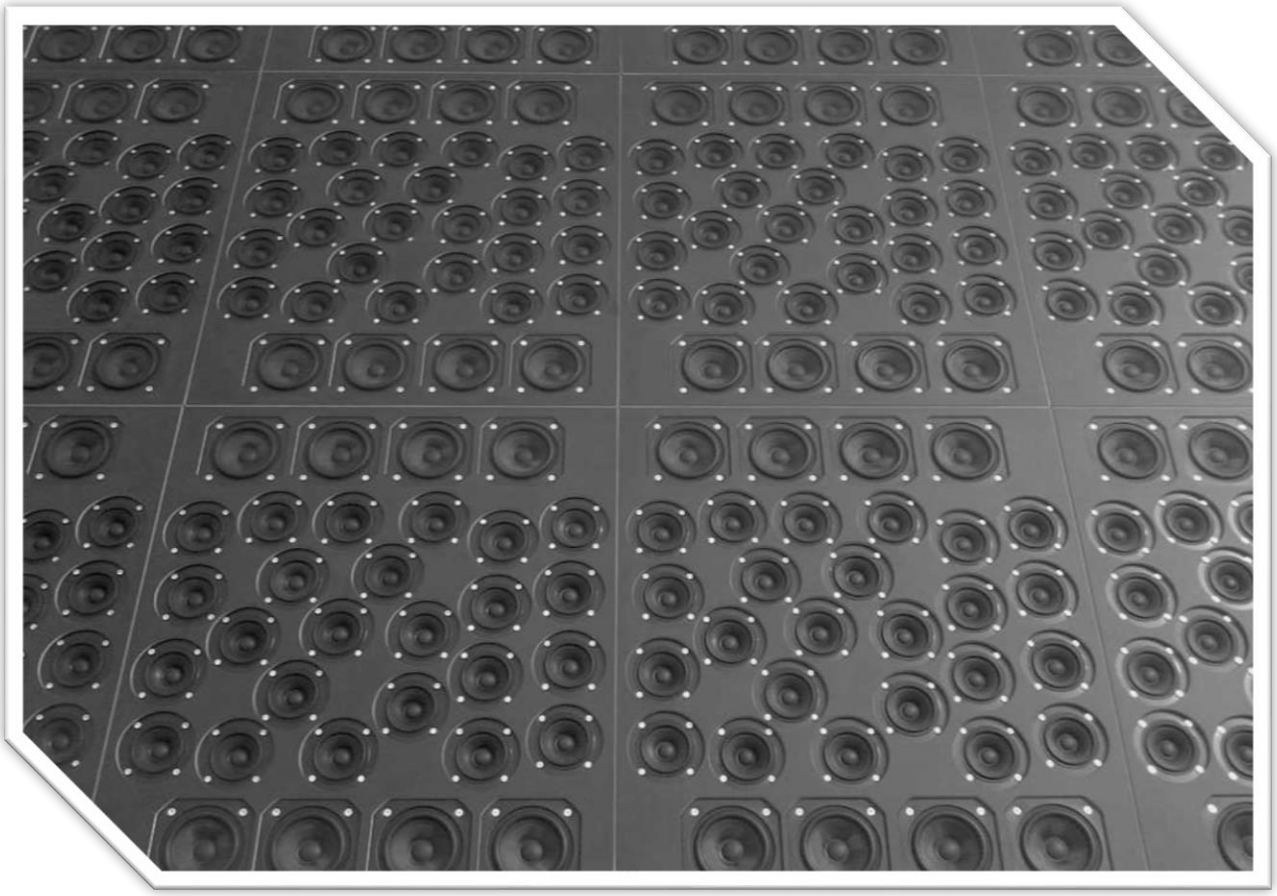

Wavefield Synthesis for Max/MSP

THE CREATION OF [WFS~]

Razvan Paisa | Master Thesis | June 27, 2016

Table of Contents

1. Introduction

Wavefield synthesis (WFS) is a rather new spatial audio rendering technique, which has a very unique characteristic: the localization of a virtual sound source does not chance with the listener's position. In other words, it does not have a so called listening sweet spot. Developed in the late 80s, it has witnessed moderate popularity, it`s main drawback being the complex and expensive hardware necessary to operate. A typical Wavefield synthesis system can have anywhere between 16 and 500 speakers. Fortunately this aspect is overcame by the opportunities it provides, allowing a new dimension of audio content to be explored. There have been musical concerts developed for WFS, games, as well as a lot of scientific research. Now that Virtual Reality is becoming mainstream, WFS could provide the sound necessary for an amazing multisensory experience.

Unfortunately another barrier in working with Wavefield synthesis is the software aspect. Most of the system existing in the world were found in University laboratories, where skilled researchers were writing custom software for their own usage. Nowadays there are companies like IOSONO who are pushing the technology commercially, but their software is not freely available.

This project is proposing to create an external Wavefield Synthesis object for a popular visual programing software called Max/MSP. This program is modular and very flexible, allowing its users to create anything from artistic installations to scientific applications. By providing developers with a easy to use tool, anyone will be able to create content for WFS, even if one does not own such a system. Hopefully this will attract interest from many areas of expertise and new applications will be found for Wavefield synthesis.

2. Problem Statement

How will a Max/MSP external object for Wavefield synthesis perform in accurately reproducing the location of virtual sound sources?

3. Analysis

Wavefield Synthesis is a technique that combines aspects of acoustics and psychoacoustics and in order to develop a software for it, research in to the aforementioned topics need to be conducted. Since the goal is to write an object for Max/MSP a solid knowledge foundation is crucial to efficiently create an external. This chapter will describe the fields that have been investigated in order to acquire the knowledge required to implement a solution to the problem statement mentioned above.

A. Object based audio production

Since the invention of stereophony in 1881 (Rumsey, 2001), the assignment of a given number of audio channels to a set of loudspeakers is (in most cases) fixed, as well as the positions of the loudspeakers themselves. The loudspeaker setup is implicitly coded in the loudspeaker driving signals. This is known as channel based reproduction and it requires a separate mix for each new reproduction setup. The only way to experience the intended sounds is to have a loudspeaker position that are similar to the positions used in the original mix (Geier & Spors, 2012). This is not a problem if the dedicated speaker setup is a two channel pair of headphones. The consistency of headphones setup allowed for binaural techniques to evolve and reproduce 3D sound accurately. Unfortunately, this is the only case when a speaker setup is similar with the one used in the mixing stage of any audio material. In the last few decades, there have been developed several massive multichannel reproduction systems. Some of those system are made up of hundreds of individual driven speakers, making the channel-based approach not feasible anymore. Not only is the sheer number of loudspeaker channels impractical to store and transmit, but also there are no standardized layouts and each system potentially has a different number of loudspeakers and different loudspeaker positions which would require an individual mix for each system. To avoid these limitations, an new method has been approached that changes the focus from the playback medium (the speakers) to the playback content, and it`s called object-based mixing (Geier & Spors, 2012). This means that instead of storing output signals, the source signals are stored as audio objects, together with their intended position and other parameters. All audio objects together form a spatial audio scene which can be stored. This method not only avoids creating and storing a mix for each reproduction setup but also makes interaction with the scene during playback possible. This concept is not limited to loudspeaker systems, a scene can also be rendered for binaural reproduction with headphones. A channel-based mix can easily be created from an object-based representation by rendering the scene to a given setup and storing the signals of the output channels.

A big implication of using an object based approach instead of channel based is that it required the creator to account for source position in the production step, as opposite to leaving it to the mix engineer. Instead of generating the final output signals by, for example, manually panning individual tracks between pairs of output channels, the sound engineer uses them as source objects and stores their virtual position and other parameters in a scene description. How exactly the actual output signals will be generated from this information is not known during the mixing

process, as it depends on the reproduction system which will be used. Therefore, the task of the sound engineer changes from creating the best possible output signals to create the best possible scene description (Geier & Spors, 2012).

The object based approach is the foundation of many new techniques for reproduction of 3D sounds, including Wavefield Synthesis.

B. Wavefield synthesis (WFS)

In order to be able to develop a fully function WFS object, understanding the technology is of utmost importance. Wave Field Synthesis is a method used to recreate an accurate replica of a sound field using the theory of waves and generation of wave fronts (Brandenburg, Brix, & Sporer, 2009). It enables the generation of sound fields with natural temporal and spatial properties within a volume or area bounded by arrays of loudspeakers (de Vries & M.Boone, 1999). The basic idea was introduced by in 1988 Guys Berkhout from Delft University of Technology, and was first published in the Audio Engineering Society (AES) journal (Berkhout, deVries, & Diemer, 1989). Over the next 10 years the technology was mainly developed in the Delft campus, and now is regarded as an important technique for reproducing 3D audio without the "sweet-spot" limitation of previous methods.

Wavefield Synthesis is fundamentally based on the wave propagation principle described by the Dutch mathematician Christiaan Huygens. He stated that any spherical wave inputs energy in the neighboring particles in the medium which in turn radiates another spherical wave. These particles can be viewed as secondary sources for the original wave. By summing the waves emitted by these secondary sources, a waveform indistinguishable from the original wave is created (figure 1). In other words, this principle states that any wave front can be regarded as a superposition of elementary spherical waves. This means that any wave front can then be synthesized from such elementary waves (Game of Life Fundation, 2010). In wave field synthesis, every point is modelled with an array of loudspeakers, each of them contributing to the desired sound field. Each loudspeaker in the array is fed with corresponding driving signal calculated by means of algorithms based on the quantitative formulation of the Huygens-Fresnel-Principle, which states that a propagating wave front can be synthesized by a superposition of simple sources placed on the wave front. The Kirchhoff-Helmholtz integral implies that an infinite number of monopoles and dipoles encircling the reproduction space is necessary to achieve perfect results. "Perfect results" includes the property that the reproduced sound field outside the listening space (behind speakers) is zero. Taking either monopoles or dipoles instead of both the sound field inside is the same and only the sound field outside is non-zero. Today most implementations of WFS are based on monopoles only. A second step to simplify WFS is to reduce the sound-field from 3D to

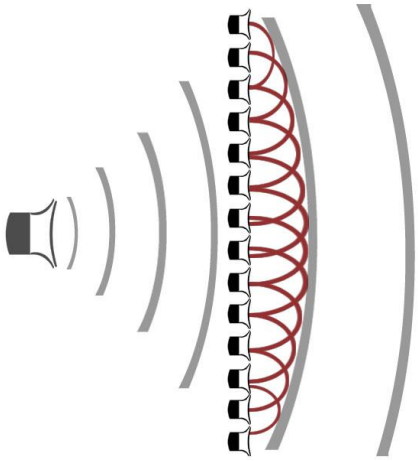

Figure 1 - WFS Huygens principle

2D, therefore all loudspeakers are located in one plane. Reducing the number of loudspeakers to a finite number limits the frequency up to which WFS provides perfect reproduction. Above the alias frequency spatial alias terms occurs. In practice it proved to be sufficient to locate a loudspeaker every 17 cm, giving an alias frequency of about 1 kHz (Spoer, 2004). From an acoustic point of view this seems to be insufficient, but due to psychoacoustic effects, decreasing the distance between loudspeakers has only marginal effects on audio quality. Work conducted on the subject of Wave Field Synthesis has allowed for a very simple formulation of the reproduction of omni-directional virtual sources using a linear loudspeaker array. Practically speaking a computer controls a large array of individual loudspeakers arranged as arrays around the listener and the computer activates each solitary loudspeaker membrane, at the time when the virtual wave front would pass through it. The driving signals for the loudspeakers composing the array appear as delayed and attenuated versions of a unique filtered signal. The maximum spacing between two adjacent loudspeakers is approximately 15 to 20 cm. This allows for optimal localization over the entire span of the listening area (Corteel & Caulkins, 2004). The spatial resolution can be increased by using a higher number of smaller speakers. An interesting setup can be found installed in the Multi-Modal Measurement Lab at the Chair of Communication Acoustics, TU Dresden (figure 2). This system consists of 464 loudspeakers and 4 subwoofers. The tweeter loudspeakers were installed with a very little spacing of 6 cm to reduce spatial aliasing artifacts.

Figure 2 - WFS system, T.U. Dresden

Using a Wavefield synthesis system one can usually reproduce three separate types of virtual sound sources:

- Point sources: virtual sources situated behind the loudspeaker array. This type of source is perceived as having a fixed position from anywhere within the system installation. The position remains stable for a single user moving around inside the installation (figure 3, b).
- Plane waves. These sounds are similar to point sources, the only difference being that the virtual position is seemingly infinite far away behind the loudspeakers. This category of sounds cannot be encountered in the real world. A good analogy is the experience one has when travelling inside a car and looking at the sun/moon. One can entertain the impression that the celestial is “following” the automobile, while the landscape passes along at high speed. The sensation of being “followed” by an object that retains the same

angular direction while one moves around inside of the listening area accurately describes the effect of a plane wave (Corteel & Caulkins, 2004)(figure 3, a).

- Focused sources: virtual sources situate in front of the array. While having similar properties as "normal" point sources for most of the listening positions, for positions between source and closest loudspeaker the sound field is inverted and there is no precise location perceived (Spoer, 2004). These virtual sound sources are created when a wave front created by the loudspeaker array converges onto a fixed position inside of the listening room (figure 3, c).

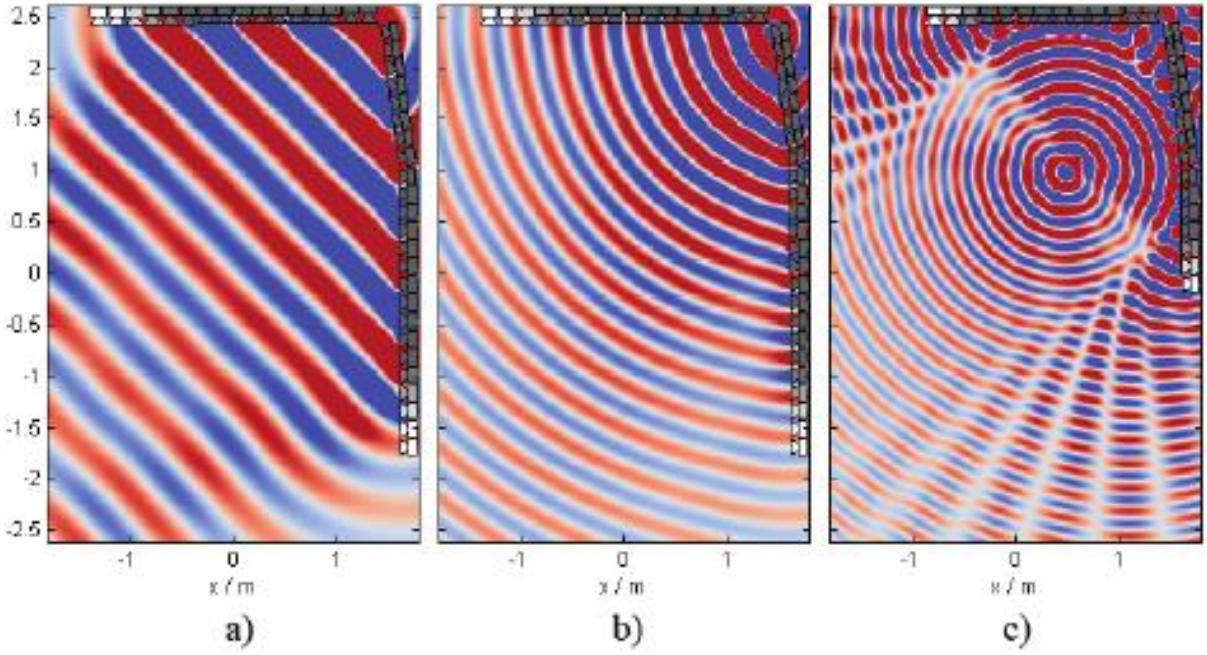

Figure 3 Wave field synthesis simulations performed with the Sound Field Synthesis Toolbox (Gari & Kob, 2015)

I. Limitations of WFS

All WFS systems are restricted to horizontal reproduction, but the Wave Field Synthesis principle is not limited to a plane. In principle, the procedure would be able to restore the sound field in all three room dimensions. Our detection in azimuth mainly works through time detection, which becomes reconstructed perfectly by the horizontal loudspeaker lines, but adding a third dimension would practically mean populating the walls in a room with loudspeakers. This would mean extremely high costs for the hardware necessary. Other techniques of reproducing 3d sounds, like Ambisonics of VBAP have shown real 3D sounds with a much smaller number of loudspeakers.

The Kirchhoff- Helmholtz integral describes an unlimited amount of elementary waves. In practice though, the number of loudspeakers is limited. As with any quantization, this causes aliasing effects. Inside the playback area, depending on the wavelength, across the room points

of higher level alternate with points of a lack in magnitude. At one dedicated point, the notches and hills have a very small bandwidth. Fortunately, such effects are less disturbing in perception (Oellers, 2013). For aliasing-free reproduction, a loudspeaker distance of less than one inch would be needed (Germany Patent No. 102009006762A1, 2009). Seemingly random spacing at defined positions would reduce aliasing, in the same manner as a wheel in any 24fps Western film no longer appear to spin backwards, if the spikes are aligned in randomized angles (Spoer, 2004).

When it comes to the loudspeaker arrangement, the fact that it is not being completely closed around the listener, the end of the radiating surface causes the truncation effect (Oellers, Wave Field Synthesis, 2013). At the end of the arrays, no further elementary waves contribute towards sound pressure. That will change the resulting superposition suddenly and a shadow wave arises. To a certain extent, this effect is avoided by decreasing the level of the outer speakers. As long as the virtual acoustic source aligns behind the loudspeakers, the shadow wave arrives at the listener later than the direct wave front. However, if the shadow wave arrives in front of the actual wave front, this is audible and disturbing to the listener (Oellers, Wave Field Synthesis, 2013).

The Wavefield synthesis principle works because it reproduces a sound field, instead of a particular sound. In order to experience the synthetic sound field only, the environment where the system is set up should not contribute to the wave front at all. In other words, the environment needs to be acoustically insolated, and the walls should not reflect any of the sounds produced by the loudspeakers. This means that the WFS system needs to be set up in an anechoic chamber, greatly increasing the costs of such a system. This aspect is mostly neglected due to practical reasons, thus the sound fields experienced will not be perceived exactly as they are synthesized.

C. Other Sound Field Synthesis Techniques

Besides Wavefield Synthesis, there are other methods of re-creating sound fields, each promising different features and constrains.

A significant step forward towards high fidelity spatial sound reproduction was made with the introduction of “Ambisonic” in the early 1970s (Ambisonics, 2015). The underlying principle for this concept was to recreate an exact sound field at a recording position. Special microphone arrays have to be used, consisting of coincident pairs of microphones that measure acoustic pressure gradients in various dimensions. More complex arrays used result in high spatial fidelity extending in larger space. And this is the main limitation of this approach – although the reproduced sound field is accurate, provided the playback system matches the recording array (or B-format), the sweet spot is rather small. For feasible loudspeaker setups the size of the artifact-free reproduction area is typically smaller than a human head at the upper end of the audible frequency range. Outside, spatial sampling artifacts arise that may be perceived as coloration of the desired sound field (Geier & Spors, 2012).

Derived from a technique called “Ambisonics Amplitude Panning” is a simple panning method entitled Vector Base Amplitude Panning (VBAP). As the name implies only amplitude panning is used, and it is used so that pairs of adjacent loudspeakers create the illusion of virtual source. For 3D reproduction, the virtual source is created between a loudspeakers triple (Geier & Spors, 2012). Using VBAP it is possible to create two- or three-dimensional sound fields where any number of loudspeakers can be placed arbitrarily. The method produces virtual sound sources that are as sharp as is possible with current loudspeaker configuration and amplitude panning methods (Pulkki, 2015)

D. Software for Wavefield Synthesis

I. SoundScape Renderer (SSR)

One of the most common tool for reproducing 3D audio is SoundSCape Renderer, developed at the T.U. Berlin and Rostock University. SSR is a versatile tool for real time spatial audio reproduction, implementing a variety of headphone- and loudspeaker-based methods like: WFS, VBAP, Ambisonics Amplitude Panning, Near-field-corrected Higher-Order Ambisonics, dynamic Binaural Synthesis, dynamic Binaural room Synthesis (BRS), as well as generic 3D audio renderer. The SSR is free software licensed under the GNU General Public License. It uses the JACK audio framework and is currently available for Linux and Mac OS X. Interaction with the program is possible using the built-in graphical user interface and via a network interface. An example for such interface is their Android SSR client, as well as a SSR remote for Max for Live. The built in GUI shows the current position of all sound sources and can be used to move them around, set their volume, their source model and other parameters. Depending on the current rendering algorithm, the GUI also shows either a head or the loudspeaker setup (figure 4). Both can be rotated and moved around. If not needed, the GUI can also be disabled. The same GUI is used for all rendering algorithms.

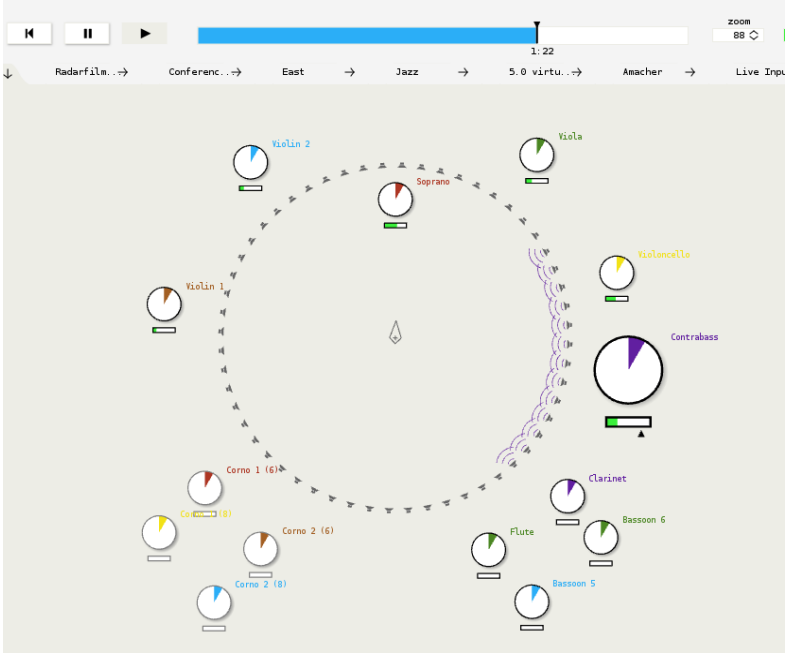

Figure 4 - SSR - WFS GUI

A great thing about SSR is that all its libraries are available as C++ libraries which can be used to implement plugins for various host programs. This way the SSR could be integrated in any audio processing software, with a little coding effort.

II. WFS Collider

WFS Collider is a tool used for wave field synthesis. It was developed by Arthur Sauer and Wouter Snoei at The Game of Life Foundation (Sauer & Snoei, 2015). It is effectively a standalone GUI and library for Super Collider, focused on WFS. It is inspired by the Digital Audio Workstation look, with a timeline, multi-track setup and busses. This also means that audio content can be organized into subgroups, making the workflow easier. WFS Collider features a plethora of effects and allows for great customization of a sound. Some examples are: filtering, convolution reverberation, loop-backing, Doppler negation, etc. The effects and parameters are organized in the popular "signal chain" manner, allowing for cascading effects and features (figure 5). Since it is based on Super Collider, every effect code can be inspected and modified as needed, providing great help for understanding the mechanics of a certain propriety.

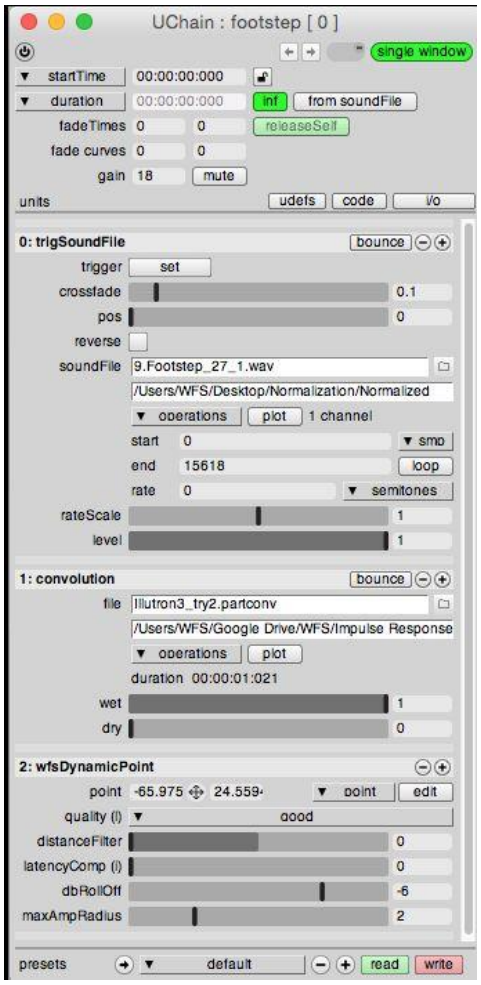

Figure 5 WFS Collider signal chain

A great feature of WFS Collider is that it can work with vectorial trajectories for its objects, meaning one can use any graphics software to control the behavior of the virtual sound sources.

All the parameters in WFS Collider can be externally controlled through Open Sound Control (OSC) protocol. This allows the audio engine to work as middleware for other applications, like game engines (Paisa, Banas, Vogiatzoglou, Serafin, & Grani, 2016). Because of the timeline approach, WFS Collider can feel a little more focused on a linear pre-set performance, rather than an interactive one, but with that does not mean that real-time interaction is hard to implement.

III. Other software

IOSONO GmbH is a leading company in the field of spatial audio, providing both software and hardware implementation of various 3D audio techniques, including WFS.

Wave 1 by Sonic Emotion is another commercial implementation of WFS principles in the form of a software and hardware package. Just as IOSONO`s implementation, it is not only focused on WFS, but also binaural, Ambisonics and Dolby Atmos.

Due to the commercial nature of these products, the information regarding the technical details is limited.

E. Perception of 3d sound

Human hearing has a very important role in our everyday orientation in space. Sounds usually convey information about the source and its location, and we have evolved to "decode" this information. The sound wave generated by an external source is diffracted by its interactions with the head and external ears. The resulting changes in temporal and intensity characteristics of the sound provide cues about the localization of the sound source (Middlebrooks & Green, 1991). In order to recreate a credible experience the psychoacoustic proprieties of sound need to be integrated in the reproduction algorithm.

A number of different properties of physical stimulation have been thought to be potential cues to the perception of auditory distance. These cues may be generally divided between those that require only one ear(monaural) for information transmission, and those that require two ears(binaural) (Zahorik, 1996). The present description of these cues will therefore be classified on the basis of either monaural or binaural requirements. Following this description, a brief discussion of cue utility to the listener will be offered.

I. Monaural cues

Monaural distance cues are hints that contain information about the distance of an object to the listener and do not analyze the signal difference between the ears. There are three popular monaural cues that will be described: sound intensity, spectral shape and the direct-to-reverberation ratio.

a) Intensity

In an acoustic free environment (with no objects to reflect sound) a stationary sound source (point-source) intensity is related to the distance from its physical location to the listener. This relation is described by an inverse square law (Zahorik, 1996). That is to say that intensity is related to distance, R, from source to listener by a factor of $1/R^2$. Since sound pressure is proportional to the square root of intensity, pressure may be said to obey a $1/R$ law with varying distance, R. In any case, the following simple rule is yielded: There is a 6 dB loss in sound pressure for each doubling of distance from the sound source. (Zahorik, 1996), and it could decrease to 4.25 dB loss in reverberant environments (e.g. a normal room) (Zahorik & Winghtman, 2001).

This law only holds for acoustic free environments. Unfortunately these environments are more theoretical than practical, being very hard to realize an environment that has no impact on the sound. Thus, the intensity cue becomes a poor approximation, especially in reverberant-rich environments. Another issue the intensity cue is it`s relativity to the listener`s position, thus reliant on the source's amplitude. If the amplitude is familiar to the listener, it could serve as an absolute distance cue, otherwise it is almost irrelevant, and other cues have a bigger influence in understanding distance.

b) Spectral shape

The main medium of travel for sound is the air. The physical proprieties of the air manipulate the sound in a predictable way, acting as a low pass filter, thus the sound source spectrum at the

listener's position varies as a function of distance. At distances greater than 15 meters, the sound absorbing proprieties of air modify the sound source spectrum significantly (Zahorik, 1996). The main propriety of air attenuating high frequencies is it`s humidity. At 40% relative humidity, one can expect an air absorption of 3-4 dB per 100 meters at 4 kHz (Zahorik, 1996).

At shorter distances (under 15 meters), when sound is considered to be in the acoustic near-field, filtering applied by the head and ear physiology may not be considered independent from the sound source, and it varies in a complex fashion as a function of distance.

The spectral shape cue is a relative one as well, requiring the listener to be familiar with the sound beforehand, in order to understand what frequencies are missing. On top of that, in order to be a relevant cue, distances of more than 15 meters is required between the listener and the sound source.

An interesting application for this cue can be found in foghorns. Foghorns are audible signals that are used to provide audible warning to ship, regarding rock outcrops, shoals, headlands, or other danger zones. These devices are turned on when the visual aids like lighthouses are obscured by fog. Sailors are trained to understand the sound filtering properties of air, to "decode" the distance to the dance zone.

c) Reverberation

Reverberation has also been shown to be important, as has been recognized for many years by scientists researching perception as well as acousticians. In environments with sound reflecting surfaces, the ratio of energy reaching a listener directly (without contact with reflecting surfaces) to energy reaching the listener after reflecting surface contact (reverberant energy) varies systematically with distance (Zahorik & Winghtman, Loudness constancy with varying sound source distance., 2001). In general, as sound sources move away from an observer in a reverberant environment, the proportion of sound energy directly reaching the observer's ears decreases, while the proportion reaching the observer's ears after reflection (and delay) from surrounding surfaces increases (Mershon & King, 1975). This can be called the direct-to-reverberant ratio.

In rooms, change in direct-to-reverberant energy ratio is primarily due to the effect of the $1/R$ law on the direct (first arriving) portion of the sound field, since the energy in the later arriving reflected portion of the sound field is relatively constant for varying source distance.

The direct-to-reverb ratio is a particularly interesting cue because it is, in theory, absolute; it does not depend on source intensity, or the familiarity of the listener with the source (Bronkhorst, 1999). It is able to code a wide range of distances in many reverberant environments, and the user can "learn" acoustical environments to improve the distance perception. The cue is also of interest because it has apparent limitations. It has been shown repeatedly that the perceived distance of a sound source in a room is compressed; it increases virtually linearly with source distance at short range, but converges to a certain limit when the source distance is increased

further. This limit acts as a sort of “auditory horizon”, which is, however, not constant, but depends on the acoustic environment (Bronkhorst, 1999).

II. Binaural Cues

When sound sources are in the near-field, binaural differences in both intensity and time are in many circumstances no longer independent of distance, as they are for planar waves. These differences are called inter-aural time difference (ITD) and inter-aural level difference (ILD) (figure 6), and are usually researched when discussing sound (radial) localization.

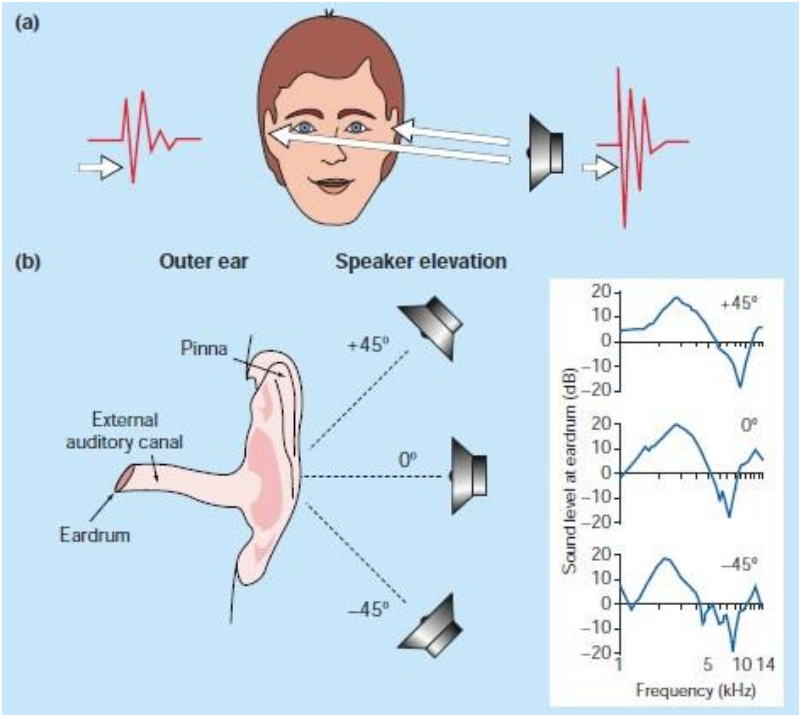

Figure 6 - (a) ITD & ILD, (b) Spectral filtering of the outer ear. (Moore & King, 1999)

Due to acoustic parallax, they are maximal along the inter-aural axis that is directly opposite to either the left or right ear. In a research conducted in 1921, Hartley and Fry (Hartley & Fry, 1921) conclude that the inter-aural intensity difference for a pure tone source(1860 Hz) on the inter-aural axis can differ for distances between 87.5 and 17.5 cm by as much as 20dB. Their study is pure theoretical, and the values are derived assuming a spherical head.

Another binaural cue referred as motion-induced intensity rate of chance can be observed when the listen is changing his/hers position in relation to the sound source. This method of understanding distance shows better results when compared directly to a static listener

III. Summary

It is well known that our ability to perceive distances of sound sources depends on several different cues. Because both the availability and reliability of these cues are dependent on the given acoustical situation, the auditory system likely combines information from multiple cues to produces stable distance percepts. Cues that have been studied are sound intensity, spectral shape, the direct-to-reverberant energy ratio, and inter-aural differences. Not all cues are equally effective in all circumstances. Inter-aural differences, for example, only show a clear dependence on distance when the source is not further away than about 1 m. The direct-to-reverb ratio is, evidently, only relevant when the environment contains reflecting surfaces. Intensity has the drawback that it is a relative cue: it can only be interpreted correctly when the listener has priori

knowledge of the source intensity. In addition, the intensity cue is less useful in the indirect field of a sound source that is placed in a reverberant environment – it is then determined by the, virtually constant, level of the reflections, and not by the, distant-dependent, level of the direct sound.

F. Developing a Max/MSP object

Max/MSP is a visual programing environment designed for real time processing. It is widely used in the performance arts scene, music production, artistic installations as well as a powerful sound design tool. The workflow is can be compared with a modular synthesizer, since it is based on connecting object that execute various tasks (mathematical operators, audio processing, networking, etc.).

Figure 7 A suite of Max/MSP objects

A Max/MSP external is a small piece of code loaded into Max and used as a template to create objects on demand. In its code, an external consists of a set of C functions that perform signal processing and define the external's behavior in response to its messages. These objects are used as building blocks inside the programing software. There are many pre-loaded object, to suit various tasks (figure 7). All object in Max/MSP have at least one inlet and/or outlet connection. By default, every object shows one inlet. Additional inlets appear to the right of the default inlet, with the rightmost inlet being created last (Cycling '74, 2011). Inlets are essentially message translators while outlets define connections between objects and are used to send messages from one object to the objects to which it is connected. What is not obvious about an outlet however, is that when a number is sent out, the outlet-sending function does not return until all computation "below" the outlet has completed (Cycling '74, 2011). This stack-based execution model is best illustrated by observing a patch with the Max debugger window.

There are two general categories of inlets/outlets one object can have: non-signals (integer, float, message, bang, etc.) and signals. There are a few fundamental differences between these two types, which have great impact in the performance of the object. First and foremost, the non-signal input is executing its task only when it receives any input in a hot inlet. A hot inlet, as opposite to a cold inlet is the one triggering the execution of the code in the object. When it comes to a signal inlet, the object executes its task continuously, as long as the DSP audio engine is turned on.

There is another subtle difference between non-signal and signal objects. A non-signal object responds as soon as it receives a message, sending data to its outlets as fast as possible. A signal object stores up a certain number of samples in a floating-point array called a signal vector, and then passes all of the samples in that signal vector on to the next signal object. In Max/MSP the

size of the signal vector—called the signal vector size—may be set globally in the DSP status window. The signal vector size determines the minimum latency of the audio system as a whole. As such the signal vector size is a system parameter that involves an important trade-off. On the one hand, larger signal vector sizes are more CPU-efficient than small ones since more samples are processed on each vector, reducing the overhead of calling the DSP loop. Setting the signal vector size to an arbitrary low two samples will noticeably increase the CPU demand of most patches involving audio objects. On the other hand, large signal vector sizes can make the system feel sluggish since messages are sent only once per vector. Even worse, the signal vector size determines the lower limit for delay with feedback, because a sample must cycle through an entire vector in one object before being fed back to an object higher up in the DSP chain (Lyon, 2011).

Any Max/MSP external follows a common internal structure. The anatomy of a Max object has several mandatory components and tasks (Cycling '74, 2011):

- Header files that include the headers required by Max/MSP, “usually ext.h” and “ext_obex.h” as well as “z_dsp.h” when writing an MSP external. These header files provide definitions for most functions and data types used in the Max/MSP libraries. These files and libraries are found in the Max/MSP SDK, freely available on the Cycling ’74 website.
- The definition of the object structure. The first component is a of the structure is a “obj” “t_object” or “t_pxobject”. This is a proxy that containts “t_object” as well as a few other components that describe the state of any particular instance of the object, like which of the inlets or outlets are connected. The “t_object” structure itself contains all inlets and outlets of an object, along with a list of its messages and methods. The use of proxies enables Max/MSP audio objects to receive both signals and floats at the same inlet (Lyon, 2011). After creating the “obj” any other data types can be created. The structure created will be used in the prototypes to whatever functions are to be used in the external, so it’s compulsory that the structure declaration is placed above these prototypes.
- Function prototypes are declarations for the functions used in the code. The prototype provides the name, parameters, and return type of the function. The prototype for a function may be read by inspection from its first line. It may seem redundant to provide these prototypes, but the compiler uses them to check if the use of the function corresponds to how it is defined. Thus prototypes are useful for avoiding such bugs as providing arguments of the wrong type or number to the functions. Prototypes are required for almost all functions used in Max external code.
- An initialization routine called “main()” is called when Max/MSP loads the object for the first time. In the initializations routine are defined one or more class consisting of the following: telling Max about the size of the object's structure and how to create and destroy an instance, defining methods that implement the object's behavior (e.g. binding a function to a specific type of input like int. or bang), in some cases defining attributes

that describe the object's data, registering the class in a name space (Cycling '74, 2011). This is a good place to print the authorship information. Unlike most of the functions we will write for Max externals, “main()” does not require a function prototype. C functions are assumed by the compiler to return type int. unless another return type is specified. The “main()” function is required to return an int. Since there are no arguments specified for “main()”, it is permissible to omit its function prototype (Lyon, 2011).

- Unlike “main()” function that is called only the first time an external is loaded, any object needs to have an new instance routine that is called every time a new instance of the object is loaded. This allocates the memory to create the necessary instance, and initializes it. In this step the necessary inlets and outlets are declared.
- All the previous steps are mostly setting up the necessary platform to execute the code in the perform routine. In this routine the behavior of the object is written. In contrast to the Max convention in which non-signal inlets and outlets are numbered from right to left, in this function the inlets are called from left to right. The order in which these signal vectors are passed is determined in the DSP method (Lyon, 2011).
- Last step required to have a functional Max/MSP external is to connect the external to the DSP chain. This method is called whenever the audio processing is turned on, or whenever the DSP chain is rebuilt, such as by adding a new audio object to the Max/MSP patch. This DSP method consists solely of a call to the Max/MSP function “dsp_add()”. The first argument to “dsp_add()” is the name of the external’s perform routine. The perform routine calculates one signal vector worth of samples for the external. The second argument tells “dsp_add()” how many more arguments to expect. The next argument, is a pointer to the object. The next arguments are pointers to the object’s inlets and outlets. The function “dsp_add()” does not distinguish between inlets and outlets. They are all just arrays that will be passed to the perform routine. The final argument is the number of samples to be processed during each call to the perform routine. This number is identical to the signal vector size as defined in the Max/MSP DSP status window.
- There is one type of functions that could/should be implemented in any Max/MSP external called message handlers. These functions define the actual behavior for an object by writing C functions that will be called when our object is sent messages of a particular type. A separate message handler needs to be implemented for each type of expected input (e.g. int., bang, signal, etc.). As mentioned before, these functions are bounded to their expected input type in the initialization routine described above.

4. Design

In the early stages of the project, the general concept of the external had to be defined, in order to facilitate the development. The most important things to be clarified are the desired behavior of external object, referred from now as “[wfs~]”. This can be split in two distinct areas, based

on the two roles one can have: creator and listener/audience, but first setting the object`s functionalities should be covered.

A. Functionalities

The [wfs~] object should be able to synthesize wave fields, based on the coordinates of a given sound, as well as the characteristics of the speaker system. Considering every WFS system is different, and there is no standard for setting up one, the object should accommodate arrays with any number of speakers, speaker dimensions and distance between speakers. When it comes to speaker arrangement, the two most popular setups are circular and rectangular, and the [wfs~] should be able to synthesize wave fronts for both types. A particular case for the rectangular setup is the truncation effect mentioned in the Analysis chapter (page). This effects occurs when there is a big difference in pressure in the ends of the array, as the red area in figure 8 shows. This effect can be diminished by lowering the output amplitude of the first/last few speakers in the array. A logarithmic curve is usually implemented.

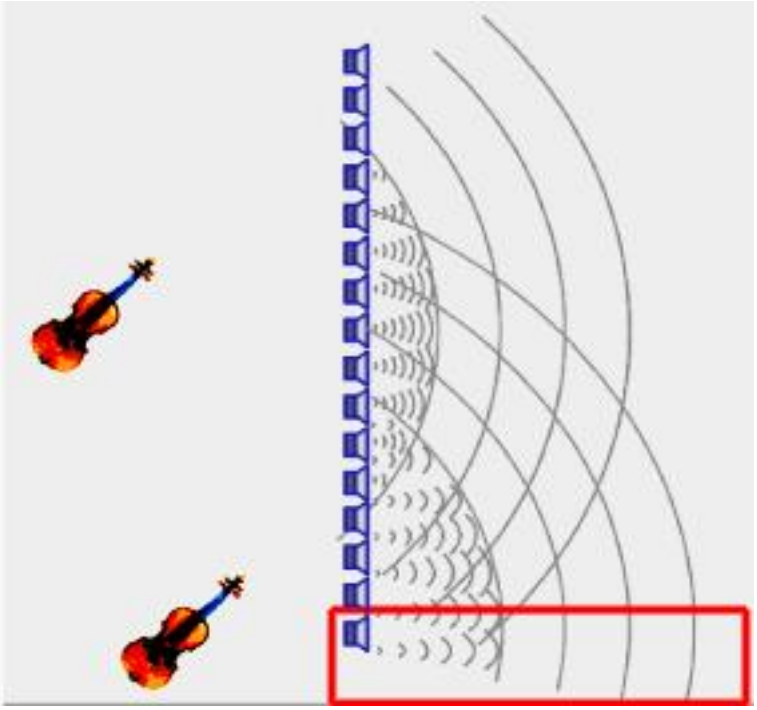

Figure 8 Truncation effect in the end of the array

Usual Wavefield synthesis software can produce 3 types of virtual sources, point sources, focused sources and plane waves, and the [wfs~] should be able to do this as well. A certain problem represents the transition between a point source to a focused source, in other words moving a virtual sound source in front the speaker array, from a position that was behind. This is because the delay times become inverted at the transition point, creating an audible click. A popular approach is to interpolate the delay times, keeping them at 0 on the crossing point. Another solution is to crossfade between the two signals, solution that should be implemented in [wfs~] as well.

An inspiration for the behavior of [wfs~] is the [ezdac~] object, that works as the audio output of the stereo bus. From personal observation, having a big number of [ezdac~] object in one`s patch does not have any effect on the performance of said application. This can mean that the object does not execute it`s functionalities for each instance, but instead sums them up and executes only once Max/MSP instance. This should optimize the performance necessary to run complex

patches. In the same manner the [wfs~] should not execute all the heavy calculation for each instance of the object, but instead sum all the signals send to it and execute once/sample.

B. Experiencing [wfs~]

Starting with the listening/audience position, experiencing sounds played through the [wfs~] object, should be as engaging and accurate as possible for the given setup. In order to achieve this, the object should not introduce unwanted latency, or undesired audible sounds, like moving artefacts, in the form clicks, pops, crackles, etc. On top of that, preserving the audio quality of the input signal is a priority.

When it comes to the perception of virtual sound sources and their position, [wfs~] should perform as good as other, stand alone, WFS software. In order to achieve this, the external is expected to replicate the behavior of sound influencing the localization task, specifically an accurate sound attenuation over distance, and an accurate filtering, simulating the effect of air absorption of high frequencies.

C. Working with [wfs~]

If approaching the external from a Max/MSP developing position, the [wfs~] object should stay true to the Max philosophy of modular visual programing, where each object does has one task. To facilitate the usage of multiple instances of the object, without having to re-introduce the system settings (nr of speakers, dimensions, offsets, etc.) every time, another object could be implemented that behaves as a global setup object for the [wfs~] external. A similar approach can be found in the [transport] (that output bangs the current beat, bar, unit, BPM, and time signature), and [global transport] (figure 9) object which controls the settings for all [transport] objects. This allows the user to set up the parameter for a musical piece only once, greatly facilitating the music production process.

Figure 9 Global Transport object

As with all other Max/MSP object one can send multiple types of data in its inlets (integers, floats, messages, bangs, signals, etc.), and the [wfs~] should have flexible inputs to allow for using same inlet with different data types(e.g. interchange between floats and integers).

Lastly, in order to facilitate a good experience when working with the object, it should be able to detect when an illegal action is committed and protect against it, in order to avoid crashing the application. A good example for this would be a scenario when the user accidentally inputs a negative array size. If no protection is implemented this can easily crash Max/MSP by creating negative delays.

5. Implementation

The [wfs~] object, was written in C programing language, using xCode IDE version 8, and it was built under OSX Yosemite and El Capitan. The Max 6 API as well as Max/MSP/Jitter SDK were used extensively for building this object, since all functions required to create any Max objects are available in some of the libraries provided.

The object was written using functions working with 32 bits resolution, mainly due to the amount of literature available for 32 bits libraries as opposite to 64 bits. On top of that, by using the 32 bits libraries, backward compatibility with older versions of Max/MSP is easily achieved. The object was tested on Max 6 and Max 5, and it proved to work without any difference.

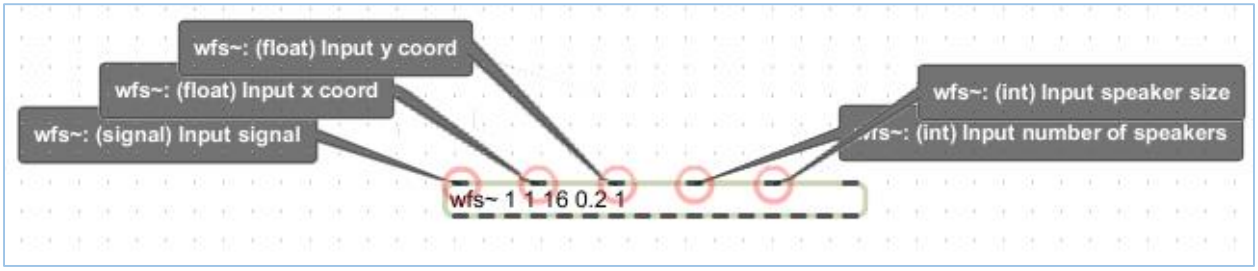

Figure 10 the [WFS~] object

A. Limitation

Working on this project challenged me to approach new systems that only increased the necessary time required to implement a fully functional Max/MSP object as described in the Design chapter. Therefore some aspects of the [wfs~] had to be prioritized. The first version of the object is hard-written to synthesize wave fronts through the speaker system available in the Aalborg University campus in Copenhagen, which features 4 arrays of 16 speakers, arranged in a square. This was necessarily because implementing a parametrical generation of MSP outlets is a very time consuming task, which only improves the workflow, but has no impact on the performance. This means that the [dac~ x] (x = speaker number) object was used to output sounds, having an individual [dac~] for each output channel (figure 11). Implementing two objects, one for control and one for synthesis was another feature that had a lower priority, and it is not implemented at the moment of writing. In order to test the object in time, only point sources were fully functional.

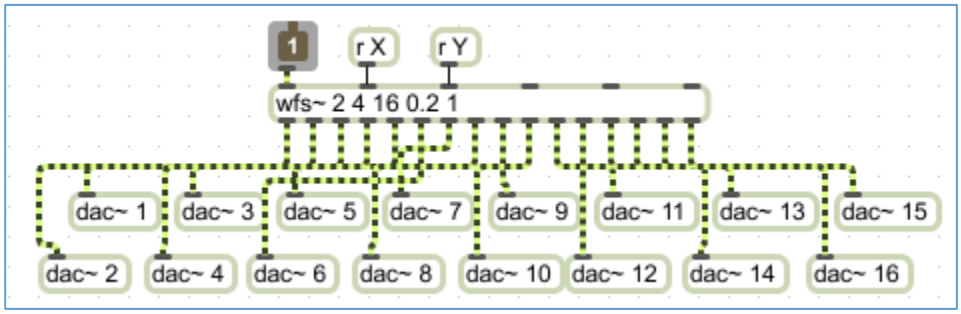

Figure 11 the [wfs~] object in use

B. Object setup

Every Max/MSP external follows is built on a common skeleton made of four functions and an object structure. Three of the functions are not called by messages and are known as routines: initialization routine, new instance routine, and perform routine. The last function, called the "dsp method" is called by the "dsp" message and thus is not referred as a routine. Next paragraphs will try to shed some light of what each of these components does in the case of [wfs~].

I. The object structure

The first component for all Max/MSP externals is the object structure. The very first component defined must be a "*t_pxobject"* type one. Next are any components that must hold their value for longer than one signal vector worth of samples. These components are often called state variables, since they maintain information about the state of the object. The [wfs~] has 20 state variables that account for sample rate, delay length, state of the inlets, etc.

II. The Initialization routine (main())

As with other software written in C (or C++ for that matter), the function prototypes need to be declared, so they can be called before they are actually written in the top-bottom code layout. An exception to this rule is the "main()" function, which does not need to have the prototype declared, because it can only return an integer, and C assumes by default that a function returns an int. Besides this the "main()" function does not take any arguments, and it is called only the first time the object is first loaded.

The main function, referred as the initialization routine, defines the object class (called "wfs_class"), gives it a name, indicate what methods to call when a new instance of the object is created or deleted and allocate some memory for the object struct. After the class is defined, there are several functions bound to specific types of inputs. This is a very important aspect, one can easily implement a plethora of functionalities with a small number of inputs (e.g. same inlet accepts signals and messages). This is a common practice in the Max/MSP environment. Binding is done by calling the "class_addmethod()"method, passing arguments like, what to bind, what method call when a certain input in received, type of input, etc. The [wfs~] binds four methods to the class, for the DSP(called when the DAC`s are turned on), for float inputs, integer inputs as well as a special class case called "assist" that is called when hovering the mouse cursors over a certain inlet(figure 12).

```
/* Bind the assist method, which is called on mouse-overs to inlets and outlets */
class_addmethod(wfs_class, (method)wfs_assist, "assist", A_CANT, 0);
```

Figure 12 - Binding the assist method

III. New Instance Routine

Whenever a new instance of the [wfs~] object created, the new instance routine is performed. For [wfs~] the function called "wfs_new() " is responsible for allocating memory, for creating inlets and outlets, reading user inputs and creating the delay line.

To create new inlets it is necessary to call the “dsp_setup() ” method, passing an argument pointing to the object struct, and the number of desired inlets. It is worth mentioning that these inlets are generic, and will accept connection for any type of input. The functions bound to specific input types, declared in the initialization routine, dictate what will happen with the different type of inputs. The [wfs~] has 6 inlets: one for the audio signal, two for coordinates(X, Y), number of speakers, speaker dimensions (in meters), and type of source (plane wave, point source, focused source).

Creating outlets is easier since it is done by calling the “outlet_new() ” function as many times as necessary. For the [wfs~] the function is in a loop calling it 16 times. Each outlet will represent an audio channel, and will be connected to a [dac~ x], where x is the channel number. A thing to remember that can cause a lot of issues, is that the outlet numbering goes from right to left, right being the first outlet.

After the inlets are created their arguments can be read by calling the “atom_erg_getfloat()”(for float inlets), with the indices of float as argument(e.g. first float encountered, second, third, etc.). This function alters the receiving variable only if an argument was found. If the user types in [wfs~] with no arguments, then no change would be made to the values sent (e.g. X, Y coordinates). The correct way to call this object and receive inlets is [wfs~ 1 1 16 0.2 1]. To avoid crashing max by pointing to a non-existing address somewhere in the program, default values are written in the new instance routine, in case one does not write the arguments in the object creation as indicated above.

Before the object instantiation process is completed, it is a good practice to enforce that inlets and outlets do not share the same memory. The object struct component “z_misc” is a bitmask that maintains the state of several flags. The flag to prevent memory sharing is set with the C bitwise or operator”|=”, which turns on the” Z_NO_INPLACE” bit of the “z_misc” bitmap (figure 13). With that flag set, audio vectors will not share memory (Lyon, 2011).

```
x->obj.z_misc |= Z_NO_INPLACE;  // force indepented signal vectors
x->sr = sys_getsr();  // find Sample rate
```

Figure 13 Independent signal vectors and SR acquisition

During the initialization stage, memory for the delay line is allocated, but this will be covered thoroughly in the chapter explaining the delay mechanism.

IV. The perform routine

The perform routine, called “wfs_perform()” is the function that executes the actual operations for any Max/MSP object. In the case of [wfs~], it delays the input signal based on the parameters send to the object. Before that it assigns variables from the object structure into local variables, a procedure called de-referencing. This operation is an efficiency measure, since it is more efficient to use local variables than to pull them off of the object structure each time they are needed, especially if they will be used repeatedly in a loop. In addition, it makes the code easier to read. However, since the operation is now performed with local variables, any changes to

these variables that need to be saved must be reassigned back into the object structure at the end of the perform routine.

A very important task that has to be performed in this routine is pulling the object structure, signal vectors, and signal vector size(figure 14) from the integer array passed as argument for the "wfs_perform()" function(in this case it is simply called w). It is important to know that the first position in the array is the object pointer, followed by the signal vector pointer, and lastly is the position indicating the buffer size (a.k.a. block size) for the current Max patch. It is crucial to get the order right since it will most likely cause a crash if passed to the DSP method in the wrong order. Lastly Max does not distinguish between inlets and outlets and treats them all like identical signal vectors.

```
/* Copy signal vector pointers */
t_float *input = (t_float *) (w[2]);
t_float *x_coord = (t_float *) (w[3]);
t_float *y_coord = (t_float *) (w[4]);
t_float *nr_speakers = (t_float *) (w[5]);
t_float *spk_size = (t_float *) (w[6]);
t_float *arr_shape = (t_float *) (w[7]);
t_float *out_1 = (t_float *) (w[8]);
t_float *out_2 = (t_float *) (w[9]);
```

Figure 14 Copying signal vector pointers

Inside the "*perform routine"* lies the code responsible for the actual signal processing in the object, called the DSP loop. This is performed while there is incoming sample available and it consists of applying the necessary delay and level attenuation for each individual channel. This is performed in a loop for every single sample processed. Inside this loop the linear interpolation is calculated for each necessary delay, as explained in the chapter covering the delay. The formulas for amplitude and delay length will be described in a separate sub-chapter.

After every output sample has been delayed and attenuated it is assigned to its respective output. When this task is completed the write index and inputs index are incremented to process the next sample.

The very last step in the DSP method is to return a pointer to the next entry address in the DSP chain.

V. The DSP method

The DSP method is bound to be called whenever the DAC object is turned on or whenever the DPS chain is rebuilt. Max/MSP builds the DSP chain whenever the DACs are turned on, or when any change is made to the DSP chain, such as adding or removing signal connections, changing the sampling rate, or changing the signal vector size.

One of the tasks executed in this method for the [wfs~] object is to adjust to changes in the sampling rate. As figure (figure 13) shows, the sampling rate is determined in the new instance routine, but it is a good idea to check again. This assures that if the sampling rate has changed since the object was instantiated, it will adapt to the new sampling rate. If the sampling rate has in fact changed, it is needed to reallocate memory for the delay line. The key test is to compare the object's stored sampling rate to the sampling rate found on one of the signal vectors (which is guaranteed to be the most up-to-date value). If a discrepancy is found, the calculation of memory is redone. However, rather than allocating memory from scratch as done in the new

instance routine, here the memory is resized by using the function “sysmem_resizeptr() “. In the case where the memory is not available an error message is printed and return from the function, without adding the object to the DSP chain.

The main purpose of the DSP method is to add the external to the DSP chain. This is done solely by calling the Max/MSP function “dsp_add()”.The first argument is the name of the external’s perform routine. In this case it is “wfs_perform()”. The perform routine calculates one signal vector worth of samples for the external. The second argument tells how many more arguments to expect. This number represent all the inlets and outlets, as well as the block size. The function “dsp_add()” does not distinguish between inlets and outlets. They are all just arrays that will be passed to the perform routine. The final argument is the number of samples to be processed during each call to the perform routine. This number is identical to the signal vector size as defined in the Max/MSP DSP status window.

```
dsp_add(wfs_perform, 24, x,  sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec, sp[3]->s_vec, sp[4]->s_vec,
sp[5]->s_vec,sp[6]->s_vec,sp[7]->s_vec,sp[8]->s_vec,sp[9]->s_vec,sp[10]->s_vec, sp[11]->s_vec,
sp[12]->s_vec, sp[13]->s_vec, sp[14]->s_vec, sp[15]->s_vec,sp[16]->s_vec,sp[17]->s_vec,sp[18]->s_vec,
sp[19]->s_vec, sp[20]->s_vec,sp[21]->s_vec,sp[0]->s_n);
```

Figure 15 adding the object to the DSP chain

C. Other management functions

Besides the mandatory function mentioned above, the [wfs~] has three methods that are designed to aid the user in its regular usage of the object. The assist method is creating pop-ups with information about each inlet or outlet whenever the user hovers the mouse over an inlet/outlet.

A pair of functions, which are called when there is a received an integer of float message respectively, is ensuring that the user will not accidentally crash the program. Figure 16 is showing a particular case, when the 4th inlet, responsible for the speaker size, is enforcing a positive speaker size.

```
case 4:
    if(f < 0.0)
    {
        error("wtf~:Speaker size cannot be negative, restoring to default(0.2) ", f);
        f=0.2;
    }
```

Figure 16 enforcing a positive speaker size

D. Delay

The key element in Wavefield synthesis is a delay determined by the position of the virtual source to each speaker that will eventually playback the incoming signal. A delay is simply an array of stored samples. The lengths of the array in samples is the duration in seconds multiplied by the sampling rate. For example at a sample rate of 44100 a one second delay line will contain 44100 samples. Unfortunately long delays can use extensive amount of resources so is it is mandatory practice to implement dynamic memory allocation, since Max imposes a size limit of 32 kilobytes

for its object structure, so a static memory allocation would not fit. Max/MSP provides a pair of memory functions called “sysmem_newptr()” and “sysmem_freeptr()” that allocate memory and free it respectively. In order to allocate memory, first it is necessary to declare how much memory is necessary. Once the required amount of memory has been allocated to a pointer that points to a block of memory of the appropriate size, it can be treated like a regular array, of statically allocated memory. To insure that [wfs~] will perform decent on slower computer as well, a maximum delay time of 500ms was implemented. While 500ms sounds like not much time for a maximum value, considering the speed of sounds of aprox. 344 m/s, 500ms will be exactly 172, indicating the maximum distance a virtual source can have from the speaker array. The maximum delay length was use as a reference for dynamically allocation of memory by converting from milliseconds to samples, and using that value to find out the necessary size in bytes for the 500ms. The function “sizeof (float)” comes in very handy for this operation. The last step is to make call the “sysmem_newptr()” with the desired amount in bytes to allocate(figure 17) . Once memory has been allocated using the delay line splits into two separate operations: reading and writing.

```
x->delay_length = x->sr * x->maximum_delay_time + 1;
x->delay_bytes = x->delay_length * sizeof(float);
x->delay_line = (float *) sysmem_newptrclear(x->delay_bytes);
if(x->delay_line == NULL){
    error("vdelay~: cannot allocate %d bytes of memory", x->delay_bytes);
    return NULL;
}
```

Figure 17 Memory allocation

To make good use of the limited memory available for the delay line I implemented a circular buffer that works by starting from the beginning of the array and write each new input sample into the next available location in the delay line. When the last available sample position has been written, the process repeats from the beginning and keeps on writing. In order to always know where to write the next sample, the current location in the delay line needs to be stored.

Reading from the delay line is done by converting the desired delay in milliseconds to samples at the sample rate. Next step is to read the delay line “delay” samples before the current position. A problem that occurs with this approach is when the position in the delay line is smaller than the delay length in samples, thus reading a negative position in the delay line array, and crash Max. To protect against this, the dimension of the delay line is added to the read position, continuing circularly in the delay line (figure 18).

```
read_index = write_index-idelay;
while(read_index < 0){
    read_index += delay_length;
```

Figure 18 Reading the circular buffer

Another problem encountered when reading from the delay line is that in some cases the read number is not integer. For example if a desired delay is 75 milliseconds, the conversion looks like this: 75 * 44100/1000 = 3307.5, according to the method previously described. There are multiple solutions for this problem. One would be averaging to the next sample in the delay line, truncating it or interpolate between the two sample values. By ignoring the fraction in the sample position, distortion is introduced, especially when the delay length is variable as it would be for a moving virtual source. The [wfs~] object is implementing a linear interpolation to obtain a reasonably good estimate of a hypothetical sample situated some way between two samples. The first step is to truncate the actual delay time and subtract the truncated delay time from the actual delay time. This outputs a fraction that determines the relative contribution of the two samples, one at the truncated delay time slot and a second sample taken one slot beyond the first delay time slot. The result can be seen in (figure 19).

```
samp1 = delay_line[read_index];
samp2 = delay_line[(read_index + 1) % delay_length];
out_samples[i]= samp1 + fraction* (samp2-samp1);
```

Figure 19 Linear interpolation

E. Freeing Dynamically Allocated Memory

Since the object is having memory allocated dynamically, it is not reliable to rely on the "dsp_free()" routine provided by Max/MSP to clear that memory. It is possible though, the external will still work, however the memory allocated to the delay line will not be freed when a object is destroyed. Allocating memory and not freeing this it when it is no longer needed results in what is called a memory leak. Memory leaks are insidious because they do not manifest themselves by crashing immediately. Rather they gradually suck up memory resources better used elsewhere. In the worst case they could eventually use up all available memory, causing a crash or severely degrading the performance of Max (Lyon, 2011).

A very short (3 lines of code) custom memory freeing function (figure 20) is implemented. It frees the memory associated with the object as well as the memory allocated by the object for itself. It is important to follow the order mentioned above, since the first task removes the object from the DSP

```
void wfs_free(t_wfs *x)
{
    dsp_free((t_pxobject *) x);
    sysmem_freeptr(x->delay_line);
}
```

Figure 20 Memory Freeing Method

chain as well. Until that happens it is not safe to free the dynamically allocated memory since if the DACs are on, the removal of the object might crash Max/MSP.

F. WFS delays & amplitude damping

As mentioned before the key method for re-creating approximate Wavefield is to delay the signal played through every speaker based on the position of the virtual source. To accurately calculate that it is necessary to determine the distance between each speaker, and the virtual source. The length is determined by applying Pythagoras's theorem in in the triangle formed by the following points: virtual source(S), the desired speaker (D) and the point on that determines a perpendicular line on the array and intersects the virtual source position (A). Figure 21 shows the triangle for an arbitrary sound source positioned in point S. The delay is determined by the length of the SD segment, which is $SD = \sqrt{AD^2 + AS^2}$. The coordinates for S point are known, since are passed as in the second and third inlet in the [wfs~] object. The position of the point D is easily determined by multiplying it`s speaker number with the dimensions of a speaker, value that is passed in the inlet 4.

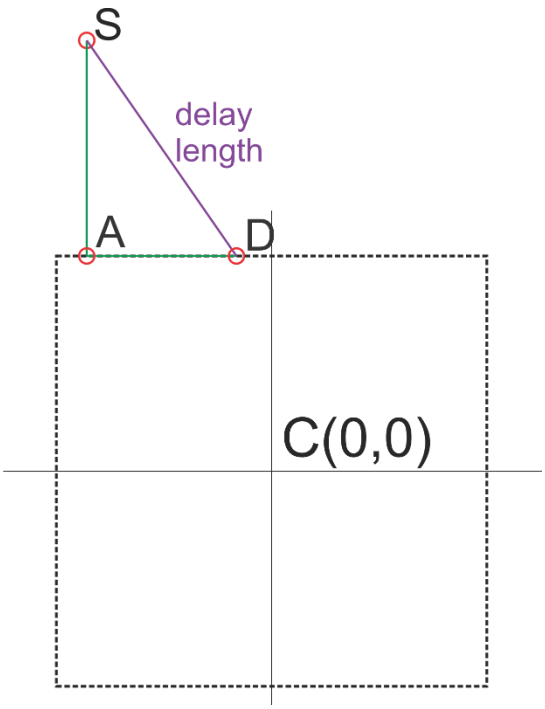

Figure 21 Distance Calculation

The final distance for each speaker can be described by the formula:

$$distance = \sqrt{((speaker\ size * speaker\ number) - X\ coodinate)^2 + Y\ coorrinate^2}$$

Once the distance is calculated using the simple formula time=speed/distance the actual delay length is calculated. The speed used for [wfs~] is C=344m/s. A last step before actually using the delay length in the software is to perform a conversion from seconds to samples by multiplying with the sample rate.

When it comes to the amplitude damping of the signal for each speaker it is usually described as the inverse square law. This law states that the intensity of the sounds chances in inverse proportion to the square of the distance from the source. For Wavefield synthesis Wouter Snoei from The Game of Life Foundation suggests that $1/(distance * \sqrt{distance})$ will yield better results. Since the [wfs~] will be compared to the WFSColider developed by the aforementioned foundation it is also implementing this damping function.

Unfortunately with a loudspeaker setup limited to the horizontal plane, the amplitude of the sound field cannot be synthesized correctly for the whole listening area. Therefore, a certain point inside the listening area is chosen as a reference point for the calculation of the amplitude.

This reference point is typically located in the center of the loudspeaker array (Muller, Geier, Dicke, & Spors, 2014). Therefore and offset equal to half the array size was applied to the distance calculation. The offset is constant for all speakers.

G. Max Routing

An attempt to use all 64 speakers with 4 instances of the [wfs~] was made. Since the signal should not play through all speakers at the same time, a routing system was implemented in Max/MSP using the JavaScript object. The JS object was a series of conditions to turn on gates that would allow signal to be routed to any of the 4 [wfs~] object. There were 8 conditions based on the position of the virtual source, one for each side, and one for each corner (pair of arrays). The two figures (figure 21 & 22) below show how this was implemented. As seen in the pictures, the arrays are turned on or off. This is not the ideal case and a system of crossfading should be implemented in order to avoid hearable clicks and pops when the arrays are suddenly turned on/off. On top of that, and the reason why the evaluation was performed on a single array is that the routing system described introduced lag in how the Max/MSP responds, that in turn introduced artefacts.

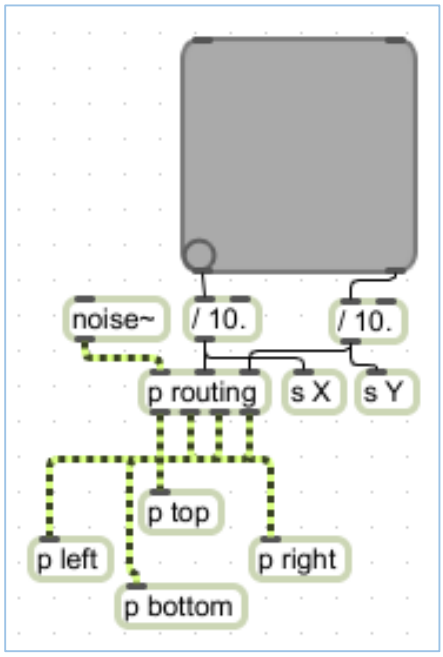

Figure 23 Routing based on position

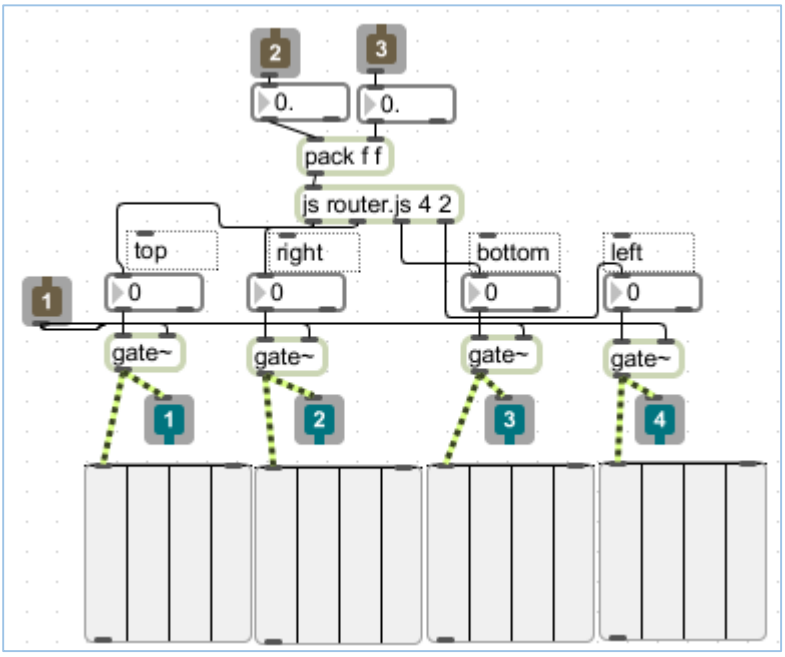

Figure 22 routing sub-patch

H. Summary

This chapter describes the process of creating a Max/MSP object for Wavefield synthesis. Any Max object follows a strict structure of functions called routines. There are four routines, each with a specific task: the “initialization routine” that is called when the object is first loaded and it indicates what functions need to be called in any specific situation. The “new instance routine” called whenever a new instance of the object is created, is mainly responsible for the creation of inlets and outlets. A third function called “the perform routine” is written and inside it lies the

actual behavior of the object. The “DSP method” is the last of the 4 routines and it is the one that adds the external to Max`s DSP chain.

In order to synthesize a wavefield, the fundamental delay is created. In order to create a functional delay it is important to allocate the necessary memory, create a circular buffer to store samples for delay and protect it from possible incorrect address pointing. Lastly, an efficient Max/MSP external can have a custom function for freeing memory when the object is deleted from the patch.

Once the delay system was functional a simple mathematical algorithm was implemented to compute the necessary delay for each individual speaker in the WFS system. Besides the time aspect, an amplitude attenuation is implemented in order to accurately simulate the behavior of sound in real world. A third element that would seem obviously necessary is filtering. Due to the natural low pass characteristic of a WFS array, it is safe to ignore it when working with such a system.

6. Evaluation

In order to see if the [wfs~] satisfies the qualities described in the design chapter, an experiment was conducted using the Wavefield synthesis system in the AAU Copenhagen Multisensory Experience Lab. The hypothesis for this experiment was: *The [wfs~] object performs similar with WFSColider when it comes to the accuracy of virtual static source location detection when using one array of 16 speakers*.

The WFSColider software was used as a reference for comparison because of its availability and previous good experiences with the software, including in studies that focus on distance perception. It is worth mentioning that this experiment is not considering WFSColider as a standard when it comes to localization accuracy.

The decision to use only one array was made because the routing system implemented in Max/MSP produce situation when the sound location was really confusing. This was happening especially when the sources were placed in a corner position. When it comes to the static aspect of the sources, it came as a decision after the pilot test. The users could perceived the motion of the sound, both with [wfs~] and WFSColider, but the direction of this movement was ambiguous.

For this experiment, the participants were exposed to three sounds, a female voice (8 seconds long), and some acoustic guitar chords (8 seconds) and noise bursts (5 seconds). All sounds were created for this experiment so there is complete control over the recording and the quality. All sounds were uncompressed, mono files, sampled at 44.1 kHz. Each sound was repeated 5 times for each of the two conditions. The order of the files was randomized for each participant.

The participants had the task of reporting the position of each sound they heard, by using a Wacom Cintiq 22HD touchscreen, that shown a max patch as figure shows (figure 24)

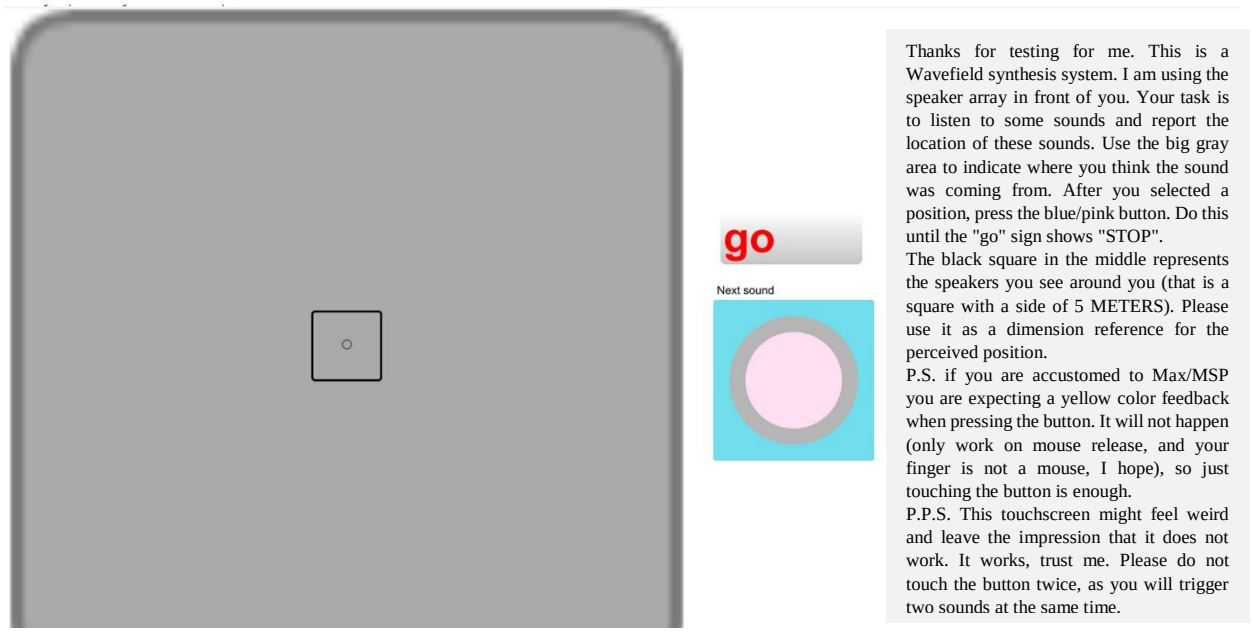

Figure 24 Max Patch displayed on the touchscreen

The text in the right introduces participants to the experiment, explaining what is expected of them.

There were a total of 19 participants mostly students at AAU, divided into three females and 16 males. Most of the participants have some sort of experience working with audio systems, while a handful of them worked with the exact Wavefield synthesis used for this experiment. All participants declared they have normal hearing.

The system designed for this experiment was autonomous and it was in responsible for the random position of each sound source, the random order, as well as the triggering. Sounds played through WFSColider were also triggered from Max/MSP, as well as their position. The connection was made using the Open Sound Control Protocol.

For each sound location generated, the system would log this position, for further comparison with the reported location. After 30 (3*5*2) sound were played, all the data is written in a log file and saved on the drive. The location range was x: 2 to 15 meters, and y -15 to 15 meters. The graph below (figure 25) show the randomly generated sound location for the guitar sounds. Since all three pairs where using the same random generation system, their distribution is very similar.

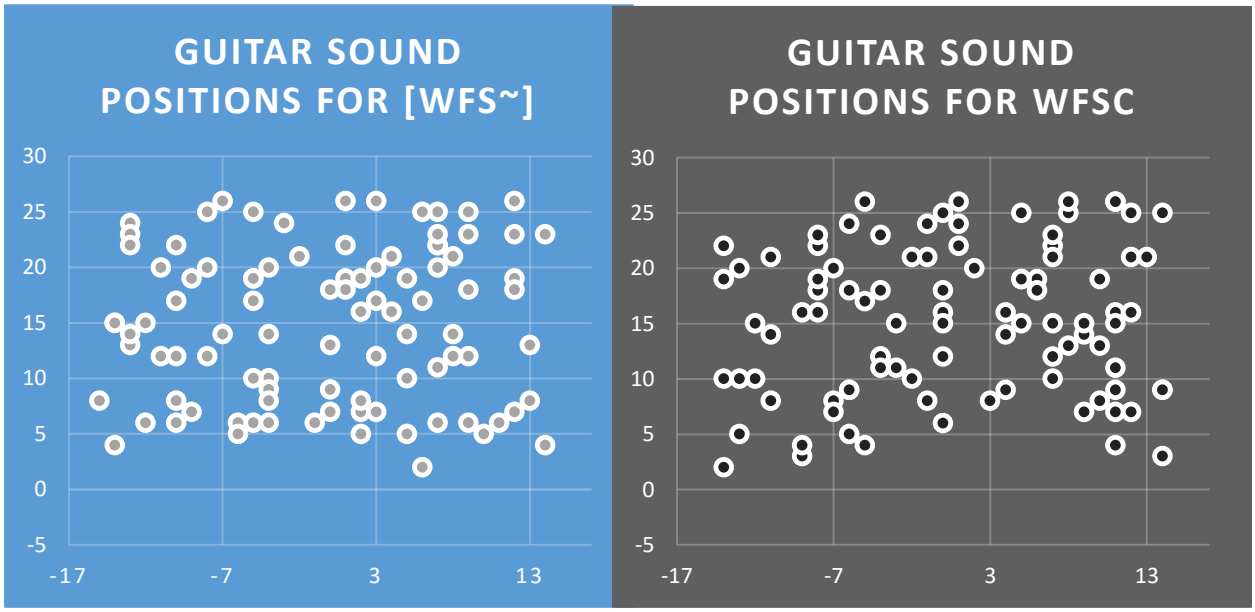

Figure 25 – Random position for the guitar sound for [wfs~] and WFSColider

7. Results

The data that was gathered during the experiment was used to evaluate the performance of the [wfs~] object in relation to WFSColider software. This is done by comparing the distance from the generated sound virtual location to the reported perceived position. Since the experiment is a "within group" design, with only one variable, an analysis of mean would suit to understand the data. For this a method called paired-sample t-test was used. This method was chosen as opposite to an independent sample t-test simply because the two conditions are experimented by the same group of people, thus it is not possible that the data for each condition is independent. The t-Test returns a value, t; with larger I values suggesting higher probability of the null hypothesis being false (Lazar, Feng, & Hochheiser, 2004). In other words, the higher the t value, the more likely the two means are different. Normally it is used a 95% confidence interval in significance test. So any t value that is higher than the corresponding t value at the 95% confidence interval suggests that there is a significant difference between participants (Lazar, Feng, & Hochheiser, 2004). The null hypothesis in this case is: "There is a statistical difference between the two systems".

The data obtained in the experiment was analyzed using Microsoft Excel's paired t-test function. It was performed independent for each of the three sounds, as well as a whole, and it is presented in the table below (table 1).

	Mean Difference (meters)	Standard Dev. Difference (meters)	Standard Error Difference (meters)	T Value	T alpha half 95% CI	Lower Confidence Level (meters)	Higher Confidence Level (meters)
Guitar	-2.23885	7.214302	0.740172	3.17767	1.98580	-3.70868	-0.76902
Noise	-0.16051	8.285663	0.850091	0.38964	1.98580	-1.84863	1.527599
Voice	-2.0016	7.657372	0.78563	2.52880	1.98580	-3.5617	-0.44149
Overall	-1.46699	7.214302	0.740172	3.26305	1.96838	-2.92393	-0.01005

Table 1 t-Test results for the data obtained

Means (m) for:	Guitar	Noise	Voice	Overall
Max/MSP	12.12271741	13.08480139	12.33448528	12.51400136
WFSColider	14.36156687	13.24531507	14.33608462	13.98098885

Table 2 – Means for the difference in the virtual location vs. the reported location

The most important data that table 1 provides is the relationship between the T value and the necessary T value for having 95% confidence interval in order to reject the null hypothesis. It is easy to see that overall, there is no statistical difference between the [wfs~] and WFSColider, when it comes to static sources played back through one array of 16 speakers. If it would be to compare the three categories of sounds independently, the t test indicates that in the case of the noise sound clip there is not enough statistical difference to reject the null hypothesis.

Another aspect that can be deducted from table 1 is that the mean differences for all cases are negative. This indicates that the participants performed better when hearing sounds played

through the [wfs~] object. Unfortunately since the "miss" distance average it is so big (over 12 meters), the small differences of up to 2 meters do not really indicate a great increase in performance, fact confirmed by the rejection of the null hypothesis.

Discussion

The experiment conducted surfaced some important information, besides the one mentioned above. Probably the most important one is the huge perception error, witnessed for both the [wfs~] object, as well as the WFSColider. Looking at the table 2, that reports the average "miss" distance when it comes to the sound localization, it could be tempting to conclude that the whole experiment is a failure, and none of the systems actually works in reproducing 3D sounds. This behavior was not experienced before, when using the Wavefield synthesis system with all 64 speakers. The series of figures below confirm the poor localization on both software, but it also shows that it is more likely to perceive a as coming from the left when WFSColider was used.

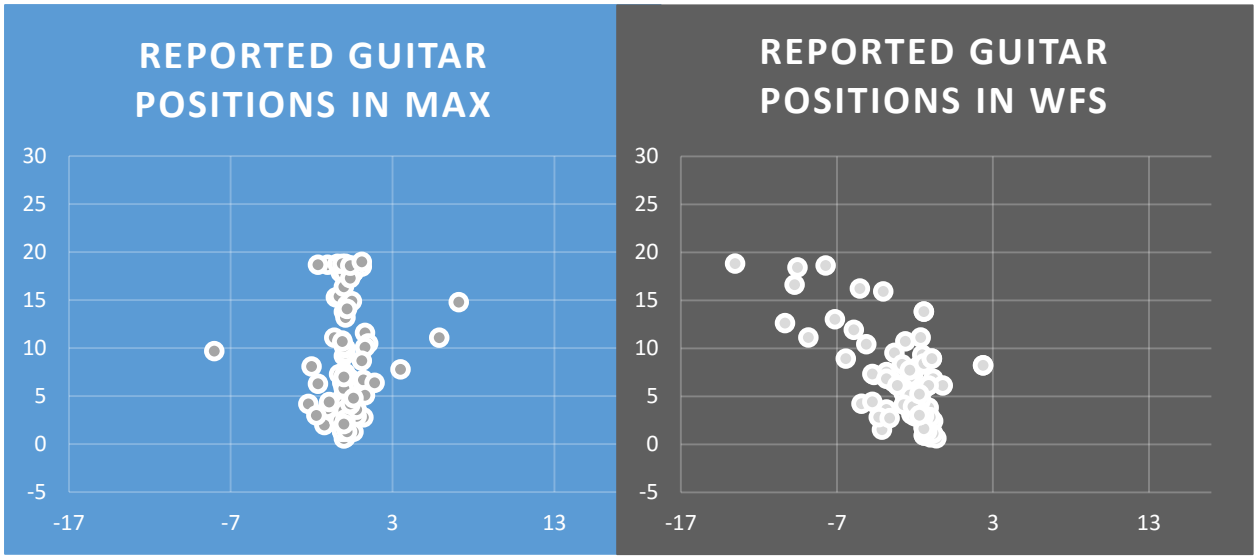

Figure 26 – Comparison of reported Guitar sounds

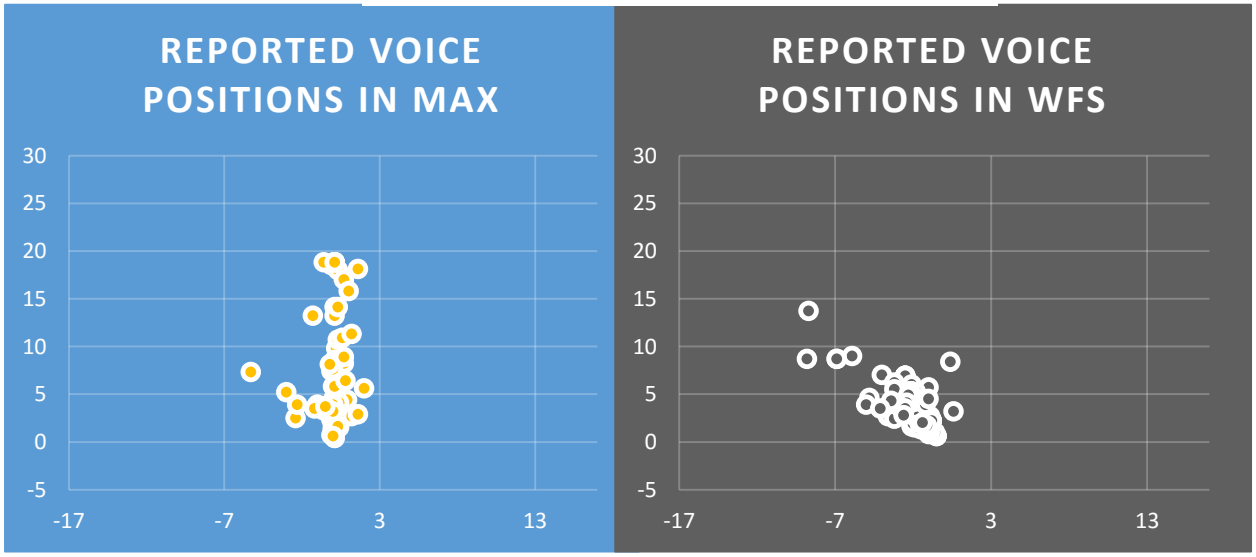

Figure 27 Comparison of reported Voice positions

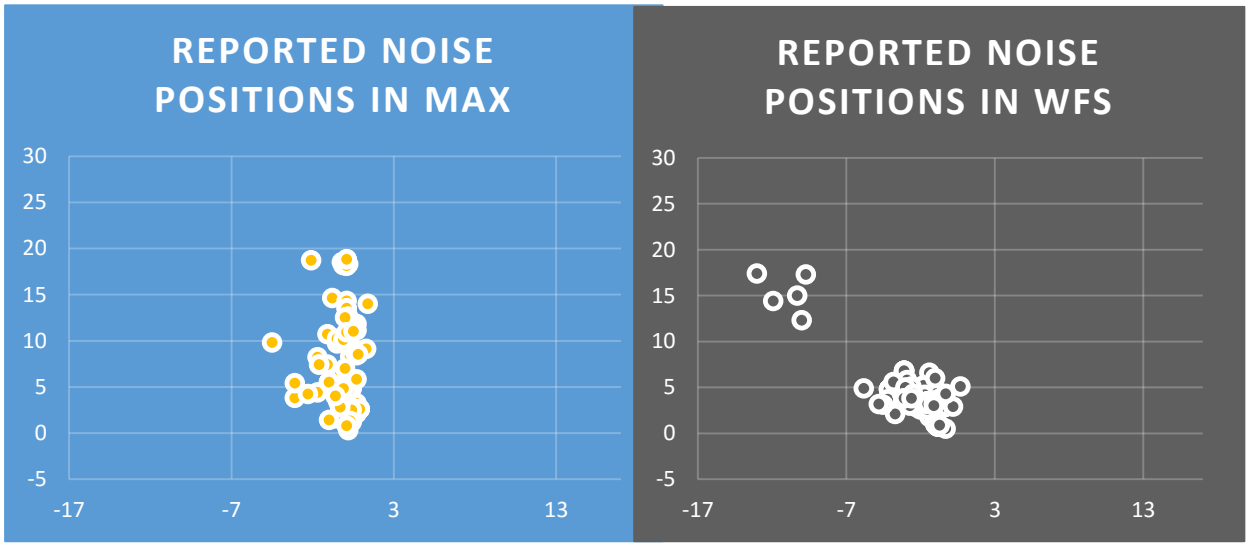

Figure 28 Comparison of reported noise

These figures also show that most of the sounds are localized right in front of the user. This is an indication that the randomly generated position could have been too far and are more likely to be perceived as plane waves than point sources. An improvement in localization could be achieved by exposing participants to much closer, moving sources, since the human hearing has an easier time understanding positions of non-static sources. A further experiment could provide insight into this issue.

An interesting problem reported by the participants was that it was hard to localize sounds because of the visual stimuli. Some of them reported that it was very tempting to look at the individual speakers in the array and associate that with the position of the virtual source. Since this practice would clearly influence the reported distances, some measures should be taken to prevent this. A curtain covering the speakers could be a good solution, if blindfolding the participants is not a possibility (due to the necessity to report positions on a screen).

Since the experiment conducted compare the performance of two pieces of software performing the same task, the output of this experiment is reporting a relative performance. It would be very interesting and insightful to perform a stand-alone localization experiment with the fully working [wfs~]. Such an experiment should feature a vast assortment of sounds and combine both moving and static sources, in order to obtain information about the system`s performance. Such a test should be conducted when further improvements to the [wfs~] object will be made, according to the description provided in the design chapter.

8. Conclusion

This report followed the creation of a Max/MSP external to be used for Wavefield Synthesis. The first prototype was written in C, using the Max SDK. In order to obtain the necessary knowledge to create the object, topics like perception of distance and other 3D audio techniques have been reviewed. Following a thorough analysis of the existing Wavefield synthesis software, a Max object was designed that should be easy to work with and accommodate a vast array of potential setups, called [wfs~]. The first prototype was designed to work exclusively for the Wavefield synthesis system installed in the Aalborg University Copenhagen Campus. In order to test the performances of the [wfs~] object, an experiment was designed. The experiment asked 19 participants to report the location of 3 types of sounds, played through the [wfs~] object as well as the WFSColider software, as a control condition. The results show that there is no statistically significant difference between the two playback systems, when it comes to localizing static sound sources played through a linear array of 16 speakers. The evaluation surfaced a problem when testing for static sound sources, which are really hard to localize accurately, average “miss” distance being over 12 meters. Considering the above, a further test needs to be conducted, when further refinements are done to the [wfs~] object as well as the testing methodology.

Overall the project proves to be heading on the right direction, in order to provide a complex and simple to use Wavefield synthesis method for a popular program: Max/MSP. Hopefully by creating a more user accessible software alternative for Wavefield Synthesis, the technology will become more popular and attract researchers interested in 3D audio.

References

Ambisonics. (2015). *Ambisonics*. Retrieved from Ambisonics Introduction: http://www.ambisonic.net/

Berkhout, deVries, & Diemer. (1989). Acoustic Holography for Sound Control. *Audio Engineering Society*.

Brandenburg, K., Brix, S., & Sporer, T. (2009). *Wave field synthesis.* Ilmenau.

Bronkhorst, A. (1999). *Modeling auditory distance perception in rooms.* Soesterberg.

Corteel, E., & Caulkins, T. (2004). Sound Scene Creation and Manipulation using Wave Field. *Digital Audio Effects.* Napoli.

Cycling '74. (2011). *Max 6 API Documentation.* Retrieved from https://cycling74.com/sdk/MaxSDK-6.0.4/html/chapter_inout.html

de Vries, D., & M.Boone, M. (1999). WAVE FIELD SYNTHESIS AND ANALYSIS USING ARRAY. *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics.* New York.

Game of Life Fundation. (2010). *The Game of Life*. Retrieved from About wave field synthesis: http://gameoflife.nl/en/about/about-wave-field-synthesis/

Gari, S. V., & Kob, M. (2015). Perceptual evaluation of focused sources in a concert hall. *German Annual Conference on Acoustics.* Nuremberg.

Geier, M., & Spors, S. (2012). Spatial Audio with the SoundScape Renderer. *TONMEISTERTAGUNG – VDT INTERNATIONAL CONVENTION.*

Hartley, R., & Fry, T. (1921). The Binaural Location of Pure Tones. *Physics Revised*, 431.

Lazar, J., Feng, H. J., & Hochheiser, H. (2004). *Research Methods in Human Computer Interaction.*

Lyon, E. (2011). *Designing Audio Object for Max/MSP and PD.* Middleton: A-R Editions Inc.

Mershon, D., & King, E. (1975). Intensity and reverberation as factors in the auditory perception of egocentric distance.

Middlebrooks, J., & Green, D. (1991). Sound Localization by Human Listeners. In *Annual Review of Psychology.*

Muller, J., Geier, M., Dicke, C., & Spors, S. (2014). The BoomRoom: Mid-air Direct Interaction.

Oellers, H. (2009). *Germany Patent No. 102009006762A1.*

Oellers, H. (2013). *Wave Field Synthesis.* Erfurt.

Paisa, R., Banas, J. S., Vogiatzoglou, I., Serafin, S., & Grani, F. (2016). Design and evaluation of a gesture driven wavefield synthesis auditory game. *New Interfaces For Musical Expression.* Brisbane.

Pulkki, V. (2015). *Virtual Sound Source Positioning Using Vector Base Amplitude Panning.* Helsinki.

Rumsey, F. (2001). *Spatial Audio.* Focal Press.

Sauer, A., & Snoei, W. (2015). Retrieved from The Game of Life Foundation: http://gameoflife.nl/

Spoer, T. (2004). WAVE FIELD SYNTHESIS - GENERATION AND REPRODUCTION OF NATURAL SOUND. *Digital Audio Effects.* 2004.

Zahorik, P. (1996). *Auditory Distance Perception.*

Zahorik, P., & Winghtman, F. (2001). Loudness constancy with varying sound source distance. *Nature Neurosience*, p. 78.