An Eye Tracking Interface for Musical Expression for People with Disabilities

Matthew C. Dunlap

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: Leila Adu-Gilmore
Reader: Robert Rowe

May 3, 2020

Abstract

An exploration of various interfaces for musical expression led to the creation of an interface that allows a person who lacks the use of their limbs to be able to perform any composition created in the Ableton Live digital audio workstation. This system was devised using a combination of the Unity gaming engine, Max for Live, and Ableton so that the interface would be easy to install and as flexible as possible. In addition, the system utilizes common eye tracking hardware made by Tobii Dynavox that may already be in use by a person who has limited functionality of their limbs. In order to demonstrate the feasibility of the newly created interface, a composition was written and performed. During the creation of the composition and rehearsal of the performance, the product was refined until it functioned properly and could allow some amount of improvisation. The result was an interface that could control any parameter within Ableton using a combination of eye gaze and head motion. Furthermore, the user can trigger scenes within Ableton to control the flow of the composition. This product could conceivably allow a person with disabilities to completely compose a piece and perform it without the aid of another person.

Acknowledgements

I would like to thank my sister, Sierra Rose Zachery (Rosie), for inspiring me to explore the topic of this thesis. It is my hope that through the use of technology, people with disabilities may be free to express themselves. May this research and others that come after afford people like Rosie more abilities that many of us take for granted.

Contents

List of Figures

1 Introduction

Countless interfaces exist for the purpose of expressing oneself musically. However, many of these interfaces require the use of fine motor skills or the use of limbs, which people with physical disabilities may not have. In fact, there are those who are locked into a body that does not allow for any expression other than the use of their eye gaze. These people are sealed within their physical bodies, and lack the ability to express themselves in meaningful ways. This can be a horrifying prospect, considering that previously someone may have been able to perform meaningful expression, and then, through some accident or degradation of their body, lost this ability. Additionally, other people may have never been able to express themselves and are just an intelligence trapped behind a gate waiting to be let out. Think about how many musicians and great musical works have been lost due to disability. Therefore, it is the goal of this project to research and develop a way to harness this form of expression, namely eye gaze, to allow for a physically disabled person to express themselves musically.

Playing keyboard, string, wind, and percussion instruments requires the use of both hands and/or fine motor movements in the mouth. While useful for expression to the able-community, these common interfaces exclude a group of the population who could benefit from self-expression through music. A family member who is severely physically disabled has the ability to nod and vocalize to an extent, but her primary method of communication is eye gaze. With this outlet of expression, she may interact with others and make her wants and needs known. While eye gaze can be an effective tool for communication to close family and medical personnel, none of the established musical interfaces effectively utilize

this resource for musical input. Those interfaces that do allow for creation of music via eye gaze do so utilizing a grid of icons [Vamvakousis and Ramirez, 2016, Polli, 1999, Hornof, 2014], which does not lend itself well to real-time musical input, the way an interface such as a keyboard does.

Early eye-tracking hardware was bulky, cumbersome, invasive, and very expensive [Polli, 1999]. However, eye tracking hardware is now entering the consumer market [Tobii Technology AB, 2019a] and it has become possible to create an interface with affordable, off-the-shelf hardware. In addition, software development tools now exist for rapid development of gaming software. Since the target audience for the consumer eye tracking devices are gamers, it makes sense to utilize the same tools that are used to develop gaming software. Therefore, it is the goal of this project to create a musical interface utilizing the Unity [Unity Technologies, 2019] game engine development environment and Max [Cycling '74, 2019] visual programming language that would allow a person with severe disabilities to perform music in real-time. Instead of using a grid of icons or a menu-driven solution, the project will explore a unique approach using three-dimensional objects and algorithms to create music. It is the hope that doing so will unlock the potential that is stored inside of musicians who are unable to express themselves otherwise.

2 Literature Review

As demonstrated through several performances and projects [Vamvakousis and Ramirez, 2016, Polli, 1999], the desire for music creation, performance, and consumption exists among those with disabilities. Various interfaces have been used to adapt standard software or custom-made software. These interfaces included single buttons modifying off-the-shelf devices, brain-controlled interfaces, head motion tracking, eye tracking utilizing webcams, and eye tracking using dedicated hardware.

2.1 Common Interfaces for People with Disabilities

2.1.1 Button Interface

A typical interface used for people with disabilities is the addition of a large button or buttons connected to a modified device [Saunders et al., 2011, p. 165]. Most often, the button would be connected to a switch inside a radio or tape player in order to start the playback when held, and stop upon release of the button. Saunders made clever use of a single button to control an MP3 player using the built-in protocols that allowed control of the device through the headphone jack. The interface could be operated like the radio or tape-based interfaces or be placed into "latch mode," in which playback was started on the first press and stopped on the second press. Thus, no modifications were required, and this method could be used to adapt any off-the-shelf MP3 player [Saunders et al., 2011, p.162]. However, this simple interface did not allow the user to control other settings on the

device, such as volume or transport controls. More buttons would be required to add more controls. While this was a cost-effective way of enabling a user to control a device, it did not address the issue of lower functioning motor skills or a lack of limbs. Therefore, this interface did not help those people who would require an interface adapted for those specific needs.

2.1.2 Brain Controlled Interface

While button and touchscreen interfaces have been useful for people with disabilities, they require the use of limbs and the ability to have fine motor control over those body parts. Fortunately, there are other interfaces that bypass the need for limbs entirely. Researchers have developed brain-controlled interfaces (BCI) which allow a user to control a computer without requiring any physical interaction. One such interface enabled the user to compose music in the open-source notation software, MuseScore [Pinegger et al., 2017, p. 19]. The hardware consisted of a biosignal amplifier that read EEG signals from the brain via six electrodes connected to the user's scalp [Pinegger et al., 2017, p. 3]. The interface consisted of a grid of icons or commands displayed on the screen. The user placed notes on the page through a selection of note duration, adding modifiers, such as accidentals or dots, then, finally, selecting the pitch class. The user corrected mistakes by choosing commands for navigation between notes [Pinegger et al., 2017, pp. 5-6]. This system was tested on seventeen non-disabled people as a "proof of concept" for testing on people with disabilities [Pinegger et al., 2017, p. 3]. While the BCI was proven to be accurate and successful in allowing one to copy and free-compose music [Pinegger et al., 2017, p. 14], the hardware used in the study

was somewhat time-consuming and inconvenient to use, as it required the soaking of electrodes in water, then the placement of the electrodes on the scalp [Pinegger et al., 2017, p. 3]. Furthermore, a less invasive and easier-to-setup solution has been found for interfacing with computers: eye tracking.

2.2 Use of Eye Tracking

In recent decades, the movement of the eyes has been harnessed to control computers for the performance and creation of music. Andrea Polli discovered that saccadic eye movement, or "rapid jumps of the eye used to shift gaze to a chosen object" [Polli, 1999, p. 405], was a useful means to create music in live performances. In one performance, she utilized this movement for accurate control of the timing of contrasting sounds [Polli, 1999, p. 408]. While the eye tracker was used to create several novel performances, Polli concluded that the use of eye tracking induced a similar experience in her as a performer that a traditional instrument does when one is performing or improvising music [Polli, 1999, p. 409]. Considering that people with disabilities may have once been musicians who utilized traditional instruments, eye tracking may be a good solution for those who are accustomed to such experiences.

In addition, gaze data was also used to create a virtual instrument called the EyeHarp [Vamvakousis and Ramirez, 2016]. The user gazed at a specific item on the screen; then, after a specified dwell time, the item on the screen was considered to be selected in the interface [Vamvakousis and Ramirez, 2016, p. 3]. This interface would be very familiar to those who have utilized step sequencers.

Therefore, it may appeal to musicians who previously used similar interfaces on traditional devices and software. Furthermore, it was shown that with practice, disabled musicians could become quite adept as using the EyeHarp [Vamvakousis and Ramirez, 2016, p. 7] and the portion of the interface that allowed the user to create melodies in tempo [Vamvakousis and Ramirez, 2016, p. 12]. However, Vamvakousis discovered that there was a steep learning curve with the interface and that performing with the eyes is harder than using the hands. Consequently, this barrier may dissuade those who wish to utilize eye tracking for the creation of music.

A combination of saccades and gaze data has been used for precise control when the interface contains a complex number of elements. Polli described how she "saccadically [swept]" her eye into a general area then gazed and "finely adjust[ed] movement" for precision access to an interface element on the screen [Polli, 1999, p. 408]. These two aspects of eye tracking were instrumental in allowing the selection of elements within the interface. Although this resulted in a more crowded interface being hosted, accessing creative controls of a composition via this two-step mapping movement allowed for further expression of complex ideas.

2.2.1 Use of standard web-cams for eye tracking

More recently, researchers have found another method of detecting eye movements utilizing standard web-cams, which could easily connect computers and smartphones. Software for control of instruments by a webcam was implemented

in a school in Upstate New York that enabled the real-time control of software instruments [Augereau et al., 2018]. This interface allowed the students who could not participate in a drum circle to play drums utilizing head movements. The author found that the inclusion of those children who were previously unable to participate created a much more positive learning environment for all of the students in the class. Furthermore, students that did have the motor skills needed to participate were using the interface for creative expression that they would not normally experience otherwise.

Additionally, machine learning was used to train a convolutional neural network to detect eye movements in real time, but this required 1,500 participants crowdsourced using an iPhone app [Krafka et al., 2016]. Even though crowdsourcing the training data was possible with time, and the model was good at detection, the technology is not yet commercially available [Krafka et al., 2016, p. 8]. However, the ubiquity of smart phones makes this solution an attractive choice especially since Krafka concluded that the eye tracking solution could out-perform dedicated eye tracking solutions already on the market [Krafka et al., 2016, p. 8].

2.2.2 Use of specialized eye tracking hardware

While webcams had been cost-effective, they placed a heavy burden of processing the image data onto the CPU of the computer receiving the input [Park et al., 2015, p. 905]. This image conversion process has been known to take place at a rate of 30 to 1,000 times per second [Hornof, 2014, p. 462]. Therefore, hardware containing silicon and dedicated to the processing of eye tracking image data re-

lieved the burden on the CPU and reduced potential bottle-necks. Another advantage that the dedicated eye trackers had was that they reflected near-infrared light off the eye to determine the position based on the color of the pupil. In addition, the dedicated device determined the position of the eyes in a three-dimensional space [Sundstedt, 2012, p. 15]. This allowed for the devices to work in lower light than a standard web cam and gave an additional axis (x, y and z) to map for music-making.

2.2.3 Consumer-focused Hardware and Software

Much of the dedicated hardware for eye tracking can be extremely cost-prohibitive for music-making [Hornof, 2014, p. 462]. Recently, however, cheaper alternatives designed for consumers have arrived on the market. One device, the Tobii 4C was successfully used to track which words were read from a paragraph using the Tobii application programming interface (API) to send the gaze positions to a web application for processing [Augereau et al., 2018, pp. 109-110]. Since the Tobii Eye Tracker 4C contained a dedicated application-specific integrated circuit (ASIC) for the processing of eye tracking data, it reduced the burden of processing by the CPU [Tobii Technology AB, 2019a]. Therefore, a computer with fewer resources was able to perform the eye tracking.

2.3 Types of Interactions Used in Eye Tracking

2.3.1 Grid Type

A grid style interface in which commands or options are chosen has been used in many of the types of interfaces. For example, the EyeHarp utilized a grid of dots to represent pitch classes in relation to time. A dot was selected by gazing and dwelling for 700 milliseconds [Vamvakousis and Ramirez, 2016, p. 3]. Duet for Eyes, a musical performance by people with severe disabilities, presented a grid of icons which the performer gazed upon to produce musical output [Hornof, 2014, p. 464]. While this type of interface is intuitive, the use of dwell time for selection made for accurate selection of items but may not be ideal for use as a real-time instrument. However, Vamvakousis concluded that a more efficient interface could be devised by utilizing another selection methodology such as a button or blinking [Vamvakousis and Ramirez, 2016, p. 12].

2.3.2 Virtual Mouse Type

Another interface, Ableton Live, was controlled using an eye tracker in a virtual mouse mode. Like above, a grid interface was used to select commands. However, the commands were used to position a mouse pointer and emulate a left or right click. This permitted the use of functions within the application that were not controllable via MIDI commands [Hornof, 2014, p. 465]. While this style of interface allowed the user to perform using familiar software, it was more time consuming than using a real mouse.

2.3.3 Blink Interface

Utilizing an inexpensive web-cam and common computer hardware, an eye tracking interface was designed to allow a user with several disabilities to select from menu options by blinking [Su et al., 2008]. This system conclusively eliminated the need for an external hardware selection input device, such as a button. Like the grid-style interface, the user would select menu options from a grid. Selection of the menu item would either show a sub-menu with other options or perform an action depending on where in the selection tree the user's gaze resided [Su et al., 2008, p. 353]. For accurate detection, the number of frames between each blink was used to determine if the blink was voluntary or involuntary [Su et al., 2008, p. 354]. This type of interface could resolve the problem of slow selection due to gaze dwell time or could be beneficial as a secondary input method in combination with an eye tracker.

2.3.4 Wheel Type

A different interface from the grid type was utilized in the EyeHarp, which consisted of a pie-chart-like interface. This interface worked by immediately playing back the selection that was being gazed upon. Other options could be selected immediately by looking at different interface elements that appeared as melodies were being performed [Vamvakousis, 2016, p. 3]. This was an ideal interface for building a musical instrument but made it more difficult for creating complex melodies.

2.4 Conclusion of Literature Review

A variety of interfaces have been used to allow people with limited abilities to control computers and create music. Many of these techniques can be combined to allow one to prepare a composition then perform it live. Adapting current software for use with eye tracking hardware did have some limitations, as it was more time-consuming and sometimes difficult to utilize. Creation of custom software tailored to the use of eye tracking was ideal, as shown in the case of Polli's performances and the EyeHarp. However, several of the authors demonstrated that more work could still be done to improve on those interfaces. And, since dedicated hardware has become cheaper, more experimentation is feasible to make a better interface. Therefore, building on these prior works, an eye tracking musical interface that combines eye gaze and head motion was the focus of this project. Also, instead of using a mouse or grid like interface, the design focused on six knob-like parameters to allow for improvising material and a single object in which the user gazes to change pre-composed material.

3 Method

3.1 Preliminary Work

3.1.1 Neon Puzzle Cubes

Before attempting this thesis, two projects that utilized eye tracking were created in order to become familiar with Tobii's various software development kits. The first was a composition, "Neon Puzzle Cubes", which was performed using the eye tracker, the Unity game engine, and Max for the control of the composition and production of the audio. This project involved creating a user interface in Unity that consisted of four objects laid atop a moving background that played in an infinite loop (Figure 1).

Three of the objects were cubes and one object was a sphere flattened into a disk. Each cube was assigned an instrument in Max. Similar to rolling a die, rotating a cube would change what the corresponding instrument played, based on probability. The cubes could be rotated by looking at one of four invisible disks that are directly in front of the object and set up to be gaze-aware objects (Figure 2).

When the user gazed at the top or left side of a cube, it would rotate on that axis and stop when it reached ninety degrees. Since the Unity script updates every frame refresh, the cube could potentially pass ninety degrees several times until the script saw exactly ninety degrees. This was accidentally discovered as a way to reduce the amount of control the user has over the cube and coincidentally gave

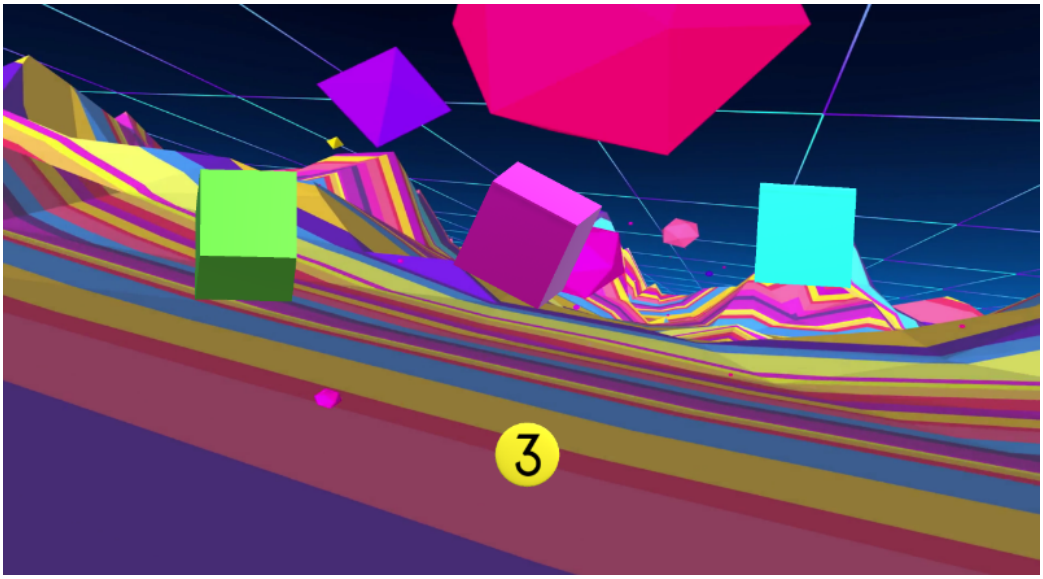

Figure 1: **The Neon Puzzle Cubes interface.** This interface that was made in Unity utilized three cubes that were rotated using the eyes. Each cube would stop rotating at a random time after it was spun, then send its positional information to Max. This information was used to choose a motive for the instrument that corresponded to the cube. The scene disc is shown below the three cubes.

it the appearance of a dice roll.

The disk on the bottom of the interface is a gaze-aware object that allows the user to change scenes during the performance. The scene number is received from Max visual programming language and is overlaid as text over the scene disk. The scene will change after the user gazes at the disk for more than three seconds. During this time, the disk will fade in color from bright yellow to a dark red to indicate to the user that a change is happening. Once the three seconds have elapsed, a scene change request will be sent to Max via the Open Sound Control (OSC) protocol.

On each frame update, Unity sends the raw X, Y, Z, and W transform rotation values of each cube to Max. Also, scene requests are sent from Unity to be handled

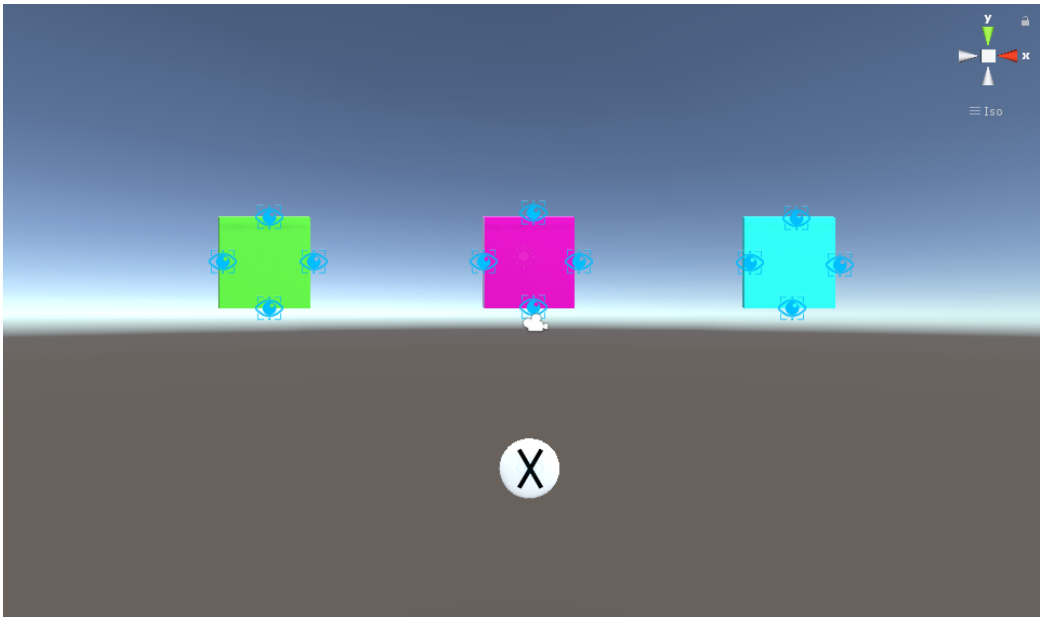

Figure 2: **The Neon Puzzle Cubes in Unity.** Shown is the interface that is in Figure 1 when working in the Unity editor. Note the gaze aware objects that are denoted by icons that look like small blue eyes.

by the composition controller in Max. Once a scene has been changed by the composition controller, Max sends the current scene number back to Unity so that it will be displayed to the user. In addition, the number is used to change other elements of the interface in Unity.

The composition starts with a black background and no cube shown on the screen; only the scene control is visible, with an 'X' which indicates nothing is being played. The performer starts Scene Zero by looking at the scene control disk. Meanwhile, nothing appears on the screen while the Leia Organa instrument plays a melody of whole notes. Eventually, a drum loop is started in the Drum Machine module with the pitch shifted to the lowest setting. The loops are randomized every four measures. This gives the piece a ghostly or lonely feel. When

Scene One is chosen, the drum loops slowly pitch shift to normal while continuing to randomize every four measures. The background video fades in and the center cube appears. The composition controller opens the first instrument in juke joint so that it may begin to accept numbers from the corresponding voter unit. The main drum loop continues to be played at normal speed until Scene Four. Scenes two and three each reveal a new cube which introduces a new instrument into the mix. During one of these scenes a final static instrument plays a repeating motif. Finally, scene four removes the drum loop, closes all of the gates for the cube instruments, removes the cubes from view. The background video fades to black while the organ comes back in to play a final note. The static motive plays two more times and then silence.

3.1.2 Turner

While thinking of ways that eye tracking could be useful for people who do not have the use of their limbs, pianists came to mind, as both their hands and feet are occupied when playing. Many times, a pianist will perform complex pieces which require the use of a score during the performance. Traditionally, pianists hire another pianist or musician to sit beside them and turn the pages of the score as both hands are busy manipulating the keyboard, and the feet are used to press the pedals. This only leaves their head and eyes free to gesture for someone to turn the page. The human page-turner reads the score along with the pianist and will turn the page precisely when the pianist has played to nearly the end of a page or nods their head for the page to be turned.

To fulfill the function of turning pages using eye gaze, my second eye tracking project, a new Windows application, was created and called ”Turner.” This application utilizes the Tobii API to track a pianist's gaze at a PDF document of the score. The musical score is displayed via Drawboard PDF [Drawboard Pty Ltd, 2019], a standard PDF reader application, run in tandem with the newly developed ”Turner” application, which subsequently displays an opaque overlay on the left and right sides of the screen. Despite being on top of the application, Turner allows the user to click and manipulate any software that is operating underneath (Figure 3). When the pianist gazes at the left overlay, the left arrow key on the computer keyboard is pressed, turning back one page in the score, while a gaze to the right overlay activates the right arrow key, turning one page forward in the score.

During the testing of this product, users discovered that a gaze time of around 250 milliseconds was the ideal amount. This allotment prevented accidental page turns, while allowing the page turn to happen quickly enough during a live performance. Conversely, a longer length of gaze time made the interface feel sluggish and non-responsive to the user. Users also determined that a wider active strip on the right-hand side would be beneficial for less precise gaze for a forward page turn. This feedback was used to make the new Unity interface adjustable so that the user could tailor the interface to suit their performance by adjusting responsiveness of the various interface elements.

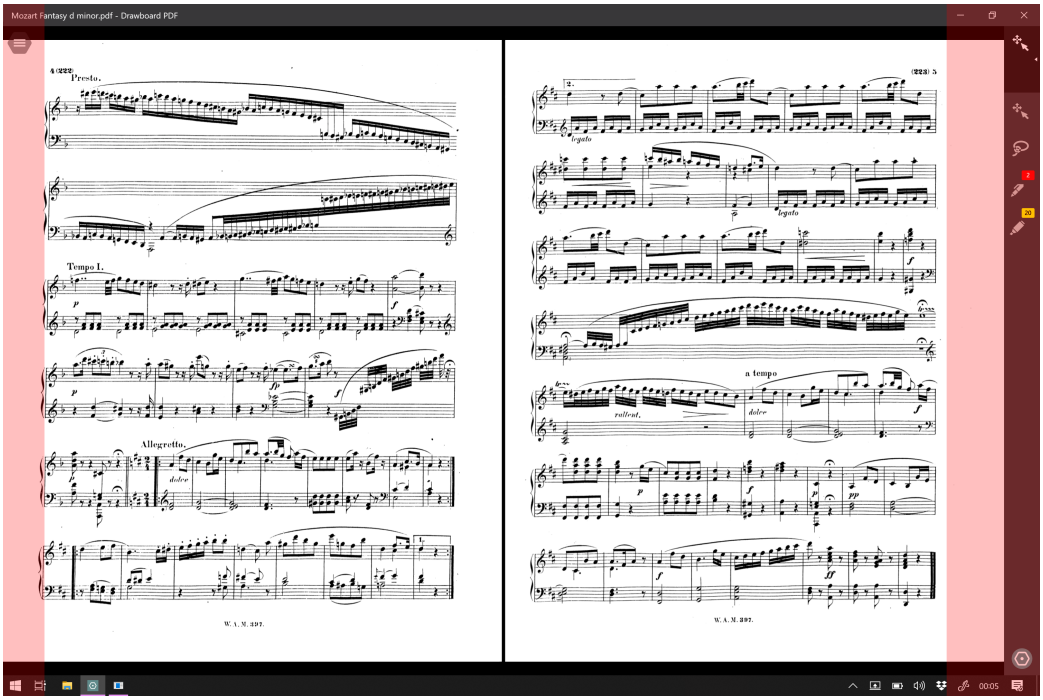

Figure 3: **The Turner application.** Screenshot of Turner application overlay that is running over a PDF reader application. When gazing at the red overlays on either side of the screen for greater than 250 milliseconds, the page will be changed in the PDF document: the right side advances a page while the left side regresses a page.

3.2 Approach

3.2.1 Product Design

After the Neon Puzzle Cube and Turner projects demonstrated that eye tracking could be used as a viable control device for musical applications, requirements were drawn up to create another interface that people with disabilities could utilize. Those requirements were: the performance of a composition without the use of limbs and an interface that was reusable for multiple compositions. The hope was that this platform could be released as an open-source tool which any person

with disabilities could use to perform their own compositions.

Consequently, this tool was made to work on a popular digital audio workstation, Ableton, so that any user could install and use the program without the need to understand anything too technical. As a result, Max for Live devices were created, as they are easily installed by even a novice computer user. A user must simply download the files from the website, then drag them into the Ableton interface to be used. This ease of installation goes further, as the Windows executable for the Unity portion does not even need to be installed like most applications. The user runs the application by clicking on the executable file for the program then Windows asks for permissions to write configuration information to the registry. In addition, both the Unity interface and the Max for Live devices should function without any special configuration to either program's settings. The only other requirement would be that the user install the drivers for the eye tracker and calibrate them according to the manufacturer's instructions.

When designing the new product, a user-centered design philosophy was applied, in which the product would be tailored around constant user feedback. User-centered design was chosen because it allows the designer to positively impact the life of an individual, while giving the user independence to perform tasks themselves [Willkomm, 2005, pg. 4]. The user's input was vital as they are the ones to ultimately use the product and would know the best way it could be used by someone of their specific ability. However, since eye-tracking is already a tool used by people with disabilities for controlling a computer [Sundstedt, 2012, pg. 2], the product should be usable by anyone who is familiar with this technology. The interface that was created is akin to installing an application that utilizes a

standard method of input such as a keyboard and mouse: while some amount of practice with the interface may be required, it uses features of the eye tracking device that are used in other software packages from both Tobii and game manufactures. Therefore, a decision was made to allot more time to development of the interface, as it would be difficult to garner feedback from a very specific and protected population. Perhaps the product can be further refined in the future using the user-centered design process as described above.

3.2.2 Composition & Performance

In place of the user-centered design philosophy, a composition was created which exploited the interface that had been developed. This allowed testing of the product in a high-pressure environment via a performance of the composition. Testing was done on the interface for responsiveness, error checking, and feasibility of using different features during rehearsals. Each rehearsal of the composition allowed for the iterative design process to take place. Furthermore, experts knowledgeable in the fields of both computer music and new interfaces for musical expression evaluated the performance of the composition and gave insight into how the interface functioned in this setting. Feedback from the expert panel was garnered in order to make future changes in how expressive the interface could be during a live setting.

Since Neon Puzzle Cubes was already used in a live performance setting, it was used as the basis for ideas in the new interface design. The new composition borrowed the concept of the exploration of improvisation utilizing pre-composed

motives and rhythms organized within a scene structure. However, instead of creating another scene manager tailored specifically for a single composition, Ableton was utilized as it has a reusable scene-based interface called the ”session view.” This view is designed specifically for live performance, as it allows for improvisation using non-linear clips which can be triggered in any order. In addition, clips may be easily arranged so that they may be triggered all at once using the scenes [Ableton Inc., 2020].

The composition, entitled sixtyHertz, is a programmatic journey about an electron being thrust into a copper atom's electron cloud. When the electron enters the cloud of the copper atom, it creates chaos, then another electron in the atom is forced out, creating electrical current. This action was demonstrated through the use of an overall structure that builds from low to high dynamically. In North America, the frequency in which the current of electricity alternates is 60 Hz. Therefore, a drone that was tuned so that a B♭ equals approximately 60 Hz was utilized as the tonal center. Borrowing from spectral musical ideas, harmonics of this drone were exploited for chordal and melodic structures throughout the piece. This was to give the impression of the many sympathetic vibrations that occur as a result of various electrical and mechanical devices that reproduce the 60 Hz drone along with other frequencies. Improvisational elements included adding these harmonic tones to the drone within the beginning of the piece and controlling the amplitude of each sinusoidal wave form to create tension. During the climax of the piece, static and rapid movement were added to portray the electron's interaction with others within the cloud. These elements were created by manipulating samples recorded from various narrow-band FM broadcasts using a software defined radio device plugged into a computer. The entire piece builds

to this climax until a monophonic synthesized pad drones multiples of the 60 Hz frequency. This was to give the impression of the electron that was kicked out from the cloud to journey alone until the process starts again in the neighboring copper atom. The use of drones and long tones was used in order prevent the need for rapid movements when using the interface.

3.3 Materials & Procedure

A combination of Max programming and the Unity gaming engine was utilized to create the interface to control the composition (Figure 4). Both pieces communicate using the Open Sound Control (OSC) protocol [Freed and Wright, 2019]. Unity was used to create the user-facing elements that would be accessible via the Tobii API library available for the Unity gaming engine. Based on user input from the Unity interface, instructions are sent via OSC over a local network interface to Max, where they are interpreted and acted upon. Ableton contains all of the pre-composed material as well as the algorithms that are used to make new material. In addition, Ableton contains the instruments and sounds that perform the music.

3.3.1 Tobii Eye Tracker

At the heart of this project is the Tobii 4C Eye Tracker [Tobii Technology AB, 2019a] that was used to control the composition. Tobii provides developers a means of reading the data from the eye tracker via an application programming interface (API) included in all of the Tobii software development kits (SDK). Developers can access the real-time data for user presence along with gaze, eye, and

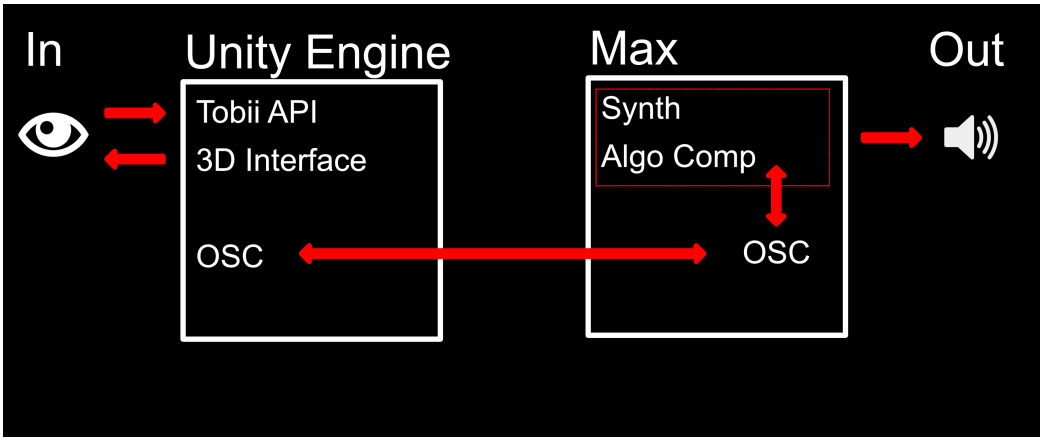

Figure 4: **Unity and Max communication.** Shown here is how the two applications will communicate commands with one another. OSC will be the protocol that is used for real-time communication via a network interface that is on the local computer. On the input side is the person looking at the eye tracker, and on the output side is a set of speakers.

head position [Tobii Technology AB, 2019b]. Some or all of these inputs can be combined to create an interactive application that only requires the movement of eyes and/or head movement.

3.3.2 Unity

Since the focus of the Tobii 4C Eye Tracker product is gaming [Tobii Technology AB, 2019a], Tobii provides a specific SDK for the Unity gaming engine. Unity is a combination of a programming framework, code editor, and three dimensional (3D) object creation tool. Every object or asset created in Unity can be manipulated through scripting in the C# programming language [Unity Technologies, 2019, Menard, 2015].

Using Tobii's Unity SDK, 3D objects can be made "gaze aware," allowing the

developer to track when the user is looking at the object and for how long [Tobii Technology AB, 2019b]. Gaze aware objects can be used for the selection of objects or to allow the user to perform other manipulations using head motion. For example, in Neon Puzzle Cubes, the scene change button was a disc-shaped object that was made gaze aware in the Unity editor. When the user looked at the scene button, a C# script would begin changing the color of the button from yellow to red. Once the button changed to red, the script would ask Max to change the scene. This change would take place as long as the user was gazing at the object. If the user were to look away before the scene changed, the button would return to the color yellow and the counter would reset until the user gazed at the object again. Therefore, Unity provided an advantage over the other SDKs because it allowed the developer to quickly create objects with which the user could directly interact. Otherwise, precious development time would be wasted in linking the object's position on the screen with the point at which the user is gazing.

3.3.3 Max for Live

Information about the manipulation of the Unity objects was communicated via the OSC protocol to an application written in the Max visual programming language. This language was chosen as it allows for rapid development of interactive applications that produce and manipulate audio and has many ways of facilitating communication between other software [Cycling '74, 2019]. The application programmed in Max decoded and interpreted the user input from Unity to control Ableton via Max for Live. Since Ableton contains a mode that is tailored for live performances, precomposed material could be pre-loaded, then controlled by the

user efficiently in the live performance.

4 Results

4.1 Interface Design

Much of the interface design was borrowed from the Neon Puzzle Cubes project. Like before, there is a scene disk that controls scene changes using eye gaze, and the user rotates three dimensional objects to send parameters via OSC. However, the number of objects were doubled to six and now they are controlled using a combination of eye gaze and head motion. When the user gazes at one of the objects for a period of time, it becomes unlocked to be controlled by head motion. Since the movement of the object mirrors the movement of the user's head, a three dimensional model of a head was downloaded from a popular 3D model sharing site and used for the six objects [Aakash, 2018].

The interface for the Neon Puzzle Cubes project was designed exclusively for use in that specific project. However, since the goal of this project was to create an interface that would eventually allow other people to perform their own compositions, it was decided to make the new interface reusable across multiple projects. The resulting implementation was created in Max for Live so that the user could control aspects of a performance using the session view in Ableton. The user places the Max for Live devices in an Ableton session in the same way that they would use other built-in devices and plugins. Also, since many of the included devices in Ableton use Max for Live for interface elements (such as buttons, toggles, text entry, etc.) and as the underlying engine, the device looks and runs the same as other parts of Ableton.

4.1.1 Main Interface

When the user opens the compiled Unity application, they are first greeted with a setup window that asks the user to choose display resolution, graphics quality, and monitor settings (Figure 5). Another tab contains settings to map keyboard and mouse buttons to specific inputs intended for playing games. The only setting that affects the application are the settings in the graphics tab; the rest may be ignored.

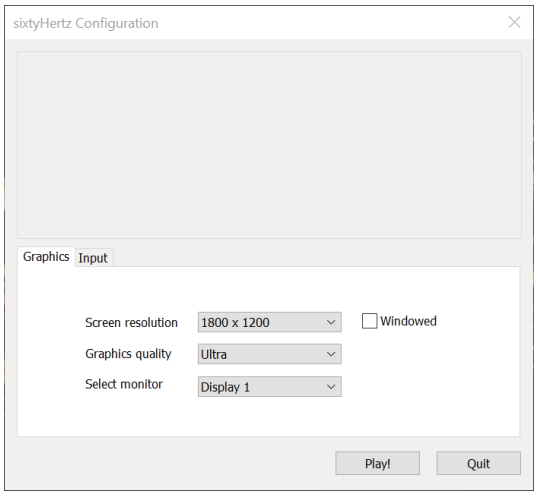

Figure 5: **The Unity interface setup window.** When the Unity interface application is launched, a settings window appears asking the user to setup the graphical settings for the three dimensional interface. Note that the input tab appears but is not used for this application.

Upon pressing the button marked ”Play!”, the user is greeted by the main interface: six heads spaced evenly in two rows and three columns (Figure 6). Between the two rows of the heads, a black scene disk is in the center of the screen displaying a red letter ’X’. This indicates that the Ableton Max for Live device is not sending information yet. Otherwise, it will display the currently playing scene.

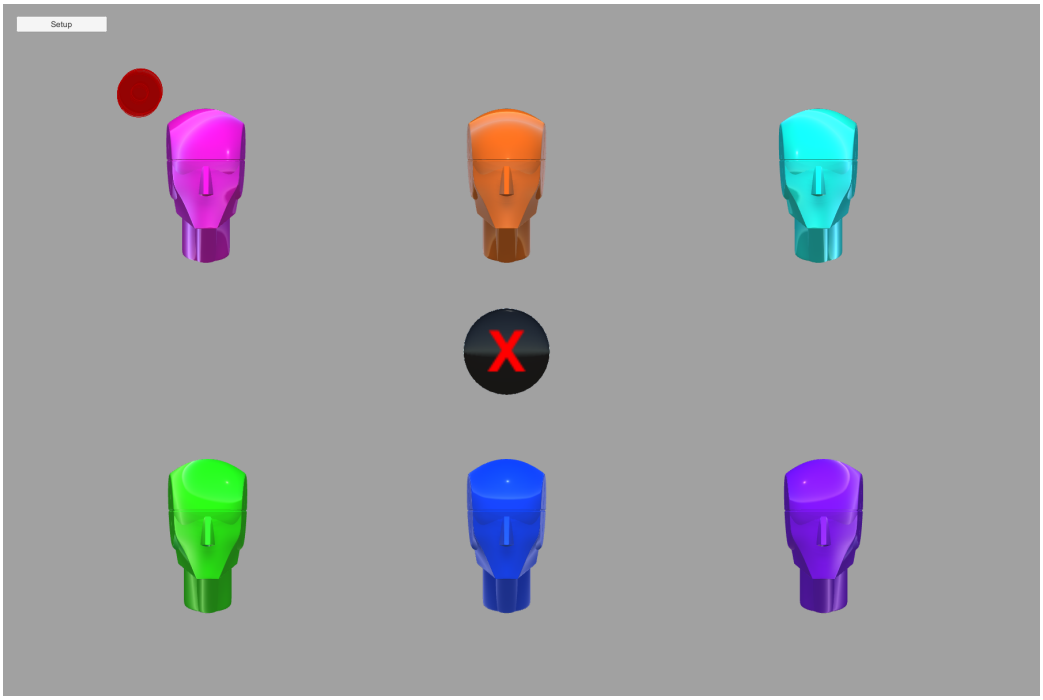

Figure 6: **The Unity interface.** After setting the graphical settings, the main interface screen appears. This interface contains the objects that the user interacts with using a combination of eye gaze and head rotation. Once clicked, the button in the top-left corner activates a screen that contains settings for the operation of the application.

If the Tobii eye tracker is connected and functioning correctly, each head will appear to be a different color. However, if the eye tracker is disconnected or not working, all of the heads will be the color red (Figure 7). If this occurs, the user may simply plug in the device and the heads will become multi-colored indicating the eye tracker is working properly.

Also, a large red dot that acts as the cursor appears wherever the user is gazing. This gives the user feedback so that they know that the eye tracker is functioning and indicates if it is calibrated correctly. When the user gazes at any of the interactive objects on the screen, they slowly change to the color white. When a head

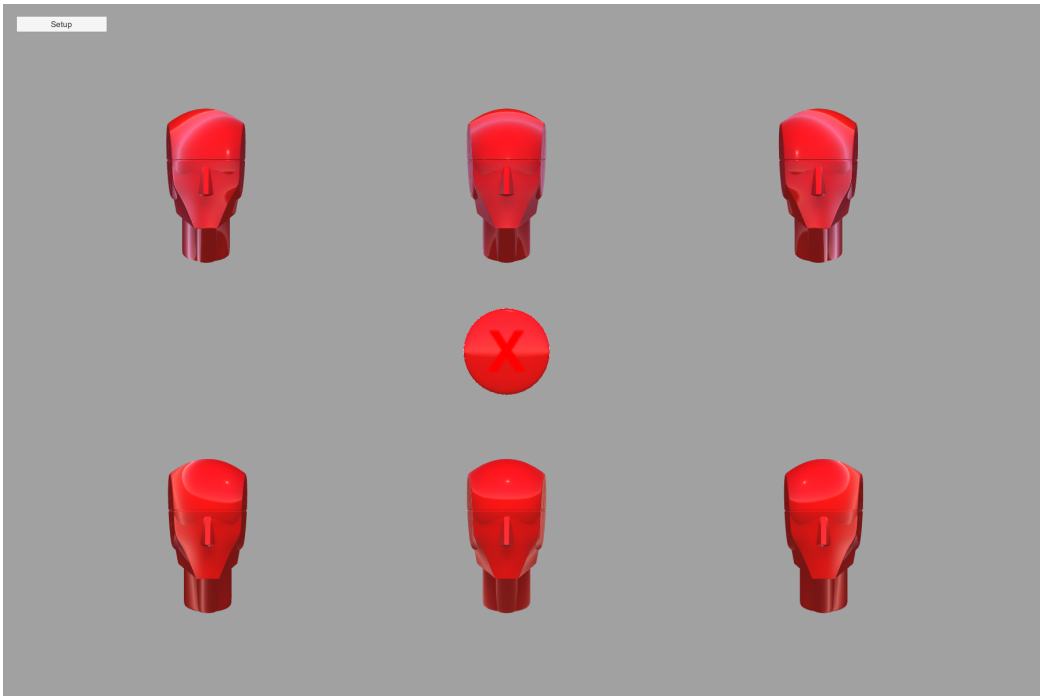

Figure 7: **The Unity interface without the eye tracker connected.** If the Tobii eye tracker is not functioning or is disconnected, all interface objects change to the color red. Once the eye tracker is reconnected, the interface elements return to the color scheme shown in figure 6.

becomes completely white (Figure 8), it communicates to the user that it is ready for manipulation using head motion. For the scene disk, changing to the color white tells the user that it has sent the scene change signal to Ableton.

If the user gazes at a head object, they ”latch” onto the object and are able to move it using their head movement. During testing of the interface, it was discovered that it was difficult to adjust the pitch of the head. Therefore, the movement is limited to a single axis in which the yaw of the head may be controlled. Reducing the motion of the head also reduced the number of parameters that can be controlled to one, but allowed for more accurate control. In addition, since the head object moves the same direction that the user's head moves, the manipulation

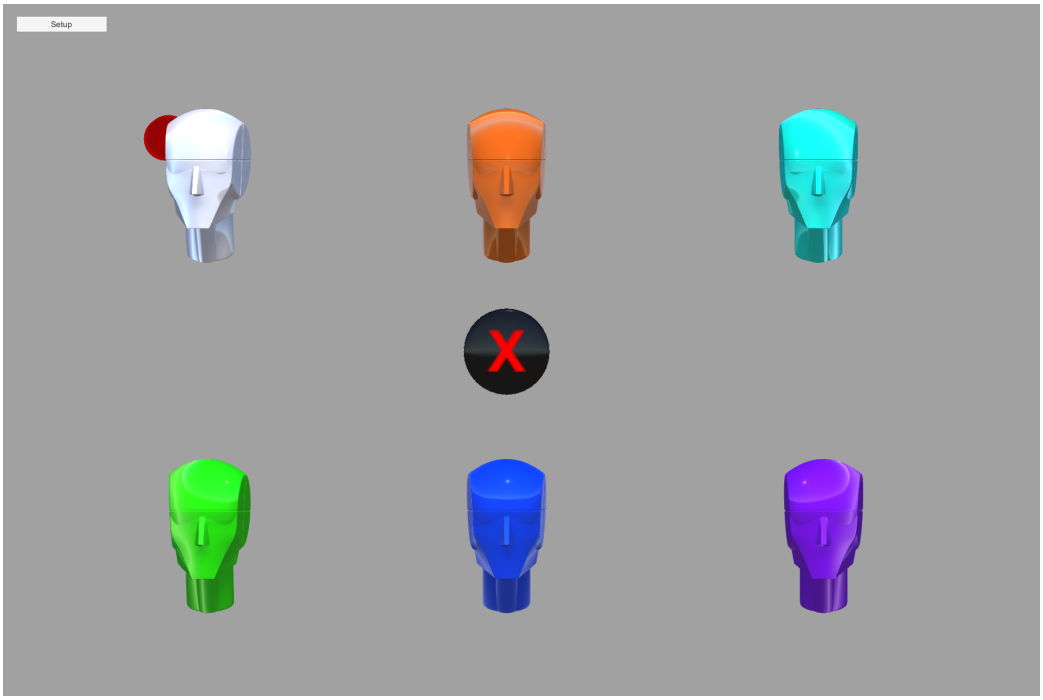

Figure 8: **Head gaze selection in Unity interface.** If the user gazes upon an interface object, the object changes to the color white using linear interpolation based on the number of seconds chosen in the setup screen shown in figure 9.

feels as natural as looking in a mirror.

4.1.2 Settings

A button labeled ”setup” is located on the top-left side of the screen. This button, when clicked with the mouse, changes to the setup screen (Figure 9). The user may also press the escape key on the computer keyboard to enter this screen as well. Displayed on the screen are several configuration options that pertain to operation of the application.

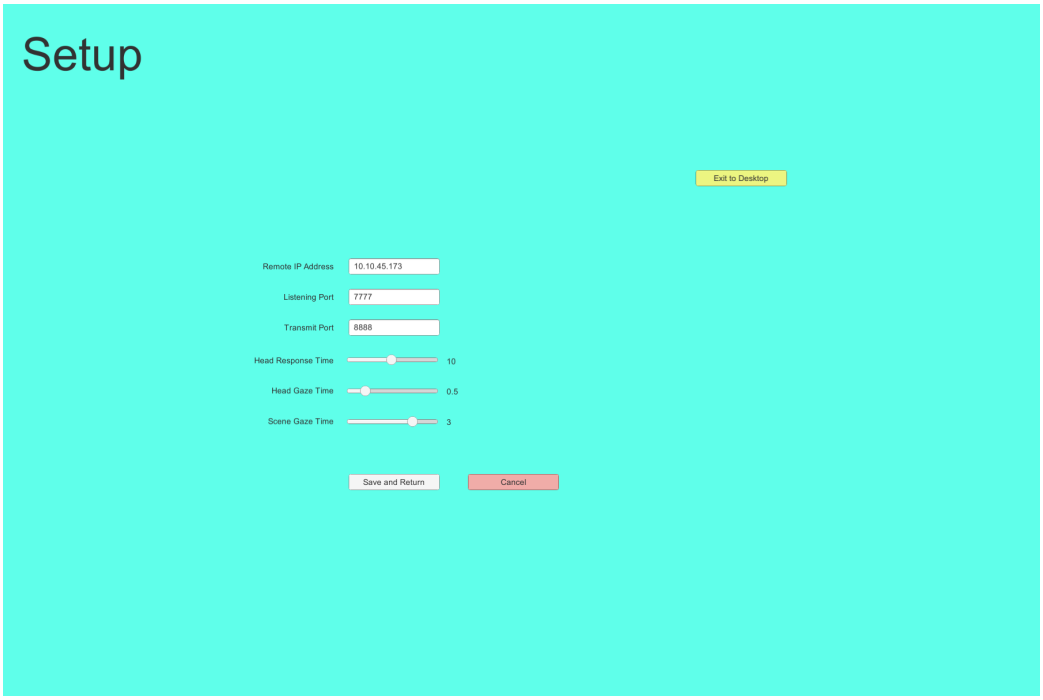

Figure 9: **The settings screen.** The setup screen allows the user to adjust parameters for OSC communication, head object responsiveness, and how long it takes for the objects to be triggered or manipulated.

The remote IP address setting corresponds to the network IP address to the computer that is running Ableton. If using the same computer for both Ableton and the Unity application, then it should be set to the local loop IP address of ”127.0.0.1”. This allows for communication across both applications without entering the network outside of the computer.

Two settings exist to setup which User Datagram Protocol (UDP) port that the OSC data will be communicated through. The UDP is a low-overhead protocol that requires a port to communicate for each source and destination process [Davidson, 1988, p. 61]. Therefore, both a listening port and a transmit port setting exist to accommodate this requirement. The listening port is the port that

Ableton uses to communicate to the Unity program and the transmit port is used to send information back to Ableton.

When the user moves a head object using their own head motion, a multiplier is used to control how responsive the head is to their movements. This prevents the head from feeling like it is moving out of control or moving too sluggishly. A slider labeled ”head response time” allows the user to change this setting.

Gaze dwell time can be adjusted using the head and scene gaze time sliders. These adjust the amount of time in seconds it takes for the objects to trigger or become available for manipulation. Again, as mentioned above, the color of the object changes from the resting color to white. This setting affects the amount of linear interpolation time is used for this color transition to occur. Therefore, the user is presented with visual feedback that indicates something is happening.

Three buttons allow for navigation away from the settings screen. The ”save and return” button will save the settings using the Unity PlayerPrefs function which saves data into the Windows registry. This allows settings to persist after the application has been closed. A cancel button discards any setting changes and returns to the main interface screen. The ”exit to desktop” button quits the application.

4.1.3 OSC Communication

Using OSC, the Unity interface controls two Max for Live devices within Ableton: dunlap.ableton.maxHeadroom (maxHeadroom) and dunlap.ableton.maxHeadCtl

(maxHeadCtl). Since the two components communicate with each other over a network, they can either run on the same computer using a local loop network interface or via a wifi or ethernet network on two separate computers.

Each head object in the main interface acts as a knob control that sends a ranged value directly to the maxHeadCtl using the OSC protocol. This value is scaled to be between 0 and 127 so that it is similar in range to MIDI parameters such as velocity. This way, if the interface is repurposed in Max, it could be used to interact with other Max objects that operate within this range of values.

The scene disk communicates bidirectionally to maxHeadroom via OSC. When the disk reaches the gaze time as indicated in the settings screen, it sends a Max bang through OSC, which tells maxHeadroom that the scene disk has been triggered. As the scene changes in Ableton, the scene number is sent so that it may be shown in the scene disk.

4.2 Max for Live Devices

For those unfamiliar with Ableton, the session view consists of a grid of MIDI or audio clips that may be looped or played a single time. Each column on the grid represents a MIDI controlled instrument that the clip controls or an audio track that the clip plays through. The rows allow the performer to fire all clips contained within the row at once (Figure 10). Each row is called a scene and can be triggered using a play button on the screen that corresponds to each row. In addition, Max for Live has access to Ableton controls and functions through an API called the Live Object Model (LOM) (Figure 11).

Figure 10: **Ableton.** The Ableton digital audio workstation application session view allows users to trigger individual clips to be run in loops or once. A row of scenes can be triggered at once by using the play buttons located on right side of each row.

The LOM was manipulated in Max for Live using three objects: live.path, live.object, and live.observer. The live.path object grabs the current path so that live.object can perform actions or get the state of things using messages. For example, a message can be sent through live.object that asks if the piece is currently being played back within the Ableton session. If nothing is being played back, a message may be sent through live.object to begin the playback.

While live.object can retrieve the status of a particular function within Ableton, it can only do so on command; Ableton will only send a response once it has received a message from Max for Live. However, live.observer can request the status of LOM function once and Ableton will continuously push updates to the

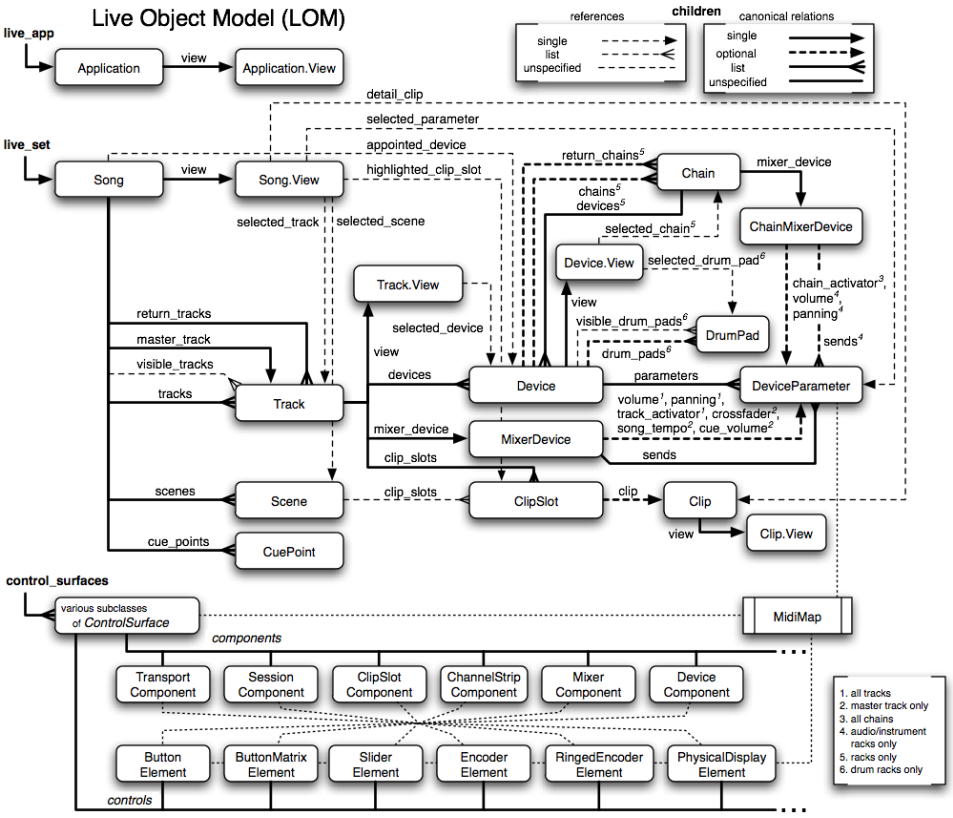

Figure 11: **The Live Object Model** [Cycling '74, 2019] allows for getting and setting parameters within the Ableton application in Max for Live. Also, functions such as the play buttons mentioned in figure 10 can be controlled through this API.

live.observer object.

4.2.1 dunlap.ableton.maxHeadroom

The dunlap.ableton.maxHeadroom (maxHeadroom) Max for Live device uses the LOM to manage and control scene changes. In addition, it is used to setup OSC communication parameters: the IP address/port of the Unity interface running

on the remote device and the port for transmitting OSC data (Figure 12). Since the device performs functions globally–meaning it drives the whole session rather than one track, maxHeadroom is only required to be on a single track for the entire session.

Figure 12: **The maxHeadroom device** is required on at least one track in the Ableton session as it contains the setup parameters for the OSC communication. No more than one maxHeadroom device should be placed in the session.

As mentioned above, the maxHeadroom device carries out the task of scene selection in Ableton and reports the scene number to the Unity interface. These functions are contained within a Max abstraction named dunlap.ableton.sceneManager (sceneManager) (Figure 13). The sceneManager abstraction queries the scene identification numbers in Ableton using live.path and live.object. These scene IDs are indexed and stored in a list for later reference. If the user changes the layout or number of the scenes within Ableton, they must refresh this list by using the refresh scenes button located within the maxHeadroom device.

When the user requests a scene within the Unity interface, a counter is incremented within maxHeadroom (Figure 14). The current counter number is then received by the sceneManager abstraction, which is used to reference the scene ID number via the scene ID index. The corresponding scene ID number is then sent

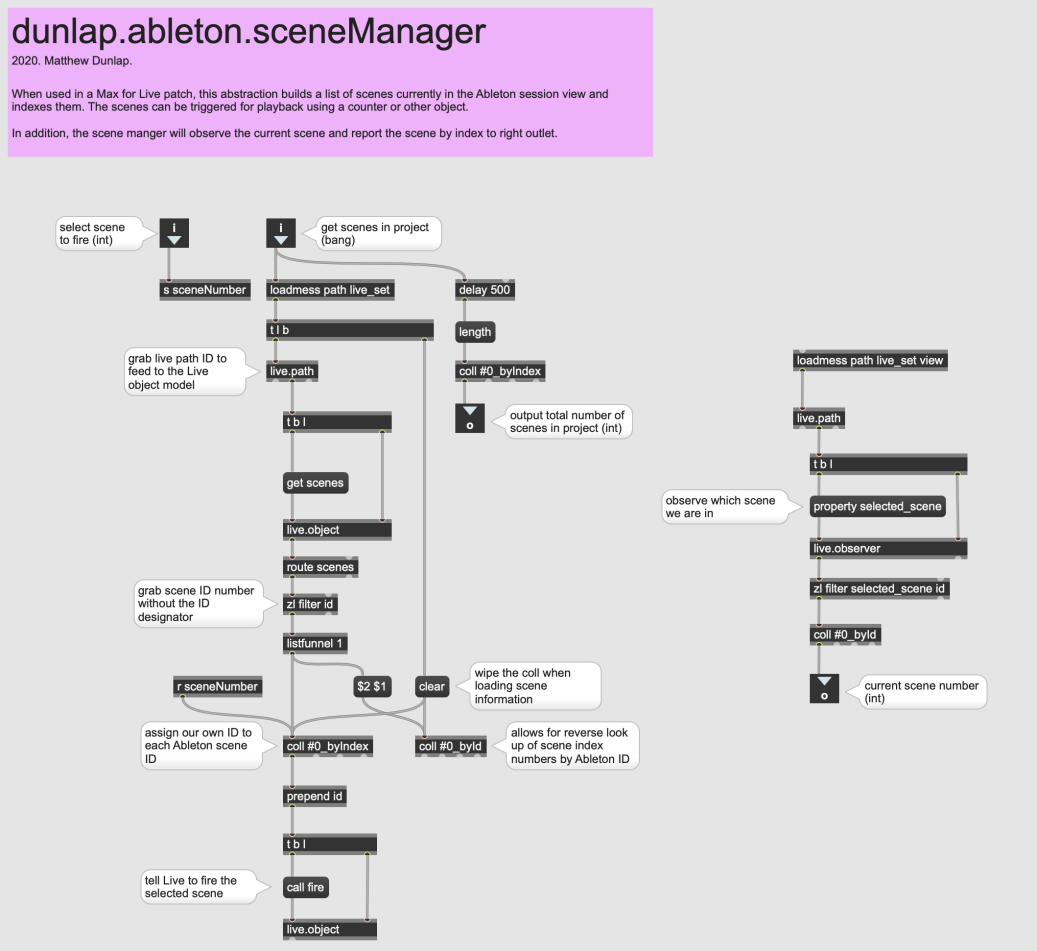

Figure 13: **The sceneManager Max abstraction** receives scene information from the Live Object Model (Figure 11). It may also trigger scenes in Ableton by using the ”fire” command.

to the LOM via live.object along with a ”call fire” command. This command tells Ableton to trigger the scene with that particular scene ID number. If no scenes are being played, Ableton will immediately begin playback for the session starting with the selected scene ID. If a scene is currently being played, the next scene will begin upon the start of beat one of the next measure. This allows the user to begin playback by simply requesting a scene or to play the next scene seamlessly

without an abrupt transition to the next part of the piece.

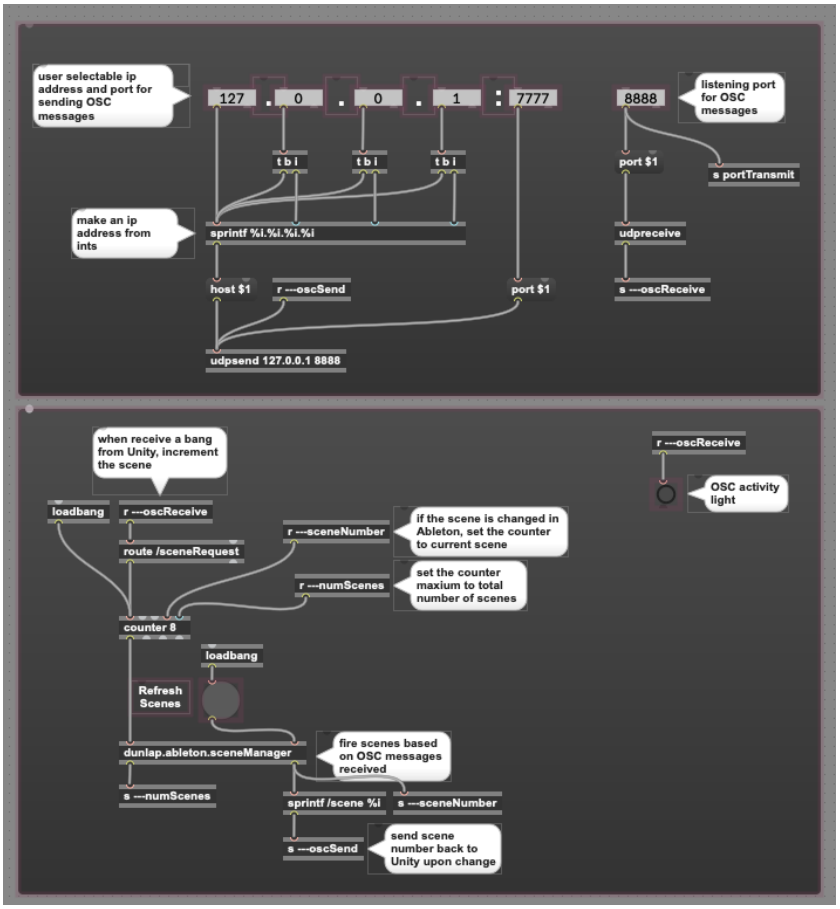

Figure 14: **The maxHeadroom patch.** Shown is the Max for Live patch for the maxHeadroom device (Figure 15). The OSC setup parameters are parsed so the udpsend and udpreceive Max objects can properly communicate. In addition, the sceneManager abstraction (Figure 13) is used to control which scene is selected. The scene disk from the Unity interface sends a bang via OSC once it has been triggered. This bang increments a counter object which has its maximum set to the number of scenes from the sceneManager abstraction. The counter sends the new scene number to be triggered to the sceneManager which is then ”fired” via the Live Object Model (Figure 11).

Once the scene changes, the currently playing scene ID is then pushed to the sceneManager via a live.observer object. A reverse-lookup is performed by cross-referencing a list of scene indices indexed by the Ableton scene identification

numbers. The corresponding scene index is then sent back to the Unity interface via OSC so that the currently playing scene is shown in the scene disk. This method of having the actual scene index number that is being played by Ableton being sent to the Unity interface, rather than the counter number, was chosen so that the scene can be changed via onscreen interface and be represented accurately within the Unity interface.

One last function of the sceneManager abstraction is to report a count of the total number of scenes within the Ableton session. This number is then used to set the maximum parameter for the scene counter object so that the counter remains within the bounds of the amount of scenes contained within the Ableton session. If the user requests a scene while the largest scene number is being played by Ableton, the counter goes back to zero and Ableton starts the beginning of the piece.

4.2.2 dunlap.ableton.maxHeadCtl

When considering how to control device parameters within Ableton, there was a desire to manipulate different device parameters for each scene using the same head in the Unity interface. Also, a user-friendly way of mapping a device control to each head was sought so that any user familiar with Ableton could map the heads to any control in a standard way. Therefore, a separate live device, dunlap.ableton.maxHeadCtl (maxHeadCtl) was created to be used in an extremely flexible way. Either, one could have a single maxHeadCtl device for each scene and each device control, or the user could have the same maxHeadCtl for all

scenes but for one device control.

Mapping a control to a head is done by pressing a map button on the maxHeadCtl device then choosing a control located on the same track. Then the user chooses a head number and for which scene they wish the mapping to be active. If the mapping should be valid for the entire piece, then the user will select the all scenes option (Figure 15). In addition, the user could map one head to multiple parameters at once. This is done by using the MultiMap Max for Live device along side the maxHeadCtl device. Conversely, multiple maxHeadCtl devices can be added to use multiple heads to control a single parameter.

Figure 15: **The maxHeadCtl Max for Live device** maps a specific head to a control in another Ableton device. The control may be filtered by specific scene or allowed to control the mapping for all scenes. Shown here, the maxHeadCtl is mapped to the frequency in the Distort Ableton device. Head number one will be allowed to control the parameter for all scenes in the Ableton session. In addition, the head is limited to a minimum of 20% and maximum of 83% of the frequency. This means that the head cannot control the frequency below 166 Hz or above 7.22 KHz.

The maxHeadCtl patcher (Figure 16), reuses the sceneManager abstraction to grab the number of scenes and the currently playing scene number from the Ableton session. It uses this information to populate a list of scene index numbers. If the user specifies a scene number, the index chosen is compared against the

currently playing scene. If there is a match, a gate will open allowing information from the Unity controller to change the mapped parameter. If the user chooses the ”All” option from the list, another gate allows for the information to flow from the Unity interface at all times. Head numbers are placed in another list that allows the patch to filter the information coming from the Unity interface so that it matches the head that was selected.

Code was borrowed from the MultiMap Max for Live device which allows the user to map the head/scene chosen to any parameter within the same track. This was simply copied and pasted with minimal changes being made. Therefore, the interface should be familiar to anyone who has used the mapping functionality within Ableton. The user presses the map button, then selects which parameter will be mapped. The minimum and maximum controls allow the user to select the range of the mapped parameter that will be affected by head movement. In addition, a setting changes the amount of curve of the parameter motion from linear to logarithmic. Lastly, a toggle button permits the user to temporarily remove the mapping.

4.3 Performance

In order to evaluate whether the interface could be used to express music creatively, a composition was written for the interface so that it might be showcased within a performance. However, due to the COVID-19 global pandemic, the performance could not be done without breaking social distancing guidelines set first by state and local governments. Therefore, to showcase the interface safely, the

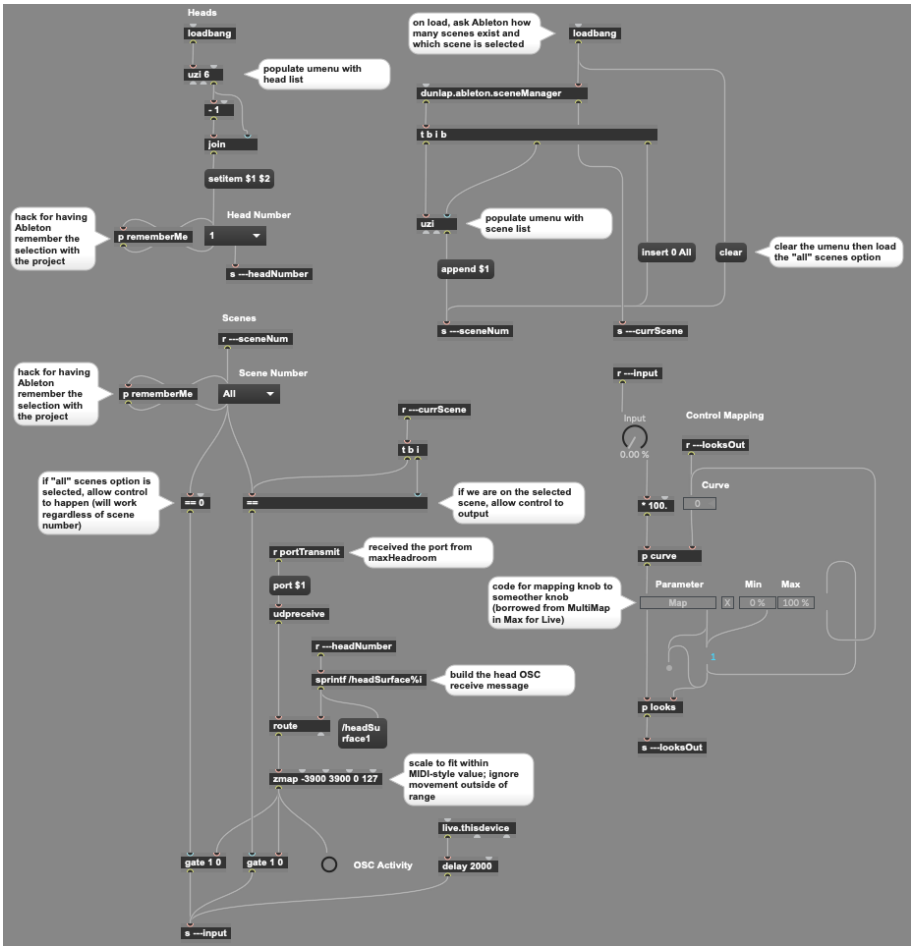

Figure 16: **The maxHeadCtl patch.** Shown is the Max for Live patch for the maxHeadCtl device in figure 15. The device uses the sceneManager abstraction (Figure 13) to fill the scene number list and to filter out any inputs from the heads that are not in the selected scene. In addition, the patch receives the head movement data over OSC and transform the number to a range of 0-127. This is then sent to the mapped input knob. Note that only the head number chosen in the head number list is received via OSC. Code borrowed from the MultiMap Max for Live device allows for the mapping of parameters to the chosen head number. Multiple copies of the maxHeadCtl device can be placed for different heads, tracks, or scenes.

performance was pre-recorded and distributed to a panel of experts for critique. The Unity interface and Ableton DAW ran on separate computers: a Microsoft Surface Go tablet and a MacBook Pro, respectively. Both were captured using

screen recording software built into their respective operating systems. In addition, a camera was setup so that the main focus was on the performer's face. This allowed the audience to see how the eye and head movement correlated to the change in both the music and the movement within the Unity interface. The audio from the performance was captured by recording to the arrangement view while performing in session view using the global record function. Once everything was recorded, the performance was edited into two separate videos using Final Cut Pro. The first video is the complete performance using a split-screen setup so that the user can see all three angles of video at once. Another video is used to explain how the interface works using text overlaid onto edited segments of the same performance.

4.4 Expert Analysis

After the performance, feedback from was gathered from three separate people who are considered to be experts in both computer music composition and designing interfaces for musical expression. Many of the interface elements communicated to the experts that they affected parts of the composition. For example, one expert noted that they correlated a change within the piece to the changing of scenes on the scene disk. While they didn't know that the changes were specifically scenes, they did associate the scenes as a change in some state. In addition, the same expert noted that they could see that the head was being selected by the cursor for movement. Conversely, another expert said they forgot that the piece was being controlled by any unusual control as they had gotten drawn into the piece.

Two experts wondered if the background music in the explanation video was performed using the same interface. While the music was not performed using the device, it could easily be transformed into a piece that can be performed with same system. The background music, entitled ”Guru Meditation Error”, was composed in Ableton using LFOs to drive random parameter or arpeggiation generators; one could replace the LFOs and generators to control the parameters and arpeggiations in a similar fashion as sixtyHertz. This thought experiment demonstrates the flexibility of the interface's design for performing other compositions by the same composer.

5 Conclusions

When starting this project, the goal was to make a unique interface for the performance of a composition by people who lack the use of their limbs. Inspiration was drawn from the works of other people who had made various interfaces that utilized eye-tracking or head motion. However, during the design phase of the interface, a combination of both elements was discovered to be an excellent way to control multiple elements of a composition. Furthermore, more exploration resulted in an interface that could easily be reused across multiple compositions. Therefore, instead of a simple interface, a small platform was designed that combined an off-the-shelf eye tracker, a typical digital audio workstation, and custom software. With a good set of instructions, this system could be installed by differently-abled people to control their compositions during a live performance. In addition, an eye tracker and software combination developed by Tobii called PC-eye Plus Access exists to allow a user to control a Windows computer using eye gaze for mouse and keyboard commands [Tobii Dynavox AB, 2020]. While this product is more expensive than the consumer version, it opens up more possibilities for the target user to control other aspects of their computer. In fact, using this setup, a person could conceivably compose a piece in Ableton by assigning the parameters to the heads in the interface, then perform it using the custom interface.

Now that an interface has been created and tested, the next step is to continue to make improvements by distributing the software to those who require a way to perform using eye and head tracking. This will be done using a combination of hosting the software on Github and communicating the existence of the software

through organizations that work with a population that has disabilities. Once the software has been hosted, a manual for the installation and operation of the software should be published for those users who are not familiar with using Max for Live devices within Ableton. Since the software operates via a network connection, other uses can be devised for live performances that require a controller that is remotely placed from the computer that is creating the sounds. Additionally, the interface could be useful for other instances in which the performer has their hands occupied such as a pianist that requires a change of background elements or audio effects within their performance.

References

[Aakash, 2018] Aakash, V. (2018). Andrew loomis head: 3d model. https://sketchfab.com/3d-models/andrew-loomis-head-d2020b07408b4f238004f04d082a3193. Accessed February 2020.

[Ableton Inc., 2020] Ableton Inc. (2020). Ableton live 10 manual: Session view. https://www.ableton.com/en/manual/session-view/. Accessed April 2020.

[Augereau et al., 2018] Augereau, O., Jacquet, C., Kise, K., and Journet, N. (2018). Vocabulometer: A web platform for document and reader mutual analysis. pages 109–114.

[Cycling '74, 2019] Cycling '74 (2019). Cycling '74: Tools for sound, graphics, and interactivity. https://cycling74.com/products/max. Accessed April 21 2019.

[Davidson, 1988] Davidson, J. (1988). *Introduction to TCP*. Springer New York, New York.

[Drawboard Pty Ltd, 2019] Drawboard Pty Ltd (2019). Drawboard pdf: Easy pdf markup for windows 10. https://www.drawboard.com/pdf/. Accessed December 17 2019.

[Freed and Wright, 2019] Freed, A. and Wright, M. (2019). Introduction to osc. http://opensoundcontrol.org/introduction-osc. Accessed April 21 2019.

[Hornof, 2014] Hornof, A. J. (2014). The prospects for eye-controlled musical performance. pages 461–466.

[Krafka et al., 2016] Krafka, K., Khosla, A., Kellnhofer, P., Kannan, H., Bhandarkar, S., Matusik, W., and Torralba, A. (2016). Eye tracking for everyone. pages 2176–2184.

[Menard, 2015] Menard, M. (2015). *Game development with Unity*.

[Park et al., 2015] Park, J., Jung, T., and Yim, K. (2015). Implementation of an eye gaze tracking system for the disabled people. *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 904–908.

[Pinegger et al., 2017] Pinegger, A., Hiebel, H., Wriessnegger, S. C., and Muller-Putz, G. R. (2017). Composing only by thought: Novel application of the p300 brain-computer interface. *PLoS One*, 12(9):e0181584.

[Polli, 1999] Polli, A. (1999). Active vision: Controlling sound with eye movements. *Leonardo*, 32(5):405–411.

[Saunders et al., 2011] Saunders, M. D., Questad, K. A., Cullinan, T. B., and Saunders, R. R. (2011). Adapted digital music players for individuals with severe impairments. *Behav Interv*, 26(2).

[Su et al., 2008] Su, M., Yeh, C., Lin, S., Wang, P., and Hou, S. (2008). An implementation of an eye-blink-based communication aid for people with severe disabilities. pages 351–356.

[Sundstedt, 2012] Sundstedt, V. (2012). Gazing at games: An introduction to eye tracking control. *Synthesis Lectures on Computer Graphics and Animation*, 5(1):1–113.

[Tobii Dynavox AB, 2020] Tobii Dynavox AB (2020). Pceye plus product page. `https://www.tobiidynavox.com/en-US/devices/eye-gaze-devices/pceye-plus-access-windows-control/`. Accessed May 2020.

[Tobii Technology AB, 2019a] Tobii Technology AB (2019a). Tobii eye tracker 4c. `https://gaming.tobii.com/product/tobii-eye-tracker-4c/`. Accessed April 21 2019.

[Tobii Technology AB, 2019b] Tobii Technology AB (2019b). Welcome to the tobii developer zone. `https://developer.tobii.com/`. Accessed April 21 2019.

[Unity Technologies, 2019] Unity Technologies (2019). Unity 2019: Performance by default, high-fidelity real-time graphics, and artist tools. `https://unity3d.com/unity`. Accessed December 16 2019.

[Vamvakousis, 2016] Vamvakousis, Z. (2016). *Digital musical instruments for people with physical disabilities*. Thesis.

[Vamvakousis and Ramirez, 2016] Vamvakousis, Z. and Ramirez, R. (2016). The eyeharp: A gaze-controlled digital musical instrument. *Front Psychol*, 7:906.

[Willkomm, 2005] Willkomm, T. (2005). *Assistive Technology Solutions in Minutes*. ATECH Services, Concord, New Hampshire.

A Appendix

A.1 Links to Source Code and Videos

Neon Puzzle Cubes full writeup and source code:

`https://github.com/boomninjavanish/neonPuzzleCubes`

sixtyHertz (the interface) source code:

`https://github.com/boomninjavanish/sixtyHertz`

sixtyHertz performance video:

`https://vimeo.com/413004326`

sixtyHertz development video:

`https://vimeo.com/413027865`

A.2 Unity Interface C# Scripts

A.2.1 HeadManipulation.cs

```
1  using UnityEngine;
2  using Tobii.Gaming;
3
4  public class HeadManipulation : MonoBehaviour
5  {
6      // used to check if objects is being gazed upon
7      private GazeAware gazeAwareComponent;
8
9      // numbers to track how long user is gazing
10     // object will become fully selected after timer has
           elapsed
11     private float gazeTime;
12     private float gazeTimeElasped = 0.0f;
13     private float gazeTimeStarted = 0.0f;
14     private bool gazeStarted = false;
15
16     // used to lerp between materials
17     new Renderer renderer;
18
19     // map to the the head that this script attaches to
20     public GameObject head;
21
22     // change to how responsive the movement to be
23     public float responsiveness;
24
25     // materials to use for object when selected and
           deselected
26     public Material selectedMaterial, deselectedMaterial,
           noTobiiMaterial;
27
28     // Start is called before the first frame update
29     void Start()
30     {
31         // setup the rederer for lerping between colors
32         renderer = GetComponent<Renderer>();
33
34         // get gaze aware objects
35         gazeAwareComponent = GetComponent<GazeAware>();
36
37         // set the gaze time and responsiveness to the vars in
```

```
            playerprefs
        gazeTime = PlayerPrefs.GetFloat("headGazeTime");
        responsiveness = PlayerPrefs.GetFloat("
            headResponseTime");

    }

    // Update is called once per frame
    void Update()
    {
        // See if we are gazing on this object
        if (gazeAwareComponent.HasGazeFocus)
        {
            // set the time started to the current time when
                the gaze has begun
            if (gazeStarted == false)
            {
                gazeTimeStarted = Time.time;
                gazeStarted = true;
            }

            // start changing the color of the head
            if (gazeTimeElasped < gazeTime)
            {
                // set the elapsed time
                gazeTimeElasped = Time.time - gazeTimeStarted;

                // lerp the material color to show that it is
                    about to be selected
                float lerpTime = Mathf.PingPong(
                    gazeTimeElasped, gazeTime) / gazeTime;
                renderer.material.Lerp(deselectedMaterial,
                    selectedMaterial, lerpTime);
            }

            if (gazeTimeElasped > gazeTime)
            {
                // take the head pose information from the
                    user
                var headPose = TobiiAPI.GetHeadPose();

                // translate the head pose to a position for
                    rotation of head
                if (headPose.IsRecent())
                {
```

```
75                      // filter out x and z axis from obeject
                            movement
76                      Quaternion headPoseNoXZaxis = Quaternion.
                            Euler(0, headPose.Rotation.eulerAngles
                            .y, 0);
77
78                      // invert axes from head position
79                      headPoseNoXZaxis = Quaternion.Inverse(
                            headPoseNoXZaxis);
80
81                      // move the head based on y axis movement
82                      head.transform.rotation = Quaternion.Lerp(
                            head.transform.rotation,
                            headPoseNoXZaxis, Time.
                            unscaledDeltaTime * responsiveness);
83                  }
84              }
85
86          }
87          else
88          {
89              // change back to deselected material depending on
                    if eye tracker is connected
90              if (TobiiAPI.IsConnected)
91              {
92                 renderer.material = deselectedMaterial;
93              }
94              else
95              {
96                  renderer.material = noTobiiMaterial;
97              }
98
99              // reset the gazeStarted flag & gazeTimeElapsed
100             gazeStarted = false;
101             gazeTimeElasped = 0.0f;
102         }
103
104
105     }
106 }
```

A.2.2 HeadOSC.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HeadOSC : MonoBehaviour
{
    // setup open sound control
    public OSC osc;

    void Update()
    {
        // start OSC message object
        OscMessage message = new OscMessage
        {
            // set the osc addresses to the head's name
            address = "/" + this.name
        };

        // send the y rotation value via OSC
        //convert to int because Max had difficulty with precise floats
        message.values.Add((int) (this.transform.rotation.y * 10000));
        osc.Send(message);

    }
}
```

A.2.3 SceneController.cs

```
using UnityEngine;
using Tobii.Gaming;

public class SceneController : MonoBehaviour
{
    // public objects
    public OSC osc;
    public TextMesh sceneText;
    public Material selectedMaterial;
    public Material deselectedMaterial;
    public Material noTobiiMaterial;

    // used to lerp between materials
```

```
14          new Renderer renderer;
15
16          // eye tracker
17          private GazeAware gazeAwareComponent;
18
19          // use timer to track how long we are looking in
                seconds
20          private float gazeTimeElasped = 0.0f;
21          private float gazeTimeStarted = 0.0f;
22          private const float GAZE_TIME = 2.0f;
23          private bool gazeStarted = false;
24          private bool messageSent = false;
25
26          // track the scene which we are in
27          private int sceneNumber;
28
29          // Start is called before the first frame update
30          void Start()
31          {
32              // setup the rederer for lerping between colors
33              renderer = GetComponent<Renderer>();
34
35              // setup the scene changing object to be gaze
                    aware
36              gazeAwareComponent = GetComponent<GazeAware>();
37
38              // receive the OSC messages
39              osc.SetAddressHandler("/scene", OnReceive);
40
41
42          }
43
44          // Update is called once per frame
45          void Update()
46          {
47              // See if we are gazing on this object
48              if (gazeAwareComponent.HasGazeFocus)
49              {
50                  // set the time started to the current time
                        when the gaze has begun
51                  if (gazeStarted == false)
52                  {
53                      gazeTimeStarted = Time.time;
54                      gazeStarted = true;
55                  }
```

```
56
57                  // start changing the color of the head
58                  if (gazeTimeElasped < GAZE_TIME)
59                  {
60                      // set the elapsed time
61                      gazeTimeElasped = Time.time -
                            gazeTimeStarted;
62
63                      // lerp the material color to show that it
                             is about to be selected
64                      float lerpTime = Mathf.PingPong(
                            gazeTimeElasped, GAZE_TIME) /
                            GAZE_TIME;
65                      renderer.material.Lerp(deselectedMaterial,
                             selectedMaterial, lerpTime);
66                  }
67
68                  if (gazeTimeElasped > GAZE_TIME)
69                  {
70                      // send scene request via OSC
71                      if (messageSent == false)
72                      {
73                          // send scene request to Max
74                          // Max will send back a scene number
                                to confirm the change has taken
                                place
75                          OscMessage sceneRequest = new
                                OscMessage();
76                          sceneRequest.address = "/sceneRequest"
                                ;
77                          sceneRequest.values.Add("bang");
78                          osc.Send(sceneRequest);
79                          messageSent = true;
80                      }
81                  }
82
83              }
84              else
85              {
86                  // change back to deselected material
                        depending on if eye tracker is connected
87                  if (TobiiAPI.IsConnected)
88                  {
89                      renderer.material = deselectedMaterial;
90                  }
```

```
91                  else
92                  {
93                      renderer.material = noTobiiMaterial;
94                  }
95
96                  // reset the gazeStarted flag, gazeTimeElapsed
                        , and messageSent flag
97                  gazeStarted = false;
98                  gazeTimeElasped = 0.0f;
99                  messageSent = false;
100             }
101         }
102
103
104         // stub that is called when receiving an OSC /scene
                message
105         void OnReceive(OscMessage message)
106         {
107             // get the scene number
108             sceneNumber = message.GetInt(0);
109
110             // update the scene number on the text
111             sceneText.text = sceneNumber.ToString();
112
113         }
114     }
```