Depth-Controlled Spatial Audio with High Order Ambisonics in a Speaker Ring

Music and Technology Final Project - David Buzzell (dbuzzell)

Table of Contents

Purpose / Introduction **3**
Dedications 3
Overview **3**
Packages 4
dp.kinect2 4
HoaLibrary v2 4
TouchOSC 5
Hardware Setup 5
Code 6
Patch 1: Kinect Data Collection (Windows) 6
Patch 2: OSC Processing (Mac) 7
Patch 3: HoaLibrary / Audio Output (Mac) 8
TouchOSC Patch (Android) 9
Demonstration **10**
Text Description 10
Step-by-Step Walkthrough 11
Setup 11
Windows/Kinect 11
Host Mac 11
TouchOSC 12
Demonstration 12
Limitations 13
Conclusion **14**
Appendix A: Figures **16**
Appendix B: Package Info **16**

Purpose / Introduction

The purpose of this project is to explore the creative and technical abilities of depth tracking as a controller for spatial audio output using High Order Ambisonics in a speaker ring formation. This project combines many different packages and libraries in an application setting, allowing users to abstract audio from 2 stereo speakers to many speakers in a 360° setup.

Ever since I first applied for the Music and Technology Bachelor's degree, I began exploring the many possibilities for music to be furthered and enhanced by technology, finding where I could fill the gaps between musicians and technology creators. Upon thinking of ideas for my Music and Technology Final Project, I knew that I wanted to work on something that had a very practical and representable demonstration of itself, such that a user to directly see for themselves the benefits of this project and want to start making creative projects right away. As such, this project stands to measure itself by the success of its demonstration and by the satisfaction of the auditory experience above all else. Therefore, this write-up is simply a supplementary description and explanation of the whole project, and should be used in conjunction with a full demonstration of the project for the auditory experience.

Dedications

While the Music and Technology Final Project is meant to be an individual project, this project would not have been possible with the help and support of several key players. Thank you to Tom Sullivan, Jesse Stiles, Sharon Johnston, Riccardo Schulz, Roger B. Dannenberg, the IDeATe network, Dale Phurrough, CICM, Hexler, Cycling '74, Microsoft, and all of my friends and family that have supported me throughout the long nights and frustrating software problems along the way.

Overview

The basic description of this project is a system that tracks your body part locations in a depth field to control spatial audio in a speaker ring. The depth information is tracked by the Kinect sensor (see Figure 1) connected to the Windows computer. This information is filtered down to only the left and right hand depth parameters and is sent to the host Mac computer. It is then processed for volume control and position of source sound, which is output to the soundboard and speaker ring.

This basic dataflow description lacks the information on the many packages and setup necessary for a successful demonstration of this project. Below is more detailed information on these components of the project.

Packages

The following packages are required to be installed and configured before a live demonstration of this project:

dp.kinect2

The *dp.kinect2* package is an external extension designed for Max users to extract information from their Microsoft Kinect sensors. Some of the most information passed on by this package include:

- Depthmap: a grayscale Jitter matrix, viewable as a refreshable image, that displays depth information through the luminosity of the pixels (in this case, the darker points relate to points closer to the Kinect sensor)
- Colormap: a RGB Jitter matrix, viewable as a refreshable image, that is simply the output of the video camera on the Kinect sensor (used mainly for debugger's reference)
- Skeleton data: message information provided after processing by the Kinect sensor that relates to the physical dimensions for "human"-looking objects in frame.
- Joint information: part of the skeleton data, the Kinect sensor breaks up the skeleton into 20 joints (see Figure 2). The two most important joints for this project are the left and right hand.

All information from this package is output in Max-friendly messages, thus making it very accessible within a Max patch. This package does require a personal license to be purchased and used for any computer using the software. See Appendix B for more information on usage and distribution.

HoaLibrary v2

HoaLibrary is a collection of C++ and FAUST classes and objects for Max, PureData and VST destined to high order ambisonics sound reproduction. The whole library is free, open source, and made available by the CICM, the research center of music and computer science of the Paris 8 University. This project uses the Max implementation of this library, on the host Mac computer in the IDeATe Media Lab.

Ambisonics is a surround sound format, meant for spherical audio, that encodes into four channel audio (B-format), separate from the speaker output. This allows the audio to be decoded for any particular speaker setup, without major loss of quality or extra conversion techniques for unique setups. While higher order ambisonics is designed for larger setups (i.e. involving complete spherical speaker setups), lower orders can be used for circular surround sound in a speaker ring with high precision and fidelity.

For this project, this library is used to implement third order ambisonics for a 8 speaker output, along a regularly spaced ring. Further implementations of this project could attempt to reposition the speakers in a spherical shape, but the previous setup of this lab made this loudspeaker setup the most suiting for this project.

In this library in particular, there is the feature as part of the ambisonic encoding process to “optimize” the ambisonic audio for different audiences. There are three possible options for the weighting used by this optimization process:

- Basic: no such optimization is used
- MaxRe: used when the audience is confined to the center of the circle
- inPhase: used when the audience covers all of the circle

There is an included note in the help file explaining this optimization that these methods are purely theoretical and the best choice is to trust your ears. With this in mind, the demonstration of this project uses the “inPhase” optimization for the best showing of the effectiveness of this spatial audio project.

TouchOSC

TouchOSC for Android is a modular Open Sound Control (OSC) control surface allowing the user to send and receive OSC messages over Wi-Fi. *TouchOSC* is a very convenient way to control properties and parameters wirelessly from a mobile device. Even more convenient, *TouchOSC* communicates with OSC messages, which can be sent and received by Max patches since OSC is built on top of UDP (User Datagram Protocol).

In this project, an Android smartphone is loaded with a *TouchOSC* patch that has toggle buttons and faders that directly control the output and tracking of sound sources. This information is sent wirelessly through OSC to the host Mac computer, which formats this information in a Max patch and forwards it to audio processing. There is also information from the computers (i.e. Kinect depth information, source volume information) that is sent back to the Android smartphone via OSC, for debugging and quick reference.

Hardware Setup

In conjunction with these software packages, there are several hardware components that are necessary for this project. Starting with the depth capture, this project uses Microsoft Kinect for Xbox One sensor with the Kinect Adapter for Windows PC. This is connected via USB 3.0 and configured using the Kinect SDK. After successful setup and configuration, the dp.kinect2 package can interface with the Kinect sensor via the SDK.

Since the Kinect SDK is only officially supported on Windows machines and IDeATe's Media Lab hosts a Mac desktop, the information is distributed between both computers. On the Windows computer, there is a Max patch that only collects the Kinect tracking information and sends it to the Mac computer through UDP. All of the audio processing is done on the Mac computer, with two separate Max patches running concurrently.

Moving to the sound output, IDeATe's media lab uses specific hardware for their speaker ring. From the computer, the audio runs to the soundboard, in this case the Behringer X32 (see Figure 3), through an ambisonic preset set for an 8 speaker setup. The media lab's host computer can use this setup by using the 'uTRACK32' audio device in Max 7.

These speakers are then connected to the audio outputs of the soundboard, in a ring formation evenly spaced along the perimeter. These speakers were spaced along a 4 meter diameter circle, with their positions marked on the floor in green tape (see Figure 4). The speaker outputs were basically predetermined by the setup of the audio output cables, but the setup was made to be as close as possible to the representation in the Max patch on the hose Mac computer.

Code

The majority of code developed for this project was developed in the Max 7 environment. There are three main patches that run concurrently (one on the Windows machine, two on the Mac machine) with very specific purposes, as to minimize wasted bandwidth and computation lag. Pictures and visual aides for each patch can be found in Appendix A for figures.

Patch 1: Kinect Data Collection (Windows)

The first patch of the dataflow line starts on the Windows machine with the Kinect sensor attached. This patch uses the *dp.kinect2* plug-in exclusively, so the license should be registered to the windows machine running this patch (see Appendix B for more details on license registration).

In order to start the data collection, a toggle button activates the *open* command and the *qmetro* (queue-based metronome) to the *dp.kinect2* object. The *open* starts the stream from the Kinect sensor, and the *qmetro* polls it every 33 milliseconds (0.033 seconds, which is close to the framerate of the Kinect sensor at 30 fps), allowing for small drops and jitter in the frames. Using *@depthmap 1* and *@colormap 1* as arguments in the *dp.kinect2* object, it outputs a Jitter matrix of both depth and color that we can view as streaming video using two *jit.pwindow* objects for visual reference. With *@skeleton 1*, we also get depth coordinates for the skeleton data (as x-y-z-confidence points) from the Kinect sensor.

Using Max's *route* and *zl.slice* objects, the patch can pack only the left hand (*l_hand*) and right hand (*r_hand*) information into a combined buffer. This buffer is then sent via UDP to the host Mac computer, which will process this packet of information, eventually for audio output.

Currently, no information is sent back to the Windows machine or this patch, since the Kinect sensor and *dp.kinect2* package take up a lot of bandwidth and processing power. With more powerful and capable machines, it is possible that all processing could be done on one machine and then just sent out to the audio output, especially if internet connectivity or UDP unreliability is a severe issue.

Patch 2: OSC Processing (Mac)

This second patch is the main hub of all the information that is used in this project. All of the messages and code run through this patch, including the other Max patches and *TouchOSC* patch, either via OSC/UDP or local messages sent via Max patches (i.e. the *send/receive* objects in Max). There is also a large amount of debugging information that is viewable and managed in this patch as well.

The first patch sent packets of the left and right hand joint information to this second patch via UDP. Using the correlated UDP information for the Windows computer, the second patch receives the joint information via the *udpreceive* object. From there, several *zl.slice* objects parse the joint information into the individual coordinates of a left and right hand, saved in appropriate variables (i.e *rx, ry, rz,* etc.).

From the *TouchOSC* patch (see TouchOSC Patch: Android), the button information sent via an OSC message is parsed from the *fromphoneudp* object, using *zl.slice* and *fromsymbol @separator /* objects. The button information from the toggles are then directed to *gate* objects that control the position of the audio sources in the 2D map. These positions are calculated from the X and Z coordinates of the right hand (*rx, rz*), with some adjustment using the *scale* objects. From empirical testing in the current setup used in the IDeATe Media Lab, I found the maximum and minimum values of the X and Z coordinates to be about the ones used in the following table (see Figure 5). These scaled coordinates are then sent to *pak* objects that format them to be used in a GUI object, *hoa.map*, that graphically displays the location of each source sound (activated by each toggle) on a cartesian map (see Figure 6). The *sourcelist* from *hoa.map* is sent via local Max message (*s sourcelist*) to Patch 3.

The toggles also control two other gates that control the effect of the Y coordinate of the right hand (*ry*). When opened, the Y coordinate is scaled by similarly empirically tested maximum and minimum values (see Figure 5) and then passed into Patch 3 as the gain control for each audio source (*s gain1*, *s gain2*). This gain volume is also sent via OSC to the *TouchOSC* patch, to give a visual reference on the mobile device of the relative gains for each audio source. This means that while the X and Z coordinates of the right hand control the location of each audio source on the circular plane to the loudspeakers, the Y coordinate relates to the volume of each source.

There is a final button that relates to the master On/Off switch for all audio. This signal from the *TouchOSC* patch is accepted here and then directly routed to the third patch (*s space*).

The last part of this page is all of the debugging functions that helped when testing and creating this project. *TouchOSC Debug* is a simple OSC/UDP test function that turns on a light if it receives a push from the *TouchOSC* patch. *MasterUDP Debug* is a parsing function for every OSC message sent from the mobile *TouchOSC* patch. It uses the same parsing objects and functions as described earlier, and it also adds visual buttons to indicate when each of the toggles are activated on or off. The final debug set, *Kinect Debug*, simply takes the right hand coordinates (*rx, ry, rz, rc*) and sends them to GUI and text values via OSC to the *TouchOSC* Patch.

Patch 3: HoaLibrary / Audio Output (Mac)

Once the positions of the audio sources are calculated by *Patch 2*, *Patch 3* takes this information and does the necessary calculations for the audio output. These positions are sent directly to a *hoa.2d.map~* object, which will encode the audio using third degree ambisonics based on the positions (*r sourcelist*). In addition, the gains calculated from the Y coordinate (*r gain1*, *r gain2*) are fed into two gain faders, along with two *selector* subpatches that select the audio sound (*cycle~*, *saw~*, *pink~*) based on the menu selection. From the *hoa.2d.map~* object, we get 4 channels of output (7 signal lines):

- W (harmonic 0): similar to an omnidirectional microphone pickup
- -X/+X (harmonic -1/+1): similar to a figure-eight microphone pickup (front/back)
- -Y/+Y (harmonic -2/+2): similar to a figure-eight microphone pickup (left/right)
- -Z/+Z (harmonic -3/+3): similar to a figure-eight microphone pickup (up/down)

These channels are sent to a viewing object (*hoa.2d.scope~*) for a pictorial representation of the sound field as it is encoded. This representation is supposed to simulate the sound pressure fields of the two sources encoded, relative to the center of the map. It works to see a rotating audio source in the ring, but it not the most intuitive viewing object for a listener in the ring (see *hoa.2d.meters~*, discussed below).

Also attached to the same encoded lines is a optimization object, *hoa.2d.optim~*. While not necessary for this project, it can be helpful in order to get the best auditory experience for a demonstration. In this project, the main optimization used is the *inPhase* method.

After optimization, the signal lines are decoded by a *hoa.2d.decoder~* object. The decoding process can be very different and tailored for each type of setup, with many different parameters to adjust. Here are some of the few that are used in this project:

- *Channels*: slightly confusing wording, for the channels this attribute refers to the number of **loudspeaker** channels (not ambisonic channels, which is 4 for this third degree ambisonic setup).
- *Mode*: the decomposition mode for the particular type of loudspeaker setup. *Ambisonic/Regular* for loudspeakers evenly spaced in a circle. *Binaural* for two speakers, ideally in a stereo "headphones" setup. *Irregular* for any loudspeaker setup not evenly spaced on a circle.
- *Angles*: the degree placements along the circle for each loudspeaker. This really is only important for *irregular* setups, as both *ambisonic* and *binaural* have set positions on the circle.

For this project, there are <u>8</u> channels, set up in a <u>ambisonic</u> arrangement. These parameters are also important and get forwarded to the visual meter graphic object, *hoa.2d.meters~*. Unlike

the encoded GUI object (*hoa.2d.scope~*), these meters are meant to measure the actual gain outputs from each speakers, a more user-intuitive check. The 8 decoded signal lines are sent to a Digital-Audio Converter (*hoa.dac~*), which is then sent to the Behringer X32 soundboard and then to the loudspeakers.

There are two ways to control the audio "power switch" for this patch. The original way, for debugging purposes, used a *key* object to turn the switch on and off using the spacebar (#32). In an actual demonstration, it is far easier to use the *TouchOSC* patch's yellow "Audio On/Off" switch located near the toggle switches on the second page (see TouchOSC Patch: Android). The patch will only reset audio gains on a fresh restart of the patch, so be careful when turning on the audio output to check for any high values left off the last demonstration.

TouchOSC Patch (Android)

The *Max* patches process and calculate most of the data and values. However, the *TouchOSC* patch controls the dataflow and calculations, using GUI controls that send OSC messages to Patch 2. The *TouchOSC* patch has two pages: the first one for debug functions, the second one to control the audio sources.

The first page, *Debug Page*, starts with a *UDPReceive/UDPSend Test* on top of the page, with a toggle button and LED. When the toggle button is activated, the LED light should switch on and off if the mobile device is connected on the same network as the host MAc computer. These tests work together show successful UDP/OSC connection. Below these tests are three faders, corresponding to the X, Y, and Z coordinates of the right hand. These coordinates, sent from Patch 2, give a visual representation to the user of their right hand coordinates. Below that is the confidence value for the current coordinate reading, or how accurate the Kinect sensor believes its tracking is. Since it only takes three possible values (i.e. 0, 0.5, 1), the green LED will only light for a confidence value of 1, ensuring that the current tracking is the most accurate as possible. Tracking near or outside of the "maximum/minimum" values (see Figure 5) will result of low confidence values, and may result in audio "dropouts" or poor auditory experience.

The second page, *App Page*, is only two toggle buttons, two corresponding faders, and a small On/Off switch in the bottom-lefthand corner. This small switch is the master audio on/off switch, a simple OSC message to Patch 2 (*s space*). The toggle buttons, corresponding to each audio source, allow the position of the sound source to be changed by the X and Z coordinates of the right hand when the toggle button is active. Once deactivated, the source's position is kept in the last place it was tracked. The faders give a visual meter for the gain for each source, which are also only changed when the corresponding toggle is activated.

It is important to note that while this project uses a mobile device as a controller via *TouchOSC*, the phone does not provide any tracking information or depth coordinates to the other Max patches. The Kinect sensor only tracks the right hand and reports those coordinates, not anything on the phone in particular. This means you can hold the phone in any hand or position

without it affecting the audio, but be mindful of the right hand throughout the whole demonstration.

Demonstration

This write-up is meant to be a supplement to an actual live demonstration for this project. In the event that this is not possible, a video demonstration can be found here (see Figure 7). In addition, below is a textual description of the demonstration, along with a detailed setup guide in order to replicate the demonstration on personal equipment.

Text Description

Begin in the center of the speaker ring, facing in the directions of the arrows taped on the floor, to where the Kinect camera is located. To test that the mobile device is connected to the same local network properly, perform the *UDPReceive/Send Debug* test. Once complete, turn on the Windows Max patch, which will begin joint tracking and reporting this information to the mobile device. Move the right hand independently in all 3 directions (relative to the Kinect): X direction (left-right), Y direction (up-down), Z direction (near-far). Make sure that your movements match with the coordinate readings on the mobile device.

Once the configuration tests are complete, begin the actual demonstration. Turn on the first audio source and control the gain using your right hand in the Y direction. After reaching a comfortable gain level, perform a circular walk inside the speaker ring to control the position of the sound source. Return to the original spot, and turn off the audio source. Repeat these instructions for the second audio source.

After this, be creative now that the basics have been demonstrated. Place the sound sources in two separate areas of the ring, move them together or separately, change them with different gains. Turn all the audio off and you have a complete demonstration of this project.

Step-by-Step Walkthrough

This walkthrough is broken down into many parts, based on setup and demonstration parts, and then based on devices needed. The setup parts can be completed in any order; the demonstration parts should be completed linearly. All of the links for code and installation files can be found in Appendix B.

Setup

This setup does not include any information to replicate this project outside of the IDeATe Media Lab. There are plenty of setup options and features for the Host Mac machine and the Speaker Ring that are not discussed in this guide. This project has only been tested and demonstrated in the IDeATe Media Lab, and thus is not guaranteed to work in any other setups or equipment.

Windows/Kinect

1. Startup your Windows machine. Make sure you are running a 64-bit version of Windows 10 (Win 8.1 minimum) with a 3.1 GHz processor (2.6 GHz used in demonstration) and a devoted USB 3.0 port for the Kinect v2 (or Kinect adapter for the Xbox One sensor).
2. Install the Kinect Configuration Verifier and the SDK. Plug in the Kinect USB port and run the Verifier and unsure everything passes. Place the Kinect sensor on top of the loudspeaker that designates the top of the ring.
3. Make sure you are running Max 7 with a proper license. Install the dp.kinect2 package (most cases, just put the whole library folder inside *Documents > Max 7 > Packages*) and register your purchased license.
4. Run the *kinectcollector.maxpat* patcher, make sure the depthmap and colormap are correct.
5. Connect to the local network used in the Media Lab (see Figure 8). Keep track of this machine's IP address.
6. Press Ctrl+E to get into edit mode, then Ctrl+Alt+E to get out of presentation mode. In the *udpsend* object, change the IP address to the IP address of the host mac. Make sure the port number listed matches with the host mac.

Host Mac

1. Turn on the Mac machine in the IDeATe Media Lab. Sign on to *Media Lab User* (no password).
2. Open these two patches: *oscprocessing.maxpat*, *hoaoutput.maxpat*
3. Double-click on the *hoa.dac~* object in the *hoaoutput* patch to open up Audio Settings. Make sure the first few settings match with these (see Figure 9).
4. Connect to the local network used in the Media Lab (see Figure 8). Keep track of this machine's IP address.
5. In the *oscprocessing* patch, press Ctrl+E to get into edit mode, then Ctrl+Alt+E to get out of presentation mode.
    a. In the *udpreceive* object connected to *fromphoneudp*, make sure the port number matches the outgoing port in *TouchOSC*'s settings.
    b. In the *udpsend* object connected to *tophoneudp*, change the IP address to the phone's IP address. Make sure the port number matches the incoming port in *TouchOSC*'s settings.
    c. In the *udpreceive* object connected to *fromkinectudp*, make sure the port number matches the windows machine.

TouchOSC

1. Install *TouchOSC* onto your smartphone of choice. Download *controller.touchosc* onto this device as well.
2. In Settings, go to Layout, Add from File, and select *controller.touchosc* within your phone's file navigation system.
3. Click Done to open the layout used for this project.
4. Connect to the local network used in the Media Lab (see Figure 8). Keep track of this machine's IP address.
5. In Settings, go to OSC. Make sure the host matches the IP address of the host mac. Make sure the port numbers match the opposite direction of the ports from the host mac (i.e. outgoing should match *fromphoneudp*, incoming should match *tophoneudp*).

Demonstration

1. Open the *oscprocessing* and *hoaoutput* patches on the Host Mac machine. Turn on the Behringer X32 and set it to the ambisonic preset. Make sure the speaker are all plugged in and turned on.
2. Open the *controller* layout in *TouchOSC*. Click on the yellow button below "UDPReceive Test". If a yellow light toggles on/off, the *UDPReceive/Send Test* has passed.
3. Click on the "X" button on the *kinectcollector* patch. Check that the depthmap and colormap are on and correct.
4. Stand in the center of the speaker ring, facing towards the Kinect sensor. Move your right hand to the left, making sure not to go outside of the limits (if the green "confidence" light is off, you are outside of the limits). The X coordinate slider on the *TouchOSC* layout should slide to the left with your right hand. Repeat for moving the right hand to the right.
5. Move your right hand up, away from the ground, making sure not to go outside of the limits. The Y coordinate slider on the *TouchOSC* layout should slide up with your right hand. Repeat for moving the right hand to the ground.
6. Move your right hand away from the sensor, making sure not to go outside of the limits. The Z coordinate slider on the *TouchOSC* layout should slide down with your right hand. Repeat for moving the right hand towards the sensor.
7. Open the second page on the *TouchOSC* layout. Press the yellow on/off button in the bottom-lefthand corner.
8. Select the red "Source 1" toggle button. As you raise your right hand up, the red slider should also increase as well as the audio gain for the first sound source.
9. With the "Source 1" toggle still active, walk around the speaker ring slowly. Your position should be tracked and visible on the *oscprocessing* patch by the *hoa.map* object.
10. Once the walk is complete, return to your original spot, lower your right hand to lower the gain as much as possible (without losing Kinect confidence), and deselect the "Source 1" toggle.
11. Repeat steps 8-10 for the blue "Source 2".

12. Use the “Source 1” and “Source 2” to place the two sounds at different gains, with different locations in the speaker ring. Once creativity is complete, press the yellow on/off switch again to shut off all audio.

Limitations

As with any project, the theoretical success is always limited by the real world factors that add up from each components’ limitations. This project aimed for the highest auditory experience over all else, which means some things were not as preserved or maintained properly. This is also a proof-of-concept project, and with some tweaking this project could be extended to many different creative and useful applications.

Starting at the beginning of the project, the Kinect sensor is powerful but not a perfect device. It has a cone-like pickup pattern (see Figure 10), with limits that are pretty hazy and not well defined. Sometimes the device will perform very well outside of its stated limits, and then other times it will drop unexpectedly. The device can also perform in a “near-mode”, which is mainly used for facial detection and closeup gestures, but it switches into this mode arbitrarily and without outside control or influence. Thus, the Kinect is not the perfect device to use in a circular shape testing near its limits. Thankfully, following the confidence values give a rough idea of how well the Kinect is performing near the limits, and with enough experience a full demonstration can be given without audio dropout.

In regards to dropout, the most nerve-racking part of developing this project is in how the information is sent wirelessly between three devices. *UDP* is known to be “unreliable” for critical applications, since messages can be dropped or never sent without checking or confirmation messages. Since *OSC* is built on top of *UDP* as well, there can be times where messages from any of the devices are missed or not sent. However in practice, it seems that this demonstration does not put much overload to the network, thus there is no real reason for messages to be dropped regularly. It is possible that some of the coordinate information is not sent, but the framerate is so high such that a few dropped coordinates does not affect the audio enough. Thankfully all of the audio signal is wired, so there should not be any major dropout or jitter issues. Sometimes the *TouchOSC* messages can be delayed much more than the *UDP* messages. This is probably due to the GUI aspect to *TouchOSC*, and this demonstration happened to use an older smartphone with relatively weak processing power as compared to other devices on market as of writing this write-up. All in all, there may be some dropouts in messaging, but not enough to detract from the experience.

These are the two main factors for any sort of detraction from a demonstration. In the original demonstration plan, there was hope to have multiple people controlling each audio source, each with different parts of the body. While the Kinect is able to track multiple people, the issue came down to demonstration capability, as it did not seem that there would be enough performance space for each person to control their audio source separately. There was also plans for gesture control and using Machine Learning to recognize certain movements to perform switches or

tasks. Unfortunately, the other libraries used in this project took up a lot of time to research and configure, and so the ML libraries were never added to this project. With more time and resources, it would be exciting to implement all of these goals and ideas in a fuller project. In researching the ambisonics library, the original plan was to demonstrate this audio in a spherical bubble instead of a circular ring. As much as the desire was there to make that spherical speaker system, the IDeATe Media Lab was best configured for a circular ring setup and thus it persisted into this demonstration.

Conclusion

This Music and Technology Final Project provides a demonstration of the abilities of using depth information as a way to control ambisonic audio in a speaker ring formation. This project is just one of the many possible demonstrations to this creative endeavour. There are many factors to this project that can be expanded, continued, or perfected. With more time and resources, there's plenty that I would wanted to fix and redo for myself. However, this project is much more than the end goal for me. I learned even more about each individual aspect of this project, how to use these packages, the theory behind everything, and the many different ways these things can be used to achieve even bigger tasks and goals.

The idea for this project first started as the basis of another research project, featured in Jesse Stiles and Silas Riener's *Blue Name* (see Figure 11). The project was to use Kinect sensors to track body movement and joint locations in order to adapt and change musical sounds and effects, in realtime. The project opened a whole new experience for myself, and I started looking for the next application of this project. I eventually met up with Jesse again to talk about the "next big thing" in the audio world, and ambisonic technology came up. Ambisonics, which I knew a little bit about from VR/AR, allowed for spatial and 3D audio in a much more simplistic and manageable way than I had previously thought possible before. Combine the two ideas together, and that's essentially where the idea of this project came from. For me, this project has been exciting to see from start to finish, and a great way to show my growth and understanding in Music and Technology since I got to build up on this project from previous experience.

The whole point of art and music is not what the artist intends with their work; it is the listener's experience that will determine the worth or contribution of the artist's work. I hope that anyone who gets to experience this live demonstration of this project can get a new inspiration for something they have wanted to created, maybe build and expand upon this project even. There are endless possibilities of the applications of this project and the intricate details of it. If nothing else, I appreciate the time and experience that I have been given to explore this creative idea, and how willing and open everyone has been in allowing me to complete my degree program with a project I am very proud of. Thank you for taking the time to read this write-up. I hope you get to enjoy a full demonstration of this project.

Appendix A: Figures

Figure 1: Pictures of the Kinect Sensor

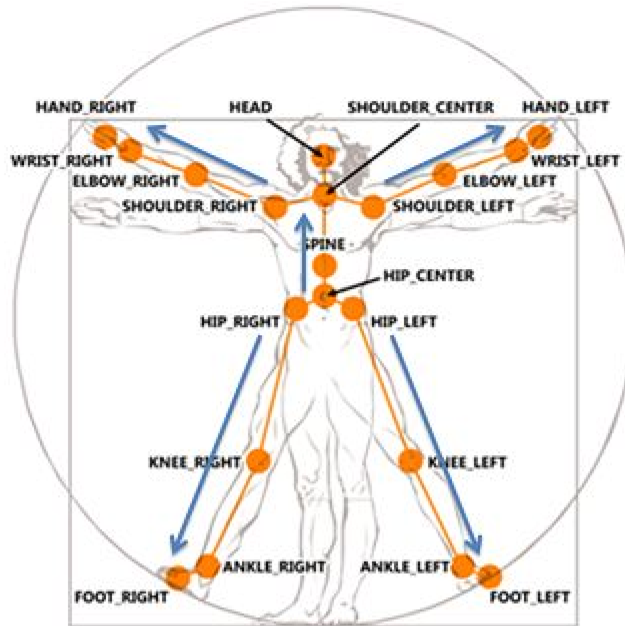
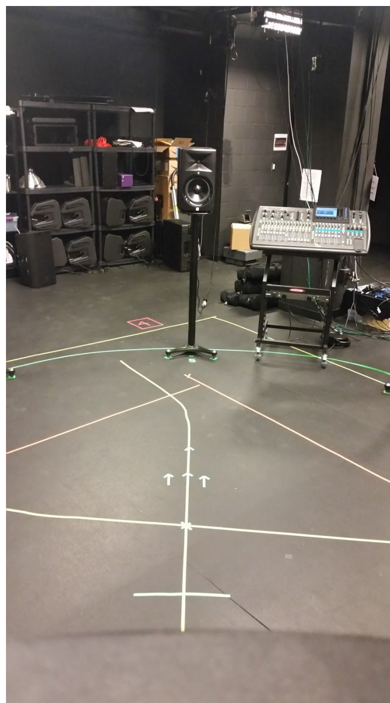


Figure 2: Locations of the 20 Kinect-trackable joints

Figure 3: Behringer X32 used in the IDeATe Media Lab



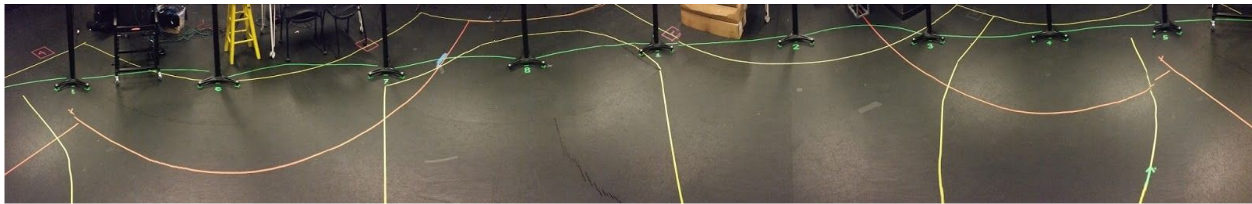


Figure 4: Pictures of the Speaker Ring and the Green Tape Markings

	Minimum Value	Maximum Value
X	-2.	1.85
Y	-1.5	0.5
Z	-0.65	4.

Figure 5: Empirically Tested Bounds for Right Hand coordinates

NOTE: These were tested using the full demonstration of the project and were evaluated by the limit in which the Kinect sensor lost reliability to confidently track the right hand. Since the Kinect has a cone shape of vision for tracking, these values are rough approximations to the cartesian limits of the circular ring space, and thus may differ from the actual limits within a margin of error. It is demonstrable that these limits are satisfactory for a wide enough range of position control, but should not be tested at its extremes for the best auditory experience.

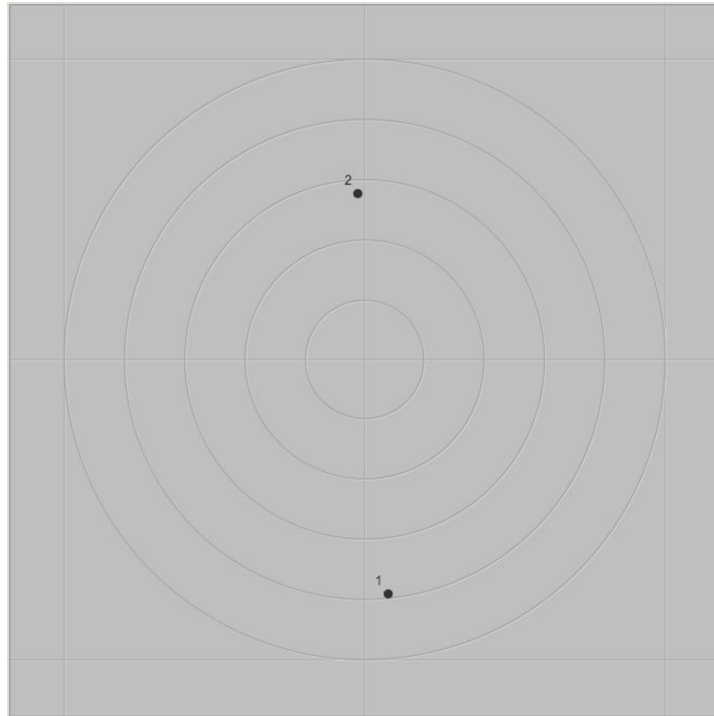


Figure 6: *hoa.map* GUI object, in Cartesian Mode

https://www.youtube.com/watch?v=XfvVWxyaobQ
Figure 7: Link to Video Demonstration

SSID: MediaLabLighting
Password: the media lab
Figure 8: IP Settings for the local network (include the spaces in the password)

Figure 9: Max Audio Settings for Host Mac

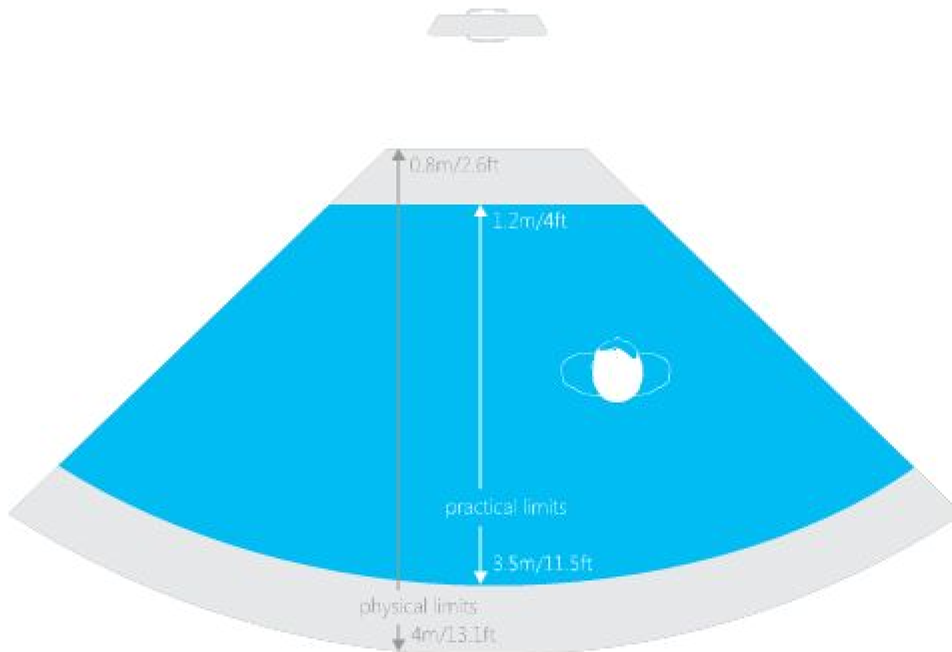


Figure 10: The Kinect's pickup pattern

https://vimeo.com/137528856
Figure 11: Link to *Blue Name*

Appendix B: Package Info

More information on Kinect's skeleton data can be found here:
https://msdn.microsoft.com/en-us/library/hh973074.aspx

More information on the IDeAte Media Lab and the Behringer X32 can be found here:
https://github.com/jts3k/MediaLab/blob/master/sound-console-tutorial.md

The Terms and Conditions of the *dp.kinect2* package can be found here:
https://hidale.com/terms/

The *dp.kinect2* package can be purchased here:
https://hidale.com/shop/dp-kinect2/

The *HoaLibrary* package can be found here:
http://www.mshparisnord.fr/hoalibrary/en/downloads/max/
https://github.com/CICM/HoaLibrary-Max

More information on Ambisonics can be found here:
https://ambisonic.info/
https://www.youtube.com/watch?v=C0xRAf9-XeU
https://www.youtube.com/watch?v=DVZ2U7uLT0w

More information on *TouchOSC* can be found here:
https://hexler.net/software/touchosc-android
https://hexler.net/docs/touchosc-configuration-connections-osc

Appendix C: Install Locations

Here is a Github repository of the Max patches and *TouchOSC* layout used in this repository:
https://github.com/xhamyd/MusicTechFinalCapstone

Here is the most recent link to the Kinect Configuration Verifier:
https://developer.microsoft.com/en-us/windows/kinect/hardware-setup

Here is the most recent link to the Kinect SDK:
https://www.microsoft.com/en-us/download/details.aspx?id=44561

Here is the link to most recent Cycling ‘74 Max platform:
https://cycling74.com/downloads