MEPHISTO: A SOURCE TO SOURCE TRANSPILER FROM PURE DATA TO FAUST

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ABDULLAH ONUR DEMIR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MODELLING AND SIMULATION

JUNE 2015

Approval of the thesis:

MEPHISTO: A SOURCE TO SOURCE TRANSPILER FROM PURE DATA TO FAUST

submitted by **ABDULLAH ONUR DEMIR** in partial fulfillment of the requirements for the degree of **Master of Science in Modelling and Simulation Department, Middle East Technical University** by,

Prof. Dr. Nazife Baykal
Dean, Graduate School of **Informatics** _____

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Head of Department, **Modelling and Simulation** _____

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Supervisor, **Modelling and Simulation Department, METU** _____

Examining Committee Members:

Prof. Dr. Cem Bozşahin
Cognitive Science Department, METU _____

Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu
Modelling and Simulation Department, METU _____

Prof. Dr. Nihan Kesim Çiçekli
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Alptekin Temizel
Modelling and Simulation Department, METU _____

Assist. Prof. Dr. Murat Yılmaz
Computer Engineering Department,
Çankaya University _____

Date: **11.06.2015**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ABDULLAH ONUR DEMIR

Signature :

ABSTRACT

MEPHISTO: A SOURCE TO SOURCE TRANSPILER FROM PURE DATA TO FAUST

Demir, Abdullah Onur

M.S., Department of Modelling and Simulation

Supervisor : Assoc. Prof. Dr. Hüseyin Hacıhabiboğlu

June 2015, 47 pages

PD and Max/MSP softwares are widely used for procedural sound design for games. However, the software cannot be easily integrated with games. On the other hand, various optimized synthesizes can be developed using low level languages by Faust. Transporting models and applying the same algorithms from PD to Faust is really a burden to procedural sound designers. The aim of this thesis is to write a source to source transpiler which transpiles PD source to Faust source. After created, the transpiler is assessed by speed and stability tests. Finally, pros and cons of the transpiler are assessed.

Keywords: High performance sound synthesis, Pure Data, Faust

ÖZ

MEPHISTO: PURE DATA'DAN FAUST'A KAYNAKTAN KAYNAĞA ÇAPRAZ DERLEYİCİ

Demir, Abdullah Onur
Yüksek Lisans, Modelleme ve Simülasyon Bölümü
Tez Yöneticisi : Doç. Dr. Hüseyin Hacıhabiboğlu

Haziran 2015 , 47 sayfa

Oyunlar için ses sentezinde prosedürel olarak PD ve Max/MSP yazılımları kullanılmaktadır. Fakat PD programları oyunların ya da uygulamaların içerisine optimize olarak yerleştirilememektedir. Bu işi yapan Faust dili ile optimize ve düşük programlama dilleri ile sentezler geliştirilebilmektedir. Ancak PD'den Faust'a modelleri aktarmak ve aynı algoritmayı Faust'ta ayrıca uygulamak çok yük getiren bir iş olarak karşımıza çıkmaktadır. Amacımız PD ya da Max dillerinden Faust'a geçiş yapmamızı sağlayacak kaynaktan kaynağa bir derleyici yazmaktır. Bu derleyici yapıldıktan sonra da hız ve denge testleri uygulanıp derleyici daha etkili hale getirilecektir. Öte yandan ise bu derleyicinin avantajları ve dezavantajları değerlendirilecektir.

Anahtar Kelimeler: Yüksek performanslı ses sentezleme, Pure Data, Faust

To my family, girlfriend and best friends

ACKNOWLEDGMENTS

First of all, I am so grateful to my advisor, Hüseyin Hacıhabiboğlu, who raised me with his friendly attitude when I lost my motivation and did not leave me alone by directing me with his valuable feedbacks. Most importantly, I appreciate him for believing in that we will give a great contribution to the field.

I would like to give my heartfelt gratitude to breath of my life, my love, Deniz Emiroğulları for believing in and always supporting me by giving of herself in every condition.

I also want to express my thanks to my friend, Ali Osman Ataç, who has the same frame of mind with me and understands me in every situation.

It is with immense gratitude that I acknowledge the scholarship (2210) of Scientific and Technological Research Council of Turkey for my master study.

Last but not the least, I am grateful to my family, my father Mehmet Demir and my mother Ayşe Demir for supporting me and my ideas through all my life and teaching me how to behave freely. I wish to give my thanks to my brother and sister, Halil İbrahim Demir and Yasemin Nur Demir who take me as model to themselves and support me every time.

Special thanks to my dear dog, Dali, for her unconditional love.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

PD	Pure Data
IDE	Integrated Development Environment
PDA	Personal Digital Assistant
PUDAC	Pure Data Compiler
ANTLR	Another Tool for Language Recognition
CCITT	International Telegraph and Telephone Consultative Committee
CPU	Central Processing Unit
GPL	General Public License

CHAPTER 1

INTRODUCTION

Sound synthesis is crucially important for games and virtual reality applications. High quality audio has a decisive role while creating realistic environments and evoking interest in user experience in video games [1]. There are two common techniques used to create high quality sounds in games. As a first technique, prerecorded clips, also known as Foley sounds, are used. Although prerecorded clips provide perfect realism and low computational cost, memory footprint becomes the main bottleneck. Prerecorded clips should reside in memory since the I/O latency of the disk is unacceptable. Besides, loading every sound in physical memory is not a very good solution because of the fact that typically a limited amount of hardware resources is allocated for sounds in games. Moreover, it is particularly hard if not impossible to record sounds such as jet engines, gunshots, rain or wind due to physical and practical reasons.

The second alternative is based on the concept of parametric sound synthesis with which impressive results can be obtained by algorithmic means. Such algorithms allow the model-based generation of hard-to-record sounds by simple signal generators and signal processing methods yet provide convincing results. This approach extends the sound designer's palette by providing her with the means to construct and control virtual sound objects. Synthesized sounds are computer programs that can be executed and parametrically adjusted in real time [2].

1.1 Pure Data

Pure Data (PD) [3] is a popular *data-flow* programming language with which composers, performers and developers can synthesize sounds without writing code but by using graphical objects and connections between these objects (Fig. 1.1). PD does real time computations using a Max-like message interpreter and scheduler and operates on vector samples in order to minimize the interpretation overhead and to satisfy the needs of the real time audio applications. However, sample level computations have to be carried out by using external plugins or primitives. Hence, PD programs depend on a run-time environment. As a result, although not impossible, embedding interpreted PD programs in games or other systems is generally difficult and inefficient in comparison with code written directly in compiled languages like C or C++ [4].

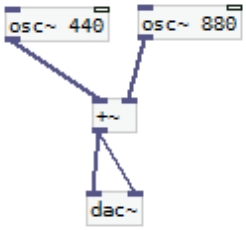


Figure 1.1: Example of a PD patch

1.2 Faust

Faust (Functional AUdio STreams) is a functional, *block diagram* programming language designed and used specifically for processing digital audio signals in real-time [5]. It can be used to create high-performance audio applications and plug-ins. While Faust is well-suited for signal processing, it lacks sufficiently elaborate control mechanisms offering only basic user interface elements like buttons, sliders and number boxes. FaustWorks IDE allows displaying block diagrams of signal flows which are written in the Faust language to visualize the application. Faust was designed 1) to be highly expressive, 2) to have clean mathematical semantics, and 3) to be highly efficient [6, 7]. A simple Faust program is displayed in (1.1) and Fig. 1.2 displays the corresponding block diagram.

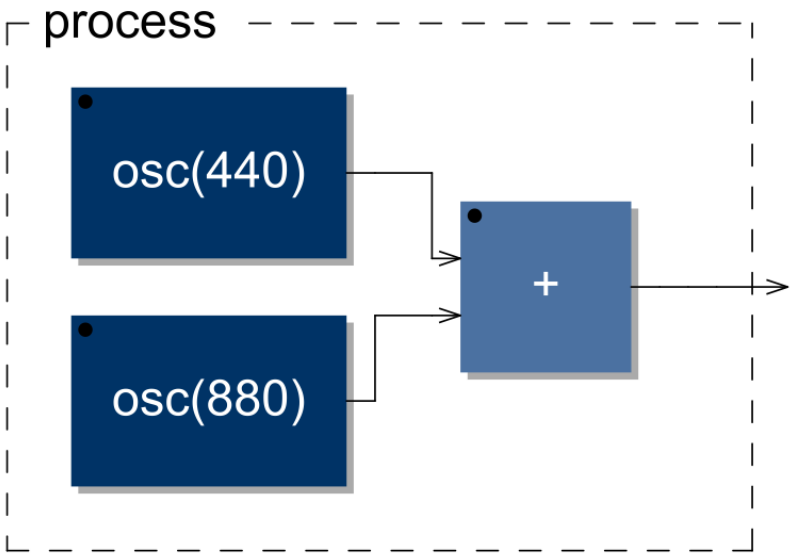


Figure 1.2: Block diagram representation of a Faust source code

$$\begin{aligned} &\texttt{import}(\text{"}\texttt{music.lib}\text{"});\\ &\texttt{process} = \texttt{osc}(440) + \texttt{osc}(880); \end{aligned} \tag{1.1}$$

1.3 Motivation

Data-flow languages such as Pure Data, Bidule or Max are popular since programming audio graphically is much easier than writing code. Besides, such languages allow changing parameters and observing the results on-the-fly. However, a particular problem with PD is that the developed algorithms are difficult to integrate in the final product due to the reduced performance[1], necessity to bundle the run-time environment with the final product (e.g. a game or a mobile app) and incompatibility with existing development frameworks.

In contrast, programs written in Faust can be directly translated to optimized C++ code for sound processing, can be embedded into a final product in a more straightforward way and also provide a higher performance. However, the main drawback of the Faust language is that the programmer has to learn functional programming concepts, syntax and semantics of Faust and mathematical descriptions of signals to be able to code in Faust [7].

This thesis presents Mephisto [9], a source-to-source transpiler from Pure Data to Faust which aims to bridge the gap between the two programming languages and to facilitate easier design and integration of audio algorithms in relevant software development processes.

1.4 Organization of the thesis

This thesis is organized as follows. Chapter 2 presents a brief information about sound synthesis methods which are used over decades. Afterwards, it describes what the procedural sound synthesis is and gives information about the tools on which procedural sound synthesis algorithms can be implemented. Finally, it discusses the previous studies which were developed in order to embed the designed and implemented procedural sound algorithms into the games. Chapter 3 presents the transpiler design. Firstly, the parsing technique is described. Afterwards, the original algorithm and data structure of the transpiler and its semantics are explained. Finally, the functions of the transpiler used with the data structure in the context of the transpiler algorithm are presented. Chapter 4 presents the examples. There are 3 enchanting examples in this chapter. All of them are real examples of procedural sound synthesis algorithms designed in Pure Data. They were transpiled with Mephisto and the results showed the real power of it. Chapter 5 presents the evaluation of Mephisto. Firstly, the conventions and limitations are discussed. Afterwards, the examples' performance comparisons between the previous works and Mephisto are displayed. Chapter 6 concludes the thesis by recommending the future works which will enhance Mephisto in a rapid manner and saying the final words.

[1] Pure Data is reported to be roughly three orders of magnitude slower than its C equivalent for multiplying floating point numbers [8].

CHAPTER 2

BACKGROUND

2.1 Digital Sound Synthesis

Bell Telephone Laboratories started the first experiments in synthesis of sound by computers in 1957 in Murray Hill, New Jersey [10, 11, 12]. Max Mathews and his colleagues working at Bell Telephone Laboratories proved that sound synthesis by a computer was possible with respect to any pitch scale or waveform, including time-varying frequency and amplitude envelopes. They programmed their first digital sound synthesis program, Music I, which was written directly in terms of machine instructions for IBM 704 [13].

The program written by Mathews aimed at generating a single waveform: an equilateral triangle signal. Notes could be specified only in terms of pitch, waveform and duration [11]. A psychologist, Newman Guttman used Music I and composed a monophonic etude called *In a Silver Scale* on May 17, 1957 [14]. It became the first composition synthesized by the process of digital-to-analog conversion. Although it was the first piece, the computer's potential of generating any frequency precisely was shown. After Music I, a dozen of software synthesis systems have been developed by researchers named in the concept of Music N [13, 15].

2.2 Synthesis Methods

Variation of some aspects of a signal (*carrier*) with respect to another signal (*modulator*) is called *modulation* in electronic and computer music. Modulation synthesis is preferable to additive and subtractive synthesis methods compared in terms of parameter data, memory requirements and computation time [16, 17]. Modulation uses fewer oscillators (typically two to six) while additive and subtractive techniques need much more computational resources. Modulation is used by a few table-lookup, multiplication, addition operations based on the selected modulation. Since, fewer parameters are required than additive and subtractive techniques, modulation techniques are always found easier. By changing parameter values over time, modulation techniques easily produce time-varying spectra. Modulations can generate rich dynamic sounds reaching close to natural instrumental notes when carefully regulated.

2.2.1 Ring Modulation

In theory, ring modulation is similar to amplitude modulation [18]. It is the multiplication of two bipolar[1] audio signals. One is the *carrier*, C, and the other one is the *modulator*, M. The ring modulated signal $R(t)$ at time t is given by $R(t) = M(t)C(t)$. When the frequency of the *modulator*, M is below 20 Hz, the amplitude of the *carrier*, C, changes at the frequency of, M. Nonetheless, if the modulator's frequency is in audible range, the timbre[2] of C changes. For each center frequency in the *carrier*, the *modulator* generates sidebands that are located at $f_C + f_M$ and $f_C - f_M$ where f_C and f_M are the frequencies of the *carrier* C and the *modulator* M, respectively.

There are two typical usages of this technique. One is to modify the sampled *carrier* signals like human or instrument voices by sine wave modulators. The other one is to create a fully synthesized virtual sounds using sine waves in harmonic or in-harmonic ratios [19].

2.2.2 Amplitude Modulation

In amplitude modulation, two signals are used. One is the carrier signal and the other one is modulator signal which is actually the information signal. Modulator signal is generally in low frequencies up to about 4 kHz while the carrier signal has higher frequencies which are generally higher than 500 kHz. Modulator signal modulates the instantaneous amplitude value of the carrier signal by its amplitude while the frequency of the carrier remains constant [20]. Fig. 2.1 represents the process visually.

Formally, let us define the signals as,

$$C(t) = Acos(\omega_c * t)$$

$$M(t) = \beta cos(\omega_m * t)$$

Here the amplitude A of the carrier signal $C(t)$ is modulated by $M(t)$ in order to create the equation,

$$A(1 + M(t))cos(\omega_c * t)$$

Additionally, the magnitude of $M(t)$ should be less than or equal to 1 due to reasons of demodulation of the signal.

Amplitude modulation has been used in various applications since the radio technology is emerging. However, it is not used as widely as previous years but it can be still found. It is easy and simple to implement amplitude modulation but it is not that much efficient in terms of spectrum efficiency and power usage. Hence, frequency modulation and dozen of digital modulation formats are preferred to amplitude modulation [20].

[1] Bipolar signal is a signal whose values range symmetrically with respect to the zero point.
[2] The character or quality of a musical sound or voice as distinct from its pitch and intensity.

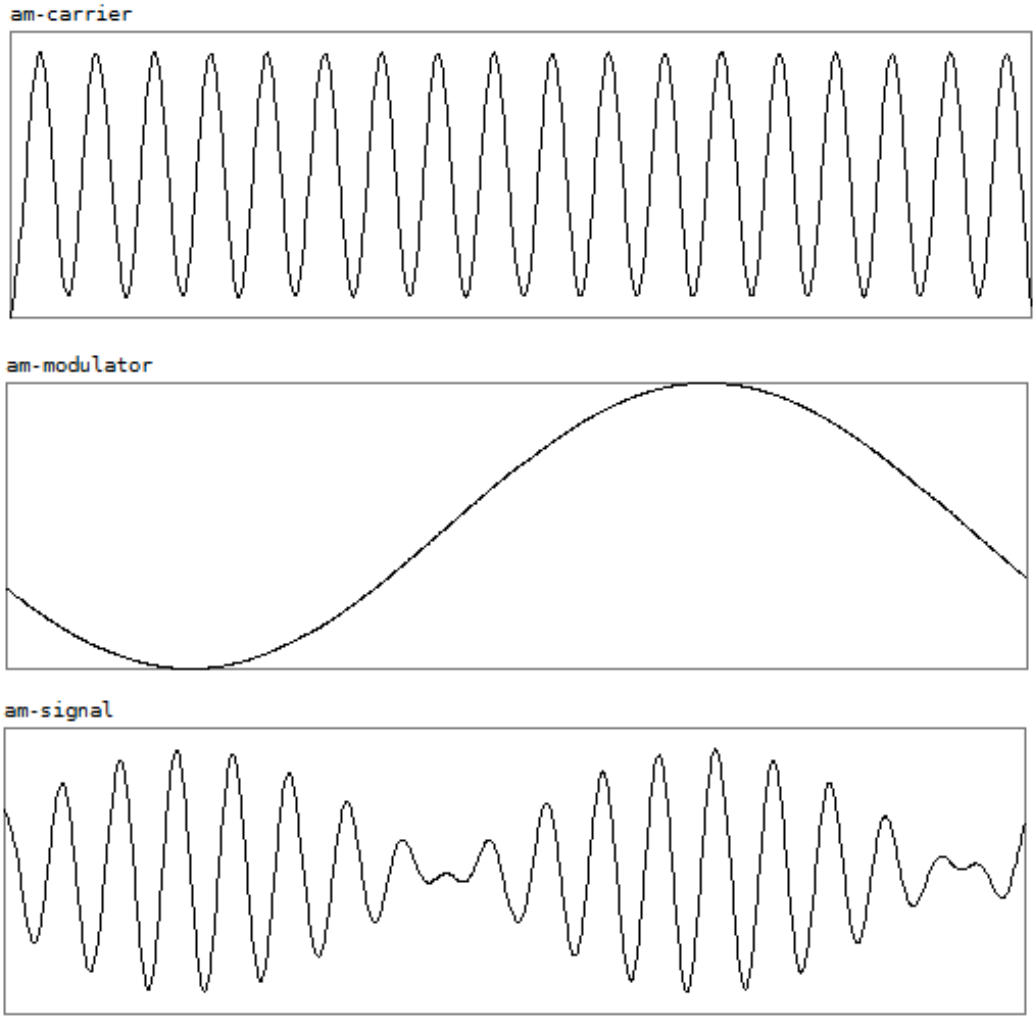


Figure 2.1: A single frequency sine wave modulating a higher frequency carrier signal. The variation of the amplitude of the carrier signal is clearly seen in *am-signal*.

2.2.3 Frequency Modulation

In frequency modulation, two signals are used as in amplitude modulation. One is the carrier $C(t)$ and the other one is the modulator $M(t)$. Modulator signals are generally up to 4 kHz while the carrier signals change between 88 and 108 MHz. The difference of frequency modulation from amplitude modulation is that the instantaneous frequency of the carrier signal is varied by the modulator signal. Fig. 2.2 represents the process visually.

2.3 Procedural Sound Synthesis

Methods including the standard ones described above are routinely used in synthesis algorithms that aim to generate sounds for games, movies and virtual reality

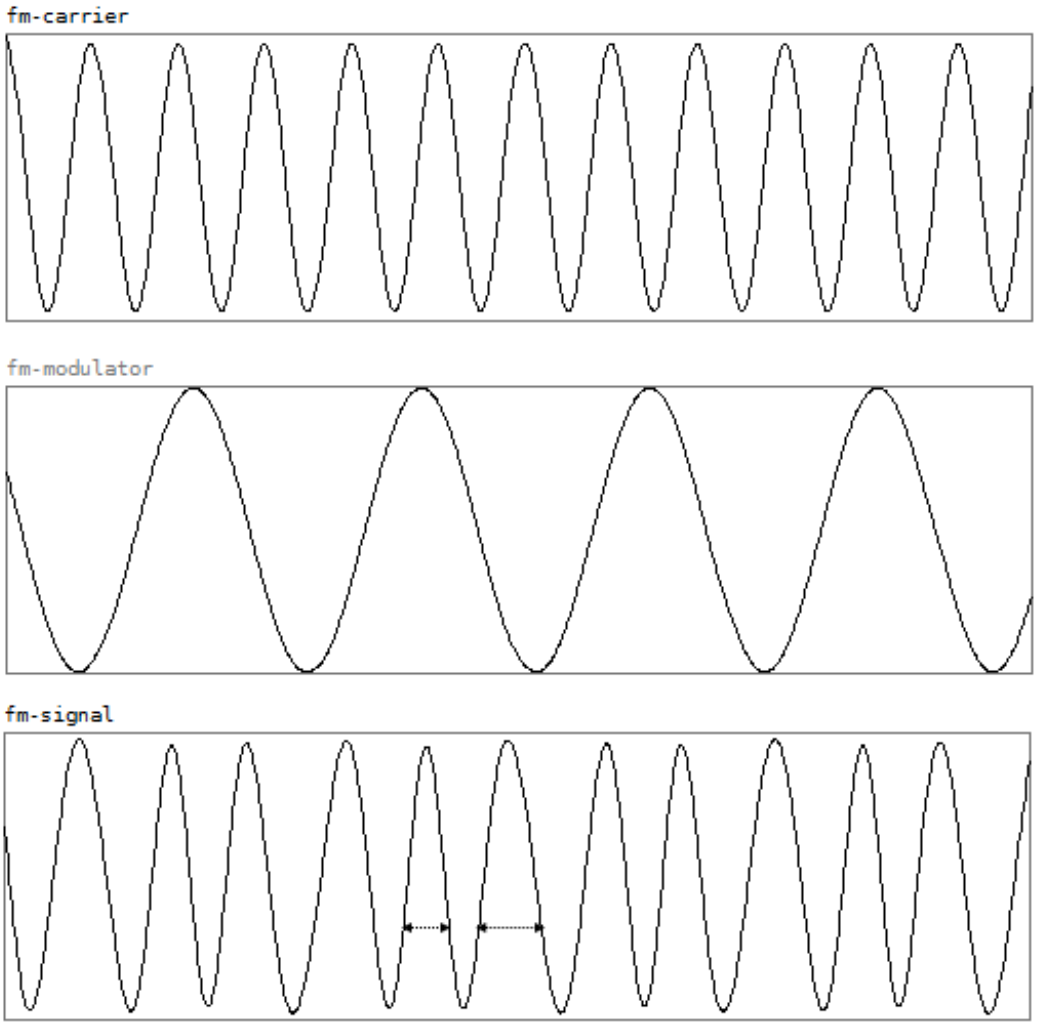


Figure 2.2: A single frequency sine wave modulating a higher frequency carrier signal. The variation of the frequency of the carrier signal is clearly seen in *fm-signal*.

applications [21]. Algorithms that can generate realistic sounds are typically much more complex.

A variety of software tools and languages exist for this purpose. The most common and known ones are Pure Data [22], Max [23], Bidule [24], SuperCollider [25] and CSound [26].

Pure Data is designed specifically for audio processing. Additionally it can handle OpenGL supported graphics [27] or random access full-motion video [28] by the help of miscellaneous extensions. Nonetheless, *signals* carrying audio data are allowed to be batch processed in Pure Data's runtime while control operations like *messages* are demanding much more computation resources because of the amount of excessive load from Pure Data's interpreter. Although Pure Data requires less amount of computational resources than its ancestors like *Max/MSP*, the programs which demand less computational resources are desirable because of the fact that smaller and inexpensive systems have very little computational ability. When this

bottleneck is considered, Pure Data's interpreter is decisive for the patch to be able to run in real time.

Max and Bidule have graphical user interfaces like Pure Data. They contains audio modules and units which can be wired each other using the interface. Although various sound synthesis algorithms can be implemented by these programs, they cannot be embedded into games since they run on the interpreter of the program like Pure Data.

SuperCollider and CSound have their programming syntaxes like Faust. Actually, they are programing languages. Various sound synthesis algorithms can be implemented on these programs by writing code in the syntax of these programs. However, even though the sound algorithm is written in code, it runs on the interpreter as on the other tools. Hence, the designed synthesis algorithms and procedural sounds cannot be embedded into games directly.

2.4 Related Work

There are three significant projects similar to what is presented in this thesis: *PureData Anywhere*, *PuDaC* and *libpd*.

2.4.1 PureData Anywhere

In order to decrease the overhead of Pure Data's interpreter on embedded systems, there is a project called PureData Anywhere [29]. It is a clone of Pure Data running only on Personal Digital Assistants (PDA's) on which Linux installations are supported. Since embedded systems have the drawback of not having floating point hardware, PureData Anywhere has a disadvantage. Since message logic uses floating point math which is emulated on the embedded system and the emulation uses many resources, applications which are specialized on controls rather than signals cannot perform as expected on the embedded systems.

2.4.2 PuDaC

In a thesis work [8], a compiler named PuDaC (PUre DAta Compiler) was created in order to address the same performance issues. The compiler considers the data as if it consists of two parts: high bandwidth signals (audio) and low bandwidth signals (control). The equivalent of each PD object in the patch is transformed into C language. Connections are written as function calls. As a result, the PD patch is transformed into a C program which can run efficiently on embedded systems having limited computational resources. However, the C program is not optimized specifically for audio processing in contrast to Faust generated C++ code.

2.4.3 libpd

libpd [30] is a free and open source project which enables the usage of *Pure Data* almost everywhere from embedded devices to phones and computers.

libpd is not a derivation of PD but it is the PD itself. Only difference is that it is embeddable and runs anywhere without any concern independent of hardware. If the system can run native code, than it can also run *libpd*.

PD is a powerful program which musicians, composers or developers use passionately. However, patches created on PD could not appear on the different systems having various configurations and operating systems. They were dependent to the PD interpreter. By the coming of *libpd*, PD is embedded into the systems. Hence, PD patches can be incorporated in games, Android or iOS apps, or programs written in C running on Linux embedded systems, or Mac, or Windows, or anywhere else that can run native code. *libpd* helps PD run in the application or game itself as a compiled program.

libpd has become very popular for these benefits. There is also a book about *libpd* which tells making mobile applications with it [31]. Additionally, various mobile applications have been made available on the iTunes App Store and Google Play Store [32, 33, 34].

CHAPTER 3

TRANSPILER DESIGN

Mephisto is developed in this thesis in order to enable Pure Data users to incorporate the audio algorithms they design into games and other applications by utilizing the highly optimized C++ code created by Faust. With this motivation, Mephisto transcompiles Pure Data sources into Faust sources and the creation of optimized C++ code is then left to the Faust compiler.

Transcompilation process consists of four steps: 1) parsing the PD source, 2) creating PD object tree and traversing it, 3) creating Faust source on-the-fly while traversing the tree, and 4) compiling the transpiler-generated Faust code to obtain optimized C++ code. A visual representation is shown in Fig. 3.1.

3.1 Parsing

ANTLR v4 is a powerful parser generator used to read, process, execute, or translate structured text such as program source code, data, and configuration files [35]. ANTLR v4 also has the capability to carry out lexical analysis, token generation and parse tree creation. Given these advantages that simplify and integrate the work-flow, we implemented the Pure Data parser with ANTLR v4.

Each object positioned on the canvas of a PD patch is represented as a row in the source file. Each row contains its object definition and parameters. By using the format specification and ANTLR v4, a formal language description, i.e. the *grammar* of Pure Data language, has been created (see Appendix C). This grammar is used by ANTLR in generating a Java program that reads the PD source files and builds a data structure representing the input also known as a *parse tree*. The row nodes of an example parse tree are shown in Fig. 3.2. Additionally, it automatically generates *tree walkers* (listeners) that can be used to visit the nodes of this tree to create the PD object tree. A parse tree listener interface consists of simple event listeners triggered by the built-in tree walker [35]. ANTLR v4 generates enter and exit methods for each node in the parse tree. Enter event is triggered when tree walker enters a node. Exit event is triggered when tree walker completes traversing the node's children and leaves that node. By using these listeners, current node is determined and can be processed based on its context. Details of the file format specification of Pure Data source used by Mephisto can be found in PD's distribution site [36].

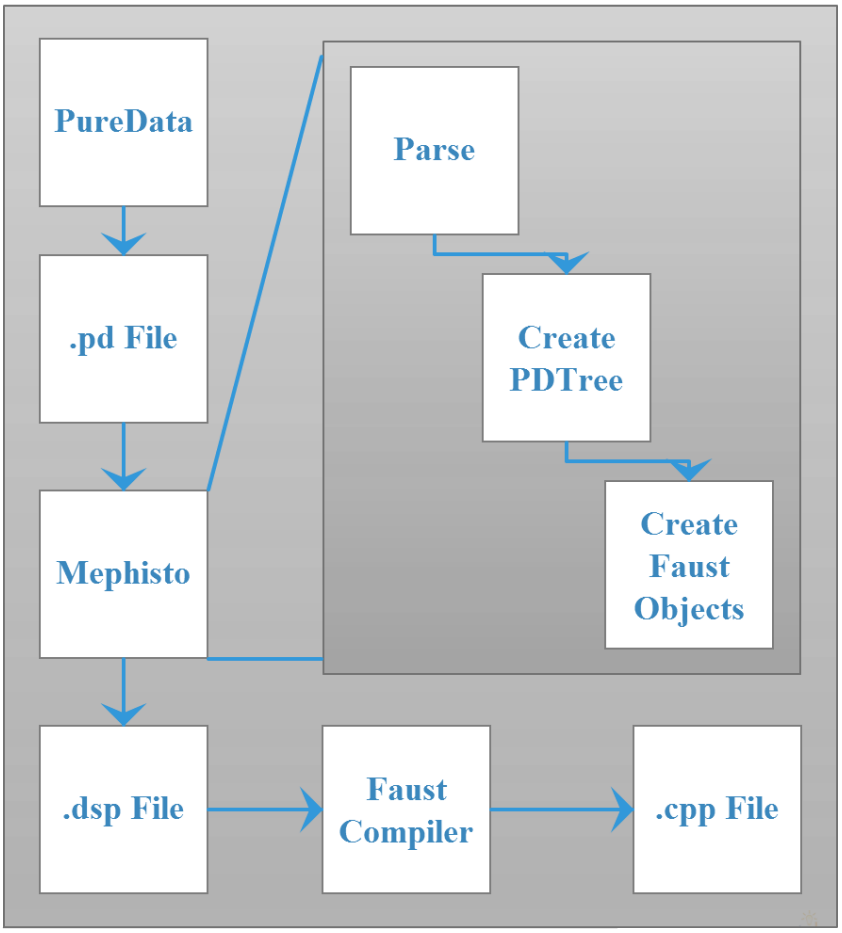


Figure 3.1: Transpiling pipeline of Mephisto

3.2 Traversal of the Parse Tree

In order to define the semantics of PD objects and the connections between them, we have created a data structure so that it becomes the nodes of a tree which will be called *PDTree*. The class diagram of the data structure is seen in Figs. 3.3a and 3.3b.

`Pair` class represents an outgoing connection of a PD object. `objectNumber` represents unique ID (`UID`) of that PD object and `outletNumber` represents which outlet of that PD object belongs to the referred connection.

`PDObject` class represents a PD object itself. `defaultVal` keeps the default value of PD object statically set in the PD patch. For PD objects with multiple initialization arguments, `args` array is used instead of the `defaultVal` attribute. The most important part of this class is `objectInlets` attribute which is a hash map whose keys are representing inlets of the PD object. Each key holds a `Pair` instance. For example, index `0` holding a `Pair` object which is `2,0` means that the first outlet of the PD object whose `UID` is `2` is connected to the first inlet of current PD object. This is repeated for each index. The `outputs` attribute represents the current PD object's outlets. Each key refers to each outlet mapping the key `0` to outlet `1` and so on. Fig. 3.4 illustrates the data structure and its usage.

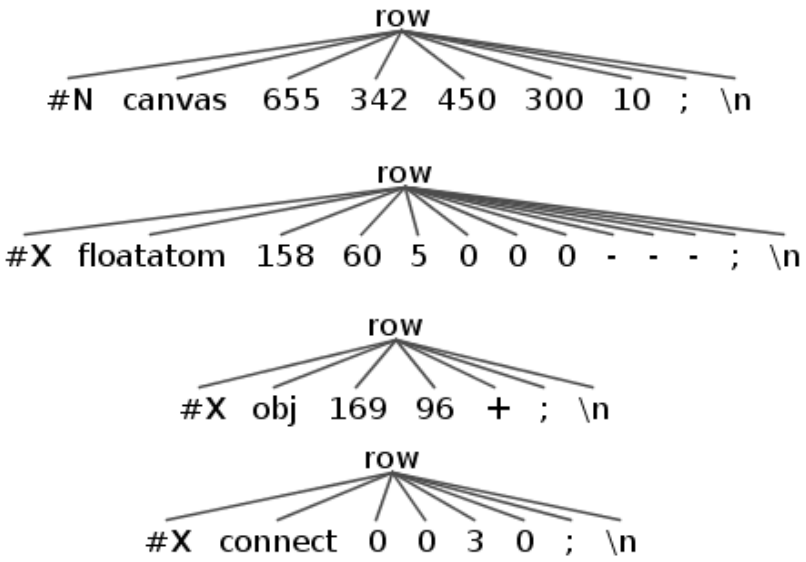


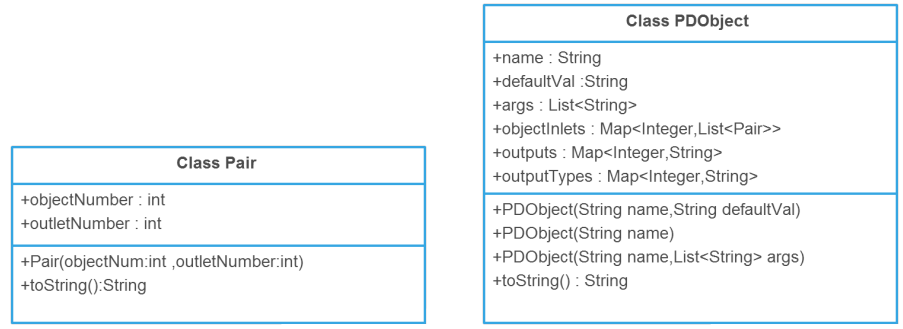
Figure 3.2: Each row represents a PD object. Each element in a row represents arguments of that object. All tokens are generated from PD source in Mephisto's parsing stage

While the tree walker walks on the ANTLR v4 generated parse tree, it dispatches listener events. We use auto-generated listeners which are `enterRow` and `exitFile`. `PDObject` instances are created in `enterRow` callbacks. `enterRow` callbacks are called with a `context` argument keeping the children of that row node in the parse tree. Since all arguments and parameters which are children of that node are found in the `context`, a `PDObject` with a unique ID is easily created without explicitly traversing its children. After the creation of a `PDObject` instance, it is pushed to a global list in order to be accessed later. Additionally, if `enterRow` callback is called for a connection token which is represented as `connect` in PD source, a `Pair` instance is created representing a connection and inserted into corresponding `PDObject`'s hash-map.

At the end of the traversal of the parse tree, `PDTree` is created. The `dac ~` object in the PD patch becomes the root of the `PDTree`. By the end of the traversal, `exitFile` callback function is called. The second traversal is started in this callback function beginning from the root of the `PDTree` and following depth-first search (see Fig. 3.5)

3.3 Traversal of `PDTree` and Creation of Faust Code

The second traversal is started on `dac ~` object on `PDTree` by calling `exitFile` callback. At first, the hash-map `objectInlets` of the `PDObject`, whose name is `dac ~`, is scanned. First key is `0` which represents the first inlet of the object. Additionally, the value of the key is a `Pair` object which represents connection



(a) Pair Class representing connections in a PD patch

(b) PDObject Class representing PD objects in a PD patch

Figure 3.3: PDObject and Pair classes

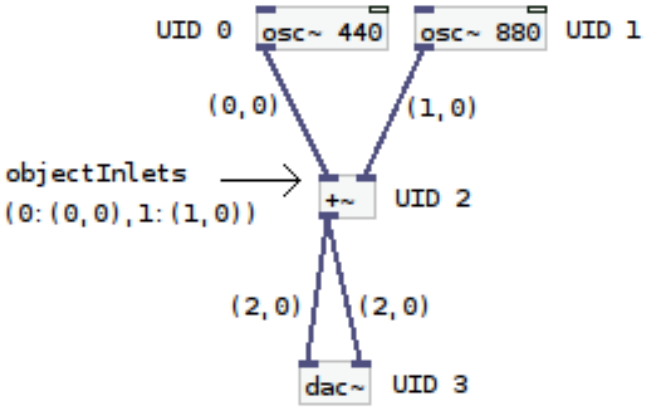


Figure 3.4: Visualization of usage of Pair Class and PDObject Class instances

coming from outlet of another `PDObject` to this inlet. Traversal is carried out by recursion with a function defined as `createObject_setOutput(int objectNumber,int outletNumber)`. After giving the value (`Pair`) of the key `0` to this function as `<Pair instance>.objectNumber` and `<Pair instance>.outletNumber`, it calculates its output recursively, sets the `outputs` and `outputTypes` of that object and finally returns the Faust equivalent outlet value of that `PDObject`. This process is applied to each key in the `objectInlets` attribute of each `PDObject`.

Since `dac ~` object in Pure Data outputs signals coming in its inlets directly, it is transpiled to `process` which is the entry point of generated Faust program. Inlets of the `dac ~` object are mapped to different parallel channels in Faust program. When all objects are traversed, transpilation is completed. As a result, a Faust source file is created.

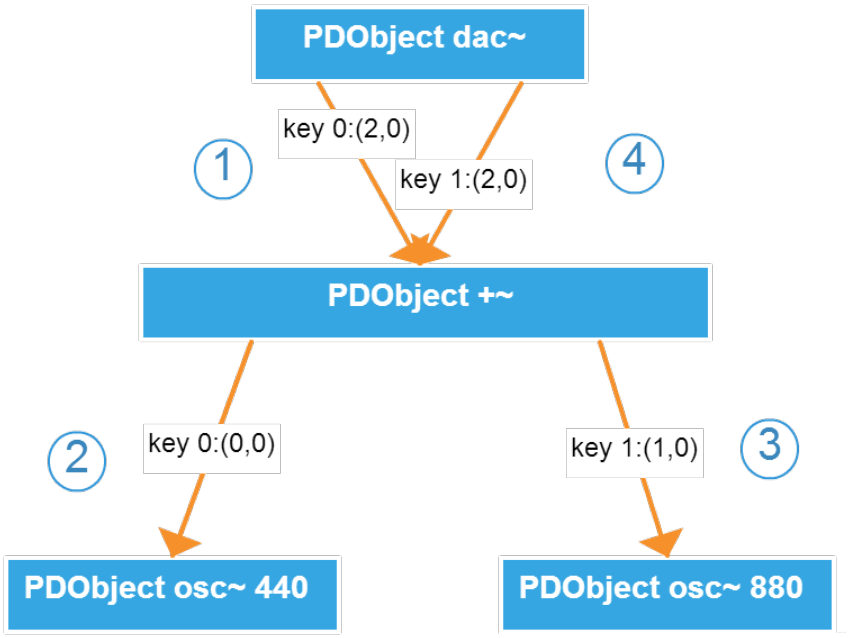


Figure 3.5: PDTree and its traversal. The numbers represent traversal order

3.3.1 createObject_setOutput(int objectNumber,int outletNumber) Function

The objective of this function is to create the Faust equivalents of PD objects. `objectNumber` argument defines `UID` of the `PDObject` stored in the mentioned global list. `outletNumber` defines which outlet value of the object is returned. First of all, function gets `PDObject` from the list by its `objectNumber`. After the object is obtained, its name is checked. Based on the object's name, the function calculates values coming from other connected objects into the object inlets by using the function itself recursively. After getting all incoming values for each inlet, it creates a Faust function definition based on its semantics whose arguments consist of these inlet values. Once the function definition is created, the function sets the object's outputs. Since Faust is a functional programming language, the defined function's name called with its arguments is set and returned as an output.

3.4 Compiling transpiler-generated Faust code

On a system where Faust compiler is installed, the resulting `.dsp` file is compiled and a `.cpp` file is generated. As a result, a well optimized C++ code is created without writing any lines of code using the presented transpiler, Mephisto. The generated C++ code can be embedded in a rather straightforward way in any desired project written in C++.

CHAPTER 4

EXAMPLES

In this chapter, usage of Mephisto on three examples retrieved from Andy Farnell's "Designing Sound" book [2] will be demonstrated. Design decisions and patterns of the sound synthesis patches are outside the scope of this thesis and will not be discussed. Instead, Mephisto transpilation process will be explained, intermediate code compilation by Faust will be discussed. The performance comparisons between the applications which uses Mephisto created code, the ones which run with *libpd* and on Pure Data are going to be presented in evaluation Chapter 5.

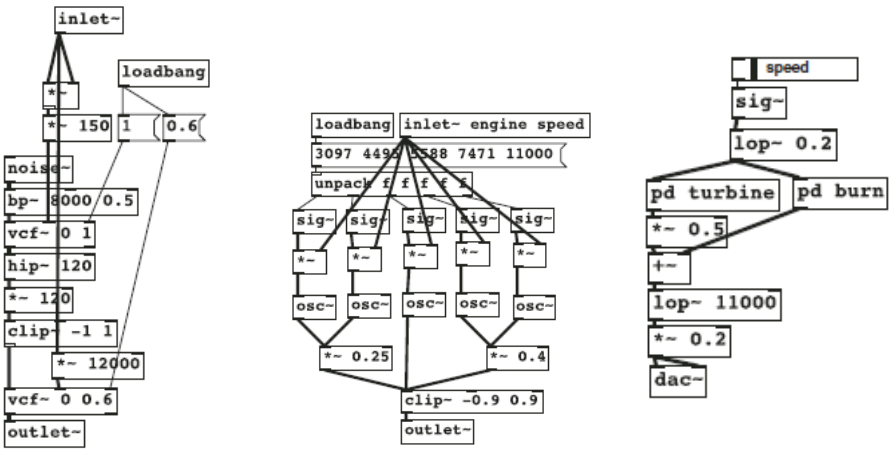
4.1 Jet Engine

In this patch the sound of a jet plane's engine is produced. It simulates a small aircraft's engine of the turbofan type. Although it does not emulate a real jet engine of military fighter jets, it demonstrates broad characteristics of a simple engine design. Military type jet engines also have much in common with this design.

This patch is selected since 1) it represents a real use-case scenario (e.g. in a flight simulator) and 2) Faust is well-suited and specialized for signal processing rather than control heavy applications as stated in Section 1.2. Additionally, Mephisto has limited capability to transpile patches including mostly control objects.

Pure Data implementation of jet engine consists of one patch and two sub-patches. One of the sub-patches represents the *forced flame* of the engine (Fig. 4.1a) while the other sub-patch represents the *turbine* of the engine (Fig. 4.1b). The *main* patch uses these two sub-patches and combines them with the speed control and low pass filters in Fig. 4.1c.

In order to transpile the *main* patch via Mephisto, it should first be flattened. Firstly, let us transform the *forced flame* patch. Since Mephisto does not implement the `loadbang` object in PD, it should be removed from the patch. When `loadbang` object is removed, two message objects which are `1` and `0.6` become open-ended. These message objects should be clicked manually by the user in the flattened patch to make it run correctly on PD. Considering Mephisto created Faust code, these two messages are going to be represented as check boxes. Then, these boxes are going to be interchanged with the control codes from the game or event triggers when it is being embedded to the game as C++ code.



(a) Forced Flame Patch (b) Turbine Patch (c) The *main* patch of jet engine

Figure 4.1: All patches of the jet engine patch

`inlet ~` and `outlet ~` objects are also removed since the flattened patch is no longer going to need these. In the *main* patch, the outlet of the `lop ~ 0.2` object is coming into the inlet of the forced flame patch. Since we have removed the `inlet ~` object in the *forced flame* patch, we should connect inlets of each object which are fed by the `inlet ~` object with the `lop ~ 0.2` object's outlet. Additionally, same process should be applied for the `outlet ~` object. The connections coming into the inlet of this object should be connected to inlets of the objects in the *main* patch which are connected to the outlet of the *forced flame* sub-patch. To illustrate, the outlet of the `vcf ~ 0 0.6` object in the *forced flame* sub-patch should be connected to the second inlet of the `+ ~` object in the *main* patch.

Secondly, let us transform the *turbine* sub-patch. Here, there are three objects which will be considered. These are `loadbang` , `inlet ~ engine speed` and `outlet ~` objects. The reason why `loadbang` object is removed is obvious. Remaining `3097 4495 5588 7471 11000` message object should be clicked manually by the user when the flattened patch runs.

In the process of removal of `inlet ~ engine speed` object, all `* ~` objects which are fed by its outlet should be connected with the `lop ~ 0.2` object's outlet in the *main* patch. Similarly, the outlet of the `clip ~ −0.9 0.9` object should be connected to the inlet of the `* ~ 0.5` object in the *main* patch. Finally, the slider which is used to set engine speed should be replaced with a number object since Mephisto does not support PD sliders. Fig. 4.2 represents the flattened patch.

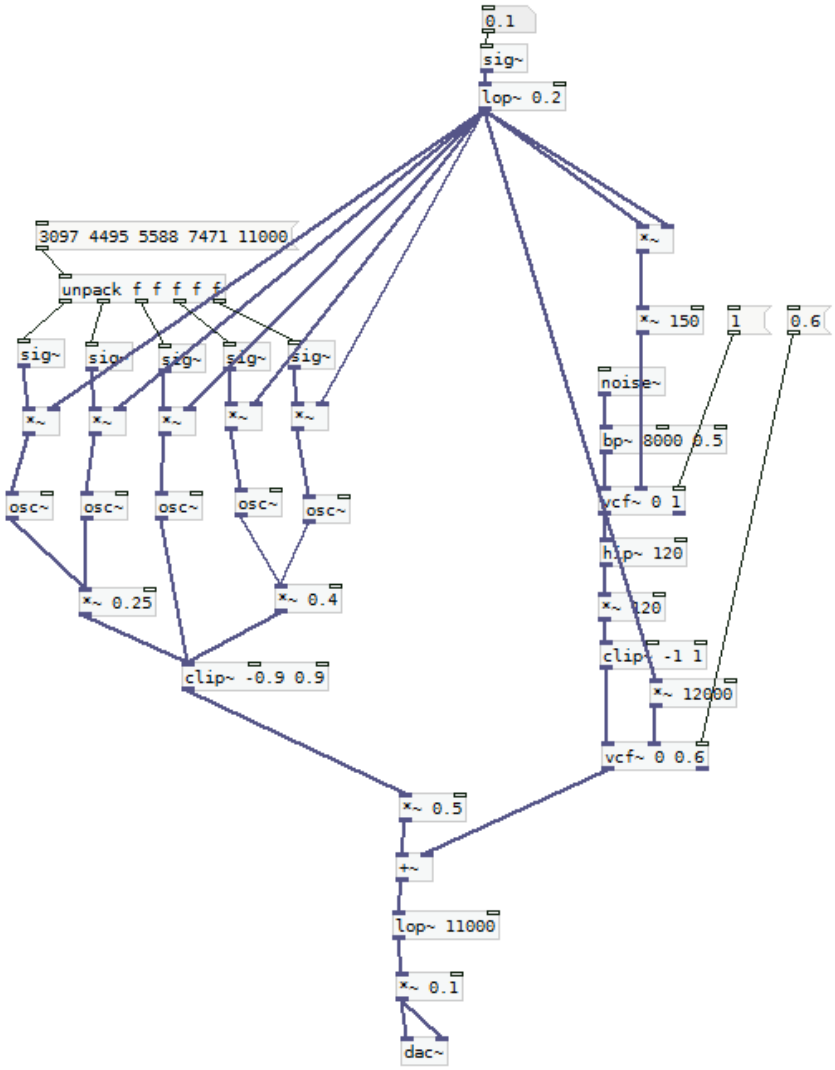
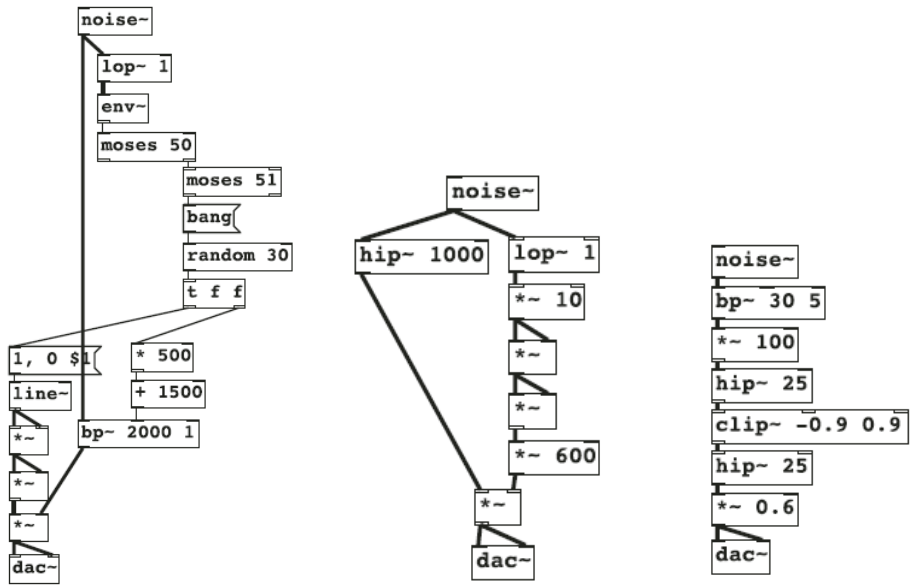


Figure 4.2: Flattened jet engine patch

When this patch is transpiled with Mephisto, a *.dsp* file which is written in Faust is created. Then, the mission of Mephisto is accomplished at this point. Afterwards, the *.dsp* file is compiled by the highly optimized Faust compiler. After compilation, *C++* code is generated. The generated Faust code can be seen in Appendix B.

4.2 Fire

In this example, the process of transpilation of a Pure Data patch which synthesizes natural fire sound is going to be presented. In this section, only the flattening process and adaptation for Mephisto are going to be discussed.



(a) Crackling subpatch of the fire patch (b) Hissing subpatch of the fire patch (c) Lapping subpatch of the fire patch

Figure 4.3: All sub-patches of the fire patch

Fire patch consists of 3 sub-patches emulating the sound of a log fire. Fig. 4.3a represents the crackling effect of the log fire which is small scale explosions caused by stress in the burning log. Fig. 4.3b represents the hissing effect of the fire which is caused by out-gassing. Fig. 4.3c represents the lapping effect of the fire which emulates the sound of the combustion process. Fig. 4.4 displays the main patch of the fire consisting of the combined sub-patches.

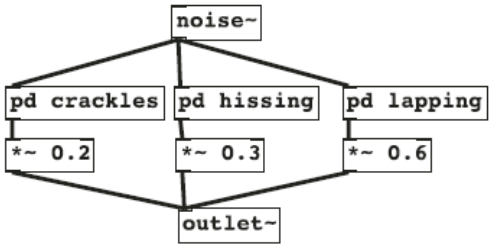


Figure 4.4: The main patch of the fire

Let us examine the crackling subpatch. It contains control objects as well as signal objects. Since `moses`, `bang`, `env~` and `random` objects are not implemented in Mephisto, this subpatch should be skipped in the flattening process. Skipping does not affect the synthesis since it is an additive effect.

Let us consider hissing subpatch. When the main patch is examined, it is obvious that two operations are needed in order to flatten this subpatch. Firstly, the `noise~` object in the subpatch should be removed and the inlets of the objects that are connected to the outlet of the removed `noise~` object should be connected to main

patch's `noise ~` object. Secondly, the `dac ~` object should be removed in the subpatch and the `* ~` object's outlet should be connected to the `* ~ 0.3` object's inlet of the main patch.

Lastly, let us consider the lapping subpatch. The same operations as in the hissing subpatch should be applied to this patch in order to create the flattened patch.

In order to create a more realistic fire sound, a patch is created in [2] using four instances of the fire patch with different kinds of filters. Fig. 4.5 represents the patch. Finally, when the outlet of the fire patch is connected to all these filters, the flattened patch which is desired for Mephisto is created as seen in the Fig. 4.6. After this point, the flattened patch can be compiled with Mephisto and a *.dsp* file can be obtained for further compiling with Faust in order to create *C++* code which can be embedded into the desired project.

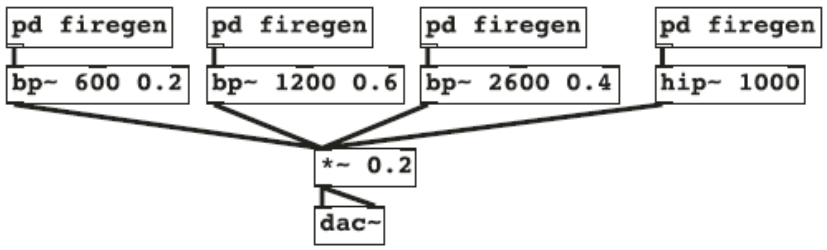


Figure 4.5: The 4-instance patch of the fire

4.3 Wind

In most of the games, when outside is considered, sound of wind is essential. In order to achieve a good quality wind sound, long recordings should be embedded into games. However, these recording occupy considerable amount of memory. If short recordings are embedded, this time the sound becomes repetitive. Additionally, it is really hard to obtain a good quality wind record. Wind sound can be synthesized procedurally with very low cost.

In this section, a patch which procedurally creates wind sound is presented. The complete patch consists of four parts. First patch uses a very low frequency (0.1 Hz) oscillator to set wind speed as seen in Fig. 4.7a. This patch uses two more patches. One is used in creating the gust and the other one is used in creating the squall components of wind as seen in Figs. 4.7b and 4.7c, respectively. Finally, a patch combines all of these patches by adding some static wind noise as seen in Fig. 4.8.

Considering these patches, it is necessary to flatten all of them into one patch in order to transpile with Mephisto. Firstly, let us consider the wind gust patch. The `inlet ~` object is removed. Then, `+ ~ 0.5` object's inlet is connected to the outlet of the `* ~ 0.25` object in the wind speed patch. Then, `outlet ~` object is removed and `* ~` object's outlet is connected to the inlet of the `+ ~` object in the wind speed

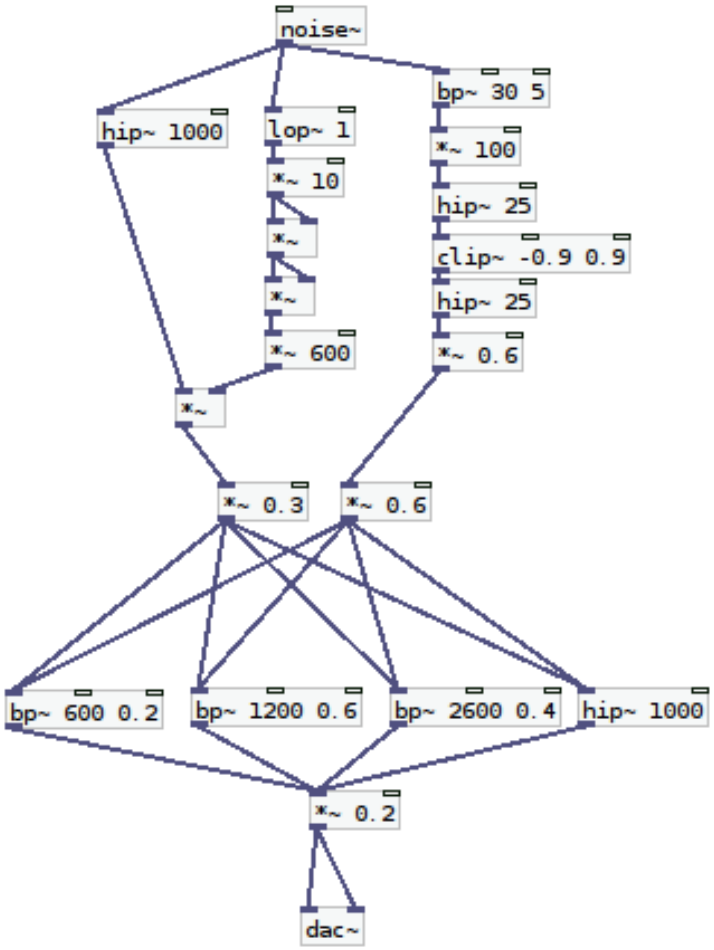
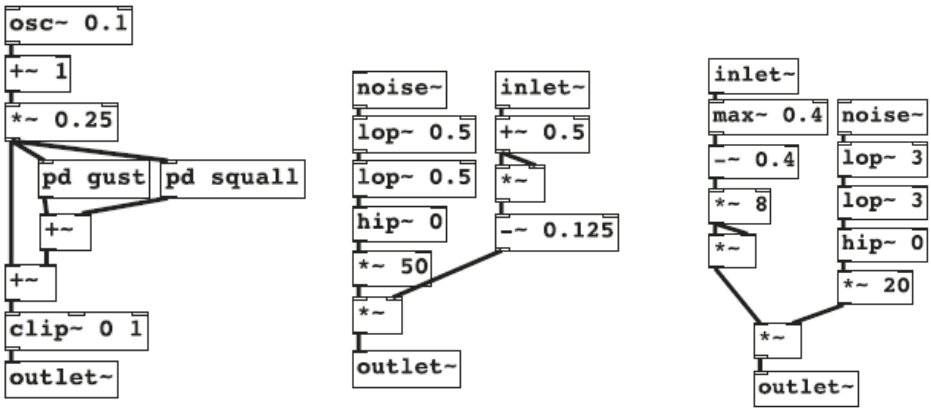


Figure 4.6: The flattened patch of the fire

patch which is right after the `pd gust` object. To continue, let us consider the wind squall patch. Here the same operations should be applied in order to flatten this patch. Remove the `inlet ~` and `outlet ~` objects and connect the open ended inlets and outlets accordingly.

In Fig. 4.8, some objects are not recognized by Mephisto. These are the `s ~`, `r ~`, `catch ~`, `vd ~`, `delwrite ~` and `throw ~` objects. In order to flatten this patch, the semantics of the unsupported objects should be known. First of all, `s ~` and `r ~` objects work as sender and receiver, respectively. They provide readability to patches. This means that, the connection coming to the inlet of the `s ~ white` object can be connected to the inlet of the `bp ~ 800 1` object which is connected to the outlet of the `r ~ white` object directly. In the same way, `throw ~` and `catch ~` objects can be connected. Since Mephisto does not transpile delay objects and delays are not critically important for this patch, they can be ignored. Hence, the connection coming to the inlet of the `delwrite ~ a 3000` object can be connected to the inlet of the `+ ~ 0.2` object which is connected to the outlet of the `vd ~ a 0` object. Finally, `fcpan ~ 0.51` object can be ignored since panning is out of consideration. Fully integrated and flattened patch can be seen in Fig. 4.9.



(a) The wind speed patch (b) The wind gust patch (c) The wind squall patch

Figure 4.7: All patches which construct the wind sound

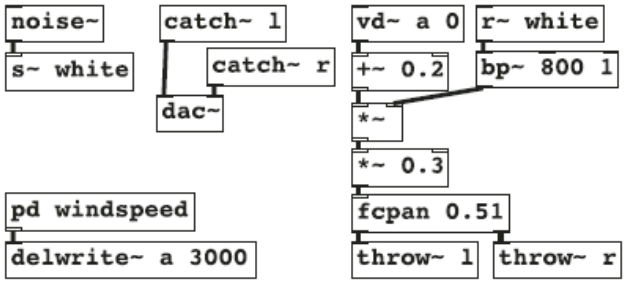


Figure 4.8: The static noise used with wind

After this point, the flattened patch can be compiled with Mephisto and a *.dsp* file of it can be obtained for further compiling with Faust in order to create a *C++* code which can be embedded into desired project.

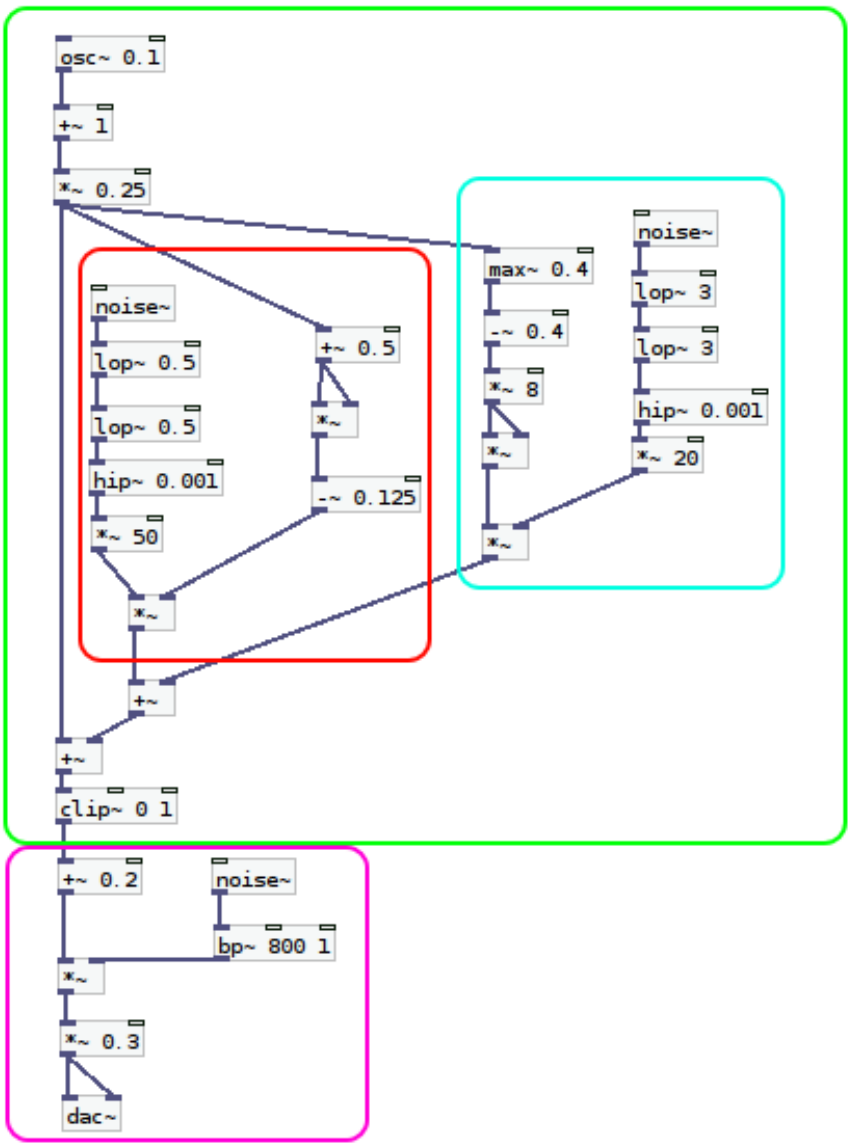


Figure 4.9: The flattened wind patch. Red, cyan, green and purple boxes represent wind gust, wind squall, wind speed and static noise patches respectively.

CHAPTER 5

EVALUATION

5.1 Conventions and Limitations

There are significant differences between Pure Data and Faust. Mephisto uses certain conventions to reconcile these differences. At its present version, Mephisto also has certain limitations. These conventions and limitations are listed below:

1. Mephisto ignores the *cold inlet* mechanism in Pure Data. Since Faust is specialized on audio signal processing, everything is considered as a signal and there is no messaging or control mechanism except for elementary user interfaces. Hence transpiling process ignores the cold inlet semantic of PD while adapting it into Faust code.

2. Mephisto does not transpile PD patches unless the patch includes a `dac ~` object. PD patch should be formed as a connected tree [37] in which there is a `dac ~` object since Mephisto transpiles PD patches by constructing a `PDTree` (Sec. 3.2) whose root is a `dac ~` object. Objects which are not included in this connected tree are ignored by Mephisto.

3. Mephisto's main aim is to transpile PD patches focused on signal blocks and control blocks are not covered as stated. However, simple control mechanisms like number box, message, `trigger`, `pack` and `unpack` objects are implemented. In addition, mathematical operators and logical operators in both control and signal blocks in PD are also implemented in Mephisto by converting them to signal blocks since both messages and signals in PD are transpiled into signals in Faust. Hence, all control logic except for elementary controls should be implemented separately in C++ (or any other programming language that Faust would support in the future) in which the transpiled code will be embedded. Consider the patch in Fig. 5.1, for example. The patch synthesizes the standard CCITT dialing tone [38]. It consists of nearly all signal objects except for two control objects which are messages that control starting and stopping the generated output. Those are transpiled into check-boxes in Faust code in order to preserve content integrity. However, these control statements should be properly implemented and connected to (possibly event-driven) control code in which they will be embedded. In other words, controls shown as two message boxes in the patch should be implemented manually. Faust code automatically generated for the CCITT patch (Fig. 5.1)

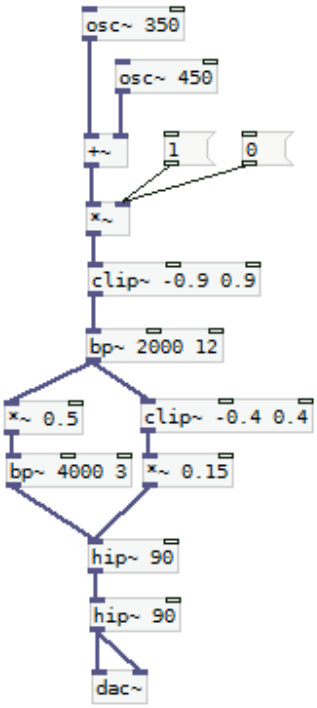


Figure 5.1: Dialing tone patch synthesizing the standard CCITT dialing tone

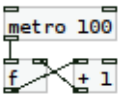
by Mephisto as well as the block diagram view of the same Faust code are given in the Appendix A.

4. Mephisto does not yet support sub-patches and external objects as it is designed to parse only one PD source. Allowable objects are predefined in transpiler and objects other than these are not recognized (see Appendix D). Therefore, PD projects that include sub-patches or external objects should be flattened to a single PD patch.

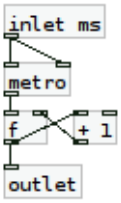
5. Fig. 5.2a shows the regular convention to generate a counter in Pure Data which is widely used in synthesis algorithms. However, Mephisto does not recognize the `float` object in PD. In addition, the structure of the counter violates the tree structure of `PDTree` since it has a cycle. Since trees are graphs which do not include cycles according to graph theory and cycles will cause Mephisto to enter infinite recursion, counters cannot be created in the conventional way. In order to alleviate this problem, Mephisto comes with the special PD patch, `fcounter`, which is an abstraction of Fig. 5.2b. PD programmers should use this patch as a counter instead of the conventional one.

5.2 Performance

Performance of three different ways to execute algorithms designed in Pure Data are compared. Specifically, three different conditions were tested in the benchmark: 1) Pure Data natively running a patch, 2) the same patch executed via the embeddable



(a) Conventional counter used in most of the PD patches



(b) Mephisto Fcounter abstraction sub-patch. Used instead of conventional counter

Figure 5.2: Conventional counter vs Mephisto FCounter

Pure Data library *libpd* [30] and 3) C++ code generated by Faust and Mephisto compiled with JACK [39]. The test environment was on a virtual machine which had 1 processor and 4096 MB memory and Ubuntu (32 bit) operating system. The host machine had Intel Core i7-3630QM 2.40 GHz processor, 8 GB memory and Windows 8.1. In each case the relevant process was isolated and the average CPU utilization was obtained by the profiler, Audria [40]. The patch that was used in benchmarking these three cases consists of the product of the outputs of two oscillators for a duration of 30 s. The duration that the process was sampled was 35 s in each case. At the sampling rate of 44.1 kHz, this amounts to the generation of 2,646,000 floating point numbers and 1,323,000 floating point multiplications in total. This patch was chosen since it had the smallest complexity when it was compared with the ones in Chapter 4, *Examples*. The results showed that Mephisto creates highly optimized code which uses averagely 2% of CPU while running either minimalistic patches or complex patches. Figure 5.3 shows the average CPU utilization for each case of the benchmark. It may be observed that the highly optimized code generated by Mephisto and Faust outperforms libpd-based implementations and reaches the performance of the native Pure Data. In the case of Pure Data only, the average CPU utilization starts at zero and plateaus at slightly less than 1%. In the case of libpd, the same process results in around 20% utilization. For C++ code generated via Mephisto by Faust, the average utilization starts at 20% and quickly falls to just above 1%. These results indicate that generating C++ code by Faust via Mephisto is promising in terms of the potential performance gains that it can provide. Additionally, the performance tests of the examples in the Chapter 4 also show that the usage of the average CPU remains same even if the patch is much more complex. These data give Mephisto users the promise of a constant 2% CPU usage in their games so that they can synthesize their sounds without considering the limited resources of their devices. The performances of the examples are seen in figures 5.4, 5.5 and 5.6.

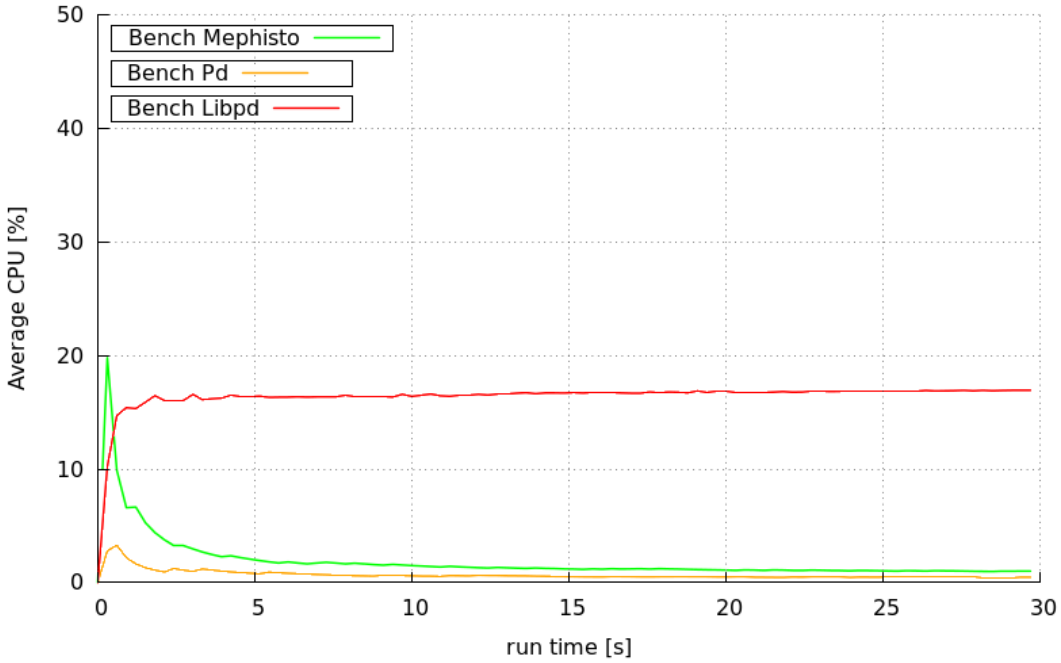


Figure 5.3: The average CPU utilization of the benchmark

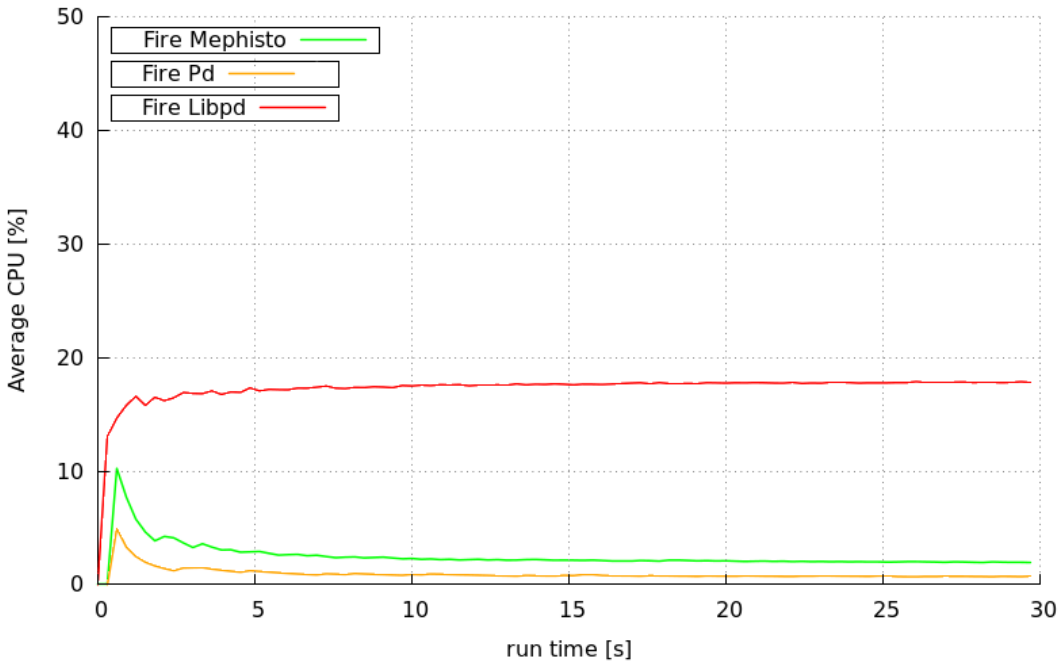


Figure 5.4: The average CPU utilization of the fire patch

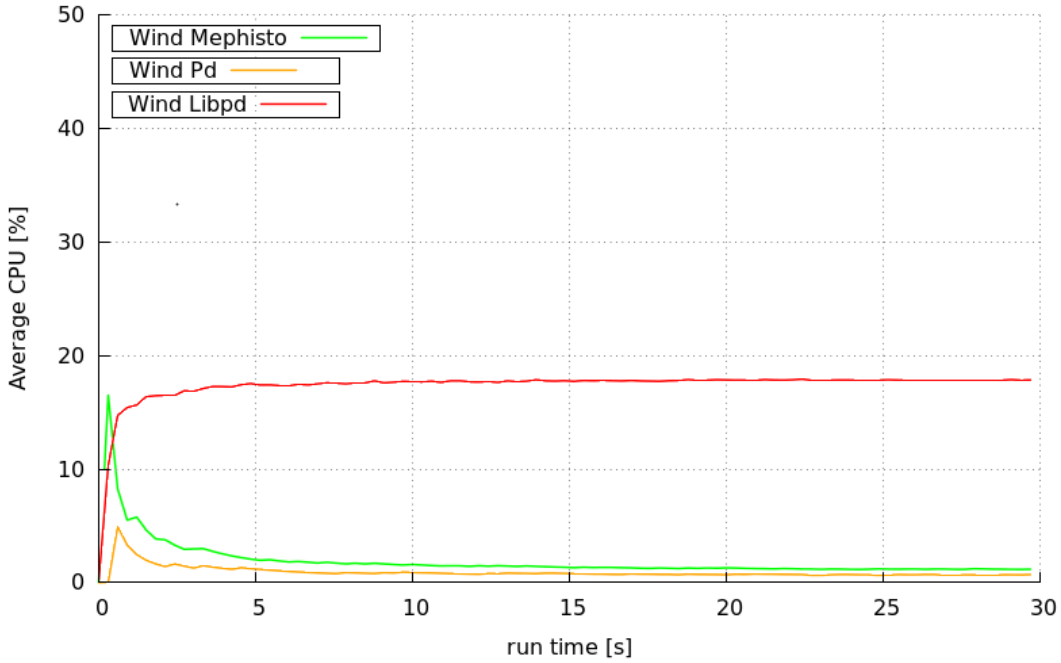


Figure 5.5: The average CPU utilization of the wind patch

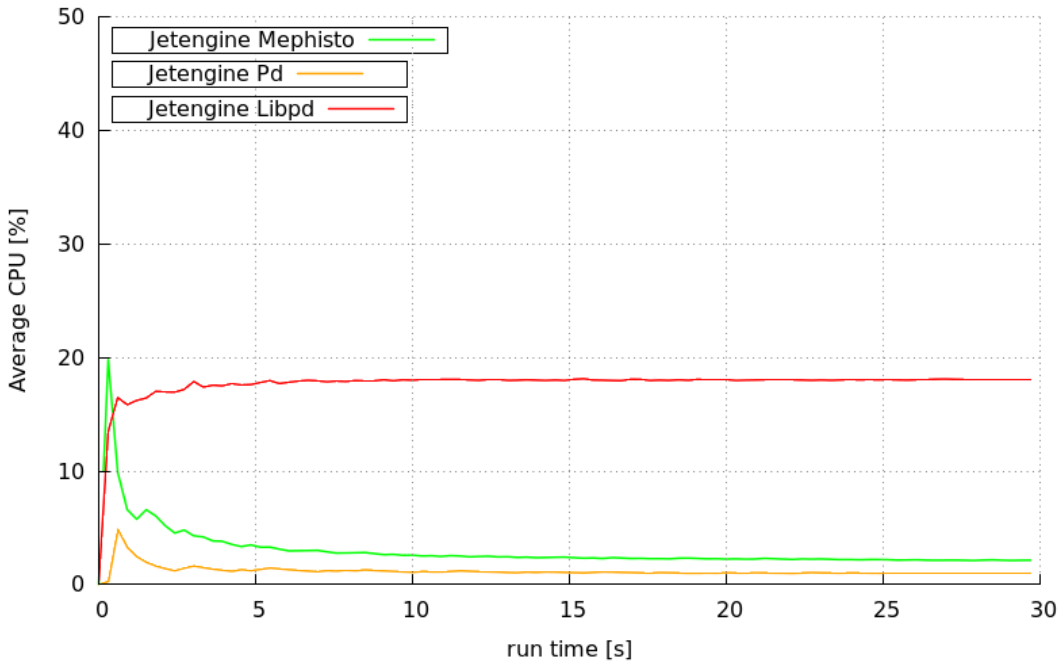


Figure 5.6: The average CPU utilization of the jet engine patch

CHAPTER 6

CONCLUSION

6.1 Future Work

Mephisto is at a stage that basic sound synthesis algorithms designed in Pure Data can be transpiled into Faust code in order to obtain highly optimized C++ code for use in games and other applications. However, the limitations outlined in the previous section remain to be solved.

Mephisto ignores `bang` messages and cold inlets. Since everything in Faust is considered as signals, it is hard to implement a single control message in Faust. A possible solution this problem can be developed by designating an impulse signal as a bang message. Consider a basic summation patch which adds two numbers in PD. One number is connected to hot inlet, the other one is connected to cold inlet. When the number connected to cold inlet is changed, the result in PD does not reflect the change. It waits for an implicit `bang` from the hot inlet. Whenever the number connected to the hot inlet is changed, the result reflects both changes. In Faust, the lack of messages means that both numbers will be treated as signals and there is no gating mechanism as in PD. The deficiency caused by Faust's lack of such control mechanisms could be filled by creating specialized Faust functions and revising the traversal mechanism.

PD patches which can be transpiled by Mephisto should include a `dac~` object and form a connected tree whose root is the `dac~` object. It is critical that the patch does not include any cyclic connections. However, Mephisto could be further developed that it could transpile PD patches containing disjoint or cyclic trees [41] with no constraints on the root objects.

In addition, PD abstractions for Faust code could be written for Mephisto so that it could deduce what the semantic of the abstraction is and does not care about its internal structure. Then a Faust implementation of that abstraction would be created and added to the source code of Mephisto. The example for `fcounter` can act as a starting point.

The work presented in this thesis acts as a proof-of-concept showing that transpilation from Pure Data to Faust is possible and thus the present version of Mephisto does not cover more than a few PD objects. Any additional object that is added to Mephisto would increase the palette of objects available to the PD programmer. For this reason,

Mephisto will be released with the GPL v3.0 license and made open source to allow other developers to develop it even further.

6.2 Conclusion

Pure Data is both a Turing complete programming language and a very useful tool for designing audio algorithms from scratch. However, since Pure Data is an interpreted (as opposed to compiled) language, it presents important performance issues. Therefore, once the prototype algorithms are designed, the designer or the developer still has to carry the burden of writing code to integrate the algorithm in a final standalone app such as a game or an audio app typically using another high-level language such as C++. While this approach is feasible, the effort required from the C++ programmer is considerable and a solution that can automatically generate C++ code from Pure Data code would be practically very beneficial. In order to address this problem, a transpiler from Pure Data to Faust, named Mephisto, is described in this thesis. While the Mephisto can generate Faust code from Pure Data patches, Faust is used as an intermediate language which already has a compiler that can generate C++ code.

REFERENCES

[1] D. Brandon Lloyd, Nikunj Raghuvanshi, and Naga K. Govindaraju. Sound synthesis for impact sounds in video games. In *Proceedings of the Symposium on Interactive 3D Graphics and Games 2011*. ACM, 2011.

[2] Andy Farnell. *Designing Sound*. MIT Press, Cambridge, MA, USA, 2010.

[3] Miller Puckette. Pure Data: Another integrated computer music environment. In *Proc. Second Intercollege Comp. Mus. Concerts*, pages 37–41, Tachikawa, Japan, 1996.

[4] Yann Orlarey, Dominique Fober, and Stéphane Letz. FAUST: an efficient functional approach to DSP programming. In *New Computational Paradigms for Computer Music*. Editions Delatour, Paris, France, 2009.

[5] Yann Orlarey, Dominique Fober, and Stéphane Letz. An algebra for block diagram languages. In *International Computer Music Conference*, Göteborg, Sweden, September 2002.

[6] Julius O Smith III. Audio signal processing in Faust, 2010.

[7] Yann Orlarey, Dominique Fober, and Stephane Letz. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9):623–632, 2004.

[8] Robert N Jacobs. *A Wireless Sensor-based Mobile Music Environment Compiled from a Graphical Language*. PhD thesis, MIT Media Lab, Cambridge, MA, USA, September 2007.

[9] Abdullah Onur Demir and Hüseyin Hacıhabiboğlu. MEPHISTO: A Source to Source Transpiler from Pure Data to Faust. In *Sound and Music Conference*, Maynooth, Ireland, July 2015.

[10] E. David, M. Mathews, and H. McDonald. Description and results of experiments with speech using digital computer simulation. In *Proceedings of the 1958 National Electronics Conference*, pages 766–775. New York: Institute of Radio Engineers, 1958.

[11] C. Roads. Interview with Max Mathews. *Computer Music Journal*, 4(4):15–22, 1980.

[12] P. Wood. Recollections with John Robinson Pierce. *Computer Music Journal*, 15(4):17–28, 1991.

[13] Curtis Roads. *The computer music tutorial*. MIT press, 1996.

[14] N. Guttman. Personal Communication. 1980.

[15] M Mathews and Lawrence Rosler. Graphical language for the scores of computer-generated sounds. *Music by computers*, pages 84–114, 1969.

[16] Piotr Kleczkowski. Group additive synthesis. *Computer Music Journal*, pages 12–20, 1989.

[17] Vesa Valimaki and Antti Huovilainen. Antialiasing oscillators in subtractive synthesis. *Signal Processing Magazine, IEEE*, 24(2):116–125, 2007.

[18] Harold S Black. *Modulation theory*. van Nostrand, 1953.

[19] James Dashow. Looking into sequence symbols. *Perspectives of New Music*, pages 108–137, 1987.

[20] AP Godse UA Bakshi. *Communication Engineering*. Technical Publications, 2009.

[21] Mark Grimshaw. *Game Sound Technology and Player Interaction: Concepts and Developments: Concepts and Developments*. IGI global, 2010.

[22] Miller Puckette. Pure data. In *In Proceedings of the International Computer Music Conference*, pages 269–272. International Computer Music Association, 1996.

[23] Max. [Online]. Available:`https://cycling74.com/products/max/`.

[24] Bidule. [Online]. Available:`http://www.plogue.com/products/bidule/`.

[25] SuperCollider. [Online]. Available:`http://www.audiosynth.com/`.

[26] CSound. [Online]. Available:`https://csound.github.io/about.html`.

[27] GEM. [Online]. Available:`http://gem.iem.at/`.

[28] PDP. [Online]. Available:https://puredata.info/downloads/pdp.

[29] Günter Geiger. Pda: Real time signal processing and sound generation on handheld devices. In *International Computer Music Conference (ICMC)*, 2003.

[30] libpd. [Online]. Available:http://libpd.cc/about/.

[31] Peter Brinkmann. *Making Musical Apps: Real-time audio synthesis on Android and iOS*. O'Reilly, 2012.

[32] Sonaur. [Online]. Available:http://libpd.cc/projects/sonaur/.

[33] CloudSynth SoundCloud Synth. [Online]. Available:http://libpd.cc/projects/cloudsynth-soundcloud-synth/.

[34] Nodebeat. [Online]. Available:http://libpd.cc/projects/nodebeat/.

[35] Terence Parr. *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf, 2012.

[36] Unofficial PD v0.37 fileformat specification, October 2004.

[37] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*. Macmillan London, 1976.

[38] International Telecommunications Union (ITU). *Application of tones and recorded announcements in telephone services*. CCITT Recommendation E.182, 1998.

[39] JACK Audio Connection Kit. [Online]. Available:http://jackaudio.org/.

[40] audria - A Utility for Detailed Resource Inspection of Applications. [Online]. Available:https://github.com/scaidermern/audria.

[41] Martin Nilsson and Hidehiko Tanaka. Cyclic tree traversal. In *Third International Conference on Logic Programming*, pages 593–599. Springer, 1986.

APPENDIX A

MEPHISTO GENERATED FAUST CODE AND ITS BLOCK DIAGRAMS

A.1 Mephisto-generated Faust Code for the CCITT dialling tone patch

CCITT dialling tone patch was given as an example earlier in the thesis in Fig. 5.1. Faust code automatically generated by Mephisto for this patch is:

```
CCITT.dsp
import("music.lib");
import("math.lib");
import("filter.lib");

osc1=osc(350);
osc2=osc(450);
checkbox3=checkbox("1");
msg3 = checkbox3 * 1;
checkbox4=checkbox("0");
msg4 = checkbox4 * 0;
clip7(s) = if (s < (-0.9), (-0.9) , if ( s>0.9 ,0.9 ,s)
);
resonbp8=clip7(((((osc1+osc2))*((msg3)+(msg4)))))
:resonbp(2000,12,1);
clip10(s) = if (s < (-0.4), (-0.4) , if ( s>0.4 ,0.4 ,s)
);
resonbp11=((resonbp8)*0.5):resonbp(4000,3,1);
resonhp13=(resonbp11+((clip10((resonbp8)))*0.15))
:highpass(1,90);
resonhp14=(resonhp13):highpass(1,90);
process=resonhp14,resonhp14;
```

A.2 Mephisto-generated Faust Code's Block Diagrams for the CCITT dialling tone patch

Figures below show the different block diagrams generated for the same Pure Data patch. The figures were generated using FaustWorks IDE. The figures follow the tree traversal order as explained in the text, starting with the `dac~` object and following through to the `osc~` objects and the message boxes (emulated using checkboxes in Mephisto-generated Faust code.).

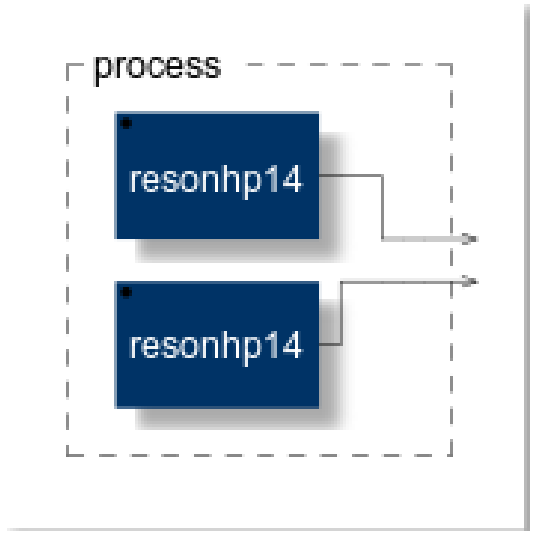


Figure A.1: Equivalent of "dac~" object

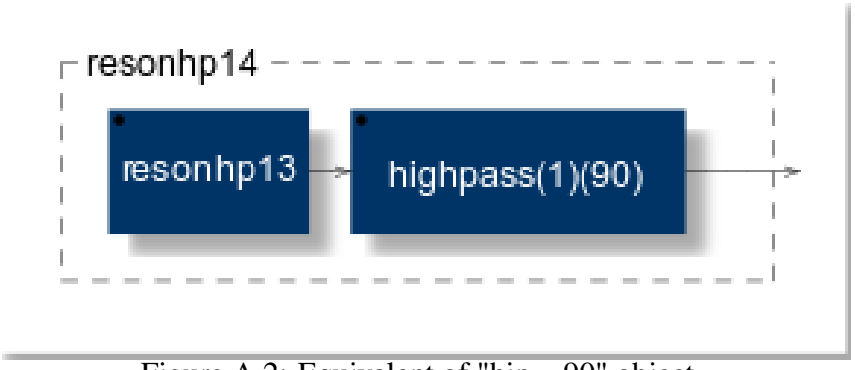


Figure A.2: Equivalent of "hip~ 90" object

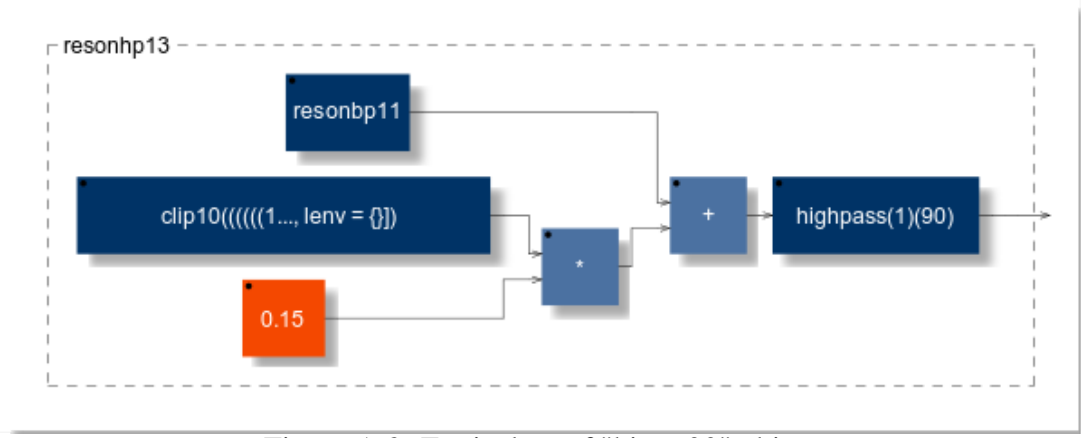


Figure A.3: Equivalent of "hip~ 90" object

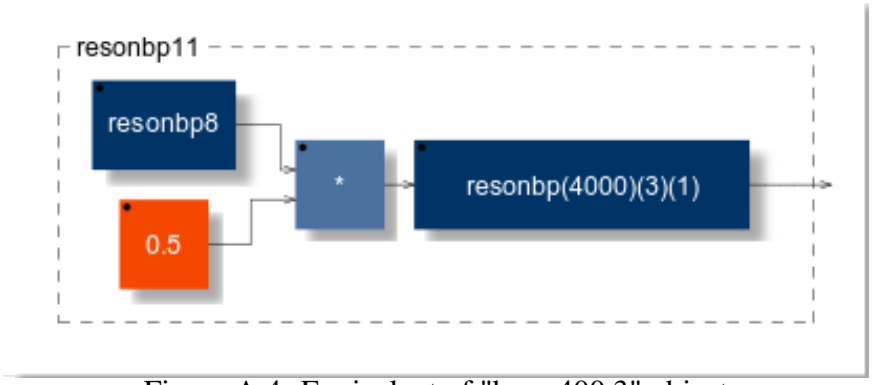


Figure A.4: Equivalent of "bp~ 400 3" object

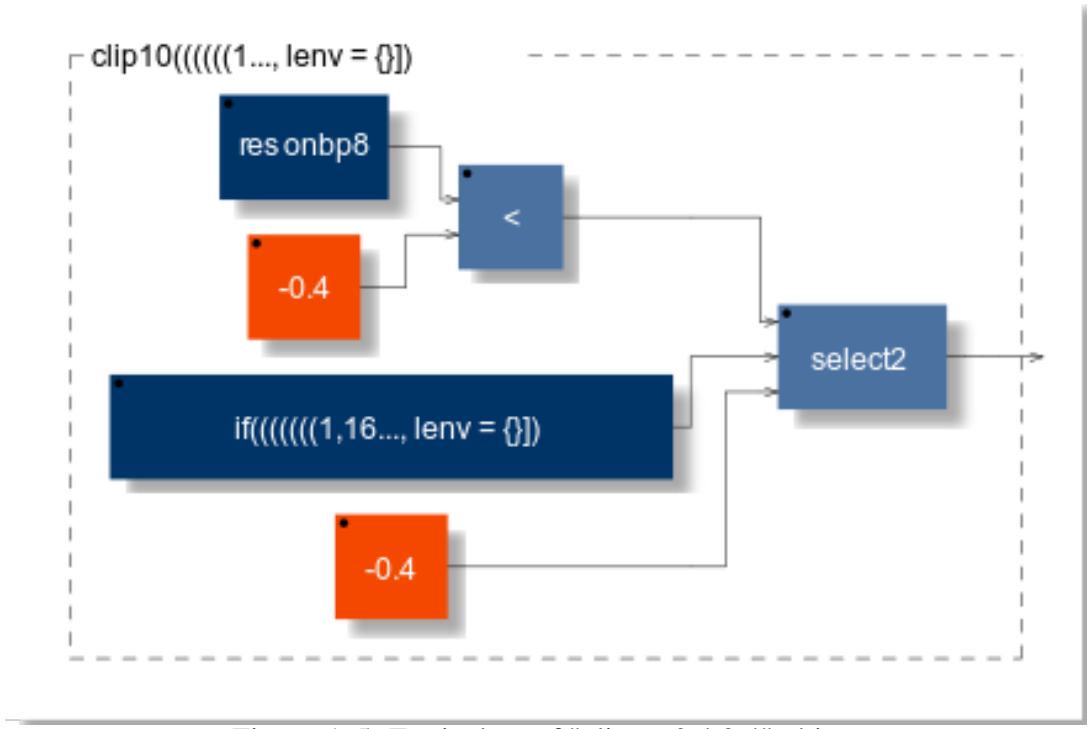


Figure A.5: Equivalent of "clip~ -0.4 0.4" object

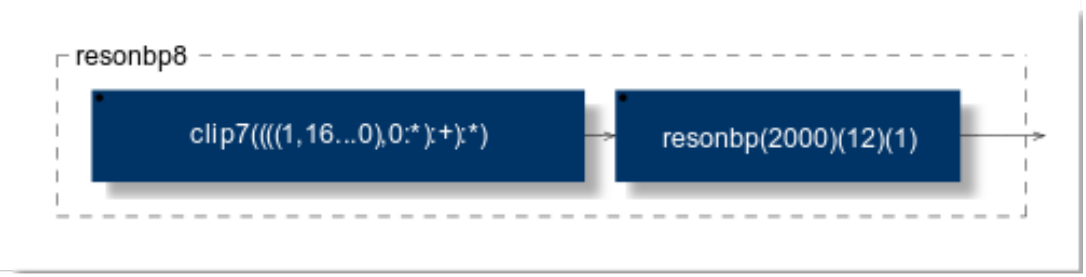


Figure A.6: Equivalent of "bp~ 2000 12" object

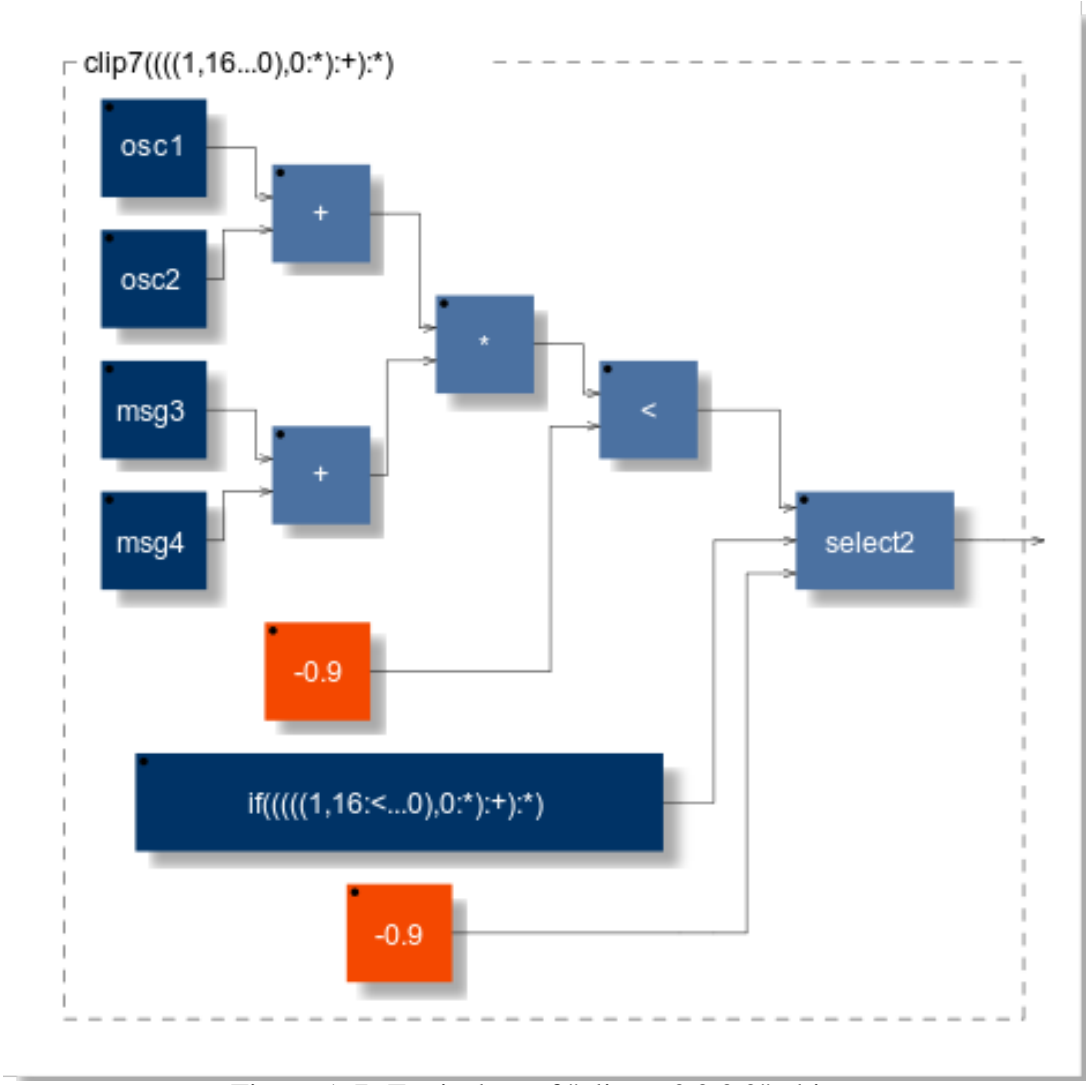


Figure A.7: Equivalent of "clip~ -0.9 0.9" object

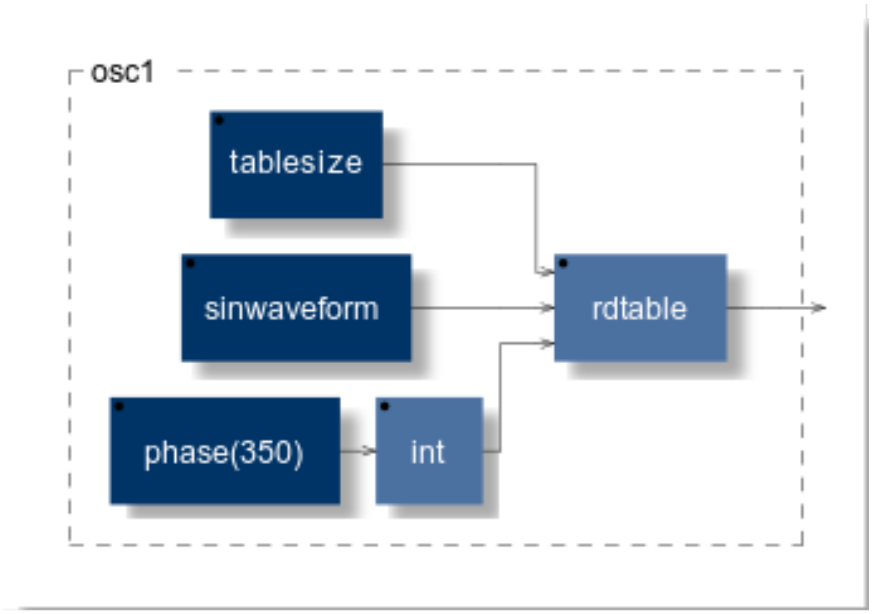


Figure A.8: Equivalent of "osc~ 350" object

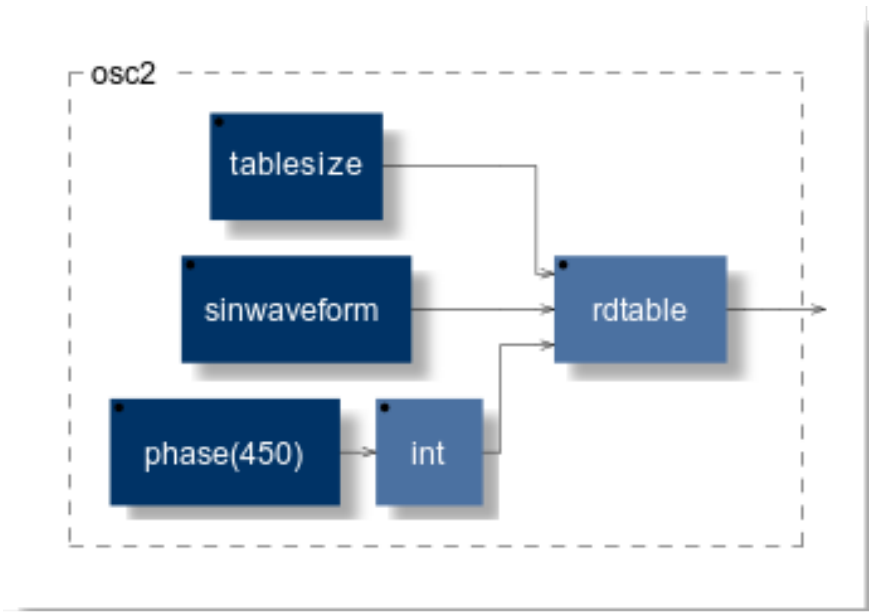


Figure A.9: Equivalent of "osc~ 450" object

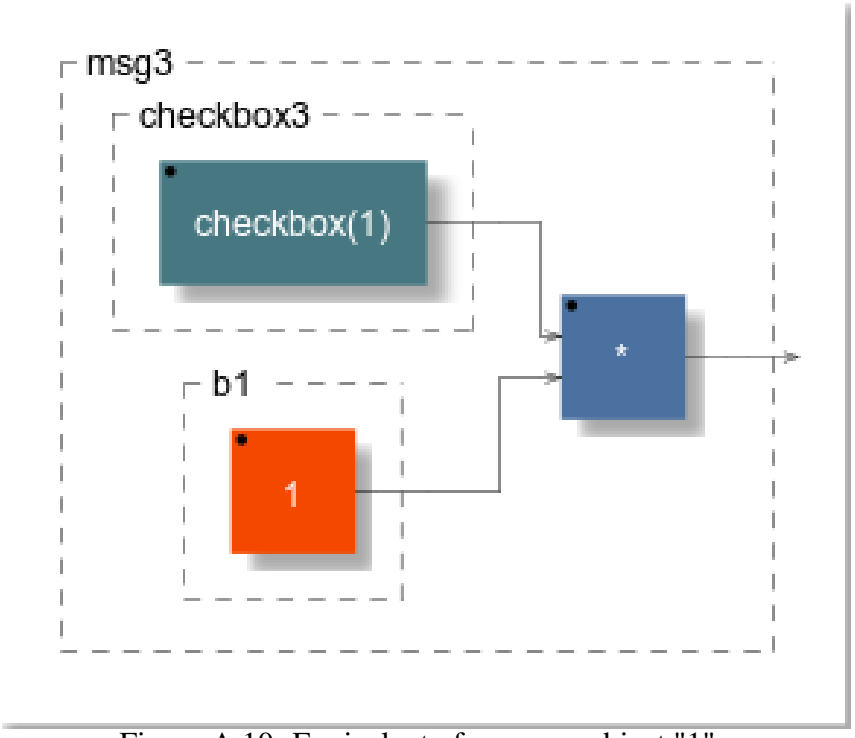


Figure A.10: Equivalent of message object "1"

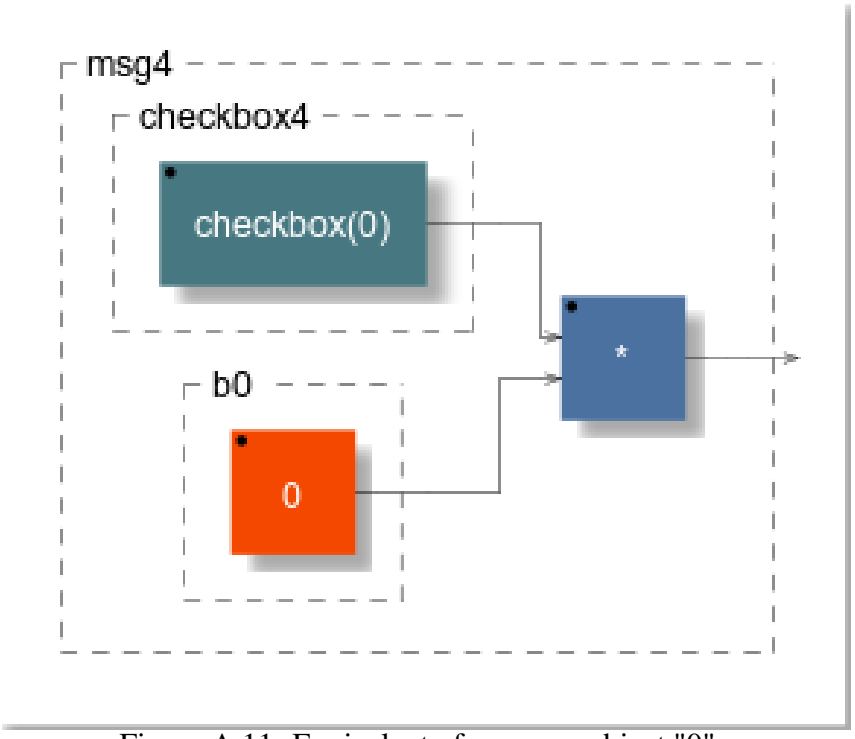


Figure A.11: Equivalent of message object "0"

APPENDIX B

MEPHISTO GENERATED FAUST CODE OF THE JET ENGINE PATCH

Jetengine patch was given as an example earlier in the thesis in Fig. 4.2. Faust code automatically generated by Mephisto for this patch is:

```
import("music.lib");
import("effect.lib");
import("filter.lib");
import("math.lib");

clip35(s) = if(s<(-1),(-1),if(s>1,1,s));
resonlp1=(number37):lowpass(1,0.2);
resonbp32=noise:resonbp(8000,0.5,1);
resonhp33=((resonbp32:moog_vcf_2b(((msg29)/10),
((((resonlp1)*(resonlp1)))*150)):*(70))):highpass(1,120);
resonlp4=((((clip25((osc20+((osc18+osc19)*0.25)+
((osc21+osc22)*0.4))))*0.5)+(clip35((((resonhp33)*120))):
moog_vcf_2b(((msg30)/10),((resonlp1)*12000)):*(70)))):
lowpass(1,11000);
number37=hslider("number 37",0.5,0.1,1,0.1);
checkbox12=checkbox("3097 4495 5588 7471 11000");
osc19=osc((((checkbox12*4495))*(resonlp1))));
osc18=osc((((checkbox12*3097))*(resonlp1))));
osc21=osc((((checkbox12*7471))*(resonlp1))));
osc20=osc((((checkbox12*5588))*(resonlp1))));
osc22=osc((((checkbox12*11000))*(resonlp1))));
clip25(s) = if(s<(-0.9),(-0.9),if(s>0.9,0.9,s));
checkbox29=checkbox("1");
msg29 = checkbox29 * 1;
checkbox30=checkbox("0.6");
msg30 = checkbox30 * 0.6;

process=((resonlp4)*0.1),((resonlp4)*0.1);
```

APPENDIX C

LIMITED GRAMMAR OF PURE DATA WRITTEN IN ANTLR V4 FOR MEPHISTO

Extended information and clarification can be found in ANTLR 4 reference [35].

```
grammar Rows;

file: (row)+ ;
row:XOBJ type=CONNECT INT? INT? INT? INT? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=OSC expr? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=PHASOR INT? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=DAC SEMICOLON NL
    |NOBJ type=CANVAS INT? INT? INT? INT? INT? SEMICOLON NL
    |XOBJ type=FLOATATOM expr expr expr expr expr expr MINUS MINUS
       MINUS SEMICOLON NL
    |XOBJ type=OBJ INT INT name=PLUS number=(INT|FLOAT)? SEMICOLON
       NL
    |XOBJ type=OBJ INT INT name=MINUS number=(INT|FLOAT)? SEMICOLON
       NL
    |XOBJ type=OBJ INT INT name=MULTIPLY number=(INT|FLOAT)?
       SEMICOLON NL
    |XOBJ type=OBJ INT INT name=DIVIDE number=(INT|FLOAT)? SEMICOLON
        NL
    |XOBJ type=OBJ INT INT name=POW number=(INT|FLOAT)? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=TRIGGER (INT|FLOAT|STRING)+
       SEMICOLON NL
    |XOBJ type=MSG INT INT (INT|FLOAT)* SEMICOLON NL
    |XOBJ type=OBJ INT INT name=FCOUNTER SEMICOLON NL
    |XOBJ type=OBJ INT INT name=MOD INT SEMICOLON NL
    |XOBJ type=OBJ INT INT name=CLIP expr expr SEMICOLON NL
    |XOBJ type=OBJ INT INT name=VCF expr expr SEMICOLON NL
    |XOBJ type=OBJ INT INT name=BP expr expr SEMICOLON NL
    |XOBJ type=OBJ INT INT name=HIP expr? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=LOP expr? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=MAX expr? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=MIN expr? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=GT expr? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=LT expr? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=EQ expr? SEMICOLON NL
    |XOBJ type=OBJ INT INT name=UNPACK (INT|FLOAT|STRING)+ SEMICOLON
        NL
    |XOBJ type=OBJ INT INT name=COS SEMICOLON NL
    |XOBJ type=OBJ INT INT name=NOISE SEMICOLON NL
    |XOBJ type=OBJ INT INT name=TGL INT INT STRING STRING STRING
```

```
        expr expr expr expr expr expr expr expr expr SEMICOLON NL
    |XOBJ type=OBJ INT INT name=SIG number=(INT|FLOAT)? SEMICOLON NL
        ;

expr: (INT|FLOAT|VAR)                  #Single
    |expr (MULTIPLY|DIVIDE) expr       #MulDiv
    |expr (PLUS|MINUS) expr            #Single
    |MINUS expr                        #Minus
    |LPAREN expr RPAREN                #Paren
    ;
XOBJ: "#X";
NOBJ: "#N";
DIVIDE: "/" | "/~";
MULTIPLY: "*" | "*~";
MINUS: "-" | "-~";
PLUS: "+"| "+~";
GT: ">";
LT: "<";
EQ: "==";
FLOATATOM: "floatatom";
OSC: "osc~";
PHASOR: "phasor~";
DAC: "dac~";
MOD: "mod";
FCOUNTER: "fcounter";
SIG: "sig~";
CLIP: "clip~";
BP: "bp~";
HIP: "hip~";
LOP: "lop~";
VCF: "vcf~";
OBJ: "obj";
CANVAS: "canvas";
CONNECT: "connect";
MSG: "msg";
POW: "pow";
UNPACK: "unpack";
COS: "cos~";
NOISE: "noise~";
TGL: "tgl";
MAX: "max~";
MIN: "min~";

INT: DIGIT+;
FLOAT: DIGIT+ "." DIGIT*;
DIGIT: [0-9];
TRIGGER : "trigger";
VAR: "$"STRING;
STRING: [a-zA-z0-9]+;
SEMICOLON : ";";
LPAREN: "(";
RPAREN: ")";
TAB : [ t]+ -> skip ;
NL : " r"? " n";
```

APPENDIX D

THE PD OBJECTS RECOGNIZED BY MEPHISTO

The full list of recognized PD objects by Mephisto in addition to number boxes, messages and user interface elements is as follows: `osc~`, `phasor~`, `dac~`, `+~`, `+`, `-~`, `-`, `*~`, `*`, `/~`, `/`, `pow`, `trigger`, `mod`, `clip~`, `vcf~`, `bp~`, `hip~`, `lop~`, `max~`, `min~`, `>`, `<`, `==`, `unpack`, `cos~`, `noise~`, `sig~`,