Toward a More Flexible Composition Environment: Reimagining Computer Music Interfaces

Jeffrey B. Holland

Submitted in partial fulfillment of the requirements for the
Master of Music in Music Technology
in the Department of Music and Performing Arts Professions
Steinhardt School
New York University

Advisor: Dr. Tae Hong Park
Reader:

April 25th, 2022

ABSTRACT

Modern music producers primarily use Digital Audio Workstations (DAWs) for arrangement, engineering, and often composition as well. While every DAW offers a unique set of tools and features, they all share a common goal of providing users with the ability to produce music on their computers. To achieve this objective, certain assumptions must be made about what kinds of music users wish to create, how they want to create it, and what kind of interface will allow them to attain maximum levels of creativity. At this point in time, prevailing methods of DAW design are geared toward the most prevalent Western musical conventions, such as equal temperament and simple meter. Consequently, composers who wish to explore experimental musical territory beyond these paradigms are forced to either work against the grain of the DAW they are using, or to use musical programming platforms such as Max/MSP, Pure Data, Supercollider, or Csound; all of which require a specific technical skill set. This thesis seeks to provide a more intuitive and effective interface for experimental music composition; one that does not require programming skill but provides an elevated level of flexibility for experimental artists without assuming the intent to color within the lines of conventional musical confines. The resulting software toolkit was evaluated by a panel of three computer music experts at New York University. The experts' comments showed that this project was successful in creating a flexible, malleable, and creatively inspiring interface for music production, but also that future work is needed for the outcome to grow into a fully-fledged DAW.

ACKNOWLEDGEMENTS

First, huge thanks to Dr. Tae Hong Park for advising me on this thesis, from initially encouraging me to pursue the idea to giving final feedback, and everything in between. Special thanks as well to Dr. Agnieszka Roginska for her guidance, especially with regard to drafting the paper and conducting the expert panel.

Praise and gratitude to all the professors who have expanded my understanding of electronic music and the creative process: Peter Hamlin, Morton Subotnick, Leila Adu-Gilmore, and Dafna Naphtali. I am also indebted to the professors who patiently taught me to understand digital signal processing in both theory and practice, and to program and create powerful music and audio tools through code: Brian McFee, Ernesto Valenzuela, Schuyler Quackenbush, and Dirk Vander Wilt, as well as the aforementioned Dr. Park. Additional acknowledgements to all the other professors who lent their invaluable expertise and advice on my journey through the Music Technology program: Paul Geluso, Robert Rowe, Barry Greenhut, Alan Silverman, Kevin Killen, and Jean-Luc Cohen.

Huge thanks to Sean Goldie for providing advice on my coding journey, sharing endlessly helpful resources, always being willing to let me bounce ideas off him, and being a friend I can always rely on. Special thanks to Jack Tipper for also being a springboard for ideas and helping to clean up my prose. Thanks as well to Gregor McWilliam, Ricardo Moreno Sayavedra, Daniel Braunstein, and Eugene Markin for being dear friends who will talk with me about anything, from music and programming to films, politics, and beyond.

Boundless praise to Eric Benoit for being an absolute inspiration as a fellow artist and human being, a close friend and confidant for life, and someone who has always believed in me.

Enormous gratitude to my mother and late father for caring deeply for me and about me; providing advice, encouragement, and wisdom; and being fully supportive of my niche interests and ambitions throughout my unconventional academic career. Same to my sibling, Matthew, for always being by my side to support, comfort, and understand me. Thanks also to my two cats, Soot and Sand, for providing emotional support and a healthy dose of spontaneity.

Finally, extremely special thanks and love to Emily Wallen for staying with me through all the ups and downs, co-parenting Soot and Sand with me, and encouraging me when I felt like I would never finish this thesis.

TABLE OF CONTENTS

INTRODUCTION

a. Research questions

With inspiration and guidance from the artistic and theoretical works of composers who have pursued and achieved innovations in the art of creating electronic and/or experimental music, who have eschewed outdated norms and propelled the musical world at large toward the infinite unknown of an aesthetically liberated future, this thesis aims to explore a novel design for an electronic music production interface that offers greater flexibility than those currently in use. This versatility is particularly targeted toward composers of experimental electronic music, whose needs may not be met by the tools currently in use due to several limitations which will be discussed in the following sections.

An *interface* herein is defined as the connecting medium between a person and a computer. It is a way for a person to enter data and commands into the computer (input), and for the machine to convey responses (output) back to the user. Most electronic musicians today utilize computers, and many of them choose a DAW as their primary interface with the underlying hardware. This project aims to create an improved interface that links composers with the powerful creative abilities that computers can afford, avoiding, when possible, the assumptions and limitations which stifle creativity and constrain the creative possibilities of the artform. Communication is a central focus of any interface, including this project; both communication between the user and the program, and communication between the various parts of the program, play crucial roles in the effectiveness of the tools provided.

The goal of the project, in short, is to facilitate exceptionally fluid communication between the user and the program so that the user can turn sonic ideas into reality. It should inspire creativity, allow for full customization, maximize flexibility, and function intuitively. These goals, and the degrees to which they were achieved, will be elaborated on over the course of this document.

b. Context: Electronic Music

Before delving into the functions of the interface itself, it is necessary to place it within the larger history of electronic music composition and its interfaces. Prior to the advent of the electronic music synthesizer and the tape recorder, composers were limited to writing scores that would be performed on one or more acoustic instruments after the period of composition, often by individuals other than the composer.

Listeners could only experience the music by attending a performance and witnessing a performer's interpretation in person. A written score can only be so precise in its instruction, and therefore inevitably leaves many aspects of the music to the discretion of the performer. In the event that a composer performs their own work, there is still a gap between the moment of composition and the moment of performance, introducing a discontinuity between the two stages of the process. Even improvised performance requires a level of planning and practice in advance, which is in and of itself a preliminary form of composition, so there is still a measure of division between the two. In that scenario, all the more aspects of the music are controlled by the performer rather than the composer.

The advent of the synthesizer and the tape recorder erased this barrier between composition and performance, effectively merging the composer and performer into a single entity acting in unison. While sitting at the controls of a synthesizer or splicing and overdubbing recorded sounds, it is possible to compose and perform a piece of music at the same time, without relying on a dedicated performer to interpret a written score. Electronic music composers can take stock in the fact that machines, unlike human players, can reproduce sounds with exact precision. Thus, to achieve an elevated level of control over the sound output, one need only provide the machine with the instructions necessary to create the sound one wants to hear. With knobs, patch cables, sliders, lines of code, or any other input methods an interface might consist of, the user can specify the particular qualities of their intended sonic creation, including pitch, duration, dynamics, and timbre. Therefore, the composer of electronic music is free to exercise far more control over the specific characteristics of their sound than composers ever did prior to this technology.

c. Context: Digital Transformation

Today, much of electronic music has migrated to the digital realm, and the DAW is the predominant tool used by most electronic music producers and composers. Considering the precision and processing power of a modern CPU, composers have a much greater degree of control over the characteristics of sounds than was possible with the tape recorder and synthesizer. While this is indeed the case for clever, computer literate composers, many common digital music interfaces limit the creativity of their users. For example, digital hardware interfaces like MIDI controllers often emulate traditional acoustic instruments like piano keyboards, drum sets, and woodwinds. Electronic instruments are frequently designed to approximate the sound and character of traditional instruments, in many cases merely sampling the

recorded sounds of traditional instruments. The spectrum of music composition software should be expanded to liberate and stimulate the creativity of composers by opening the door to new realms of electronic sound, rather than retracing the well-worn path of traditional compositional techniques. This thesis will build upon the ideas of past composers while also looking toward the future of composition, by designing an original software interface to immerse composers in their craft and provide the freedom to turn the most daring sounds from imagination into reality.

The most popular DAWs available today exhibit a variety of design approaches, giving composers and producers different interfaces through which to execute their practice. Many of these offer unique and powerful features, facilitating fruitful creative possibilities. However, it can simultaneously be observed that most DAWs, despite the impressive capabilities they do offer, are nonetheless encumbered by certain limitations which discourage, if not outright prevent, their users from exercising the full extent of their imagination. Creators who want the freedom to transcend these barriers may find it necessary to design their own interfaces through computer programming. Programming languages specifically designed for creating music, such as Pure Data, Max/MSP, Csound, and Supercollider, are an excellent choice for those who have the time, ability, and motivation to fashion their own tools. For many composers, however, using a programming language as their primary toolkit may not be an optimal situation. Though programming allows for a high flexibility of sound and interface customization, it can require technical skills far removed from composition, and it can be distracting and overwhelming to simultaneously focus on music and computer programming. For these reasons, many composers are stuck using DAWs, which rely on inherited metaphors from decades-old devices like tape machines and analog mixing boards, and which neither inspire nor liberate creativity to outpace these archetypes. This thesis seeks to uproot traditional paradigms of music software by creating a program that challenges the assumptions made by DAW developers and expands the universe of computer music composition by offering a new degree of flexibility, without the weblike complexities and steep learning curve of a programming language. The core aspiration of this work is to meet the needs of composers who may use unorthodox methods to push the boundaries and challenge the norms of music and sound art, and who may not be willing or able to embrace the task of learning to program computers for compositional ends.

d. The Problem with DAWs

Limitations of DAWs typically result from normative assumptions which are "baked" into the program, due to the influence of cultural and social forces which dictate the ways in which people conceive of music and sound. The most deeply entrenched of these cultural and social forces stem from the hegemony of Western music theory. This can in turn be understood as a by-product of European colonialism and its accompanying Eurocentric affectations of supposed aesthetic superiority. Non-Western cultures, of course, have their own traditional music theories and cultures agnostic of Western norms, and have practiced their musics since long before eminent European and American composers in the Western tradition first began to break the molds of conventional tonality, rhythm, and timbre in the early 20th century. To take one example in the domain of pitch, the *maqam* scales, which form the basis of the tonal system of traditional Arabic and Turkish music, diverge from Western music theory by incorporating quartertones, as well as scales that are not transposable between keys. It is uncommon to find a DAW or other program in which one could easily compose music within a non-Western paradigm. In terms of tonality, most DAWs are firmly rooted within the Western tradition of equal temperament. This convention offers only a lattice[1] of semitones, with notes typically labeled and arranged in the form of a piano keyboard following the traditions of Western notation, making it difficult to compose outside of this structure.[2]

The goal of this project is not to create an interface that specifically favors the composition of Arabic or other non-Western styles of music, though more such interfaces are needed. Rather, the goal is to take an approach of flexibility and openness by assuming as little as possible about the choices that the composer will make, allowing them to explore with minimized restrictions the infinite domain of conceivable methodologies by which to organize sound. For instance, regarding the question of pitch as we have been discussing it, it should not be assumed that the user wishes to work with pitch intervals of quartertones, semitones, whole tones, or any other specific interval of pitch. It should be possible for the user to specify any pitch or frequency intervals of their choosing. It should not even be assumed that the user wants to work with a system of intervals between stable pitches: they may hope to work with continuously changing pitches, unpitched or ambiguously pitched sounds, controlled random or statistically distributed pitches, close tone clusters, or some combination of these methods. In general, the interface should

[1] The term "lattice" to refer to traditional Western pitch notation structure is borrowed from Wishart (1996).
[2] Generalized statements about DAWs are based on the author's experiences with Ableton Live, Pro Tools, Logic Pro, GarageBand, and FL Studio.

furnish the highest possible degree of flexibility in all compositional choices relating to dimensions of the music, including not only pitch, but also rhythm, timbre, and dynamics.

e. Benefits of Technology

Despite this lack of flexibility in current DAWs, there is much to celebrate about the achievements of engineering which have led to the advanced computer music technology available today. The rapid transformation of the technological landscape has streamlined the processes by which electronic music is made. To understand how drastically the technology has improved, one need only imagine the composition process of such seminal works of electronic music as Karlheinz Stockhausen's *Gesang der Jünglinge* (*Song of the Youth*, 1955) and Morton Subotnick's *Silver Apples of the Moon* (1967). Cumbersome equipment, expensive tape, and arduous editing procedures have given way to ultrafast CPU precision, terabytes of affordable data storage, and seamless click-and-drag editing. Curtis Roads is one example of a composer who has taken today's technology to new heights by composing in a uniquely digital sonic language: miniscule subdivisions of time form grains of sound that Roads arranges into lush, complex patterns, reminiscent of natural processes such as swarms of insects and bubbling rivers (Roads, 2011). Without today's computer technology, composing in this fashion would be extremely difficult if not downright impossible. John Cage foresaw these changes with astonishing accuracy in his 1937 essay "The Future of Music: Credo", wherein he predicted that forthcoming composers would exercise control over "the entire field of time", using "frames" or small fractions of a second as the basic temporal units of their compositions (Cage, 1961). As a result of this newfound time-precision, Cage envisioned that "no rhythm will be beyond the composer's reach." Indeed, composers like Roads who work with granular synthesis and processing have actualized the future that Cage predicted, and the sample-level control offered by our computer audio devices and digital signal processing (DSP) algorithms is an exact manifestation of what Cage described decades prior.

Cage's clairvoyance notwithstanding, it is unclear from the typical DAW that control of the entire field of time is available to the composer. Most DAWs offer, by default, subdivisions of time intervals into groups of four on a grid, as is the Western rhythmic convention. Attempting to place timed events outside of these intervals, in other words "off the grid", is to work against the design of the software, which encourages grid alignment by default. There is a range of flexibility in the sense that the user can determine the tempo of the piece (measured in beats per minute) to increase or decrease the length of

temporal subdivisions. Dynamically changing the tempo over the course of a piece is also a possibility. Many DAWs permit deactivation of the grid in order to place timed events in a free state; however, it can be difficult to achieve precise timing or musical flow in this manner. Even though it is possible to transcend the rhythmic grid, this grid remains the default option in most DAWs, as does the semitone lattice for selecting pitches via MIDI, the only note-triggering system available in most DAWs. Thus, with regard to the manipulation of time and pitch, there is considerable room for improvement.

There is nothing inherently wrong with these default grid settings nor with MIDI; they exist for convenience, to help users get started making music immediately, instead of needing to manually set specific pitch and time values. However, hidden underneath these default settings, there are assumptions about the fundamental form of music. The main assumption is that composers wish to use the same kind of musical notation that the Western tradition has homogenized for hundreds of years: a grid of stable semitones on the vertical axis for pitch, and subdivisions of four on the horizontal axis for rhythm. These presumptions about the default way to produce music serve to subconsciously influence composers to produce music in conventional ways, implying that all other approaches are simply distortions of the norm.[3] Evaluating the impact of these design choices on the musical output of their users is beyond the scope of this thesis. There is no practical way to entirely avoid such influences, as they are by nature hard to pin down. Nevertheless, software designers can still strive to create interfaces that steer clear of influencing their users to think in a particular way—a conventional way—about the dimensions of music, i.e., rhythm, pitch, timbre, and dynamics. The goal is to encourage creativity rather than conformity.

f. Dimensions of Music

The limitations described regarding the typical DAW and its lack of creative flexibility correspond to pitch and duration as the two core dimensions of music. Pierre Boulez (1971) deemed these two concepts to be "the basis of a compositional dialectic," whereas he considered dynamics and timbre to be "secondary categories," contrasting the precision with which pitch and duration have historically been notated in musical scores with the vague markings and adjectives[4] used to indicate dynamics and timbre.

[3] For extended reflections on how devices can exhibit subconscious influences on the way humans think and act, see Lanier (2010).

[4] "Mezzo-forte", Italian for "medium-loud", is one example of a vague indication of dynamics. Conventional notation offers little specificity in this regard. Timbral indications are all but absent, as if it were assumed that each instrument has a fixed timbre. This is true to some extent for acoustic instruments: for example, one cannot make a violin sound like a trombone. Instrumentalists do have limited control over timbre within the capabilities of their

Trevor Wishart (1996) challenged Boulez's conception, arguing that the predominance of pitch and duration in traditional Western music is a result of notation methods prioritizing those aspects of music, and not vice versa. To serve this thesis's goal of inspiring an expanded range of creative possibilities for sonic art, the resulting interface should not overemphasize pitch and duration, instead embracing the multifarious dimensions of sound, giving composers tools to tap into the outer reaches of sonic potential. With this objective in mind, methods are sought for which the additional dimensions of sound—timbre and dynamics—can be manipulated in order to create stimulating results.

instrument, but because composers have few ways to give timbral instructions in their scores, timbral variations and gestures in acoustic music are often left to the discretion of the performer.

LITERATURE REVIEW

a. Foundations of Composition

An investigation of what constitutes "traditional music" in the Western paradigm informs our discussion of how experimental music subverts those conventions. Haydn and Mozart were seminal in developing the forms and methods of what is now known as the Classical period of music: a time when structured tonality consisting of major and minor keys was a rigid system in which all pitch was organized. Structurally, the compositions follow a pattern of what would later be called “developmental variation”: they begin with a short, simple, melodic motif, and gradually build on this motif by creating variations of it, developing it into a far more complex structure and modulating between keys while doing so (Damschröder, 2012). The chief advantage of this approach is that listeners can recognize the original motif in all its different variations and keys; therefore, listeners do not lose interest as they might with simple repetition, nor do they become overwhelmed by complexity. The original motif reappears in variations, guiding the listeners as they follow the gradual unfolding of the music.

The composer Arnold Schoenberg (1874-1951) was among the first to truly break from the imposed tonal structures in the Western tradition (Jenkins, 2016). As a result of Schoenberg's experimentations with tonality, critics labeled him as a composer of "atonal" music, despite his revulsion to the term, as his music is not in any way removed from tonality. The hallmark of his style is a method known as the "twelve-tone technique", which consists of using all twelve tones of the chromatic scale with equal emphasis on each tone, situating the music outside of any distinct key. This is, of course, a radical departure from conventional tonality; and yet, structurally, Schoenberg's music is not detached from the Classical tradition. It was Schoenberg himself who coined the term "developmental variation", which refers to the process described in the previous paragraph of starting with a simple motif and then building on it through gradual variations to create a more complex structure. The popularity of developmental variation as a compositional method is evident from its ubiquity in Western music throughout the centuries, from Haydn and Mozart to Schoenberg and beyond.

Schoenberg may have departed from conventional tonality, but he was not so radical as to eschew tonality altogether. One of the most well-known composers to make this leap was John Cage, a landmark artist of the post-war avant-garde. Cage challenges listeners to find aesthetic interest in sounds that are typically classified as "noise" rather than music, arguing that noise has the potential to be an essential component of musical composition (Cage, 1961). Cage's work broaches questions about the very nature of music

itself and insists on exploring new methods of musicality, rather than imitating the methods of the 18th and 19th centuries. Writing in 1937, he expressed interest in "electrical instruments" and their potential to offer complete control over tone, rhythm, and timbre as well as the "field of time", from the tiniest perceptible temporal delineations to extended swathes of duration. He showed less interest in using electrical instruments to merely imitate traditional practice, such as thereminists who only used the novel instrument to perform classical works, rather than taking advantage of its unique characteristics to forge a new and distinctive musical style. Cage predicted that in the music of the future, "the only constant connection with the past" would be "the principle of form." In other words, while methods of tonality, rhythm, timbre, and most other aspects of music will change, according to Cage's predictions, musical forms would still follow similar guiding principles, such as developmental variation.

After post-war avant-garde composers like Cage upended the fundamental basis of tonality upon which the Western musical canon had lain, other composers sought to invent new structures of music to replace the traditional paradigms. Iannis Xenakis, a contemporary of Cage, aimed to reimagine music using models drawn from mathematics, statistics, and physics. His style is characterized by countless minuscule grains of sound as well as long *glissandi*, or held notes that continuously ascend or descend in pitch—a unique approach he achieved with both acoustic and electronic instruments. Xenakis called his compositional method "stochastic", as it made use of randomly determined probabilistic methods to organize sound in new ways. He rejected what he termed the rigidly linear "causality" of the past—such as the structured tonality of Haydn and Mozart—in favor of alternative, mathematically determined, probabilistic arrangements (Xenakis, 1971). Xenakis did not think of stochastics as a purely human invention; rather, he points out the independently stochastic temporal organization of events, such as sounds, in nature. Sounds that consist of a large quantity of small individual components, such as rain falling on a metal surface or a field of cicadas, are distributed in time according to stochastic principles, without any human intervention to make them so. Hence, Xenakis can be seen not as an iconoclast who abstracts away from tradition, but one who insists on returning and listening more closely to nature's traditions of sound which have existed since long before humans began inventing sounds of their own and calling them music.

To summarize this sparse, pitch-focused highlight reel of the history of contemporary music: Classical composers like Haydn and Mozart mastered the techniques of conventional tonality; serialist composers like Schoenberg abstracted away from conventional tonality and toward a new kind of tonality; Cage and his contemporaries abstracted away from tonality altogether and toward noise and randomness; and finally, Xenakis and others sought to rise from the ashes of tonality and toward a new form of sound

organization which reflects the natural world. Having absorbed this background information on the development of avant-garde electronic music, the stage is set to turn to the ideas circulating in contemporary electronic music today.

b. Contemporary Composition

One of the primary issues concerning composers today is the field of time and how it relates to music. As noted previously, Cage predicted as early as 1937 that electronic methods would give composers greater control over time. This prediction was evidently correct: with the help of computers, composers such as Curtis Roads can meticulously craft rich tapestries of sound out of fine "grains" (Roads, 2001). Not only has digital technology given composers a tremendous depth of control over the field of time, but physicists have also made discoveries that overthrow previous scientific understandings of what time is and how it functions, most notably Einstein's theory of special relativity and discoveries relating to quantum particles. In other words, time operates in ways far more complex than was previously assumed. Composers have since had to reckon with a new notion of the field of time in which their music exists.

One theorist who has taken on this challenge is Guerino Mazzola (2019), who discusses the musical applications of Stephen Hawking's concept of "complex time", i.e., time that consists of both real and imaginary components. He contends that the idea of complex time can shed light on the nature of artistic consciousness by resolving the conundrum that the mind exists in "no time" yet exudes a "rich processuality" consisting of "memory, technique, gestures, and the balance… between past and future moments." Mazzola therefore argues that the way in which the artistic consciousness operates during a musical performance can only be understood through complex time. This thesis aims to construct a new temporal environment for composers to work in and will thus need to consider Mazzola's argument about complex time and the understanding of time informed by modern physics.

Besides the field of time, the other most obvious and general field which every composer must consider is that of space. Filipe Lopes and Carlos Guedes (2020) formalized the characteristics of space upon which compositional approaches are structured into two categories: on the one hand, there is "aural architecture," which consists of acoustical phenomena within a space; and on the other hand, there is the "soundscape," or the environmental sounds within a space. The authors also discuss ways in which the "sonic expressiveness" of a space can be utilized to create a sense of "empathy" between the composition

and the space. Keeping these ideas in mind for this thesis will result in a creative environment informed by the relationship between music and the spatial dimensions it inhabits.

The final source to discuss in this section seeks to formalize connections between the body and mind through sonic meditation. Jiayue Cecilia Wu (2020) explores the ways in which sound can unite the body and mind as well as the physical and spiritual elements of the self, in a process known as "embodiment." Wu's work involves creating interactive art installations in which participants are paired with motion-sensor controllers to trigger audio events by moving their appendages, with the goal of facilitating embodied sonic meditation experiences uniting the mind, body, and soul. Wu discusses the origins of this practice in Tibetan Buddhism as well as the ways in which it has been applied musically by composers such as Pauline Oliveros, Éliane Radigue, and Philip Glass. It is important for this thesis to consider the confluence of spirituality, psychology, physics, and other disciplines that help to understand the relationship between humans and music, to create an interface that endeavors to realize the full potential of computer music creation.

c. Graphical User Interface (GUI) Design for Music Composition

Numerous published works explore the variety of existing GUIs intended for music creation. There is The Third Room, a software which allows users to use Microsoft Xbox's Kinect interface to control virtual instruments (Honigman et al., 2013). Chen Ji's Sakura is also noteworthy: a virtual reality game in which the gameplay revolves around creating music, with a MIDI keyboard as the controller, and an interactive audiovisual world reminiscent of Buddhist "Zen gardens" constituting the environment (Ji, 2020). One interesting smartphone app uses a 3D interface in augmented reality for controlling spatial sound in a DAW (Cassorla et al., 2020). Tarik Barri's Versum interface for 3D audiovisual musical performance boasts a design intending to "break down the barrier" between the composer and the audience in the sense that the audience can see a visual representation of what the composer/performer is doing (Barri, 2009). These articles contribute relevant context to this thesis and offer valuable insight for the exploration of visual interfaces for music composition.

A particularly pertinent example is Richard Polfreman's FrameWorks 3D, a software that contains a DAW-style environment with a 3D GUI, constructed using a combination of Max/MSP and Java (Polfreman, 2009). Data is organized in the form of "clips" containing audio, MIDI, or OSC data, represented as floating 3D rectangles displaying audio waveforms. Processing is executed from the top

down: the material in a clip can be processed and sent to a target clip placed below it in the 3D space. Plugins are called "relations" and consist of Max/MSP patches designed by Polfreman himself. Frameworks 3D is a particular source of inspiration for this thesis.

Of course, when designing an interface for music production, the relationship between the user and the product must be considered. This process in part involves the use of psychoacoustic principles to predict how users may interact with the product. Dewey and Wakefield (2016) employed this method for audio mixing interfaces by thoroughly analyzing the psychoacoustic principles behind different visualizations of level and panning. Any music production interface could benefit from the thoughtful study of these same principles. The same authors conducted additional studies in 2018 on how users respond to various kinds of visual feedback in a mixing interface, controlling the data by providing a group of users no visual feedback at all. These data bear significance for this project of creating a GUI for music creation.

d. Playing notes

One of the most pressing concerns of this thesis is reimagining the ways in which composers generate sounds, or "play notes" using synthesizer components in a DAW. Most DAWs currently use MIDI as the only way to trigger sounds with a synthesizer component or plugin. There is surprisingly sparse research into ways in which the MIDI system can be subverted, reimagined, or replaced by something more flexible and powerful. However, there are several interesting cases of technologists working with MIDI in more advanced ways.

Before going on to discuss research on advanced MIDI techniques, it is necessary to mention the MIDI Polyphonic Expression (MPE) specification, developed by the company ROLI which specializes in hardware MIDI interfaces. This new MIDI framework adds parameters such as timbre, pitch, and amplitude modulation to MIDI data. The author was unable to find any peer-reviewed research on this new technology, but its mission statement can be found on ROLI's website, where the company states that its goal is to "allow… digital instruments to behave more like acoustic instruments" (ROLI, 2022). MPE therefore has a different aim than that of this thesis: rather than emulating the expressive capabilities of acoustic instruments, this thesis is interested in discovering new techniques for musical expression in the digital domain. MPE is an interesting way to augment the capabilities of MIDI for the purpose of imitating acoustic instrument expression on digital interfaces, but it does not follow the path of inquiry upon which this thesis embarks.

Reece (2013) is concerned with education in music technology, and how software can better expose the inner workings of MIDI and digital synthesis. He uses the software Pure Data as an example of how the educational philosophy of Constructionism can be used to clarify or "construct" these challenging concepts. He believes the merit of Pure Data's GUI lies in the fact that it is easy to see what is going on "behind the scenes" with the MIDI values that control amplitude and frequency. He does not, however, take the next step of asking how values associated with MIDI could be expanded to produce a more advanced MIDI framework.

Weiburn and Plumbley (2009) approached the issue from the angle of transcribing audio into MIDI. They developed a more expressive form of MIDI that incorporates an envelope generator for both pitch and amplitude as well as an LFO, enabling MIDI notes to be written with multiple parameters of expressive metadata. Outputs are then resynthesized using Native Instruments' Kontakt to sonify the results. This framework of expressive MIDI resembles what this thesis intends to accomplish, with each MIDI note having its own set of parameters and an envelope that allows the pitch and amplitude of the sound to change over time. However, this thesis is concerned with writing MIDI in a composition environment, rather than using it to transcribe audio.

Lastly, Nelson and Thom's (2004) description of a performance testing algorithm for MIDI made an impact on this thesis, even though it does not directly relate to the core objective. These authors are concerned with such factors as latency and CPU load when using MIDI in a real-time performance environment. This provides insight into why there is no literature related to enhancing MIDI notes by giving them additional parameters, beyond the little-researched MPE framework. MIDI is often thought of as a method of communication between a hardware controller and a computer for use in a live performance environment. This thesis on the other hand is concerned with reimagining the units of data that trigger sound events in an electronic composition environment. Giving additional parameters to MIDI notes would burden live performances, where real-time responsiveness is of the utmost importance. In a composition environment, however, additional complexity could be a welcome addition to the currently limited world of MIDI, and there is room for experimentation with alternative approaches separate from MIDI altogether.

METHODS

The software development process typically occurs in several steps. In their broadest form, these steps are:

1. *Analysis*: determining the requirements that the software must satisfy, as well as use cases to describe the situations in which someone might use the system.
2. *Design*: determining the structure of the program, including classes, their attributes (the data they hold), their methods (the operations they can execute), and the interactions between these elements. This information is best conveyed via diagrams.
3. *Programming*: writing the computer code to form the body of the software itself.
4. *Evaluation*: testing the software with human subjects and/or automated appraisal tools. (Fowler, 2004)

These steps in turn can be accomplished using either a *waterfall* approach or an *iterative* approach. A waterfall approach means to complete each step of the process fully before commencing work on the next, whereas an iterative approach is the performance of multiple iterations of all steps in succession.

This thesis takes a combined waterfall-iterative approach. Each phase on its own occurred over multiple iterations, but there was also a measure of fluctuation between steps. The preliminary analysis phase placed particular emphasis on cementing the theoretical principles discussed in the introduction and literature review, by imagining ways of rendering compositional theories into practice in a computer program, and systematically acquiring the programming skills required to achieve the desired outcome. Designs consisted of Universal Modeling Language (UML) diagrams and written descriptions. Investigation into object-oriented (OO) software design patterns led to greater understanding of how programming objects should behave and interact with each other.[5] Research on data structures and algorithms was also necessary to ensure that the program would provide the requisite performance. The code was implemented in C++ using the JUCE framework.[6] The programming process took place in several iterations, with individual sections of the project materializing as separate JUCE modules before

[5] This section assumes reader knowledge of basic OO design terms. Diagrams use the UML to indicate classes and objects and how they interact. For an explanation of OO design terms and UML symbols as they are used here, see the Appendix.

[6] JUCE is a widely used software development framework for creating audio applications. It provides an extensive selection of audio and DSP tools, a complete suite of GUI elements, a core library of data structures and memory management tools, and much more. For more information, visit https://juce.com/

eventually conglomerating into a single entity. Finally, the project received evaluation from a panel of three NYU professors, all of whom are experts in electronic music and computer programming for musical applications.

a. Analysis

To explain the analysis stage of this project, it is necessary to discuss some fundamental aspects of DSP theory. Any digitally generated or captured sound signal is sampled in discrete time, associating it with certain qualities expressed in a numeric form and involved in the calculation of the signal before it is sent to the output. The numbers involved in this calculation that are directly modifiable by the user are referred to herein as *instantaneous parameters*, because they affect the qualities of a sound at any given moment—in other words, they are independent of the length of the signal (*N*), as well as the sampling rate (f_s), which, by contrast, are coined *temporal parameters* due to their relation to the way the signal inhabits time. A sine wave has only one necessary instantaneous parameter: the phase. If *x* is a discrete sampled signal of a sine wave, the value *x[n]* at a particular sample *n* is expressed as:

$$x[n] = sin(\frac{\phi n}{f_s})$$

where ϕ is the phase, *n* is the sample value, and f_s is the number of times per second that *n* is incremented. ϕ is the only instantaneous parameter because it is the only modifiable value that changes the sound of the wave at any given instant. If we want to express the wave in terms of *frequency* (*f*), the equation becomes:

$$x[n] = sin(\frac{2\pi f n}{f_s})$$

in which case *f* becomes the sole instantaneous parameter, directly proportional to ϕ (multiply *f* by the constant 2π to get ϕ). Frequency is preferable to phase for our purposes because it is an easily understandable parameter of sound; both mathematically as it denotes the number of repetitions of the waveform per second, and in the aural experience of sound as it is bound to our perception of pitch. This demonstrates that through simple operations such as introducing the coefficient 2π, instantaneous parameters can be reshaped in ways that are more convenient for users to understand.

It is easy to add a second instantaneous parameter, *amplitude*, by multiplying the entire output of the sine equation by a scalar (*A*).

$$x[n] = A\ sin(\frac{2\pi f n}{f_s})$$

Amplitude is correlated with a sense of dynamics or loudness, analogous to the correlation between frequency and pitch. It represents the vertical distance between the crests and troughs of the waveform, whereas the inverse of frequency, wavelength, represents the horizontal distance. In these ways they can be imagined as perpendicular dimensions of the audio signal; the two fundamental instantaneous parameters of a sine wave equation.

Instantaneous parameters like frequency and amplitude thus shape the dimensions of the audio waveform emanating from the machine. By contrast, the temporal parameters n and f_s have more to do with the amount of time that the signal utilizes. Other temporal parameters invisible to the equation are: n_{start}, the first value of n; n_{end}, the last value of n; and N, the number of samples, or n_{end} - n_{start}. These parameters do not impact the sound in the same way that the instantaneous parameters do. For example, a one-second sine wave ($N = f_s$), sounds the same as the first f_s samples of a 30-second sine wave ($N = 30\ f_s$). What the temporal parameters have in common is their relation to the signal's placement in time, without any contribution to the shape or sound of the signal at any given instant.

With the distinction between instantaneous and temporal parameters now established, the first design requirement for the project is primed for discussion. The goal for this music and sound production program is to offer the user as much control as possible over all parameters, both instantaneous and temporal. But it should also be recognized that the differences between these parameters require a different design approach. It is desirable to avoid the pitfalls which have affected other software due to assumptions based on convention or lack of musical intuition. The musical correlations of distinct parameter types and a methodology that will result in the most effective ways of modifying those parameters for the purposes of this project must be recognized in order to produce the maximum range of sonic possibilities. Accordingly, the first software requirement is:

1. The system must maximize control over the instantaneous parameters, such as frequency and amplitude, of all sounds.

As for the temporal parameters, there is a divergence between the way these parameters are represented in signal processing and the way they are conceived of in music. Western notation imagines time as a grid, and the modern DAW follows suit. The grid is divided by the number of beats per measure in the time signature, which can be any positive integer, most frequently three or four. The distance between the beats is determined by the tempo. If the user wishes to take advantage of the advanced time precision that digital mechanisms offer, this configuration is suboptimal. The parameters n_{start} and n_{end} are hard boundaries at both ends of a given signal, but every individual value of n between these boundaries can be a playground for signal manipulation. In the context of this project, mechanisms can be provided for distributing values evenly along a time-grid, but this should by no means be a requirement, and it should be easy to make subtle time manipulations, such as moving a sound event back or forward in time by a certain number of samples. The proposed program offers users complete control over the "field of time", actualizing the extent of John Cage's prediction in the second software requirement:

2. The system must maximize control over the temporal parameters, such as start time and end time, of all sounds.

One quality of sound has yet to be discussed: timbre. This sonic element is notoriously difficult to define with any degree of specificity, so it is commonly defined as all characteristics of sound distinct from pitch (or frequency) and intensity (or amplitude). Timbre is related to a signal's content on the frequency spectrum, which in periodic signals can be helpfully conceived of as a harmonic series. Therefore, a sine wave has the simplest timbre of any waveform because it only contains a fundamental frequency without any other frequency components.[7] With the onset of this investigation into ways of factoring in timbre, many more potential instantaneous parameters than simple frequency and amplitude begin to arise. More temporal parameters present themselves as well, for as different ways to affect timbre are sought, a conclusion emerges that changing the frequencies or amplitudes of sounds and their components dynamically over time can affect the timbre in diverse ways. On a zoomed-in timescale, dynamic changes in amplitude translate into changes in timbre or frequency spectrum. The greater the complexity with which a waveform varies in time, the richer its timbre becomes. Hence the third software requirement:

3. The system must contain mechanisms by which the control and modulation of parameters can result in sounds with unique and complex timbres.

[7] Except for negative frequency, which is not audible and therefore not relevant to this discussion.

This requirement relates to the ways in which a signal's instantaneous parameters are changed over time, which is where the boundary between instantaneous and temporal parameters blurs. If a set of operations are applied to the parameters of a signal over a defined period of time, there is a change in what is perceived as the timbre of the sound. An envelope is one common example of how this is done: it outputs a set of values which can affect the parameters, such as amplitude or frequency, of a signal over time. The parameters of the envelope itself are traditionally defined as *attack*, *decay*, *sustain*, and *release* (ADSR). However, there is nothing special about ADSR; it is simply a conventional way to think about the envelopes of sounds in Western music. Attack, decay, and release are just intervals of time, and sustain is but one of the output values of the envelope. The other output values of the envelope in an ADSR framework are implied: there is the initial value, or the point at which the ascending slope of the attack begins (implied to be zero); the peak value, or the uppermost point at which the attack ends and the decay begins (implied to be the peak of the amplitude or any other parameter controlled by the envelope); and the end value, or the point at which the downward slope of the release ends (implied to be zero). This framework is convenient for describing sounds which start with a zero value, rise to a peak, taper to a sustained value, and finally descend back to zero. The word "sustain" also implies the use of a keyboard instrument or controller, as this is the value at which the envelope's output remains when the user is holding down a key. The ADSR framework is a convenience for a particular type of music making, and for this project it is desirable to transcend it and find more comprehensive ways of describing envelope.

This thesis aims to satisfy the functional requirements described above. There is one additional non-functional requirement which is less crucial than the other three, but remains a noteworthy concern:

4. The system should exhibit a design that is visually pleasing and intuitive to composers.

It is important to have an enticing interface in any music software. It encourages creative flow by enhancing the user's mood, making them feel that they are in an aesthetically appealing environment. However, the author has kept in mind that GUI designs are always ripe for renovation without requiring substantial change to the internal workings of the project. The functional requirements have therefore been the focus of this research, with decoration of the interface kept to a minimum.

b. Design

The design of this project is object-oriented. Determining which objects to create and what data and functions they should contain required a full analysis of the dimensions and characteristics of sound. For example, one characteristic of sound is that it inhabits an interval of time. Therefore, one of the fundamental structures of the program is the Event, whose contents include a start time, an end time, and several pointers to Modules. Module is a base class encompassing any object that either directly creates sound (Audio Module) or modifies Parameters of Audio Modules (Control Module). Through the Control Module interfaces, the user can modify Parameters by adding Input Channels, which communicate with Output Channels that are already built into the Audio Modules. Every Event is a member of a Timeline object, which in turn is a member of a Project object. To summarize, the hierarchy descends as follows: Project, Timeline, Event, Module, Parameter, Channel. This object hierarchy allows the user to work at every level, from macro (zoomed out, as in the Project Window) to micro (zoomed in, as in the Parameter Window).

A class diagram (fig. 1) depicts the hierarchy described above. In the diagram, each class representation displays a few of the most important class members, as well as non-trivial, non-inherited member functions. It should also be noted that if a class inherits from another class (as denoted by a white arrowhead), it can be assumed that the derived class contains all the data and methods from the base class. This is not a thorough, detailed representation of the program, but rather a broad-strokes overview.

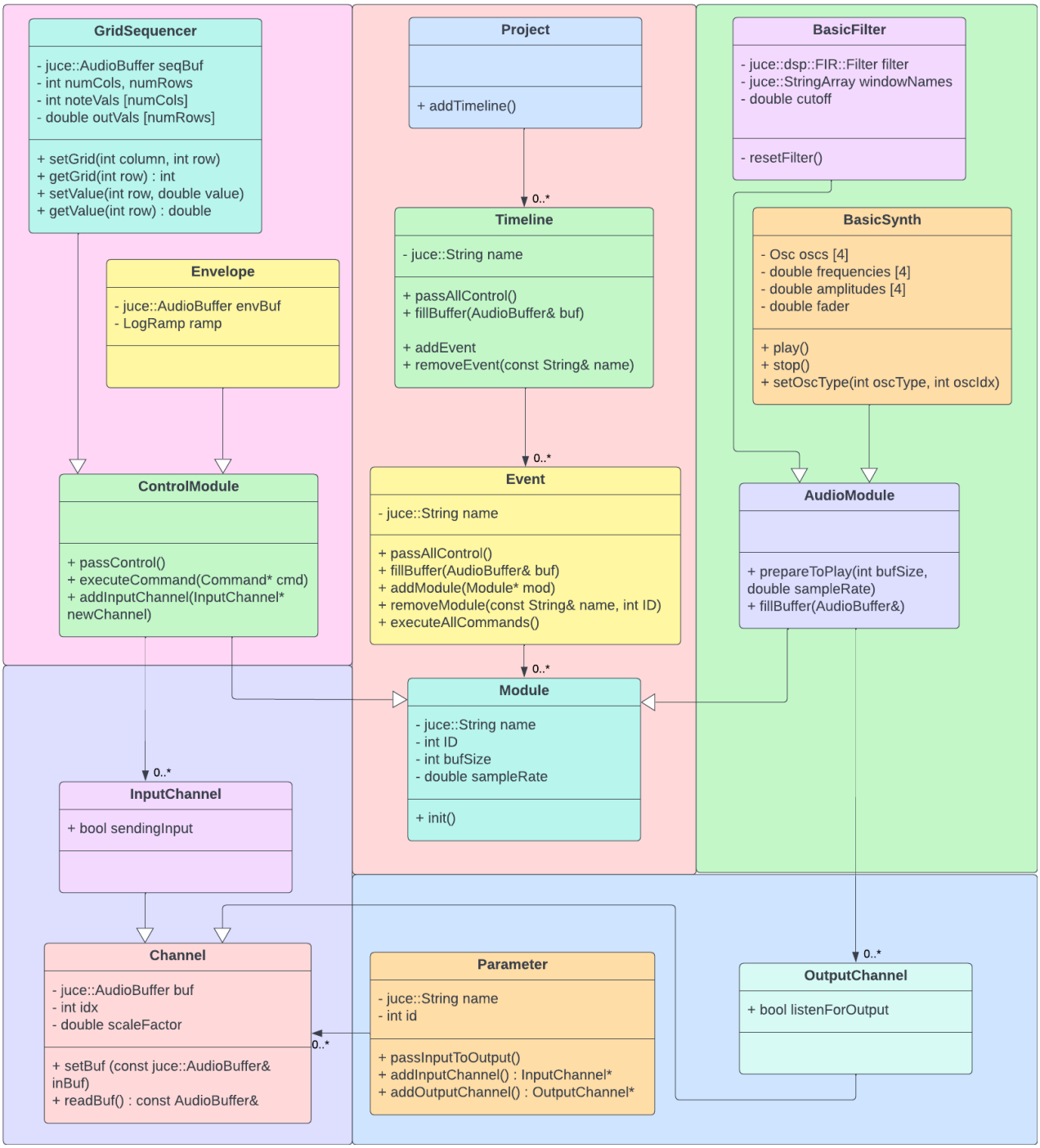


Figure 1: The class diagram indicates the hierarchical object structure of the project. See the Appendix for an overview of the UML symbols used here.

Figure 1 displays the hierarchy in which the objects in the program are organized, with the Project at the top of the hierarchy. The program automatically creates one object of the Project class at startup. The rest of the objects in the hierarchy are added dynamically by the user at run time. The user can add Timelines

to the Project, Events to a Timeline, and Modules to an Event. The Module class is where inheritance begins to come into play. Control Modules and Audio Modules are two derived classes of Module, each containing specialized member functions to suit their needs. Every Control Module needs functions for passing control data to any Audio Modules connected to it, executing a command telling it to send that data and adding Input Channels in order to connect an Audio Module.[8] On the other hand, every Audio Module needs functions for preparing to play audio and filling a buffer with audio. Audio Modules are responsible for creating their own Parameter objects, as well as Output Channels to accept data from those Parameters. The project currently has two Control Modules: Grid Sequencer and Envelope, in addition to two Audio Modules: Basic Synth and Basic Filter.

Despite bearing crucial importance for the project, many classes are not included in the diagram, as they do not elegantly mesh with the hierarchical structure shown above. These will be described in the following subsection on Programming.

Beyond software requirements and OO structure diagrams, the design of this project also consists of use cases. A documented use case can help guide the programmer toward developing the project for the anticipated needs of the end user. The following illustrates a basic use case:

A user creates a new empty project and is greeted with the Project Window. Here they initialize a Timeline; after populating the Timeline with Events, they can "play" the Timeline to hear all the Events in sequence. They click on the Timeline to navigate to the Timeline Window. They create an Event and click on it to navigate to the Event window. In the Event Window, there is a dropdown menu with a list of Control Modules, and another dropdown offering a list of Audio Modules. The user adds a Grid Sequencer from the Control Module menu and a Basic Synth from the Audio Module menu. When they click on the Basic Synth, they see the interface for the synth, which contains four panels, each presenting a frequency slider, an amplitude slider, a pan slider, and four toggle boxes labeled "Sine", "Square", "Saw", and "Triangle", respectively. Each of these four panels represents a synthesizer oscillator. The user clicks "Play" and hears the default sine oscillators. They adjust the frequency, amplitude, and pan sliders to create a unique blend of up to four voices.

[8] This is one point where language becomes a bit confusing. Input Channels and Output Channels are named from the perspective of the Parameter object. An Input Channel sends data to a Parameter; the data is input from the Parameter's perspective, even though it is output from the Control Module's perspective. Similarly, an Output Channel collects data from a Parameter and sends it to an Audio Module. From the Parameter's perspective, the data is output, although from the Audio Module's perspective, it is input.

Next, the user clicks the "Back" button to return to the Event Window, then clicks on the Grid Sequencer to navigate to its Window. There is a grid of sixteen columns and eight rows, forming a total of 128 squares which can be selected to create a pattern. Each row has a slider which can be adjusted to change the output value. There is a "Randomize" button which can be clicked to create a random pattern in the grid as well as random values in the sliders. There is also an "Auto-randomize" toggle that can cause the sequencer to automatically randomize at the end of each phrase, so that the pattern never repeats. Once the user has designed a pattern and set the output sliders to their satisfaction, they click on the bottom panel to connect a Parameter to the interface and open the Parameter Window. Because the Basic Synth has already been instantiated, the Parameter Window is aware of the possible Parameters that can be modified: the frequencies and amplitudes of all oscillators. The user can select the frequency for the first oscillator and then open an Input Channel into that Parameter. This will cause the synthesizer to "listen" for output from the Parameter as well, thus forging a connection between the sequencer and the synthesizer. The frequency played by the synthesizer's first oscillator will be controlled by the sequencer.

Now the user wishes to control when and how the sequencer is triggered. They navigate back to the Event Window and add a Command to the interface. The Command is an established design pattern in OO programming, referring to an object which encapsulates a request for an action to be taken (Gamma et al., 1995). Through the Command object, the user tells the sequencer when to trigger its pattern, and can specify whether the pattern should play once, multiple times, or on an infinite loop. More Grid Sequencers or other Control Modules can be added, as well as more Commands to trigger the other Control Modules. The user can play and overlap the patterns, or loop them all, adjusting start and end times as necessary. They can navigate back to the Timeline Window to simultaneously play all the Commands at the times specified. They can navigate back to the Project Window and play a Timeline, which will play all the Events at their specified times. Alternatively, they can play all the Timelines in sequence, and by arranging the Timelines in whichever way they choose, they can create an original composition. This is how all the pieces of the project come together in a modular way to allow the user to create their own work of art.

c. Programming

This project was programmed in C++ using the JUCE framework. The programming took place over several iterations. Some components of the project were developed in distinct code projects, such as the

Timeline Window, the Event Window, the Grid Sequencer, Envelope, and Basic Synth Modules, and even a Frequency Modulation (FM) Synth Module which is not available in this prototype version. It took time to develop ways for dynamically generated objects such as Events and Modules to communicate with each other smoothly, coming down to using static vectors of unique pointers to objects. Most of the classes in the program have a corresponding "Base" class which contains one such static vector of objects of that class.[9] For example, the Module Base class contains a static vector of unique pointers to Modules. These Module objects can be accessed from anywhere in the program by instantiating the Module Base as a static variable. The static property ensures that the same Module data will be accessed every time. In this way, Control Modules can send data to Parameter objects, and Audio Modules can retrieve the data to produce sound.

In addition to the Base classes, the other most important family of classes which are not present in the class diagram (fig. 1) is Window classes. The Window base class inherits from JUCE's Component class, which is a base class for any object that appears on the screen and accepts mouse input. Each derived Window class contains the GUI elements for interacting with its associated object; for instance, the Project Window (fig. 2) contains a button for adding Timelines to the Project. There is one Window class corresponding to almost every class shown in the diagram. Even some abstract base classes, such as Module, Control Module, and Audio Module have a corresponding Window class, from which the Windows for final Module classes are derived. The remaining figures in this section are screenshots of every Window in the program.

[9] The author wishes to draw attention to the distinction between "Base" classes (a concept specific to this project) and *base classes* in the sense of OO inheritance (see the Appendix for an overview of OO design terms).

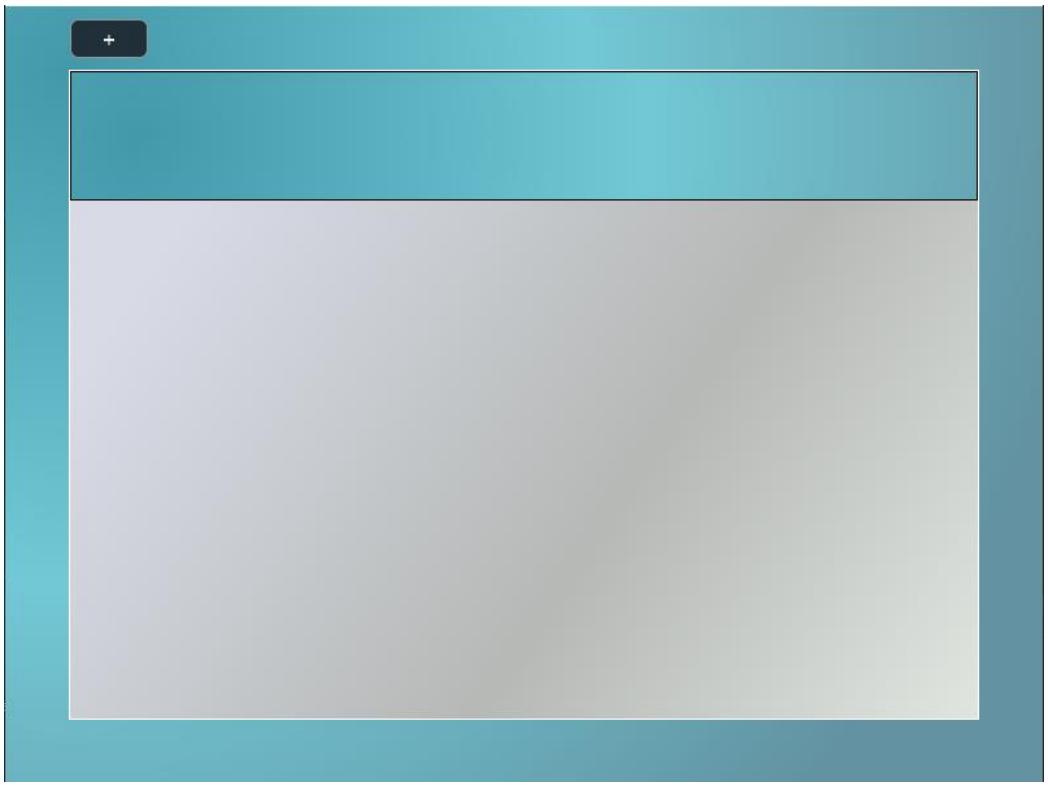


Figure 2: Project Window

In the Project Window (fig. 2), the user can add Timelines by clicking on the button labeled “+”. In this example, one Timeline has been added, as represented by the blue rectangle, known internally as a Timeline Rectangle. The user can left click on the Timeline Rectangle to navigate to the Timeline Window (fig. 3).

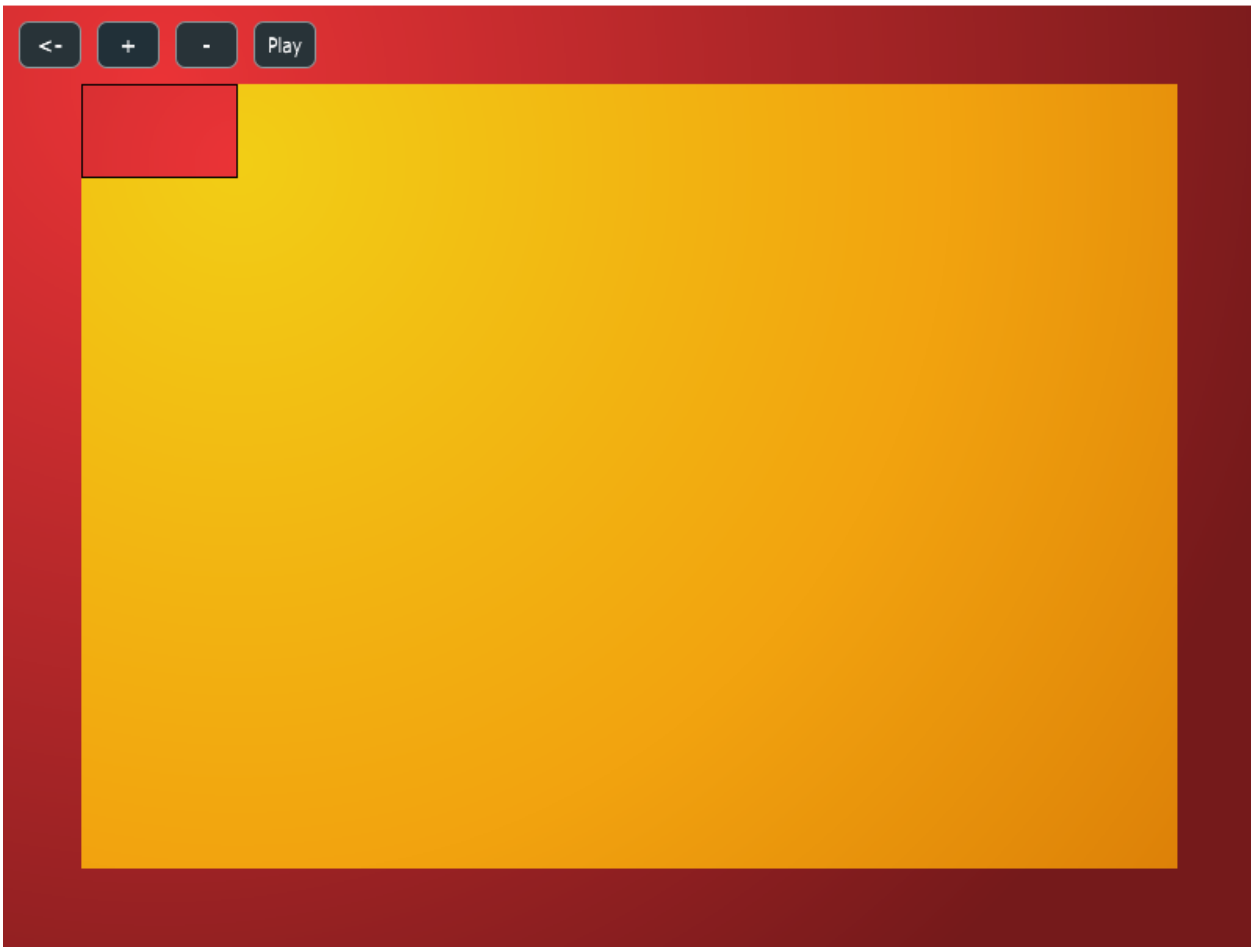


Figure 3: Timeline Window

From the Timeline Window (fig. 3), the user can add and remove Events to/from the Timeline. As in the Project Window, the user adds an Event to the Timeline by clicking on the button labeled “+”, and an Event Rectangle (the red rectangle in the figure) will appear. If the user right-clicks on the Event rectangle, a white border will appear around the rectangle indicating that it has been selected. When the user then clicks on the button labeled with a minus sign, the selected Event is removed (or multiple Events are removed, if the user selected more than one). If the user selects an Event and then clicks on the button labeled “Play”, all of the Event's Commands are triggered. The user can left click on the Event Rectangle to navigate to the Event Window (fig. 4), and the button in the top left corner labeled with a left-facing arrow (“<-“) navigates back to the Project Window (fig. 2).

Figure 4: Event Window

From the Event Window (fig. 4), the user can add and remove Modules and Commands to/from the Event. The four currently existing Modules are Envelope and Grid Sequencer (Control Modules), as well as Basic Synth and Basic Filter (Audio Modules). The first two buttons labeled "+" and "-" are for adding and removing Control Modules, and the dropdown menu just to the right is for selecting which Control Module to add. Further to the right is the same button and dropdown menu arrangement, but for adding and removing Audio Modules.

Commands are a way for the user to instruct a Module to do something—to play a single note or a set of notes, to loop the set of notes ad infinitum, or to stop playing. When the user adds a Command by clicking on the button to the far right labeled with a plus sign, a thin yellow rectangle containing two dropdown menus and text editor fields appears. This rectangle is called a Command Line. The Command Line's first dropdown list on the left shows all currently added Control Modules. It shows the name to

differentiate classes, and ID number to delineate instances of classes—as in “GridSequencer0” for the first Grid Sequencer. By default, this dropdown list selects the most recently added Module, if one exists. It automatically updates each time a Module is added.

The second dropdown list is for selecting which Command to execute when the user clicks the Play button (the conspicuous green triangle on the left side of the Command line). Currently available Commands are "Play", "Loop", and "Stop”. The final two items in the Command line (for now) are for the start and end times (measured in seconds). These can be entered into the text editor boxes by typing the value and pressing the return key. If the start time is not earlier than the end time, the input is invalid and thus ignored.

Figure 5: Basic Synth Window

In the Basic Synth Window (fig. 5), the four oscillators of the Basic Synth each possess sliders for frequency, amplitude, and stereo panning. If a Parameter begins to accept input from a Control Module

(such as the Grid Sequencer), the slider for that Parameter will cease to have any effect. Each oscillator can produce sine, square, sawtooth, and triangle waveforms, which can be selected using the toggle boxes at the bottom of each oscillator panel. To hear any output from the Synth, regardless of whether any Parameters are being controlled by a Control Module, the user must press the "Play" button.

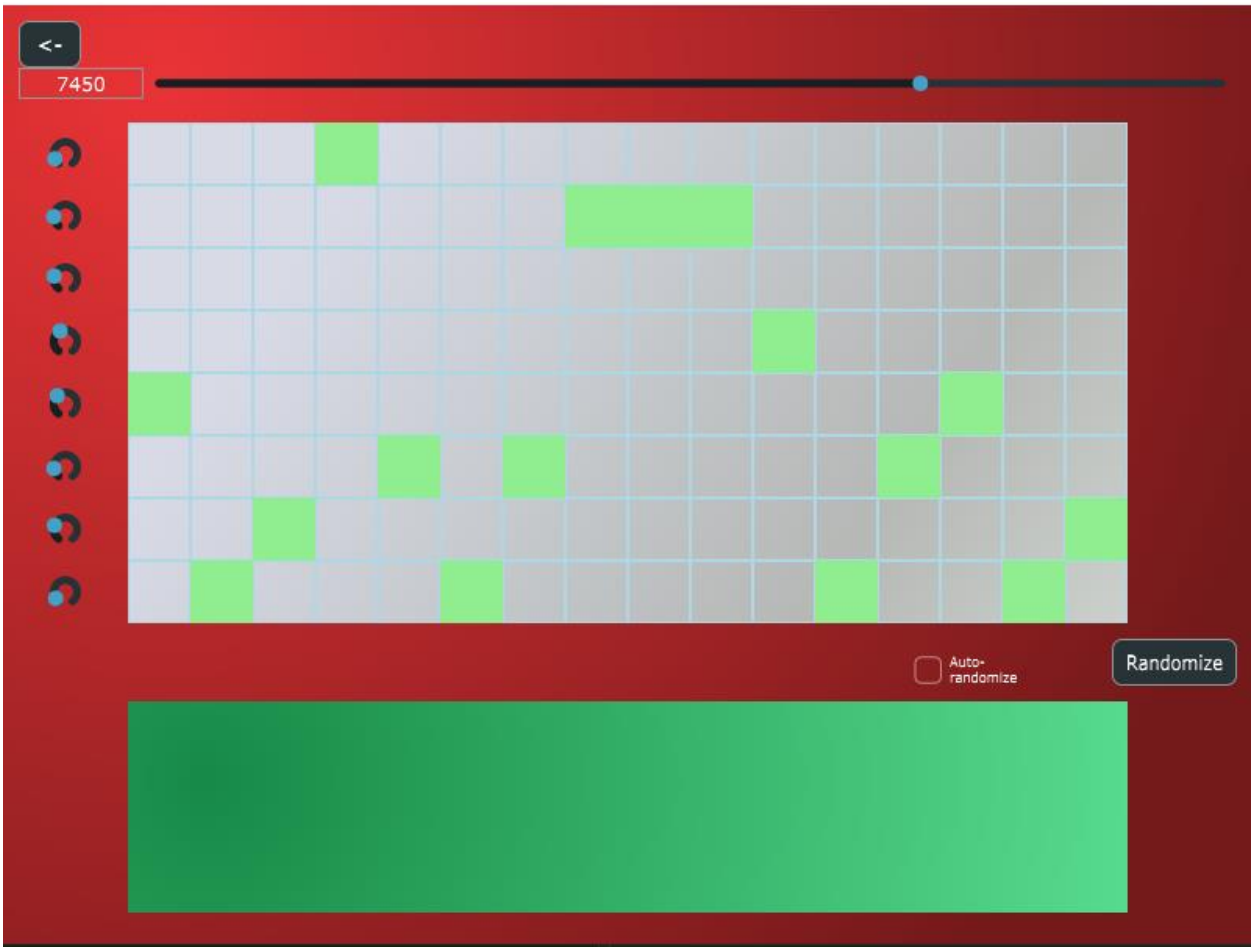


Figure 6: Grid Sequencer Window

In the Grid Sequencer Window (fig. 6), the user can create a pattern by clicking on the grid to change the note being played and adjust the dials on the left side to choose the value that each row of the sequencer will output. The slider at the top determines the length of each "note" in samples. Moving the slider to the left makes the tempo faster, and vice versa. The user can also press the "Randomize" button to generate a random pattern; the grid spaces and output dials are all set to random values when this button is pressed. The "Auto-Randomize" toggle button, when set to "true", tells the sequencer to systematically randomize the values at the beginning of each cycle. In this way, the pattern essentially never repeats. The green

panel at the bottom of the Window is for connecting the sequencer to an Audio Module via Parameters. The user can simply click on the panel to add a Parameter. A rectangle will appear, which they can select to navigate to the Parameter Window.

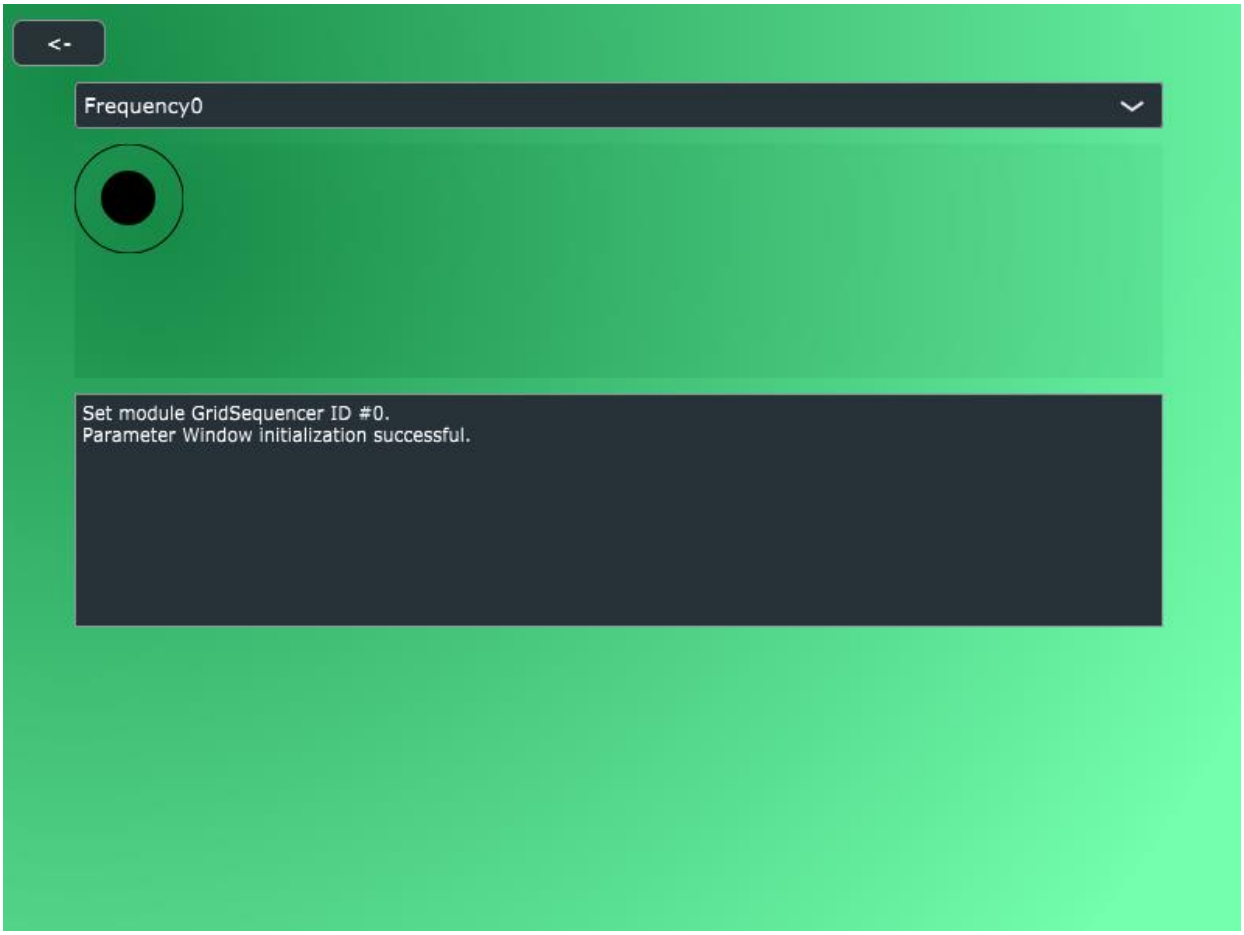


Figure 7: Parameter Window

The Parameter Window (fig. 7) is where the user can create Input Channels for sending data from a Control Module to an Audio Module. In the figure, an Input Channel has been opened to the frequency of the first oscillator of the Basic Synth. With this Channel open, the user can go back to the Event Window and send a "play" or "loop" Command to the sequencer, which will send the sequencer's output to the frequency of the oscillator. Multiple Grid Sequencers and Commands can be created to play various patterns simultaneously or in sequence. Thus, the Parameter and Channel structure facilitates communication between Modules. The debugging text field can be seen beneath the main Channel

interface, confirming the successful initialization of the Parameter Window for that instance of this Module.

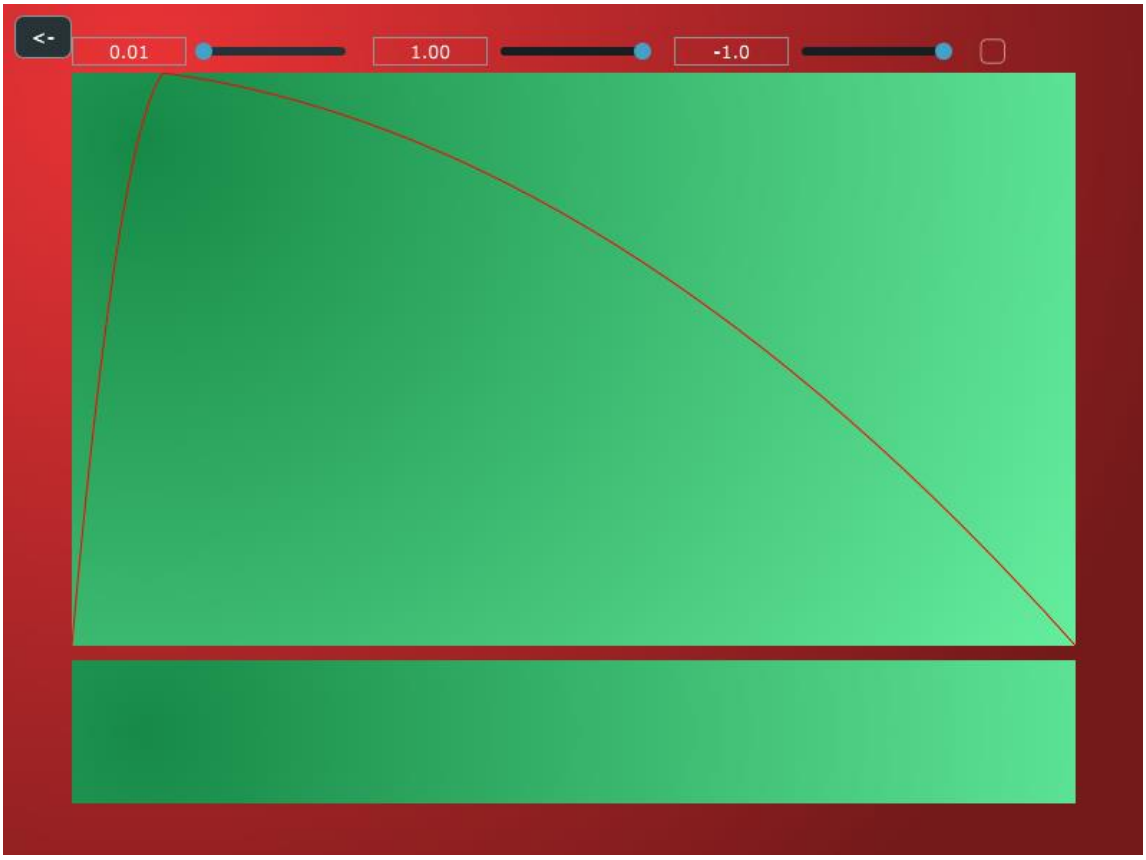


Figure 8: Envelope Window

The Envelope Module provides three sliders, one checkbox, and a visual representation of the Envelope curve. First is a slider that determines the attack time in seconds—the horizontal distance covered by the first line of the curve. The second slider determines the decay time in seconds, represented by the second line of the curve. Next is the third slider, the amplitude mid-point of the attack and decay in decibels, affecting the shape of the curve. The checkbox determines whether the slope will increase or decrease over time. Like the Grid Sequencer, the Envelope also has space for adding Parameter controls at the bottom, helping the user navigate to the Parameter Window to open a new Input Channel.

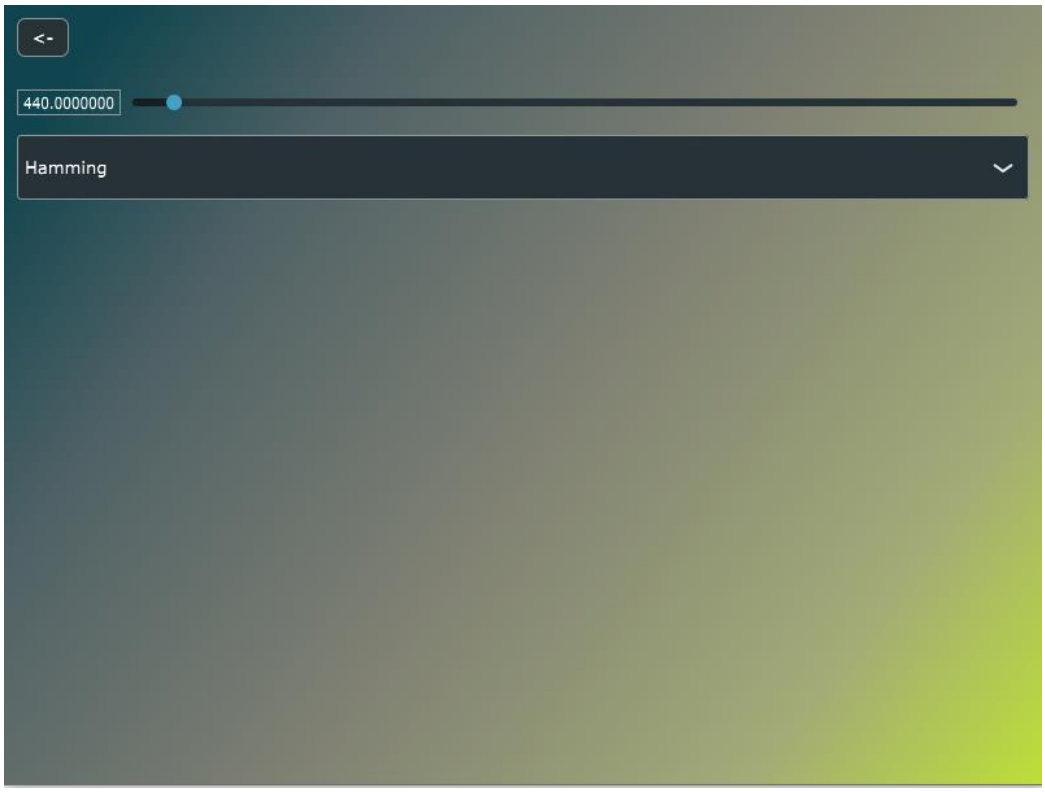


Figure 9: Basic Filter Window

The final Module is the Basic Filter. This Module offers a low-pass filter with a selection of available windowing functions. There is one slider to control frequency, and a drop-down menu to select the windowing function. The filter features a compulsory custom distortion that introduces a pleasant clicking texture (still under development).

RESULTS & ANALYSIS

An expert panel consisting of three professors in the Music Technology program at New York University evaluated this thesis for how effectively it met its core objectives. The first professor has at least 25 years of experience and is renowned in the field of audio coding and processing algorithms. The second has at least 28 years of experience and specializes in electronic music composition, computer music analysis, and DSP. The third evaluator has at least 45 years of experience, with special expertise in electronic music composition, music cognition, and software development for musical applications. Overall, the three professors possess extensive experience in both the creative and technical aspects of electronic music, as well as computer programming for musical applications, although their specialties vary significantly within these fields.

Panel sessions were conducted with each professor on an individual basis, each session lasting 30 minutes. The sessions consisted of a brief presentation about the project and its goals, a demonstration of the software itself, and then a round of questioning in which critical feedback was solicited. To refrain from biasing the professors' comments, and to allow more time for discussion and feedback, the presentation portion comprised only a brief, broad-strokes overview of the project and objectives. The five key goals, as articulated in the presentation, were:

1. *Interface*: An intuitive, inspiring, and aesthetically pleasing interface.
2. *Customization*: As much customization of sound as possible.
3. *Creativity*: Encouragement of experimentation and creativity.
4. *Flexibility*: As much as possible, a lack of bias toward any particular musical paradigm.
5. *Functionality*: No bugs and fast performance.

The questions posed to the professors were structured around these goals and expressed as follows:

1. How has the project reached or failed to reach its goals so far?
2. What ideas come to mind in terms of implementing these goals?
3. If you were a user, what would you want added or changed?
4. What are some concerns/considerations that may have been overlooked?

These questions stimulated a discussion where the professors outlined the ways in which they felt the project succeeded or had room for improvement. An account follows of how the professors evaluated the project according to each of the five goals.

a. Interface

The evaluators' opinions regarding whether the interface was intuitive, inspiring, and aesthetically pleasing exhibited variation. One professor said that the project had a "comprehensive, intuitive interface", whereas another professor found that the interface did not make sense. This range of responses indicates that the quality of an interface being "intuitive" is subjective, and therefore what might be intuitive to one individual may not necessarily be intuitive to someone else. Nonetheless, the experts did offer suggestions for improving the interface design. One expressed a need for additional labels to help the user identify the components on the screen. For example, it was suggested that the rectangles representing Timelines, Events, and Modules be labeled with their names and ID numbers, and that "tool-tips" be added which would display the name of the component and a brief description when a user hovers over it with their mouse. Another professor recommended adding the ability to copy data from one Module to another, so that the user would not need to begin from a blank slate every time they create and open a new Module. On a similar note, the professors requested the ability to save project data to a file; this feature is a core objective for future work. In general, they desired a more streamlined interface with less required configuration on the user's part. Overall, however, the evaluators praised the usability and visual aesthetic of the interface.

b. Customization

The professors found the ability to customize sound in the project satisfactory, with the caveat that there are only a few Modules available in the prototype version. They kept in mind that they were evaluating the project as a proof of concept rather than a finished piece of software, and therefore were generous in mentally extrapolating what had already been programmed to imagine what can be done going forward. The evaluators were impressed with the Parameter functionality and the ability to send data from Control Modules to Audio Modules via Commands. They also appreciated the addition of randomness to the Grid Sequencer, allowing the creation of stochastic patterns with endless permutations. One professor requested the incorporation of audio file read-in and playback functionality. This is planned via a forthcoming Sampler Module; however, it is not yet available for evaluation. In summary, within the constraints of the Modules created for this prototype, the customization of sound was deemed satisfactory.

c. Creativity

The professors concluded that encouragement of creativity and experimentation was one of the project's major strengths. One observed that the use of sinusoids and other simple waveforms for the synthesizer Module encourages users to experiment with basic tones. In contrast, another professor challenged the assumption of universality inextricable from this goal. He remarked, for example, that to create an interface that would satisfy "both Cage and Xenakis" would be virtually impossible. The professor's suggestion was to reframe the goal of the project to not aspire toward universal creativity, but to allow the project to fill the niche of a specific way of thinking about creativity. This professor also recommended a few ways in which the creativity and experimentation in the project could be enhanced, arguing that the project's structure could become more expansive if Events were able to trigger other Events, creating a generative process that is more fluid and less static. The evaluator also proposed the creation of a method for the user to insert their own code—something that can be done in Max/MSP. These potential features show promise as ways to push the envelope and extend the program's encouragement of creativity and experimentation.

d. Flexibility

In the presentation for the experts before each feedback session, the goal of flexibility was articulated as follows: “No bias toward Western music or any specific musical paradigm.” The professors took issue with the phrase "no bias” as used in this statement. They argued that it is impossible to create a tool that does not contain any assumptions about music composition or any bias toward a particular way of thinking. One professor argued that such a tool would be not only impossible, but undesirable, because a lack of bias would mean that there are no distinguishing characteristics that make the software unique and attractive to users. Better terms to simply describe the aim of this project are "flexibility" or "malleability". A flexible interface is one that allows for many unique styles of composition, especially outside of the Western musical paradigms that dominate popular composition interfaces. The evaluators also commented on the fact that this project, like any project, has its own baked-in assumptions. For example, the concept of working with an Event-based system of time is a particular way of thinking about music which is not necessarily the way all composers think. The grid-based sequencer itself is also representative of a particular way of thinking about music, especially since the grid is divided into sixteen

rows, reinforcing the Western musical convention of dividing rhythms into groups of four. There is no true escape from the assumptions and paradigms that find their way into the designs of the tools we create. However, this project sought to liberate composers from certain assumptions such as equal temperament and conventional meter, and the professors lauded the boldness with which the resulting program undermines these assumptions.

e. Functionality

The project's functional performance satisfied the professors. There are no issues with latency or CPU drain; the program is responsive and fast. However, the project is not entirely bug-free in its current interaction. An error occurred when deleting Modules from the Event Window: an exception was thrown during deletion, causing program execution to halt. There are also some functions that need further development to be more robust for future releases; for example, the Envelope Module does not yet offer the ability to play without looping, nor the ability to stop. The Filter Module also produces an unintended buzzing sound artifact. Upcoming iterations of the program will make a point of addressing these issues.

In terms of suggestions for improving the functionality of the program, one professor recommended creating an automatic "checking mechanism" running as a background thread, to ensure that all the variables are correctly set and that the objects are properly connected. Such a mechanism would help to smooth execution and prevent such errors from arising. More safety procedures of this kind are forthcoming in future versions of the program.

Overall, the professors were complimentary of the project, while also sharing thoughtful suggestions for improvement. One professor described the program as "beautiful" and "a nice bit of programming." The final verdict was that the prototype is on the path toward usability as a music composition and production tool; however, it needs further development and testing to become a fully usable, powerful interface for music creation.

DISCUSSION & FUTURE WORK

This project in its current state is the first step toward a fully functional piece of software supporting a liberated future for electronic music composition. The experts' comments, as reported and analyzed in the previous section, indicate that a variety of additional features and improvements could bring the project to new heights of success in achieving its goals. In this section, ideas and plans for the future of the project are conceived and elaborated, with the intention of transforming this proof of concept into a wholly operative computer music composition tool that accomplishes more of the project's goals. To that end, this section is organized in the same way as the previous section—the five goals of the project as previously described will illuminate the ways in which the project may be made more successful in achieving its intended objectives.

a. Interface

The author agrees with the experts' remarks that the usability of the interface could be improved by the addition of labels and/or tooltips on GUI objects, as well as the ability to save project data and copy data between objects. Users should be able to duplicate or copy and paste Timelines, Events, and Modules to save time and streamline creative flow. In addition to these suggestions, there are numerous features which should be added to increase the usability of the Windows. For example, the Project Window needs the ability to delete and rearrange Timelines. In addition, the process of adding Input Channels to communicate between Modules should be simplified. One idea is to make the connection from within the Event Window rather than requiring navigation to the Control Module Window and then the Parameter Window to open a new Input Channel.

One major way to improve usability is to make the GUI more customizable. Users should be able to easily change the color and name of any Timeline, Event, or Module. This would allow the user to design their own composition environment, making the program more inspirational and suitable for their unique compositional style. The GUI should also be more informative about what is going on in the internal structure of the project. For instance, in the Project Window, a Timeline Rectangle should show a simplified diagram of the Events, Commands, and Parameters within that Timeline, to improve navigability within the interface and make it easier for users to work on a macro level.

Many software programs have either a static header bar or a sidebar that remains visible as part of the GUI, and contains core operation categories such as “File”, “Edit”, “View”, etc. Implementing the ideas described above as well as other features could be simplified by the addition of a menu bar of this kind. One advantage of this approach is its familiarity to most if not all users, as most programs use this structure. However, a disadvantage is the potential disruption of creative flow caused by the intrusion of a verbose menu bar that needs to be navigated to access certain features. The minimalist approach that the interface currently exhibits is attractive due to the potential for creative immersion without the interference of language. Keyboard shortcuts and right-click dropdown menus would be preferable to the use of a menu bar. If a menu bar must be added, it could be hidden until the user clicks in a particular location, such as a “menu button” in a corner of the interface. These interface-related considerations will be part of the design process moving forward.

b. Customization

To improve the ability to customize sound, the project requires a greater diversity of Modules. The anticipated Sampler Module, for instance, would be able to read and play back audio files. Ideally, this Module will provide all the basic features one expects from a sampler beyond simple playback, such as changing pitch, playing in reverse, and adjusting the playback start and end points. Being able to import audio files will allow for any captured sound to become a part of the project, enhancing the sonic variety of the user’s output.

Another forthcoming Module is the Frequency Modulation (FM) Synthesizer. FM synthesis is the process of using oscillators to dynamically modify (or “modulate”) the frequencies of other oscillators, creating a timbre entirely different from that of the oscillators without modulation. Incorporating the FM Synth Module will expand the range of timbres that users can produce, allowing for enhanced sound customization. This is especially true if the Envelope Module or other Control Modules are used to modify the frequencies of modulating oscillators in the FM synthesizer, because doing so causes the timbre to morph over time, paving the way for new sounds of great complexity and dynamism.

The prototype for the forthcoming FM Synth Module is shown in Figure 10. There are two oscillators: the carrier and the modulator (future versions may increase the number of oscillators to allow for even more customization). The slider labeled "cFreq" determined the carrier frequency, "mFreq" the modulator frequency, "cAmp" the carrier amplitude, and "mAmp" the modulator amplitude. The four text editors

and buttons labeled "Set range" can be used to adjust the range of each of the sliders respectively, allowing for a greater degree of precision in Parameter control. The four toggles labeled "Sin", "Squ", "Saw", and "Tri" are used to select sine, square, sawtooth, or triangle waveforms for both oscillators simultaneously; forthcoming versions will allow for separate waveform control for each individual oscillator. The FM synthesizer already provides powerful capabilities sound customization, and future versions will expand these capabilities even more, while also assimilating the FM Synth as a Module in the larger project, allowing it to interact with the Grid Sequencer, the Envelope, and any other Control Modules which may be added in the future.

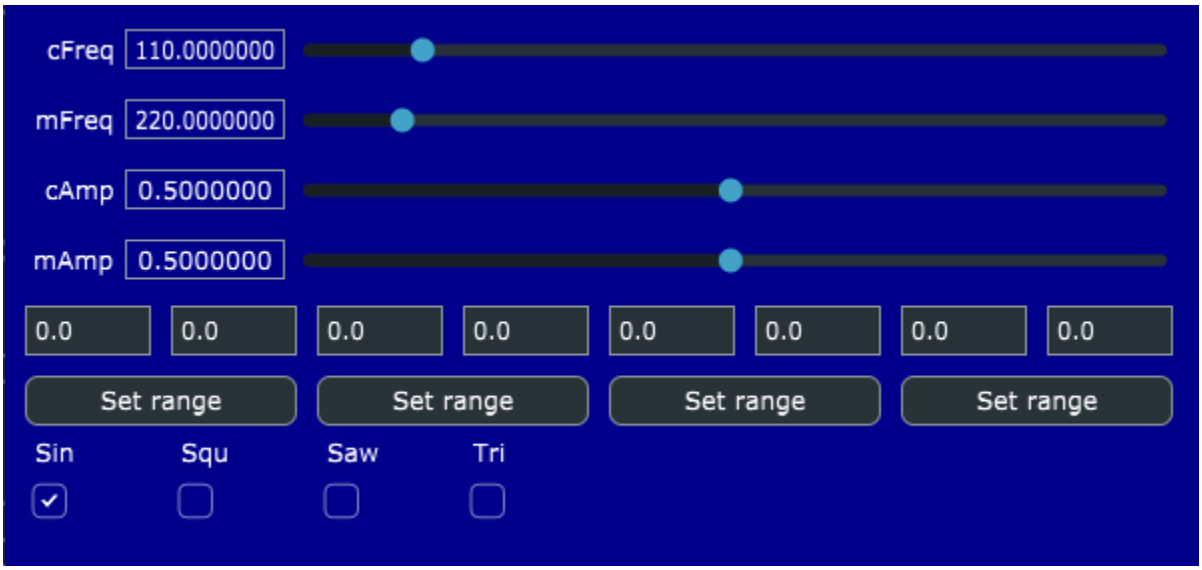


Figure 10: The forthcoming FM Synthesizer Module interface.

Beyond adding more Modules, the ways in which Modules can communicate with each other also need to be expanded. Currently, the interactions between Control Modules and Audio Modules are limited to the Grid Sequencer and Envelope manipulating the frequencies and amplitudes of oscillators in the Basic Synth. It is not currently possible to have more than one Control Module affecting the same Parameter simultaneously. Ideally, the connections between Modules should be made easily and intuitively. In addition, there needs to be a more customizable and effective way of controlling signal flow. For Modules that process audio data, there should be a way to directly control which signals pass through the Modules and when. No such mechanism presently exists. Communication between Modules was one of the biggest concerns during development, and the program would still benefit from improvement in this area.

c. Creativity

Expanding the number and diversity of existing Modules as well as the ways in which Modules can interact with one another would heighten the creative possibilities of the project. More interface customization could also improve the inspirational power of the project, as noted in subsection A. Another way to improve creativity is to introduce features that lean toward unconventional composition styles, such as generative composition. As noted in the previous section, one professor suggested Events which can generate other Events, establishing a self-sufficient creative process that takes advantage of the generative powers of the program. This feature could become a compelling source of new and interesting sounds, emboldening users to explore creative possibilities they may not have otherwise considered.

In addition to the Sampler and FM Synthesizer Modules discussed previously, a suite of audio processing effect Modules would also be a desirable addition for encouraging creativity. Reverb, delay, and compression are three of the most basic and commonly used audio effects, all capable of dramatically altering sonic character. Conventional audio effects such as these will broaden the artistic possibilities of existing sounds. However, there should also be unique audio effects that are not necessarily available in every music production environment. Granular time processing and spectral processing come to mind as two areas in which creativity could be expanded beyond the range of what is natively found in a conventional DAW or audio effect rack. The more audio processors and effects added to the project, the more the possibilities will expand, spurring creative flow to thrive in this virtual music creation environment.

d. Flexibility

The Grid Sequencer Module, in its current prototype version, is a good starting point for tonal and rhythmic pattern design and arrangement. Nonetheless, there are several ways in which its flexibility could be improved. Adding the ability to change the number of columns and rows in the sequencer is one way to enhance its flexibility. Additionally, incorporating a separate note length slider for each individual column could allow variable-length patterns to be devised, enabling irregular rhythms outside of conventional meter. If these features were implemented, the Grid Sequencer could behave less like a sequencer and more like a versatile melody or pattern composer, allowing for more rhythmic and tonal possibilities to be introduced and experimented with.

The Envelope Module is likewise a good starting point for designing parameter control curves, but could also be made more adaptable to compositional needs. The Envelope currently offers two phases of change: attack and decay. By contrast, the forthcoming improvements for this Module will allow the user to add a theoretically unlimited number of phases to the Envelope curve. This is accomplished through the use of *breakpoints* and *control points*. The prototype for the improved Envelope interface is shown in Figure 10. The red dots on the Envelope curve are breakpoints, whereas the green dots are control points. The button labeled "+" can be clicked to add a breakpoint (and a control point, if the breakpoint is not the first to be added), while the button labeled "clear" erases the entire set of Envelope data and clears the curve display. The two sliders on the left control the X and Y coordinates of the breakpoint to be added, while the two sliders on the right do the same for the control point. Users can click and drag on the breakpoints in all directions on the two-dimensional plane to design the envelope curve. The interface for this new and improved Envelope Module is currently in the process of testing with audio. The two-phase Envelope will be replaced with this more versatile Envelope in future versions.

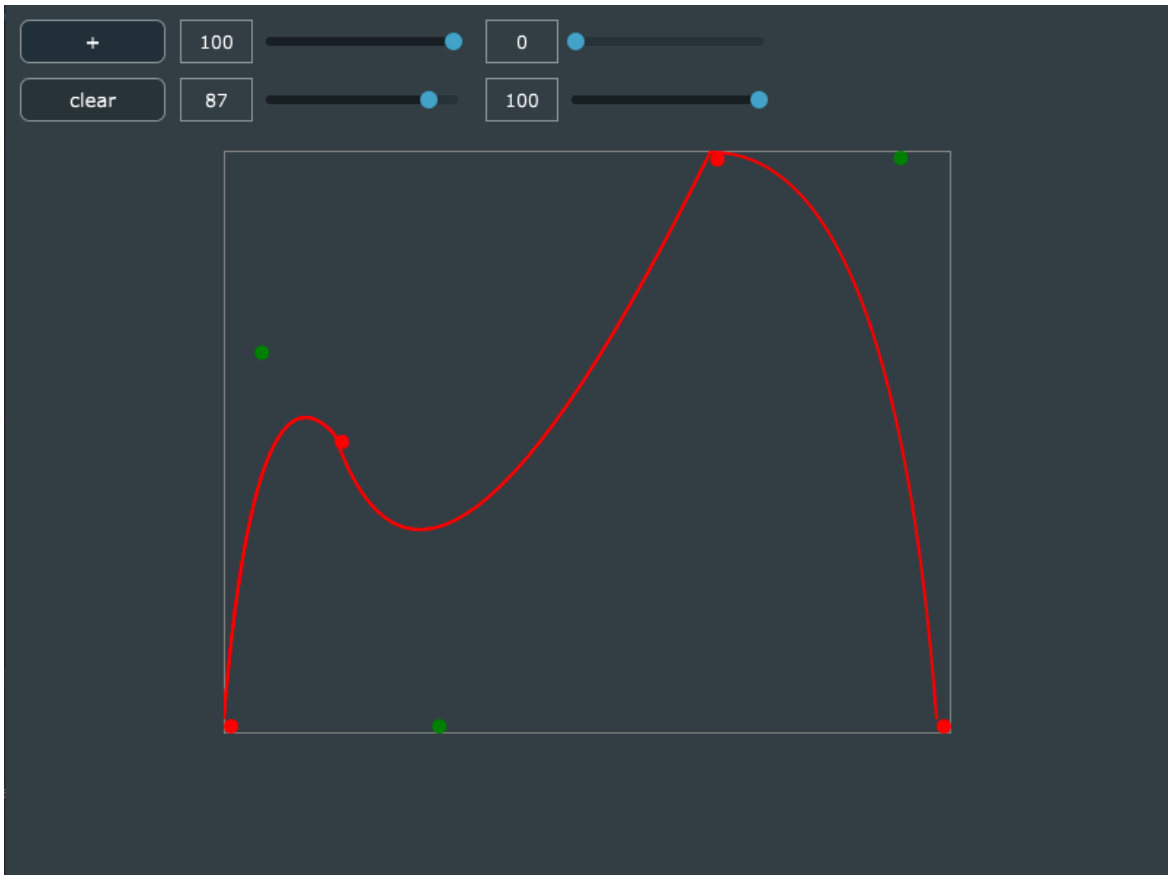


Figure 11: The forthcoming Envelope interface design.

Many of the ideas mentioned in previous sections—adding more Modules, expanding communication methods between Modules, introducing generative compositional methods—would also improve the flexibility of the interface. In general, the more the project grows in both scope and diversity, the more flexible it will become, and the broader the variety of sounds and musical patterns that can be created with it. Making the project open-source may be advantageous, to empower other programmers who have ideas for Modules to create them independently. This could contribute to the expansion of the project and its flexibility for all users.

e. Functionality

Although performance latency does not appear to be a limiting issue at present, the project still contains some bugs to eliminate. One of the best ways to prevent bugs in code is to use safety checks. For example, if it is known that a certain variable will cause errors if it does not match a certain value or range of values, the program can check if the variable is set to an appropriate value, and if it is not, throw an exception or initiate a process that will attempt to fix the problem. The code in its current state does implement this sort of solution for various components, but a more comprehensive error handling system would prevent bugs and keep the program running smoothly. Measures for ensuring fast and smooth performance will continue to be pursued as the project grows.

CONCLUSION

This thesis was sparked by an intense curiosity about how music composition interfaces could be made more creatively inspiring and flexible, as well as a fervent desire to create something that falls outside the limitations of conventional DAWs, to allow for the greatest possible exploitation of modern computer processing capabilities. The project strongly emphasizes the different dimensions of sound (frequency, amplitude, etc.) and the ways in which they can be manipulated.

This research was informed by the recent history of avant-garde music composition and inspired by the works of contemporary electronic experimenters such as Iannis Xenakis, Morton Subotnick, and Curtis Roads. Inspiration from the work of music technologists behind innovative and unique music production tools was also influential to the project's development.

Preliminary analysis consisted of requirements and use cases, each helping to clarify the vision and goals for the project. The software architecture was designed using an object-oriented framework with dedicated diagrams to make the structure clearer. Programming took place over several iterations, initially fragmented but eventually coalescing into one large project. The resulting program contains a Project, Timeline, Event, Module, and Parameter structure, with four Modules available for use: two Control Modules, and two Audio Modules. At the time of evaluation, the project encompassed ninety-four source files and 5,331 lines of code.

Feedback was given by a panel of three experts at NYU Steinhardt's Music Technology program. The experts offered a variety of helpful suggestions related to each of the project's five goals as they were articulated to the panel: interface, customization, creativity, flexibility, and functionality. The experts were satisfied with many aspects, but they saw room for improvement in several areas as well. These suggestions and comments will be considered during further work on the project.

This thesis represents the beginning of a larger series of explorations into the creation of music and sound interfaces.

REFERENCES

Barri, T. (2009). Versum: audiovisual composing in 3D. *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 264-265. http://doi.org/10.5281/zenodo.1177473

Boulez, P. (1971). *Boulez on music today*. Faber and Faber.

Cage, J. (1961). *Silence: Lectures and Writings*. Wesleyan University Press.

Cassorla, A., Kearney, G., Hunt, A., Riaz, H., Stiles, M., & Murphy, D. (2020). Augmented reality for DAW-based spatial audio creation using smartphones. *Journal of the Audio Engineering Society*.

Damschröder, D. (2012). *Harmony in Haydn and Mozart.* Cambridge University Press.

Dewey, C., & Wakefield, J. (2018). Elicitation and quantitative analysis of user requirements for audio mixing interface. *Journal of the Audio Engineering Society*.

Dewey, C., & Wakefield, J. (2016). Novel designs for the audio mixing interface based on data visualisation first principles. *Journal of the Audio Engineering Society*.

Fowler, M. (2004). *UML distilled: applying the standard object modeling language.* Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Honigman, C., Walton, A., & Kapur, A. (2013). The Third Room: A 3D virtual music paradigm. *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 29-34.

Jenkins, J., ed. (2016). *Schoenberg's program notes and musical analyses.* Oxford University Press.

Ji, C. (2020). Sakura: A VR musical exploration game with MIDI keyboard in a Japanese Zen environment. *2020 IEEE Conference on Games (CoG), Osaka, Japan*, pp. 620-621.

Lanier, J. (2010). *You are not a gadget: A manifesto.* Knopf Doubleday Publishing Group.

Lopes, F. & Guedes, C. (2020). Composing music with a space. *Perspectives of New Music, 58*(1), pp. 5-22.

Mazzola, G. (2019). Consciousness, creativity, and complex time in music. *Perspectives of New Music, 57*(1-2), pp. 431-439.

Nelson, M., & Thom, B. (2004). A survey of real-time MIDI performance. *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 35-38.

Polfreman, R. (2009). Frameworks 3D: Composition in the third dimension. *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 226-229.

Reece, A. (2013). Constructionist learning with Pure Data: A graphical environment for audio and MIDI programming. *Journal of the Audio Engineering Society*.

Roads, C. (2001). *Microsound.* MIT Press.

ROLI. (2022). *MIDI Polyphonic Expression*. https://roli.com/mpe

Wishart, T. (1996). *On sonic art*. Overseas Publishers Association.

Xenakis, I. (1971). *Formalized music: Thought and mathematics in music.* Pendragon Press.

Welburn, S., & Plumbley, M. (2009). Rendering audio using expressive MIDI. *Journal of the Audio Engineering Society*.

Wu, J. C. (2020). From physical to spiritual: Defining the practice of embodied sonic meditation. *Organised Sound, 25*(3), pp. 307-320.

APPENDIX

a. Object-Oriented Design and Programming in C++

This section contains an overview of the basic Object-Oriented (OO) principles for individuals unfamiliar with OO paradigms or C++. It is recommended reading in order to understand the Methods section of this thesis. Code examples are given to solidify the concepts.

Classes

A *class* is a set of data, and a set of operations that can be performed on that data. An *object* is an instance of a class. When a program has multiple instances of a class, the computer's memory contains multiple copies of the data contained within the class. For example, the following C++ code defines a class that represents a point in 3D space with integer x, y, and z coordinates:

```
class Point3D
{
      int x{ 64 }, y{ 35 }, z{ -52 };
};
```

Although this code defines a class, it does not define an instance of that class. In other words, the class has not been *instantiated*. This means that an object has not been created, so the class's member variables have not been stored in the computer's memory. One line of C++ code instantiates the class and creates an object in memory:

```
Point3D point1;
```

Here an object of *type* Point3D, with the *identifier* "point1", has been instantiated. Identifiers are also known as variable names. They help to distinguish between different instances of the same class or other classes. Memory addresses (pointers) and class member data are two other ways to distinguish between instances of the same class.

Encapsulation

There is now an instance, "point1", of the Point3D class. However, this object as it is currently defined cannot be accessed or modified in any way, because of its *accessibility* permissions. The accessibility of the class members x, y, and z is *private*, meaning that it is not possible to access or modify these variables outside of the class definition itself. This is because the accessibility level of the class was never specified, and in C++ the default accessibility for class members is private.[10] In order to make it possible to modify or access these variables outside of the class, one of two actions must be taken: either the class member data must be set to *public*, or public *member functions* (also known as *methods*) that can modify or access the variables must be created.

The principle of *encapsulation* says that it is best practice to make all class member data private. If a variable needs to be accessed or modified outside of the class definition, public member functions should be written which can do that.[11] Encapsulating the variables in public member functions allows the programmer greater control over when and how the class members are modified or accessed, and better ensures that they will not be modified or accessed in unanticipated ways. Encapsulation is the idea that it is best to keep data contained as private class members so that they can only be modified or accessed in the ways the programmer(s) intended. Functions that modify class data are often called *setter* methods, while functions that access class data without modifying it are called *getter* methods.

So, if we redesign the class according to the principle of encapsulation, it will look something like this:

```
class Point3D
{
public:
      void setX(int newX) { x = newX; };
      int getX() const { return x; };


      void setY(float newY) { y = newY; };
      int getY() const { return y; };
```

[10] A *struct* behaves the opposite way; its members are public by default. This is the only functional way in which classes and structs differ in C++.
[11] There is a third level of accessibility, *protected*, in which class variables are accessible only in the class definition *and* in the definition of any derived classes (see the next subsection, "Inheritance").

```
    void setZ(std::string newZ) { z = newZ; };
    int getZ() const { return z; };
private:
    int x{ 64 }, y{ 35 }, z{ -52 };
}
```

For readers unfamiliar with C++, do not worry about any details of the above code that lack clarity. It will suffice to recognize that the class now contains public member functions which can be called from outside the class to either set (modify) or get (access) the member variables of the class. Keeping the variables themselves private like this can prevent errors such as accidentally modifying a variable when only read access was intended. It also allows the programmer(s) to put additional code in the setter and getter methods if they so desire. For instance, they might want to send a message to the GUI to change the point's position on the screen every time its coordinates are changed, or keep track of how many times the coordinate is accessed for debugging purposes. Any code can go into the setter and getter methods.

Inheritance

In the following code, the example is expanded by creating three different colors of 3D points. The colored points will be RedPoint, BluePoint, and GreenPoint. These three classes will all contain x, y, and z integers, just like the Point3D class. One way to make this happen would be to rewrite or copy and paste the same code three times. This is called code duplication; it is considered poor style, and is also computationally inefficient. Inheritance is one of many ways to avoid code duplication. The shared data and functions can be defined once, in a class known as the *base class*. By inheriting from the base class, there can be as many *derived classes* as desired, all of which share the data and functions from the base class. Here are three classes derived from the Point3D class defined above:

```
class RedPoint : public Point3D
{
   std::string color{ "Red" };
};
```

```
class BluePoint : public Point3D
{
   std::string color{ "Blue" };
};

class GreenPoint : public Point3D
{
   std::string color{ "Green" };
};
```

Three derived classes of Point3D are defined above. Because they inherit from Point3D *publicly*, they all have the private integer variables x, y, and z, as well as public setter and getter functions for these variables. Any derived classes of Point3D will automatically contain all of its data and functions.

Polymorphism

To demonstrate polymorphism, a class called "PaintableObject" will be defined. Any class that needs to be painted on the screen should inherit from this class. Derived classes will inherit a member function called "paint" which takes one parameter: a reference to an object called Graphics (which is assumed to contain member functions "drawPoint" and "drawRect" to draw points and rectangles to the screen).

```
class PaintableObject
{
public:
     virtual void paint(Graphics& g) = 0;
}
```

Here, the *virtual* keyword indicates that the "paint" member function can be redefined (in other words, change its *implementation*) in derived classes. Different derived classes of PaintableObject will need to implement the "paint" function in various ways, because, for instance, drawing a point is not the same operation as drawing an ellipse or a rectangle. "= 0" indicates that the function has no default implementation; that is to say, it is a *pure virtual function*. A class that has any number of pure virtual functions is called an *abstract class* because it can only be instantiated in the form of a derived class

which implements all its pure virtual functions, never directly. Conversely, a class that can be instantiated directly is called a *concrete class*.

Point3D will inherit from the PaintableObject class and implement the paint function as follows:

```
void Point3D::paint(Graphics& g)
{
      g.drawPoint(x, y, z);
}
```

Another way of implementing the paint function would be necessary for a different PaintableObject derived class, such as one called Rectangle:

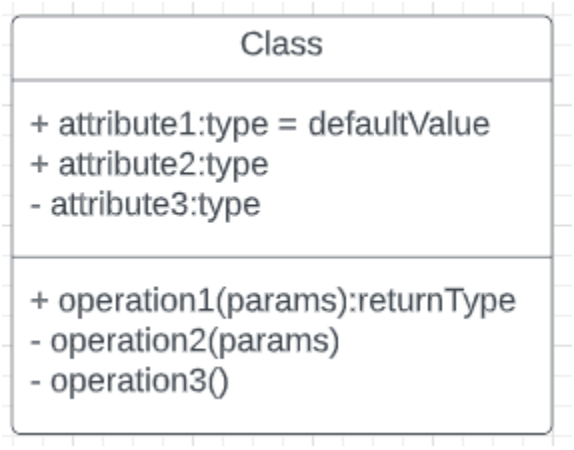
```
void Rectangle::paint(Graphics& g)
{
      g.drawRect(a, b, c, d);
}
```

The result is that the paint method can be called on any object that inherits from PaintableObject. The caller does not have to know which derived class they are calling the method on. If the class inherits from PaintableObject, they can call the method and it will paint the object to the screen, regardless of what the object looks like or how the paint method is implemented. This is the essence of polymorphism: "poly" means many, and "morph" means shape. Different objects with different "shapes" can be used in exactly the same way. Virtual functions accomplish this in C++.[12]

b. Universal Modeling Language (UML)

Now that the fundamentals of OO design and programming have been clarified, the meanings of UML diagram symbols can be explained.

[12] Actually, polymorphism is accomplished in two ways in C++. Virtual functions offer run-time polymorphism, while templates offer compile-time polymorphism. The distinction between these (and an explanation of what templates are and how they work) is beyond the scope of this text.



The three-tiered rounded rectangle above represents a class. The top level is the name of the class, the second level shows its attributes (or member variables), and the third level shows its operations (or member functions). "+" indicates public accessibility, and "-" indicates private.

A black arrowhead indicates a class that is *contained* within another class, as a member variable. An object that is a member variable of another class can be represented either in the class attribute list, or as a black arrowhead connecting the two classes.

A white arrowhead with a black outline indicates inheritance. The class with the arrowhead facing it is the base class.

Multiplicity (a number shown at the base or head of an arrow) indicates the number of objects of a class in the program. If there is no multiplicity, a multiplicity of one is implied. The most common multiplicity indicator is "0..*" which indicates zero or more objects of a given class.