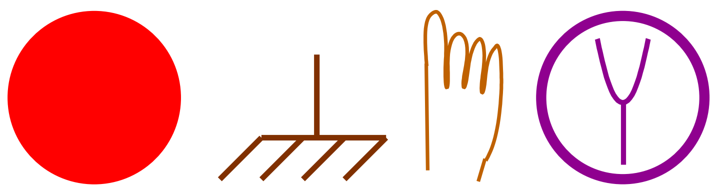
(d) "Generate" pull-down menu

Figure 10: Mockup of *Synth-A-Modeler* pull-down menus (by Edgar Berdahl)

all actions needed to manage the *MDL* file. Here the user can open, save, create a new and close the model as well as quit the application. The "Edit" (fig. 10b) entry provides actions for copy and paste, undo and redo, selection of objects and defining raw *FAUST* code, which is shown as "define variables." It shows also actions for zooming in and out, reversing direction of the connection of objects. The "Insert" menu (fig. 10c) will show a list of all possible physical object that are available in *Synth-A-Modeler Designer*, and upon selecting one of the items, the object will be added to the canvas. The color of the entries indicate the color of the corresponding physical objects available in *Synth-A-Modeler Designer*. The "Generate" (fig. 10d) menu has two main entries, Generic Faust Code and External Object. When clicking the first one the *Synth-A-Modeler Compiler* will be called to generate a `.dsp` file from the `.mdl` file. When clicking on the External Object item, the *FAUST* compiler will be called to generate an external object.

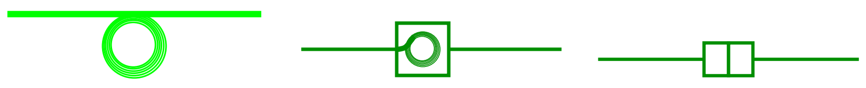
Figure 10d does not show it, but after specifying external targets in the applications' preferences, a list of possible targets will be shown in the "Generate" menu and the user has to select one, before executing the generation of an external object.

The figures 11, 12 and 13 show the icons of the physical objects which are implemented in *Synth-A-Modeler Compiler* at the time of this writing. Another visual specification is the ability to display



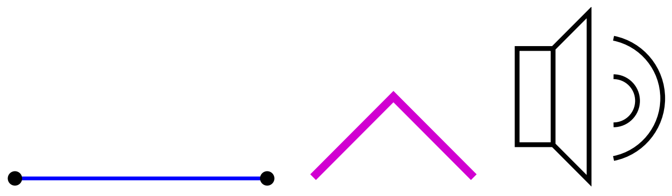
(a) Mass (b) Ground (c) Port (d) Resonator

Figure 11: Mass-like object icons



(a) Link (b) Touch (c) Pluck

Figure 12: Link-like object icons



(a) Waveguide with junctions (b) Termination (c) Audioout

Figure 13: Waveguide and audioout object icons

connections between objects in two ways. Apart from the direct connection, i. e. a straight and possibly diagonal line from one object to another, it should be possible to switch to a segmented display of connections. In other words, the connections are rectangular, broken into vertical and horizontal lines. This would make a large model in some cases more clear and visible. This is analog to segmented patch cords in Max/MSP. Figure 14 shows the two connection display types. The application has to provide an interface for editing parameters of an object. On double clicking the physical object, a window pops up, which presents and lets the user adjust all object parameters, such as unique identifier or specific parameters described in table 1, e. g. the stiffness of a link-like object. The units of the parameters should always be shown in the parameter editing window. To enter raw *FAUST*

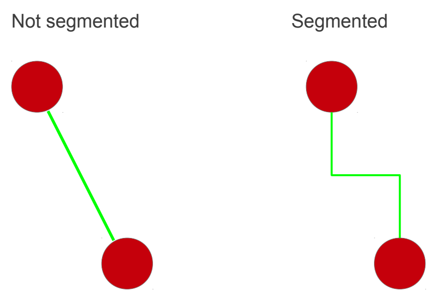


Figure 14: Comparison, unsegmented and segmented connections

code there is another window, which can be opened on clicking on "define variable" item in the "Edit" pull-down menu. It provides the user a text-editor like interface to enter *FAUST* code line by line. The gain values of audio outputs are set in a dialog window which opens when the user clicks on an audio connection between an object and an audioout object.

4.1.2 *Technical specification*

In addition to the visual specifications in section 4.1.1, also technical specifications have been defined for *Synth-A-Modeler Designer*, to ensure a positive user experience while editing physical models. This section will discuss the most important points.

Due to the fact that not all objects can be connected to each other, *Synth-A-Modeler Designer* should prevent such cases. Mass-like objects are only connected via link-like objects and vice versa. That means for example, that two link-like objects should not be able to get connected. A direct connection between a mass and a port object should also be prevented.

Also for waveguide objects there are certain connection rules. A waveguide object can only be connected to a termination or a junction object, and link-like objects but no mass-like objects can connect to a junction object, with the additional restriction of maximum one link-like object per junction. An exception is the audioout object, which only can be connected to all other objects via a special audio connection object, which only will be visible in the graphical representation of an `.mdl` file, not in the file itself. Additionally, the physical model specification file format has to be extended, to store the graphical position on the editing canvas. It has been decided to store the x and y values at the end of the objects line as "commented text" with a preceding # sign. An object with position values (100,150) would look like the following code snippet from an `.mdl` file:

```
mass(0.001, 0.0, 0.0), m1; # 100, 150
```

Please note the end of the line. This notation would prevent the *Synth-A-Modeler Compiler* to parse the values and eventually fail because of the unsupported code. In case of missing position data, *Synth-A-Modeler Designer* should be able to assign position values and align the objects automatically. The specific method and its implementation will be discussed later in this thesis in section 6.2.

As discussed in section 4.1.1 the editing canvas has to be zoomable. When the canvas is zoomed in and not everything of its contents is visible, scroll bars should make it possible to get to all parts of the viewport. This behavior should be controlled not only by clicking and dragging the scroll bars with the mouse, but also with key bindings and with commands in the pull down menus. It should also be prevented to assign the same identifiers to two objects and therefore the parameter editing window should automatically implement a method to prevent this. Furthermore, it should be possible to do common interactions, that are standard in many applications and which the user is familiar with, such as copy, paste and cut of objects. The parameters have to be copied along with the objects, however the identifiers have to be changed automatically. Additional housekeeping should also be implemented, to correct values that are entered by the user and does not fulfill format specifications, e. g. when the user enters a numeric value as integer, it should be automatically corrected to a floating point value, because *Synth-A-Modeler Compiler* only handles floating point values.

4.2 SOFTWARE ARCHITECTURE

The *Synth-A-Modeler Designer* is intended to be a desktop application for the platforms *Mac OS*, *Windows* and *Linux* and multiple architectures, such as *Intel*[18]/*AMD*[19] 32 and 64 bit processors as well as *ARM*[20] processors. The main critical part for a good performance of the application is the graphical visualization of the physical objects and should be carefully designed. Apart from an object oriented and modular design, which makes a collaboration and later extension of the application easier, it has been decided to implement the application using the MVC pattern. The MVC pattern is a group of three class types and originates from *Smalltalk-80*[21] [23], where it has been used to build user interfaces. It consists of three kinds of objects: the Model, the View and the Controller. That means that the functionality of an application is divided into these three groups. The

18 http://www.intel.com
19 http://amd.com
20 http://www.arm.com/
21 Smalltalk is a object-oriented, dynamically typed programming language http://smalltalk.org/

model holds the data to be presented and implements also logic of the data structures and is independent from the controller and view. The view is responsible for displaying the data of the model and also for the interaction between user and application. Usually it knows about the model and about the controller, but it is not responsible for processing data generated from user interactions. The controller manages views and gets commands from them and sends them to the model to manipulate its data. A schematic of the MVC components collaboration shows fig. 15. This kind of decoupling of com-

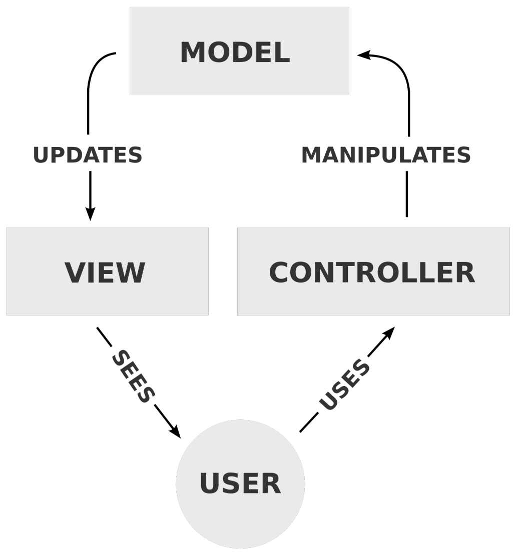


Figure 15: Collaboration of MVC components[22]

ponents makes the design of the application more flexible, reusable and modular. It will be easier to modify and add functionality later to the application, which accommodates the general requirements of the *Synth-A-Modeler* project. The decoupling of the view and model is established by a subscribe/notify protocol [23]. Furthermore, as Erich et al. states, a view has to mirror the state of the model exactly with its appearance [23]. Every time the data of the model gets changed, it notifies dependent views to update themselves. This strategy allows having different views for the model. In the case of *Synth-A-Modeler Designer*, the different views would be the representation of the objects on the canvas on the one hand and the listing of object parameters on the other. Both views get their data from the same model but they provide a different representation of it. MVC also allows to modify the response to user input without modifying its visual display, e. g. it is possible to change a responding action, which is caused by performing a keyboard shortcut command.

22 Source: https://commons.wikimedia.org/wiki/File:MVC-Process.svg

5 TECHNOLOGY SELECTED

This chapter will focus on the technology selected for the development of Synth-A-Modeler Designer. The goal was to find the optimal tools for the tasks needed for the development, matching the requirements and optimizing development time. For example, it has been decided to build a cross-platform application in order to make it as accessible as possible, which made the pool of options smaller. The *JUCE* Library is the C++ toolkit used to develop Synth-A-Modeler Designer. Also the Regular Expression Library for parsing the model specification file had to be chosen to support all platforms. Google's *re2* library fulfilled the needs for this task while being fast, having small overhead and simple syntax. The following sections will give a deeper insight into the specific tools used to develop Synth-A-Modeler Designer.

5.1 JUCE C++ LIBRARY

The JUCE (Jules' Utility Class Extensions) library[23] is a general purpose C++ class library for the development of cross-platform applications. It has been released in 2004 and is developed and maintained by *Raw Material Software*[24] and has a dual GPL/commercial license. The library is similar to C++ libraries like Qt[25] or wxWidgets[26] and contains almost all parts and tools needed for the creation of application, which use GUIs, graphics, sound or networking. It provides also wrapper classes for common audio plugins like VST, AudioUnit[27] and ProTools'[28] RTAS[29] and AAX[30] formats. With the *JUCE* library it is possible to write applications and deploy them on various platforms with the same codebase. The supported platforms are the following:

- Windows XP, Vista and Windows 7,
- Mac OS 10.4 and later,

23 http://rawmaterialsoftware.com/juce.php
24 http://rawmaterialsoftware.com
25 https://qt-project.org/
26 http://wxwidgets.org/
27 AudioUnit is a audio plugin architecture developed by Apple http://developer.apple.com/documentation/MusicAudio/Conceptual/AudioUnitProgrammingGuide/
28 Misc production software by the company Avid http://www.avid.com/products/family/Pro-Tools
29 RTAS is an audio plugin architecture used in ProTools
30 AAX is an audio plugin architecture used in ProTools

- iOS 2.1 and later,
- Linux with kernel 2.6 and later, and
- Android with NDK-v5 and later.

JUCE supports the use of following compilers:

- GCC version 4 and later,
- LLVM Clang version 1.5 and later, and
- Microsoft Visual Studio Visual C++ version 6 and later.

As the JUCE website [24] states, JUCE provides an extensive range of features, which consist of classes for graphical user interfaces, graphics, OpenGL, networking, cryptography, image processing, audio, XML parsing and other features. With this capacity, the usage of third-party libraries can be reduced to a minimum. Compared to other libraries like *Qt* and *wxWidgets*, JUCE has a lot of audio related features, originating from the fact that JUCE was developed as part of the development of the audio sequencer software Tracktion.[31] *JUCE* has a very good and thorough documentation,[32] a user forum,[33] and provides very good example projects which are a good starting point for development with the JUCE library. Although it is possible to set up *JUCE* projects manually, an application called *Introjucer* has been developed for the purpose of creating and managing projects. The user has to specify the files and settings for a project, and then the Introjucer generates the necessary project files for the various platforms. It generates the Xcode project on Max OS and for iOS, Visual Studio project files on Windows, Linux Makefiles and Ant[34] builds for Android. *JUCE* provides also the visual GUI editor application "The Jucer", which makes possible to edit and create user interface components and save them as C++ code. *JUCE* has been awarded with Dr. Dobb's Jolt Productivity Award[35] and has been used in many free and commercial projects.[36] As stated in the beginning of this section *JUCE* has a dual license. It is released under the GNU General Public License version 2 [25], which means that it can be copied and distributed freely and does not cost anything, when used in open-source applications. This also means it has some restriction regarding the usage of third party libraries, and it has to be made open-source. It is also possible to purchase a commercial license to avoid the restrictions and to be able to use *JUCE* for closed-source projects.

31 http://www.tracktion.com/

32 http://rawmaterialsoftware.com/juce/api/classes.html

33 http://rawmaterialsoftware.com/index.php

34 Ant is a tool for automated building of Java software http://ant.apache.org/

35 http://www.drdobbs.com/joltawards/jolt-productivity-awards-app-libraries-a/227200111

36 http://rawmaterialsoftware.com/wiki/index.php/3rd-party_JUCE_Applications

Until February 2012,[37] JUCE was a regular C++ library, which had to be built as a static or dynamic library in order to link against it and use it in a project. However, the developer of *JUCE*, Julian Storer recently transformed it into a "unity build." As OJ Reeves writes in his article, a unity build is technique to save compilation and link time by including many *cpp* files (in this case the whole *JUCE* library) into a single separate *cpp* file and to compile only that unity *cpp* file [26]. Additionally, *JUCE* is divided into several logical modules for the purpose not having to include all the functionality provided by *JUCE*, only the needed. The modules are the following:

- audio basics
- audio devices
- audio formats
- audio plugin client
- audio processors
- audio utils
- browser plugin client
- core
- cryptography
- data structures
- events
- graphics
- gui basics
- gui extra
- opengl
- video

For the development of the Synth-A-Modeler Designer not all of *JUCE*'s functionality is required and the only modules needed are

- core
- cryptography
- data structures
- events
- graphics
- gui basics
- gui extra

With this feature to separate the library, compilation time can be saved and also the compiled executable size will be smaller. The selection of the modules happens in the *Introjucer*. There is also the option to include the *JUCE* codebase into the project to guarantee a specific version of the library while under development. The other option is to have *JUCE* at an external location and not to include it in the project. For the time of the development the latter option will be used to be able to follow the steady development of *JUCE*. When Synth-A-Modeler Designer will be released, it will be switched to the

37 `https://github.com/julianstorer/JUCE/commit/aa6e9d38deca22d661218cabcbb745f6a0fea64b`

first option to prevent breakage in case of some changes in the *JUCE* API.

When creating a project with the *Introjucer* tool, it creates a reasonable directory structure as the following structure shows:

```
JuceExampleProject/
 ├─ Builds/
 │   └─ Linux/
 │       └─ Makefile
 ├─ JuceLibraryCode/
 │   ├─ module/
 │   ├─ AppConfig.h
 │   ├─ BinaryData.h
 │   ├─ BinaryData.cpp
 │   └─ JuceHeader.h
 ├─ Source/
 │   └─ Main.cpp
 └─ JuceExampleProject.jucer
```

A project directory holds the *Introjucer* project file (`JuceExampleProject.jucer`), the `Builds` directory with files needed to build the application (in the example only a Linux makefile is shown), the *JuceLibraryCode* directory with the *JUCE* library files and of course the `Source` directory with the application's source files. The folder *JuceLibraryCode* consists of an additional `modules` folder, which contains the *JUCE* header files. `AppConfig.h` has all preprocessor definition for the *JUCE* modules. When having a binary file, i. e. the application's icon or images for the user interface, the usual way is to include them in *Introjucer* and mark them as binary files. *Introjucer* automatically converts them into the source files `BinaryData.h` and `BinaryData.cpp` which then can be included in the project. This way it is not necessary to distribute assets separately, but rather including them directly into the application binary file. The file `JuceHeaders.h` is the file which has to be included in the example application to be able to use *JUCE* code.

5.2 REGULAR EXPRESSIONS

For the lexical analysis and tokenization of an *MDL* file in Synth-A-Modeler Designer, it has been decided to use regular expressions. "Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings" [27]. With REs it possible to define patterns that represent a set of character or the order of a set of character. There are many cross-platform RE libraries written in C++. Two popular libraries are *PCRE*[38] and Boost.Regex.[39] The latter has been integrated into the latest C++ standard *C++11*,[40] but to be

38 http://pcre.org/
39 http://www.boost.org/doc/libs/1_53_0/libs/regex/doc/html/index.html
40 http://www.open-std.org/jtc1/sc22/wg21/docs/standards#14882

compatible with older compilers, which do not implement *C++11*, it has been decided to use an external library. One criteria for the RE library was to be cross-platform. Both mentioned libraries are cross-platform. Another criteria was also to have a small footprint, i. e. to be very small and with no or few dependencies. *PCRE* and Boost.Regex do not match these criteria and other libraries had to be evaluated. Finally, it has been decided to use Google's *RE2*.[41] It uses also the Perl syntax like *PCRE*. *RE2* is very fast, because in contrast to the other mentioned libraries, it uses automata theory [28] which guarantees a linear search run time. Details about the implementation can be found in Cox [29]. The library is not completely compatible with the Windows operating system, but there is a Windows port[42] available which is used in the Windows version of Synth-A-Modeler Designer. This section will not go into detail how REs work and the RE syntax, however it will only describe parts, which are related to the usage in Synth-A-Modeler Designer. Consider a line from an *MDL* file like the following:

```
mass(0.001, 0.0, 0.0),m1;
```

In order to read and separate all values, we do not go through the line character after character, instead we define a RE pattern which matches and extracts the values. The matching RE pattern is the following:

```
\A\s*(mass)\(\s*(\s*[^\n\r\a\e\f]*\s*)\s*\)\s*,\s*([a-zA-Z\d]*)\s*;\s*$,
```

which would achieve the same result in *Perl*. This RE would extract the type of the object (`mass`), the list of parameters (`0.001, 0.0, 0.0`) and the unique identifier (`m1`). The parameter list has then to be extracted into single parameter values separately. A full list of *RE2*'s syntax can be found on the *RE2* website.[43]

5.3 GIT

Git[44] has been chosen to be the source code management (SCM) system for the development of Synth-A-Modeler Designer. *Git* is a distributed revision control and SCM system and was developed by Linux Torvalds for the development of the Linux kernel. Its main requirement was to be very fast and to ensure the integrity of the source code which is managed by *Git* [30]. The integrity is secured cryptographically and it is guaranteed that same code that goes into the system and comes out, remains exactly the same and will not be corrupted. As mentioned before, *Git* is a distributed system and unlike other popular and widespread SCM systems like *Subversion*,[45] which

41 https://code.google.com/p/re2/
42 https://code.google.com/p/re2win/
43 https://code.google.com/p/re2/wiki/Syntax
44 http://git-scm.com/
45 http://subversion.apache.org/

keeps its content in a central repository, you can commit changes locally and no network connection is needed to perform these actions. Its feature to merge changes reliably and fast and additional useful features like stashing uncommitted changes and committing changes line-by-line enhance productivity enormously. *Git* is used by many major open-source projects like the Linux kernel, Gnome or Android and companies like Google, Twitter and Netflix. *Git* has been used in many personal projects in the past and has been proven as very reliable and flexible.

5.4 GITHUB

Github[46] is a web-based hosting service for source code and software development. As the name suggests it uses *Git* as SCM system. *Github* was founded in 2007 and is with more than 3 million user the most popular[47] web service for software development. The company offers free accounts with public repositories as well as paid accounts which enable the use of private repositories. *Github* not only enables hosting of the source code, it is possible to set up a Wiki and a Bug tracker and an Issue system. The Wiki host the helps system for *Synth-A-Modeler* and all feature discussions and bug reporting will happen on *Github*. With *Github*'s collaboration features it is a very good platform to work together it was an easy decision to choose the hosting of *Synth-A-Modeler Designer*. The project's URL is:

`https://github.com/ptrv/Synth-A-Modeler`

46 https://github.com/
47 https://github.com/blog/865-github-dominates-the-forges

6 IMPLEMENTATION

This chapter describes the implementation details of the *Synth-A-Modeler Designer* application in its final state. Additionally, section 6.3 documents how to extend *Synth-A-Modeler Designer* with a new type of object, and particularly it will go into detail which steps are needed for the extension and where exactly in the source code new code has to be added to make the new object fully functional. The final section of this chapter (section 6.4) will describe some limitations that have come up during implementation.

The project's source code directory structure is the following:

```
Synth-A-Modeler/
 ├── SaM/
 ├── extras/
 ├── gui/
 └── juce/
```

The `SaM` directory consist of the *Synth-A-Modeler Compiler*, the directory `juce` contains the *JUCE* source code, while the directory `gui` holds all files for the *Synth-A-Modeler Designer*. All other files, such as installer scripts, model specification file syntax highlighting for two popular text editors *Emacs*[48] (see appendix A.3) and *VIM*[49] (see appendix A.4) and files not primarily related to *Synth-A-Modeler Compiler* and *Designer* can be found in the directory `extras`. This chapter will only describe the contents of the `gui` folder, whose content is shown below.

```
gui/
 ├── BinaryData/
 ├── Builds/
 ├── Docs/
 ├── JuceLibraryCode/
 ├── Libs/
 │    └── re2/
 ├── Source/
 └── Testsuite/
```

Because of the fact that the project was created with the *Introjucer* program, it is based on an directory structure shown in section 5.1, and therefore only parts of it will discussed, which has not been covered in section 5.1. The folders `BinaryData`, `Docs` and `Libs` are self descriptive and contain the documentation files, the binary files used

48 https://www.gnu.org/software/emacs/
49 http://www.vim.org/

in the graphical interface and the external libraries that are used in the project, such as the *re2* regular expressions library.

The structure of the source files is grouped thematically into folders and is the following:

```
Source/
 |__Application/
 |__Controller/
 |__Graph/
 |__Models/
 |__Utilities/
 |__View/
```

6.1 APPLICATION COMPONENTS

This section will provide in-detail information about *Synth-A-Modeler Designer's* application components and their implementation. This includes the implementation of the core components (section 6.1.1) which are responsible for the application's main infrastructure, such as the main application class, managing windows or the undo/redo mechanism that has been implemented to provide common desktop application functionality. In addition, the section will describe other major components and functionality, for example the internal representation of a model specification file (section 6.1.2), the different view components (section 6.1.4) or the interaction of *Synth-A-Modeler Designer* and external tools (section 6.1.6), such as the *Synth-A-Modeler Compiler*, described in chapter 3 and the *FAUST* exporter tools, described in section 3.4.

6.1.1 *Core components*

One of the core components of *Synth-A-Modeler Designer* is the class `SynthAModelerApplication` which is a subclass of `JUCEApplication` and the main entry point o the application. It gets instantiated in `Main.cpp` with the macro `START_JUCE_APPLICATION` and handles the initialization and shutdown of the application. Every *JUCE* application has to declare a subclass of `JUCEApplication` and implement its pure virtual functions. This way, *JUCE* handles all the platform specific code under the hood and the library user does not have to implement it by himself or herself. The `SynthAModelerApplication` class is also responsible for the window management. It consists of an instance of `OutputWindow`, which is an read-only text editor window displaying the output from the *Synth-A-Modeler Compiler* and *FAUST* tools. When running *Synth-A-Modeler Designer* there is only one instance of `OutputWindow` and when editing multiple models, all compilers write to that one instance, using the function

```
void writeToDebugConsole(const String& title,
                                const String& textToWrite,
                                bool isBold)
```

or

```
void writeToDebugConsole(const String& textToWrite,
                         bool isBold)
```

The former also prints a title before the actual text. The last function argument, `bool isBold`, can be used to influence the font type for text to print. Setting it to `true` prints the text with bold font and using `false`, prints the text with a regular font. `SynthAModelerApplication` also has a list of `MainAppWindow` objects, stored as pointers in an `OwnedArray` data type. `OwnedArray` is a array-like data type provided by *JUCE* for holding pointer type objects, which takes the ownership of the object and will delete the object automatically when it is removed from the array or the array itself gets deleted.

`MainAppWindow` is the model editing window, which is a subclass of `DocumentWindow`. It is basically a re-sizable window with a title bar, close, minimize and maximize buttons. A `DocumentWindow` functions as a frame for holding a content component, which will contain all view components related to model editing functionality and will be later discussed in detail in section 6.1.4. A new window is created for every *MDL* file, except for the case when the window only contains an untitled model, e. g. a model which has no file attached to it. In that case the *MDL* file will be loaded into the untitled window. The `MainAppWindow` is therefore responsible to close, open and set the *MDL* file properly. Another important role of the `MainAppWindow` is to hold the two controllers, the `MDLController` and the `ObjController`, which are responsible to delegate all actions coming from the view to the model and vice versa and will be discussed later in section 6.1.5. The controllers are wrapped with a `ScopedPointer`, which is a data type similar to a regular pointer with the exception that it gets deleted when it goes out of scope, e. g. when the `MainAppWindow` gets deleted by closing an *MDL* file. This makes memory management easier, because the implementation does not have to take care of the deletion of the object and ensures that the memory will definitely be freed.

The menu bar is also generated in `SynthAModelerApplication`. It has to be made sure that a proper platform specific menu bar gets created individually. On a machine with *Mac OS* running the menu bar is different. It is placed on top of the screen and not like on *Windows* and *Linux* in the application window itself. *JUCE* offers the function `setMacMainMenu(MenuBarModel* menuBarModel)` to activate the Mac specific menu. When *Synth-A-Modeler Designer* is running on a *Mac* this function gets called and it sets the menu bar. The menubar entries are not generated all in one place, rather they get added to the menu bar in the classes, where its functionality appears. For example, the menu bar item for closing the application gets added to the menu bar in the `SynthAModelerApplication` class, because it is a functionality for the whole application, however the menu item to insert a new object to the model editing canvas will be added to the menu bar in their respective components because it is a command which is related to a model and its window and not to the whole application. This way the menu items appear only where they have functionality. Otherwise they are deactivated.

To improve readability of the source code, all commands that are passed within the application have named values. These commands are for example used when clicking a menu item. The commands are integer values and can be used in `switch` statements. Listing 5 shows some `CommandIDs`

Listing 5: Example `CommandIDs`

```
namespace CommandIDs
{
    static const int newFile                = 0x200010;
    static const int open                   = 0x200020;
    static const int closeDocument          = 0x200030;
    static const int saveDocument           = 0x200040;
    static const int saveDocumentAs         = 0x200045;
}
```

and listing 6 demonstrate how they are used.

Listing 6: Example usage of `CommandIDs`

```
switch (commandID)
{
    case CommandIDs::newFile:
        // create new mdl file
        break;
    case CommandIDs::open:
        // open mdl file
        break;
}
```

To provide persistent user settings for the application throughout multiple sessions, the class `StoredSettings` has been introduced. The primary function of this class is to read and write customizable application parameters to a settings file. It is implemented as a singleton class, which is a pattern, whose intent is to ensure that a class has only one instance with a global point of access to it [23]. This pattern guaranties that the application is not writing the file twice at the same time. It also provides a mechanism to save the settings file at a default platform specific location. On *Linux* it is the user's `HOME` folder, whereas on *Mac OS* settings are stored in

`~/Library/Application Support/`. The settings file itself is an XML-formatted[50] file as shown in listing 7.

Listing 7: *Synth-A-Modeler Designer* user settings file

```
<?xml version="1.0" encoding="UTF-8"?>

<PROPERTIES>
  <VALUE name="currentexporter" val="puredata"/>
  <VALUE name="recentFiles" val=""/>
  <VALUE name="lastFiles" val=""/>
  <VALUE name="showcompilerwindow" val="1"/>
  <VALUE name="lastMainWindowPos" val="241 120 800 600"/>
  <VALUE name="lastDebugWindowPos" val="1 47 400 400"/>
</PROPERTIES>
```

An additional core feature of *Synth-A-Modeler Designer*, which also has been specified in the requirements for the application, is undo and redo of user interactions and edits on the elements of a model. These include adding and removing objects, changing positions or changing the parameter values of objects. It is also a feature, which is implemented in many application and makes it possible to switch between different changes and also to revert back to an older state of the model. The user will be able to use this feature by clicking on the corresponding menu item or executing the shortcut key combination, which is `Ctrl+Z` for undo and `Ctrl+Shift+Z` for redo. On *Mac OS* the shortcut uses the `Cmd` key instead of the `Ctrl` key. *JUCE* provides classes to implement this functionality. The relevant classes are `UndoManager` and `UndoableAction`. The `UndoManager` manages a list of undo and redo actions and makes it possible to move backward and forward through the list. The actions are subclasses of `UndoableAction`, which perform all needed commands. To perform an actual action, an object has to be created and passed to the `UndoManager`'s `perform()` function with the signature:

```
bool UndoManager::perform(
             UndoableAction* action,
             const String & actionName = String::empty
)
```

`UndoManager` also supports grouping of actions. A group is called a "transaction" and contains all performed actions between calls to the function `beginNewTransaction()`. When performing an undo or redo, the "transaction" with the grouped actions gets undone or redone. In *Synth-A-Modeler Designer* every open model instantiates its own `UndoManager`, which provides a separate undo/redo history for each open model. As mentioned, an action is an instance of a subclass of `UndoableAction`, which has to be created for all actions that should

50 http://www.w3.org/TR/REC-xml/

be undo-able or re-doable. Listing 8 shows the pseudo code for a `AddObjectAction`, which adds an object and removes it when the user performs an undo.

Listing 8: Pseudo code for an `UndoableAction` to add an object

```
class AddObjectAction : public UndoableAction
{
public:
    AddObjectAction()
    {
    }
    ~AddObjectAction()
    {
    }
    bool perform()
    {
        Object* obj = new Object();
        addObject(obj);
        objectAdded = obj;
        return true;
    }
    bool undo()
    {
        removeObject(objectAdded);
        return true;
    }
private:
    Object* objectAdded;
};
```

Writing an `UndoableAction` involves the implementation of a `perform()` and an `undo()` function and in the case of `AddObjectAction` also to store the new value. Listing 8 is a very much simplified example and the creation of an element involves a lot more then the code suggests. Further details will be discussed later in section 6.1.5 and all `UndoableActions` can be observed in the class `ObjectActions.h` in *Synth-A-Modeler Designer*'s source code.

6.1.2 *Internal MDL representation*

When parsing a model specification file in *Synth-A-Modeler Designer*, it has to be transformed into objects within the application. This procedure is mandatory for having access to its contents and to be able to use the model for further processing. This is a common process in software development when dealing with data and is called *serialization*. Consequentially, data-types for all possible model elements have to be generated. In the case of *Synth-A-Modeler Designer* it has been decided to use the `ValueTree` class, provided by *JUCE*, for the purpose. `ValueTree` is a tree structure data-type for storing data without a strict form and also supports the `UndoManager`. It stores its content in a list of named properties and can hold any number of sub-trees. Each `ValueTree` has a type name, essentially like an XML tag in an XML file, and can be also converted easily to an XML file and vice versa. The best practice for `ValueTree`s, according the *JUCE* documentation [31], is to create each object on the stack because its contents are store as shared objects types, which means, an instance is only a reference to the original value and can be copied around cheaply. To actually create a deep copy of a `ValueTree`, an explicit call to `createCopy()` is required. The documentation explains further, that all methods of `ValueTree` that change data take an optional `UndoManager` object, which is used to track changes to the object [31]. A simple model specification file as shown in listing 2 is represented internally in *Synth-A-Modeler Designer* like in listing 9 and is created by `ValueTree`'s method `toXmlString()`.

Listing 9: Internal representation of an *MDL* file

```
<?xml version="1.0" encoding="UTF-8"?>

<synthamodeler mdlName="simple.mdl" mdlPath="/tmp/simple.mdl">
  <masses>
    <mass posX="368" posY="264" identifier="m1">
      <parameters>
        <parameter value="1.0"/>
        <parameter value="0.0"/>
        <parameter value="0.0"/>
      </parameters>
    </mass>
    <ground posX="365" posY="390" identifier="g">
      <parameters>
        <parameter value="0.0"/>
      </parameters>
    </ground>
    <port posX="195" posY="293" identifier="dev1"/>
  </masses>
  <links>
    <link identifier="l1" startVertex="m1" endVertex="g">
      <parameters>
        <parameter value="0.001"/>
        <parameter value="0.001"/>
        <parameter value="0.0"/>
      </parameters>
    </link>
    <touch identifier="t1" startVertex="m1" endVertex="dev1">
      <parameters>
        <parameter value="0.001"/>
        <parameter value="0.001"/>
        <parameter value="0.0"/>
      </parameters>
    </touch>
  </links>
  <audioobjects>
    <audioout posX="515" posY="264" identifier="aLeft" optional="
        ">
      <sources>
        <audiosource value="m1*(1000.0)"/>
      </sources>
    </audioout>
  </audioobjects>
</synthamodeler>
```

As previously stated, the internal representation is analog to the XML file and will be used for further explanations, e. g. the root tag `synthamodeler` is analog to the root in the `ValueTree`, whose name property is `synthamodeler`. An important detail of `ValueTrees` is that the property name has to be an `Identifier` object, which represents a `String` identifier and is very fast to copy but slow to initialize. Therefore *Synth-Modeler Designer* initializes all identifiers when the application starts in the class `ObjectIDs.h`. All identifiers are within the namespace `Ids` and to access an identifier, it has to be called like a regular variable by its name, i. e. `Ids::value`. Listing 10 shows how to create an `ValueTree` structure.

Listing 10: Example assembling an *MDL* file structure

```
ValueTree mdl(Ids::synthamodeler);

ValueTree masses(Objects::masses);
mdl.addChild(masses, -1, nullptr);

ValueTree mass(Ids::mass);
mass.setProperty(Ids::identifier, "m1", nullptr);

masses.addChild(mass, -1, nullptr);
```

The `addChild()` function has the following signature.

```
void addChild (const ValueTree &child,
               int index,
               UndoManager *undoManager)
```

The second argument `index` specifies the index when inserting a child. Using -1 inserts the child at the end of the list. The function signature of `setProperty()` is

```
ValueTree& ValueTree::setProperty(const Identifier & name,
                                  const var & newValue,
                                  UndoManager * undoManager)
```

The root ValueTree that represents the root of an *MDL* file is stored in the class `MDLFile`, which is the model for a model specification file in *Synth-Modeler Designer* and is a subclass of *JUCE*'s `FileBasedDocument` class and `ValueTree::Listener` class. By subclassing `ValueTree::Listener` and registering `MDLFile` on the root `ValueTree`, it is possible to get events from the root `ValueTree` when it changes. The following callback methods have to be implemented to use the `ValueTree::Listener`.

-
  ```
  valueTreePropertyChanged (ValueTree& tree,
                                const Identifier& property);
  ```

-
  ```
  valueTreeChildAdded (ValueTree& parentTree,
                           ValueTree& childWhichHasBeenAdded);
  ```

-
  ```
  valueTreeChildRemoved (ValueTree& parentTree,
                             ValueTree& childWhichHasBeenRemoved);
  ```
- `valueTreeChildOrderChanged (ValueTree& parentTree);`
- `valueTreeParentChanged (ValueTree& tree);`

As previously mentioned, `MDLFile` inherits from `FileBasedDocument`, which is a *JUCE* class that "takes care of the logic involved with the loading and saving of some kind of document [31]". The `MDLFile` is exactly such a document, which needs to be loaded, saved and which has to take care of the history of opened files. Therefore `MDLFile` has to implement following pure virtual functions:

- `String getDocumentTitle();`
- `Result loadDocument (const File& file);`
- `Result saveDocument (const File& file);`
- `File getLastDocumentOpened();`
- `void setLastDocumentOpened (const File& file);`

By implementing the latter functions, `MDLFile` is provided with all functionality that is involved with opening and saving an *MDL* file, which also includes presenting the user with all dialog windows that are necessary when performing these actions. It also keeps track of whether the *MDL* file has changed since it was last loaded or saved. When something gets changed, its `changed()` method gets called and a flag will be set to be aware of whether it needs to be saved.

6.1.3 *MDL parsing and writing*

One of the key functionalities of *Synth-A-Modeler Designer* is the ability to read and write *MDL* file. The classes that provide these features are `MDLParser` and `MDLWriter`, which are quite small classes providing only the parsing and writing functionality. Listing 11 and listing 12 show the header files with the function signatures for parsing and writing *MDL* files.

Listing 11: Class for parsing an *MDL* file

```
class MDLParser
{
public:
    MDLParser(MDLFile& mdlFile_);

    bool parseMDL(const File& f);
private:
    MDLFile& mdlFile;
};
```

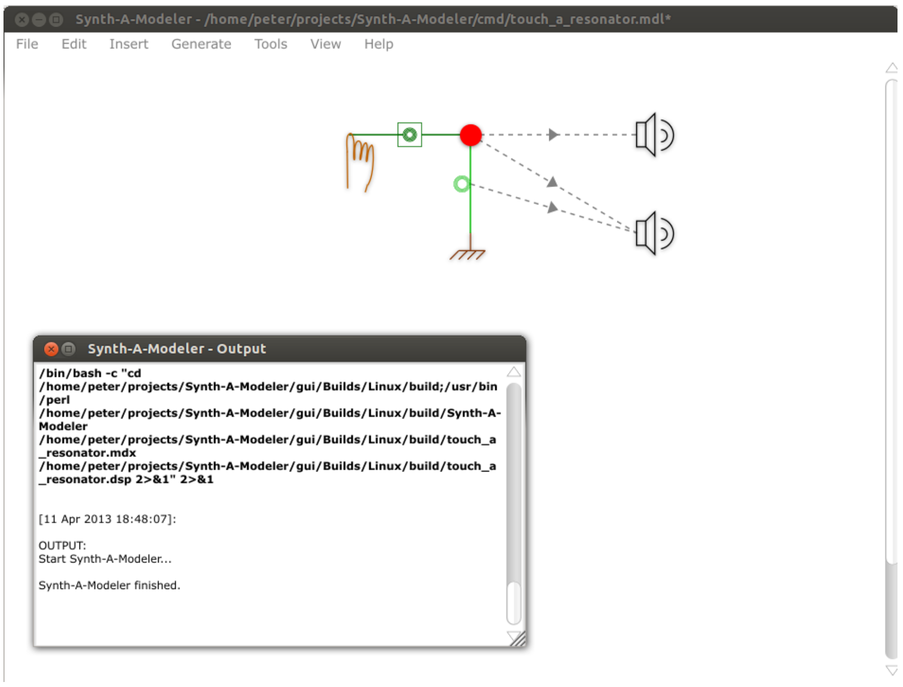
Listing 12: Class for writing an *MDL* file to disk

```
class MDLWriter
{
public:
    MDLWriter(MDLFile& mdlFile_);

    bool writeMDL(const File& saveFile);
    String getMDLString();
private:
    MDLFile& mdlFile;
};
```

Both classes get an `MDLFile` reference in the constructor, which is assigned to an instance variable. Passing a reference ensures that the `MDLFile` instance is valid and not a null pointer. The only public functions these classes provide, is a `parseMDL()`, which takes the input file as argument and `writeMDL()`, which takes the output file respectively. The `MDLWriter` has also a private function `getMDLString()`, which actually generates the string that will be written to a file. Parsing an *MDL* file is straight forward, by iterating its content line by line and applying regular expressions to recognize the type of objects. Using *JUCE*'s `ValueTree`, as mentioned in section 6.1.2, the internal structure of the *MDL* gets assembled within the `parseMDL()` function. Lines consisting the # sign and blank lines, will be skipped. There is one exception, however, when a line starts with two # signs. This is the keyword for a comment object. It is to mention that the *MDL* file is only iterated once and the type of object is identified by testing a line sequentially against all possible types. When writing an *MDL* file to disk, the file gets generated every time from scratch and the original file, stored on the hard drive, gets overwritten. A better solution would be to only update the parts that has been changed. This, however, would involve much more logic and it has been decided to postpone the implementation of this feature to a future release.

6.1.4 *Views*

This section will discuss the graphical components of the *Synth-A-Modeler Designer*. Most of the user interface is accomplished with the following classes and objects. A screenshot of the user interface is shown in fig. 16.



Synth-A-Modeler - /home/peter/projects/Synth-A-Modeler/cmd/touch_a_resonator.mdl*
File
Edit
Insert
Generate
Tools
View
Help
Synth-A-Modeler - Output

```
/bin/bash -c "cd
/home/peter/projects/Synth-A-Modeler/gui/Builds/Linux/build;/usr/bin
/perl
/home/peter/projects/Synth-A-Modeler/gui/Builds/Linux/build/Synth-A-
Modeler
/home/peter/projects/Synth-A-Modeler/gui/Builds/Linux/build/touch_a
_resonator.mdx
/home/peter/projects/Synth-A-Modeler/gui/Builds/Linux/build/touch_a
_resonator.dsp 2>&1" 2>&1


[11 Apr 2013 18:48:07]:

OUTPUT:
Start Synth-A-Modeler...

Synth-A-Modeler finished.
```

Figure 16: *Synth-A-Modeler Designer* user interface

To be able to use *JUCE*'s extensive GUI functionality, all user interface components have to subclass some of these classes. To be precise, all components that are part of the user interface in *Synth-A-Modeler Designer* are subclasses of the `Component` class, which is the base class for all *JUCE* interface objects. The *JUCE* `Component` class provides all basic functionality for GUI components. It manages the component hierarchy, visibility properties, position and bounds of components, mouse interaction and the functionality for painting itself on the screen. A full description with all functions can be viewed in the online documentation [31]. As discussed in section 6.1.1, the `MainAppWindow` class holds the content component — the main GUI component, which is implemented in the `ContentComp` class. The latter is a container for the `ObjectsHolder` class, which holds all components representing physical elements. It also implements the zoom functionality of the editing canvas by wrapping it with the `Viewport` class, which consists of the `ObjectsHolder` class.

Another important role of `ContentComp` is to build all menubar entries, that are specific to editing. As previously stated, `ObjectsHolder` represents the model editing canvas. Besides holding all object components, it also provides a `LassoComponent`, a component that acts as a rectangular selection region, which the user drags with the mouse to select groups of objects [31]. The selectable objects are collected in a container data structure of the type `SelectedItemSet`. To be able to get added to a `SelectedItemSet`, the objects have to subclass `SelectableItem`, which provides an `isSelected` function, that sets the selection state. `ObjectsHolder` subclasses `LassoSource` to provide the functions for finding out which items are within the lasso and to change the list of selected items, by implementing the two pure virtual functions `findLassoItemsInArea()` and `getLassoSelection()`.

The `LassoComponent` works the following way. In `ObjectsHolder`'s `mouseDown` or `mouseDrag` event, the `beginLasso()` method is called. By passing it a suitable `LassoSource` object, such as the `ObjectsHolder` class, it can use it to find out which items are in the active area. When `ObjectsHolderreceives` a `mouseDrag` event, `lassoDragged()` has to be called to update the lasso's position. The `LassoSource` will calculate and update the current selection. When the drag finishes and the mouse button is released, `endLasso()` should be called, which makes the lasso rectangle invisible. `ObjectsHolder` is also responsible for displaying a visual grid for helping to align objects, dispatch commands comming from the user interface to the objects controller and to initiate automatic redrawing of the objects. Automatic redrawing will be discussed in detail in section 6.2.

The main actors on the editing canvas are the object components, representing the physical elements from a model specification file. The base class for object components is the class `BaseObjectComponent`, which is subclassed by all physical objects. Its purpose is to provide the connection to the controller and therefore to the actual data of the model. This is done by storing a reference to `ObjController`. It also implements the context menu, which gets displayed when clicking on an object with the right mouse button. All object components are also subclasses of `SelectableObject`, which provides the interface for the use with the `LassoComponent`. The relation between component classes can be seen in fig. 17. A more detailed overview about the class hierarchy is shown in fig. 38 on page 102.

By the time of writing, there are four types of object components:

- `ObjectComponent`
- `LinkComponent`
- `AudioOutComponent`
- `CommentComponent`

These classes implement individual functionality for specific element types, because not all objects have the same properties, from the perspective of a GUI component. The `ObjectComponent` is a subclass of `BaseObjectComponent` and is the class that is used for all objects that can stand alone, such as mass-like objects. For example, links can

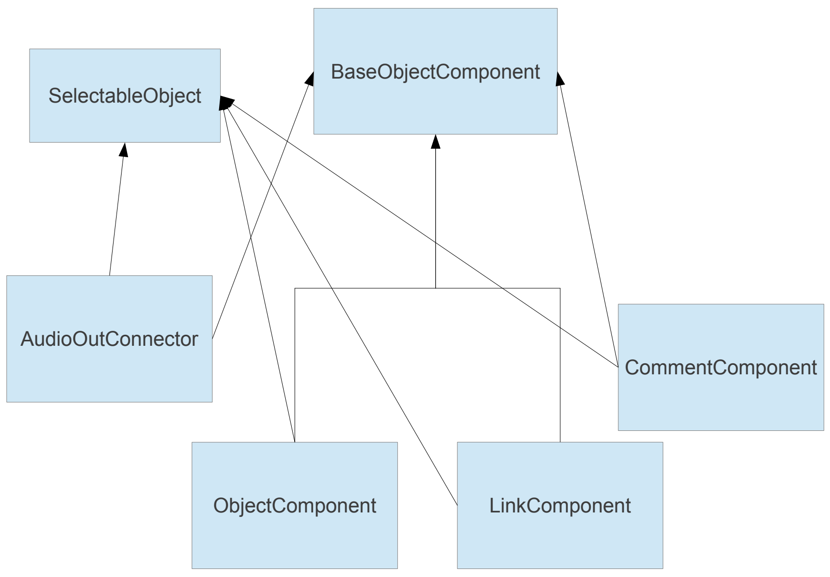
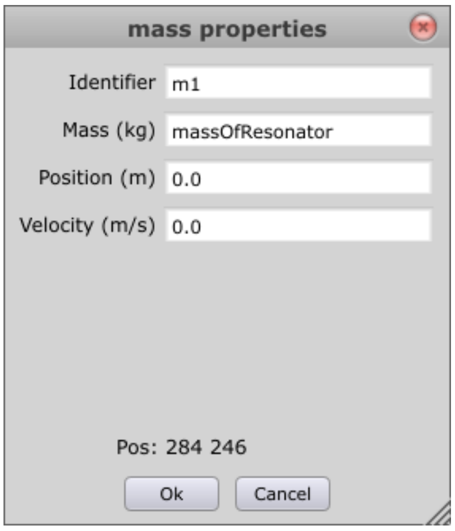


Figure 17: Object component inheritance

not stand alone. They always have to connect two objects. The user interaction to add a link-like object is to select two mass-like objects and then to use the add link command to make the connection. It is no possible to have a detached link on the editing canvas. When removing an object that is an `ObjectComponent`, the attached links have to be removed also, which is the reason that an `ObjectComponent` keeps track of connected `LinkComponents`, stored as pointers in an `Array<LinkComponent*>` field. In contrast to a `LinkComponent`, the `ObjectComponent` has to maintain its position, whereas the `LinkComponent`'s position depends on the objects that it connects together. On the other hand, a `LinkComponent` has to be aware of the connected objects, and therefore it has two `ObjectComponent*` pointer fields, `startObj` and `endObj`. The `CommentObject` can not have connections. It is floating on the editing canvas and it is supposed to be like a label in the physical model. It enables the user to add comments or descriptions to the physical model. The idea has been derived from other applications, like *PureData* and *Max/MSP*, which implement the same kind of object. This feature is also very handy for documenting objects and it is used for help patches. The `AudioConnectorComponent` is a bit different to the object types mentioned before, because it does not appear in an *MDL* file. Its appearance is defined by an `ObjectComponent` that represents an `audioout` in the *MDL* file. The `AudioConnectorComponent` does not derive from `BaseObjectComponent` and has no data attached to it. When the user creates an audio connection, the data of the attached `audioout` objects get modified.

All objects share some identical features, however, they implement these differently, for example when the user double clicks an object. All objects except the `AudioOutConnector` and the `CommentComponent` open a `PropertiesPanel` window for editing the object parameters as shown in fig. 18. The `PropertiesPanel` is a small window containing



Identifier: m1
Mass (kg): massOfResonator
Position (m): 0.0
Velocity (m/s): 0.0
Pos: 284 246

Figure 18: Window for editing object parameters

all parameters for an object. The entries can be changed by modifying the values in the `TextEditor` boxes. The content component of the `PropertiesPanel` is an individual subclass of the abstract class `ObjectPropertiesComponent`. It consist of all basic functionality that all subclasses share, such as applying and canceling changes and all callback function for the text editors. It also provides two pure virtual functions, `readValues()` and `writeValues()`, that the subclasses have to implement. The individual implementations for the different object types set up also all individual GUI elements, such the number of text fields and labels for the parameters. All modifications made in the `PropertiesPanel` are managed by the *MDL*'s `UndoManager` and hence they are undoable and redoable. Furthermore, the text fields provide basic corrections when the user entered a parameter value that is not formatted properly. Listing 13 shows the function for correcting parameter values.

Listing 13: Function for fixing parameter value if not formatted properly

```
String Utils::fixParameterValueIfNeeded(const String& paramVal)
{
        if( paramVal == String::empty)
                return "0.0";

        String tmpVal;
    StringArray operators;
    StringArray params;
    if(paramVal.containsAnyOf("*+-/"))
    {
        String tmp = "";
        for (int i = 0; i < paramVal.length(); ++i)
        {
            if(paramVal[i] == '*' || paramVal[i] == '+'
                || paramVal[i] == '-' || paramVal[i] == '/')
            {
                String op = "";
                op << paramVal[i];
                operators.add(op);
                params.add(tmp);
                tmp = "";
            }
            else
            {
                tmp << paramVal[i];
            }
        }
        if(tmp.compare("") != 0)
            params.add(tmp);

        for (int i = 0; i < params.size(); ++i)
        {
            tmpVal << params[i];
            if(params[i].containsOnly("0123456789")
               && params[i].indexOf(".") == -1)
                tmpVal << ".0";
            if(i < operators.size())
                tmpVal << operators[i];
        }
    }
    else
    {
        tmpVal = paramVal;
        if (tmpVal.containsOnly("0123456789")
            && tmpVal.indexOf(".") == -1)
            tmpVal << ".0";
    }

        return tmpVal;
}
```

The function's purpose is to check whether the entered parameter has a floating point notation with the decimal point. It checks also combinations of variables and values for correct formatting. Double clicking an `AudioConnectorComponent` opens a small window with a text field for setting the gain value for that connection. Again, when setting this value, the result gets assigned to the gain value of the connected `audioout` object. Double clicking on an `CommentComponent` has no effect. Its only settings are the font size and color and the displayed text. To change font properties, the context menu has an entry for changing the text color, which opens a tooltip window with a `ColourSelector` object, as shown in fig. 19.

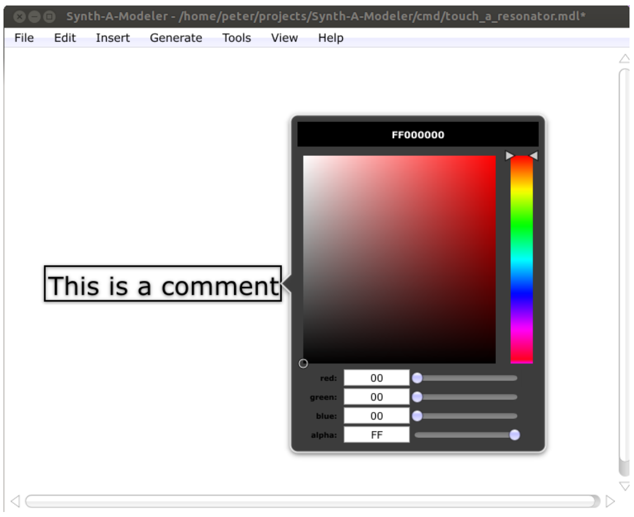
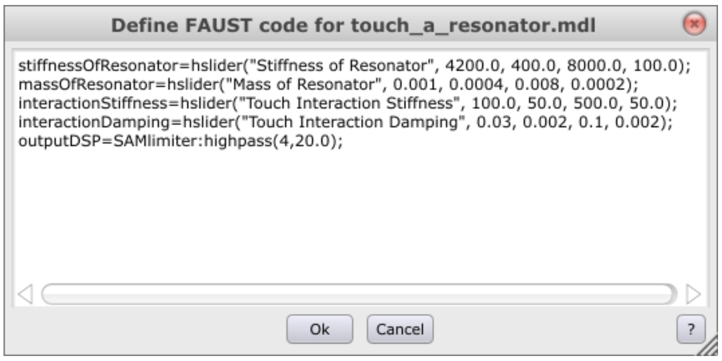

Synth-A-Modeler - /home/peter/projects/Synth-A-Modeler/cmd/touch_a_resonator.mdl*
File Edit Insert Generate Tools View Help
This is a comment
FF000000
red: 00
green: 00
blue: 00
alpha: FF

Figure 19: Setting font color of a `CommentComponent`

The next user interface element is the `FaustcodePanel`, which can be used for entering and editing raw *FAUST* code. *FAUST* code is represented in the model specification file as lines beginning with the following text: "`faustcode:`". The panel consist of a `TextEdit` component, which is a basic text editor. The text will be saved in the *MDL* exactly as it is displayed in the `FaustcodePanel`, and the user is able to perform common text editing functionality such as cut, copy and paste.



```
stiffnessOfResonator=hslider("Stiffness of Resonator", 4200.0, 400.0, 8000.0, 100.0);
massOfResonator=hslider("Mass of Resonator", 0.001, 0.0004, 0.008, 0.0002);
interactionStiffness=hslider("Touch Interaction Stiffness", 100.0, 50.0, 500.0, 50.0);
interactionDamping=hslider("Touch Interaction Damping", 0.03, 0.002, 0.1, 0.002);
outputDSP=SAMlimiter:highpass(4,20.0);
```

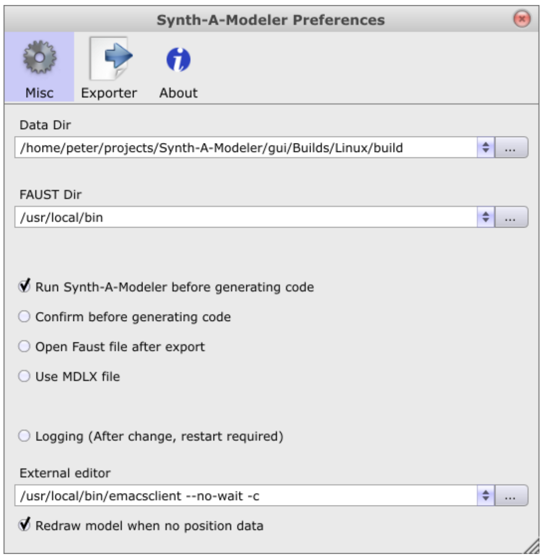
Figure 20: *FAUST* code input and edit window

The preferences panel provides access to the application settings and exporter commands. As previously described in section 6.1.1, all values that are available for modification in the settings window, are stored on hard disk. Its content component is a tabbed pane with the following three tabs:

- Misc
- Exporter

- About

The Misc tab provides various important settings such as the *Data Directory* and *Faust executable* paths as well as custom settings for the interaction with the application. The former will be discussed in section 6.1.6. A screenshot is shown in fig. 21.



Data Dir
/home/peter/projects/Synth-A-Modeler/gui/Builds/Linux/build

FAUST Dir
/usr/local/bin

- [✓] Run Synth-A-Modeler before generating code
- [ ] Confirm before generating code
- [ ] Open Faust file after export
- [ ] Use MDLX file
- [ ] Logging (After change, restart required)

External editor
/usr/local/bin/emacsclient --no-wait -c

- [✓] Redraw model when no position data

Figure 21: Misc tab in preferences window

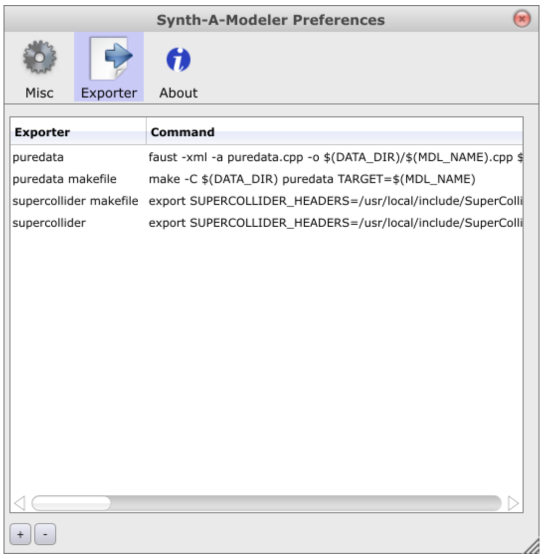
The Exporter panel provides a two column table, which lists all exporter commands and will be described in detail in section 6.1.6. Here the user can specify commands for exporting models to various binary formats such as externals. One of these commands is set in the menu bar as default commands and gets executed when the user runs the generate binary command via keyboard shortcut or menu bar. Figure 22 shows the commands list in the preferences window.

6.1.5 *Controllers*

One of the three key components within the MVC pattern is the controller. As discussed in section 4.2, the controller serves as the bridge between the model and the view. The *Synth-A-Modeler Designer* has two controller classes, `MDLController` and `ObjController`.

6.1.5.1 *MDLController*

`MDLController` handles all actions that are related to the `MDLFile`. Every *MDL* file has one instance of a `MDLController` and the `MDLFile` instance is created within this class with a `ScopedPointer`, which helps to manage allocated memory by automatically deleting the object when closing the file. This way the `MDLFile` is tightly bound to the controller. It also provides functions for opening, saving and closing an *MDL* file. For instance, when the user clicks on the menubar



Exporter	Command
puredata	faust -xml -a puredata.cpp -o $(DATA_DIR)/$(MDL_NAME).cpp $
puredata makefile	make -C $(DATA_DIR) puredata TARGET=$(MDL_NAME)
supercollider makefile	export SUPERCOLLIDER_HEADERS=/usr/local/include/SuperColli
supercollider	export SUPERCOLLIDER_HEADERS=/usr/local/include/SuperColli

Figure 22: Exporter tab in preferences window

entry for opening a new *MDL* file, the click event gets delegated from the view class to the `MDLController`, where the actual opening of an existing file gets performed. Another domain of this controller is the management of the exporting functionality of *Synth-A-Modeler Designer*. Therefore an instance of the class `SAMCmd`, which implements the functionality for executing external commands, is stored here. The `SAMCmd` class will be discussed later in section 6.1.6 and so no further details will be mentioned here. All commands for exporting the *MDL* file from the menubar or keyboard command are delegated to the `MDLController` where the relevant functions `generateFaust()` and `generateExternal()` are called.

6.1.5.2 *ObjController*

The second controller, which manages all actions related to visual objects is the `ObjController`. Visual objects are the graphical representation of the objects in an *MDL* file. Additionally `ObjController` and is one of most prominent classes within *Synth-A-Modeler Designer*, consisting of the complete logic for controlling the visual objects and view components. Therefore `ObjController` keeps track of all objects by storing them in `OwnedArrays`, a data structure which takes ownership of the inserted elements. When removing an element from an `OwnedArray` it also can be deleted automatically. The behaviour is controlled with a boolean function parameter. This is quite practical for memory management. Once an object is created with the keyword `new` and inserted into an `OwnedArray`, the only step needed to remove the element from the array is to delete the object and free its memory allocation. The following arrays are member variables in `ObjController`, which store all of the graphical objects of a model:

- `OwnedArray<ObjectComponent> objects`
- `OwnedArray<LinkComponent> objects`
- `OwnedArray<AudioOutConnector> objects`
- `OwnedArray<CommentComponent> objects`

Another important member variable is the list of selected objects, which is stored in the variable `SelectedItemSet<SelectedObject*> sObjects`. Every time the user selects a physical object on the editing canvas with the lasso tool or by clicking on an object, it gets added to `sObjects`, and if the object becomes deselected, it is removed from the set.

An additional important functionality of `ObjController` is the handling of object actions, such as adding and removing objects or changing their positions. Therefore, all types of objects (`ObjectComponent`, `LinkComponent`, `AudioOutComponent` and `CommentComponent`) have a set of functions to provide this functionality. For `ObjectComponent`, these functions are:

-
  ```
  void addNewObject(ObjectsHolder* holder,
      ValueTree objValues);
  ```

-
  ```
  ObjectComponent* addObject(ObjectsHolder* holder,
      ValueTree objValues, int index, bool undoable);
  ```

-
  ```
  void removeObject(ObjectComponent* objComp,
      bool undoable, ObjectsHolder* holder);
  ```

The other three object types have similar function declarations but the function parameters and implementations differ somewhat. The `addNewObject()` function gets called when the user adds a new object with the user interface, so within the function there is only a call to `addObject()` and a call to `sObjects.selectOnly()`. The latter sets the newly added object as the only selected object. The call to `addNewObject()` is also ab undoable action. `addObject()` is more complex. It performs all steps to register the new object within the application. When the last argument, `undoable` is `true`, an `UndoableAction` gets created in order to perform the adding of an object and to make the action undoable. The `AddObjectAction`'s perform function calls the `addObject()` function in turn, but this time with `undoable` set to `false`. Then the code in the `else` part gets called, which is actually the code for adding the object. Listing 14 shows the implementation of `addObject()`.

Listing 14: Function for adding a new object

```
ObjectComponent* ObjController::addObject(
    ObjectsHolder* holder, ValueTree objValues,
    int index, bool undoable)
{
```

```
    if(undoable)
    {
        AddObjectAction* action = new AddObjectAction(
            this, objValues, holder);
        owner.getUndoManager()->perform(
            action, "Add new Object");

        return objects[action->indexAdded];
    }
    else
    {
        const Identifier& groupName = Utils::getObjectGroup(
            objValues.getType().toString());
        ValueTree mdl = owner.getMDLTree();
        ValueTree subTree = mdl.getOrCreateChildWithName(
            groupName, nullptr);

                subTree.addChild(objValues,-1, nullptr);
        idMgr->addId(objValues.getType(),
                     objValues[Ids::identifier].toString(),
                     nullptr);

        ObjectComponent* objComp = new ObjectComponent(
            *this, objValues);
        objects.insert(index, objComp);

        holder->addAndMakeVisible(objComp);
        holder->updateComponents();
        changed();
        return objComp;
    }
}
```

`removeObject()` is pretty similar to `addObject()`, but with the functionality to remove an object and unregister it within the application. It also consists of the same conditional to check whether the removal is undoable or not. Additionally it checks also if the object to remove has connected links and audio connections. If there are connections, it deletes them before removing the object. As already mentioned, the other three object types have also these functions, but they will not be described in detail, because they are principally the same and the only difference is the code that is related to how these objects are created.

Changing the position of objects is also implemented in `ObjController`, by providing the functions `startDragging()`, `stopDragging()`, `dragSelectedComps()` and `moveSelectedComps()`. These functions get called when the user drags objects or moves their position with the arrow keys. It handles not only changing positions of a single object, but also manages setting position of multiple selected objects. The breakdown for changing the position of objects, into these functions is important, because all steps need different implementation. `startDragging()` stores the start position of objects, `dragSelectedComps` performs the actual position change by calling the objects' `setPosition()` function and `stopDragging` completes via the `UndoManager` transaction. The latter is needed to have one undoable transaction for the end position and the start position but not for the intermediate positions between. The object's `setPosition()` function is quite similar to `addObject()` by the means of having a conditional, which checks whether setting the position is undoable or not. If it is undoable, an `UndoableAction` is created to perform the position change by calling the `setPosition()` function again.

`ObjController` implements cut, copy and paste functionality. *JUCE*'s `ValueTree` is a major help to perform these actions. It provides a function to get the objects as XML string, `createXml`. When calling the function `copySelectionToClipboard`, the selected objects' XML representation gets copied to the system clipboard as a `String` with a special XML Tag, `SAMOBJECTS`. Listing 15 show a clipboard entry that contains a mass object.

Listing 15: Representation of a mass object when copied to the clipboard

```
<SAMOBJECTS>
  <mass posX="304" posY="200" identifier="m1">
    <parameters>
      <parameter value="massOfResonator"/>
      <parameter value="0.0"/>
      <parameter value="0.0"/>
    </parameters>
  </mass>
</SAMOBJECTS>
```

When the user pastes the previously copied content, the `paste()` function looks for the special tag in the system clipboard. If the special tag is present, it recreates the `ValueTree` with help of the function `fromXml()`.

An additional functionality in `ObjController` is the `tidyUp()` function. As the function name suggests it tidies up the objects by aligning the objects horizontally or vertically depending on the positions of the objects. The implementation is based on the function with the same name in *PureData*.

The *Synth-A-Modeler Designer* offers also a more advanced feature to automatically set the position of objects and will be discussed in section 6.2. It uses a graph based algorithm and is mentioned here because all references to the objects are stored in the `ObjController` and creating the graph is done by calling the function `makeGraph`, which assembles a directed graph from the objects.

6.1.6 *Executing external commands*

One of the key features of the *Synth-A-Modeler Designer* is to generate *FAUST* source code and other binary externals from model specification files. This is done by executing external programs, such as the *Synth-A-Modeler Compiler* and the *FAUST* executable or exporter scripts provided by *FAUST*. This section will describe how this is implemented in *Synth-A-Modeler Designer*.

First of all, there are two actions the user can perform from the "Generate" menu in the menu bar or with keyboard shortcuts:

- Generic Faust code
- Binary

The first calls the *Synth-A-Modeler Compiler*, the second calls whatever command is specified in the preferences under the "Exporter" tab (see section 6.1.4). When the user performs one of these actions, it gets sent from the `MainAppWindow` to the `MDLController` functions `generateFaust()` or `generateExternal()`. Then, these two functions call other functions from within the `SAMCmd` class, which implements all functionality regarding external commands. Before discussing the `SAMCmd` class, the *Data Dir*, which has previously been mentioned in section 6.1.4, has to be elaborated upon. The fact that that *Synth-A-Modeler Designer* works with external commands implies that problems regarding installation paths and working directories of these programs can occur. For example, when running the *Synth-A-Modeler Compiler*, it has to be assured that `physicalmodeling.lib` is installed within the same directory, or else the compiler cannot import it properly. This would result in the compiler not being able to create the output. To work around this issue, it has been decided to introduce a directory called the *Data Dir*, where the compiler and all other mandatory scripts and libraries are installed and which is also the output directory for the compilation results. The user can set this directory via the preferences panel.

The `SAMCmd`, whose class declaration is shown in listing 16, is a very small class.

Listing 16: `SAMCmd` class

```
class SAMCmd
{
public:
        SAMCmd();
        ~SAMCmd();

        bool isSynthAModelerCmdAvailable();
        bool isSAMpreprocessorCmdAvailable();
        bool isCmdAvailable(const String& cmdStr);
        bool isPerlAvailable();
        bool isFaustAvailable();

        const String generateFaustCode(const String& inPath,
                                       const String& outPath,
                                       bool useSamConsole = true);

        const String generateExternal(const String& mdlPath,
                                      const String& exporter,
                                      bool useSamConsole = true);

private:
    const String runPerlScript(const String& script,
                               const String& inPath,
                               const String& outPath,
                               bool useSamConsole);
};
```

It has only functions for checking the existence of the needed external programs and functions for running these.

`generateFaustCode()` runs the two *Perl* scripts `SAM-preprocessor`, and `Synth-A-Modeler` and the output of this function is the *FAUST* source code containing the current model. The function parameters are self explanatory input and output paths. The third argument is used to switch the output of the scripts to the `OutputWindow` on or off. When running the GUI it is set to `true`, but when running the application from the command-line (see section 6.1.7) or in unit tests (see section 6.1.8), the graphical components are not available and thus, it has to be prevented from calling the `OutputWindow` functions. Internally `generateFaustCode()` calls the function `generatePerlScipt()`, which is marked as private because it is intended to be used only within `SAMCmd`.

`generateExternal()` is used to generate the external binaries. Its parameters are the path to the *MDL* file and a `String exporter`, containing the command to run. The function is very generic and basically it is possible to run all kinds of commands, not only the *FAUST* executable.

Both `generateExternal()` and `generateFaustCode()` use an internal function that starts a child process from within the application, by using architecture specific functions.

In section 3.4, commands for generating externals were discussed and presented. The only difference of the previously mentioned commands to the ones used in the *Synth-A-Modeler Designer* is that in the case of the *Synth-A-Modeler Designer*, placeholder variables can be employed. These are `DATA_DIR`, `MDL_NAME` and `FAUST_DIR`. When the user specifies them within the command, they will be replaced by the actual value. This makes the commands independent from the user's settings and installation paths. The names are self explanatory, `DATA_DIR` gets changed to the current *Data Dir*, `MDL_NAME` is the file name of the current *MDL* file without the file extension and `FAUST_DIR` is the location of the *FAUST* executable. Listing 15 shows the default exporter commands that are provided by the *Synth-A-Modeler Designer*

Listing 17: Default exporter commands

```
<?xml version="1.0" encoding="UTF-8"?>

<PROPERTIES>
  <VALUE name="puredata" val="faust -xml -a puredata.cpp -o $(
      DATA_DIR)/$(MDL_NAME).cpp $(DATA_DIR)/$(MDL_NAME).dsp;mkdir
       -p $(DATA_DIR)/puredatadir; g++ -DPD -fPIC -Wall -O3 -
      mfpmath=sse -msse -msse2 -msse3 -ffast-math -shared -Dmydsp
      =$(MDL_NAME) -I/usr/include/pdextended -o $(DATA_DIR)/
      puredatadir/$(MDL_NAME)~.pd_linux $(DATA_DIR)/$(MDL_NAME).
      cpp; faust2pd -r 10 -s $(DATA_DIR)/$(MDL_NAME).dsp.xml; mv
      -f $(DATA_DIR)/$(MDL_NAME).pd $(DATA_DIR)/puredatadir"/>
  <VALUE name="puredata makefile" val="make -C $(DATA_DIR)
      puredata SAMTARGET=$(MDL_NAME)"/>
</PROPERTIES>
```

An exporter is defined by its name, which is displayed in the "Generate" menu bar entry and its value, which is the actual command that get executed. The first exporter command in listing 17, might look very complicated but after careful observation, it can be seen that it consist only commands already presented in this thesis (see section 3.4) and other commands that move the output to a special folder. A collection of other commands can be viewed and edited on the project's Wiki page on *Github*:

`https://github.com/ptrv/Synth-A-Modeler/wiki/Exporters`

6.1.7 *Command-line support*

Generating the desired output or external binary with the *Synth-A-Modeler Designer* or the *Synth-A-Modeler Compiler* involves many steps. For example, when the user wants to create a *PureData* external from a model specification file without opening the *Synth-A-Modeler Designer*, the steps for that would be

- Run `SAM-preprocessor` script to generate intermediate `.mdx` file from the `.mdl`
- Run `Synth-A-Modeler` compiler script to generate `.dsp` *FAUST* source file
- Run commands to generate the desired external binary

Of course the user can use one of the Makefiles that are shipped with the *Synth-A-Modeler Designer* package. There are several Makefiles for different output external. However, to provide a unified user experience and to minimize the amount of different tools and scripts the user has to use in order to get to the "final product", *Synth-A-Modeler Designer* offers a command-line interface to aggregate all these commands. Listing 18 shows the output, when running *Synth-A-Modeler Designer* from the command-line with the `--help` flag.

Listing 18: Help text of the *Synth-A-Modeler Designer* command-line interface

```
./Synth-A-Modeler-Designer --help

JUCE v2.0.37
Synth-A-Modeler!

Usage:

 Synth-A-Modeler --compile /path/to/mdl_file /path/to/dsp_file
    Compiles a mdl_file to a dsp_file.

 Synth-A-Modeler --binary exporter_name mdl_file
    Generates a binary.

 Synth-A-Modeler --list-exporters
    Lists all available exporter commands.

 Synth-A-Modeler --list-exportersd
    Lists all available exporter commands with detail.

 Synth-A-Modeler --print-xml /path/to/mdl_file
    Prints xml structure of mdl_file to stdout.

 Synth-A-Modeler --clean
    Cleans DATA_DIR.

 Synth-A-Modeler --version
    Prints version information.
```

Besides compiling a model to *FAUST* code, the user also can generate binary externals, using one of the specified exporter commands. The possible exporter commands are set in the *Synth-A-Modeler Designer* preferences, as already described in section 6.1.4 and section 6.1.6. It is also possible to list all available exporters in short and detailed format or to clean up the *Data Dir*. i. e. removing all temporary files.

6.1.8 *Unit testing*

During the development of the *Synth-A-Modeler Designer* the format of the model specification file changed several times and therefore the `MDLParser` and `MDLWriter` classes had to adopt these changes in order to provide the desired functionality and to work properly. Thus a unit test suite has been developed in parallel for testing critical code. Unit testing is a procedure in software development, which tests different units of the source code to decide if they work as expected [32]. Of course, unit tests are not implemented for all components of the *Synth-A-Modeler Designer*, since GUI components and user interactions cannot be tested without the actual user that performs the actions.[51] However there are a few test for some critical parts of the software. The test suite provides tests for `MDLFile`, `MDLParser`, `MDLWriter` and the `SAMCmd`. Every class to test has a counterpart test class, which runs functions to compare the output of the class to test with an expected value. In the case of the development of the *Synth-A-Modeler Designer*, when the model specification format changed, running the test resulted in failure. Depending on the amount of the tests, the parts that did not work correctly after the changes has been made, could be detected easily. After fixing the code, the tests have been run to verify the new implementation. When the test succeeded, the new code worked as expected, or when it failed, further adjustments had to be made. This had to be repeated until all test succeeded. This was an easy and efficient way to find and debug erroneous code.

Fortunately, *JUCE* provides classes for unit testing. The two key classes are the `UnitTest` class and the `UnitTestRunner` class. The former is the base class that has to be subclassed in order to implement a unit test, and the latter is a helper class that runs a set of unit tests. The test classes are included into the *Synth-A-Modeler Designer*'s source code, but are only compiled into the application if it is compiled with debug symbols. This way release builds are not affected and the application binary size can be reduced. To run the tests, the special keyword `--test` has to be passed to the application when executing. It starts and exits the unit tests before all GUI components get initialized.

6.2 AUTOMATIC POSITIONING OF OBJECTS

One of the requirements for the *Synth-A-Modeler Designer* was to implement a strategy for automatic positioning of objects, which means that the objects should have their position on the editing canvas assigned automatically. This can be useful in several situations. For example, some *Synth-A-Modeler* models were created before the *Synth-A-Modeler Designer* has developed and before having the possibility to edit model specification file visually, the user would have had to edit the source file of a model manually in a text editor. However, position data was introduced with the *Synth-A-modeler Designer*, so early models did not have position data. Loading old models caused the *Synth-A-Modeler Designer* to place all the objects at the origin, and the user had to reposition each objects manually.

51 A simulator could be implemented to simulate the user, but this is too extensive for the scope of this work.

Another consideration was the integration of the visual representation of the physical objects into the creative process and therefore to provide the user with the possibility to reorganize and restructure the model visually. For these reasons, the functionality to automatically reposition the objects has been introduced in *Synth-A-Modeler Designer* by implementing a force-directed algorithm that acts on the graph of objects to align their position. This section will describe the algorithm, how it was implemented in the *Synth-A-Modeler Designer* and will show some example models and their visual representations when applying the algorithm.

6.2.1 *Force-directed algorithm*

One of the most versatile algorithms for the calculation of graph layouts consisting of simple undirected graphs are according Tamassia force-directed algorithms, also known as spring embedders Tamassia [33]. Tamassia writes that these algorithms calculate the graph layout only using information from the graph structure itself, instead of using domain-specific information. Tamassia goes on to argue that the results are aesthetically pleasing, very symmetrical and have crossing-free layouts Tamassia [33].

As the name already suggests, force-directed algorithms assign forces to the edges and nodes of a graph. The algorithm of Eades from 1984 and the algorithm of Fruchtermann and Reingold, use spring forces, such as in Hooke's Law to simulate the attraction between the nodes [33]. However, if the forces were only attractive, then the objects would tend to dump up over one another. Hence, repulsive forces are also implemented, similar to those of electrically charged particles, acting on the nodes and are based on Coulomb's law.

Forced-based algorithms, works best when applied to small graphs, while using them with large graphs result in poor performance. According Tamassia, the scalability of force-based algorithms is affected by two main reasons. The first is that physical models have typically a lot of local minima[52] and even with sophisticated improvements of the algorithms in order to avoid them, do not result in good layouts. The second reason for bad scalability of large graphs is caused by resolution problems, because the vertex separation in this case is very small and therefore can lead to unreadable graphs Tamassia [33].

The force-based algorithm applies two forces: Coulomb's law (see (3)) and Hooke's law (see (4)) [34].

$$F_{ij}^{C} = \beta \frac{\vec{x_i} - \vec{x_j}}{\|\vec{x_i} - \vec{x_j}\|^3} \tag{3}$$

52 These are multiple solutions

$$S_{ij} = \frac{1}{2}k(\|\vec{x_i} - \vec{x_j}\| - d_{ij})^2 \qquad (4)$$

The repulsive force (3) is based on the physical equivalent of electrical forces and is similar to the condition when the nodes would have an electrical charge. The idea is that we add a force F^C_{ij} to the nodes that is inversely proportional to the square of the distance between the nodes. The attractive force (4) that is caused by the spring is zero if the distance of two nodes is the rest length d of the spring and grows by the square of its elongation or compression.

Listing 19 shows pseudo code for a basic force-based algorithm, which is the same that has been implemented in *Synth-A-Modeler Designer*.

Listing 19: Pseudo code of the force-based graph drawing algorithm

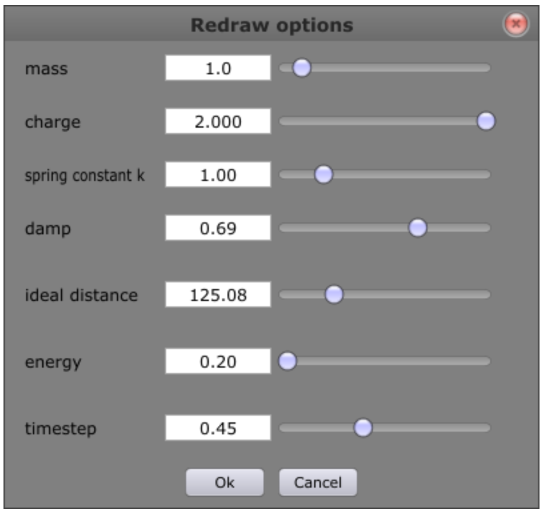
```
set up initial node velocities to (0,0)
set up initial node positions randomly
loop
    total_kinetic_energy := 0
    for each node
        net-force := (0, 0)

        for each other node
            net-force := net-force + Coulomb_repulsion( this_
                node, other_node )
        next node

        for each spring connected to this node
            net-force := net-force + Hooke_attraction( this_node
                , spring )
        next spring

        this_node.velocity := (this_node.velocity + timestep *
            net-force) * damping
        this_node.position := this_node.position + timestep *
            this_node.velocity
        total_kinetic_energy := total_kinetic_energy + this_node
            .mass * (this_node.velocity)^2
    next node
until total_kinetic_energy is less than some small number
```

The first step, when claculating a graph layout, is to set all velocities to zero and give all nodes a random position. Random position provides a good distribution of the nodes. Then, all nodes of the graph have to be iterated. The total force that is acting on one node is calculated by applying Coulomb's law on the current and each other node and to add up the resulting forces. Furthermore the spring attraction has to be calculated with all connected nodes and added to the total force. Then the velocity can be calculated for the current node by multiplying the total force with the timestep and adding it to the current velocity. One crucial parameter is a damping value that has to be multiplied with the valocity. Without damping, the algorithm would never stop. The new position can then be calculated by multiplying the new velocity with the timestep and adding it to the old position. The total kinetic energy is the sum of all individual node forces and is used to stop the calculation if it is below a defined value.

6.2.2 *Graph drawing components*

Although the algorithm used in the *Synth-A-Modeler Designer* is quite simple, the focus was not to implement a perfect graph drawing algorithm, but to provide an architecture, that allows the application to be easily extended with new drawing algorithms. The architecture that served as model for the implementation of the graph drawing components in the *Synth-A-Modeler Designer* is a simple graph visualization application for the *Processing* language, implemented by Kamermans [35]. It consists of a `Node` class that all objects have to subclass and which manages the objects' positions and the connections to other nodes by keeping track of incoming and outgoing nodes. Another component is the `DirectedGraph` class, which manages the graph structure and the logic for changing its layout. Functions such as `addNodes()` and `linkNodes()` are used to build the graph structure. The function `setFlowAlgorithm()` takes a pointer to a `FlowAlgorithm` object, which is an abstract class with its only function `reflow()`. The latter function has to be implemented by all graph drawing algorithms in order to be able to be used in `DirectedGraph` as such. After the graph has been built, the function `reflow()` in `DirectedGraph` has to be called to initiate the calculation of the new graph layout. Because the recalculation of the graph is an iterative process, the `reflow()` function has to be called several times, until it returns `true` to indicate that the calculation is done. The redrawing process can be started from the menu bar entry *Edit* → *Redraw*. The command gets dispatched in the `ObjectsHolder` class to the `redrawObjects()` function, which takes a `CommandID` argument with the type of redraw algorithm. Here, the first step is to create the graph by creating a new instance of `DirectedGraph` object and calling `makeGraph()` on the `ObjController`. The next step is to set the `FlowAlgorithm` and finally to start the timer function, which calls the algorithm's `reflow()` function repeatably, until it returns true, to indicate that the recalculation is done.

Due to the sensibility of the parameter settings of the force-based algorithms, the calculated layouts can vary quite much and sometimes it is necessary to adjust the parameters of the algorithms. Therefore, a window for setting the force-based algorithm parameters has been implemented to enable the possibility for the user to experiment with different parameter configurations. Figure 23 shows the window with the different parameters among others, such as mass, spring constant and damping.



mass	1.0
charge	2.000
spring constant k	1.00
damp	0.69
ideal distance	125.08
energy	0.20
timestep	0.45

Figure 23: Window for setting redraw options

6.2.3 *Graph drawing examples*

At the time of writing, *Synth-A-Modeler Designer* consists of two algorithms for redrawing the graph: `CircleFlowAlgorithm` and `ForceBasedAlgorithm`. The former is a simple algorithm that repositions all mass-like objects into a circular form and the latter is a force-based algorithm as discussed in section 6.2.1. A simple unordered model as shown

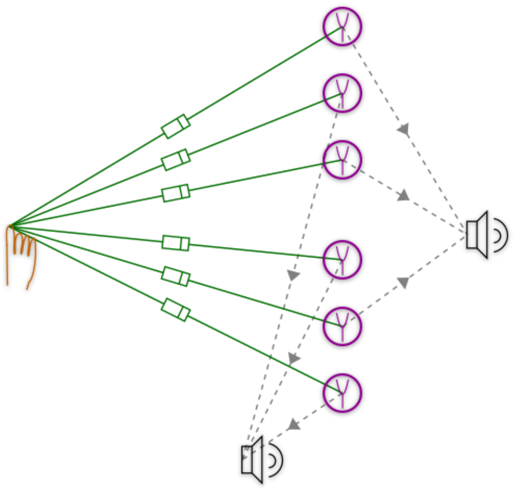


Figure 24: An unordered `guiro` model

in fig. 24, changes its form after recalculating its layout with the `CircleFlowAlgorithm` to a circle, as shown in fig. 25. Figure 26 shows the `guiro` model after applying the `ForceBasedAlgorithm`.

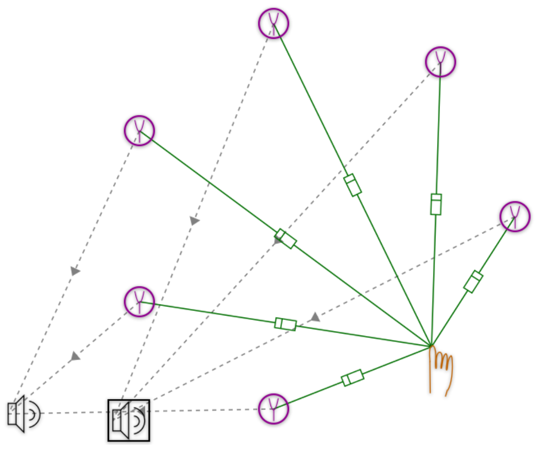


Figure 25: The `guiro` model after applying `CircleFlowAlgorithm`

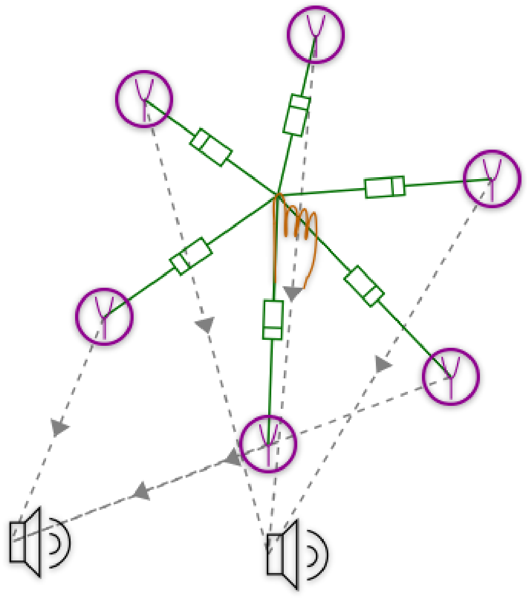
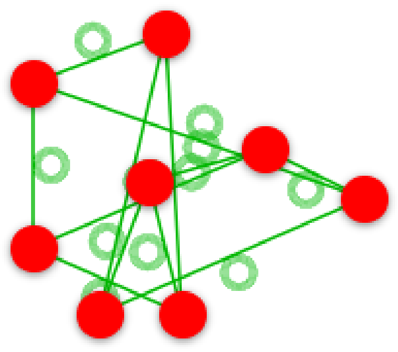


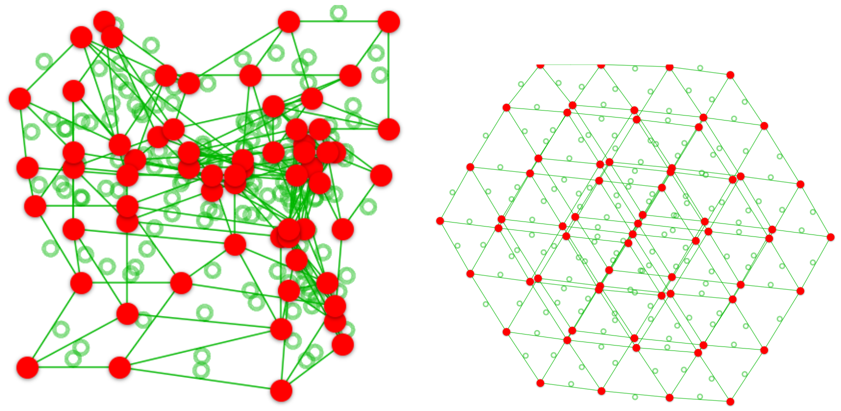
Figure 26: The `guiro` model after applying `ForceBasedAlgorithm`

The following images (see fig. 27, fig. 28, fig. 29, fig. 30, fig. 31 and fig. 32) show models before and after their layout has been recalculated by the force-based algorithm. As can be seen in figs. 26 to 29 and 32 the algorithm clearly has the ability to reveal some structures in the models to the user. It should be emphasized that, due to the random initial positions, the final positions can vary widely. However, this can be seen as a feature — the model designer may wish to search through some possible ways of visualizing a model in order to arrive at a visually pleasing result.



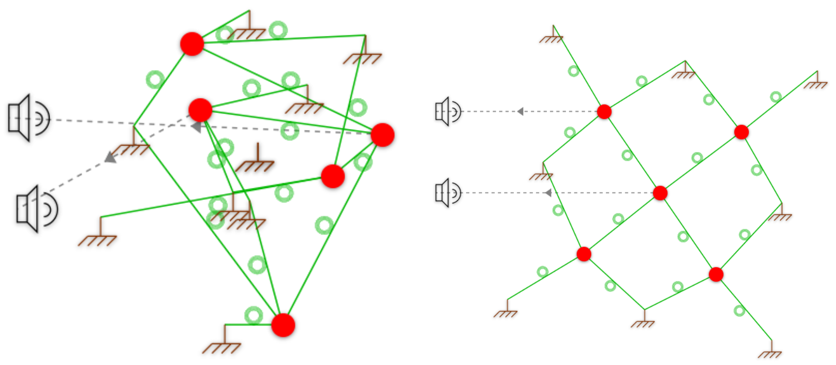
(a) Before (b) After

Figure 27: Cube model layout comparison



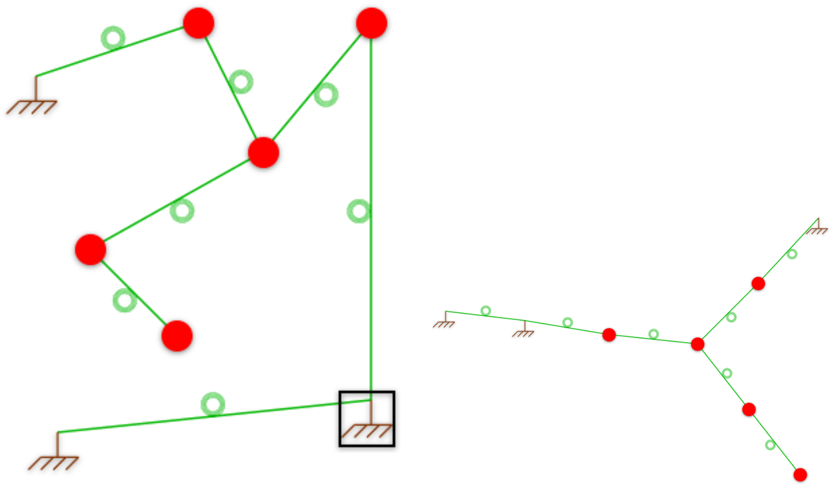
(a) Before (b) After

Figure 28: Big cube model layout comparison



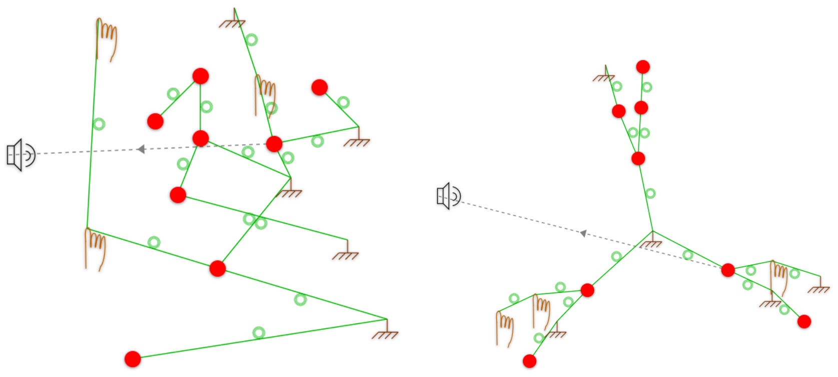
(a) Before (b) After

Figure 29: Percussion model layout comparison



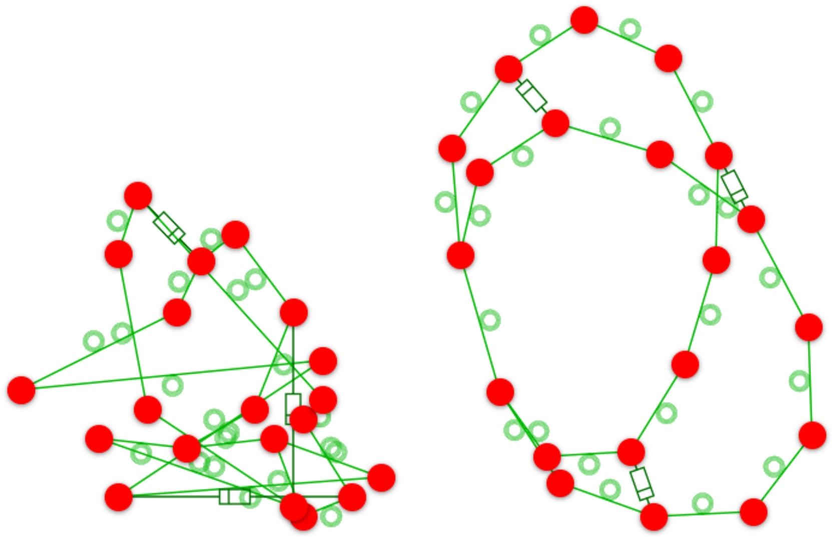
(a) Before (b) After

Figure 30: Random mass interaction model layout comparison



(a) Before (b) After

Figure 31: Another random mass interaction model layout comparison



(a) Before (b) After

Figure 32: Random model layout comparison

6.3 EXTENDING SYNTH-A-MODELER DESIGNER WITH A NEW OBJECT

New objects can be added to the *Synth-A-Modeler Designer*. This section demonstrates the steps, which are needed to extend *Synth-A-Modeler Designer* with a new type of object and should serve as manual for further extensions. The explicit source code lines will be shown as listings and new code will be marked as such with inline comments wrapped around it. The beginning tag is `// begin` and the close tag is `// end`.

The first step is to define the type of the new object, e. g. to define if it is a mass-like object, link-like object or a new type of object that has not been defined yet. The first two types have already been implemented and therefore templates already exist showing how to implement a new one. If the new type is an object type that is different from any other existing object, the steps needed to implement it will be different. In *Synth-A-Modeler Designer* there are the `AudioConnectorComponent` and the `CommentComponent`, which do not match the implementation of the common object types, such as mass-like and link-like objects.

However, the new object that will be implemented within the context of this section, is a link-like object: the *pulsetouch* object. Figure 33 shows the icon, which is almost identical to the regular *touch* icon, except the purple background color. Now that the object type is

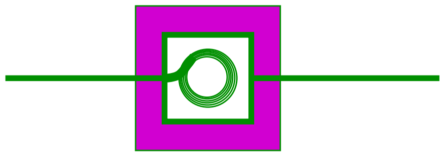


Figure 33: Icon of the pulsetouch object

defined, the object needs to be created. First an Id has to be defined in `Models/ObjectIDs.h`, by adding `DECLARE_ID (pulsetouch);` in the namespace `Ids`. Next the functions for creating the `pulsetouch` entity should be defined in the source code file `Models/ObjectFactory.cpp`. When `createNewLinkObjectTree()` gets called with `Ids::pulsetouch`, then `createNewPulsetouchObject()` gets called and returns a new `pulsetouch` `ValueTree`. Listing 20 show the new code in context.

Listing 20: New code in `ObjectFactory.cpp`

```
// ...

// begin
static ValueTree createNewPulsetouchTree(const String& newName,
                                         const String&
                                             startObject,
                                         const String& endObject)
```

```
{
    StoredSettings& settings = *StoredSettings::getInstance();
        ValueTree newTree(Ids::pulsetouch);
    StringArray p;
    p.add(settings.getDefaultValue("pulsetouch_stiffness", "100.0
        "));
    p.add(settings.getDefaultValue("pulsetouch_damping", "0.1"));
    p.add(settings.getDefaultValue("pulsetouch_offset", "0.0"));
    p.add(settings.getDefaultValue("pulsetouch_pulsemult", "0.0")
        );
    p.add(settings.getDefaultValue("pulsetouch_pulsetau", "0.0"))
        ;
    p.add(settings.getDefaultValue("pulsetouch_pulselen", "0.0"))
        ;
    ValueTree paramsTree = ObjectFactory::createParamsTree(p);
    newTree.addChild(paramsTree, -1, nullptr);
    newTree.setProperty(Ids::identifier, newName, nullptr);
    newTree.setProperty(Ids::startVertex, startObject, nullptr);
        newTree.setProperty(Ids::endVertex, endObject, nullptr);

        return newTree;
}
// end

// ...

ValueTree ObjectFactory::createNewLinkObjectTree(const Identifier
    & linkType,
                                    const String& newName,
                                    const String& startObject,
                                    const String& endObject)
{
    if(linkType == Ids::link)
        return createNewLinkTree(newName, startObject, endObject)
            ;
    else if(linkType == Ids::touch)
                return createNewTouchTree(newName, startObject,
                    endObject);
    // begin
    else if(linkType == Ids::pulsetouch)
        return createNewPulsetouchTree(newName, startObject,
            endObject);
    // end
    else if(linkType == Ids::pluck)
                return createNewPluckTree(newName, startObject,
                    endObject);
        else if(linkType == Ids::waveguide)
                return createNewWaveguideTree(newName,
                    startObject, endObject);
    else
                return ValueTree::invalid;
}
```

Listing 21 shows the new code in `Views/ObjectsHolder.cpp` that gets executed when the user creates a new object, either from the menu bar or via context menu.

Listing 21: New code in `ObjectsHolder.cpp`

```
bool ObjectsHolder::dispatchMenuItemClick(
    const ApplicationCommandTarget::InvocationInfo& info)
{
    // ...
    switch (info.commandID)
    {
    // begin
    case CommandIDs::insertPulsetouch:
        objController.addNewLinkIfPossible(this,
            ObjectFactory::createNewLinkObjectTree(Ids::
                pulsetouch,
                objController.getNewNameForObject(Ids::pulsetouch
                    ),
                                                   startObj,
                                                       endObj));
        break;
    // end
    }
}

// ...

void ObjectsHolder::showLinkPopupMenu(String so, String eo)
{
    PopupMenu m;
    m.addSectionHeader("Add...");
    m.addItem (1, "Linear Link");
    m.addItem (2, "Touch Link");
    // begin
    m.addItem (3, "Pulsetouch Link");
    // end
    m.addItem (4, "Pluck Link");
    m.addSeparator();
    m.addItem (5, "Waveguide");
    m.addSeparator();
    m.addItem (6, "Audio Connection");

    // begin
    else if (r == 3)
    {
        objController.addNewLinkIfPossible(
            this, ObjectFactory::createNewLinkObjectTree(
                Ids::pulse, objController.getNewNameForObject(
                    Ids::touch), so, eo));
    }
    // end
}
```

`getNewNameForObject()` is a function in `Utilities/IdManager.h`, which returns a new name for the object and has to be extended with a member variable `SortedSet<String> pulsetouchIds;` and added to `Utilities/IdManager.cpp` at the obvious places. Next, `LinkComponent` has to be extended to be aware of the new object. The constructor gets extended by the assignment of a new color for the `pulsetouch` object and in the function `drawPath()`, the code for drawing the object needs to be added. Because it is quite similar to the `touch` object only a few lines of code have to be added to generate the outline rectangle, This is shown in listing 22.

Listing 22: New code in `LinkComponent.cpp`

```
LinkComponent::LinkComponent(ObjController& owner_, ValueTree
    linkTree)
: BaseObjectComponent(owner_, linkTree),
    lastInputX (0),
    lastInputY (0),
    lastOutputX (0),
    lastOutputY (0),
    segmented(false)
{
    // begin
    else if(data.getType() == Ids::pulsetouch)
    {
        color = Colour(0xff006f00);
    }
    // end
}

void LinkComponent::drawPath(Graphics& g)
{
    // ...

    // begin
    else if(data.getType() == Ids::touch ||
            data.getType() == Ids::pulsetouch)
    {
        // ..

        // begin
        iconPath = ResourceLoader::getInstance()->
            getPathForLinkId(
            data.getType(), 0, 0, iconWidth, iconHeight);
        // end

        // ...

        // begin
        if(data.getType() == Ids::pulsetouch)
        {
            Colour c = currentColor;
            g.setColour(Colours::indigo);

            Path outlineRect;

            outlineRect.addRectangle(0 - (iconWidth * 0.15),
                                     0 - (iconHeight * 0.15),
                                     iconWidth + (iconWidth *
                                         0.3),
                                     iconHeight + (iconHeight *
                                         0.3));

            outlineRect.applyTransform(
                AffineTransform::translation((-iconWidth/2), -
                    iconHeight/2));
            outlineRect.applyTransform(
                AffineTransform::identity.rotated(rotateVal)
                                    .translated((x1 + x2) * 0.5f,
                                                (y1 + y2) * 0.5f)
                                                    );
            PathStrokeType stroke(6.0f);
            stroke.createStrokedPath(outlineRect, outlineRect);
            g.fillPath(outlineRect);

            g.setColour(c);
        }
        // end
    }
    // end

    // ...
}
```

Listing 23 shows the small addition in `Utilities/ResourceLoader.cpp` to return the icon path for the `touch` when calling it for the `pulsetouch` object.

Listing 23: New code in `ResourceLoader.cpp`

```
Path ResourceLoader::getPathForLinkId(const Identifier& linkId,
                                      float x, float y, float w,
                                          float h)
{
    // begin
    else if(linkId == Ids::touch
        || linkId == Ids::pulsetouch)
    // end
    {
        return getPathForTouch(x, y, w, h);
    }
}
```

The next step is to implement the remaining parts in the graphical user interface, which are the menu entry and the editing properties window. To show `pulsetouch` within the menu bar, additions have to be made in `Application/Application.cpp`, `Application/CommandIDs.h` and `View/ContentComponent.cpp`, mainly by adding new menu bar entry code and the new `CommandID`. Listing 24, listing 25 and listing 26 show these changes.

Listing 24: New code in `Application.cpp`

```
PopupMenu SynthAModelerApplication::MainMenuModel::
    getMenuForIndex (
    int topLevelMenuIndex, const String& /*menuName*/)
{
    // ...
    else if (topLevelMenuIndex == 2)
    {
        // ...
        // begin
        menu.addCommandItem(commandManager, CommandIDs::
            insertPulsetouch);
        // end
        // ...
    }
    // ...
}
```

Listing 25: New code in `CommandIDs.cpp`

```
namespace CommandIDs
{
    // ...
    // begin
    static const int insertPulsetouch       = 0x2030b3;
    // end
    // ...
}
```

Listing 26: New code in `ContentComponent.cpp`

```
void ContentComp::getAllCommands(Array <CommandID>& commands)
{
    const CommandID ids[] = {
        // ...
        // begin
        CommandIDs::insertPulsetouch,
        // end
        //...
    };
}
```

```
void ContentComp::getCommandInfo(CommandID commandID,
    ApplicationCommandInfo& result)
{
    switch (commandID)
    {
    // ...
    // begin
    case CommandIDs::insertPulsetouch:
        result.setInfo("Pulsetouch Link", "", CommandCategories::
            inserting, 0);
        break;
    // end
    // ...
    }
}
```

The inclusion of the `pulsetouch` object into `ObjectPropertiesPanel` involves more changes, because it has to support all individual object types and their properties. The amount of the changes depends on the type of the new object. The more features it has, the more changes and additions are needed in `ObjectPropertiesPanel`. The `pulsetouch` is a link-like object and it has three additional parameters compared to the regular `touch` object: the pulse multiplier, the pulse tau and the pulse length. Listing 27 shows the new code that has to be added in `LinkPropertiesComponent` in order to support the new `pulsetouch` object, which is basically extending the class with three additional `Label` and `TextEdit` components and adding code to write and read the additional parameters.

Listing 27: New code in `ObjectPropertiesPanel.cpp`

```
class LinkPropertiesComponent : public ObjectPropertiesComponent
    {
public:
    LinkPropertiesComponent(ObjectPropertiesPanel* op_,
                            ObjController* objController_,
                            ValueTree data_,
                            UndoManager* undoManager_)
        : ObjectPropertiesComponent(op_, objController_, data_,
            undoManager_),
      // begin
      laPulseMult("laPulseMult", "Pulse multiplier"),
      tePulseMult("tePulseMult"),
      laPulseTau("laPulseTau", "Pulse tau"),
      tePulseTau("tePulseTau"),
      laPulseLen("laPulseLen", "Pulse length"),
      tePulseLen("tePulseLen")
      // end
    {
        // begin
        else if (data.getType() == Ids::pulsetouch)
        {
```

```
        tePulseMult.addListener(this);
        addAndMakeVisible(&tePulseMult);
        laPulseMult.attachToComponent(&tePulseMult, true);
        tePulseTau.addListener(this);
        addAndMakeVisible(&tePulseTau);
        laPulseTau.attachToComponent(&tePulseTau, true);
        tePulseLen.addListener(this);
        addAndMakeVisible(&tePulseLen);
        laPulseLen.attachToComponent(&tePulseLen, true);

    }
    // end
    }
    void resized()
    {
    // begin
    else if (data.getType() == Ids::pulsetouch)
    {
        tePulseMult.setBounds(100, 100, getWidth() - 110, 22)
            ;
        tePulseTau.setBounds(100, 130, getWidth() - 110, 22);
        tePulseLen.setBounds(100, 190, getWidth() - 110, 22);
        offset = 30;
    }
    // end
    }

    void readValues()
    {
    // begin
    else if(data.getType() == Ids::pulsetouch)
    {
        tePulseMult.setText(data.getChildWithName(Ids::
            parameters).getChild(3)[Ids::value].toString());
        tePulseTau.setText(data.getChildWithName(Ids::
            parameters).getChild(4)[Ids::value].toString());
        tePulseLen.setText(data.getChildWithName(Ids::
            parameters).getChild(5)[Ids::value].toString());
    }
    // end
    }

    bool writeValues()
    {
    // begin
    ValueTree pa5, pa6;
    //end

    // begin
    else if(data.getType() == Ids::pulsetouch)
    {
        pa3.setProperty(Ids::value,
                                Utils::fixParameterValueIfNeeded(
                                    tePos.getText()),
                                undoManager);
                pa4.setProperty(Ids::value,
                                Utils::fixParameterValueIfNeeded(
                                    tePulseMult.getText()),
                                undoManager);
                pa5 = paramsTree.getChild(4);
                pa5.setProperty(Ids::value,
                                Utils::fixParameterValueIfNeeded(
                                    tePulseTau.getText()),
                                undoManager);
                pa6 = paramsTree.getChild(5);
                pa6.setProperty(Ids::value,
                                Utils::fixParameterValueIfNeeded(
                                    tePulseLen.getText()),
                                undoManager);
            }
            // end
            }

private:
    // begin
    Label laPulseMult;
    TextEditor tePulseMult;
    Label laPulseTau;
    TextEditor tePulseTau;
    Label laPulseLen;
    TextEditor tePulseLen;
    // end
};
```

An additional important change is also in `ObjectComponent.cpp`. In order to make the `pulsetouch` connectable with mass-like objects, new code has to be added in `canBeConnected()` function, shown in listing 28.

Listing 28: New code in `ObjectComponent.cpp`

```
bool ObjectComponent::canBeConnected(const Identifier& objId)
{
    // begin
    if(objId == Ids::link || objId == Ids::touch ||
        objId == Ids::pluck || objId == Ids::pulsetouch)
        return canBeConnectedToLinks();
    // end
}
```

Now reading and writing of *MDL* files have to support the new object type. The additional code is shown in listing 29, listing 30 and listing 31.

Listing 29: New code in `MDLParser.cpp`

```
bool MDLParser::parseMDL(const File& f)
{
    // ...
        for (int i = 0; i < lines.size(); ++i) {
        // ...
        else if(re.fullMatch(SAMRegex::getLinkLine(), line))
        {
            // begin
            else if (values[0].compare("pulsetouch") == 0)
            {
                linkTree = ValueTree(Ids::pulsetouch);
            }
            // end

            // ...

            // begin
            if(linkTree.getType() == Ids::pluck ||
                linkTree.getType() == Ids::pulsetouch)
                numParams = 4;
            // end
        }
        // ...
    }
}
```

Listing 30: New code in `SAMRegex.cpp`

```
const char* SAMRegex::link = "(link|pluck|touch|pulsetouch)";
```

Listing 31: New code in `MiscUtilities.cpp`

```
const Identifier& Utils::getObjectGroup(const Identifier& ident)
{
    // begin
    else if(ident == Ids::link || ident == Ids::touch
            || ident == Ids::pluck || ident == Ids::pulsetouch)
        return Objects::links;
    // end
}
```

The last addition is to add new default values for the new object in `BinaryData/default_values.xml`, as shown in listing 32 and extend `Application/SAMLookAndFeel.cpp` for the desired appearance of the `pulsetouch` objects' menu item color as shown in listing 33.

Listing 32: New default values

```
<?xml version="1.0" encoding="UTF-8"?>
<PROPERTIES>
  <!-- begin -->
  <VALUE name="pulsetouch_stiffness" val="100.0"/>
  <VALUE name="pulsetouch_damping" val="0.1"/>
  <VALUE name="pulsetouch_offset" val="0.0"/>
  <VALUE name="pulsetouch_pulsefreq" val="0.0"/>
  <!-- end -->
</PROPERTIES>
```

Listing 33: New values in `SAMLookAndFeel.cpp`

```
static const Colour menuColourInsertPulsetouch(0xff006f00);

void SAMLookAndFeel::drawPopupMenuItem(Graphics& g,
                                       int width, int height,
                                       const bool isSeparator,
                                       const bool isActive,
                                       const bool isHighlighted,
                                       const bool isTicked,
                                       const bool hasSubMenu,
                                       const String& text,
                                       const String&
                                           shortcutKeyText,
                                       Image* image,
                                       const Colour * const
                                           textColourToUse)
{
    // ...
    // begin
    else if(text.compare("Pulsetouch Link") == 0)
        textColour = menuColourInsertPulsetouch;
    // end
    //...
}
```

The necessary steps to include a new mass-like object would be quite similar to the ones that are required for a link-like object and in most cases new objects are either mass-like or link-like. In the case of a custom object type, more source code has to be added in order to fully implement it. The parser and reader class has to get support for the new type and `ObjController` would have to be extended with function that deal with adding, removing or changing position of an element and also building up the graphical components would need to get support for a new type. Additional `UndoableAction` classes for the new object type would also be required. However, there are four types of objects implemented in the *Synth-A-Modeler Designer*, which can be used as a guidance for creating new objects.

6.4 PROBLEMS AND LIMITATIONS

Although the current state of *Synth-Modeler Designer* fulfills all requirements, there are also limitations. A significant limitation is the redraw algorithm, which does not work reliably with large models. The reason is the nature of the algorithm, which does not scale well for large graphs, and has been discussed in section 6.2.1. With small models the algorithm produces good results but as soon as the number of objects grows, several issues occur. The first is the redraw animation. It gets slow when the model consists of many objects. The second issue is that the model grows very large in space with big models, in which case the user has to modify the parameters of the algorithm to improve the result. Another issue of the redrawing components in *Synth-Modeler Designer* is that the redraw algorithm only currently works for models that are contiguous. Models that consist of multiple pieces can not be redrawn yet. This is a result of the graph building algorithm used in the application, which is capable to build only one graph out of one model. In the future, this issue could be solved by first partitioning models into submodels of connected components. Then, each submodel would be randomly oriented about its origins. Finally, the redraw algorithm could be reapplied independently to each submodel.

Another limitation in the *Synth-A-Modeler Designer* is the graphical rendering. Although the implementation of the rendering engine works well for small and medium sized models, performance goes down for large models, which contain thousands of objects. By performance, the author means here the interaction of the user with the objects. Dragging objects around with the mouse can be slow with large models. The main cause is here of course that the rendering is done in software and could be improved by transferring the calculations on to the graphics card, which however would involve major changes in the codebase.

7 MODELING THE OUD

7.1 THE OUD

It was decided to create the first published physical model of the oud using *Synth-A-Modeler Designer*. The difficulty of designing non-western instruments with physical modeling is not exactly a technical one. Finding more information on non-western instruments is an obstacle which makes it harder to design them. This section will give a brief historical and technical overview of the instrument.

7.1.1 *History*

The oud is a plucked, string instrument with a pear-shaped body made of wood, originating from the Eastern regions [36]. A similar instrument in the western world is the Lute, which is very much influenced by the Arabic oud. In Europe the lute was very popular starting from medieval times to the 18th century. The Arabic oud was first introduced into Europe during the Moorian occupation of Spain (711–1492). The Arabic oud was and is played with a plectrum, which has been adopted by European players in the beginning. First evidence of a Moorish oud is shown on paintings from the 9th and 10th century. Famous players, like *Ziryab* came to the court of Andalusian emir *Abd al Rahman II* (822–852).

Goode [37] writes about the myth around the invention of the Arabic oud,

> [T]he oud was invented by Lamak, a descendant of the biblical Cain. When his son died, Lamak is said to have hung his remains in a tree and seen in the skeleton the bowled body and elegant neck.

The oldest evidence of it, dates back to Southern Mesopotamia (modern Iraq), over 5000 years ago on a seal found by *Dr. Dominique Collon* and is nowadays kept at the British Museum. With little differences, this instrument and close family members are found in all ancient Middle East civilizations as part of their music culture. Nowadays there are slightly different versions of the traditional Arabic oud in Turkey, Greece and Armenia. They have different tunings and also a different tone compared to the oud use in the Arab world.

The neck of the oud is made of light wood and has no frets, which makes it for the player easier to play glissandi and vibrato and allows microtones to be played, as in the Arabic *maqam*[53] system [38]. Figure 34 shows an oud. As clearly visible in fig. 34, the neck is bent

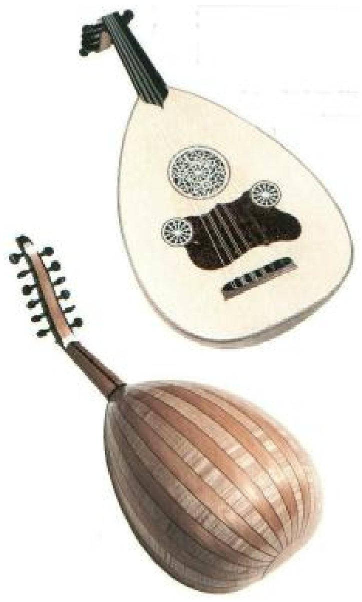


Figure 34: Oud front and rear view[54]

backward almost 90°. Although there are different string arrangements, in most cases there are five pairs of strings, tuned in unison, and a single bass string. Historically the strings were made from gut or in combination with metal. Nowadays instrument builders use also nylon strings but gut strings are still available for use. However, gut strings are more authentic, and they tend to have irregularities in pitch when the humidity level changes. Nylon strings stay in tune more reliably but of course with a slightly different sound, which might not be as authentic [36].

7.1.2 *Principle of operation*

From the perspective of the principle of operation, the oud, or generally a lute, is according Rossing the same as guitar-type instruments and is a system of coupled vibrators [39]. Figure 35 shows a simple schematic of a guitar which can be applied also to lutes and the oud. Rossing states, that plucked strings store energy, however they only emit a small amount of sound [39]. Instead, they transmit the energy, caused by the vibration, onto the top plate and bridge. These vibrations then are transported to the back plates, ribs and the air cavity. Thus, sound is radiated by the vibrating plates and the sound hole. As fig. 35 shows, at low frequencies, acoustic sound radiates from the

53 Maqam is a set of notes with traditions that define relationships between them, habitual patterns, and their melodic development.
54 Source: http://www.oud.net/acoustic.htm

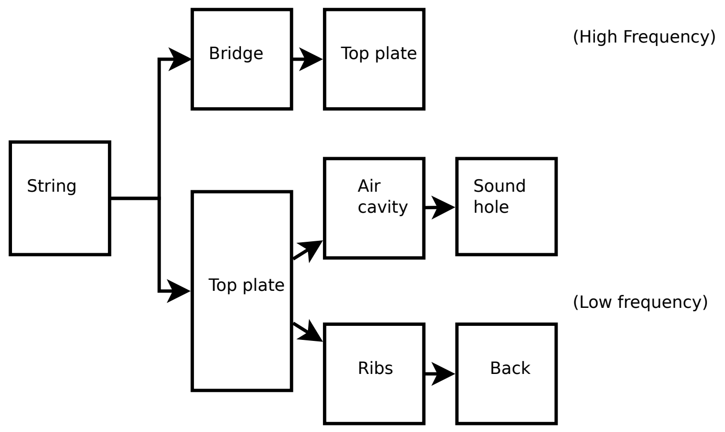


Figure 35: Simple schematic of a guitar [39]

top and back plates as well as from the sound hole. In contrast, at high frequencies, nearly all of the sound radiates from the top plate.

7.2 MODELING THE OUD IN SYNTH-A-MODELER

This section will go through the process of modeling an oud with the *Synth-A-Modeler Designer*. The goal is to create a model for the instrument as authentic as possible. While still resulting in a relatively simple model, which could be edited and customized by future users or even transmuted into other instruments or hypothetical configurations. This is the general *Synth-A-Modeler* philosophy. The sound of an instrument is also influenced by its body, which modulates the sound of the strings and is also responsible for non-linear parts of the spectrum. For that reason this section will only describe how to model the strings of an oud.

7.2.1 *The modeling*

The first step is to create a new empty model and to add the objects that represent a string. As described in section 7.1.1, the oud has five string pairs which are tuned in unison. One string can be modeled with two `waveguide` objects connected together with a `junction` object and on each other free sides a termination. The junction is the point where a link can be connected, which makes it possible to model the plucking or touching of the string. The other two free ends, which are connected to `termination` objects represent the fixed points of the string, which is in the case of an oud, the neck and the bridge. Figure 36 shows the string in *Synth-A-Modeler Designer*. The string

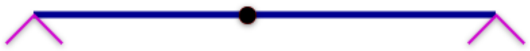


Figure 36: A string in *Synth-A-Modeler*

has to be copied multiple times according to the number of strings on the instrument. After adding all the 10 strings to the editing canvas, a `port` object has to be added, which then has to be connected with the 10 junctions via a `pluck` link object. This will be the connection to the "outside", a port to a external controller, representing the plucking of the strings. To be able to listen to the sound that this model generates, two `audioout` objects will be added, and will be connected to the terminations. Now, the physical model for the oud is set up and can be seen in fig. 37.

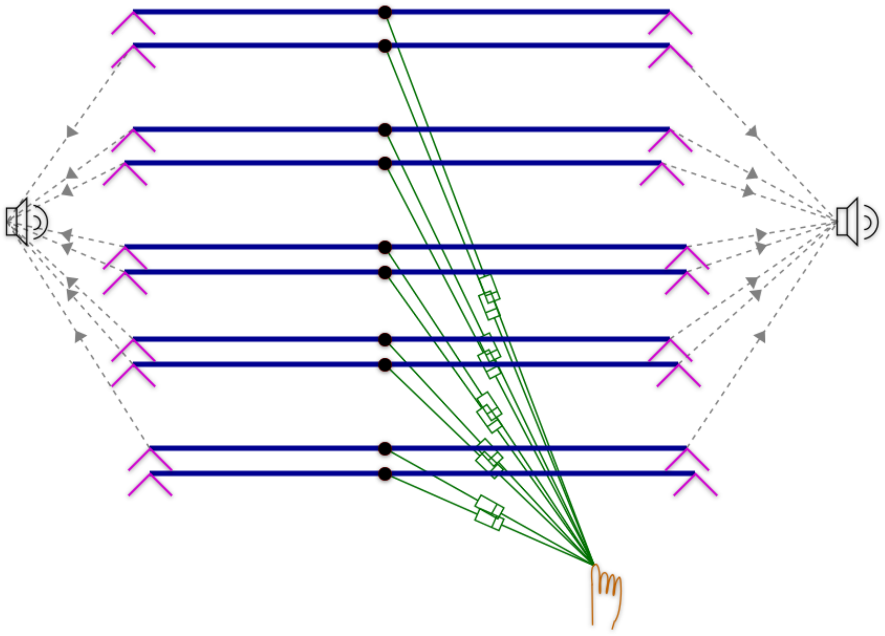


Figure 37: The oud model in *Synth-A-Modeler*

Next the parameters of the objects have to be adjusted. The spatial alignment of the strings can be set by specifying the displacement parameter of the `junctions` in meters. From the lowest string to the highest the distance is around 5 cm and the distance has to be split up to the strings. The lowest pair of strings has a displacement of 0 m the second lowest 0.01 m and so on, until it ends at 0.04 m displacement of the highest pair of strings. There has to be also added a slight displacement between the string pairs by a couple of millimeters. To define the pitch of the strings, the `waveguides` have to be modified respectively. Its parameters are the characteristic wave impedance in $\mathrm{N\,m^{-1}\,s^{-1}}$, which is set to $2.5\,\mathrm{N\,m^{-1}\,s^{-1}}$ and the type of string, which is defined by the maximum time delay in s and the current time delay in s and is set to $0.027\,\mathrm{s}$ and the frequency of the string respectively. To achieve a more natural sound, each pair of strings are slightly mistuned with respect to another to achieve beating, as commonly occurs with real ouds. The beat frequency is the difference in the frequencies of each pair of strings. These beat frequencies of the string pairs are adjusted so that they all differ to produce a richer sound. The source code can be found in the Appendix.

7.2.2 *Commuted Synthesis*

At the moment sound radiation is not implemented in *Synth-A-Modeler*, but it is possible to achieve psychoacoustically similar effects with commuted synthesis [1]. Commuted synthesis however, can be integrated into the physical models by using *FAUST* DSP code which processes the output audio signal. The following basic definition shows an `audioout` object for a `fret` object.

```
audioout,a1,(fret)*vol:outputDSP;
```

The sound of the `fret` object is multiplied by a volume control and passed through `outputDSP`. `outputDSP` is a dynamic range limiter that takes care of the output signal in case of clipping.

It is also possible to add reverberation to the output audio signal, like in the following code:

```
audioout,a1,(fret)*vol:outputDSP:SAMfreereverb;
```

To simulate the body modes of a cello, one can apply a filter with the followin code:

```
audioout,a1,(fret)*vol:outputDSP:SAMfreereverb:bodyResp2;
```

These filter coefficients were fit by Esteban Maestre and are defined as follows in SAM-fx.lib using six two-pole two-zero filters placed in series:

```
bodyResp2 =
tf2np(1.0, 1.5667, 0.3133, -0.5509, -0.3925) :
tf2np(1.0, -1.9537, 0.9542, -1.6357, 0.8697) :
tf2np(1.0, -1.6683, 0.8852, -1.7674, 0.8735) :
tf2np(1.0, -1.8585, 0.9653, -1.8498, 0.9516) :
tf2np(1.0, -1.9299, 0.9621, -1.9354, 0.9590) :
tf2np(1.0, -1.9800, 0.9888, -1.9867, 0.9923);
```

Another option would be, to apply an impulse response from a truncated body with using a filter (also defined in SAM-fx.lib):

```
audioout,a1,(fret)*vol:outputDSP:SAMfreereverb:bodyResp1;
```

Although this method is easier to apply, it is much less efficient to compute.

In the oud model, the output audio is taken from the string terminations at each end (see fig. 37), which is not physically correct but sounds interesting. To produce a rich sound, one channel is filtered using `bodyResp1` and the other channel is filtered using `bodyResp2`.

8 CONCLUSIONS

8.1 CONCLUSION

This thesis has given an insight into the development of a cross-platform application, the *Synth-A-Modeler Designer*, in order to provide a graphical environment to physical modeling. For a long time the mathematical nature of physical modeling provided a large barrier for artist, musicians or students, who were often not able to design own instruments and sounds based on physical models. While there already exist several software systems that try to make physical modeling more accessible to artists, or which provide other than mathematical approaches, the barriers are still very high to use and apply these systems. It is either the cost or the complexity that keeps users away from physical modeling.

With the *Synth-A-Modeler* project, the main focus is on providing an open platform for physical modeling. But moreover, to lower the barrier for artists to start designing physical models in order to create new sounds. The *Synth-A-Modeler Designer* software enables users to create and design physical models visually, and it might also provide a less complex point of entrance to getting started with physical modeling. In an educational context the *Synth-A-Modeler Designer* could be used not only for creating new sounds and instruments, but also for describing physical mass-interaction systems in general. Additionally, designing physical models graphically could also be part of the creative process, which might help to make physical modeling more accessible.

The *Synth-A-Modeler Designer* has been developed with free and open-source software components in order to make it freely available. But furthermore, to also make it easier to explore, extend and develop it further. Some components, such as the automatic redrawing feature, has purposely been implemented with a focus on extension and further exploration.

An early version of *Synth-A-Modeler Designer* has been used successfully in a laboratory session at Stanford University. The students were able to prototype new physical models graphically and they could create models which they had not yet imagined before. Letting the students use the software in its early stage helped also to discover some bugs and errors, which could only be noticed by an extensive use of the software.

8.2 FUTURE WORK

While the *Synth-A-Modeler Designer* is already functional and almost all initial requirements have been implemented, there are some parts that need improvement. During the development process, additional ideas came up, which were not considered at the time of planning and designing.

As already mentioned, the graphical rendering system in the current state is not optimal for large models. Therefore, it would be very important to rewrite the parts of the software that are responsible for the rendering in order to do calculations on the graphics card.

The automatic redraw system is still very basic in its current state and it would benefit very much from improvements. Handling multiple graphs in one model is essential, and this should be a main focus during the expansion process of the application. The redraw performance is not very high in its current state, and improving its performance would provide the user with a better usability experience. However, the performance is influenced by multiple factors, such as the rendering system and the actual redraw algorithm. Improving these would most likely boost its performance.

To help new users to start using the *Synth-A-Modeler Designer*, it would be advisable to improve the help system, as well as add more examples that describe the individual objects. In its current state, the *Synth-A-Modeler Designer* provides basic example models.

In some cases it might be more convenient to edit the *MDL* file manually and to add comments for describing parts of the model. When loading an *MDL* file, the *Synth-A-Modeler Designer* ignores these comments. However, when saving an *MDL* file with the *Synth-A-Modeler Designer*, the file gets completely rewritten and the manually added comments are deleted. A focus of a possible extension should be the improvement of the save functionality in order to alter only the parts that have been edited, instead of rewriting the file completely.

Part IV

APPENDIX

A APPENDIX

A.1 EXAMPLE *.mdl* FILE

Listing 34: A complete mdl file

```
# format [type]([default parameters]),[identifying_name],([
  label1],[label2],...)
# mass is a normal 1-D mass
# port is a port to the external world, for example to a haptic
   device,
# or just for a position
# input and force output.
mass(0.01,0.0,0.0),m1;
mass(1e-2,0.0,0.0),m2;
mass(0.03,0.0,0.0),m3;
port( ),dev1;
# The ground object is like an infinite mass that never moves.
  It always
# stays at the same position. format [type] [unique identifier]
   [label]
# [initial position in m] ground g1 g 0 Link-type objects
  connect from
# one mass-type object to another mass-type object. For that
  reason,
# each of the links must specify the two mass-type objects that
   it
# connects.
#
# format:
# [type]([default parameters]),[identifying_name],[starting_
  vertex],
# [ending_vertex],([label1],[label2],...)
# For instance for a linear link we have link([stiffness in
# N/m],[damping in N/(m/s)],[center position offset in
# m]),[unique_linkname],[starting_vertex],[ending_vertex],([
  label1],[label2],...)
#
# Or for a nonlinear, switching "BUT" contact link, we have
# contact([stiffness in N/m],[damping in N/(m/s)],[offset for
  engagement
# in m]),[unique_linkname],[starting_vertex],[ending_vertex],([
  label1],[label2],...)
#
# Or for a nonlinear plucking link, we have pluck([stiffness in
# N/m],[damping in N/(m/s)],[Minimum displacement difference
  for contact
# in m][offset for engagement in m]),[unique_linkname],[
  starting_vertex],
# [ending_vertex],([label1],[label2],...)
#
# In this model, we have just three simple linear links
link(12000.0,0.003,0.0),l1,m1,m2;
link(12000.0,0.004,0.0),l2,m2,m3;
link(12000.0,0.005,0.0),l3,m2,dev1;
# Then a list of audio-only outputs
# audioout [index of additional audio outlet] [identifier of
  source 1]
# [gain 1] [identifier of source 2] [gain 2] ...
# Here is a stereo audio output example:
audioout,a1,m1*1000.0;
audioout,a2,l2*100.0;
```

A.2 OUD MODEL

Listing 35: The model of an oud

```
# MDL file for Synth-A-Modeler
#
# Edgar Berdahl, 2012
# Audio Communication Group
# Technical University of Berlin
#
#
  -------------------------------------------------------------------

#
# This program is free software; you can redistribute it and/or
   modify
# it under the terms of the GNU General Public License as
  published by
# the Free Software Foundation; either version 2 of the License
  , or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be
  useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty
  of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public
  License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

```
port(),dev0; # pos 627,709
port(),dev1;


pluck(k,R,maxdisp,0.0),p0,junct0,dev0;
pluck(300.0,0.1,0.003,0.0),p1,junct0cpy,dev0;
pluck(300.0,0.1,0.003,0.0),p2,junct0cpy1,dev0;
pluck(300.0,0.1,0.003,0.0),p3,junct0cpy2,dev0;
pluck(300.0,0.1,0.003,0.0),p4,junct0cpy3,dev0;
pluck(300.0,0.1,0.003,0.0),p5,junct0cpy4,dev0;
pluck(300.0,0.1,0.003,0.0),p6,junct0cpy5,dev0;
pluck(300.0,0.1,0.003,0.0),p7,junct0cpy6,dev0;
pluck(300.0,0.1,0.003,0.0),p8,junct0cpy7,dev0;
pluck(300.0,0.1,0.003,0.0),p9,junct0cpy8,dev0;


waveguide(2.5,simpleString(0.027,(1.0/(noteC-1.0))*relPos-
  simpleStringTermDelay(bridgeFc))),wg0,bridge,junct0;
waveguide(2.5,simpleString(0.027,(1.0/(noteC-1.0))*(1.0-relPos)
  -simpleStringTermDelay(fretFc))),wg1,fret,junct0;
waveguide(2.5,simpleString(0.027,(1.0/noteC)*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy,bridgecpy,junct0cpy;
waveguide(2.5,simpleString(0.027,(1.0/noteC)*(1.0-relPos)-
  simpleStringTermDelay(fretFc))),wg1cpy,fretcpy,junct0cpy;
waveguide(2.5,simpleString(0.027,(1.0/(noteG-1.0))*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy1,bridgecpy1,junct0
  cpy1;
waveguide(2.5,simpleString(0.027,(1.0/(noteG-1.0))*(1.0-relPos)
  -simpleStringTermDelay(fretFc))),wg1cpy1,fretcpy1,junct0cpy1;
waveguide(2.5,simpleString(0.027,(1.0/noteG)*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy2,bridgecpy2,junct0
  cpy2;
waveguide(2.5,simpleString(0.027,(1.0/noteG)*(1.0-relPos)-
  simpleStringTermDelay(fretFc))),wg1cpy2,fretcpy2,junct0cpy2;
waveguide(2.5,simpleString(0.027,(1.0/(noteD-1.0))*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy3,bridgecpy3,junct0
  cpy3;
waveguide(2.5,simpleString(0.027,(1.0/(noteD-1.0))*(1.0-relPos)
  -simpleStringTermDelay(fretFc))),wg1cpy3,fretcpy3,junct0cpy3;
waveguide(2.5,simpleString(0.027,(1.0/noteD)*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy4,bridgecpy4,junct0
  cpy4;
waveguide(2.5,simpleString(0.027,(1.0/noteD)*(1.0-relPos)-
  simpleStringTermDelay(fretFc))),wg1cpy4,fretcpy4,junct0cpy4;
waveguide(2.5,simpleString(0.027,(1.0/(noteA-1.0))*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy5,bridgecpy5,junct0
  cpy5;
waveguide(2.5,simpleString(0.027,(1.0/(noteA-1.0))*(1.0-relPos)
  -simpleStringTermDelay(fretFc))),wg1cpy5,fretcpy5,junct0cpy5;
```

```
waveguide(2.5,simpleString(0.027,(1.0/noteA)*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy6,bridgecpy6,junct0
  cpy6;
waveguide(2.5,simpleString(0.027,(1.0/noteA)*(1.0-relPos)-
  simpleStringTermDelay(fretFc))),wg1cpy6,fretcpy6,junct0cpy6;
waveguide(2.5,simpleString(0.027,(1.0/(noteF-1.0))*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy7,bridgecpy7,junct0
  cpy7;
waveguide(2.5,simpleString(0.027,(1.0/(noteF-1.0))*(1.0-relPos)
  -simpleStringTermDelay(fretFc))),wg1cpy7,fretcpy7,junct0cpy7;
waveguide(2.5,simpleString(0.027,(1.0/noteF)*relPos-
  simpleStringTermDelay(bridgeFc))),wg0cpy8,bridgecpy8,junct0
  cpy8;
waveguide(2.5,simpleString(0.027,(1.0/noteF)*(1.0-relPos)-
  simpleStringTermDelay(fretFc))),wg1cpy8,fretcpy8,junct0cpy8;


termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridge; # pos 160,146
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fret; # pos 672,146
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy; # pos 672,178
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy; # pos 160,178
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy1; # pos 672,258
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy1; # pos 160,258
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy2; # pos 664,290
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy2; # pos 152,290
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy3; # pos 688,370
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy3; # pos 152,370
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy4; # pos 688,394
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy4; # pos 152,394
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy5; # pos 672,458
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy5; # pos 160,458
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy6; # pos 680,482
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy6; # pos 160,482
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy7; # pos 688,562
```

```
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy7; # pos 176,562
termination(simpleStringTerm(-1.0+pow(10.0,-fretAtten:float),
  fretFc)),fretcpy8; # pos 696,586
termination(simpleStringTerm(-1.0+pow(10.0,-bridgeAtten:float),
   bridgeFc)),bridgecpy8; # pos 176,586



junction(0.04),junct0; # pos 400,136
junction(0.04),junct0cpy; # pos 400,168
junction(0.03),junct0cpy1; # pos 400,248
junction(0.03),junct0cpy2; # pos 400,280
junction(0.02),junct0cpy3; # pos 400,360
junction(0.02),junct0cpy4; # pos 400,384
junction(0.01),junct0cpy5; # pos 400,448
junction(0.01),junct0cpy6; # pos 400,472
junction(0.0),junct0cpy7; # pos 400,552
junction(0.0),junct0cpy8; # pos 400,576



faustcode: freqMIDI=hslider("Frequency [MIDI]", 27.0, 20.0,
  84.0, 0.05);
faustcode: freq = 440.0*pow(2.0,(freqMIDI-69.0)/12.0);
faustcode: k=hslider("Pluck stiffness [N/m
  ]",250.0,50.0,1000.0,50.0);
faustcode: R=hslider("Pluck damping [N/(m/s)
  ]",0.1,0.0,2.5,0.01);
faustcode: relPos=hslider("Pluck position along length
  ",0.29,0.01,0.99,0.01);
faustcode: maxdisp=hslider("Pluck half-width [m
  ]",0.006,0.001,0.01,0.001);
faustcode: bridgeAtten=hslider("Bridge attenuation
  ",2.5,0.5,7.0,0.01);
faustcode: bridgeFc=hslider("Bridge cutoff freq [Hz
  ]",5000.0,1000.0,fs,10.0);
faustcode: fretAtten=hslider("Fret attenuation
  ",2.5,0.5,7.0,0.01);
faustcode: fretFc=hslider("Fret cutoff freq [Hz
  ]",5000.0,1000.0,fs,10.0);
faustcode: moveStrings=hslider("Waveguides position adjustment
  ",1.0,0.1,1.4,0.01);
faustcode: vol=hslider("Z Volume",0.3,0.01,1.0,0.01) :
  onePoleBLT(10.0); // LP filter with cutoff frequency at 10Hz
  smoothes volume control signal
faustcode: outputDSP=highpass(4,20.0);
faustcode: noteF=174.61;
faustcode: noteA=220.0;
faustcode: noteD=293.66;
faustcode: noteG=392.0;
faustcode: noteC=523.25;
```

```
audioout,a0,(bridge+bridgecpy+bridgecpy1+bridgecpy2+bridgecpy3+
  bridgecpy4+bridgecpy5+bridgecpy6+bridgecpy7+bridgecpy8)*vol:
  outputDSP:bodyResp1:SAMfreereverb; # pos 59,336
audioout,a1,(fret+fretcpy+fretcpy1+fretcpy2+fretcpy3+fretcpy4+
  fretcpy5+fretcpy6+fretcpy7+fretcpy8)*vol:outputDSP:bodyResp2:
  SAMfreereverb; # pos 851,336
```

A.3 EMACS SUPPORT

Listing 36: Source of sam-mode for Emacs

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Synth-A-Modeler mode (basic syntax highlighting)
;; author: ptrv <mail@petervasil.net>
;; based on the FAUST mode by rukano:
;; https://github.com/rukano/emacs-faust-mode
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Installation:
;;
;; Put sam-mode.el to your load-path and add this to
;; your .emacs:
;;
;; (setq auto-mode-alist (cons '("\\.mdl$" . sam-mode)
  auto-mode-alist))
;; (autoload 'sam-mode "sam-mode" "Synth-A-Modeler editing mode
  ." t)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defvar sam-keywords
  '("mass" "ground" "port" "resonator" "link" "touch" "pluck"
    "waveguide" "termination" "junction" "audioout"))

(defvar sam-functions
  '("simpleString" "simpleStringTerm"))

(defvar sam-ui-keywords
  '("faustcode"))

;; optimize regex for words
(defvar sam-variables-regexp "[A-Za-z][A-Za-z]*")
(defvar sam-arguments-regexp "[0-9]")
(defvar sam-operator-regexp "\\([~!_@,<>:;]\\)")
(defvar sam-math-op-regexp "[=\+\{\}()/*-]")
(defvar sam-keywords-regexp (regexp-opt sam-keywords 'words))
(defvar sam-function-regexp (regexp-opt sam-functions 'words))
(defvar sam-ui-keywords-regexp (regexp-opt sam-ui-keywords '
  words))
```

```
;; create the list for font-lock.
(setq sam-font-lock-keywords
  `(
    (,sam-function-regexp . font-lock-type-face)
    (,sam-ui-keywords-regexp . font-lock-builtin-face)
    (,sam-math-op-regexp . font-lock-function-name-face)
    (,sam-operator-regexp . font-lock-constant-face)
    (,sam-keywords-regexp . font-lock-keyword-face)
    ))

;; define the mode
(define-derived-mode sam-mode fundamental-mode
  "SAM"
  "Major mode for editing Synth-A-Modeler files"

  ;; code for syntax highlighting
  (setq font-lock-defaults '((sam-font-lock-keywords)))

  ;; modify the keymap
  (define-key sam-mode-map [remap comment-dwim] '
    sam-comment-dwim))

;; comment dwin support
(defun sam-comment-dwim (arg)
  "Comment or uncomment current line or region in a smart way.
For detail, see `comment-dwim'."
  (interactive "*P")
  (require 'newcomment)
  (let ((deactivate-mark nil) (comment-start "#") (comment-end
    ""))
    (comment-dwim arg)))

(modify-syntax-entry ?#  "< b" sam-mode-syntax-table)
(modify-syntax-entry ?\n "> b" sam-mode-syntax-table)
```

A.4 VIM SUPPORT

Listing 37: Vim support source

```
" There has to be a file called filetype.vim in your ~/.vim
  directory
" with the following content:
" sam filetype file
"       if exists("did_load_filetypes")
"         finish
"       endif
"       augroup filetypedetect
"         au! BufRead,BufNewFile *.mdl          setfiletype sam
"       augroup END
"
"
```

```
" Synth-A-Modeler syntax file
" Language:     Synth-A-Modeler
" Maintainer: Peter Vasil <mail@petervasil.net>
" Version:      0.1
" Last change:  2012-10-04

" remove any old syntax stuff hanging around
syn clear

""""""""""""""""""""""""""""""""""""""""""""""""""
" sam primitives
syn keyword samPrims mass ground port resonator link touch
  pluck waveguide termination junction audioout faustcode

""""""""""""""""""""""""""""""""""""""""""""""""""
" sam operators
syn keyword samOps simpleString simpleStringTerm

""""""""""""""""""""""""""""""""""""""""""""""""""
" sam comments
syn match       samComment      "#.*$"
" syn region    samComment      start="/\*" end="\*/" contains=
  samOperator keepend extend

""""""""""""""""""""""""""""""""""""""""""""""""""
" sam operators
syn match samOperator   "+"
syn match samOperator   "-"
syn match samOperator   "*"
syn match samOperator   ":"
syn match samOperator   ","

""""""""""""""""""""""""""""""""""""""""""""""""""
" sam brackets
syn match samAoperator  "("
syn match samAoperator  ")"

" String
syn region samString    start=+"+ skip=+\\\\\|\\"+ end=+"+

" Color definition
hi link samAoperator    Statement
hi link samPrims        Label
hi link samOps          Identifier
hi link samOperator     Special
hi link samComment      Comment
hi link samString       String

" The name of the syntax is sam
let b:current_syntax = "sam"
```

A.5 BUILDING *synth-a-modeler* ON A *beagle-board*

This section will briefly describe how to compile *Synth-A-Modeler Designer* on a *Beagle-Board*[55] with an *ARM* processor, which is not officially supported by *JUCE*. *BeagleBoard* is small computer used for embedded computing. The actual hardware, used for testing, a *BeagleBoard xM*, has been provided by Edgar Berdahl. In order to compile *Synth-A-Modeler Designer*, some code has to be modified in the *JUCE* sources. The description has been tested on *Satellite CCRMA*,[56] an Ubuntu[57] based Linux distribution, created to serve as "platform for building embedded musical instruments and embedded art installations" [40].

1. The first step is, to install the dependencies for *JUCE*. The following list, taken from a forum entry on the *JUCE* website, shows all commands to install the libraries:

   ```
   sudo apt-get -y install g++
   sudo apt-get -y install libfreetype6-dev
   sudo apt-get -y install libx11-dev
   sudo apt-get -y install libxinerama-dev
   sudo apt-get -y install libxcursor-dev
   sudo apt-get -y install mesa-common-dev
   sudo apt-get -y install libasound2-dev
   sudo apt-get -y install freeglut3-dev
   sudo apt-get -y install libxcomposite-dev
   ```

2. We have to disable shared memory to be able to connect the Beagle-Board via Ethernet and ssh forwarding. This makes it possible to use GUIs applications over X11. When using the Beagle-Board directly, this is not necessary.

   In `AppConfig.h` edit the preprocessor command:

   ```
   #define    JUCE_USE_XSHM 0
   ```

3. We have to disable 64 bit Atomics in

   ```
   juce/module/juce_core/memory/juce_Atomic.h
   ```

   by commenting out the corresponding code

   ```
   #elif JUCE_GCC
     #define JUCE_ATOMICS_GCC 1        // GCC with intrinsics
   ```

55 http://beagleboard.org/
56 https://ccrma.stanford.edu/~eberdahl/Satellite/
57 http://www.ubuntu.com

```
// COMMENT OUT THE IF STATEMENT HERE------->
 //#if JUCE_IOS || JUCE_ANDROID // 64-bit ops will compile but not link on th
    #define JUCE_64BIT_ATOMICS_UNAVAILABLE 1
  //#endif

//======================================================================
```

4. Then we have to specify the target architecture by running the following command to set the `TARGET_ARCH` environment variable

```
export TARGET_ARCH=-march=armv7-a \
    -O3 -mtune=cortex-a8 -mfpu=neon \
    -mfloat-abi=softfp
```

5. Finally we can build the application by executing

```
cd Synth-A-Modeler/gui/Builds/Linux
make
```

A.6 CLASS HIERARCHY

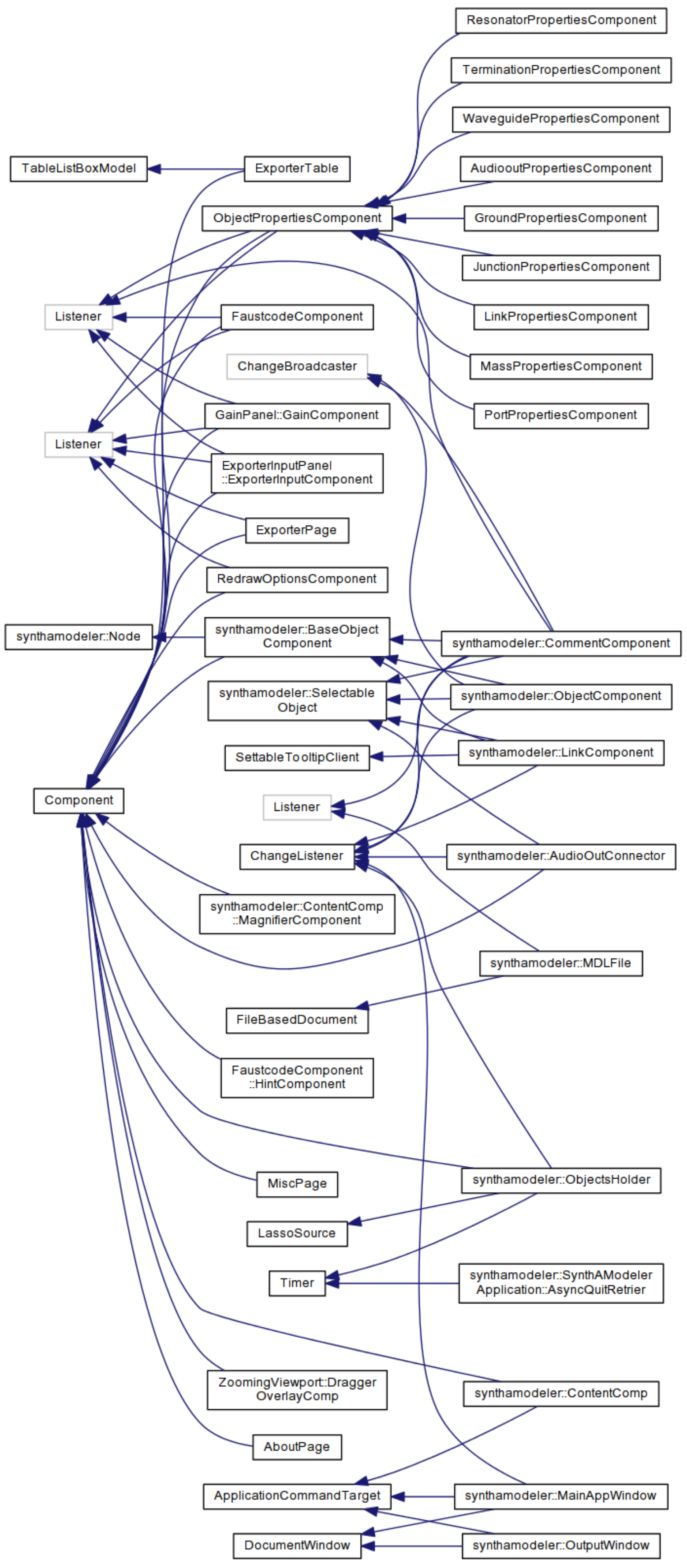

ResonatorPropertiesComponent
TerminationPropertiesComponent
WaveguidePropertiesComponent
TableListBoxModel
ExporterTable
AudiooutPropertiesComponent
ObjectPropertiesComponent
GroundPropertiesComponent
JunctionPropertiesComponent
Listener
FaustcodeComponent
LinkPropertiesComponent
ChangeBroadcaster
MassPropertiesComponent
GainPanel::GainComponent
PortPropertiesComponent
Listener
ExporterInputPanel ::ExporterInputComponent
ExporterPage
RedrawOptionsComponent
synthamodeler::Node
synthamodeler::BaseObject Component
synthamodeler::CommentComponent
synthamodeler::Selectable Object
synthamodeler::ObjectComponent
SettableTooltipClient
synthamodeler::LinkComponent
Component
Listener
ChangeListener
synthamodeler::AudioOutConnector
synthamodeler::ContentComp ::MagnifierComponent
synthamodeler::MDLFile
FileBasedDocument
FaustcodeComponent ::HintComponent
synthamodeler::ObjectsHolder
MiscPage
LassoSource
Timer
synthamodeler::SynthAModeler Application::AsyncQuitRetrier
ZoomingViewport::Dragger OverlayComp
synthamodeler::ContentComp
AboutPage
ApplicationCommandTarget
synthamodeler::MainAppWindow
DocumentWindow
synthamodeler::OutputWindow

Figure 38: Class hierarchy diagram

BIBLIOGRAPHY

[1] J.O. Smith III. Physical audio signal processing. *Julius Orion Smith III Homepage, https://ccrma.stanford.edu/~jos/pasp/*, 2004. (Cited on pages 2, 7, and 87.)

[2] E. Berdahl and J.O. Smith III. An introduction to the Synth-A-Modeler compiler: Modular and open-source sound synthesis using physical models. In *Proceedings of the Linux Audio Conference, Stanford, CA*, 2012. (Cited on pages 2, 3, 12, 14, 16, and 25.)

[3] Rudolf Rabenstein, Stefan Petrausch, Augusto Sarti, Giovanni De Sanctis, Cumhur Erkut, and Matti Karjalainen. Blocked-based physical modeling for digital sound synthesis. *Signal Processing Magazine, IEEE*, 24(2):42–54, 2007. (Cited on page 3.)

[4] J.O. Smith III. A basic introduction to digital waveguide synthesis. *Center for Computer Research in Music and Acoustics (CCRMA), Stanford University. http://ccrma.stanford.edu/~jos/swgt*, 2006. The reference is recommended for the technically inclined. (Cited on pages 6, 7, and 8.)

[5] N. Castagne and C. Cadoz. Creating music by means of 'physical thinking': The musician oriented genesis environment. In *Proceedings of the fifth annual Conference on Digital Audio Effects, Hamburg, Germany*, 2002. (Cited on page 8.)

[6] Kees Van den Doel and Dinesh K Pai. Modal synthesis for vibrating objects. *Audio Anectodes. AK Peter, Natick, MA*, 2003. (Cited on page 8.)

[7] David Bolton. Definition of beta. http://cplus.about.com/od/glossar1/g/betadefinition.htm, April 2012. [Accessed 2013-02-02 16:22:04]. (Cited on page 9.)

[8] N. Castagné, C. Cadoz, A. Allaoui, O. Tache, et al. G3: Genesis software environment update. *arXiv preprint arXiv:0911.4642*, 2009. (Cited on pages 9 and 10.)

[9] C. Cadoz, A. Luciani, and J.L. Florens. A modeling and simulation system for sound and image synthesis: The general formalism. *Computer music journal*, 17(1):19–29, 1993. (Cited on page 9.)

[10] Tcl sourceforge project. http://tcl.sourceforge.net/. [Accessed 2013-02-02 20:27:50]. (Cited on page 10.)

[11] F. Iovino, R. Caussé, and R. Dudas. Recent work around modalys and modal synthesis. In *ICMC: International Computer Music Conference, Thessaloniki Hellas, Greece*, pages 356–359, 1997. (Cited on page 10.)

[12] R. Polfreman. Modalys-ER for OpenMusic (MfOM): virtual instruments and virtual musicians. *Organised Sound*, 7(3):325–338, 2002. (Cited on pages 10 and 11.)

[13] IRCAM Music Representations Team. Openmusic website. `http://repmus.ircam.fr/openmusic/home`, 11 2012. [Accessed 2013-02-04 14:12:45]. (Cited on page 10.)

[14] The Open Group. Regular expressions. `http://pubs.opengroup.org/onlinepubs/007908799/xbd/re.html`, 1997. [Accessed 2013-03-06 12:58:54]. (Cited on page 13.)

[15] Grame. FAUST. `http://faust.grame.fr`, 2011. [Accessed 2013-03-06 13:35:03]. (Cited on page 13.)

[16] Inc. Elgris Technologies. Elgris / edif implementation. `http://www.elgris.com/content/edif_overview.html`, 2005. [Accessed Fri Jan 18 2013 19:20:32]. (Cited on page 16.)

[17] GCC Team. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`, 03 2013. [Accessed 2013-03-07 13:26:50]. (Cited on page 19.)

[18] J. Smith III. Audio signal processing in Faust. *online tutorial: `https://ccrma.stanford.edu/~jos/aspf/aspf.html`*, 2010. [Accessed 2013-03-06 15:12:24]. (Cited on page 19.)

[19] IEM - Institute of Electronic Music and Acoustics. Puredata website. `http://puredata.info/`, 2013. [Accessed 2013-03-08 12:30:36]. (Cited on page 19.)

[20] Gräf Albert. Interfacing PureData with Faust. *LINUX AUDIO*, page 24, 2007. (Cited on page 19.)

[21] SuperCollider community. SuperCollider website. `http://supercollider.sourceforge.net/`, 2013. [Accessed 2013-03-08 17:12:52]. (Cited on page 21.)

[22] Scott Wilson, David Cottle, and Nick Collins. The SuperCollider Book. 2011. (Cited on page 21.)

[23] Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. Design patterns: Elements of reusable object-oriented software. *Reading: Addison Wesley Publishing Company*, 1995. (Cited on pages 30, 31, and 41.)

[24] Raw Material Software. Summary of juce's features. http://rawmaterialsoftware.com/jucefeatures.php, 2013. [Accessed 2013-03-12 15:26:10]. (Cited on page 33.)

[25] Free Software Foundation. GNU General Public License, version 2. http://www.gnu.org/licenses/gpl-2.0.html, 2013. [Accessed 2013-03-12 16:05:53]. (Cited on page 33.)

[26] OJ Reeves. The Magic of Unity Builds. http://buffered.io/posts/the-magic-of-unity-builds, Decenmber 2007. [Accessed 2013-03-12 18:10:28]. (Cited on page 34.)

[27] The IEEE and The Open Group. The open group base specifications issue 6, regular expressions. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html, 2004. [Accessed 2013-03-14 15:17:18]. (Cited on page 35.)

[28] Russ Cox. Regular Expression Matching Can Be Simple And Fast. http://swtch.com/~rsc/regexp/regexp1.html, 2007. [Accessed 2013-03-12 17:52:54]. (Cited on page 36.)

[29] Russ Cox. Regular expression matching in the wild. http://swtch.com/~rsc/regexp/regexp3.html, 2010. [Accessed 2013-03-14 16:22:54]. (Cited on page 36.)

[30] Scott Chacon. *Pro Git*. Apress, 2009. (Cited on page 36.)

[31] Raw Material Software. JUCE Documentation. http://www.rawmaterialsoftware.com/juce/api/index.html, 2013. [Accessed 2013-03-29 18:39:29]. (Cited on pages 44, 46, 48, and 49.)

[32] D. Huizinga and A. Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley, 2007. ISBN 9780470165164. URL http://books.google.com.pk/books?id=PhnoE90CmdIC. (Cited on page 63.)

[33] Roberto Tamassia. *Handbook of graph drawing and visualization*. Chapman & Hall/CRC, 2007. (Cited on page 64.)

[34] Mauro Brunato. Advanced business intelligence techniques 13: Force-based graph layout algorithms. Video, http://www.youtube.com/watch?v=2nA6vbIeX1s, 07 2012. [Accessed 2013-06-27 10:38:47]. (Cited on page 64.)

[35] Mike Kamermans. Simple graph visualisation. http://processingjs.nihongoresources.com/graphs/, 2011. [Accessed 2013-04-20 18:52:05]. (Cited on page 66.)

[36] M. Jahandideh, S. Khaefi, A. Jahandideh, and M. Khaefi. Using the root proportion to design an oud. In *Proceedings of the*

11th WSEAS international conference on Acoustics & music: theory & applications, pages 35–39. World Scientific and Engineering Academy and Society (WSEAS), 2010. (Cited on pages 83 and 84.)

[37] Erica Goode. A Fabled Iraqi Instrument Thrives in Exile. http://www.nytimes.com/2008/05/01/world/middleeast/01oud.html, 2008. [Accessed Sat 19 Jan 2013 23:55:31]. (Cited on page 83.)

[38] Maqam World. What is a Maqam? http://www.maqamworld.com/maqamat.html, 2003. [Accessed Sat 19 Jan 2013 22:33:00]. (Cited on page 84.)

[39] T.D. Rossing. *Science of String Instruments*. Springer Verlag, 2010. (Cited on pages 84 and 85.)

[40] Edgar Berdahl and Wendy Ju. Satellite CCRMA. https://ccrma.stanford.edu/~eberdahl/Satellite/. [Accessed 2013-03-21 18:12:54]. (Cited on page 100.)

ERKLÄRUNG

Hiermit versichere ich an Eides statt, dass ich die vorliegende Masterarbeit ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Teile, die wörtlich oder sinngemäß einer Veröffentlichung entstammen, sind als solche kenntlich gemacht. Die Arbeit wurde noch nicht veröffentlicht oder einer anderen Prüfungsbehörde vorgelegt.

Berlin, den 25. Juli 2013

Peter Vasil