Towards An Ambisonic Audio Unit Plug-in Suite

Aristotel Digenis

Supervised by: Dave Malham & Jez Wells

MSc Music Technology

University of York

August 27th 2004

Abstract

This project deals with the development of spatial audio software components that are recognised and used by audio editing software. Such software components are known as plug-ins and have been available for quite some time for spatial audio applications. However, they have not been available in the new Audio Unit format, which is part of Apple's OS X Core Audio engine.

A three plug-in suite has been ported from the VST format. B-Decoder decodes Ambisonic (spatial audio technology) material for playback over a variety of speaker configurations. B-Processor allows for the rotation, tilting, and tumbling of Ambisonic sound fields. Lastly, B-Binaural decodes Ambisonic material for playback over headphones, with the ability to decode for different ear types.

This Audio Unit suite however has not simply been ported from the VST format. Features have been greatly improved in terms of performance and usability. More importantly new features have been implemented.

The availability of Ambisonic plug-ins in a greater variety of formats results in greater access by audio engineers and composers whose audio production software of choice might only support a specific plug-in format. As access to Ambisonic tools is improved, composers and audio engineers can further explore spatial audio production using Ambisonics in a software environment they are familiar with.

Acknowledgments

The author would first like to thank his supervisors Dave Malham and Jez Wells for their help and advising throughout this project. Next, thanks must been given to the members of the Core Audio, Music DSP, and Surround mailing lists, as well as the Hydrogen Audio forum. Without their help and suggestions this project would not have been successfully completed. In addition the author would like to thank the members of the 2004 MA/MSc Music Technology course, for the useful conversations, and feedback regarding this project. Finally a big “thanks” to the author’s family and Huang Ying-hsuan for the ongoing encouragement and motivation.

Dedicated to Teli Ladakakou

Table of Contents

1. Introduction

With the increasing support by audio software on the OS X platform for the new Audio Unit plug-in format, it is desirable to provide Ambisonic software "tools" to the users of those audio programs. While Ambisonic plug-ins already exist for the OS X platform in the VST format, it is not a format that all audio software support.

It is possible to "wrap" VST plug-ins inside Audio Unit "shells." This is possible with software such as *VST to AudioUnit Adapter* by *FXpansion* where a VST plug-in is enclosed within an Audio Unit plug-I "shell." Audio programs that support Audio Units will recognise the converted plug-ins, and allow their use. The cost of such software is reasonable and the additional processing required within the plug-in "shell" is tiny ("less than 0.1% per instance on a 550MHz PowerBook G4" according to the developers). However, the conversion may not always be successful (Bigwood). In order to avoid such a problem and to take advantage of the new features available only by Audio Units, a complete re-write of the plug-ins is required. In addition the new plug-ins wisll be available freely on the Internet, avoiding the need for users to use commercial conversion tools such as the AudioUnit Adapter.

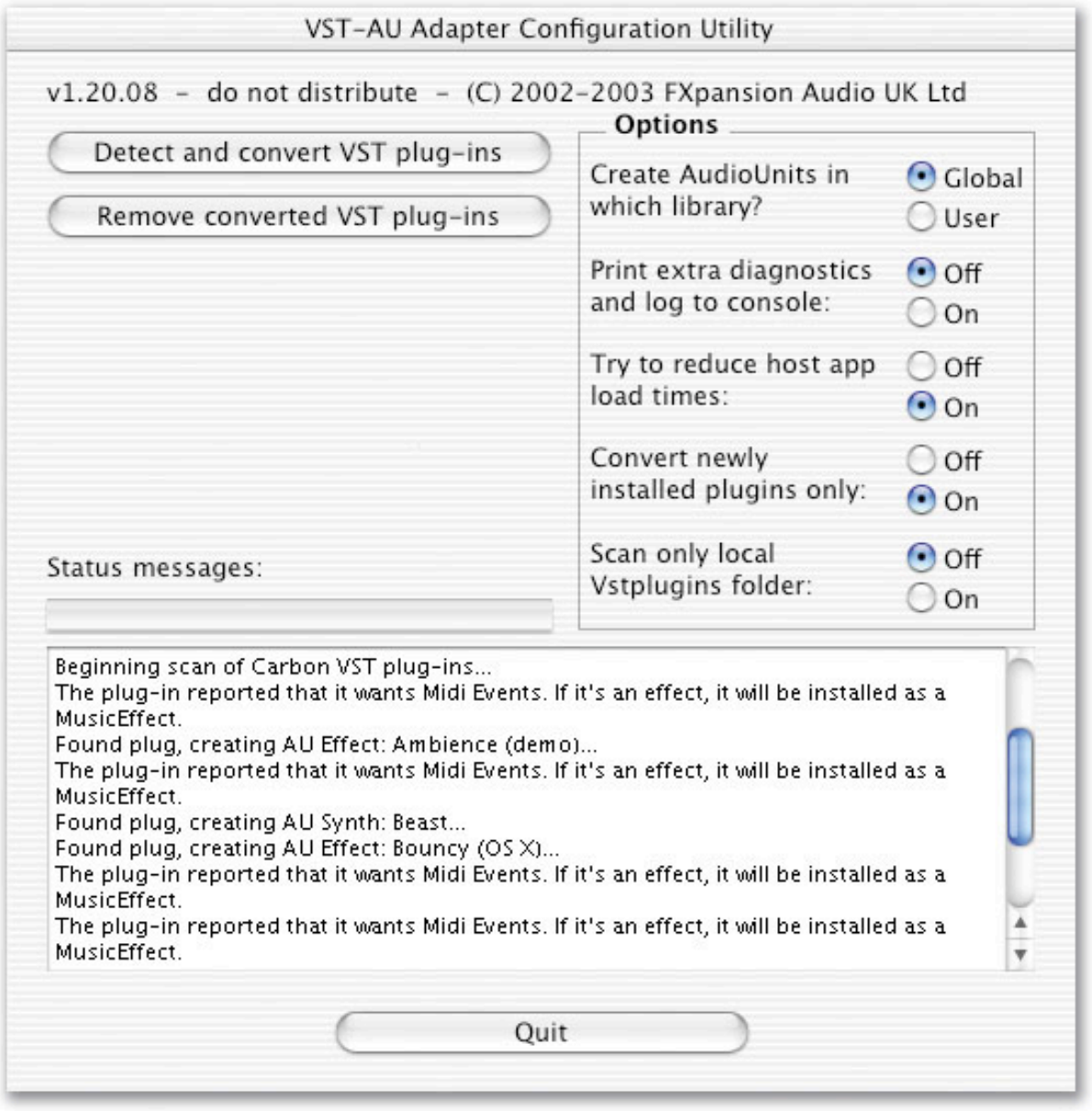


Figure 1: Screen shot of Fxpansion's VST to AU Adapter

It is important to note that while some the source code for current Ambisonic plug-ins is freely available on the Internet, this project does not aim to simply "port" the code to the Audio Unit format. Dave Malham, the primary supervisor of this project, wrote a lot of the existing code. Being more interested in the Ambisonic technology than software engineering, his focus was not placed on code efficiency, flexibility, or expansion. For this project the code will be written with the use of multi-dimensional arrays, iterations, and object-oriented design to make the code shorter, tidier, and more easily expandable at later stages.

In addition, the new plug-ins will have additional features, which will be discussed later in the report. Testing of the plug-ins will be carried out, but test feedback will also be possible through an online feedback form on the web site where the plug-ins will be made available. Aside from testing, the plug-ins will need to undergo specific testing to ensure they meet the format standards specified by Apple.

The next chapter deals with the plug-in technology and the available formats, followed by chapter three which covers the basic human methods for spatial audio perception. Chapter four consists of detailed information about the spatial audio technology known as Ambisonics. The available plug-ins for Ambisonics are discussed and compared in the fifth chapter. Chapter six deals with the implementation of the Ambisonic Audio Units and describes some of the difficulties faced and the ways they were overcome. The testing of the plug-ins and how compliance with the Audio Unit standard was ensured is the topic of chapter seven. The report concludes with the author's ideas regarding the plug-ins and the entire process of the project itself. It continues to suggest possible areas for future work and research.

As this project involves a variety of different areas (plug-ins, software engineering, Ambisonics, Binaural), not all areas will be covered in detail. For example, while Ambisonics as a technology will be covered quite a lot, the psychoacoustics behind it will be kept to a minimum. Also, it is not the project's aim to explain or teach how to program Audio Unit plug-ins, as this in itself is a huge task and there is documentation for this already. Software code will be kept to a minimum throughout the report and used only when necessary in order to explain a processing or data structures implemented. Readers who are familiar with C++ programming and are interested in the details of the implementation, can read the clearly commented code in the appendices.

2. Plug-ins

Plug-ins in the context of digital audio refer to small digital signal processing programs. These are not stand-alone programs, meaning they require a host-program to operate. In a way, plug-ins act as additions to audio sequencing programs, which expand the sequencer's capabilities. The processing offered from plug-ins range from compressors, equalizers, noise and hum removers, to countless other special effects. There are several different plug-in formats, some more advanced and popular than others.

In 1996 Steinberg introduced VST (Virtual Studio Technology). This technology allowed software developers to create VST plug-ins for audio and MIDI processing. Music-sequencing software could load such plug-ins and allow the user to apply processing and effects offered by the plug-ins. Hundreds of VST plug-ins were soon being developed, either commercially or freely. Its cross-platform design makes it a very popular plug-in format.

Another popular audio plug-in format is the one used in Microsoft's Direct X audio and multimedia standard. This format is only available on the Windows platforms and is simply referred to as Direct X plug-in. MOTU's (Mark of the Unicorn) Digital Performer and AudioDesk programs use their own format called MAS. However, this format is amongst the least popular as it is only recognised by software developed by MOTU itself. This is farther limited as most of MOTU software is restricted to the Macintosh platform, which itself has a very small market share compared to the Windows platform.

In March of 2001, Apple announced their new operating system, OS X 10.0. It had a new audio engine, named Core Audio. It promised great capabilities and features, one of those being Audio Units (AU). Also spelled AudioUnits, this was a new plug-in format, which was

quickly supported by the majority of music production software companies that developed products for OS X. MOTU's Digital Performer, Emagic's Logic Audio, Peak, Rax, and Plogue's Bidule, are the just some of those software packages which have supported the new format. Just like the rapid increase in VST plug-ins development, Audio Units are also growing in numbers, and rapidly. However, Audio Units are only available for the OS X platform.

Where Microsoft's Direct X has its own plug-ins, Apple's Core Audio has Audio Units. Both are the operating system's audio engine and they both have their own plug-in format. While there are several plug-in formats used by audio program developers, when the software is intended to operate on a Microsoft operating system, Direct X plug-in hosting capabilities are often included in the program. Similarly, there is an increase in the audio programs for Apple's operating system that offer Audio Unit hosting as well.

Being a new plug-in format, Audio Unit has gained from avoiding the limitations of other formats. The GUI is better defined and as a result is more likely to work properly on a larger number of hosts. A single Audio Unit can have a several GUIs, meaning different skins can be presented to the user for different levels of detail parameter use. Even for the default GUI, there are more options for the type of parameters that can be used, such as drop-down menus, on/off switches, and text boxes. Parameters can have a variety of default values (dB, meters, Hz, etc.), ranges (not limited like VST ranges of 0 to1), and even control over specific in/out port parameters.

One of the biggest features of Audio Units is the signal input and output handling. A single plug-in can specify several possible in/out capabilities, such as 1-in/1-out, 1-in/2-out, 5-in/3out etc. If not specified, then the plug-in by default can handle any number of ins and outs

as long as they match. This is more powerful and processing efficient than in VST where for example if a plug-in has the capability for 2-in/2out, it can also do mono but still needs to process both channels, and the host only uses one of the outputs. In addition to this benefit, Audio Units allow for multiple busses of inputs and outputs.

A VST plug-in needs to be instantiated (opened) before the host can query it for information such as the number of parameters, their names, values, and many more. This can be a processor expensive operation as memory is allocated, samples are loaded in, and the GUI is drawn. This expensive processing is avoided in Audio Units, as the plug-ins have two different loading states. An Audio Unit is "opened" upon instantiation, where the host can query it for all the necessary information. When the plug-in is ready to start processing, it is 'initialized' where all the memory allocations, and other process expensive operations take place.

The limitation of the 7-bit (0-127) value range of VST has been replaced with the ability to use fractional values for notes and controllers, and multiple instances of the same notes. Plug-ins can detect changes in sequencer states such as sampling rate and latency, and notify related portions of the plug-in in order to take any necessary action. Finally, they have a more intelligent and efficient way of handling and storing presets.

With these advantages over other plug-in formats, it is easy to see why the majority of audio software on the OS X platform have adopted the AU format so quickly.

3. Perception of Spatial Audio

Humans rely on several cues to localize sound sources. While some suit some sound sources better, it is the combination of these cues that allows us humans to determine the origin of sounds.

The first cue works for sounds with frequencies up to 700Hz and is known as Interaural Time Difference (ITD). This occurs when sounds arrive at each ear at different times, which results in phase differences. For this reason, this cue is also known as Interaural Phase Difference (IPD).

For sounds with frequencies greater than 700Hz, the Interaural Level Difference (ITD) cue is used. This is a result of sounds arriving at one ear before arriving at the other. The extra distance required for the sound to reach the second ear results in a drop of amplitude in the perceived sound.

Interaural cues ITD and ILD are only useful for localization for horizontally positioned sounds. For vertical localization, the Head Related Transfer Function (HRTF) cue is used. The shape of the outer ear, the head, and upper torso all affect the frequency response of the sound arriving in the ear. The effect they have on the spectrum is defined by the HRTF, and each human ear has a unique response. If one listens through somebody else's HRTF, localization would be poor, especially for the front and rear sound image. While mainly for vertical sound sources, HRTF helps define horizontal sounds. An example of this is when two sound sources of same elevation, and opposite azimuth positions have the same ITD and ILD. Therefore, HRTF is used to clarify the source of the sound.

The mentioned cues detect the direction of a sound, but not its distance from the listener. There are four cues for distance as stated by Dave Malham (1998:170). The level of reverb in an environment does not change much, while the level of the direct sound does change depending on the distance of the sound source. An increase in ratio of direct sound to reverberation is perceived as the source getting closer. Another cue is the "pattern of directions and delays for the early reflections off surfaces in the environment." The third cue is the frequency spectrum of the perceived sound. The further a sound is, there is an attenuation of the higher frequencies. Lastly, in an anechoic environment, sound decreases 6dB (a bit less in non-anechoic environment) for every doubling of distance according to the inverse square law.

4. Ambisonics

a. History

Noticing the short comings of audio playback technologies such as Quadraphonics, a group of British researchers in the early 1970s decided to develop a surround sound technology that would "enable a musical performance to be captured on tape or another medium, for transmission via available or future distribution media to the consumer, and where it would be replayed in a conventional living room in which as far as possible the original sound and acoustic environment of the original performance would be recreated. (Elen (b))" The team made up by researchers including Michael A. Gerzon from the Mathematical Institute in Oxford, and Professor Peter Fellgett of the Cybernetics department at Reading University, set out to create the surround sound system that was going to make up for the errors and shortcomings of other systems. The technology was to be called "Ambisonics" meaning "surround sound (Elen (b))."

Conventional audio systems such as Stereo and Quadraphonics, try to produce a sound field by utilizing amplitude changes to localize sound sources. To achieve a satisfactory spatial image using a conventional stereo set up, the two speakers must be set at an angle of 60 degrees. If the two speakers are set at a wider angle, a gap starts to appear in the middle of the stereo image. This is a result from using only amplitude changes (Elen (2), Rumsey 2001).

In Quadraphonics, where four speakers are set 90 degree apart from each other and relied on only amplitude changes for localization, the results were poor. The front image was poor, the rear image was even poorer, and there was hardly any localization on the sides. When a sound was panned around the listener in a constant circle, the sound seemed to be pulled into the

speakers, creating a “ping-pong” effect. In addition, sound could only be placed at the perimeter of the speaker array, not allowing for sounds to appear outside or within the speaker array. The best results were achieved by sitting at the very centre of the speaker array, an area in audio terms known as the “sweet spot.” The sweet spot in Quadraphonics was very small.

Ambisonics takes into account several techniques that the human ear and brain rely on to localize sound. It was developed using psychoacoustic and physical principals to ensure its capabilities went far beyond those possible when simply using loudness for localization. Interestingly enough, by the mid 1970's, Ambisonics offered a full three-dimensional sound, something that has only been offered by other surround sound systems in recent years.

b. Technology

i. First Order

The core of Ambisonics lies in the B-Format. The first order B-Format is made up of four audio channels titled W, X, Y, and Z. These four channels hold all the information required to playback full three-dimensional sound. Channel W is an omni-directional signal. The other three channels hold spatial information in the method of sum-and-difference, similar to the original stereo technique invented by Alan Dower Blumlein in the 1930’s. Channel X holds information similar to a figure-of-eight microphone facing the front side (front minus back). In the same manner, channel Y faces the left side (left minus right), and channel Z faces the up side (up minus down). These polar patterns represent the following set of spherical harmonics (Elen (b), Rumsey 2001).

W = 0.707

X = cos(angle) * cos(elevation)

Y = sin(angle) * cos(elevation)

Z = sin(elevation)

Michael Gerzon from the University of Oxford initially suggested the use of spherical harmonics for defining directional audio. The four channels of the B-Format define the "first" order of Ambisonics. More specifically, channel W is considered to be the "zeroth" order, and the addition of channels X, Y, and Z raise the sound field to "first" order. In his paper published in 1973, he went as far as to define the spherical harmonics to be added to the B-Format to reach second and even third order Ambisonics.

However, the orders were defined with Cartesian coordinates and by 1975 Ambisonics systems were being explained with the use of polar coordinates. In his later work, he only mentioned first order systems and also used polar coordinates. As a result, higher than first order system needed to be redefined to suit and be consistent with the polar coordinate method that had become standard. While spherical harmonics were used by other professionals such as mathematicians, chemists, and physicists, none of their methods suited the spherical harmonic methods of Ambisonics.

ii. 1.5 Order

In his 1995 MSc thesis, Jeffery Bamford suggested the addition of two channels to the B-Format. The extra resolution provided by these two new channels dealt only with the horizontal plane and therefore this was not considered a full second order three-dimensional (periphonic) Ambisonic system but more of a 1.5 order system. The functions for these two new channels are as follow (Bamford 1995).

U = cos(2angle) * cos(elevation)

V = sin(2angle) * cos(elevation)

iii. Second Order

"Accordingly, using the notation in Kaplan (2000), with all multipliers which remain constant over the surface of the sphere normalised to 1 and all similarly constant signals subsumed in the W signal (Malham (b))," Dave Malham defined the full second order Ambisonic functions shown below.

R = sin(2elevation)

S = cos(angle) * cos(2elevation)

T = sin(angle) * cos(2elevation)

U = cos(2angle) – cos(2angle) * sin(2elevation)

V = sin(2angle) – sin(2angle) * sin(2elevation)

The addition of these components enlarges the listening area where localisation effectively occurs based only on energy psychoacoustic cues. This was later proven to be true in a statistical analysis by Jerome Daniel and others (Daniel & Rault & Polack 1998) and early listening tests (Malham (b)).

Apart from the increase in required audio channels for second order Ambisonics, additional loudspeakers are required. For first order horizontal playback a minimum of four loudspeakers are required for horizontal playback, and a minimum of eight loudspeakers for three-dimensional playback. For second order horizontal playback the minimum is raised to six loud speakers, and twelve for periphonic playback.

A problem appeared when mixing first and second order material. The decoding process significantly differs when dealing with first order only material and mixed first and second order material. Adjusting the encoding and decoding processes can fix this. However this takes away from one of Ambisonics' benefits, which is that the composer or audio engineer, can create a piece of music without having to consider what system and loudspeaker configuration it will be played back over. The solution was to provide two versions of channel W. The first being "W" for first order only systems, and the second one being "W'" for first and second order mixed systems.

iv. Furse-Malham Set

Dave Malham along with Richard Furse later improved on the second order set, titled "Furse-Malham" set. "The intention of the Furse-Malham formulation is that the coefficients generated will be normalised to have maximum values which are either + or - 1.0. This ensures that the signals in the channels have optimum dynamic range. This necessitates a departure from a mathematically pure formulation of spherical harmonics, because some harmonics need to have a scaling factor applied (Malham (a))."

v. Third Order

Jerome Daniel created the third order set but the W component did not have a normalization factor as used in the Furse-Malham set. The W signal had a rate of 0.707 (-3dB) just like in the set initially defined by Gerzon and others. Changing this to 1 (0dB) would be inconsistent with existing systems and therefore has been left unchanged. Below are the functions for third order Ambisonics (Daniel 2000).

$K = \sin(\text{elevation}) (5 \sin^2(\text{elevation}) - 3) / 2$

$L = 8\sin(\text{angle}) \cos(\text{elevation}) (5\sin^2(\text{elevation}) - 1) / 11$

$M = 8\cos(\text{angle}) \cos(\text{elevation}) (5\sin^2(\text{elevation}) - 1) / 11$

$N = \sin(2\text{angle}) \sin(\text{elevation}) \cos^2(\text{elevation})$

$O = \cos(2\text{angle}) \sin(\text{elevation}) \cos^2(\text{elevation})$

$P = \sin(3\text{angle}) \cos^3(\text{elevation})$

$Q = \cos(3\text{angle}) \cos^3(\text{elevation})$

vi. Beyond Third Order

Explicitly deriving spherical harmonics beyond the third order involves somewhat complex and tedious mathematical manipulations. There is however a method to generate higher orders from an existing third order set, using a recursive function. Jerome Daniel has used this technique to simulate with a computer the behaviour of horizontal Ambisonic systems up to the 15th order.

It is important to determine the point at which the addition of spherical harmonic orders will make no significant difference to the listener's perception. In addition, while there is an exponential growth in storage, transfer, and processing technology for audio that allows for the growing Ambisonic orders, a suitable standard of orders would be advantageous for manufacturers and consumers. After all, the higher the order of the systems being proposed, the higher the number of loudspeakers users are expected to use. Even if affordable, this requires users to make dedicated installations, something which a lot of consumers don't bother to do even with existing stereo systems.

c. Recording

Recording a live performance or environment in Ambisonics requires a special microphone. The microphone is called “SoundField Microphone” and is manufactured in England by SoundField. Under the grill of the microphone a tetrahedral array of capsules will be found. These capsules capture audio in the B-Format, which are then fed into a special pre-amplifier that is designed to work with the multiple capsules.

The availability of a pre-amplifier which also acts like a processor, allows the user to rotate and tilt the sound field both during the recording, and after the four B-Format channels have been recorded. Additional possible manipulations include the ability to move forwards and backwards inside the sound field, or totally change the tetrahedral capsule’s behaviour to capture sound in different polar patterns such as cardioid, and omni. The processor also allows for the outputs of a two channel stereo mix that can be used with conventional stereo set-ups. Similar processors extract the appropriate channels out of the B-Format to drive the presently popular 5.1 surround sound set-up.

With the SoundField Microphone limited to first order Ambisonics, attempts were made to develop microphones of higher orders. In his PhD thesis, Philip Cotterell proposes the theory behind a second-order Ambisonic microphone (Cotterell 2000), but this has never been made. France Telecom has been working on a prototype 4^{th} order microphone, but it is a prototype and there is no guarantee whether it will ever be commercially available. At the time of writing, the only commercially available Ambisonic microphone is the first order SoundField.

d. Playback/Decoding

The B-Format channels can be played through an Ambisonic decoder to drive virtually any number and configuration of speakers. Unlike other systems where the number of speakers driven depends on the number of channels available, Ambisonics has no such limitations. The decoder extracts from the B-Format channels a set of interrelated speaker feeds. The speakers work together to recreate the sound field as accurately as possible. Because Ambisonics is based on a consistent mathematical formulation, the decoder can supply interrelated speaker feeds for nearly any speaker positioning. This feature is definitely an advantage over conventional surround sound systems.

e. Effects

i. Rotate, Tilt, Tumble

An Ambisonic sound field can be rotated (turn on the Z axis), tilted (turn on the Y axis), or tumbled (turn on the X axis) by manipulating the B-Format function. For example to rotate the sound field by 'A' amount of degrees (counter clockwise), the functions would be as follow.

$$W' = W$$

$$X' = X * \cos(-A) - Y * \sin(-A)$$

$$Y' = Y * \cos(-A) + X * \sin(-A)$$

$$Z' = Z$$

Combinations of rotate, tilt, and tumble are also possible without the need for excessively more computational power. However it is important to note that the order in which these combinations occur, will have different results.

ii. Mirror

As the name suggests, this effect mirrors sounds in a sound field so they appear on opposite sides of the sound field they originally were in. On the Pan-Rotate unit by Audio+Design this was possible with a rotary knob. At one end of the knob setting, the sound field and the sounds within it were unaffected. At half rotation of the knob, the sound field collapsed into the centre, hence loosing directionality. At the knob's final position, the sounds were on diametrically opposite positions in the sound field than they were originally. The effect was not popular as it only worked well at the extreme settings. In between settings resulted in diffusion, which was not liked by most users. This was a result of limitations of the analogue technology used at that time. The effect works better in the digital domain, as it was implemented in later years.

iii. Dominance / Zoom

Dominance is also known as zoom and allows the user to focus more on sounds in the front of the sound field with lesser focus on sounds at the rear. With the ability to rotate the sound field and face in any direction, this process can be reversed and used in a way to allow for the "zooming" in at any direction. There have been three variations of this effect, which at one stage was also known as "width control." Dominance differs from the type of zoom used by a camera, because when the camera zooms in to a direction, the angles of objects in the front will widen, while in sound field dominance the objects in the direction of the zoom will move closer together (Malham 2003:50).

f. Mixing/Synthesising

With the SoundField microphone, it was possible to record live sound fields, but in order for Ambisonics to gain popularity and have any chance of becoming an industry standard, it had to be made possible to produce Ambisonic material in the conventional multi-track studio environment. Experts began to design equipment to allow for this production capability and with the rapid growth of computing power, software were designed to suit this capability.

i. Hardware

In the early Eighties, Chris Daubney designed an Ambisonic mixer that was built by Alice Stancoil Ltd. Richard Elen was part of a group of experts who designed and built pan pots and other hardware that could be used with conventional audio engineering equipment, to add Ambisonic mixing capabilities.

Later on a company in Reading, England called Audio & Design Recording, produced the Ambisonic Mastering Package. It was this set of equipment that allowed for proper Ambisonic mixing to take place. Designed by Dr. Geoff Barton, the package was made up of three separate rack processors and a professional studio Ambisonic decoder of B-Format with 3 or 2 channel UHJ material (Elen (b)). In late 1983, the first studio with full multi-track Ambisonic recording and mixing capabilities was established. It was in Crunchfield Manor in Berkshire of England. It was only ten miles away from Reading, which was the location for the Ambisonic Technology Centre. This made it an ideal location for both using and trying out the latest products (Elen (b), Elen (a)).

One of the units was a Pan-Rotate processor that featured eight pairs of continuously rotating knobs. One of the knobs in each pair, allowed for a signal to be panned around the listener

within the circumference of the speaker array. The second knob acts like a distance control by varying the amounts of channels W, X, Y and Z. The first of the above mentioned knobs could be used to place a sound northwest of the listener, and the second knob to move the sound from the circumference of the speaker array closer to the centre of the array or further away from the edge of the speaker array. An extra feature was the rotating knob, which allowed for the entire soundfield to be rotated. Each of the other eight set of knobs could be set as "pre" or "post" this rotator knob (Elen (a)).

At the backside of the Pan-Rotate processor unit there were the individual inputs for the eight channels which were controlled by the eight sets of knobs. Four outputs on the backside provide the feeds of the final Ambisonic B-Format mix created within the unit. In addition, there are eight more inputs that make up two sets of B-Format signals inputs. One of these sets of inputs is a pre-rotator, and the other set of inputs is a post-rotator. These two options can be used to link similar units together (Elen (a)).

The second unit the "B-Format Converter" was a simple unit with no controls other than the power switch. It took feeds from four console groups and an auxiliary send, and used the console pan pots to pan across each quadrant of the sound field depending on the pair of groups selected. This unit also provided output for a B-Format signal (Elen (b))."

The third rack was the "Transcoder". Its main function was to create UHJ signals from B-Format material, but it could also be used in another way. The user could input two stereo channels such as a front and rear stage like in Quadraphonics. Two knobs on the front panel allow to set the width of both the front and rear images, resulting in a simple Ambisonic mix in UHJ format (Elen (a)).

In 1983 Keith Mansfield's album titled *Contact*, was the first album to be mixed with Ambisonics. It was part of the KPT Music Library and was pressed on vinyl material by Nimbus Records. The following year, under collaboration between KPM and Nimbus, the first Ambisonic mixed Compact Disc was released. It also was the third Compact Disc made in the United Kingdom, titled *Surprise, Surprise*, by Chin & Cang (Elen (b)).

Several other devices were made but most did not go beyond the prototype phase. Dave Malham's joystick Ambisonic panner was one of those. It was a joystick where the stick could be moved along any combination of the X and Y axis. The unique feature of this joystick was that the stick could be moved upward and downward to control the Z axis of the audio in the sound field.

ii. Software

The rapid growth in computing power and its falling price has resulted in software programs that offer the same and more features of the hardware mentioned. Some of this software was in the form of stand-alone applications that are discussed here, while others were in the form of plug-ins which are covered in the following chapter.

In 1989, O'Modhrain, an MSc Music Technology student at the University of York developed *SurroundSound*. It operated on an Atari ST computer, offering the following processes for Ambisonic sound fields (1989).

- Conversion of a mono or quadraphonic material to first order B-Format.
- Rotate about the vertical or arbitrary axis.
- Tilt about the vertical or arbitrary axis.
- Tumble about the X, Y, or arbitrary axis. This could be done at a specified speed.

- Spin about the vertical or arbitrary axis. This could be done at a specified speed.
- Mirror the sound field about any of the three axes. Speed for mirroring also possible.

A year later, John Clarke from the same course developed AmbiCont (Ambisonic Controller) for an Atari Mega Computer with a parallel interface board. With the use of a GUI and a mouse, sound sources could be placed inside a first order Ambisonic sound field. The sound field could then be processed by rotation, tilt, tumble, and mirroring (1990).

Dave Malham and Tony Myatt of the University of York came up with CSound scripts that allowed for a monophonic sound to be statically placed in an Ambisonic sound field. Start and end points made it possible for the sound to be moved along the sound field edge. However the movement can only happen between two points with linear interpolation. Furthermore, the compiling and processing time was very slow (Malham & Myatt 1995).

Another offline Ambisonic mixing software was the MGP Toolkit, which was also developed at the University of York, this time by Dr. Ambrose Field. The toolkit depended on an Irix based audio sequencing program called Mix. The user generated a multi-track audio sequence in the Mix program, and then Cartesian coordinates were set for points in space for the individual tracks. This toolkit allowed for more flexible positioning of sound sources than the CSound script method. The distance of the sound could be set and reverb applied to sources placed behind the speakers (Mooney 2001).

Another University of York student Dylan Menzies-Gow developed Lamb (Live Ambisonics). As the name suggests his approach allowed for real-time positioning of sound sources in an Ambisonic sound field. One of the features was that sound source positioning could be applied with a user interface (as opposed to being typed in through the computer

keyboard) using the MIDI (Musical Instrument Digital Interface) protocol. It offered three modes for positioning.

When in 'position' mode, a press of a key on the MIDI keyboard moved the position of the sound to the coordinates pre-defined by that key. The coordinates could be defined with controller messages. The speed at which the sound would move to the new position depended on the velocity of the pressed key. In 'cartesian' mode, six keys were set as 'cursor' keys (up, down, left, right, front, back). The press of a key moved the sound in the corresponding direction. The amount of movement depended on the velocity of the pressed key. When in 'cylindrical' mode, the same position occured as when in Cartesian mode but this time with cylindrical coordinates.

Assigning different octaves of the MIDI keyboard to different sound sources allowed for the simultaneous control of multiple sources. MIDI keyboards and controllers could also be used for other parameters such as the rotation of the sound field. While a MIDI keyboard could be used for all the mentioned features, the computer's keyboard and mouse interface could also be used. In fact advanced features such as sound field delay and dominance, could only be adjusted by the QWERTY keyboard and mouse (Mooney 2001, Menzies 1996).

CASED (Composers' Ambisonic Spacial EDitor) developed in April 2000 at the University of York by Ruth Fitzsimmons and David McLeish, "is an editor which allows composers to spacialise mono audio files by specifying coordinates in 3D space and time locations in the given audio file. The program produces an '.scd' (Spacial Coordinates Data) file which can be read and ambisonically processed (Fitzsimmons & McLeish 2000)." The advantage of this program was that it visually represented the location of the sound in all three dimensions. However just like the other programs, the processing was not done in real-time.

In 2001 Dave Malham released an Ambisonic software package called BF which was made up of four sections. The first section was BFPan that panned monophonic sounds in a B-Format sound field. The second section was BFProc which allowed for the rotation, tilting, and tumbling of a B-Format sound field. BFMix was the third section which mixed B-Format material, and the last section was BFDec which decoded B-Format material. While the package had possibly the best Ambisonic capabilities of the time and was capable of second order Ambisonics, it required the user to control it with the use of TCL (Tool Command Language) scripts. The user would need to be familiar with the commands (Mooney 2001).

Most of the Ambisonic software discussed in this chapter did not offer a visual representation of the sound source location and movement. Those that did were limited in terms of graphical appearance and design. All the software except one, only worked with first order Ambisonics, and only one offered real-time processing.

g. Distribution Formats

If Ambisonics was to ever be a popular system, it would have to be compatible with many of the systems available at the time. There were two challenges that had to be resolved before the system had any possible commercial success. The first was that up to that date there was no form of media widely used by consumers that could store the information of the four separate tracks that make up the first order Ambisonic B-Format. The second challenge was that in order to experience Ambisonics, the listener was required to have a B-Format decoder to drive the speakers. Without the decoder, one could not listen back the recording through a conventional stereo and mono system.

The solution to the above challenges was similar to the method used by Quadraphonics namely, to store four tracks on two-track mediums. By using either of phase, amplitude change, frequency change, and summing/difference techniques, four channels could be encoded into two. Those two channels could then be printed on a medium such as vinyl, and when the listener played the vinyl back using special Quadraphonic equipment, all four channels could be extracted to feed four speakers. In addition, if the listener did not have special Quadraphonic equipment, if properly implemented, the playback of stereo or mono was still possible with the use of standard stereo or mono sound equipment.

The developers of Ambisonics decided to come up with a matrix system that would allow Ambisonic material to be stored on two tracks as well as offering both stereo and mono compatibility.

> *"The BBC was experimenting with surround sound at the same time and they adopted Matrix H (presumably the eighth one they came up with: it was allegedly based on QS principles) for experimental surround broadcast. The Ambisonic team also developed a number of matrixing schemes, with varying names: the BBC's Matrix H and the Ambisonic team's 45J matrixing system were combined as Matrix HJ. Work done on the Nippon Columbia UD-4 ("universal Discrete 4-Channel") quad system were also brought into the mix, along with research by Duane Cooper at the University of Illinois, and the final result was referred to as "UHJ" ("Universal HJ"). This is sometimes referred to as "C-Format"* (Elen (b))."

The UHJ format in its full state contained four channels titled "Q", "T", "R", and "L". When all four channels were available, the listener would experience full three-dimensional sound. Channel "Q" could be removed loosing the height information, and the remaining three-

channel version UHJ would provide a very accurate horizontal surround image. Channel "Q" could also be band-limited to result in a 2.5-channel UHJ that still provide good horizontal imaging. Finally, the most common UHJ form was the 2-channel UHJ. Even though the results were not as significant as with the 3 or 2.5-channel UHJ, the surround image was still good.

The different UHJ configurations required a UHJ decoder but when the 2-channel UHJ was played back over a standard stereo system with no decoder, the listener would experience a "3D Super-Stereo" sound.

> *"In this case, the phase and level relationships in the signal, which are based on aspects of the way in which the brain localizes sound sources in nature, lead to a certain amount of "aural decoding" in which the brain tries to make sense of the signals by giving them surround positions and a "super stereo" effect that goes way beyond the speakers* (Elen (b))*."*

Mono capability is achieved by summing up channels "L", and "R" of the two-track UHJ.

In recent years with the development of multi-channel mediums such as the Digital Versatile Disc (DVD), and Super Audio Compact Disc (SACD), the need to create a UHJ mix can be avoided. The recent 5.1 surround sound configuration has become the standard sound amongst many homes and studios. The configuration officially known as the ITU-R BS 775-1 Recommendation set by the International Telecommunications Union specifies the official loudspeaker positioning for 5.1 surround sound (Figure 2) (Lund 2000). The set-up is made up of five channels titled Left, Centre, Right, Left Surround, and Right Surround. There is also the use of an optional subwoofer enclosure, which can be placed anywhere in the

listening area since humans cannot determine the direction or origin of low frequencies. Ambisonics would have to be adopted to take advantage of this new standard in a way that would reach the majority of people.

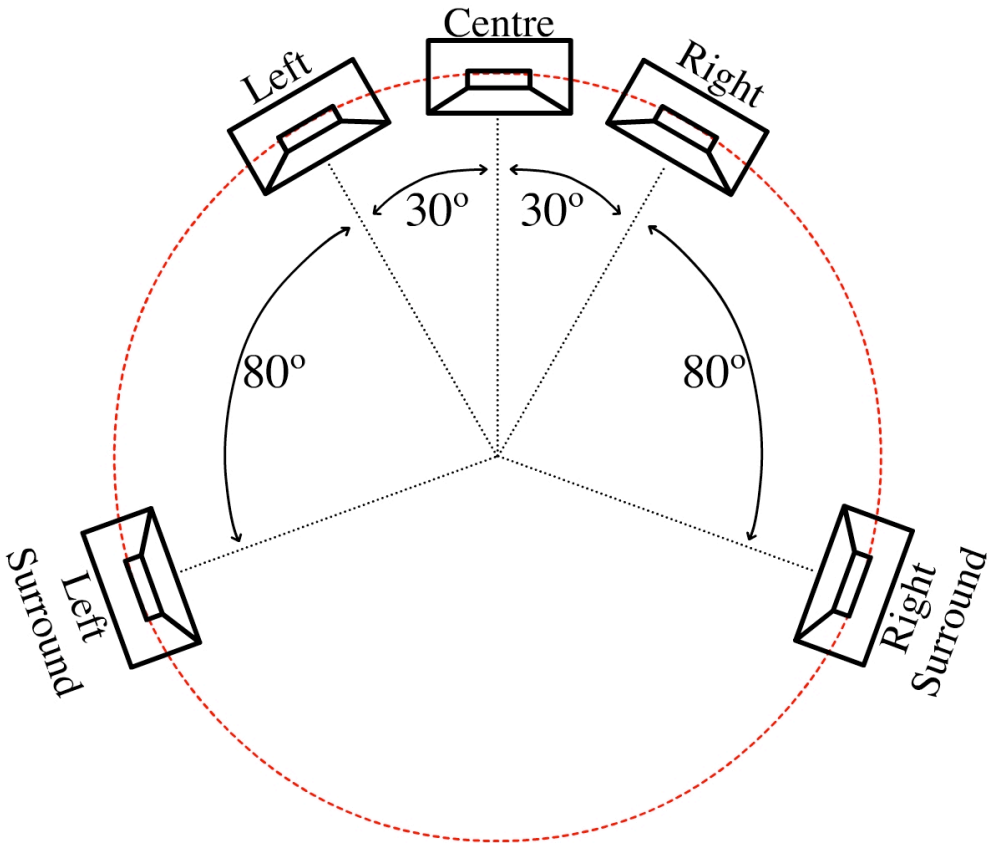


Figure 2: Loudspeaker configuration according to ITU-R BS 775-1.

Ambisonic material could be delivered as a four channel UHJ, but a decoder would still be required to drive the speaker feeds if the listener wants to have more than the a "3D Super-Stereo" sound. Instead of creating UHJ decoder units or adding UHJ decoding capabilities to the DVD and SACD players in the form of DSP (Digital Signal Processing) chips, it would be cheaper and more successful to create another format which would require no processing.

A solution is the proposed "G-Format." It is based on the "B-Format+" which is an advanced form of the initial Ambisonics "B-Format." This B-Format+ is not to be confused with Dr.

Thomas Chen's B+ Format, which is the addition of stereo to B-Format. Due to the fact that Ambisonics has the ability to drive any configuration of speakers, the material can be decoded to drive a 5.1 set-up. The speaker feeds could then be mastered onto a DVD. This process creates the G-Format, which can be played back over a 5.1 set-up with the use of any DVD player. There is no need for any Ambisonic decoding by the consumer.

The downfall of this format is that there is no height information, and the speakers must be positioned strictly according to the 5.1 standards. But the downfalls can still be made up for. It is possible to extract the original B-Format out of the G-Format, including height information. This would then allow users with dedicated Ambisonic hardware to utilize the full benefits of Ambisonics, such as driving any configuration of speakers. However, the exact specifications for the G-Format have not been set so it has is not widely used.

In addition, mediums such as DVD and SACD allow the use of a set of additional two-channel tracks. During the preparation process of the G-Format, it is possible to also simultaneously create a two-channel UHJ mix for the spare tracks on the mediums. This way, those who have a 5.1 set-up could enjoy the G-Format Ambisonic mix, and those who use a standard stereo set-up would experience the Super-stereo effect of the two-channel UHJ. The medium combining both G-Format and two-channel UHJ are titled "G+2" (Elen (c)).

An extra benefit of the G+2 format is that in standard audio production, the creation of an audio DVD requires two separate mixes. One for the 5.1 set-up, and the other for the plain stereo mix. With the G+2 format, only one mix is necessary and once that mix is ready, virtually any configuration of speaker arrangements and formats can be generated with substantial results.

h. Failure to Gain Popularity

With all the benefits and clear advantages of Ambisonics, some may wonder why the technology is not more widely used in audio production. Technically, the creators of the technology have done a great work, but where they have failed is promoting and convincing the market to use it. The creators did not have the required funding to make an attempt at marketing the system, and so they approached the National Research Development Corporation. The NRDC was an organization funded by the English government, which took inventions from inventors who lack the funding to promote the their own inventions. The organization would then approach investors and companies and attempt to get them to licence the invention. If the invention was licensed, then the profits would go to the inventors, with the NRDC earning a share of the income.

The NRDC being an organization which specialized in licensing inventions, had the funding and resources to deal with such matters, but not good enough for Ambisonics. If the invention being licensed were one that could be best dealt by having an exclusive licensee, then having the NRDC handling the negotiations would be an ideal action. For Ambisonics to succeed, it would need to become an audio system standard used by as many licensees as possible. This is where the NRDC failed to successfully promote Ambisonics. Ambisonics needed a large-scale promotion to persuade record labels, equipment manufacturers, artists, producers, and the public to use the advanced system in order to succeed. Live demonstrations, exhibitions, and press conferences are just some of the promotional methods needed. The NRDC was more suited for dealing directly with single licensees.

As time passed, eventually Nimbus Records became the first licence holders. Nimbus Records was a record label that focused on producing classical recordings utilizing

authenticity and the simplest and cleanest signal path during the recording process. The use of the SoundField microphone proved to be a great tool. Other companies that eventually became licence holders included hardware manufacturers Calrec, Audio & Design Recording, Maple, Avesco, and Minim.

Most of those companies lost or gave up the licence, months after obtaining it, without creating any Ambisonic products. Nimbus Records was the only licence holder who has held and contributed to the development of Ambisonics throughout the entire period of three decades, and eventually became the exclusive licence holder. The label tried to approach Japanese electronics manufacturers in order to persuade them to include Ambisonics in their home theatre products.

Mitsubishi was the first company to pick up on Ambisonics creating the DA-P7000 system. It worked entirely on digital components, offering UHJ decoding as well as Dolby Surround material. It also allowed for the conversion of Dolby Surround material to be converted into UHJ. Finally, the unit included a feature where signals could be processed to provide 'Super Stereo' to any type of signal whether UHJ or normal audio. Other manufacturers soon followed such as Onkyo and Meridan, manufacturing home theatre components with Ambisonic capabilities.

Some may argue that the failure for Ambisonics to gain market popularity, was not the NRDC's fault. In the early 1970s when the NRDC and the Ambisonics community was pushing to promote the technology, only the SoundField microphone was available for Ambisonic production. Artists and record producers who had creative ideas of music production and required the hardware to mix in Ambisonics, were not available till the 1980s.

While they waited for the Ambisonic mixing hardware to become available, several years passed, and eventually Ambisonics itself seemed to disappear (Elen (d)).

i. Recent Developments and Future Work

With the rapid growth of computer processing power, computers can be used to encode and decode Ambisonics. Especially with the large market of computer based audio production software, the development of software which would allow recording studio engineers and producers to record and mix Ambisonics would without a doubt be the most appropriate way for Ambisonics to gain market success. Software based mixing tools would be more affordable and easier to distribute through the Internet, compared to t he bulky and expensive Ambisonic mixing hardware of the 1980s. In fact as already mentioned, software developers and audio enthusiasts have already developed such software. Most of them come in the form of plug-ins for the sort of audio sequencing software widely used in the recording industry.

Even though most such software approaches are impressive and carry out the processing successfully, none are yet good enough to attract the attention of the majority of recording professionals. It seems to be only a matter of time before a well developed plug-in for the already popular audio sequencing software packages is made available by an organization that has the appropriate resources and funding to promote Ambisonics. Until such software is released, Ambisonics will remain to be used and be admired by only those who have been lucky enough to come across it in the past, realising its powerful capabilities (Elen (d)).

5. Ambisonic Plug-ins

a. Encoders

There are several Ambisonic plug-ins that allow for sound sources to be panned in a sound field. They are all in VST format and all of them are available for free on the Internet. The following table lists and compares the available plug-ins. They vary slightly in features and capabilities.

Panorama has the best graphical user interface and is the easiest to use. It can encode up to second order. The rest of them have been developed by Dave Malham. Bpan_e is a first order panner without a GUI, with Bpan_e_gui being its GUI version. Bpan_m has the additional feature of mirroring and is also available with a GUI in version Bpan_m_gui. Finally Bpan3 like Bpan_m has mirroring and no graphical user interface but encodes third order Ambisonics. This is the only plug-in which can produce third order Ambisonic material.

Encoders						
Name	Panorama	Bpan_e	Bpan_m	Bpan_e_gui	Bpan_m_gui	Bpan3
Format	VST	VST	VST	VST	VST	VST
Windows	Yes	Yes	Yes	Yes	Yes	Yes
Mac (OS X)	Yes	Yes	Yes	Yes	Yes	No
Order	2	1	1	1	1	3
Input Sources	1	2	2	2	2	2
Air-Absorption	No	No	No	No	No	No
Delay Lines	No	No	No	No	No	No
Mirroring	No	No	Yes	No	Yes	No
Processing Precision	unknown	64-bit	64-bit	64-bit	64-bit	64-bit
GUI	Yes	No	No	Yes	Yes	No
License	Free	Free	Free	Free	Free	Free
Available From	www.gerzonic.net	www.dmalham.freeserve.co.uk/vst_ambisonics.html	www.dmalham.freeserve.co.uk/vst_ambisonics.html	www.dmalham.freeserve.co.uk/vst_ambisonics.html	www.dmalham.freeserve.co.uk/vst_ambisonics.html	http://www.york.ac.uk/inst/mustech/3d_audio/vst/vst3_ambisonics.html

Table 1: Available plug-ins for Ambisonic panning/encoding

Mick Fulton from the MSc Music Technology course at the University of York attempted to implement time delay lines and air-absorption filters for Bpan3. Unfortunately the task was not successfully completed.

b. Decoders

Similarly with the encoders, there is a variety of decoder plug-ins. DecoPro and Emigrator are developed by the same individual and offer nice graphical user interfaces. They differ in terms of number of loudspeakers they can decode, advanced features such as delay lines, and more importantly the one that is freely available only decodes to preset speaker arrangements. Both of them are limited to second order decoding. Since the literature review of this project, the same developer has released EmigratorPro whose only new feature is 64-bit processing.

The Surround Zone plug-in acts as a decoder, virtual microphone, and processor all in one plug-in. It has a very impressive graphical user interface but it is only a first order decoder and to horizontal-only speaker array configurations presets.

The remaining plug-ins are all free but only one has a graphical user interface. B-Dec however is also limited to first order decoding, and only decodes over pre-defined speaker arrays. In comparison with Surround Zone, it has the advantage that some of the presets are fully three-dimensional. Finally, Bdec3 is the same as B-Dec but extended to third order.

In early 2004 the author developed 3AmdiDecAD , which added two new features to Dave Malham's third order VST decoder. They were speaker distance-amplitude compensation, and fully adjustable order-balance per speaker and are explained below.

For every speaker, there is a slider with the range of 0 to 10 meters, with the default set at 5. This parameter represents the distance of the speaker from the centre of the listening area. The ability to specify the individual speaker distance from the centre of the listening area, allows decoding for speaker array configurations where the speakers are not equally distanced from the centre of the listening area. Previously, if a speaker was moved away from the speaker array perimeter, the user needed to adjust the volume slider of the moved speaker to compensate for the volume change caused by the distance moved.

The distance-amplitude feature allows the user to set the speaker's distance from the centre only, and the compensated amplitude is automatically adjusted by the plug-in. If the amplitude of a speaker reaches the maximum, and its distance setting is further increased, the amplitude will not be increased. Instead the amplitude for all the other speakers will be

decreased accordingly. This prevents amplifiers and speakers from distorting, while maintaining an accurately amplitude balanced speaker array.

The amplitude to distance ratio is determined by the inverse square law.

The formula is:

$$\text{dB of change} = -10\log(\text{old distance}/\text{new distance})^2$$

The inverse of this is:

$$\text{dB of change} = (-10\log(\text{old distance}/\text{new distance})^2)^{-1}$$

$$= 10\log(\text{old distance}/\text{new distance})^2$$

While the distance slider affects the amplitude slider, the amplitude slider does not affect the distance slider.

In B-Dec3 it is possible to adjust the balance between the components of different Ambisonic orders globally. This was changed in 3AmdiDecAD by adding this control for every single speaker individually. One way to implement this would be to add a slider for every speaker, with a range of zero to three. The slider would only be able to select values 0, 1, 2, or 3. If the user selected 2 for 2nd order then the variables for the third order processing of that channel would be set to zero. As a result processing would only include the channels of the zero, first, and second order of spherical harmonics.

However a second approach, and the one which was implemented was to add four sliders for all sixteen individual speakers, in addition to the four sliders at the end of the plug-in that control the order balance of the decoder. The sliders control the percentage of the zero, first,

second, and third base. They are by default set to 70.7%, 75%, 50%, and 30% just like the initial four ‘global’ order balance sliders. The user can now control the orders on a per speaker basis. The original four order sliders are still available at the end of the plug-in and act as ‘global’ controls. For example, the movement of the global first order slider sets all the first order sliders of all the speakers. The advantage of this approach is that the user can select distinctively between order sets, but also mix orders together in variable amounts.

Decoders							
Name	DecoPro	Emigrator	EmigratorPro	Surround Zone	B-dec	Bdec3	3AmbiDecAD
Format	VST	VST	VST	VST	VST	VST	VST
Windows	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Mac (OS X)	Yes	Yes	Yes	Yes	Yes	No	Yes
Order	2	2	2	1	1	3	3
Order Weighting	Yes (Master)	No	No	No	No	Yes (Master)	Yes (Master & Individual)
Controlled Opposites	No	Yes(Slider)	Yes(Slider)	No	No	No	No
Amplitude-Distance Compensation	unknown	No	No	No	No	No	Yes
Speakers	16	12	12	8	8	16	16
Manual Configuration	Yes	No	No	No	No	Yes	Yes
Delay Lines	Yes	No	No	No	No	No	No
Processing Precision	64-bit	32-bit	64-bit	unknown	64-bit	64-bit	64-bit
GUI	Yes	Yes	Yes	Yes	Yes	No	No
License	Commercial	Free	Commercial	Commercial	Free	Free	Free
Available From	www.gerzonic.net	www.gerzonic.net	www.gerzonic.net	www.soundfield.com	www.dmalham.freeserve.co.uk/vst_ambisonics.html	http://www.york.ac.uk/inst/mustech/3d_audio/vst/vst3_ambisonics.html	www.digenis.ws

Table 2: Available plug-ins for Ambisonic decoding

c. Virtual Microphones

The following plug-ins allow the user to place virtual microphones inside an Ambisonic sound field, and output the signals that would be picked up by those microphone in that virtual sound field. Most offer control over the microphone direction, elevation, stereo angle, and polar pattern.

The Surround Zone plug-in has a very impressive visual representation of the microphone polar patterns, their angles, and position which all is dynamically updated with the changes of settings. Bformat2surround is more of a decoder than a virtual microphone. It is not based on the B-format decoding functions to derive the outputs, but numerous virtual microphones pointed in the directions of the loudspeakers to be played through. Its controls include the setting of polar patterns for the microphones. Finally, B-Mic developed by Dave Malham has a simple GUI allowing the user to control the direction for pointing of the microphones and their polar pattern. Unfortunately all of the plug-ins are limited to first order processing.

Virtual Microphones			
Name	Bformat2surround	B-Mic	Surround Zone
Format	VST	VST	VST
Windows	Yes	Yes	Yes
Mac (OS X)	No	No	Yes
Order	1	1	1
Output Channels	7	2	2
Polar Pattern Control	Yes	Yes	Yes
Microphone Positioning	No	Yes	Yes
Processing Precision	64-bit	64-bit	unknown
GUI	Yes	Yes	Yes
License	Free	Free	Commercial
Available From	http://www.padraigkitterick.com	http://www.york.ac.uk/inst/mustech/3d_audio/vst	www.soundfield.com

Table 3: Available plug-ins for virtual Ambisonic microphones

d. Processors

In the table below are plug-ins that do various Ambisonic processing. Dave Malham developed the B-Proc plug-in with contributions from the 2000-2001 Music Technology MA/MSc students (Bloor, Newton, Tait, Perry). It is available as a VST for both Windows and Macintosh operating systems and allows the user to rotate, tilt, and tumble first order Ambisonic sound fields. It offers a basic GUI, which consists of three rotary buttons. These can be assigned to control the rotation, tilt, and tumble in various different orders. The order in which these processes are done can have varying results.

With the contribution of the same Music Technology students, Dave Malham also developed the B-Zoom VST plug-in. Also cross-platform and with a basic GUI, it allows the user to "zoom in on, or away from, a point on a complete first order B-Format soundfield." This is achieved by three controls. Azimuth and elevation for the direction to zoom, and another control for the amount of zoom.

The Surround Zone plug-in by SoundField offers both the features of B-Proc and B-Zoom, also on both platforms. It only allows for the rotation, and tilting of a sound field, with no control over tumble. However it offers a great GUI, which does come at a price.

Finally, another plug-in by Dave Malham is the B-Plane Mirror. As the name suggests, it allows for the insertion of a virtual mirror on a plane of the sound field. It offers a basic GUI and three controls; azimuth and elevation for the positioning of the mirror in the sound field, and a control for the type of mirroring. The control varies "from collapsing the field onto the plane perpendicular to the axis when it is at the centre point, to normal field presentation

when fully towards the 'Normal' end point at +1 and mirrored fully towards the opposite direction at -1.0, the 'Mirror' end point."

Processors				
Name	Surround Zone	B-Proc	B-Zoom	B-Plane Mirror
Format	VST	VST	VST	VST
Windows	Yes	Yes	Yes	Yes
Mac (OS X)	Yes	Yes	Yes	No
Other OS	Sadie 5	No	No	No
Order	1	1	1	1
Rotate	Yes	Yes	No	No
Tilt	Yes	Yes	No	No
Tumble	Yes	Yes	No	No
Zoom	Yes	No	Yes	No
Mirror	No	No	No	Yes
Processing Precision	unknown	64-bit	64-bit	64-bit
GUI	Yes	Yes	Yes	Yes
License	Commercial	Free	Free	Free
Available From	www.soundfield.com	http://www.york.ac.uk/inst/mustech/3d_audio/vst	http://www.york.ac.uk/inst/mustech/3d_audio/vst	http://www.york.ac.uk/inst/mustech/3d_audio/vst

Table 4: Available plug-ins for manipulating Ambisonic sound fields

e. Ambisonics to Binaural

In 1990, James Kelly completed his MSc in Music Technology dissertation at the University of York, titled "B-Format to Binaural Transformation." In early 2004 for the Sound In Space module, Charles Gregory from the same course achieved this in the form of a plug-in. It was of VST format only for the PC platform which successfully converted first order Ambisonics to Binaural signals.

This transformation is achieved by decoding the B-Format to a virtual speaker array. The signal for each speaker is then filtered with the HRTF of the speaker's direction for each ear. For example, the signal for a speaker at forty-five degrees clockwise will be filtered with the HRTF of the left ear for a sound at forty-five degrees. This will make up the left ear portion of

the binaural result. The speaker signal will also be filtered with the HRTF of the right ear for a sound at forty-five degrees. The result will make up the right ear of the binaural signal.

As each virtual speaker's signal needs to be filtered with two HRTF (one for each ear), in a virtual quad speaker setup eight HRTF would need to be filtered. However this can be reduced to six and reduce the processing time, by decoding to a virtual quad speaker array that is rotated by ninety degrees. Such a rotated configuration would consist of a speaker at the centre front, centre rear, ninety degrees left, and ninety degrees right. The front and rear speakers are positioned in such a way that the HRTF for their corresponding locations are the same for either left or right ear. Therefore the signal of the front virtual speaker only needs to be filtered with the HRTF of zero degrees position, and that same signal makes up both the left and right ear binaural signal. Same situation for the rear speaker, it will be filtered with the 180 degree HRTF and the result will go to both the left and right ear.

Charles Gregory's plug-in uses this type of speaker array with six filters. The HRTF used are from research carried out at the MIT Media Lab by Bill Gardner and Keith Martin. He has used the compact 128 tap HRTF set. A larger set of 512 taps is available on the web site of MIT Media Lab, but then this would require an even greater amount of processing power, something that was already was a problem.

6. Implementation

a. B-Processor

The first plug-in worked on was B-Processor. Keeping with Dave Malham's style of plug-in naming, it is based on his VST B-Proc that rotates, tilts, and tumbles Ambisonic sound fields. As this would be the least complicated of the final plug-ins in terms of parameters, it was chosen as the one to start development. This proved helpful, for learning and understanding the Audio Unit framework.

In Dave Malham's non-GUI B-Proc plug-in the ranges for the controls are in the range on 0 to 360 degrees (anti-clockwise). If the user wishes to rotate the sound field by 30 degrees clockwise, they must rotate the sound field around the back first. In B-Processor, the ranges are changed to –180 to 180 degrees. In the GUI version of B-Proc, this is not a problem as the controls are continuous rotary buttons. When they reach the maximum of the range, they continue turning, but going back to the minimum value. In future versions of B-Processor where a GUI may be developed, the same continuous rotary controls will be used.

At the time of writing, all Ambisonic processors that dealt with the rotation, tilting, and tumbling of the sound field were limited to first order sound fields. Part of this project is to try and extend that to higher orders. The rotation (about the Z axis) matrices are already defined and applied to B-Processor. However the tilt and tumble matrices are more complicated, in particular for third order and beyond.

As Dave Malham found during his MPhil thesis, the European Union Similugen Esprit Open Long Term Research project investigates "the use of spherical harmonics for defining

illumination in visual rendering systems, a clearly related task (2003).” No simple solution for the rotational matrices was found until 1995. When one was found in 2000, no details of the method were made available. A field that also uses spherical harmonics is Chemical Physics. During a literature search in that field, Dave Malham came across a 1999 publication by Choi, Ivanic, Gordon, and Ruedenberg (1999), “which gives a stable recursive formula for rotations of spherical harmonics.” He suggests that this formula could be adaptable for use in Ambisonics.

It was hoped that this formula could be used in B-Processor to allow for the rotation, tilting, and tumbling of sound fields up to third order. The formula does not appear to have limitations on the order of spherical harmonics being processed, so ideally this plug-in could at later stages easily be extended to even higher orders as well. Unfortunately the publication proved to be too difficult to understand, even with the help of a fellow classmate with stronger math skills, Charles Gregory.

While the rotate, tilt, and tumble processes up to third order using the recursive formula was not achieved, B-Processor was still implemented to allow for the rotation only, up to third order. A drop down menu labelled “Order” was implemented with two options, “1st”, and “Up to 3rd – Rotation only.” When the “1st” order option is selected, all rotate, tilt, and tumble parameters are processed, but when the option for higher order processing is selected, only rotation is applied. Finally, the plug-in uses a drop-down menu for the user to select the order in which the manipulations should take place. The possible order options are, Rotate-Tilt-Tumble, Rotate-Tumble-Tilt, Tilt-Rotate-Tumble, Tilt-Tumble-Rotate, Tumble-Rotate-Tilt, and Tumble-Tilt-Rotate.

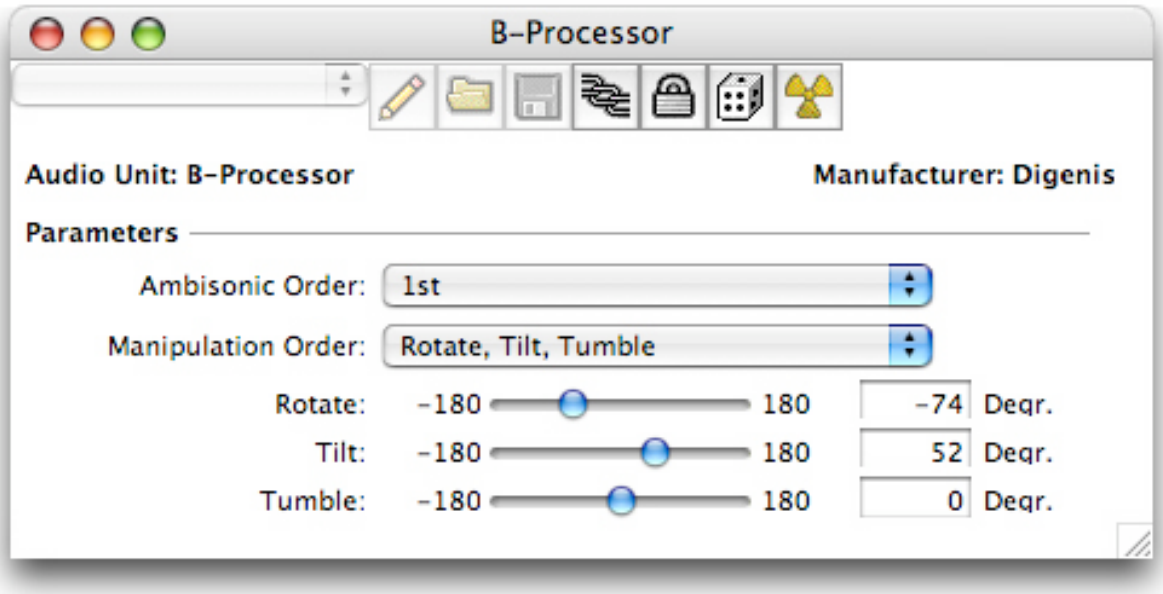


Figure 3: Screen shot of B-Processor's default GUI in Plogue Bidule

b. B-Decoder

This plug-in was intended to have all the features of the author's 3AmdiDecAD, including distance-amplitude compensation, and order balance setting on a per-speaker basis as well as globally.

However in 3AmdiDecAD, there are 132 parameters in the form of sliders. This makes the use more difficult. Ideally, in a custom GUI version of such a plug-in, detail parameters such as order-balance per speaker would be hidden, and only shown by choice of the user. Audio Unit's default GUI allows for parameters that are drop down menus. This capability, which is not possible in the VST format, helped implement the more than 132 parameters of B-Decoder without cluttering the screen.

There is a drop down menu labelled "Speaker #" that has options numbered 1 to 16, representing all the speakers the plug-in can decodes to. Bellow the menu are eight sliders for

the azimuth, elevation, distance, amplitude, zeroth, first, second and third order balances. The settings of these eight parameters depend on the speaker number selected by the “Speaker #” menu. For example, selecting speaker number six would adjust the eight parameters bellow to reflect the settings of speaker six. This design greatly reduces the screen space required for parameters, without the need for a custom GUI. The layout of the plug-in can work for any number of speakers the plug-in is extended to decode for.

While the plug-in offers controls over all the parameter on a per speaker basis, adjusting the same parameters for all the speakers at once can be tedious. For this reason, global parameters control the order balances for all the speakers at once. Global parameters also are present for the distance (radius of the speaker array) and amplitude, which were not available in 3AmbiDecAd or B-Dec3. As a result, if the user wishes to change the amplitude to –6dB for sixteen speakers, it can be done with the change of one slider, not sixteen.

At the very top of the plug-in is another drop down menu labelled “Ambisonic Order” with the options “Zeroth”, “First”, “Second”, and “Third.” As the names suggest, these options set the order up to which the decoder should process. It also works as a preset for the order balance between components of different orders. For example, selecting the option “Third” will set the global order bases (and in turn all the individual order balances) at 70.7%, 75%, 50%, and 30%. The values for these settings are stored in a two dimensional array of floats as shown bellow.

```
// Default values for order balance
const float ambisonicOrderBases[4][4] =
{
    {1.0, 0.0, 0.0, 0.0},        // Zeroth Order
    {0.707, 1.00, 0.0, 0.0},     // First Order
    {0.707, 0.75, 0.5, 0.0},     // Second Order
    {0.707, 0.75, 0.5, 0.3}      // Third order
};
```

While Jerome Daniel has published formulas for optimum balancing of components, during tests in real concert hall scenarios, Dave Malham found that these should only be used as a guideline. As a more general guideline he suggests that as the number of speakers increase, channel W should decrease. In addition, "third order components will be set lower than second order ones which will be less than first. First order components, in turn, need to be set higher than zero'th order components (Malham, 2003:120)." The settings used for this plug-in follow this general guide, but users should experiment to find the ideal settings for their scenario and set-up.

Just like in B-Processor, the ranges for the azimuth, and elevation parameters were changed from 0 to 360 (anti-clockwise) to –180 to 180 (negative values are left from centre and positive values are right from centre), and the values are shown as degrees. The distance parameters are shown in meters, ranging from 0 to 10. Amplitudes are shown in decibels with a range of infinity and 0. The minimum range actually is –40dB at which point the display changes to shown the sign of infinity. Order base parameters are shown as percentages.

B-Decoder has fourteen speaker array presets, three of which are periphonic. A three-dimensional array of floats is used to store the settings for these presets. In every preset, there is a value which represents the number of speakers that preset consists of. This is later used to decode only the required number of speakers for that preset. Other information contained for each preset, is the azimuth (range of –180 to 180), and the elevation (range of –90 to 90) for each speaker. Bellow is an example declaration of such an array.

```
// Values for presets
float allPresets[][3][16] =
{
    {   // 3D 8 - Cube
        {8},
        {45.0, 135.0, -135.0, -45.0, 45.0, 135.0, -135.0, -45.0},
        {-45.0, -45.0, -45.0, -45.0, 45.0, 45.0, 45.0, 45.0}
    }
};
```

The presets available are mono, stereo, quad, pentagon, 5.0 (5.1 positions but without the LFE channel), hexagon, 7.0 (7.1 setup but also with no LFE channel), octagon, decadron, dodecadron, horizontal 16, 8 speakers in 3D, 12 in 3D, and 16 in 3D. Consideration was given to implementing the LFE channels for the 5.0 and 7.0 presets to make them 5.1 and 7.1. This would have been achieved by feeding a copy of channel W through a low pass filter. However the 5.1 standard recommends that the band limited (maximum 120Hz) LFE channel is only intended for "sub-bass" frequencies, such as special effects and not simply an additional bass channel (Rumsey, 2001:90). The decision was that the additional programming and processing for this implementation was not worth it for this specific and time restricted project.

Putting features aside, B-Decoder differs from B-Dec3 in terms of software design. In Bdec3, individual variables are declared and used for every azimuth, elevation, or other parameters. This leads to large sized code, which is more difficult to adapt for more speakers carry calculations. B-Decoder is based on an object-oriented design where there are 'speaker' objects. Each speaker contains its own parameter variables, and methods for the necessary calculations. In addition to the benefits of such an approach, 'speaker' objects (and other variables) are stored in arrays, which reduces code size. Even more importantly it allows for the use of iterations for processing.

The benefits of such a design are the ease of expanding the plug-in for more speakers.
In general, all aspects of the code are programmed in a way to allow flexibility and expandability. For example, the "Speaker #" drop down menu options and their numberings are generated during construction of the plug-in object, and are determined by a global variable that defines the number of speakers.

With regards to performance, this is also an area of improvement over B-Dec3. It will process only for the required number of speakers and only for the required number of orders.

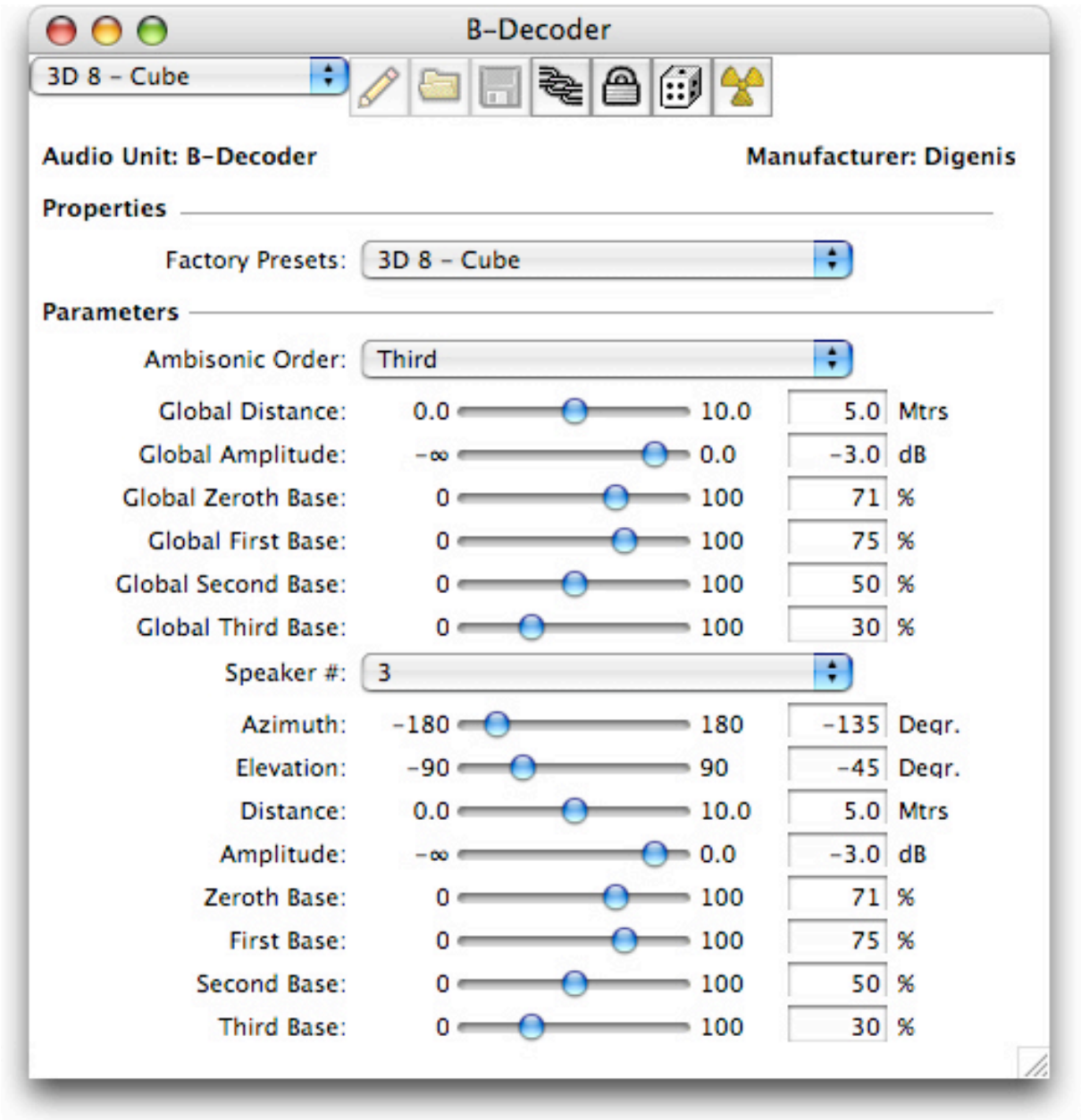


Figure 4: Screen shot of B-Decoder's default GUI in Plogue Bidule

c. B-Binaural

The aim of B-Binaural was to make Charles Gregory's plug-in available in the Audio Unit format. In addition, ways would be considered to make it more processing efficient. If this

were achieved then, a greater number of speakers would be used to achieve three-dimensional decoding. Furthermore, the process would be extended to third order Ambisonics.

The first stage of the implementation was to simply port his code into an Audio Unit plug-in and test it. On the author's 667MHz G4 Apple PowerBook system, the plug-in required 147% of the CPU. The audio output was full of clicks, pops, and discontinuities, as the computer could not keep up with the required processing tasks. It was now even more important to find ways to improve the performance of the transformation.

The Ambisonic decoding portion of B-Binaural is based on the code used in B-Decoder. It offers the same flexibility, processing efficiency, and expandability. It also includes four speaker array presets, two of which are periphonic.

Until this point, filtering was done by convolution in the time domain. The process is quite CPU intensive. Another way to achieve filtering is to use FFT Convolution, which is multiplication in the frequency domain. The incoming audio signal and the filter are converted from the time domain to the frequency domain using a FFT (Fast Fourier Transform). The frequency components of the incoming signal and the filter are then multiplied. An Inverse FFT on the multiplication result gives a filtered signal in the time domain. In general, where filters of more than 64 taps are used, it is faster to use multiplication in the frequency domain instead of convolution in the time domain.

There are plenty of libraries and classes available to perform the FFT and its inverse. Some are fasters than others, while some are easier to implement. It was up to the author to choose a suitable one. Such a code was found in the CD-ROM of the book *"A Programmer's Guide To*

Sound" by Tim Kientzle. Unfortunately it would not compile, and before too much time was lost trying to fix it, alternatives were found.

The next solution was Apple's vBigDSP library. It offers AltiVec optimization for G4 processors, which would further improve performance. Unfortunately, when including it in the plug-in header it would not compile either. After discussions on mailing lists and forums, Stephen Davis from Apple mentioned that the code is out of date and suggested how to fix it. However, the result was still hundreds of compiler errors. Members of the Core Audio mailing list recommended the vDSP class that is part of Apple's Performance/Accelerate library. They suggested that while vBigDSP is easier to use, vDSP is faster in most cases. It appears that vBigDSP is intended for processes where data length is 2^{16} or greater.

After a lot of difficulties, complications, and delays, the implementation of vDSP worked. One of these delays was a result of a problem with loading the impulse response data into a Double Complex Split structure. The following function does just that. Variable `dPtrHRTFImpulseResponses` needs to be a pointer pointing to dynamically allocated memory, and cannot be a static array. While the end result of a dynamically allocated memory block and a static sized array is the same, it turned out the following function only deals with the first type.

```
ctozD((DOUBLE_COMPLEX *) dPtrHRTFImpulseResponses[a][b][c], 2,
&earHRTFDoubleComplexSplit[a][b][c], 1, nOver2);
```

An alternative and possibly better FFT library would have been FFTW (Fastest FFT In The West). It is cross platform, free, and more importantly claims to perform great for a variety of processors. The use of this library would be useful if at later stages this plug-in would be ported to non-PowerPC processor systems, such as Windows VST formats. However for this

particular project, since Audio Unit can only operate on OS X which in turn can only run on PowerPC CPU systems, the use of an Apple tuned library seemed logical and appropriate. In fact, according to tests by the developers of FFTW, vDSP performs better for the FFT size used in B-Binaural.

Due to the dramatic performance improvement, the complete 512 tap HRTF filters were used as opposed to the 128 tap HRTF set used by Charles Gregory. This itself should improve the binaural experience. In addition it was extended to decode up to third order Ambisonics over several different speaker arrays with many speakers. The smallest speaker array is a quad but it is not ninety degree rotated. This requires a total of eight filters instead of six, but the processing efficiency of B-Binaural allows for this. It can be argued that using a standard quad array with eight filters provides better localization, as there are no on-axis speakers. There is however no evidence to support this at this point, and neither is it the purpose of this project to experiment on this.

When convolving a signal of n samples by a filter of size m, the result will be of length n+m-1. The signal expands by m-1 samples to accommodate for the filter's 'ring.' In time domain convolution this ring is added to the end of the signal and is known as 'linear.' However, in FFT convolution (multiplication in the frequency domain) the result is 'cyclic' meaning that the ring wraps around to the start of the resulting signal. A way to overcome this is a process called overlap-save.

An alternative and the chosen way to deal with the filter ring is to carry out the FFT on arrays that have extra space at the end of them to allow for the signal expansion. After the convolution, the m-1 samples after the first n samples of the result are stored in a temporary buffer. This buffer is added to the n samples of the following convolution result, and the

buffer is updated to store the ring of the new convolution result. This method is known as overlap-add and is used in this plug-in. The process overview is as follows.

1. Store 512 samples of incoming audio to an array of size 1024
2. Zero-pad the remaining half of the array
3. Forward FFT the array contents using a 1024 FFT size
4. Multiply the frequency domain of the array with that of impulse response
5. Inverse FFT the result array contents using a 1024 FFT size
6. Scale down the array contents
7. Add overlap buffer contents
8. Update overlap buffer to hold the end of the convolution result
9. Output first 512 samples
10. Repeat from the beginning

Unfortunately the output signal was distorted and had a ‘buzz’ effect. At early listening trials there was a sense of spatial information in the binaural signal, but was not clear due to the errors in the signal. At a meeting with second supervisor Jez Wells, attempts were made to figure out the cause of this ‘buzz’ in the signal. He suggested that more trials be carried out using sine waves and look at the waveforms of the results.

A new plug-in was written without any Ambisonic decoding or other features of B-Binaural whose large code made debugging more complicated and difficult. It simply FFT convolved an input signal with an impulse response, and output the result. When looking at the visual representation (waveform) of the output signal, discontinuities could be seen. These occurred at constant intervals of 512 samples. At the end of every 512 samples, the wave did not match

in amplitude with that of the beginning of the next 512 samples. This did not take place when the multiplication of the frequency domains command was commented out.

After further discussions with Jez Wells, it was thought that windowing should be applied to frame blocks before the convolution and during overlap-add. This would reduce frequency leakage and discontinuities between sample blocks as the start and end of each sample block would have the same amplitude of zero. Windowing and the use of a more complicated overlap method before outputting the signal should eliminate any artefacts in the signal.

There are a variety of different window functions all with their advantages and disadvantages. The one chosen for this plug-in was the Hanning function (not to be mistaken with Hamming function). This was applied to the first incoming 512 samples before the zero padding of the rest of the array. At this point, the result was affected by the rapid changes in amplitude change for every 512 samples. In addition there still were artefacts like before. During discussion with Dave Malham and Dr. Ambrose Field, it was suggested to try different window functions. Unfortunately these did not get rid of the small artefacts either.

The rapid change of amplitude every 512 samples was expected, as an appropriate overlap-add method was not implemented yet. This proved to be far more complicated than anticipated. Due to the windowing function, a 50% overlap was required between sample blocks. The implementation of such an operation proved to be too difficult to complete.

Next step was to look back at literature and in particular chapter eighteen of The Scientist and Engineer's Guide to Digital Signal Processing (Smith, 1999), which deals with FFT convolution. While several sources and people on mailing lists and forums suggest the need for windowing in FFT convolution, other sources (including the mentioned book) do not.

Attention was again turned to going over and checking the code without the windowing functions.

After significant time and effort, the problem was not found and the issue was raised on forums and mailing lists on the Internet. The problem and question on the need to window was discussed and a section of the code was posted. Some people insisted that it is mandatory to use windowing while others argued that windowing was not needed. Further more, those who believed there was no need for windowing found that the posted code seemed correct. Finally, Daniele Terdina, a member of the Music-DSP mailing list noticed a problem with the following function

```
zvmulD(&incomingSignal, stride, &impulseResponse, stride,
&resultSignal, stride, log2n, 1);
```

The argument log2n states the number of elements of the two signals that function zvmulD is to multiply. Variable log2n had a value of 10, meaning that only the first 10 samples of the input signal and impulse response were being multiplied/convolved. This was changed to the correct variable nOver2 that had a value of 512.

This fixed the problem of distortion and 'buzzing'. However there were still some crackles, but this time of very low amplitude and only on the right ear output. It turned out the problem was in the array used to store the overlap samples. The array has four dimensions (described in greater detail later). The last two dimensions were declared as [511][2] which were used to store 511 (filter size –1) samples for 2 ears. Later in the program the array was used but with the last two dimensions with ranges of [2][511]. This did not give a compilation or runtime error, however the result was artefacts in the right channel of the output signal. It was fixed by

changing the arrays declaration to a [2][511] form, and with this correction the FFT convolution was working correctly.

Just like with the other two plug-ins, arrays were used where possible in order to reduce the code size, variable labelling, and future expandability. The following code is for three dimensions of pointers to dynamic allocated memory, which is where the HRTF impulse responses are loaded at the beginning of the plug-in.

```
double *dPtrHRTFImpulseResponses[numberOfSpeakerRings]
[numberOfAngles][numberOfEarTypes];
```

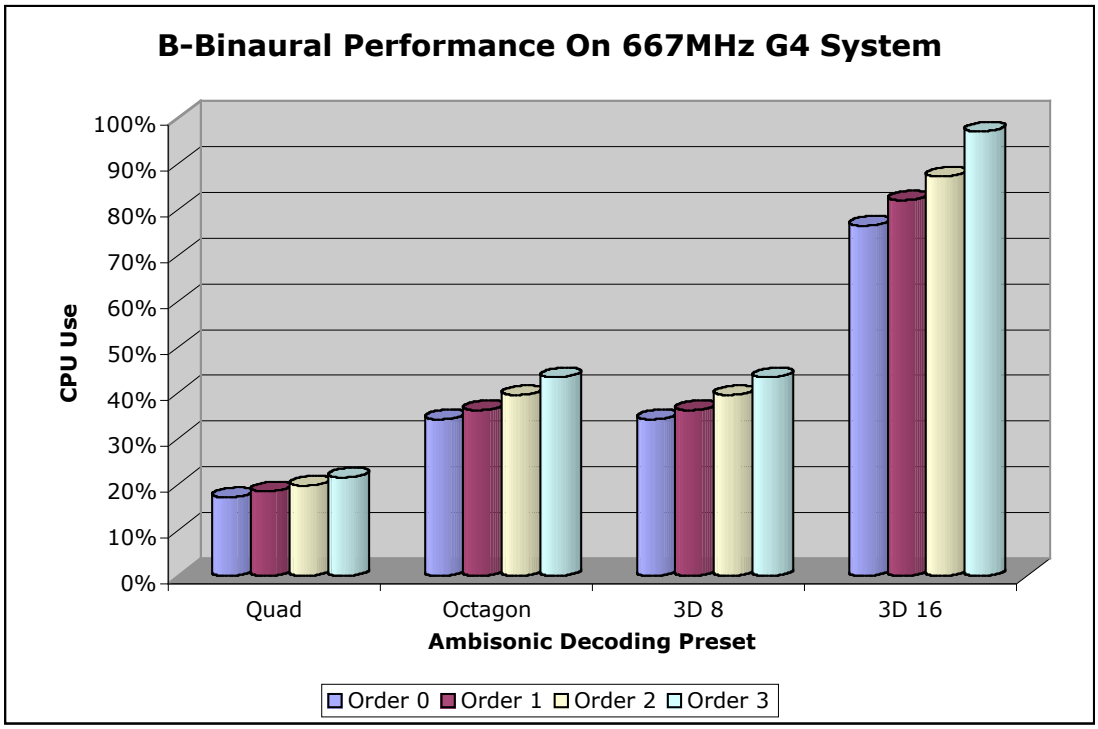
The first dimension refers to three elevation positions, the second refers to eight azimuth positions, and the third refers to two types of ears. They all point to new memory allocations of 1024 double variables, making the array four-dimensional. For example, to access the fifth sample of the HRTF impulse response for azimuth 45, elevation 40, and ear type large, one would use the following command.

```
double x = dPtrHRTFImpulseResponses[2][1][1][5];
```

The same approach was used for the data containers used for the Double Split Complex type objects needed for the FFT functions, the overlap-add buffers, and arrays containing the signal from the decoded virtual speakers.

As mentioned earlier the plug-in offers two different sets of HRTF sets for two different types of ears. One is for normal, and the other is for large ears. With the use of a drop down menu, it is possible to choose the type of HRTF in an attempt to suit the user's personal ear shape more closely.

The following chart illustrates the performance improvement when using FFT convolution instead of convolution in the time domain. While Charles Gregory's code required 147% of the CPU for first order decoding with 6 convolutions of 128 tap filters, B-Binaural achieves third order decoding with 16 convolutions of 512 tap filters before reaching 100% on the CPU indicator.



Graph 1: Graph showing CPU usage of B-Binaural on author's system

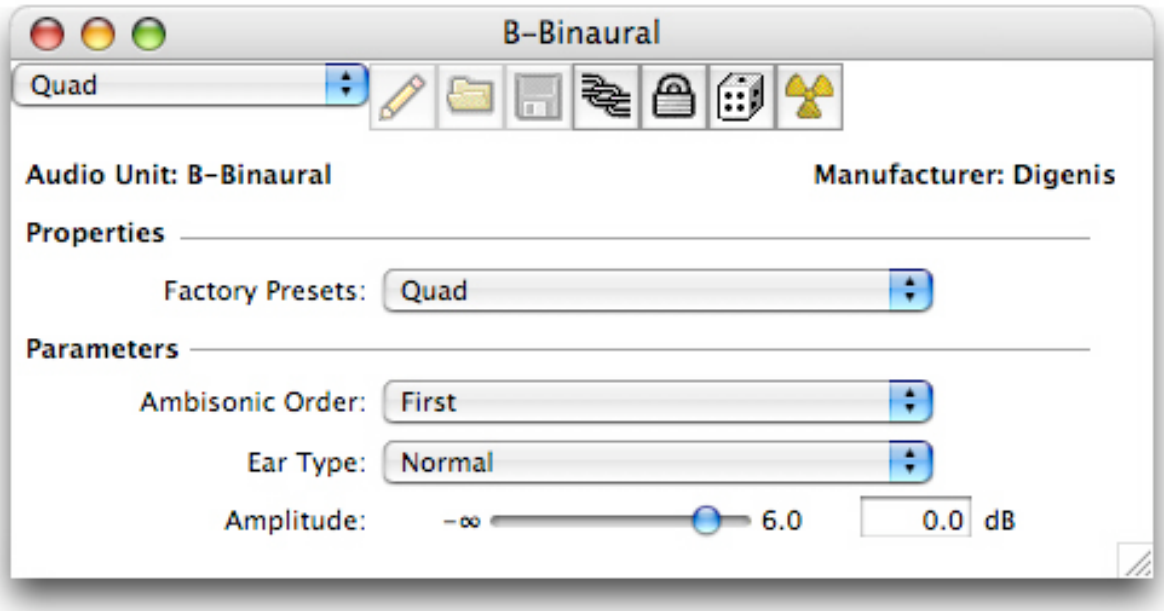


Figure 4: Screen shot of B-Binaural's default GUI in Plogue Bidule

7. Testing & Additions

In order to ensure the Audio Units conform to the AudioUnit API standard and expectations, Apple has made available a command-line utility called AUValidation. Its purpose is exactly that. To validate Audio Units and output a report containing all the internal tests carried out on the plug-in and the final result, Pass or Fail. If successful and the report can be sent to Apple, and permission is given to the developer to use the official Audio Unit logo when distributing them. All three plug-ins pass this validation process and the author has the option to apply for this logo use permission. The results can be found in the appendices.

As the AUValidation tool only checks for operational correctness and semantics of plug-ins, it was necessary to carry out personal tests to ensure the proper functionality of the plug-ins.

All the plug-ins were tested with Plogue Bidule version 0.6601 as it was freely available, offers Audio Unit support, and more importantly was easier to use for a large number of channel input and outputs. A M-Audio FireWire 410 soundcard was used to output the multiple speaker signals that were fed to a home theatre system. New cables and connectors had to be made to connect the 1/4” outputs of the sound card with the 1/8” stereo inputs of the home theatre. Both live Ambisonic recordings captured with the SoundField microphone, and synthesized sound fields from mono sound sources using the Panorama VST plug-in were used for testing.

Aside from testing over a home theatre system, tests were carried out in the Rymer Auditorium of the university’s new Music Research building. The intention was to test the plug-ins over a fourteen-speaker array. All the required equipment was gathered and set up. Unfortunately there were problems with the interaction between the Bidule and the MOTU

896 sound card used. As a result, the M-Audio FireWire 410 sound card was used which could only drive up to eight speakers.

During testing, a problem appeared with B-Decoder. The decoded sound field appeared very diffused and sources appeared to bounce in unusual directions. After careful reviewing of the code, an obvious code error was noticed in the processing function. The following code shows that speaker outputs were stored into the incoming B-format arrays, which of course were later used to calculate more speaker feeds.

```
// Store the values of the B-Format inputs for that frame.
for(int speakerCounter = 0; speakerCounter < speakersInPreset;
speakerCounter++)
{
    audioData[speakerCounter][j] = (float)
    (audioData[W][j] *
    speakerPointer[speakerCounter].GetSpeakerCoefficient(W)) +
    (audioData[X][j] *
    speakerPointer[speakerCounter].GetSpeakerCoefficient(X)) +
    (audioData[Y][j] *
    speakerPointer[speakerCounter].GetSpeakerCoefficient(Y)) +
    (audioData[Z][j] *
    speakerPointer[speakerCounter].GetSpeakerCoefficient(Z));
}
```

This was corrected by storing the values of the incoming B-Format to temporary variables and those variables used for accumulating the signal for each speaker. This is shown bellow.

```
// Store the values of the B-Format inputs for that frame.
for(int a = 0; a < kNumberOfInputs; a++)
{
    tempBFormat[a] = audioData[a][j];
}
// Derive the output for all the speakers in the preset and store it in
the buffer.
for(int speakerCounter = 0; speakerCounter < speakersInPreset;
speakerCounter++)
{
    audioData[speakerCounter][j] = (float)
    (tempBFormat[W] *
    speakerPointer[speakerCounter].GetSpeakerCoefficient(W)) +
    (tempBFormat[X] *
    speakerPointer[speakerCounter].GetSpeakerCoefficient(X)) +
    (tempBFormat[Y] *
    speakerPointer[speakerCounter].GetSpeakerCoefficient(Y)) +
    (tempBFormat[Z] *
    speakerPointer[speakerCounter].GetSpeakerCoefficient(Z));
}
```

In-ear headphones were used in tests of B-Binaural. The sound field perception is best in the "Quad" preset. Presets with more virtual speakers resulted in slight phase. With eight speakers in a "Cube" formation, it does not sound as clean as the "Quad" preset. This gets worse in the two other presets where there are more than four virtual speakers at the same elevation.

In Plogue Bidule version 0.6601, all three of the plug-ins work perfectly. However they do not work with Rax, or DSP Quattro. A possible cause of this is that both of the hosts do not support more than two-channel inputs or outputs, and all the plug-ins require sixteen-channel inputs. The developer of Rax was contacted regarding this matter, and he confirmed that Rax would only supports plug-ins with up to two inputs or outputs. The exchanged emails can be found in the appendices.

Before making the plug-ins available, a few final steps were required. The plug-ins needed to be given specific identification information. The manufacturer tag needed to be registered with Apple and had to be limited to four characters. The registration application was filled out and the author was granted permission to use the code "Dige" for his plug-ins. The individual names assigned for each plug-ins were "Proc" for B-Processor, "Deco" for B-Decoder, and "Bina" for B-Binaural.

Next, a custom icon was designed for the plug-ins. The letter "d" in a fancy font was used. Transparency and shadowing was applied to match the popular styling of icons on the OS X platform. The result can be seen at the figure below.

Figure 5: Icon for the Ambisonic Audio Unit plug-ins

When B-Processor and B-Decoder were finished, they were uploaded to the author's website. Their availability was announced in the Surround mailing list. In order to get feedback from users about the compatibility of the plug-ins with software the author did not own, an online feedback form was implemented. This form can be seen in the appendix. Unfortunately nobody used the form for feedback.

8. Conclusions & Further Work

Several Ambisonic VST plug-ins have been successfully ported to the Audio Unit format, making them the first Ambisonic plug-ins in this format. These plug-ins have not simply been ported but improved in terms of software design resulting in greater expandability, and processing efficiency. Furthermore, new features have been added, and are listed below.

- B-Processor
    - 2nd and 3rd order rotate.
- B-Decoder
    - Several presets of speaker arrays.
    - Global controls for distance, amplitude, and order balance.
    - Per speaker controls for azimuth, elevation, distance, amplitude and order balance
    - Distance-amplitude compensation per speaker.
    - Drop down menu allows user to choose the speaker's controls to view and edit as opposed to showing all the parameters for all the speakers at the same time, which clutters the screen.
    - Option to decode Zeroth, First, Second, or Third order, doing only the required processing for the selection.
    - Processes only for the number of speakers in the selected preset.
- B-Binaural
    - Up to 3rd order from 1st.
    - Two and three dimensional speaker presets.
    - Option to decode Zeroth, First, Second, or Third order, doing only the required processing for the selection.
    - Processes only for the number of speakers in the selected preset.

- Use of multiplication in the frequency domain as opposed to convolution in the time domain, for greater performance.
- Optimised for the G4 processor Velocity Engine resulting in greater performance, hence more virtual speakers and higher HRTF filter banks.
- 512 HRTF filter taps as opposed to 128 used by C. Gregory.
- Option to use different set of HRTF depending on the size of the ear.

Just like with most projects, even if all the goals set out at the beginning of the project have been successfully completed, there is more work and research that can be done. In fact it is usually the case that upon completion, more questions have been raised since commencing the project. This project has been no exception. Future work in this field could include the following.

The implementation of the recursive formula for the rotation, tilt, and tumble of spherical harmonics mentioned earlier, would be a great addition to B-Processor. Once this was done, it would also be easier to extend it to even higher than third order Ambisonics.

B-Decoder could have automatic time delay lines added. This would be useful for decoding to speaker arrays of a large radius, where the time delay for the signal to travel from the speaker to the listener would be greater. Another useful addition would be air absorption filters.

B-Binaural could be improved by using HRTF sets recorded at higher sampling rates, more taps, and with a greater variety of ear types. This would however require more processing and add a greater delay as FFT convolution works in blocks of samples. There are however ways to overcome this delay. The HRTF set used has a minimum elevation of –40 degrees. HRTF sets with elevations lower than –40 would be beneficial. A simpler addition would be to add

parameters for the user to control the Ambisonic order balance, just like in B-Decoder. Lastly, just like the user can select the type of the ear to convolve the signal for, the plug-in could allow the choice of different headphone types (in-ear, over-the-ear, etc.).

While the default Audio Unit GUI offers full control over the plug-in parameters, a custom GUI could provide better and more accurate control. The benefit would not only be for control purposes but also for aesthetics and appearance.

The time restrictions of this project resulted to the completion of three Ambisonic plug-ins. This leaves at least another three Ambisonic effects to be added to this suite in future work. The effects are mirroring, zooming (dominance), and finally panning. Of the three, the panner is of greatest importance, as it is currently the only way to create higher than first order Ambisonic material without a SoundField microphone.

The entire project proved to be far more difficult than initially expected. This was in part due to the complexity of the Audio Unit API but more importantly the lack of documentation for its development. As the API is relatively new, there is very little support for it compared to the VST framework and documentation. Audio Units are extremely flexible and not limited simply to audio effects/processors and virtual instruments. Even the mailing list search function is under developed, where search results do not appear in chronological order. Fortunately, the few members of the Core Audio mailing list were helpful.

Finally, the author feels the project has been successful. Both the goals and learning outcomes initially stated have been achieved. The author has learned about Audio Unit development, gained an insight into Ambisonic technology, and completed a suite of the first Ambisonic plug-ins in the Audio Unit format. The learning outcomes have been exceeded as the author

now has a better understanding of convolution both in the time and frequency domain. It has been an intensive but enjoyed project, making this field of work the author's primary career path choice.

9. References

Bamford J. (1995) *An Analysis of Ambisonic Sound Systems of First and Second Order* Unpublished MSc thesis, University of Waterloo.

Bigwood R. (2003) *DP4 and FXpansion's VST-to-AU Adapter*. Sound on Sound, December.

Choi C., Ivanic J., Gordon M., Ruedenberg K. (1999) *Rapid and Stable Determination of Rotation Matrices Between Spherical Harmonics by Direct Recursion.* Volume 111, Number 19, Journal of Chemical Physics.

Clarke J. (1990) *Real Time Ambisonic Soundfield Controller*. Unpublished MSc thesis, University of York.

Cotterell P. (2002) *On the Theory of the Second-Order Soundfield Microphone*. Unpublished PhD thesis, University of Reading.

Daniel J. (2000) *Representation de Champs Acoustiques, Application a la Transmission et A la Reproduction de Scenes Sonores Complexes dans un Contexte Multimedia*. University of Paris 6: Paris, France. Available from: http://gyronymo.free.fr (Accessed 20 May 2004).

Daniel J., Rault J., Polack J. (1998) *Ambisonics Encoding of Other Audio Formats for Multiple Listening Conditions*. In AES 105th Convention. (Corrected version available by contacting the authors at Centre Commund'Etudes de Tele-diffusion et Telecommunications, Cesson Sevigne, France).

Elen (a) R. *Ambisonic Mixing – An Introduction*. Ambisonic.net (online). Available from: http://www.ambisonic.net/ambimix.html (Accessed 17 May 2004).

Elen (b) R. *Ambisonic Surround-Sound in the Age of DVD*. Ambisonic.net (online). Available from: http://www.ambisonic.net/ambidvd.html (Accessed 17 May 2004).

Elen (c) R. *"G+2" A Compatible, Single-Mix DVD Format for Ambisonic Distribution*. Ambisonic.net (online). Available from: http://www.ambisonic.net/gplus2.html (Accessed 17 May 2004).

Elen (d) R. *Whatever Happened to Ambisonics*. Ambisonic.net (online). Available from: http://www.ambisonic.net/ambi_AM91.html (Accessed 17 May 2004).

Fitzsimmons R., McLeish D. (2000) *CASED:Composers' Ambisonic Spacial Editor*. Unpublished MSc thesis, University of York.

Huber M. D., Runstein E. R. (1995) *Modern Recording Techniques*. Fourth Edition. Sams Publishing: Indiana, USA 1995.

Kaplan W. (2000) *Advanced mathematics for engineers*. Addison-Wesley, Reading, England 1981.

Lund T. (2000) *Enhanced Localization in 5.1 Production*. In AES 109th Convention. Los Angeles, California, USA. 2000.

Malham, D. (1998) 'Approaches to spatialisation' (Organised Sound, vol.3,no.2). Cambridge, Cambridge University Press.

Malham (a) D. *Higher Order Ambisonics*. University of York (online). Available from: http://www.york.ac.uk/inst/mustech/3d_audio/secondor.html (Accessed 20 May 2004).

Malham, (b) D. *Higher order Ambisonic systems for the spatialisation of sound*. In ICMC99. 1999, Beijing.

Malham, D. (2003) *Space In Music – Music In Space*. Unpublished MPhil thesis, University of York.

Malham D., Myatt A. (1995) *3-D Sound Spatialization using Ambisonic Techniques*. Computer Music Journal 19:4 pp.58-70. 1995.

Menzies-Gow D. (1996) *LAmb – Live Ambisonics*. University of York: York, England. Available from: http://www.york.ac.uk/inst/mustech/3d_audio/lamb.htm (Accessed 14 May 2004).

Mooney J. (2001) *Towards a suite of VST plugins with graphical user interface for positioning sound sources within an ambisonic sound-field in real time*. Unpublished MSc thesis, University of York.

O'Modhrain S. (1989) *An Ambisonic B-format Soundfield Processing Program To Run On An Atari S.T. In Conjunction With The Composer's Desktop Project – Sound Filing* System. Unpublished MSc thesis, University of York.

Smith W. S. (1999) *The Scientist and Engineer's Guide to Digital Signal Processing*. Second Edition. California Technical Publishing: San Diego, USA 1999.

Rumsey F. (2001) *Spatial Audio*. Focal Press, Music Technology Series: Oxford, England 2001.

10. Other Sources Used

Branwell N. *Ambisonic Surround-Sound Technology for Recording and Broadcasting*. Ambisonic.net (online). Available from: http://www.ambisonic.net/branwell_arb.html (Accessed 17 May 2004).

Daniel J., Moreau S. (2004) *Further Study of Sound Filed Coding with Higher Order Ambisonics*. In AES 116th Convention. Berlin, Germany. 2004.

Eargle E. (1976) *Sound Recording*. Van Nostrand Reinhold Company, New York 1976.

Elen R. *Ambisonics – A New Age*. Ambisonic.net (online). Available from: http://www.ambisonic.net/NewAge.html (Accessed 8 March 2002).

Elen R. *Ambisonics – BBC Soundfield Experience*. Ambisonic.net (online). Available from: http://www.ambisonic.net/sfexp.html (Accessed 17 May 2004).

Elen R. *Ambisonics for Audio-Visual*. Ambisonic.net (online). Available from: http://www.ambisonic.net/ambav.html (Accessed 17 May 2004).

Elen R. *Ambisonics for the New Milennium*. Ambisonic.net (online). Available from: http://www.ambisonic.net/gformat.html (Accessed 17 May 2004).

Elen R. *DVD, Surround and Ambisonics*. Ambisonic.net (online). Available from: http://www.ambisonic.net/dvda.html (Accessed 17 May 2004).

Elen R. *Transcoding "Quad" Recordings to Ambisonics*. Ambisonic.net (online). Available from:
http://www.ambisonic.net/quaduhj.html (Accessed 17 May 2004).

Gerzon M. (1975) *Pan Pot And Sound Field Controls*. Report No.3, N.R.D.C. Ambisonic Technology. August 1975.

Glinos A. (2000) *Software Ambisonics Decoder with Automated Setup*. Unpublished MSc thesis, University of York.

Huber M.D., Runstein E.R. (1995) *Modern Recording Techniques*. Fourth Edition. Sams Publishing, Indiana 1995.

Kientzle T. (1998) A Programmer's Guide To Sound. Addison Wesley: Boston, USA 1998.

Malham D. *Homogeneous And Nonhomogeneous Surround Sound Systems*. York University (online). Available from:
http://www.york.ac.uk/inst/mustech/3d_audio/homogeneous.htm (Accessed 17 May 2004).

Morton D. *Dead Mediums: Quadraphonics* (online). Available from:
http://www.deadmedia.org/notes/30/309.html (Accessed 18 February 2002).

Philipson P. (1998) *An Investigation Into Using Ambisonics As A Surround Sound Tool In Small Project Music Studios*. Unpublished MSc thesis, University of York.

Robjohns H. (2001) *You Are Surrounded. Surround Sound Explained Part 1* (online), Sound On Sound (August 2001). Available from: http://www.sospubs.co.uk/sos/aug01/articles/surroundsound1.asp (Accessed 18 May 2004).

Runstein R.E. (1976) *Modern Recording Techniques.* Howard W. Sams & Co. Inc., Indianapolis, USA 1976.

Taylor D. (1995) *Real-Time MIDI And Graphical Ambisonics Inteface*. Unpublished MSc thesis, University of York.

Whiting J. *Ambisonics Is Dead. Long Live Ambisonics.* York University (online). Available from: http://www.york.ac.uk/inst/mustech/3d_audio/whiteart.htm (Accessed 18 May 2004).

10. Glossary

3D: Abbreviation for “three-dimensional.”

Audio Units: A plug-in format developed by Apple as part of the CoreAudio sound engine for their OS X operating system.

CD: (Compact Disc) Storage medium developed by Sony and Philips in the early 1980s. Allows for storage of audio, video, data and many more formats. Currently is the most popular format of audio distribution.

Digital Performer: Audio and MIDI sequencing computer software developed by a company called MOTU (Mark Of The Unicorn).

DVD: (Digital Versatile Disc) A new media for the distribution of data, audio, film, images and many more formats. It allows for multi-channel audio transmission at high resolution and bit depth.

LFE: (Low Frequency Effects) A band limited channel usually available in surround sound systems. Used for low frequency effects in music and more commonly movies. Represents the “0.1” of systems such as “5.1”, indicating that is not a full channel, but a band limited one.

Localization: The action of sensing the origin and direction of sound.

Loudspeaker: Device acting as a transducer, converting electric energy into acoustical energy. More commonly know as “speaker”.

OS X: An operating system developed by Apple for their Macintosh computers.

GUI: (Graphical User Interface) A visual interface for controlling computer software usually with the aide of hardware such as a keyboard and/or mouse.

HRTF: (Heard Related Transfer Function) The frequency response of an ear for a sound at a particular azimuth and elevation. Is unique for every ear as it is affected by the ear and upper body shape and size. Can be applied in a binaural scenario to a signal to make the sound source appear to be in the position of the corresponding HRTF.

Periphonic: Surround sound that is truly three-dimensional (including height).

Phase: "The degree of progression in the cycle of a wave, where one complete cycle is 360º. Waveforms can be added by summing their signed amplitudes at each instant of time. A cycle can begin at any point on a waveform having the same or different frequency and peak levels to have different amplitudes at any on point in time. These waves are said to be "out of phase" with respect to each other. Phase is measured in degrees of a cycle (divided into 360º) and will result in audible variations of a combined signal's amplitude and overall frequency response (Huber & Runstein, 1995:478)."

Plug-ins: Digital signal processing software components which act as extensions to stand-alone host software. Plug-ins can offer effects such as equalization, compression, amongst other more specialised applications.

Pro Tools: Audio and MIDI sequencing computer software developed by a company called Digidesign.

Quad: Configuration and system of audio playback where the listener is surround by four loudspeakers at 90º from each other. Also know as Quadraphonics, Quadrophonics, Quadrisonics, and Quadrasonics.

SACD: (Super Audio Compact Disc): High quality media developed by Sony and Philips, initially intended for audio storage. Provides multi-channel capabilities and sampling rates up to 64 times of the earlier CD.

Speaker Array: Configuration of loudspeakers positioning. Shape of listening area set by the shape of the loudspeaker boundary.

Surround Sound: Audio system with three or more loudspeakers placed around the listener in an attempt to create a sense of sound arriving from all directions.

Sweet Spot: The ideal listening position in an audio playback environment, usually in the centre of a listening area.

VST: (Virtual Studio Technology); A format for audio software including plug-ins which was developed by a company named Steinberg.

12. Appendices

a. AU Validation Results

i. B-Processor

```
--------------------------------------------------
VALIDATING AUDIO UNIT: 'aufx' - 'Proc' - 'Dige'
Manufacturer String: Digenis
AudioUnit name: B-Processor
Component Info: Digenis: Ambisonic B-Format Processor
--------------------------------------------------
TESTING OPEN TIMES:
COLD:
time to open AudioUnit:         4.587ms
WARM:
time to open AudioUnit:         0.075ms

AU Component Version: 2.0.0 (0x20000)
--------------------------------------------------
VERIFYING DEFAULT SCOPE FORMATS:
Input Scope Bus Configuration:
 Default Bus Count:1
    Default Format: AudioStreamBasicDescription: 16 ch,  44100 Hz, 'lpcm' (0x0000002B)
32-bit big-endian float, deinterleaved

Output Scope Bus Configuration:
 Default Bus Count:1
    Default Format: AudioStreamBasicDescription: 16 ch,  44100 Hz, 'lpcm' (0x0000002B)
32-bit big-endian float, deinterleaved

* * PASS
--------------------------------------------------
VERIFYING REQUIRED PROPERTIES:
  VERIFYING PROPERTY: Sample Rate
    PASS
  VERIFYING PROPERTY: Stream Format
    PASS
  VERIFYING PROPERTY: Maximum Frames Per Slice
    PASS
  VERIFYING PROPERTY: Last Render Error
    PASS

* * PASS
--------------------------------------------------
VERIFYING RECOMMENDED PROPERTIES:
  VERIFYING PROPERTY: Latency
    PASS
  VERIFYING PROPERTY: Tail Time
WARNING: Recommended Property is not supported

  VERIFYING PROPERTY: Bypass Effect
```

```
   PASS

* * PASS
------------------------------------------------
VERIFYING OPTIONAL PROPERTIES:
  VERIFYING PROPERTY Supported Number of Channels
    PASS
  VERIFYING PROPERTY Host Callbacks
    PASS

* * PASS
------------------------------------------------
VERIFYING SPECIAL PROPERTIES:
UIComponents available: 0

VERIFYING CLASS INFO
   PASS

* * PASS
------------------------------------------------
PUBLISHED PARAMETER INFO:

# # 5 Global Scope Parameters:
Parameter ID:0
Name: Ambisonic Order
Parameter Type: Indexed
Values: Minimum = 0.000000, Default = 0.000000, Maximum = 1.000000
Flags: Readable, Writable
Parameter has Value Strings
Num Strings = 2
Value: 0,  String: 1st
Value: 1,  String: Up to 3rd - Rotation only
  -parameter PASS

Parameter ID:1
Name: Manipulation Order
Parameter Type: Indexed
Values: Minimum = 0.000000, Default = 0.000000, Maximum = 5.000000
Flags: Readable, Writable
Parameter has Value Strings
Num Strings = 6
Value: 0,  String: Rotate, Tilt, Tumble
Value: 1,  String: Rotate, Tumble, Tilt
Value: 2,  String: Tilt, Rotate, Tumble
Value: 3,  String: Tilt, Tumble, Rotate
Value: 4,  String: Tumble, Rotate, Tilt
Value: 5,  String: Tumble, Tilt, Rotate
  -parameter PASS

Parameter ID:2
Name: Rotate
Parameter Type: Degrees
```

Values: Minimum = -180.000000, Default = 0.000000, Maximum = 180.000000
Flags: Readable, Writable
  -parameter PASS

Parameter ID:3
Name: Tilt
Parameter Type: Degrees
Values: Minimum = -180.000000, Default = 0.000000, Maximum = 180.000000
Flags: Readable, Writable
  -parameter PASS

Parameter ID:4
Name: Tumble
Parameter Type: Degrees
Values: Minimum = -180.000000, Default = 0.000000, Maximum = 180.000000
Flags: Readable, Writable
  -parameter PASS

Testing that parameters retain value across reset and initialization
  PASS

* * PASS

FORMAT TESTS:

Input/Output Channel Handling:
1-1 1-2 1-4 1-5 2-2 2-4 2-5 4-4 4-5 5-5 5-2 6-6 8-8

* * PASS

RENDER TESTS:
Input Format: AudioStreamBasicDescription: 16 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
Output Format: AudioStreamBasicDescription: 16 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
Render Test at 512 frames
Render Test at 64
Render Test at 128
Render Test at 137
Render Test at 4096

Checking connection semantics:
Connection format:
from: AudioStreamBasicDescription: 16 ch, 48000 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
to: AudioStreamBasicDescription: 16 ch, 48000 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
  PASS

Checking parameter setting
  PASS

* * PASS

AU VALIDATION SUCCEEDED.
---

ii. B-Decoder

VALIDATING AUDIO UNIT: 'aufx' - 'Deco' - 'Dige'
Manufacturer String: Digenis
AudioUnit name: B-Decoder
Component Info: Digenis: Ambisonic B-Format Decoder

TESTING OPEN TIMES:
COLD:
time to open AudioUnit: 39.180ms
WARM:
time to open AudioUnit: 3.513ms

AU Component Version: 2.0.0 (0x20000)

VERIFYING DEFAULT SCOPE FORMATS:
Input Scope Bus Configuration:
Default Bus Count:1
Default Format: AudioStreamBasicDescription: 16 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved

Output Scope Bus Configuration:
Default Bus Count:1
Default Format: AudioStreamBasicDescription: 16 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved

* * PASS

VERIFYING REQUIRED PROPERTIES:
VERIFYING PROPERTY: Sample Rate
PASS
VERIFYING PROPERTY: Stream Format
PASS
VERIFYING PROPERTY: Maximum Frames Per Slice
PASS
VERIFYING PROPERTY: Last Render Error
PASS

* * PASS

VERIFYING RECOMMENDED PROPERTIES:
VERIFYING PROPERTY: Latency
PASS
VERIFYING PROPERTY: Tail Time
WARNING: Recommended Property is not supported

```
  VERIFYING PROPERTY: Bypass Effect
    PASS

* * PASS
------------------------------------------------
VERIFYING OPTIONAL PROPERTIES:
  VERIFYING PROPERTY Supported Number of Channels
    PASS
  VERIFYING PROPERTY Host Callbacks
    PASS

* * PASS
------------------------------------------------
VERIFYING SPECIAL PROPERTIES:
UIComponents available: 0

DEFAULT PRESET: 2, Name: Quad

HAS FACTORY PRESETS
    ID:   0    Name: Mono
    ID:   1    Name: Stereo
    ID:   2    Name: Quad
    ID:   3    Name: Pentagon
    ID:   4    Name: 5.0
    ID:   5    Name: Hexagon
    ID:   6    Name: 7.0
    ID:   7    Name: Octagon
    ID:   8    Name: Decadron
    ID:   9    Name: Dodecadron
    ID:  10    Name: Horizontal 16
    ID:  11    Name: 3D 8 - Cube
    ID:  12    Name: 3D 12
    ID:  13    Name: 3D 16

VERIFYING CLASS INFO
    PASS

* * PASS
------------------------------------------------
PUBLISHED PARAMETER INFO:

# # 16 Global Scope Parameters:
Parameter ID:0
Name: Ambisonic Order
Parameter Type: Indexed
Values: Minimum = 0.000000, Default = 3.000000, Maximum = 3.000000
Flags: Global Meta, Readable, Writable
Parameter has Value Strings
Num Strings = 4
Value: 0,  String: Zeroth
Value: 1,  String: First
```

Value: 2, String: Second
Value: 3, String: Third
 -parameter PASS

Parameter ID:1
Name: Global Distance
Parameter Type: Meters
Values: Minimum = 0.000000, Default = 5.000000, Maximum = 10.000000
Flags: Global Meta, Readable, Writable
 -parameter PASS

Parameter ID:2
Name: Global Amplitude
Parameter Type: Decibels
Values: Minimum = -,àû, Default = -3.010300, Maximum = 0.000000
Flags: Values Have Strings, Global Meta, Readable, Writable
 -parameter PASS

Parameter ID:3
Name: Global Zeroth Base
Parameter Type: Percent
Values: Minimum = 0.000000, Default = 70.700005, Maximum = 100.000000
Flags: Global Meta, Readable, Writable
 -parameter PASS

Parameter ID:4
Name: Global First Base
Parameter Type: Percent
Values: Minimum = 0.000000, Default = 75.000000, Maximum = 100.000000
Flags: Global Meta, Readable, Writable
 -parameter PASS

Parameter ID:5
Name: Global Second Base
Parameter Type: Percent
Values: Minimum = 0.000000, Default = 50.000000, Maximum = 100.000000
Flags: Global Meta, Readable, Writable
 -parameter PASS

Parameter ID:6
Name: Global Third Base
Parameter Type: Percent
Values: Minimum = 0.000000, Default = 30.000002, Maximum = 100.000000
Flags: Global Meta, Readable, Writable
 -parameter PASS

Parameter ID:7
Name: Speaker #
Parameter Type: Indexed
Values: Minimum = 0.000000, Default = 0.000000, Maximum = 15.000000
Flags: Global Meta, Readable, Writable
Parameter has Value Strings

Num Strings = 16
Value: 0, String: 1
Value: 1, String: 2
Value: 2, String: 3
Value: 3, String: 4
Value: 4, String: 5
Value: 5, String: 6
Value: 6, String: 7
Value: 7, String: 8
Value: 8, String: 9
Value: 9, String: 10
Value: 10, String: 11
Value: 11, String: 12
Value: 12, String: 13
Value: 13, String: 14
Value: 14, String: 15
Value: 15, String: 16
  -parameter PASS

Parameter ID:8
Name: Azimuth
Parameter Type: Degrees
Values: Minimum = -180.000000, Default = 0.000000, Maximum = 180.000000
Flags: Readable, Writable
  -parameter PASS

Parameter ID:9
Name: Elevation
Parameter Type: Degrees
Values: Minimum = -90.000000, Default = 0.000000, Maximum = 90.000000
Flags: Readable, Writable
  -parameter PASS

Parameter ID:10
Name: Distance
Parameter Type: Meters
Values: Minimum = 0.000000, Default = 5.000000, Maximum = 10.000000
Flags: Global Meta, Readable, Writable
  -parameter PASS

Parameter ID:11
Name: Amplitude
Parameter Type: Decibels
Values: Minimum = -,àû, Default = -3.010300, Maximum = 0.000000
Flags: Values Have Strings, Readable, Writable
  -parameter PASS

Parameter ID:12
Name: Zeroth Base
Parameter Type: Percent
Values: Minimum = 0.000000, Default = 70.700005, Maximum = 100.000000
Flags: Readable, Writable

-parameter PASS

Parameter ID:13
Name: First Base
Parameter Type: Percent
Values: Minimum = 0.000000, Default = 75.000000, Maximum = 100.000000
Flags: Readable, Writable
-parameter PASS

Parameter ID:14
Name: Second Base
Parameter Type: Percent
Values: Minimum = 0.000000, Default = 50.000000, Maximum = 100.000000
Flags: Readable, Writable
-parameter PASS

Parameter ID:15
Name: Third Base
Parameter Type: Percent
Values: Minimum = 0.000000, Default = 30.000002, Maximum = 100.000000
Flags: Readable, Writable
-parameter PASS

Testing that parameters retain value across reset and initialization
PASS

* * PASS

FORMAT TESTS:

Input/Output Channel Handling:
1-1 1-2 1-4 1-5 2-2 2-4 2-5 4-4 4-5 5-5 5-2 6-6 8-8

* * PASS

RENDER TESTS:
Input Format: AudioStreamBasicDescription: 16 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
Output Format: AudioStreamBasicDescription: 16 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
Render Test at 512 frames
Render Test at 64
Render Test at 128
Render Test at 137
Render Test at 4096

Checking connection semantics:
Connection format:
from: AudioStreamBasicDescription: 16 ch, 48000 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved

to: AudioStreamBasicDescription: 16 ch, 48000 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
  PASS

Checking parameter setting
  PASS

* * PASS

AU VALIDATION SUCCEEDED.
---

iii. B-Binaural

VALIDATING AUDIO UNIT: 'aufx' - 'Bina' - 'Dige'
Manufacturer String: Digenis
AudioUnit name: B-Binaural
Component Info: Digenis: Ambisonic B-Format To Binaural

TESTING OPEN TIMES:
COLD:
time to open AudioUnit: 274.510ms
WARM:
time to open AudioUnit: 15.012ms

AU Component Version: 2.0.0 (0x20000)

VERIFYING DEFAULT SCOPE FORMATS:
Input Scope Bus Configuration:
 Default Bus Count:1
    Default Format: AudioStreamBasicDescription: 16 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved

Output Scope Bus Configuration:
 Default Bus Count:1
    Default Format: AudioStreamBasicDescription: 2 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved

* * PASS

VERIFYING REQUIRED PROPERTIES:
  VERIFYING PROPERTY: Sample Rate
    PASS
  VERIFYING PROPERTY: Stream Format
    PASS
  VERIFYING PROPERTY: Maximum Frames Per Slice
    PASS
  VERIFYING PROPERTY: Last Render Error
    PASS

* * PASS

```
------------------------------------------------
VERIFYING RECOMMENDED PROPERTIES:
  VERIFYING PROPERTY: Latency
    PASS
  VERIFYING PROPERTY: Tail Time
WARNING: Recommended Property is not supported

  VERIFYING PROPERTY: Bypass Effect
    PASS

* * PASS
------------------------------------------------
VERIFYING OPTIONAL PROPERTIES:
  VERIFYING PROPERTY Supported Number of Channels
    PASS
  VERIFYING PROPERTY Host Callbacks
    PASS

* * PASS
------------------------------------------------
VERIFYING SPECIAL PROPERTIES:
UIComponents available: 0

DEFAULT PRESET: 0, Name: Quad

HAS FACTORY PRESETS
    ID:   0    Name: Quad
    ID:   1    Name: Octagon
    ID:   2    Name: 3D 8
    ID:   3    Name: 3D 16

VERIFYING CLASS INFO
    PASS

* * PASS
------------------------------------------------
PUBLISHED PARAMETER INFO:

# # 3 Global Scope Parameters:
Parameter ID:0
Name: Ambisonic Order
Parameter Type: Indexed
Values: Minimum = 0.000000, Default = 1.000000, Maximum = 3.000000
Flags: Readable, Writable
Parameter has Value Strings
Num Strings = 4
Value: 0,  String: Zeroth
Value: 1,  String: First
Value: 2,  String: Second
Value: 3,  String: Third
  -parameter PASS
```

Parameter ID:1
Name: Ear Type
Parameter Type: Indexed
Values: Minimum = 0.000000, Default = 0.000000, Maximum = 1.000000
Flags: Readable, Writable
Parameter has Value Strings
Num Strings = 2
Value: 0, String: Normal
Value: 1, String: Large
  -parameter PASS

Parameter ID:2
Name: Amplitude
Parameter Type: Decibels
Values: Minimum = -,àû, Default = 0.000000, Maximum = 6.000000
Flags: Values Have Strings, Readable, Writable
  -parameter PASS

Testing that parameters retain value across reset and initialization
  PASS

* * PASS

FORMAT TESTS:

Input/Output Channel Handling:
1-1 1-2 1-4 1-5 2-2 2-4 2-5 4-4 4-5 5-5 5-2 6-6 8-8

* * PASS

RENDER TESTS:
Input Format: AudioStreamBasicDescription: 16 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
Output Format: AudioStreamBasicDescription: 2 ch, 44100 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
Render Test at 512 frames
Render Test at 64
Render Test at 128
Render Test at 137
Render Test at 4096

Checking connection semantics:
Connection format:
from: AudioStreamBasicDescription: 16 ch, 48000 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
to: AudioStreamBasicDescription: 2 ch, 48000 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
  PASS

Checking parameter setting
  PASS

* * PASS

AU VALIDATION SUCCEEDED.
---

b. Relevant Correspondence

i. *CoreAudio Mailing List*

Aristotel Digenis:
Hello, this is not a question about AudioUnits (more of a programming/compile errors) but I have noticed in the archives that people have used vBigDSP before. I am trying to add Apple's vBigDSP to my program. The vBigDSP has the following:

```
#include <errors.h>
#include <math.h>
#include <MacMemory.h>

#include "vBigDSP.h"
```

When I try to compile it, it cannot find errors.h and MacMemory.h

I searched the finder and noticed that there is no "errors.h" but there is "Errors.h". I tried to change the include line in the code to "Errors.h" but it still gives me the error: "vBigDSP.c:72:20: Errors.h: No such file or directory"

The file MacMemory.h also is found in the finder but the compiler cannot find it.

In fact there are several copies of both MacMemory.h and errors.h in different directories on the computer.

Has anybody else had this problem? And if so, how did you go about fixing it?

I should also say....funny thing is that the code also includes <AltiVec.h>. The compiler doesn't give me problems about it. But...the file does not exist on the hard drive! How does it give me a compiler error for the files that do exist and no compiler error for the files that don't exist?

Thank you in advance.

John:
Hi Aristotel,

You might want to consider using Apple's Accelerate framework (formerly veclib) instead, specifically vDSP. I was using the vBigDSP for a while since it was easier to use (at least for me), but then I switched to veclib and the performance increase was tremendous.

Check this link, there's some helpful sample code:

http://developer.apple.com/hardware/ve/vector_libraries.html

As for your error, it sounds like you are using Codewarrior? With xcode, just adding the Carbon framework should work. With CW, you might need to check your search paths and make modifications as necessary.

John

Aristotel Digenis:
Hello John,

Thanks for the suggestion to use Accelerate framework. I saw it before, but I assumed the vBigDSP would be faster. Perhaps not, I will try it though.

Regarding my error with the vBigDSP, I am using xCode actually. I tried to include the Carbon framework but that did not solve the problem. Thinking about it, I don't think it would, because neither the "errors.h" or "MacMemory." which the compiler cannot find, as not in the Carbon Framework. "MacMemory.h" is in the CoreServices framework, which is in my xCode project already (which is why its wierd that it doesn't find it during compilation).

There is "errors.h" in the Kernel framework, but even when I include that to my xCode project, the compiler cannot find "errors.h". Any other suggestions as to what I could possibly be doing wrong?

Thanks once again!

Stephen Davis:
The sample code is old. To fix it, you can change most of the #include references to just be:

```
#include <CoreServices/CoreServices.h>
```

and add the CoreServices framework to your application.

Using the Accelerate framework is a better option, as others have noted.

stephen

ii. Music DSP Mailing List

Aristotel Digenis:
Greetings!

I have just joined the mailing list. I have spent ages trying to figure the following problem. I wish I had found this mailing list sooner. I am trying to do FFT Convolution, however I am getting very frequent clicks and pops which result to a buzzing. I am implementing an overlap-add so I cannot see why it does it. I thought that perhaps it is a result of the signals ending at the end of the frame block at a non-zero crossing or one that does not join well with the signal of the next frame block.

I tried Hann windowing each frame block before the FFT, and the clicks do not occur then. So it would appear the edges of frame blocks do not join well after convolution. However, the resulting signal "wobbles" as the amplitude is rapidly changing between frame blocks. I am aware that this testing did not include windowing for windowed material. It seems very complicated to implement that, and would like to avoid it if I can. The question is, for what I

am doing, do I need to use Windowing? It apparently is not needed according to some of the mails in the archive and other sources(such as chapter 18 of The Scientist and Engineer's Guide to Digital Signal Processing).

I should mention that there are absolutely not artifacts for the whole process when the spectrum of the signal and the impulse response are not multiplied.

What could be causing the artifacts at the end of each convolved frame block?

Bellow is pseudo code of what I am trying to do.

-I have an impulse response of 512 samples which I zero-pad to 1024.
-FFT the impulse response.
-Zero pad an array of 511 spaces. This will later be used to store the overlap.
During the processing loop.
-Read 512 samples of audio.
-Zero pad the array to 1024.
-FFT the audio signal.
-Multiply the spectrum of the audio signal with that of the impulse response.
-Inverse FFT of the result.
-Add the overlap to the result.
-Update the overlap.
-Output the audio.
Repeat loop.

Thank you in advance!

Aristotel

Daniele Terdina:
You are right: convolution through overlap-add requires no windowing.

The procedure you are following seems correct. Only thing, as your impulse response is 512 samples and your overlap buffer is (correctly) 511 bytes, I think you should read and output chunks of audio that are 513 samples long instead of 512. (Not sure this might cause a click, though).
After each inverse FFT, you add the 511 samples from the overlap buffer to the first 511 samples of the inverse FFT result, output the first 513 samples of the result and copy the remaining 511 to the overlap buffer for the next iteration.

Other than that, if you still hear clicks my only guess is an error in your FFT or spectrum multiplication routines (probably the latter, as you say that you hear no clicks if you don't perform the spectrum multiplication.).

Also notice that if your audio files consist of integer samples (but you must be using floating point at least for all your internal calculations), the (floating point) convolution result may need to be scaled down before converting to integers again (or before sending the

audio to the sound card) to avoid clipping.

Finally, and this has nothing to do with the clicks, your convolution may be faster if you use more than 1024 samples for your FFTs. You need to experiment to find the best value for your implementation, but 4096 sample FFTs seems a good starting guess to convolve an impulse response of 512 samples.

Daniele

Sampo Syreeni:
No. If the impulse response stays constant, you do not need or even want to window.

This might seem obvious, but... Are you certain you're doing complex multiplication, and not just a real one? If you're not, I think this is one thing which could cause the symptoms you describe.

It seems correct to me. Feel free to post detailed code if you want aid in debugging it.

Aristotel Digenis:
Hello and thank you for your quick replies!

I cannot read more than 512 samples (limitation of the program structure). The rest, I am doing just like you suggest. The audio samples are floats, and the processing is done as doubles.

Regarding the spectrum multiplication. I am using Apple's vDSP library's function zvmulD(). In the documentation it is described as "Complex Vector Multiply." On of the function arguments is the "conjugate." The documentation says "Assign conjugate flag 'conjugate' a value of 1 for normal multiplication or -1 for multiplication by conjugated values of input 1." I have tried both, and both give different waveforms and both still have the clicks/pops. I am new to FFTs and DSP in general so is it normal multiplication I need for my scenario or multiplication by conjugated values of input 1? While both give clicks and pops, perhaps this may shed some light on the overall problem with the pops in the signal. I have included the actual C++ code of the process just in case anybody notices a mistake (but I doubt it as I have been going over this code for over two weeks now).

Thank you in advance!!

Aristotel

```
// Store 512 incoming audio samples into array
for(UInt32 j = 0; j < 512; j++)
{
   speakerTempBuffer[j] = audioData[j];
}
// Zero-pad the second half of the array
for(int a = 512; a < 1024; a++)
{
   speakerTempBuffer[a] = 0.0;
}
// Convert to Split Double Complex
ctozD((DOUBLE_COMPLEX *) speakerTempBuffer, 2, &speakersSplitDoubleComplex,
1, nOver2);
```

```
// FFT
fft_zripD(fft1024Setup, &speakersSplitDoubleComplex, stride, log2n,
kFFTDirection_Forward);
// Multiplication of the spectrum with the spectrum of an HRTF impulse
response
zvmulD(&speakersSplitDoubleComplex, stride,
&HRTF_0Degree0ElevationSplitDoubleComplex, stride,
&speakersSplitDoubleComplex, stride, log2n, 1);
// Inverse FFT the result
fft_zripD(fft1024Setup, &speakersSplitDoubleComplex, stride, log2n,
kFFTDirection_Inverse);
// Scale the result
vsmulD(speakersSplitDoubleComplex.realp, 1, &scale,
speakersSplitDoubleComplex.realp, 1, nOver2);
vsmulD(speakersSplitDoubleComplex.imagp, 1, &scale,
speakersSplitDoubleComplex.imagp, 1, nOver2);
// Convert to real array
ztocD(&speakersSplitDoubleComplex, 1, (DOUBLE_COMPLEX *) speakerTempBuffer,
2, nOver2);
// Add over lap
for(int a = 0; a < overlapSize; a++)
{
   speakerTempBuffer[a] += overLap[a];
}
// Update over laps
for(int a = 512; a < 1023; a++)
{
   overLap[a - 512] = speakerTempBuffer[a];
}
// Store result to ouput stream
for(UInt32 j = 0; j < 512; j++)
{
   audioData[j] = (float) speakerTempBuffer[j];
}
```

Daniele Terdina:
I don't know what parameters are required by the functions of the library you are using, but the most suspicious thing to me is that you pass log2n to vsmulD. If vsmulD is a generic array multiplication function and log2n is the log2 of nOver2, then I would expect that you should pass nOver2 or 1024 instead of log2n to vsmulD.

I would also check that vsmulD can accept the same array passed in as both source and destination.

The casts to DOUBLE_COMPLEX * may also be cause of trouble. You should not need casts if you declare all vars and arrays of the proper type. Is speakerTempBuffer complex or real? And how is DOUBLE_COMPLEX defined?

Aristotel Digenis:
You mean zvmuldD() I am assuming. I have just tried that out and YES. It now works. The size parameter should be nOver2 and NOT log2n! Its perfect!

Just for further information, in case somebody else has this problem I will answer the following.

Again I assume you mean zvmulD(). Before receiving your email with the above correction, I tried to store the result of the multiplication to a third array. This resulted in clicks/pops as well, but far less than when storing the result on the multiplication to the one of the multiplied

arrays. However with the above correction (regarding nOver2 instead of log2n) it works fine either way. You get the same succesful output signal when saving the result of the multiplication back into one of the arrays being multiplied.

The casting to DOUBLE_COMPLEX is not a problem because with Apple's vDSP library comes example code for using the functions. And this is exactly how they did it. However, the example code did not have an example use of multiplication of complex arrays. If it had, it would have saved me approximately three weeks on my project! speakerTempBuffer is type DOUBLE COMPLEX SPLIT.

Thank you VERY MUCH!

iii. Hydrogen Audio

Aristotel Digenis:
Hello everybody... I am trying to do the following but having troubles. The following is pseudo code.

(This happens once)
-Zero pad a 511 sized array(this will be used for overlap)

(The following lines loop)
-Capture 512 samples of an audio signal and store them in the first half of a 1024 size array
-Zero pad the remaining half of the array
-Window the first 512 samples with Hanning
-Convert the entire array to the frequency domain with FFT
-Multiply the contents with the frequency of an Impulse response
-Inverse FFT to get the filtered audio signal
-Add the overlap to the filtered signal
-Strore last 511 samples of the filtered signal to the overlap array
-Output the filtered signal
-Go back to capturing 512 samples etc...

The result is a filtered signal, with no clicks or pops (thanks to the windowing), but there are rapid amplitude chages. I have read just about every note I could find on the web, gone though PDF books and guides...but have not quite understood how many windowing do I do? Do I do another windowing operation somewhere? There are mentions online in places about how its done, but some just suggest it too briefly, and some go into some deep DSP theories which I admit is not one of my strong points. Could somebody who has an understanding on this topic help me out? Pseudo code or c/C++ outlines would be great.

Thanks a lot in advance!

PS: This is for a suit of plug-ins I am writing, whcih will be freely available at the end of the month. I will let you know more about them and where you can download them. But first I need to make sure they work

Wkwai:
I think you should try this approach :

Construct an array of N samples with N/2 samples shifted from previous frame.. DO NOT zero pad any samples !!

Windowed the array with a suitable window..

Pass the windowed array into a FFT

At the reconstruction side, the IFFT would reproduce the same windowed array.. Overlapped and add with the previous half window..

eg :
w(n)S_2(n) + w(N/2 + n)S_1(N/2 +n) = S_Original(n)..

Since that S_2(n) = S_1(N/2 +n) = S_Original(n) ;

then the windowing function MUST obey this equation for perfect reconstruction.. :

w(n) + w(N/2 +n) = 1

One possible window solution is the hanning (or was it hanming ???) window given by the following equation :

w(n) = 0.5*(1.0 - cos((2*PI * (n+0.5) / N))) /// Sorry for the mistake !!!

SebastianG:
It seems like you want to do plain filtering by "fast convolution" (via FFT).
There's NO need to apply a window function !

Let 'k' be the length of the impulse response of the filter.
Now, if you want an infinite signal to be convolved with this impulse response you can do this block-wise and do some overlap adding. This is legal because of the properties of the filter (being a time-invariant linear system)

Let 'x' be your signal you want to be convolved with the filter's IR.
Let 'n' be the smallest power of 2 which is graeter than or equal to k-1.

You can take blocks of n samples of your signal, convolve them seperately with the impulse response and do the overlap-add of the resulting signal blocks which have at maximum n+k-1 non-zero-samples (due to the convolving).
because of k-1<=n we know n+k-1<=2n. So, the length in samples of the convolved signal blocks are 2n at maximum. This convolution can be done via the FFT of size 2n.

Here's an example:
impulse response (k=5 samples):
|i i i i i|

your signal you want to convolve with the IR divided into blocks of length n=8:
|a a a a a a a a|b b b b b b b b|c c c c c c c c|...

you take each block of n samples pad them with zeros to 2n samples like this:
|a a a a a a a a 0 0 0 0 0 0 0 0|
and convolve it with the zero-padded impulse response:
|i i i i i 0 0 0 0 0 0 0 0 0 0 0|
to get

|a'a'a'a'a'a'a'a'a'a'0 0 0 0|
which can be done via fast Fourier-transforming both blocks, multiplication of their spectra (you better do this with the complex valued FFT and complex multiplication) and inverse transforming the result.

Now the overlap-add part:
|a'a'a'a'a'a'a'a'a'a'0 0 0 0|
+ |b'b'b'b'b'b'b'b'b'b'0 0 0 0|
+ |c'c'c'c'c'c'c'c'c'c'0 0 0 0|

The zero-padding is actually needed because the convolution you can calculate via FFT is a cyclic one and our source signal is NOT periodic.

HTH,
Sebastian

SebastianG:
QUOTE(wkwai @ Aug 14 2004, 11:28 PM)
I think you should try this approach :
[...]

(1) It's called "Hann" window. See Hann and Hamming Window

(2) Your approach is flawed and introduces temporal alias artefacts. It does not take into account that FFT convolution is cyclic. (You need zero padding to compensate for this)

Sebastian

Aristotel Digenis:
hank you for your replies and suggestions. I got rid of any windowing and taken Sebastian's sugegstion. I have been advised by others to do just that and aparrently it works! I have adjusted my code to the sugegsted method (which actually is what it was before I got carried away with the Windowing theory). However I get clicks/pops at the end of each frame block processed. This results in buzzing. I have added the code of my processing with comments bellow in case anybody can spot a problem.

Regarding the spectrum multiplication. I am using Apple's vDSP library's function zvmulD(). In the documentation it is described as "Complex Vector Multiply." On of the function arguments is the "conjugate." The documentation says "Assign conjugate flag 'conjugate' a value of 1 for normal multiplication or -1 for multiplication by conjugated values of input 1." I have tried both, and both give different waveforms and both still have the clicks/pops. I am new to FFTs and DSP in general so is it normal multiplication I need for my scenario or multiplication by conjugated values of input 1? While both give clicks and pops, perhaps this may shed some light on the overall problem with the pops in the signal.

Thanks a lot!

CODE

```
// Store 512 incoming audio samples into array
for(UInt32 j = 0; j < 512; j++)
{
  speakerTempBuffer[j] = audioData[j];
}
// Zero-pad the second half of the array
for(int a = 512; a < 1024; a++)
{
  speakerTempBuffer[a] = 0.0;
}
// Convert to Split Double Complex
ctozD((DOUBLE_COMPLEX *) speakerTempBuffer, 2, &speakersSplitDoubleComplex,
1, nOver2);
// FFT
fft_zripD(fft1024Setup, &speakersSplitDoubleComplex, stride, log2n,
kFFTDirection_Forward);
// Multiplication of the spectrum with the spectrum of an HRTF impulse
response
zvmulD(&speakersSplitDoubleComplex, stride,
&HRTF_0Degree0ElevationSplitDoubleComplex, stride,
&speakersSplitDoubleComplex, stride, log2n, 1);
// Inverse FFT the result
fft_zripD(fft1024Setup, &speakersSplitDoubleComplex, stride, log2n,
kFFTDirection_Inverse);
// Scale the result
vsmulD(speakersSplitDoubleComplex.realp, 1, &scale,
speakersSplitDoubleComplex.realp, 1, nOver2);
vsmulD(speakersSplitDoubleComplex.imagp, 1, &scale,
speakersSplitDoubleComplex.imagp, 1, nOver2);
// Convert to real array
ztocD(&speakersSplitDoubleComplex, 1, (DOUBLE_COMPLEX *) speakerTempBuffer,
2, nOver2);
// Add over lap
for(int a = 0; a < overlapSize; a++)
{
  speakerTempBuffer[a] += overLap[a];
}
// Update over laps
for(int a = 512; a < 1023; a++)
{
  overLap[a - 512] = speakerTempBuffer[a];
}
// Store result to ouput stream
for(UInt32 j = 0; j < 512; j++)
{
  audioData[j] = (float) speakerTempBuffer[j];
}
```

Aristotel Digenis:
YES! Its solved! The problem was the size of the array that the complex vector multiplication was being done for. It was log2n which in this case equals 10, which it should have been nOver2 which equals 512. I am putting the correct code bellow. It should be useful to newbies like myself who learn DSP better by reading code than text and math formulas.

Thanks for your help!!!

CODE

```
// Store 512 incoming audio samples into array
for(UInt32 j = 0; j < 512; j++)
{
```

```
 speakerTempBuffer[j] = audioData[j];
}
// Zero-pad the second half of the array
for(int a = 512; a < 1024; a++)
{
 speakerTempBuffer[a] = 0.0;
}
// Convert to Split Double Complex
ctozD((DOUBLE_COMPLEX *) speakerTempBuffer, 2, &speakersSplitDoubleComplex,
1, nOver2);
// FFT
fft_zripD(fft1024Setup, &speakersSplitDoubleComplex, stride, log2n,
kFFTDirection_Forward);
// Multiplication of the spectrum with the spectrum of an HRTF impulse
response
zvmulD(&speakersSplitDoubleComplex, stride,
&HRTF_0Degree0ElevationSplitDoubleComplex, stride,
&speakersSplitDoubleComplex, stride, nOver2, 1);
// Inverse FFT the result
fft_zripD(fft1024Setup, &speakersSplitDoubleComplex, stride, log2n,
kFFTDirection_Inverse);
// Scale the result
vsmulD(speakersSplitDoubleComplex.realp, 1, &scale,
speakersSplitDoubleComplex.realp, 1, nOver2);
vsmulD(speakersSplitDoubleComplex.imagp, 1, &scale,
speakersSplitDoubleComplex.imagp, 1, nOver2);
// Convert to real array
ztocD(&speakersSplitDoubleComplex, 1, (DOUBLE_COMPLEX *) speakerTempBuffer,
2, nOver2);
// Add over lap
for(int a = 0; a < overlapSize; a++)
{
 speakerTempBuffer[a] += overLap[a];
}
// Update over laps
for(int a = 512; a < 1023; a++)
{
 overLap[a - 512] = speakerTempBuffer[a];
}
// Store result to ouput stream
for(UInt32 j = 0; j < 512; j++)
{
 audioData[j] = (float) speakerTempBuffer[j];
}
```

iv. Granted Software

Aristotel Digenis:
Hello!

I am memeber of the CoreAudio mailing list, but I thought I would ask you this question directly as it is related to Rax. As you may have seen in the mailing list, I have completed three AudioUnit plug-ins related to Ambisonics. They work fully in Plogue Bidule, but unfortunately they have problems in Rax. When I try to insert one of them I get the following error message:

AudioUnit initialization error!
AudioUnit: B-Binaural failed to initialize: -10868

I click "OK" and the plug-in is added. However when I open the window for the GUI of the plug-in it shows a empty white window. Two of the plug-ins have 16 ins and 16 outs, and the

third plug-in has 16 ins and 2 outs. Does Rax have bus limitations? Perhaps once I register this problem will go away?

I have received emails from people who have downloaded my plug-ins and tried to use them with Rax and had this problem. I am really interested in sovling this problem so more people can use my plug-ins and your Rax together.

Any idea where the problem lies?

Thank you,

Aristotel

Grant:
Hi Aristotel,

Can your plugs be used as stereo in/stereo out? That's how effect plugs get configured in Rax.

Registration won't help btw.

Hope that helps,

Robert.

Aristotel Digenis:
Hello,

The two plug-ins can only be 16in and 16 out and the other plug-in can only be 16in 2out. I suppose this is why they won't work.

Thank you,

Aristotel

Grant:
Hi Aristotel,

Yeah I guess they need a more flexible host.

Best,

Robert.

c. Online Plug-in Feedback Form

Plugin: * --Select one--

Name:

Email:

Location:

Operating System: * --Select one--

Program: * Bidule: Logic Audio: Digital Performer: Peak: Rax: Metro: Sonasphere: Audio Hijack Pro: Intuem: Live: Other: - Please specify:

Program Version: *

Comments:

Send Clear

* = required

d. Source Code

i. B-Processor

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc.h
//  Aristotel Digenis
//  MSc Music Technology University of York
//  August 27th 2004
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#include <AudioUnit/AudioUnitProperties.h>
#include "AUEffectBase.h"
#include "AUPropertiesPostPantherAdditions.h"
#include "BProcVersion.h"
#include "math.h"
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
enum Parameters
{
    kAmbisonicOrder, kManipulationOrder, kRotate,
    kTilt, kTumble, kNumberOfParameters
};

enum AmbisonicOrder
{
    kFirst, kHigherThanFirst
};

enum ManipulationOrder
{
    kRoTiTu, kRoTuTi, kTiRoTu, kTiTuRo, kTuRoTi,
    kTuTiRo, kNumberOfManipulationOrders
};

enum BFormatChannels
{
    kW, kX, kY, kZ,
    kR, kS, kT, kU, kV,
    kK, kL, kM, kN, kO, kP, kQ,
    kNumberOfBFormatSignals
};

const double kPI = 3.141592653589793;
const double k2PI = 2 * kPI;
const double kAverageCoefficient1 = 0.99;
const double kAverageCoefficient2 = 1 - kAverageCoefficient1;
// I/O Configurations
const int kNumberOfInputs = 16;
const int kNumberOfOutputs = 16;
const AUChannelInfo channelInfo[] =
{
    {kNumberOfInputs, kNumberOfOutputs}
};
const AUChannelInfo* channelInfoPointer = channelInfo;
// Parameter Strings
static CFStringRef kAmbisonicOrderStrings[] =
{
    CFSTR("1st"), CFSTR("Up to 3rd - Rotation only")
};
static CFStringRef kManipulationOrderStrings[] =
{
    CFSTR("Rotate, Tilt, Tumble"),
    CFSTR("Rotate, Tumble, Tilt"),
    CFSTR("Tilt, Rotate, Tumble"),
    CFSTR("Tilt, Tumble, Rotate"),
```

```
    CFSTR("Tumble, Rotate, Tilt"),
    CFSTR("Tumble, Tilt, Rotate")
};
static CFStringRef kParameterStrings[] =
{
    CFSTR("Ambisonic Order"), CFSTR("Manipulation Order"),
    CFSTR("Rotate"), CFSTR("Tilt"), CFSTR("Tumble")
};
static bool sLocalized = false;

class BProc : public AUEffectBase
{
public:
    BProc(AudioUnit component);
    virtual ComponentResult GetParameterValueStrings
        (AudioUnitScope inScope,
        AudioUnitParameterID inParameterID,
        CFArrayRef * outStrings);
    virtual ComponentResult GetParameterInfo
        (AudioUnitScope inScope,
        AudioUnitParameterID inParameterID,
        AudioUnitParameterInfo  &outParameterInfo);
    virtual ComponentResult GetPropertyInfo
        (AudioUnitPropertyID inID,
        AudioUnitScope inScope,
        AudioUnitElement inElement,
        UInt32 & outDataSize,
        Boolean & outWritable);
    virtual ComponentResult GetProperty
        (AudioUnitPropertyID inID,
        AudioUnitScope inScope,
        AudioUnitElement inElement,
        void * outData);
    virtual ComponentResult SetParameter
        (AudioUnitParameterID iID,
        AudioUnitScope iScope,
        AudioUnitElement iElem,
        Float32 rValue,
        UInt32 iSchedule);
    virtual UInt32 SupportedNumChannels(const AUChannelInfo**);
    virtual OSStatus ProcessBufferLists
        (AudioUnitRenderActionFlags&,
        const AudioBufferList&,
        AudioBufferList&, UInt32);
    virtual bool SupportsTail () { return false; }
    virtual ComponentResult  Version() { return kBProcVersion; }
private:
    bool higherThanFirstOrder;
    int manipulationOrder;
    float fValue_Rotate;
    float fValue_Tilt;
    float fValue_Tumble;
    double dValue_Rotate;
    double dValue_Tilt;
    double dValue_Tumble;
    // Old Angles
    double dSinRotateAngleOld;
    double dSin2RotateAngleOld;
    double dSin3RotateAngleOld;
    double dSinTiltAngleOld;
    double dSinTumbleAngleOld;
    double dCosRotateAngleOld;
    double dCos2RotateAngleOld;
    double dCos3RotateAngleOld;
    double dCosTiltAngleOld;
```

```
    double dCosTumbleAngleOld;
    // New angles
    double dSinRotateAngle;
    double dSin2RotateAngle;
    double dSin3RotateAngle;
    double dSinTiltAngle;
    double dSinTumbleAngle;
    double dCosRotateAngle;
    double dCos2RotateAngle;
    double dCos3RotateAngle;
    double dCosTiltAngle;
    double dCosTumbleAngle;

    double newW;    //1st order
    double newX;
    double newY;
    double newZ;
    double newS;    //2nd order - only horizontal
    double newT;
    double newU;
    double newV;
    double newL;    //3rd order - only horizontal
    double newM;
    double newN;
    double newO;
    double newP;
    double newQ;
};
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMPONENT_ENTRY(BProc)

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc.cpp
//  Aristotel Digenis
//  MSc Music Technology University of York
//  August 27th 2004
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "BProc.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc::BProc
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BProc::BProc(AudioUnit component) : AUEffectBase(component)
{
    // Set I/O stream configuration
    CreateElements();
    CAStreamBasicDescription streamDescription;
    streamDescription.SetCanonical (kNumberOfInputs, false);
    streamDescription.mSampleRate = 44100;
    GetOutput(0)->SetStreamFormat(streamDescription);
    GetInput(0)->SetStreamFormat(streamDescription);

    higherThanFirstOrder = false;

    if (!sLocalized)
    {
        CFBundleRef bundle = CFBundleGetBundleWithIdentifier
            (CFSTR("com.dige.audiounit.bproc"));

        if (bundle != NULL)
        {
            for(int a = 0; a < kNumberOfParameters; a++)
            {
                kParameterStrings[a] =
```

```
                    CFCopyLocalizedStringFromTableInBundle
                    (kParameterStrings[a], CFSTR("Localizable"),
                    bundle, CFSTR(""));
            }
        }
        sLocalized = true; //so never pass the test again...
    }

    // Initialize the values of the parameters and their doubles copies
    manipulationOrder = kRoTiTu;
    fValue_Rotate = dValue_Rotate = 0.0;
    fValue_Tilt = dValue_Tilt = 0.0;
    fValue_Tumble = dValue_Tumble = 0.0;

    dSinRotateAngleOld = 1.0;
    dSin2RotateAngleOld = 1.0;
    dSin3RotateAngleOld = 1.0;
    dSinTiltAngleOld = 1.0;
    dSinTumbleAngleOld = 1.0;
    dCosRotateAngleOld = 0.0;
    dCos2RotateAngleOld = 0.0;
    dCos3RotateAngleOld = 0.0;
    dCosTiltAngleOld = 0.0;
    dCosTumbleAngleOld = 0.0;
    // Set the parameters
    SetParameter(kAmbisonicOrder, kAudioUnitScope_Global, 0,
        kFirst, 0);
    SetParameter(kManipulationOrder, kAudioUnitScope_Global, 0,
        kRoTiTu, 0);
    SetParameter(kRotate, kAudioUnitScope_Global, 0,
        fValue_Rotate, 0);
    SetParameter(kTilt, kAudioUnitScope_Global, 0,
        fValue_Tilt, 0);
    SetParameter(kTumble, kAudioUnitScope_Global, 0,
        fValue_Tumble, 0);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc::GetParameterInfo
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BProc::GetParameterValueStrings
    (AudioUnitScope inScope,
    AudioUnitParameterID inParameterID,
    CFArrayRef * outStrings)
{
    if(inScope == kAudioUnitScope_Global)
    {
        if(outStrings == NULL)
        {
            return noErr;
        }

        if(inParameterID == kAmbisonicOrder)
        {
            CFStringRef strings[2];

            for(int a = 0; a < 2; a++)
            {
                strings[a] = kAmbisonicOrderStrings[a];
            }

            *outStrings = CFArrayCreate( NULL,
                (const void **)strings, 2, NULL);

            return noErr;
```

```
        }

        if(inParameterID == kManipulationOrder)
        {
            CFStringRef strings[6];

            for(int a = 0; a < 6; a++)
            {
                strings[a] = kManipulationOrderStrings[a];
            }

            *outStrings = CFArrayCreate( NULL,
                (const void **)strings, 6, NULL);

            return noErr;
        }
    }

    return kAudioUnitErr_InvalidProperty;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc::GetParameterInfo
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BProc::GetParameterInfo
    (AudioUnitScope inScope,
    AudioUnitParameterID inParameterID,
    AudioUnitParameterInfo &outParameterInfo)
{
    ComponentResult result = noErr;

    outParameterInfo.flags = kAudioUnitParameterFlag_IsWritable
        | kAudioUnitParameterFlag_IsReadable;

    if (inScope == kAudioUnitScope_Global)
    {
        switch(inParameterID)
        {
            case kAmbisonicOrder:
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[kAmbisonicOrder], false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Indexed;
                outParameterInfo.minValue = 0;
                outParameterInfo.maxValue = 1;
                outParameterInfo.defaultValue = 0;
                break;

            case kManipulationOrder:
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[kManipulationOrder], false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Indexed;
                outParameterInfo.minValue = kRoTiTu;
                outParameterInfo.maxValue = kTuTiRo;
                outParameterInfo.defaultValue = kRoTiTu;
                break;

            case kRotate:
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[kRotate], false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Degrees;
                outParameterInfo.minValue = -180.0;
                outParameterInfo.maxValue = 180.0;
```

```
                outParameterInfo.defaultValue = 0.0;
                break;

            case kTilt:
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[kTilt], false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Degrees;
                outParameterInfo.minValue = -180.0;
                outParameterInfo.maxValue = 180.0;
                outParameterInfo.defaultValue = 0.0;
                break;

            case kTumble:
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[kTumble], false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Degrees;
                outParameterInfo.minValue = -180.0;
                outParameterInfo.maxValue = 180.0;
                outParameterInfo.defaultValue = 0.0;
                break;

            default:
                result = kAudioUnitErr_InvalidParameter;
                break;
        }
    }
    else
    {
        result = kAudioUnitErr_InvalidParameter;
    }

    return result;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc::GetPropertyInfo
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BProc::GetPropertyInfo
    (AudioUnitPropertyID inID,
    AudioUnitScope inScope,
    AudioUnitElement inElement,
    UInt32 & outDataSize,
    Boolean & outWritable)
{
    if (inScope == kAudioUnitScope_Global)
    {
        switch (inID)
        {
            case kAudioUnitProperty_IconLocation:
                outWritable = false;
                outDataSize = sizeof (CFURLRef);
                return noErr;

            case kAudioUnitProperty_ParameterStringFromValue:
                outWritable = false;
                outDataSize = sizeof (AudioUnitParameterStringFromValue);
                return noErr;

            case kAudioUnitProperty_ParameterValueFromString:
                outWritable = false;
                outDataSize = sizeof (AudioUnitParameterValueFromString);
                return noErr;
        }
```

```
    }
    return AUEffectBase::GetPropertyInfo
        (inID, inScope, inElement, outDataSize, outWritable);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc::GetProperty
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BProc::GetProperty
    (AudioUnitPropertyID inID,
    AudioUnitScope inScope,
    AudioUnitElement inElement,
    void * outData)
{
    if (inScope == kAudioUnitScope_Global)
    {
        switch (inID)
        {
            case kAudioUnitProperty_IconLocation:
            {
                CFBundleRef bundle = CFBundleGetBundleWithIdentifier
                    (CFSTR("com.dige.audiounit.bproc"));

                if (bundle == NULL)
                {
                    return fnfErr;
                }

                CFURLRef bundleURL = CFBundleCopyResourceURL
                    (bundle, CFSTR("digenis"), CFSTR("icns"), NULL);

                if (bundleURL == NULL)
                {
                    return fnfErr;
                }

                (*(CFURLRef *)outData) = bundleURL;

                return noErr;
            }
            case kAudioUnitProperty_ParameterValueFromString:
            {
                OSStatus retVal = kAudioUnitErr_InvalidPropertyValue;
                AudioUnitParameterValueFromString &name =
                    *(AudioUnitParameterValueFromString*)outData;

                if (name.inParamID != kTumble)
                {
                    return kAudioUnitErr_InvalidParameter;
                }

                if (name.inString == NULL)
                {
                    return kAudioUnitErr_InvalidPropertyValue;
                }

                return retVal;
            }
            case kAudioUnitProperty_ParameterStringFromValue:
            {
                AudioUnitParameterStringFromValue &name =
                    *(AudioUnitParameterStringFromValue*)outData;

                if (name.inParamID != kTumble)
                {
```

```
                    return kAudioUnitErr_InvalidParameter;
                }

                name.outString = NULL;
                return noErr;
            }

            case kAudioUnitProperty_SupportedNumChannels:
            {
                return noErr;
            }
        }
    }
    return AUEffectBase::GetProperty (inID, inScope, inElement, outData);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc::SetParameter
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BProc::SetParameter
    (AudioUnitParameterID iID,
    AudioUnitScope iScope,
    AudioUnitElement iElem,
    Float32 rValue,
    UInt32 iSchedule)
{
    if (iScope==kAudioUnitScope_Global && iElem==0)
    {
        switch (iID)
        {
            case kAmbisonicOrder:
                higherThanFirstOrder = (int) rValue;
                break;
            case kManipulationOrder:
                manipulationOrder = (int) rValue;
                break;
            case kRotate:
                fValue_Rotate = rValue;
                break;
            case kTilt:
                fValue_Tilt = rValue;
                break;
            case kTumble:
                fValue_Tumble = rValue;
                break;
        }
    }
    // Change 0.0 to 360.0 range to radians
    dValue_Rotate = ((-fValue_Rotate) * k2PI) / 360;
    dValue_Tilt = ((-fValue_Tilt) * k2PI) / 360;
    dValue_Tumble = ((-fValue_Tumble) * k2PI) / 360;

    return AUBase::SetParameter(iID, iScope, iElem, rValue, iSchedule);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc::SupportedNumChannel
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UInt32 BProc::SupportedNumChannels(const AUChannelInfo** outInfo)
{
    UInt32 count = 0;
    for(; channelInfoPointer && (channelInfoPointer[count].inChannels
        != 0 || channelInfoPointer[count].outChannels != 0); count++){}

    if(outInfo)
```

```
    {
        *outInfo = channelInfoPointer;
    }

    return count;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BProc::ProcessBufferLists
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
OSStatus BProc::ProcessBufferLists
    (AudioUnitRenderActionFlags& iFlags,
    const AudioBufferList& inBufferList,
    AudioBufferList& outBufferList,
    UInt32 iFrames)
{
    // Array of pointers, as many as input signals(B-Format).
    float* inAudioData[kNumberOfInputs];
    // Array of pointers, as many as ouput signals(B-Format).
    float* outAudioData[kNumberOfOutputs];
    // Pointers point to incomming audio buffers.
    for(int i = 0; i < kNumberOfInputs; i++)
    {
        inAudioData[i] = (float*) inBufferList.mBuffers[i].mData;
        outAudioData[i] = (float*) outBufferList.mBuffers[i].mData;
    }

    double dTemp[kNumberOfInputs];

    dSinRotateAngle = sin(dValue_Rotate);
    dSinTiltAngle = sin(dValue_Tilt);
    dSinTumbleAngle = sin(dValue_Tumble);
    dCosRotateAngle = cos(dValue_Rotate);
    dCosTiltAngle = cos(dValue_Tilt);
    dCosTumbleAngle = cos(dValue_Tumble);
    dCos2RotateAngle = cos(2 * dValue_Rotate);
    dSin2RotateAngle = sin(2 * dValue_Rotate);
    dCos3RotateAngle = cos(3 * dValue_Rotate);
    dSin3RotateAngle = sin(3 * dValue_Rotate);

    if(!higherThanFirstOrder)
    {
        for(UInt32 i = 0; i < iFrames; i++)
        {
            switch(manipulationOrder)
            {
                case kRoTiTu:
                    dTemp[kY] = (inAudioData[kX][i] * dSinRotateAngleOld)
                        + (inAudioData[kY][i] * dCosRotateAngleOld);
                    dTemp[kX] = (inAudioData[kX][i] * dCosRotateAngleOld)
                        - (inAudioData[kY][i] * dSinRotateAngleOld);
                    dTemp[kZ] = (dTemp[kY] * dSinTiltAngleOld)
                        + (inAudioData[kZ][i] * dCosTiltAngleOld);
                    outAudioData[kX][i] = (dTemp[kX] * dCosTumbleAngleOld)
                        - (dTemp[kZ] * dSinTumbleAngleOld);
                    outAudioData[kY][i] = (dTemp[kY] * dCosTiltAngleOld)
                        - (inAudioData[kZ][i] * dSinTiltAngleOld);
                    outAudioData[kZ][i] = (dTemp[kX] * dSinTumbleAngleOld)
                        + (dTemp[kZ] * dCosTumbleAngleOld);
                    break;
                case kRoTuTi:
                    dTemp[kY] = (inAudioData[kX][i] * dSinRotateAngleOld)
                        + (inAudioData[kY][i] * dCosRotateAngleOld);
                    dTemp[kX] = (inAudioData[kX][i] * dCosRotateAngleOld)
                        - (inAudioData[kY][i] * dSinRotateAngleOld);
```

```
            dTemp[kZ] = (dTemp[kX] * dSinTumbleAngleOld)
                + (inAudioData[kZ][i] * dCosTumbleAngleOld);
            outAudioData[kX][i] = (dTemp[kX] * dCosTumbleAngleOld)
                - (inAudioData[kZ][i] * dSinTumbleAngleOld);
            outAudioData[kY][i] = (dTemp[kY] * dCosTiltAngleOld)
                - (dTemp[kZ] * dSinTiltAngleOld);
            outAudioData[kZ][i] = (dTemp[kY] * dSinTiltAngleOld)
                + (dTemp[kZ] * dCosTiltAngleOld);
            break;
        case kTiRoTu:
            dTemp[kY] = (inAudioData[kY][i] * dCosTiltAngleOld)
                - (inAudioData[kZ][i] * dSinTiltAngleOld);
            dTemp[kZ] = (inAudioData[kY][i] * dSinTiltAngleOld)
                + (inAudioData[kZ][i] * dCosTiltAngleOld);
            dTemp[kX] = (inAudioData[kX][i] * dCosRotateAngleOld)
                - (dTemp[kY] * dSinRotateAngleOld);
            outAudioData[kX][i] = (dTemp[kX] * dCosTumbleAngleOld)
                - (dTemp[kZ] * dSinTumbleAngleOld);
            outAudioData[kY][i] = (inAudioData[kX][i] *
                dSinRotateAngleOld)
                + (dTemp[kY] * dCosRotateAngleOld);
            outAudioData[kZ][i] = (dTemp[kX] * dSinTumbleAngleOld)
                + (dTemp[kZ] * dCosTumbleAngleOld);
            break;
        case kTiTuRo:
            dTemp[kY] = (inAudioData[kY][i] * dCosTiltAngleOld)
                - (inAudioData[kZ][i] * dSinTiltAngleOld);
            dTemp[kZ] = (inAudioData[kY][i] * dSinTiltAngleOld)
                + (inAudioData[kZ][i] * dCosTiltAngleOld);
            dTemp[kX] = (inAudioData[kX][i] * dCosTumbleAngleOld)
                - (dTemp[kZ] * dSinTumbleAngleOld);
            outAudioData[kX][i] = (dTemp[kX] * dCosRotateAngleOld)
                - (dTemp[kY] * dSinRotateAngleOld);
            outAudioData[kY][i] = (dTemp[kX] * dSinRotateAngleOld)
                + (dTemp[kY] * dCosRotateAngleOld);
            outAudioData[kZ][i] = (inAudioData[kX][i] *
                dSinTumbleAngleOld)
                + (dTemp[kZ] * dCosTumbleAngleOld);
            break;
        case kTuRoTi:
            dTemp[kX] = (inAudioData[kX][i] * dCosTumbleAngleOld)
                - (inAudioData[kZ][i] * dSinTumbleAngleOld);
            dTemp[kZ] = (inAudioData[kX][i] * dSinTumbleAngleOld)
                + (inAudioData[kZ][i] * dCosTumbleAngleOld);
            dTemp[kY] = (dTemp[kX] * dSinRotateAngleOld)
                + (inAudioData[kY][i] * dCosRotateAngleOld);
            outAudioData[kX][i] = (dTemp[kX] * dCosRotateAngleOld)
                - (inAudioData[kY][i] * dSinRotateAngleOld);
            outAudioData[kY][i] = (dTemp[kY] * dCosTiltAngleOld)
                - (dTemp[kZ] * dSinTiltAngleOld);
            outAudioData[kZ][i] = (dTemp[kY] * dSinTiltAngleOld)
                + (dTemp[kZ] * dCosTiltAngleOld);
            break;
        case kTuTiRo:
            dTemp[kX] = (inAudioData[kX][i] * dCosTumbleAngleOld)
                - (inAudioData[kZ][i] * dSinTumbleAngleOld);
            dTemp[kZ] = (inAudioData[kX][i] * dSinTumbleAngleOld)
                + (inAudioData[kZ][i] * dCosTumbleAngleOld);
            dTemp[kY] = (inAudioData[kY][i] * dCosTiltAngleOld)
                - (dTemp[kZ] * dSinTiltAngleOld);
            outAudioData[kX][i] = (dTemp[kX] * dCosRotateAngleOld)
                - (dTemp[kY] * dSinRotateAngleOld);
            outAudioData[kY][i] = (dTemp[kX] * dSinRotateAngleOld)
                + (dTemp[kY] * dCosRotateAngleOld);
            outAudioData[kZ][i] = (inAudioData[kY][i] *
```

```
                    dSinTiltAngleOld)
                    + (dTemp[kZ] * dCosTiltAngleOld);
                break;
        }

    }
}
else if(higherThanFirstOrder)
{
    // Rotation only
    for(UInt32 i = 0; i < iFrames; i++)
    {
        dTemp[kX] = (inAudioData[kX][i] * dCosRotateAngleOld)
            - (inAudioData[kY][i] * dSinRotateAngleOld);
        dTemp[kY] = (inAudioData[kX][i] * dSinRotateAngleOld)
            + (inAudioData[kY][i] * dCosRotateAngleOld);
        dTemp[kS] = (inAudioData[kS][i] * dCosRotateAngleOld)
            + (inAudioData[kT][i] * dSinRotateAngleOld);
        dTemp[kT] = (-(inAudioData[kS][i] * dSinRotateAngleOld))
            + (inAudioData[kT][i] * dCosRotateAngleOld);
        dTemp[kU] = (inAudioData[kU][i] * dCos2RotateAngleOld)
            - (inAudioData[kV][i] * dSin2RotateAngleOld);
        dTemp[kV] = (inAudioData[kV][i] * dSin2RotateAngleOld)
            + (inAudioData[kV][i] * dCos2RotateAngleOld);
        dTemp[kL] = (inAudioData[kL][i] * dCosRotateAngleOld)
            + (inAudioData[kM][i] * dSinRotateAngleOld);
        dTemp[kM] = (inAudioData[kL][i] * dSinRotateAngleOld)
            + (inAudioData[kM][i] * dCosRotateAngleOld);
        dTemp[kN] = (inAudioData[kN][i] * dCos2RotateAngleOld)
            + (inAudioData[kO][i] * dSin2RotateAngleOld);
        dTemp[kO] = (inAudioData[kN][i] * dSin2RotateAngleOld)
            + (inAudioData[kO][i] * dCos2RotateAngleOld);
        dTemp[kP] = (inAudioData[kP][i] * dCos3RotateAngleOld)
            + (inAudioData[kQ][i] * dSin3RotateAngleOld);
        dTemp[kQ] = (-(inAudioData[kP][i] * dSin3RotateAngleOld))
            + (inAudioData[kQ][i] * dCos3RotateAngleOld);
        outAudioData[kX][i] = dTemp[kX];
        outAudioData[kY][i] = dTemp[kY];
        outAudioData[kS][i] = dTemp[kS];
        outAudioData[kT][i] = dTemp[kT];
        outAudioData[kU][i] = dTemp[kU];
        outAudioData[kV][i] = dTemp[kV];
        outAudioData[kL][i] = dTemp[kL];
        outAudioData[kM][i] = dTemp[kM];
        outAudioData[kN][i] = dTemp[kN];
        outAudioData[kO][i] = dTemp[kO];
        outAudioData[kP][i] = dTemp[kP];
        outAudioData[kQ][i] = dTemp[kQ];
    }
}

dSinRotateAngleOld = (dSinRotateAngleOld * kAverageCoefficient1)
    + (kAverageCoefficient2 * dSinRotateAngle);
dSin2RotateAngleOld = (dSin2RotateAngleOld * kAverageCoefficient1)
    + (kAverageCoefficient2 * dSin2RotateAngle);
dSin3RotateAngleOld = (dSin3RotateAngleOld * kAverageCoefficient1)
    + (kAverageCoefficient2 * dSin3RotateAngle);
dSinTiltAngleOld = (dSinTiltAngleOld * kAverageCoefficient1)
    + (kAverageCoefficient2 * dSinTiltAngle);
dSinTumbleAngleOld = (dSinTumbleAngleOld * kAverageCoefficient1)
    + (kAverageCoefficient2 * dSinTumbleAngle);
dCosRotateAngleOld = (dCosRotateAngleOld * kAverageCoefficient1)
    + (kAverageCoefficient2 * dCosRotateAngle);
dCos2RotateAngleOld = (dCos2RotateAngleOld * kAverageCoefficient1)
    + (kAverageCoefficient2 * dCos2RotateAngle);
```

```
    dCos3RotateAngleOld = (dCos3RotateAngleOld * kAverageCoefficient1)
        + (kAverageCoefficient2 * dCos3RotateAngle);
    dCosTiltAngleOld = (dCosTiltAngleOld * kAverageCoefficient1)
        + (kAverageCoefficient2 * dCosTiltAngle);
    dCosTumbleAngleOld = (dCosTumbleAngleOld * kAverageCoefficient1)
        + (kAverageCoefficient2 * dCosTumbleAngle);

    return noErr;
}
```

ii. B-Decoder

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder.h
//  Aristotel Digenis
//  MSc Music Technology University of York
//  August 27th 2004
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include <AudioUnit/AudioUnitProperties.h>
#include "AUEffectBase.h"
#include "AUPropertiesPostPantherAdditions.h"
#include "BDecoderVersion.h"
#include "Speaker.h"
#include "math.h"
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Parameters
enum Parameters
{
    ambisonicOrder, globalDistance, globalAmplitude,
    globalZerothBase, globalFirstBase, globalSecondBase,
    globalThirdBase, speakerNumber, azimuth, elevation,
    distance, amplitude, zerothBase, firstBase, secondBase,
    thirdBase, numberOfParameters
};
// Presets
enum Presets
{
    mono, stereo, quad, pentagon, fivePointOne, hexagon,
    sevenPointOne, octagon, decadron, dodecadron, horizontal16,
    threeD8, threeD12, threeD16
};
// I/O Configurations
const int kNumberOfInputs = 16;
const int kNumberOfOutputs = 16;
const AUChannelInfo channelInfo[] =
{
    {kNumberOfInputs, kNumberOfOutputs}
};
const AUChannelInfo* channelInfoPointer = channelInfo;
// Default values for speakers.
const float fDefaultAzimuth = 0.0;
const float fDefaultElevation = 0.0;
const float fDefaultDistance = 5.0;
const float fDefaultAmplitude = 0.5;
const float kMin_dB_Value = -30;
// Default values for order balance.
const float ambisonicOrderBases[4][4] =
{
    {1.0, 0.0, 0.0, 0.0},        // Zeroth Order
    {0.707, 1.00, 0.0, 0.0},     // First Order
    {0.707, 0.75, 0.5, 0.0},     // Second Order
    {0.707, 0.75, 0.5, 0.3}      // Third order
};
// Parameter Strings
```

```
static CFStringRef kAmbisonicOrderIndexParameterStrings[] =
{
    CFSTR("Zeroth"), CFSTR("First"), CFSTR("Second"), CFSTR("Third")
};
static CFStringRef kParameterStrings[] =
{
    CFSTR("Ambisonic Order"), CFSTR("Global Distance"),
    CFSTR("Global Amplitude"), CFSTR("Global Zeroth Base"),
    CFSTR("Global First Base"), CFSTR("Global Second Base"),
    CFSTR("Global Third Base"), CFSTR("Speaker #"),
    CFSTR("Azimuth"), CFSTR("Elevation"), CFSTR("Distance"),
    CFSTR("Amplitude"), CFSTR("Zeroth Base"), CFSTR("First Base"),
    CFSTR("Second Base"), CFSTR("Third Base")
};
static bool sLocalized = false;
// Preset Strings
const int kNumberPresets = 14;
const int kPresetDefault = 3;
const int kPresetDefaultIndex = 3;
static AUPreset kPresets[] =
{
    { 0, CFSTR("Mono") },
    { 1, CFSTR("Stereo") },
    { 2, CFSTR("Quad") },
    { 3, CFSTR("Pentagon") },
    { 4, CFSTR("5.0") },
    { 5, CFSTR("Hexagon") },
    { 6, CFSTR("7.0") },
    { 7, CFSTR("Octagon") },
    { 8, CFSTR("Decadron") },
    { 9, CFSTR("Dodecadron") },
    { 10, CFSTR("Horizontal 16") },
    { 11, CFSTR("3D 8 - Cube") },
    { 12, CFSTR("3D 12") },
    { 13, CFSTR("3D 16") }
};
// Values for presets
float allPresets[][3][16] =
{
    {   // Mono
        {1},                    // Number of speakers
        {0.0},                  // Azimuth
        {fDefaultElevation}     // Elevation
    },

    {   // Stereo
        {2},
        {-30.0, 30.0},
        {fDefaultElevation, fDefaultElevation}
    },

    {   // Quad
        {4},
        {45.0, 135.0, -135.0, -45.0},
        {fDefaultElevation, fDefaultElevation, fDefaultElevation,
            fDefaultElevation}
    },

    {   // Pentagon
        {5},
        {0.0, 72.0, 144.0, -144.0, -72.0},
        {fDefaultElevation, fDefaultElevation, fDefaultElevation,
            fDefaultElevation, fDefaultElevation}
    },
```

```
{   // 5.0
    {5},
    {-30.0, 30.0, -110.0, 110.0, 0.0},
    {fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation}
},

{   // Hexagon
    {6},
    {0.0, 60.0, 120.0, 180.0, -120.0, -60.0},
    {fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation}
},

{   // 7.0
    {7},
    {-30.0, 30.0, -110.0, 110.0, -90.0, 90.0, 0.0},
    {fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation}
},

{   // Octagon
    {8},
    {0.0, 45.0, 90.0, 135.0, 180.0, -135.0, -90.0, -45.0},
    {fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation}
},

{   // Decadron
    {10},
    {0.0, 36.0, 72.0, 108.0, 144.0, 180.0, -144.0, -108.0,
        -72.0, -36.0},
    {fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation}
},

{   // Dodecadron
    {12},
    {0.0, 30, 60.0, 90.0, 120.0, 150.0, 180.0, -150.0, -120.0,
        -90.0, -60.0, -30},
    {fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation}
},

{   // Horizontal 16
    {16},
    {0.0, 22.5, 45.0, 67.5, 90.0, 112.5, 135.0, 157.5, 180.0,
        -157.5, -135.0, -112.5, -90.0, -67.5, -45.0, -22.5},
    {fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation, fDefaultElevation, fDefaultElevation,
        fDefaultElevation}
},

{   // 3D 8 - Cube
    {8},
    {45.0, 135.0, -135.0, -45.0, 45.0, 135.0, -135.0, -45.0},
```

```
            {-45.0, -45.0, -45.0, -45.0, 45.0, 45.0, 45.0, 45.0}
        },

        {   // 3D 12
            {12},
            {0.0, 60.0, 120.0, 180.0, -120.0, -60.0, 0.0, 60.0, 120.0,
                180.0, -120.0, -60.0},
            {-45.0, -45.0, -45.0, -45.0, -45.0, -45.0, 45.0, 45.0, 45.0,
                45.0, 45.0, 45.0}
        },

        {   // 3D 16
            {16},
            {0.0, 45.0, 90.0, 135.0, 180.0, -135.0, -90.0, -45.0, 0.0,
                45.0, 90.0, 135.0, 180.0, -135.0, -90.0, -45.0},
            {-45.0, -45.0, -45.0, -45.0, -45.0, -45.0, -45.0, -45.0,
                45.0, 45.0, 45.0, 45.0, 45.0, 45.0, 45.0, 45.0}
        }
};

class BDecoder : public AUEffectBase
{
public:
    BDecoder(AudioUnit component);
    ~BDecoder();
    virtual ComponentResult GetParameterValueStrings
        (AudioUnitScope inScope,
        AudioUnitParameterID inParameterID,
        CFArrayRef * outStrings);
    virtual ComponentResult GetPresets(CFArrayRef *outData)  const;
    virtual OSStatus NewFactoryPresetSet(const AUPreset
&inNewFactoryPreset);
    virtual void SetNewSelectedPresetValues(int preset);
    virtual ComponentResult GetParameterInfo
        (AudioUnitScope inScope,
        AudioUnitParameterID inParameterID,
        AudioUnitParameterInfo  &outParameterInfo);
    virtual ComponentResult GetPropertyInfo
        (AudioUnitPropertyID inID,
        AudioUnitScope inScope,
        AudioUnitElement inElement,
        UInt32 & outDataSize,
        Boolean & outWritable);
    virtual ComponentResult GetProperty
        (AudioUnitPropertyID inID,
        AudioUnitScope inScope,
        AudioUnitElement inElement,
        void * outData);
    virtual ComponentResult SetParameter
        (AudioUnitParameterID iID,
        AudioUnitScope iScope,
        AudioUnitElement iElem,
        Float32 rValue,
        UInt32 iSchedule);
    virtual void SetAmbisonicProcessingOrder();
    virtual void DisplayParametersForSpeaker();
    virtual void AdjustAmplitudeAndDistance();
    virtual void SetGlobals(int parameter);
    virtual UInt32 SupportedNumChannels(const AUChannelInfo**);
    virtual OSStatus ProcessBufferLists
        (AudioUnitRenderActionFlags&,
        const AudioBufferList&,
        AudioBufferList&, UInt32);
    virtual bool SupportsTail () { return false; }
    virtual ComponentResult  Version() { return kBDecoderVersion; }
```

```
private:
    Speaker* speakerPointer;
    int speakerInFocus;
    int speakersInPreset;
    int ambisonicProcessingOrder;
    float fGlobalDistance;
    float fGlobalAmplitude;
    float fGlobalZerothBase;
    float fGlobalFirstBase;
    float fGlobalSecondBase;
    float fGlobalThirdBase;
    /* For generating speaker numbers during construction
    Will generate as many as specified by kNumberOfOutputs */
    CFStringRef* kSpeakerIndexParameterStrings;
    // For distance-amplitude compensation
    bool exceedingFullAmplitude;
    float temporaryAmplitudes[kNumberOfOutputs];
    // Temporary storage for B-Format Inputs.
    float tempBFormat[kNumberOfInputs];
};
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMPONENT_ENTRY(BDecoder)

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder.cpp
//  Aristotel Digenis
//  MSc Music Technology University of York
//  August 27th 2004
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "BDecoder.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::BDecoder
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BDecoder::BDecoder(AudioUnit component) : AUEffectBase(component)
{
    // Set I/O stream configuration
    CreateElements();
    CAStreamBasicDescription inputStreamDescription;
    inputStreamDescription.SetCanonical (kNumberOfInputs, false);
    inputStreamDescription.mSampleRate = 44100;
    CAStreamBasicDescription outputStreamDescription;
    outputStreamDescription.SetCanonical (kNumberOfOutputs, false);
    outputStreamDescription.mSampleRate = 44100;
    GetOutput(0)->SetStreamFormat(outputStreamDescription);
    GetInput(0)->SetStreamFormat(inputStreamDescription);

    speakerPointer = new Speaker[kNumberOfOutputs];
    speakersInPreset = 16;
    exceedingFullAmplitude = false;

    // Initialize the values of the parameters and their doubles copies
    ambisonicProcessingOrder = 3;
    fGlobalDistance = fDefaultDistance;
    fGlobalAmplitude = fDefaultAmplitude;
    fGlobalZerothBase = ambisonicOrderBases[ambisonicProcessingOrder][0];
    fGlobalFirstBase = ambisonicOrderBases[ambisonicProcessingOrder][1];
    fGlobalSecondBase = ambisonicOrderBases[ambisonicProcessingOrder][2];
    fGlobalThirdBase = ambisonicOrderBases[ambisonicProcessingOrder][3];
    speakerInFocus = 0;

    kSpeakerIndexParameterStrings = new CFStringRef[kNumberOfOutputs];

    for(int a = 0; a < kNumberOfOutputs; a++)
```

```
    {
        char check[3];
        sprintf(check, "%d", a + 1);
        char* checkPointer = check;
        kSpeakerIndexParameterStrings[a] =
            CFStringCreateWithCString(kCFAllocatorDefault,
            checkPointer, kCFStringEncodingMacRoman);
        speakerPointer[a].SetAzimuth(fDefaultAzimuth);
        speakerPointer[a].SetElevation(fDefaultElevation);
        speakerPointer[a].SetDistance(fDefaultDistance);
        speakerPointer[a].SetAmplitude(fDefaultAmplitude);
        speakerPointer[a].SetZerothBase(fGlobalZerothBase);
        speakerPointer[a].SetFirstBase(fGlobalFirstBase);
        speakerPointer[a].SetSecondBase(fGlobalSecondBase);
        speakerPointer[a].SetThirdBase(fGlobalThirdBase);
    }

    if (!sLocalized)
    {
        CFBundleRef bundle =
            CFBundleGetBundleWithIdentifier
            (CFSTR("com.dige.audiounit.bdec"));

        if (bundle != NULL)
        {
            for (int i = 0; i < kNumberPresets; i++)
            {
                kPresets[i].presetName =
                    CFCopyLocalizedStringFromTableInBundle
                    (kPresets[i].presetName,
                    CFSTR("Localizable"),
                    bundle, CFSTR(""));
            }

            for(int a = 0; a < numberOfParameters; a++)
            {
                kParameterStrings[a] =
                    CFCopyLocalizedStringFromTableInBundle
                    (kParameterStrings[a],
                    CFSTR("Localizable"),
                    bundle, CFSTR(""));
            }
        }
        sLocalized = true; //so never pass the test again...
    }
    // Set the parameters
    SetParameter(ambisonicOrder, kAudioUnitScope_Global, 0,
        ambisonicProcessingOrder, 0);
    SetParameter(globalDistance, kAudioUnitScope_Global, 0,
        fGlobalDistance, 0);
    SetParameter(globalAmplitude, kAudioUnitScope_Global, 0,
        10 * log10(fGlobalAmplitude), 0);
    SetParameter(globalZerothBase, kAudioUnitScope_Global, 0,
        fGlobalZerothBase * 100, 0);
    SetParameter(globalFirstBase, kAudioUnitScope_Global, 0,
        fGlobalFirstBase * 100, 0);
    SetParameter(globalSecondBase, kAudioUnitScope_Global, 0,
        fGlobalSecondBase * 100, 0);
    SetParameter(globalThirdBase, kAudioUnitScope_Global, 0,
        fGlobalThirdBase * 100, 0);
    SetParameter(speakerNumber, kAudioUnitScope_Global, 0,
        speakerInFocus, 0);
    SetAFactoryPresetAsCurrent (kPresets[kPresetDefaultIndex]);
    NewFactoryPresetSet(kPresets[quad]);
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::~BDecoder
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BDecoder::~BDecoder()
{
    if(speakerPointer)
    {
        delete speakerPointer;
    }

    if(kSpeakerIndexParameterStrings)
    {
        delete kSpeakerIndexParameterStrings;
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::GetPresets
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BDecoder::GetPresets(CFArrayRef *outData) const
{
    if(outData == NULL)
    {
        return noErr;
    }

    CFMutableArrayRef theArray = CFArrayCreateMutable
        (NULL, kNumberPresets, NULL);

    for(int i = 0; i < kNumberPresets; ++i)
    {
        CFArrayAppendValue (theArray, &kPresets[i]);
    }

    *outData = (CFArrayRef)theArray;

    return noErr;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::NewFactoryPresetSet
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
OSStatus BDecoder::NewFactoryPresetSet(const AUPreset &inNewFactoryPreset)
{
    SInt32 chosenPreset = inNewFactoryPreset.presetNumber;

    switch(chosenPreset)
    {
        case mono:
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
            speakersInPreset = (int) allPresets[mono][0][0];
            SetNewSelectedPresetValues(mono);
            DisplayParametersForSpeaker();
            return noErr;
            break;
        case stereo:
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
            speakersInPreset = (int) allPresets[stereo][0][0];
            SetNewSelectedPresetValues(stereo);
            DisplayParametersForSpeaker();
            return noErr;
            break;
        case quad:
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
```

```
        speakersInPreset = (int) allPresets[quad][0][0];
        SetNewSelectedPresetValues(quad);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case pentagon:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[pentagon][0][0];
        SetNewSelectedPresetValues(pentagon);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case fivePointOne:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[fivePointOne][0][0];
        SetNewSelectedPresetValues(fivePointOne);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case hexagon:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[hexagon][0][0];
        SetNewSelectedPresetValues(hexagon);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case sevenPointOne:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[sevenPointOne][0][0];
        SetNewSelectedPresetValues(sevenPointOne);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case octagon:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[octagon][0][0];
        SetNewSelectedPresetValues(octagon);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case decadron:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[decadron][0][0];
        SetNewSelectedPresetValues(decadron);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case dodecadron:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[dodecadron][0][0];
        SetNewSelectedPresetValues(dodecadron);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case horizontal16:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[horizontal16][0][0];
        SetNewSelectedPresetValues(horizontal16);
        DisplayParametersForSpeaker();
        return noErr;
        break;
    case threeD8:
        SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
        speakersInPreset = (int) allPresets[threeD8][0][0];
        SetNewSelectedPresetValues(threeD8);
```

```
            DisplayParametersForSpeaker();
            return noErr;
            break;
        case threeD12:
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
            speakersInPreset = (int) allPresets[threeD12][0][0];
            SetNewSelectedPresetValues(threeD12);
            DisplayParametersForSpeaker();
            return noErr;
            break;
        case threeD16:
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
            speakersInPreset = (int) allPresets[threeD16][0][0];
            SetNewSelectedPresetValues(threeD16);
            DisplayParametersForSpeaker();
            return noErr;
            break;
    }

    return kAudioUnitErr_InvalidPropertyValue;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::SetNewSelectedPresetValues
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BDecoder::SetNewSelectedPresetValues(int preset)
{
    for(int a = 0; a < speakersInPreset; a++)
    {
        speakerPointer[a].SetAzimuth(allPresets[preset][1][a]);
        speakerPointer[a].SetElevation(allPresets[preset][2][a]);
        speakerPointer[a].SetDistance(fGlobalDistance);
        speakerPointer[a].SetAmplitude(fGlobalAmplitude);
        speakerPointer[a].SetZerothBase(fGlobalZerothBase);
        speakerPointer[a].SetFirstBase(fGlobalFirstBase);
        speakerPointer[a].SetSecondBase(fGlobalSecondBase);
        speakerPointer[a].SetThirdBase(fGlobalThirdBase);
    }

    for(int a = speakersInPreset; a < kNumberOfOutputs; a++)
    {
        speakerPointer[a].SetAzimuth(0.0);
        speakerPointer[a].SetElevation(0.0);
        speakerPointer[a].SetDistance(0.0);
        speakerPointer[a].SetAmplitude(0.0);
        speakerPointer[a].SetZerothBase(0.0);
        speakerPointer[a].SetFirstBase(0.0);
        speakerPointer[a].SetSecondBase(0.0);
        speakerPointer[a].SetThirdBase(0.0);
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::GetParameterValueStrings
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BDecoder::GetParameterValueStrings
    (AudioUnitScope inScope,
    AudioUnitParameterID inParameterID,
    CFArrayRef * outStrings)
{
    if((inScope == kAudioUnitScope_Global))
    {
        if(outStrings == NULL)
        {
            return noErr;
```

```
        }

        if(inParameterID == ambisonicOrder)
        {
            CFStringRef strings[4];

            for(int a = 0; a < 4; a++)
            {
                strings[a] = kAmbisonicOrderIndexParameterStrings[a];
            }

            *outStrings = CFArrayCreate( NULL,
                (const void **)strings, 4, NULL);

            return noErr;
        }

        if(inParameterID == speakerNumber)
        {
            CFStringRef strings[kNumberOfOutputs];

            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                strings[a] = kSpeakerIndexParameterStrings[a];
            }

            *outStrings = CFArrayCreate( NULL,
                (const void **)strings, kNumberOfOutputs, NULL);

            return noErr;
        }
    }

    return kAudioUnitErr_InvalidProperty;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::GetParameterInfo
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BDecoder::GetParameterInfo
    (AudioUnitScope inScope,
    AudioUnitParameterID inParameterID,
    AudioUnitParameterInfo &outParameterInfo)
{
    ComponentResult result = noErr;

    outParameterInfo.flags = kAudioUnitParameterFlag_IsWritable
        | kAudioUnitParameterFlag_IsReadable;

    if (inScope == kAudioUnitScope_Global)
    {
        switch(inParameterID)
        {
            case ambisonicOrder:
                AudioUnitParameterInfo ambisonicOrderIndexParameterInfo;
                AUBase::FillInParameterName
                    (ambisonicOrderIndexParameterInfo,
                    kParameterStrings[ambisonicOrder],
                    false);
                ambisonicOrderIndexParameterInfo.unit =
                    kAudioUnitParameterUnit_Indexed;
                ambisonicOrderIndexParameterInfo.minValue = 0;
                ambisonicOrderIndexParameterInfo.maxValue = 3;
                ambisonicOrderIndexParameterInfo.defaultValue =
                    ambisonicProcessingOrder;
```

```
            ambisonicOrderIndexParameterInfo.flags =
                kAudioUnitParameterFlag_IsWritable |
                kAudioUnitParameterFlag_IsReadable |
                kAudioUnitParameterFlag_IsGlobalMeta;
            outParameterInfo = ambisonicOrderIndexParameterInfo;
            break;

        case globalDistance:
            AudioUnitParameterInfo globalDistanceParameterInfo;
            AUBase::FillInParameterName
                (globalDistanceParameterInfo,
                kParameterStrings[globalDistance],
                false);
            globalDistanceParameterInfo.unit =
                kAudioUnitParameterUnit_Meters;
            globalDistanceParameterInfo.minValue = 0.0;
            globalDistanceParameterInfo.maxValue = 10.0;
            globalDistanceParameterInfo.defaultValue =
                fGlobalDistance;
            globalDistanceParameterInfo.flags =
                kAudioUnitParameterFlag_IsWritable |
                kAudioUnitParameterFlag_IsReadable |
                kAudioUnitParameterFlag_IsGlobalMeta;
            outParameterInfo = globalDistanceParameterInfo;
            break;

        case globalAmplitude:
            AudioUnitParameterInfo globalAmplitudeParameterInfo;
            AUBase::FillInParameterName
                (globalAmplitudeParameterInfo,
                kParameterStrings[globalAmplitude],
                false);
            globalAmplitudeParameterInfo.unit =
                kAudioUnitParameterUnit_Decibels;
            globalAmplitudeParameterInfo.minValue = kMin_dB_Value;
            globalAmplitudeParameterInfo.maxValue = 0.0;
            globalAmplitudeParameterInfo.defaultValue =
                10 * log10(fGlobalAmplitude);
            globalAmplitudeParameterInfo.flags =
                kAudioUnitParameterFlag_ValuesHaveStrings |
                kAudioUnitParameterFlag_IsWritable |
                kAudioUnitParameterFlag_IsReadable |
                kAudioUnitParameterFlag_IsGlobalMeta;
            outParameterInfo = globalAmplitudeParameterInfo;
            break;

        case globalZerothBase:
            AudioUnitParameterInfo globalZerothBaseParameterInfo;
            AUBase::FillInParameterName
                (globalZerothBaseParameterInfo,
                kParameterStrings[globalZerothBase],
                false);
            globalZerothBaseParameterInfo.unit =
                kAudioUnitParameterUnit_Percent;
            globalZerothBaseParameterInfo.minValue = 0.0;
            globalZerothBaseParameterInfo.maxValue = 100.0;
            globalZerothBaseParameterInfo.defaultValue =
                fGlobalZerothBase * 100;
            globalZerothBaseParameterInfo.flags =
                kAudioUnitParameterFlag_IsWritable |
                kAudioUnitParameterFlag_IsReadable |
                kAudioUnitParameterFlag_IsGlobalMeta;
            outParameterInfo = globalZerothBaseParameterInfo;
            break;
```

```
case globalFirstBase:
    AudioUnitParameterInfo globalFirstBaseParameterInfo;
    AUBase::FillInParameterName
        (globalFirstBaseParameterInfo,
        kParameterStrings[globalFirstBase],
        false);
    globalFirstBaseParameterInfo.unit =
        kAudioUnitParameterUnit_Percent;
    globalFirstBaseParameterInfo.minValue = 0.0;
    globalFirstBaseParameterInfo.maxValue = 100.0;
    globalFirstBaseParameterInfo.defaultValue =
        fGlobalFirstBase * 100;
    globalFirstBaseParameterInfo.flags =
        kAudioUnitParameterFlag_IsWritable |
        kAudioUnitParameterFlag_IsReadable |
        kAudioUnitParameterFlag_IsGlobalMeta;
    outParameterInfo = globalFirstBaseParameterInfo;
    break;

case globalSecondBase:
    AudioUnitParameterInfo globalSecondBaseParameterInfo;
    AUBase::FillInParameterName
        (globalSecondBaseParameterInfo,
        kParameterStrings[globalSecondBase],
        false);
    globalSecondBaseParameterInfo.unit =
        kAudioUnitParameterUnit_Percent;
    globalSecondBaseParameterInfo.minValue = 0.0;
    globalSecondBaseParameterInfo.maxValue = 100.0;
    globalSecondBaseParameterInfo.defaultValue =
        fGlobalSecondBase * 100;
    globalSecondBaseParameterInfo.flags =
        kAudioUnitParameterFlag_IsWritable |
        kAudioUnitParameterFlag_IsReadable |
        kAudioUnitParameterFlag_IsGlobalMeta;
    outParameterInfo = globalSecondBaseParameterInfo;
    break;

case globalThirdBase:
    AudioUnitParameterInfo globalThirdBaseParameterInfo;
    AUBase::FillInParameterName
        (globalThirdBaseParameterInfo,
        kParameterStrings[globalThirdBase],
        false);
    globalThirdBaseParameterInfo.unit =
        kAudioUnitParameterUnit_Percent;
    globalThirdBaseParameterInfo.minValue = 0.0;
    globalThirdBaseParameterInfo.maxValue = 100.0;
    globalThirdBaseParameterInfo.defaultValue =
        fGlobalThirdBase * 100;
    globalThirdBaseParameterInfo.flags =
        kAudioUnitParameterFlag_IsWritable |
        kAudioUnitParameterFlag_IsReadable |
        kAudioUnitParameterFlag_IsGlobalMeta;
    outParameterInfo = globalThirdBaseParameterInfo;
    break;

case speakerNumber:
    AudioUnitParameterInfo speakerIndexParameterInfo;
    AUBase::FillInParameterName
        (speakerIndexParameterInfo,
        kParameterStrings[speakerNumber],
        false);
    speakerIndexParameterInfo.unit =
        kAudioUnitParameterUnit_Indexed;
```

```
            speakerIndexParameterInfo.minValue = 0;
            speakerIndexParameterInfo.maxValue = kNumberOfOutputs - 1;
            speakerIndexParameterInfo.defaultValue = 0;
            speakerIndexParameterInfo.flags =
                kAudioUnitParameterFlag_IsWritable |
                kAudioUnitParameterFlag_IsReadable |
                kAudioUnitParameterFlag_IsGlobalMeta;
            outParameterInfo = speakerIndexParameterInfo;
            break;

        case azimuth:
            AUBase::FillInParameterName
                (outParameterInfo,
                kParameterStrings[azimuth],
                false);
            outParameterInfo.unit = kAudioUnitParameterUnit_Degrees;
            outParameterInfo.minValue = -180.0;
            outParameterInfo.maxValue = 180.0;
            outParameterInfo.defaultValue = 0.0;
            break;

        case elevation:
            AUBase::FillInParameterName
                (outParameterInfo,
                kParameterStrings[elevation],
                false);
            outParameterInfo.unit = kAudioUnitParameterUnit_Degrees;
            outParameterInfo.minValue = -90.0;
            outParameterInfo.maxValue = 90.0;
            outParameterInfo.defaultValue = 0.0;
            break;

        case distance:
            AudioUnitParameterInfo distanceParameterInfo;
            AUBase::FillInParameterName
                (distanceParameterInfo,
                kParameterStrings[distance],
                false);
            distanceParameterInfo.unit =
                kAudioUnitParameterUnit_Meters;
            distanceParameterInfo.minValue = 0.0;
            distanceParameterInfo.maxValue = 10.0;
            distanceParameterInfo.defaultValue = fDefaultDistance;
            distanceParameterInfo.flags =
                kAudioUnitParameterFlag_IsWritable |
                kAudioUnitParameterFlag_IsReadable |
                kAudioUnitParameterFlag_IsGlobalMeta;
            outParameterInfo = distanceParameterInfo;
            break;

        case amplitude:
            AUBase::FillInParameterName
                (outParameterInfo,
                kParameterStrings[amplitude],
                false);
            outParameterInfo.unit = kAudioUnitParameterUnit_Decibels;
            outParameterInfo.minValue = kMin_dB_Value;
            outParameterInfo.maxValue = 0.0;
            outParameterInfo.defaultValue =
                10 * log10(fDefaultAmplitude);
            outParameterInfo.flags |=
                kAudioUnitParameterFlag_ValuesHaveStrings;
            break;

        case zerothBase:
```

```
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[zerothBase],
                    false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Percent;
                outParameterInfo.minValue = 0.0;
                outParameterInfo.maxValue = 100.0;
                outParameterInfo.defaultValue = fGlobalZerothBase * 100;
                break;

            case firstBase:
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[firstBase],
                    false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Percent;
                outParameterInfo.minValue = 0.0;
                outParameterInfo.maxValue = 100.0;
                outParameterInfo.defaultValue = fGlobalFirstBase * 100;
                break;

            case secondBase:
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[secondBase],
                    false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Percent;
                outParameterInfo.minValue = 0.0;
                outParameterInfo.maxValue = 100.0;
                outParameterInfo.defaultValue = fGlobalSecondBase * 100;
                break;

            case thirdBase:
                AUBase::FillInParameterName
                    (outParameterInfo,
                    kParameterStrings[thirdBase],
                    false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Percent;
                outParameterInfo.minValue = 0.0;
                outParameterInfo.maxValue = 100.0;
                outParameterInfo.defaultValue = fGlobalThirdBase * 100;
                break;

             default:
                result = kAudioUnitErr_InvalidParameter;
                break;
        }
    }
    else
    {
        result = kAudioUnitErr_InvalidParameter;
    }

    return result;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::GetPropertyInfo
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BDecoder::GetPropertyInfo
    (AudioUnitPropertyID inID,
    AudioUnitScope inScope,
    AudioUnitElement inElement,
    UInt32 & outDataSize,
    Boolean & outWritable)
```

```
{
    if (inScope == kAudioUnitScope_Global)
    {
        switch (inID)
        {
            case kAudioUnitProperty_IconLocation:
                outWritable = false;
                outDataSize = sizeof (CFURLRef);
                return noErr;

            case kAudioUnitProperty_ParameterStringFromValue:
                outWritable = false;
                outDataSize = sizeof (AudioUnitParameterStringFromValue);
                return noErr;

            case kAudioUnitProperty_ParameterValueFromString:
                outWritable = false;
                outDataSize = sizeof (AudioUnitParameterValueFromString);
                return noErr;
        }
    }

    return AUEffectBase::GetPropertyInfo
        (inID, inScope, inElement, outDataSize, outWritable);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::GetProperty
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BDecoder::GetProperty
    (AudioUnitPropertyID inID,
    AudioUnitScope inScope,
    AudioUnitElement inElement,
    void * outData)
{
    if (inScope == kAudioUnitScope_Global)
    {
        switch (inID)
        {
            case kAudioUnitProperty_IconLocation:
            {
                CFBundleRef bundle =
                    CFBundleGetBundleWithIdentifier
                    (CFSTR("com.dige.audiounit.bdec"));

                if (bundle == NULL)
                {
                    return fnfErr;
                }

                CFURLRef bundleURL =
                    CFBundleCopyResourceURL(bundle,
                    CFSTR("digenis"), CFSTR("icns"), NULL);

                if (bundleURL == NULL)
                {
                    return fnfErr;
                }

                (*(CFURLRef *)outData) = bundleURL;

                return noErr;
            }
            case kAudioUnitProperty_ParameterValueFromString:
            {
```

```
                OSStatus retVal = kAudioUnitErr_InvalidPropertyValue;
                AudioUnitParameterValueFromString &name =
                    *(AudioUnitParameterValueFromString*)outData;

                UniChar chars[2];
                chars[0] = '-';
                chars[1] = 0x221e; // this is the unicode symbol
                                   // for infinity
                CFStringRef comparisonString =
                    CFStringCreateWithCharacters (NULL, chars, 2);

                if ( CFStringCompare(comparisonString, name.inString, 0)
                    == kCFCompareEqualTo )
                {
                    name.outValue = kMin_dB_Value;
                    retVal = noErr;
                }

                if (comparisonString)
                {
                    CFRelease(comparisonString);
                }

                return retVal;
            }

            case kAudioUnitProperty_ParameterStringFromValue:
            {
                AudioUnitParameterStringFromValue &name =
                    *(AudioUnitParameterStringFromValue*)outData;
                Float32 paramValue = (name.inValue == NULL ?
                    GetParameter (amplitude) : *(name.inValue));

                // for this usage only values <= -40 dB (the min value)
                // have a special name "-infinity"
                if (paramValue <= kMin_dB_Value)
                {
                    UniChar chars[2];
                    chars[0] = '-';
                    chars[1] = 0x221e; // this is the unicode symbol
                                       // for infinity
                    name.outString = CFStringCreateWithCharacters
                        (NULL, chars, 2);
                }
                else
                {
                    name.outString = NULL;
                }

                return noErr;
            }

            case kAudioUnitProperty_SupportedNumChannels:
            {
                return noErr;
            }
        }
    }
    return AUEffectBase::GetProperty (inID, inScope, inElement, outData);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::SetParameter
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BDecoder::SetParameter
```

```
    (AudioUnitParameterID iID,
    AudioUnitScope iScope,
    AudioUnitElement iElem,
    Float32 rValue,
    UInt32 iSchedule)
{
    if (iScope==kAudioUnitScope_Global && iElem==0)
    {
        switch (iID)
        {
            case ambisonicOrder:
                ambisonicProcessingOrder = (int) rValue;
                SetAmbisonicProcessingOrder();
                SetParameter(globalZerothBase, kAudioUnitScope_Global, 0,
                    fGlobalZerothBase * 100, 0);
                SetParameter(globalFirstBase, kAudioUnitScope_Global, 0,
                    fGlobalFirstBase * 100, 0);
                SetParameter(globalSecondBase, kAudioUnitScope_Global, 0,
                    fGlobalSecondBase * 100, 0);
                SetParameter(globalThirdBase, kAudioUnitScope_Global, 0,
                    fGlobalThirdBase * 100, 0);
                break;
            case globalDistance:
                fGlobalDistance = rValue;
                SetGlobals(globalDistance);
                SetParameter(distance, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetDistance(), 0);
                break;
            case globalAmplitude:
                fGlobalAmplitude = (pow(10, rValue / 10));
                SetGlobals(globalAmplitude);
                SetParameter(amplitude, kAudioUnitScope_Global, 0, 10 *
                    log10(speakerPointer[speakerInFocus].GetAmplitude()),
                    0);
                break;
            case globalZerothBase:
                fGlobalZerothBase = rValue / 100;
                SetGlobals(globalZerothBase);
                SetParameter(zerothBase, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetZerothBase()
                    * 100, 0);
                break;
            case globalFirstBase:
                fGlobalFirstBase = rValue / 100;
                SetGlobals(globalFirstBase);
                SetParameter(firstBase, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetFirstBase()
                    * 100, 0);
                break;
            case globalSecondBase:
                fGlobalSecondBase = rValue / 100;
                SetGlobals(globalSecondBase);
                SetParameter(secondBase, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetSecondBase()
                    * 100, 0);
                break;
            case globalThirdBase:
                fGlobalThirdBase = rValue / 100;
                SetGlobals(globalThirdBase);
                SetParameter(thirdBase, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetThirdBase()
                    * 100, 0);
                break;
            case speakerNumber:
                speakerInFocus = (int) rValue;
```

```
                AUBase::SetParameter(azimuth, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetAzimuth(), 0);
                AUBase::SetParameter(elevation, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetElevation(), 0);
                AUBase::SetParameter(distance, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetDistance(), 0);
                AUBase::SetParameter(amplitude, kAudioUnitScope_Global, 0,
                    10 *
                    log10(speakerPointer[speakerInFocus].GetAmplitude()),
                    0);
                AUBase::SetParameter(zerothBase, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetZerothBase() * 100,
                    0);
                AUBase::SetParameter(firstBase, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetFirstBase() * 100,
                    0);
                AUBase::SetParameter(secondBase, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetSecondBase() * 100,
                    0);
                AUBase::SetParameter(thirdBase, kAudioUnitScope_Global, 0,
                    speakerPointer[speakerInFocus].GetThirdBase() * 100,
                    0);
                break;
            case azimuth:
                speakerPointer[speakerInFocus].SetAzimuth(rValue);
                break;
            case elevation:
                speakerPointer[speakerInFocus].SetElevation(rValue);
                break;
            case distance:
                speakerPointer[speakerInFocus].SetDistance(rValue);
                AdjustAmplitudeAndDistance();
                break;
            case amplitude:
                speakerPointer[speakerInFocus].SetAmplitude
                    (pow(10, rValue / 10));
                break;
            case zerothBase:
                speakerPointer[speakerInFocus].SetZerothBase(rValue / 100);
                break;
            case firstBase:
                speakerPointer[speakerInFocus].SetFirstBase(rValue / 100);
                break;
            case secondBase:
                speakerPointer[speakerInFocus].SetSecondBase(rValue / 100);
                break;
            case thirdBase:
                speakerPointer[speakerInFocus].SetThirdBase(rValue / 100);
                break;
        }
    }

    for(int a = 0; a < kNumberOfOutputs; a++)
    {
        speakerPointer[a].CalculateSpeakerZerothBase();
        speakerPointer[a].CalculateSpeakerFirstBase();
        speakerPointer[a].CalculateSpeakerSecondBase();
        speakerPointer[a].CalculateSpeakerThirdBase();
        speakerPointer[a].CalculateAzimuthInRadians();
        speakerPointer[a].CalculateElevationInRadians();
        speakerPointer[a].CalculateSpeakerCoefficients();
    }

    return AUBase::SetParameter(iID, iScope, iElem, rValue, iSchedule);
}
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::SetAmbisonicProcessingOrder
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BDecoder::SetAmbisonicProcessingOrder()
{
    fGlobalZerothBase = ambisonicOrderBases[ambisonicProcessingOrder][0];
    fGlobalFirstBase = ambisonicOrderBases[ambisonicProcessingOrder][1];
    fGlobalSecondBase = ambisonicOrderBases[ambisonicProcessingOrder][2];
    fGlobalThirdBase = ambisonicOrderBases[ambisonicProcessingOrder][3];

    SetGlobals(globalZerothBase);
    SetGlobals(globalFirstBase);
    SetGlobals(globalSecondBase);
    SetGlobals(globalThirdBase);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::DisplayParametersForSpeaker
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BDecoder::DisplayParametersForSpeaker()
{
    SetParameter(azimuth, kAudioUnitScope_Global, 0,
        speakerPointer[speakerInFocus].GetAzimuth(), 0);
    SetParameter(elevation, kAudioUnitScope_Global, 0,
        speakerPointer[speakerInFocus].GetElevation(), 0);
    SetParameter(distance, kAudioUnitScope_Global, 0,
        speakerPointer[speakerInFocus].GetDistance(), 0);
    SetParameter(amplitude, kAudioUnitScope_Global, 0,
        10 * log10(speakerPointer[speakerInFocus].GetAmplitude()), 0);
    SetParameter(zerothBase, kAudioUnitScope_Global, 0,
        speakerPointer[speakerInFocus].GetZerothBase() * 100, 0);
    SetParameter(firstBase, kAudioUnitScope_Global, 0,
        speakerPointer[speakerInFocus].GetFirstBase() * 100, 0);
    SetParameter(secondBase, kAudioUnitScope_Global, 0,
        speakerPointer[speakerInFocus].GetSecondBase() * 100, 0);
    SetParameter(thirdBase, kAudioUnitScope_Global, 0,
        speakerPointer[speakerInFocus].GetThirdBase() * 100, 0);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::AdjustAmplitudeAndDistance
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BDecoder::AdjustAmplitudeAndDistance()
{
    // Calculate the new amplitude and store it in a temporary double
    // variable
    double tempAmplitudeCalculation = 0.5 *
        pow((speakerPointer[speakerInFocus].GetDistance()
        / 5.0), 2);
    // If the new amplitude does not exceed the maximum amplitude
    if(tempAmplitudeCalculation <= 1.0)
    {
        // If the maximum amplitude had previously been exceeded
        if(exceedingFullAmplitude == true)
        {
            // Get all amplitudes of the speakers prior to exceeding
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetAmplitude(temporaryAmplitudes[a]);
            }
            // Update the amplitude parameter
            SetParameter(amplitude, kAudioUnitScope_Global, 0,
                10 * log10(tempAmplitudeCalculation), 0);
            // Mark that the amplitude does not anymore exceed the maximum
```

```
            exceedingFullAmplitude = false;
        }
        // If the new amplitude is under the maximum, store it

    speakerPointer[speakerInFocus].SetAmplitude(tempAmplitudeCalculation);
        // Update the amplitude parameter
        SetParameter(amplitude, kAudioUnitScope_Global, 0, 10 *
            log10(speakerPointer[speakerInFocus].GetAmplitude()), 0);
    }
    // If the new amplitude does exceed the maximum amplitude
    else if (tempAmplitudeCalculation > 1.0)
    {
        // If the values have not already been stored
        if(exceedingFullAmplitude != true)
        {
            // Store all amplitudes of the speakers in temporary memory
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                temporaryAmplitudes[a] = speakerPointer[a].GetAmplitude();
            }
        }
        // Mark that it will now be exceeding the maximum amplitude and
        // compensating by reducing the amplitude of other speakers
        exceedingFullAmplitude = true;
        // Even though the new amplitude exceeds the maximum, apply it to
        // the speaker
        speakerPointer[speakerInFocus].SetAmplitude
            (tempAmplitudeCalculation);
        // Adjust (reduce) all amplitudes bby the ratio which the maximum
        // is exceeded
        for(int a = 0; a < kNumberOfOutputs; a++)
        {
            speakerPointer[a].SetAmplitude(speakerPointer[a].GetAmplitude()
                * (1.0 / tempAmplitudeCalculation));
        }
        // Update the amplitude parameter
        SetParameter(amplitude, kAudioUnitScope_Global, 0, 10 *
            log10(speakerPointer[speakerInFocus].GetAmplitude()), 0);
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::SetGlobals
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BDecoder::SetGlobals(int parameter)
{
    switch(parameter)
    {
        case globalDistance:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetDistance(fGlobalDistance);
            }
            break;
        case globalAmplitude:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetAmplitude(fGlobalAmplitude);
            }
            break;
        case globalZerothBase:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetZerothBase(fGlobalZerothBase);
            }
```

```
            break;
        case globalFirstBase:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetFirstBase(fGlobalFirstBase);
            }
            break;
        case globalSecondBase:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetSecondBase(fGlobalSecondBase);
            }
            break;
        case globalThirdBase:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetThirdBase(fGlobalThirdBase);
            }
            break;
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::SupportedNumChannel
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UInt32 BDecoder::SupportedNumChannels(const AUChannelInfo** outInfo)
{
    UInt32 count = 0;
    for(; channelInfoPointer && (channelInfoPointer[count].inChannels
        != 0 || channelInfoPointer[count].outChannels != 0); count++) {};

    if(outInfo)
    {
        *outInfo = channelInfoPointer;
    }

    return sizeof(channelInfoPointer)/sizeof(*channelInfoPointer);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BDecoder::ProcessBufferLists
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
OSStatus BDecoder::ProcessBufferLists
    (AudioUnitRenderActionFlags& iFlags,
    const AudioBufferList& inBufferList,
    AudioBufferList& outBufferList,
    UInt32 iFrames)
{
    // Array of pointers, as many as input signals(B-Format).
    float* audioData[kNumberOfInputs];
    // Pointers point to incomming audio buffers.
    for(int i = 0; i < kNumberOfInputs; i++)
    {
        audioData[i] = (float*) inBufferList.mBuffers[i].mData;
    }

    if(ambisonicProcessingOrder == 0)
    {
        // For all the frames.
        for(UInt32 j = 0; j < iFrames; j++)
        {
            // Store the values of the B-Format inputs for that frame.
            for(int a = 0; a < kNumberOfInputs; a++)
            {
                tempBFormat[a] = audioData[a][j];
```

```
        }
        // Derive the output for all the speakers in the preset and
        // store it in the buffer.
        for(int speakerCounter = 0; speakerCounter <
            speakersInPreset; speakerCounter++)
        {
            audioData[speakerCounter][j] = (float)
            (tempBFormat[W] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(W));
        }
        // Zero pad the the buffers of ouputs that will not be used.
        for(int speakerCounter = speakersInPreset; speakerCounter <
            kNumberOfOutputs; speakerCounter++)
        {
            audioData[speakerCounter][j] = 0.0;
        }
    }
}

if(ambisonicProcessingOrder == 1)
{
    // For all the frames.
    for(UInt32 j = 0; j < iFrames; j++)
    {
        // Store the values of the B-Format inputs for that frame.
        for(int a = 0; a < kNumberOfInputs; a++)
        {
            tempBFormat[a] = audioData[a][j];
        }
        // Derive the output for all the speakers in the preset and
        // store it in the buffer.
        for(int speakerCounter = 0; speakerCounter < speakersInPreset;
            speakerCounter++)
        {
            audioData[speakerCounter][j] = (float)
            (tempBFormat[W] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(W)) +
            (tempBFormat[X] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(X)) +
            (tempBFormat[Y] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(Y)) +
            (tempBFormat[Z] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(Z));
        }
        // Zero pad the the buffers of ouputs that will not be used.
        for(int speakerCounter = speakersInPreset; speakerCounter <
            kNumberOfOutputs; speakerCounter++)
        {
            audioData[speakerCounter][j] = 0.0;
        }
    }
}

if(ambisonicProcessingOrder == 2)
{
    // For all the frames
    for(UInt32 j = 0; j < iFrames; j++)
    {
        // Store the values of the B-Format inputs for that frame
        for(int a = 0; a < kNumberOfInputs; a++)
```

```
            {
                tempBFormat[a] = audioData[a][j];
            }
            // Derive the output for all the speakers in the preset and
            // store it in the buffer
            for(int speakerCounter = 0; speakerCounter < speakersInPreset;
                speakerCounter++)
            {
                audioData[speakerCounter][j] = (float)
                (tempBFormat[W] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(W)) +
                (tempBFormat[X] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(X)) +
                (tempBFormat[Y] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(Y)) +
                (tempBFormat[Z] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(Z)) +
                (tempBFormat[R] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(R)) +
                (tempBFormat[S] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(S)) +
                (tempBFormat[T] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(T)) +
                (tempBFormat[U] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(U)) +
                (tempBFormat[V] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(V));
            }
            // Zero pad the the buffers of ouputs that will not be used.
            for(int speakerCounter = speakersInPreset; speakerCounter <
                kNumberOfOutputs; speakerCounter++)
            {
                audioData[speakerCounter][j] = 0.0;
            }
        }
    }

    if(ambisonicProcessingOrder == 3)
    {
        // For all the frames
        for(UInt32 j = 0; j < iFrames; j++)
        {
            // Store the values of the B-Format inputs for that frame
            for(int a = 0; a < kNumberOfInputs; a++)
            {
                tempBFormat[a] = audioData[a][j];
            }
            // Derive the output for all the speakers in the preset and
            // store it in the buffer
            for(int speakerCounter = 0; speakerCounter < speakersInPreset;
                speakerCounter++)
            {
                audioData[speakerCounter][j] = (float)
                (tempBFormat[W] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(W)) +
```

```
                    (tempBFormat[X] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(X)) +
                    (tempBFormat[Y] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(Y)) +
                    (tempBFormat[Z] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(Z)) +
                    (tempBFormat[R] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(R)) +
                    (tempBFormat[S] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(S)) +
                    (tempBFormat[T] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(T)) +
                    (tempBFormat[U] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(U)) +
                    (tempBFormat[V] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(V)) +
                    (tempBFormat[K] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(K)) +
                    (tempBFormat[L] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(L)) +
                    (tempBFormat[M] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(M)) +
                    (tempBFormat[N] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(N)) +
                    (tempBFormat[O] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(O)) +
                    (tempBFormat[P] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(P)) +
                    (tempBFormat[Q] *
                        speakerPointer[speakerCounter].
                        GetSpeakerCoefficient(Q));
                }
                // Zero pad the the buffers of ouputs that will not be used.
                for(int speakerCounter = speakersInPreset; speakerCounter <
                    kNumberOfOutputs; speakerCounter++)
                {
                    audioData[speakerCounter][j] = 0.0;
                }
            }
        }

        for(int i = 0; i < kNumberOfOutputs; i++)
        {
            outBufferList.mBuffers[i].mData = audioData[i];
        }

        return noErr;
}
```

iii. B-Binaural

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural.h
//  Aristotel Digenis
//  MSc Music Technology University of York
//  August 27th 2004
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include <AudioUnit/AudioUnitProperties.h>
#include "AUEffectBase.h"
#include "AUPropertiesPostPantherAdditions.h"
#include <Accelerate/Accelerate.h>
#include "BBinauralVersion.h"
#include "math.h"
#include "Speaker.h"

// Include Normal Ear HRTFs
#include "nplus40e000a.h"
#include "nplus40e045a.h"
#include "nplus40e090a.h"
#include "nplus40e135a.h"
#include "nplus40e180a.h"
#include "nplus40e225a.h"
#include "nplus40e270a.h"
#include "nplus40e315a.h"
#include "n00e000a.h"
#include "n00e045a.h"
#include "n00e090a.h"
#include "n00e135a.h"
#include "n00e180a.h"
#include "n00e225a.h"
#include "n00e270a.h"
#include "n00e315a.h"
#include "nminus40e000a.h"
#include "nminus40e045a.h"
#include "nminus40e090a.h"
#include "nminus40e135a.h"
#include "nminus40e180a.h"
#include "nminus40e225a.h"
#include "nminus40e270a.h"
#include "nminus40e315a.h"
// Include Large Ear HRTFs
#include "lplus40e000a.h"
#include "lplus40e045a.h"
#include "lplus40e090a.h"
#include "lplus40e135a.h"
#include "lplus40e180a.h"
#include "lplus40e225a.h"
#include "lplus40e270a.h"
#include "lplus40e315a.h"
#include "l00e000a.h"
#include "l00e045a.h"
#include "l00e090a.h"
#include "l00e135a.h"
#include "l00e180a.h"
#include "l00e225a.h"
#include "l00e270a.h"
#include "l00e315a.h"
#include "lminus40e000a.h"
#include "lminus40e045a.h"
#include "lminus40e090a.h"
#include "lminus40e135a.h"
#include "lminus40e180a.h"
#include "lminus40e225a.h"
```

```
#include "lminus40e270a.h"
#include "lminus40e315a.h"
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
enum Parameters
{
    ambisonicOrder, typeOfEar, headphoneAmplitude, numberOfParameters
};

enum AmbisonicOrder
{
    globalZerothBase, globalFirstBase, globalSecondBase, globalThirdBase
};

enum EarType
{
    normalEar, largeEar, numberOfEarTypes
};

enum SpeakerRings
{
    upper, middle, lower, numberOfSpeakerRings
};

enum ClockwiseSpeakerAngles
{
    clockwise000, clockwise045, clockwise090, clockwise135, clockwise180,
    clockwise225, clockwise270, clockwise315, numberOfAngles
};

enum Ears
{
    left, right, numberOfEars
};

enum Presets
{
    quad, octagon, threeD8, threeD16, numberOfDecoderPresets
};
// I/O Configurations
const int kNumberOfInputs = 16;
const int kNumberOfSpeakers = 16;
const int kNumberOfOutputs = 2;
const AUChannelInfo channelInfo[] =
{
    {kNumberOfInputs, kNumberOfOutputs}
};
const AUChannelInfo* channelInfoPointer = channelInfo;
// Default values for speakers
const float fDefaultAzimuth = 0.0;
const float fDefaultElevation = 0.0;
const float fDefaultAmplitude = 0.5;
const int overlapSize = 511;
const float kMin_dB_Value = -40;
// Default values for order balance
const float ambisonicOrderBases[4][4] =
{
    {1.0, 0.0, 0.0, 0.0},        // Zeroth Order
    {0.707, 1.0, 0.0, 0.0},      // First Order
    {0.707, 0.75, 0.5, 0.0},     // Second Order
    {0.707, 0.75, 0.5, 0.3}      // Third order
};
// Parameter Strings
static CFStringRef kAmbisonicOrderIndexParameterStrings[] =
{
    CFSTR("Zeroth"), CFSTR("First"), CFSTR("Second"), CFSTR("Third")
```

```
};
static CFStringRef kEarTypeIndexParameterStrings[] =
{
    CFSTR("Normal"), CFSTR("Large")
};
static CFStringRef kParameterStrings[] =
{
    CFSTR("Ambisonic Order"), CFSTR("Ear Type"), CFSTR("Amplitude")
};
static bool sLocalized = false;
// Preset Strings
const int kNumberPresets = numberOfDecoderPresets;
static const int kPresetDefault = 3;
static const int kPresetDefaultIndex = 3;
static AUPreset kPresets[] =
{
    { 0, CFSTR("Quad") },
    { 1, CFSTR("Octagon") },
    { 2, CFSTR("3D 8") },
    { 3, CFSTR("3D 16") }
};
// Values for presets
float allPresets[][3][16] =
{
    {   // Quad
        {4},                                            // Number of speakers
        {45.0, 135.0, -135.0, -45.0},                   // Azimuth
        {fDefaultElevation, fDefaultElevation,          // Elevation
            fDefaultElevation, fDefaultElevation}
    },

    {   // Octagon
        {8},
        {0.0, 45.0, 90.0, 135.0, 180.0, -135.0, -90.0, -45.0},
        {fDefaultElevation, fDefaultElevation, fDefaultElevation,
            fDefaultElevation, fDefaultElevation, fDefaultElevation,
            fDefaultElevation, fDefaultElevation}
    },

    {   // 3D 8
        {8},
        {45.0, 135.0, -135.0, -45.0, 45.0, 135.0, -135.0, -45.0},
        {40.0, 40.0, 40.0, 40.0, -40.0, -40.0, -40.0, -40.0}
    },

    {   // 3D 16
        {16},
        {0.0, 45.0, 90.0, 135.0, 180.0, -135.0, -90.0, -45.0, 0.0,
            45.0, 90.0, 135.0, 180.0, -135.0, -90.0, -45.0},
        {40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, 40.0, -40.0, -40.0,
            -40.0, -40.0, -40.0, -40.0, -40.0, -40.0}
    }
};

class BBinaural : public AUEffectBase
{
public:
    BBinaural(AudioUnit component);
    ~BBinaural();
    virtual ComponentResult GetParameterValueStrings
        (AudioUnitScope inScope,
        AudioUnitParameterID inParameterID,
        CFArrayRef * outStrings);
    virtual ComponentResult GetPresets(CFArrayRef *outData)  const;
    virtual OSStatus NewFactoryPresetSet
```

```
        (const AUPreset &inNewFactoryPreset);
    virtual void SetNewSelectedPresetValues(int preset);
    virtual ComponentResult GetParameterInfo
        (AudioUnitScope inScope,
        AudioUnitParameterID inParameterID,
        AudioUnitParameterInfo  &outParameterInfo);
    virtual ComponentResult GetPropertyInfo
        (AudioUnitPropertyID inID,
        AudioUnitScope inScope,
        AudioUnitElement inElement,
        UInt32 & outDataSize,
        Boolean & outWritable);
    virtual ComponentResult GetProperty
        (AudioUnitPropertyID inID,
        AudioUnitScope inScope,
        AudioUnitElement inElement,
        void * outData);
    virtual ComponentResult SetParameter
        (AudioUnitParameterID iID,
        AudioUnitScope iScope,
        AudioUnitElement iElem,
        Float32 rValue,
        UInt32 iSchedule);
    virtual void SetAmbisonicProcessingOrder();
    virtual void SetGlobals(int parameter);
    virtual UInt32 SupportedNumChannels(const AUChannelInfo**);
    virtual OSStatus ProcessBufferLists
        (AudioUnitRenderActionFlags&,
        const AudioBufferList&,
        AudioBufferList&, UInt32);
    virtual bool SupportsTail () { return false; }
    virtual ComponentResult  Version() { return kBBinauralVersion; }
private:
    Speaker* speakerPointer;
    int earType;
    int speakersInPreset;
    int currentPreset;
    int ambisonicProcessingOrder;
    float fHeadphoneAmplitude;
    float fGlobalZerothBase;
    float fGlobalFirstBase;
    float fGlobalSecondBase;
    float fGlobalThirdBase;
    // Used for the FFT process
    UInt32 log2n;
    UInt32 n;
    UInt32 nOver2;
    SInt32 stride;
    FFTSetupD fft1024Setup;
    int bufferCount;
    double scale;
    // Used for scalling the HRTFs
    double maxHRTFvalue;
    double minHRTFvalue;
    double scaleForHRTF;
    // Temporary storage for B-Format Inputs.
    float tempBFormat[kNumberOfInputs];
    // Double complex split containers for use with the FFT
    DOUBLE_COMPLEX_SPLIT earHRTFDoubleComplexSplit
        [numberOfSpeakerRings][numberOfAngles][numberOfEarTypes];
    DOUBLE_COMPLEX_SPLIT virtualSpeakerDoubleComplexSplit
        [numberOfSpeakerRings][numberOfAngles][numberOfEars];
    // Arrays and array pointers for HRTFs, virtual speaker signals, and
    // overlap buffers
    double *dPtrHRTFImpulseResponses
```

```
        [numberOfSpeakerRings][numberOfAngles][numberOfEarTypes];
    double *dPtrVirtualSpeakerTempBuffer
        [numberOfSpeakerRings][numberOfAngles][numberOfEars];
    double dOverLap
        [numberOfSpeakerRings][numberOfAngles][numberOfEars][overlapSize];

    // Methods for storing virtual speaker signals depending on preset
    void StoreSignalsFor2D4(float* fPtrAudioData[], UInt32 frame);
    void StoreSignalsFor2D8(float* fPtrAudioData[], UInt32 frame);
    void StoreSignalsFor3D8(float* fPtrAudioData[], UInt32 frame);
    void StoreSignalsFor3D16(float* fPtrAudioData[], UInt32 frame);
    // Methods for processing virtual speaker preset signals to binaural
    void Process2D4ToBinaural(float* fPtrAudioData[]);
    void Process2D8ToBinaural(float* fPtrAudioData[]);
    void Process3D8ToBinaural(float* fPtrAudioData[]);
    void Process3D16ToBinaural(float* fPtrAudioData[]);
};
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMPONENT_ENTRY(BBinaural)

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural.cpp
//  Aristotel Digenis
//  MSc Music Technology University of York
//  August 27th 2004
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "BBinaural.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::BBinaural
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BBinaural::BBinaural(AudioUnit component) : AUEffectBase(component, false)
{
    // Set I/O stream configuration
    CreateElements();
    CAStreamBasicDescription inputStreamDescription;
    inputStreamDescription.SetCanonical (kNumberOfInputs, false);
    inputStreamDescription.mSampleRate = 44100;
    CAStreamBasicDescription outputStreamDescription;
    outputStreamDescription.SetCanonical (kNumberOfOutputs, false);
    outputStreamDescription.mSampleRate = 44100;
    GetInput(0)->SetStreamFormat(inputStreamDescription);
    GetOutput(0)->SetStreamFormat(outputStreamDescription);
    // Set parameters for FFT processing
    log2n = 10;
    n = 1 << log2n;
    nOver2 = n / 2;
    stride = 1;
    fft1024Setup = create_fftsetupD(log2n, kFFTRadix2);
    bufferCount = 0;
    scale = (double) 1.0 / (2 * n);
    maxHRTFvalue = 0;
    minHRTFvalue = 0;
    scaleForHRTF = 0;

    for(int a = 0; a < numberOfSpeakerRings; a++)
    {
        for(int b = 0; b < numberOfAngles; b++)
        {
            for(int c = 0; c < numberOfEarTypes; c++)
            {
                /* Allocate memory for all the HRTFs to be copied into. A
                HRTF will be used for each speaker position of the largest
                speaker array the plug-in will offer. That will be 3 rings
```

```
                of 8 speakers. Each of the speakers should have a pair of
                HRTFs, for left and right ears. This is not the case here
                as the dummy head used for capturing the impulse responses,
                is symetrical.The impulse responses of a single ear can be
                used with symetry to have HRTFs for both ears. */
                dPtrHRTFImpulseResponses[a][b][c] =
                    ( double* ) malloc (n * sizeof ( double ) );
            }

            for(int d = 0; d < numberOfEars; d++)
            {
                /* Allocate memory for the temporary buffers of the virtual
                Ambisonic speakers.They are twice the size needed to allow
                for zero-padding. That will be 3 rings of 8 speakers, with
                each speaker having two buffers. Each of these two buffers
                will be used to temporarely store the signal that will
                arrive at each of the two ears. */
                dPtrVirtualSpeakerTempBuffer[a][b][d] =
                    ( double* ) malloc (n * sizeof ( double ) );
            }
        }
    }

    /* Copying the 512 samples of the impulse responses for the KEMAR
    normal ear into arrays twice the size to allow for zero-padding. */
    for(int a = 0; a < 512; a++)
    {
        // Normal ear type, upper row.
        dPtrHRTFImpulseResponses[upper][clockwise000][normalEar][a] =
            dNplus40e000a[a];
        dPtrHRTFImpulseResponses[upper][clockwise045][normalEar][a] =
            dNplus40e045a[a];
        dPtrHRTFImpulseResponses[upper][clockwise090][normalEar][a] =
            dNplus40e090a[a];
        dPtrHRTFImpulseResponses[upper][clockwise135][normalEar][a] =
            dNplus40e135a[a];
        dPtrHRTFImpulseResponses[upper][clockwise180][normalEar][a] =
            dNplus40e180a[a];
        dPtrHRTFImpulseResponses[upper][clockwise225][normalEar][a] =
            dNplus40e225a[a];
        dPtrHRTFImpulseResponses[upper][clockwise270][normalEar][a] =
            dNplus40e270a[a];
        dPtrHRTFImpulseResponses[upper][clockwise315][normalEar][a] =
            dNplus40e315a[a];
        // Normal ear type, middle row.
        dPtrHRTFImpulseResponses[middle][clockwise000][normalEar][a] =
            dN00e000a[a];
        dPtrHRTFImpulseResponses[middle][clockwise045][normalEar][a] =
            dN00e045a[a];
        dPtrHRTFImpulseResponses[middle][clockwise090][normalEar][a] =
            dN00e090a[a];
        dPtrHRTFImpulseResponses[middle][clockwise135][normalEar][a] =
            dN00e135a[a];
        dPtrHRTFImpulseResponses[middle][clockwise180][normalEar][a] =
            dN00e180a[a];
        dPtrHRTFImpulseResponses[middle][clockwise225][normalEar][a] =
            dN00e225a[a];
        dPtrHRTFImpulseResponses[middle][clockwise270][normalEar][a] =
            dN00e270a[a];
        dPtrHRTFImpulseResponses[middle][clockwise315][normalEar][a] =
            dN00e315a[a];
        // Normal ear type, lower row.
        dPtrHRTFImpulseResponses[lower][clockwise000][normalEar][a] =
            dNminus40e000a[a];
        dPtrHRTFImpulseResponses[lower][clockwise045][normalEar][a] =
```

```
        dNminus40e045a[a];
    dPtrHRTFImpulseResponses[lower][clockwise090][normalEar][a] =
        dNminus40e090a[a];
    dPtrHRTFImpulseResponses[lower][clockwise135][normalEar][a] =
        dNminus40e135a[a];
    dPtrHRTFImpulseResponses[lower][clockwise180][normalEar][a] =
        dNminus40e180a[a];
    dPtrHRTFImpulseResponses[lower][clockwise225][normalEar][a] =
        dNminus40e225a[a];
    dPtrHRTFImpulseResponses[lower][clockwise270][normalEar][a] =
        dNminus40e270a[a];
    dPtrHRTFImpulseResponses[lower][clockwise315][normalEar][a] =
        dNminus40e315a[a];
    // Large ear type, upper row.
    dPtrHRTFImpulseResponses[upper][clockwise000][largeEar][a] =
        dLplus40e315a[a];
    dPtrHRTFImpulseResponses[upper][clockwise045][largeEar][a] =
        dLplus40e270a[a];
    dPtrHRTFImpulseResponses[upper][clockwise090][largeEar][a] =
        dLplus40e225a[a];
    dPtrHRTFImpulseResponses[upper][clockwise135][largeEar][a] =
        dLplus40e180a[a];
    dPtrHRTFImpulseResponses[upper][clockwise180][largeEar][a] =
        dLplus40e135a[a];
    dPtrHRTFImpulseResponses[upper][clockwise225][largeEar][a] =
        dLplus40e090a[a];
    dPtrHRTFImpulseResponses[upper][clockwise270][largeEar][a] =
        dLplus40e045a[a];
    dPtrHRTFImpulseResponses[upper][clockwise315][largeEar][a] =
        dLplus40e000a[a];
    // Large ear type, middle row.
    dPtrHRTFImpulseResponses[middle][clockwise000][largeEar][a] =
        dL00e315a[a];
    dPtrHRTFImpulseResponses[middle][clockwise045][largeEar][a] =
        dL00e270a[a];
    dPtrHRTFImpulseResponses[middle][clockwise090][largeEar][a] =
        dL00e225a[a];
    dPtrHRTFImpulseResponses[middle][clockwise135][largeEar][a] =
        dL00e180a[a];
    dPtrHRTFImpulseResponses[middle][clockwise180][largeEar][a] =
        dL00e135a[a];
    dPtrHRTFImpulseResponses[middle][clockwise225][largeEar][a] =
        dL00e090a[a];
    dPtrHRTFImpulseResponses[middle][clockwise270][largeEar][a] =
        dL00e045a[a];
    dPtrHRTFImpulseResponses[middle][clockwise315][largeEar][a] =
        dL00e000a[a];
    // Large ear type, lower row.
    dPtrHRTFImpulseResponses[lower][clockwise000][largeEar][a] =
        dLminus40e315a[a];
    dPtrHRTFImpulseResponses[lower][clockwise045][largeEar][a] =
        dLminus40e270a[a];
    dPtrHRTFImpulseResponses[lower][clockwise090][largeEar][a] =
        dLminus40e225a[a];
    dPtrHRTFImpulseResponses[lower][clockwise135][largeEar][a] =
        dLminus40e180a[a];
    dPtrHRTFImpulseResponses[lower][clockwise180][largeEar][a] =
        dLminus40e135a[a];
    dPtrHRTFImpulseResponses[lower][clockwise225][largeEar][a] =
        dLminus40e090a[a];
    dPtrHRTFImpulseResponses[lower][clockwise270][largeEar][a] =
        dLminus40e045a[a];
    dPtrHRTFImpulseResponses[lower][clockwise315][largeEar][a] =
        dLminus40e000a[a];
}
```

```
// Find the min and max value in the HRTFs
for(int a = 0; a < numberOfSpeakerRings; a++)
{
    for(int b = 0; b < numberOfAngles; b++)
    {
        for(int c = 0; c < numberOfEarTypes; c++)
        {
            for(int d = 0; d < 512; d++)
            {
                if(dPtrHRTFImpulseResponses[a][b][c][d] > maxHRTFvalue)
                {
                    maxHRTFvalue =
                        dPtrHRTFImpulseResponses[a][b][c][d];
                }
                if(dPtrHRTFImpulseResponses[a][b][c][d] < minHRTFvalue)
                {
                    minHRTFvalue =
                        dPtrHRTFImpulseResponses[a][b][c][d];
                }
            }
        }
    }
}

minHRTFvalue = (-minHRTFvalue);

if(maxHRTFvalue >= minHRTFvalue)
{
    scaleForHRTF = 1.0 / maxHRTFvalue;
}
else
{
    scaleForHRTF = 1.0 / minHRTFvalue;
}

// Scale HRTFs to have a range up to 1.0
for(int a = 0; a < numberOfSpeakerRings; a++)
{
    for(int b = 0; b < numberOfAngles; b++)
    {
        for(int c = 0; c < numberOfEarTypes; c++)
        {
            for(int d = 0; d < 512; d++)
            {
                dPtrHRTFImpulseResponses[a][b][c][d] *= scaleForHRTF;
            }
        }
    }
}

// Zero-pad the second half of the HRTF arrays
for(int a = 0; a < numberOfSpeakerRings; a++)
{
    for(int b = 0; b < numberOfAngles; b++)
    {
        for(int c = 0; c < numberOfEarTypes; c++)
        {
            for(int d = 512; d < 1024; d++)
            {
                dPtrHRTFImpulseResponses[a][b][c][d] = 0.0;
            }
        }
    }
}
```

```
for(int a = 0; a < numberOfSpeakerRings; a++)
{
    for(int b = 0; b < numberOfAngles; b++)
    {
        for(int c = 0; c < numberOfEarTypes; c++)
        {
            // Allocate memory for Double Split Complex objects, for
            // the HRTFs
            earHRTFDoubleComplexSplit[a][b][c].realp =
                (double*) malloc(nOver2 * sizeof(double));
            earHRTFDoubleComplexSplit[a][b][c].imagp =
                (double*) malloc(nOver2 * sizeof(double));
            // Copy the normal HRTF arrays into the Double Complex
            // Split objects and convert to odd-even format
            ctozD((DOUBLE_COMPLEX *) dPtrHRTFImpulseResponses[a][b][c],
                2, &earHRTFDoubleComplexSplit[a][b][c], 1, nOver2);
            /* Convert the HRTF data inside the Double Split Complex
            objects to the frequency domain, storing them in the same
            Double Split Complex object */
            fft_zripD(fft1024Setup,&earHRTFDoubleComplexSplit[a][b][c],
                stride, log2n, kFFTDirection_Forward);
        }
        for(int d = 0; d < numberOfEars; d++)
        {
            /* Allocate memory for Double Split Complex objects that
            will later be used for the virtual Ambisonic speakers */
            virtualSpeakerDoubleComplexSplit[a][b][d].realp =
                (double*) malloc(nOver2 * sizeof(double));
            virtualSpeakerDoubleComplexSplit[a][b][d].imagp =
                (double*) malloc(nOver2 * sizeof(double));
            /* Allocate memory for Double Split Complex objects that
            will later be used for storing the result of the
            convolution */
        }
    }
}
// Zero pad all tempoprary speaker buffers (initialize)
for(int a = 0; a < numberOfSpeakerRings; a++)
{
    for(int b = 0; b < numberOfAngles; b++)
    {
        for(int c = 0; c < numberOfEars; c++)
        {
            for(int d = 0; d < 1024; d++)
            {
                dPtrVirtualSpeakerTempBuffer[a][b][c][d] = 0.0;
            }
        }
    }
}
// Zero pad all contents of the overlap buffer
for(int a = 0; a < numberOfSpeakerRings; a++)
{
    for(int b = 0; b < numberOfAngles; b++)
    {
        for(int c = 0; c < numberOfEars; c++)
        {
            for(int d = 0; d < overlapSize; d++)
            {
                dOverLap[a][b][c][d] = 0.0;
            }
        }
    }
}
```

```
    speakerPointer = new Speaker[kNumberOfSpeakers];
    ambisonicProcessingOrder = 1;
    earType = normalEar;
    fHeadphoneAmplitude = 1.0;
    speakersInPreset = 16;
    currentPreset = 0;

    fGlobalZerothBase = ambisonicOrderBases[ambisonicProcessingOrder][0];
    fGlobalFirstBase = ambisonicOrderBases[ambisonicProcessingOrder][1];
    fGlobalSecondBase = ambisonicOrderBases[ambisonicProcessingOrder][2];
    fGlobalThirdBase = ambisonicOrderBases[ambisonicProcessingOrder][3];

    if (!sLocalized)
    {
        CFBundleRef bundle = CFBundleGetBundleWithIdentifier
            (CFSTR("com.dige.audiounit.bbin"));

        if (bundle != NULL)
        {
            for (int i = 0; i < kNumberPresets; i++)
            {
                kPresets[i].presetName =
                    CFCopyLocalizedStringFromTableInBundle
                    (kPresets[i].presetName,
                    CFSTR("Localizable"), bundle, CFSTR(""));
            }

            for(int a = 0; a < numberOfParameters; a++)
            {
                kParameterStrings[a] =
                    CFCopyLocalizedStringFromTableInBundle
                    (kParameterStrings[a],
                    CFSTR("Localizable"), bundle, CFSTR(""));
            }
        }
        sLocalized = true; //so never pass the test again...
    }
    // Set the default preset
    SetAFactoryPresetAsCurrent (kPresets[kPresetDefaultIndex]);
    NewFactoryPresetSet(kPresets[quad]);
    // Set the parameters
    SetParameter(ambisonicOrder, kAudioUnitScope_Global, 0,
        ambisonicProcessingOrder, 0);
    SetParameter(typeOfEar, kAudioUnitScope_Global, 0, earType, 0);
    SetParameter(headphoneAmplitude, kAudioUnitScope_Global, 0,
        10 * log10(fHeadphoneAmplitude), 0);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::~BBinaural
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BBinaural::~BBinaural()
{
    if(speakerPointer)
    {
        delete speakerPointer;
    }
    if(fft1024Setup)
    {
        destroy_fftsetupD(fft1024Setup);
    }
    if(earHRTFDoubleComplexSplit[numberOfSpeakerRings][numberOfAngles]
        [numberOfEarTypes].realp)
    {
```

```
            delete earHRTFDoubleComplexSplit[numberOfSpeakerRings]
                [numberOfAngles][numberOfEarTypes].realp;
        }
        if(earHRTFDoubleComplexSplit[numberOfSpeakerRings][numberOfAngles]
            [numberOfEarTypes].imagp)
        {
            delete earHRTFDoubleComplexSplit[numberOfSpeakerRings]
                [numberOfAngles][numberOfEarTypes].imagp;
        }
        if(virtualSpeakerDoubleComplexSplit[numberOfSpeakerRings]
            [numberOfAngles][numberOfEars].realp)
        {
            delete virtualSpeakerDoubleComplexSplit[numberOfSpeakerRings]
                [numberOfAngles][numberOfEars].realp;
        }
        if(virtualSpeakerDoubleComplexSplit[numberOfSpeakerRings]
            [numberOfAngles][numberOfEars].imagp)
        {
            delete virtualSpeakerDoubleComplexSplit[numberOfSpeakerRings]
                [numberOfAngles][numberOfEars].imagp;
        }
        if(dPtrHRTFImpulseResponses[numberOfSpeakerRings]
            [numberOfAngles][numberOfEarTypes])
        {
            delete dPtrHRTFImpulseResponses[numberOfSpeakerRings]
                [numberOfAngles][numberOfEarTypes];
        }
        if(dPtrVirtualSpeakerTempBuffer[numberOfSpeakerRings]
            [numberOfAngles][numberOfEars])
        {
            delete dPtrVirtualSpeakerTempBuffer[numberOfSpeakerRings]
                [numberOfAngles][numberOfEars];
        }
    }

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //  BBinaural::GetPresets
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ComponentResult BBinaural::GetPresets(CFArrayRef *outData) const
    {
        if(outData == NULL)
        {
            return noErr;
        }

        CFMutableArrayRef theArray = CFArrayCreateMutable
            (NULL, kNumberPresets, NULL);

        for(int i = 0; i < kNumberPresets; ++i)
        {
            CFArrayAppendValue (theArray, &kPresets[i]);
        }

        *outData = (CFArrayRef)theArray;

        return noErr;
    }

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //  BBinaural::NewFactoryPresetSet
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    OSStatus BBinaural::NewFactoryPresetSet(const AUPreset &inNewFactoryPreset)
    {
        SInt32 chosenPreset = inNewFactoryPreset.presetNumber;
```

```
    switch(chosenPreset)
    {
        case quad:
            currentPreset = quad;
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
            speakersInPreset = (int) allPresets[quad][0][0];
            SetNewSelectedPresetValues(quad);
            return noErr;
            break;
        case octagon:
            currentPreset = octagon;
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
            speakersInPreset = (int) allPresets[octagon][0][0];
            SetNewSelectedPresetValues(octagon);
            return noErr;
            break;
        case threeD8:
            currentPreset = threeD8;
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
            speakersInPreset = (int) allPresets[threeD8][0][0];
            SetNewSelectedPresetValues(threeD8);
            return noErr;
            break;
        case threeD16:
            currentPreset = threeD16;
            SetAFactoryPresetAsCurrent (kPresets[chosenPreset]);
            speakersInPreset = (int) allPresets[threeD16][0][0];
            SetNewSelectedPresetValues(threeD16);
            return noErr;
            break;
    }

    return kAudioUnitErr_InvalidPropertyValue;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::SetNewSelectedPresetValues
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::SetNewSelectedPresetValues(int preset)
{
    for(int a = 0; a < speakersInPreset; a++)
    {
        speakerPointer[a].SetAzimuth(allPresets[preset][1][a]);
        speakerPointer[a].SetElevation(allPresets[preset][2][a]);
        speakerPointer[a].SetAmplitude(fDefaultAmplitude);
        speakerPointer[a].SetZerothBase(fGlobalZerothBase);
        speakerPointer[a].SetFirstBase(fGlobalFirstBase);
        speakerPointer[a].SetSecondBase(fGlobalSecondBase);
        speakerPointer[a].SetThirdBase(fGlobalThirdBase);
    }

    for(int a = speakersInPreset; a < kNumberOfSpeakers; a++)
    {
        speakerPointer[a].SetAzimuth(0.0);
        speakerPointer[a].SetElevation(0.0);
        speakerPointer[a].SetDistance(0.0);
        speakerPointer[a].SetAmplitude(0.0);
        speakerPointer[a].SetZerothBase(0.0);
        speakerPointer[a].SetFirstBase(0.0);
        speakerPointer[a].SetSecondBase(0.0);
        speakerPointer[a].SetThirdBase(0.0);
    }

    for(int a = 0; a < kNumberOfSpeakers; a++)
    {
```

```
            speakerPointer[a].CalculateSpeakerZerothBase();
            speakerPointer[a].CalculateSpeakerFirstBase();
            speakerPointer[a].CalculateSpeakerSecondBase();
            speakerPointer[a].CalculateSpeakerThirdBase();
            speakerPointer[a].CalculateAzimuthInRadians();
            speakerPointer[a].CalculateElevationInRadians();
            speakerPointer[a].CalculateSpeakerCoefficients();
        }
    }

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //  BBinaural::GetParameterValueStrings
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ComponentResult BBinaural::GetParameterValueStrings
        (AudioUnitScope inScope,
        AudioUnitParameterID inParameterID,
        CFArrayRef * outStrings)
    {
        if(inScope == kAudioUnitScope_Global)
        {
            if(outStrings == NULL)
            {
                return noErr;
            }

            if(inParameterID == ambisonicOrder)
            {
                CFStringRef strings[4];

                for(int a = 0; a < 4; a++)
                {
                    strings[a] = kAmbisonicOrderIndexParameterStrings[a];
                }

                *outStrings = CFArrayCreate( NULL,
                    (const void **)strings, 4, NULL);

                return noErr;
            }

            if(inParameterID == typeOfEar)
            {
                CFStringRef strings[2];

                for(int a = 0; a < 2; a++)
                {
                    strings[a] = kEarTypeIndexParameterStrings[a];
                }

                *outStrings = CFArrayCreate( NULL,
                    (const void **)strings, 2, NULL);

                return noErr;
            }
        }

        return kAudioUnitErr_InvalidProperty;
    }

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //  BBinaural::GetParameterInfo
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ComponentResult BBinaural::GetParameterInfo
        (AudioUnitScope inScope,
        AudioUnitParameterID inParameterID,
```

```
    AudioUnitParameterInfo &outParameterInfo)
{
    ComponentResult result = noErr;

    outParameterInfo.flags = kAudioUnitParameterFlag_IsWritable |
        kAudioUnitParameterFlag_IsReadable;

    if (inScope == kAudioUnitScope_Global)
    {
        switch(inParameterID)
        {
            case ambisonicOrder:
                AUBase::FillInParameterName (outParameterInfo,
                    kParameterStrings[ambisonicOrder], false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Indexed;
                outParameterInfo.minValue = 0;
                outParameterInfo.maxValue = 3;
                outParameterInfo.defaultValue = ambisonicProcessingOrder;
                break;
            case typeOfEar:
                AUBase::FillInParameterName (outParameterInfo,
                    kParameterStrings[typeOfEar], false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Indexed;
                outParameterInfo.minValue = 0;
                outParameterInfo.maxValue = 1;
                outParameterInfo.defaultValue = normalEar;
                break;
            case headphoneAmplitude:
                AUBase::FillInParameterName (outParameterInfo,
                    kParameterStrings[headphoneAmplitude], false);
                outParameterInfo.unit = kAudioUnitParameterUnit_Decibels;
                outParameterInfo.minValue = kMin_dB_Value;
                outParameterInfo.maxValue = 6.0;
                outParameterInfo.defaultValue =
                    10 * log10(fHeadphoneAmplitude);
                outParameterInfo.flags |=
                    kAudioUnitParameterFlag_ValuesHaveStrings;
                break;
            default:
                result = kAudioUnitErr_InvalidParameter;
                break;
        }
    }
    else
    {
        result = kAudioUnitErr_InvalidParameter;
    }

    return result;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::GetPropertyInfo
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BBinaural::GetPropertyInfo(AudioUnitPropertyID inID,
AudioUnitScope inScope, AudioUnitElement inElement, UInt32 & outDataSize,
Boolean & outWritable)
{
    if (inScope == kAudioUnitScope_Global)
    {
        switch (inID)
        {
            case kAudioUnitProperty_IconLocation:
                outWritable = false;
                outDataSize = sizeof (CFURLRef);
```

```
                return noErr;

            case kAudioUnitProperty_ParameterStringFromValue:
                outWritable = false;
                outDataSize = sizeof (AudioUnitParameterStringFromValue);
                return noErr;

            case kAudioUnitProperty_ParameterValueFromString:
                outWritable = false;
                outDataSize = sizeof (AudioUnitParameterValueFromString);
                return noErr;
        }
    }

    return AUEffectBase::GetPropertyInfo
        (inID, inScope, inElement, outDataSize, outWritable);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::GetProperty
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BBinaural::GetProperty
    (AudioUnitPropertyID inID,
    AudioUnitScope inScope,
    AudioUnitElement inElement,
    void * outData)
{
    if (inScope == kAudioUnitScope_Global)
    {
        switch (inID)
        {
            case kAudioUnitProperty_IconLocation:
            {
                CFBundleRef bundle = CFBundleGetBundleWithIdentifier
                    (CFSTR("com.dige.audiounit.bbin"));

                if (bundle == NULL)
                {
                    return fnfErr;
                }

                CFURLRef bundleURL = CFBundleCopyResourceURL
                    (bundle, CFSTR("Icon"), CFSTR(".icns"), NULL);

                if (bundleURL == NULL)
                {
                    return fnfErr;
                }

                (*(CFURLRef *)outData) = bundleURL;

                return noErr;
            }
            case kAudioUnitProperty_ParameterValueFromString:
            {
                OSStatus retVal = kAudioUnitErr_InvalidPropertyValue;
                AudioUnitParameterValueFromString &name =
                    *(AudioUnitParameterValueFromString*)outData;

                UniChar chars[2];
                chars[0] = '-';
                chars[1] = 0x221e; // this is the unicode symbol for
                                   // infinity
                CFStringRef comparisonString =
                    CFStringCreateWithCharacters (NULL, chars, 2);
```

```
                if ( CFStringCompare(comparisonString, name.inString, 0) ==
                    kCFCompareEqualTo )
                {
                    name.outValue = kMin_dB_Value;
                    retVal = noErr;
                }

                if (comparisonString)
                {
                    CFRelease(comparisonString);
                }

                return retVal;
            }

            case kAudioUnitProperty_ParameterStringFromValue:
            {
                AudioUnitParameterStringFromValue &name =
                    *(AudioUnitParameterStringFromValue*)outData;
                Float32 paramValue = (name.inValue == NULL ? GetParameter
                    (headphoneAmplitude) : *(name.inValue));

                // for this usage only values <= -120 dB (the min value)
                // have a special name "-infinity"
                if (paramValue <= kMin_dB_Value)
                {
                    UniChar chars[2];
                    chars[0] = '-';
                    chars[1] = 0x221e; // this is the unicode symbol for
                                       // infinity
                    name.outString = CFStringCreateWithCharacters
                    (NULL, chars, 2);
                }
                else
                {
                    name.outString = NULL;
                }

                return noErr;
            }

            case kAudioUnitProperty_SupportedNumChannels:
            {
                return noErr;
            }
        }
    }
    return AUEffectBase::GetProperty (inID, inScope, inElement, outData);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::SetParameter
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ComponentResult BBinaural::SetParameter
    (AudioUnitParameterID iID,
    AudioUnitScope iScope,
    AudioUnitElement iElem,
    Float32 rValue,
    UInt32 iSchedule)
{
    // Set a new value for a parameter.
    if (iScope==kAudioUnitScope_Global && iElem==0)
    {
        switch (iID)
```

```
        {
            case ambisonicOrder:
                ambisonicProcessingOrder = (int) rValue;
                SetAmbisonicProcessingOrder();
                break;
            case typeOfEar:
                earType = (int) rValue;
                break;
            case headphoneAmplitude:
                fHeadphoneAmplitude = (pow(10, rValue / 10));
                break;
        }
    }

    return AUBase::SetParameter(iID, iScope, iElem, rValue, iSchedule);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::SetAmbisonicProcessingOrder
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::SetAmbisonicProcessingOrder()
{
    fGlobalZerothBase = ambisonicOrderBases[ambisonicProcessingOrder][0];
    fGlobalFirstBase = ambisonicOrderBases[ambisonicProcessingOrder][1];
    fGlobalSecondBase = ambisonicOrderBases[ambisonicProcessingOrder][2];
    fGlobalThirdBase = ambisonicOrderBases[ambisonicProcessingOrder][3];

    SetGlobals(globalZerothBase);
    SetGlobals(globalFirstBase);
    SetGlobals(globalSecondBase);
    SetGlobals(globalThirdBase);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::SetGlobals
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::SetGlobals(int parameter)
{
    switch(parameter)
    {
        case globalZerothBase:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetZerothBase(fGlobalZerothBase);
            }
            break;
        case globalFirstBase:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetFirstBase(fGlobalFirstBase);
            }
            break;
        case globalSecondBase:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetSecondBase(fGlobalSecondBase);
            }
            break;
        case globalThirdBase:
            for(int a = 0; a < kNumberOfOutputs; a++)
            {
                speakerPointer[a].SetThirdBase(fGlobalThirdBase);
            }
            break;
    }
```

```
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::SupportedNumChannel
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UInt32 BBinaural::SupportedNumChannels(const AUChannelInfo** outInfo)
{
    UInt32 count = 0;
    for(; channelInfoPointer && (channelInfoPointer[count].inChannels != 0
        || channelInfoPointer[count].outChannels != 0); count++){};

    if(outInfo)
    {
        *outInfo = channelInfoPointer;
    }

    return count;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::ProcessBufferLists
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
OSStatus BBinaural::ProcessBufferLists
    (AudioUnitRenderActionFlags& iFlags,
    const AudioBufferList& inBufferList,
    AudioBufferList& outBufferList,
    UInt32 iFrames)
{
    // Array of pointers, as many as input signals(B-Format).
    float* audioData[kNumberOfInputs];
    // Pointers point to incomming audio buffers.
    for(int i = 0; i < kNumberOfInputs; i++)
    {
        audioData[i] = (float*) inBufferList.mBuffers[i].mData;
    }
    // For zeroth order processing.
    if(ambisonicProcessingOrder == 0)
    {
        // For all the frames.
        for(UInt32 j = 0; j < iFrames; j++)
        {
            // Store the values of the B-Format inputs for that frame.
            for(int a = 0; a < kNumberOfInputs; a++)
            {
                tempBFormat[a] = audioData[a][j];
            }
            // Derive the output for all the speakers in the preset and
            // store it in the buffer.
            for(int speakerCounter = 0; speakerCounter < speakersInPreset;
                speakerCounter++)
            {
                audioData[speakerCounter][j] = (float)
                (tempBFormat[W] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(W));
            }
            // Store speaker feeds into temporary buffers. Each speaker is
            // stored twice, once for each ear.
            if(bufferCount < 512)
            {
                switch(currentPreset)
                {
                    case quad:
                        StoreSignalsFor2D4(audioData, j);
                        break;
```

```
                    case octagon:
                        StoreSignalsFor2D8(audioData, j);
                        break;
                    case threeD8:
                        StoreSignalsFor3D8(audioData, j);
                        break;
                    case threeD16:
                        StoreSignalsFor3D16(audioData, j);
                        break;
                }

                bufferCount++;
            }
            // Once 512 samples are collected.
            if(bufferCount == 512)
            {
                switch(currentPreset)
                {
                    case quad:
                        Process2D4ToBinaural(audioData);
                        break;
                    case octagon:
                        Process2D8ToBinaural(audioData);
                        break;
                    case threeD8:
                        Process3D8ToBinaural(audioData);
                        break;
                    case threeD16:
                        Process3D16ToBinaural(audioData);
                        break;
                }
                // Reset sample counter.
                bufferCount = 0;

            }
            // Zero pad the the buffers of ouputs that will not be used.
            for(int speakerCounter = speakersInPreset; speakerCounter <
                kNumberOfSpeakers; speakerCounter++)
            {
                audioData[speakerCounter][j] = 0.0;
            }
        }
    }
    // For first order processing.
    if(ambisonicProcessingOrder == 1)
    {
        // For all the frames.
        for(UInt32 j = 0; j < iFrames; j++)
        {
            // Store the values of the B-Format inputs for that frame.
            for(int a = 0; a < kNumberOfInputs; a++)
            {
                tempBFormat[a] = audioData[a][j];
            }
            // Derive the output for all the speakers in the preset and
            // store it in the buffer.
            for(int speakerCounter = 0; speakerCounter < speakersInPreset;
                speakerCounter++)
            {
                audioData[speakerCounter][j] = (float)
                (tempBFormat[W] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(W)) +
                (tempBFormat[X] *
                    speakerPointer[speakerCounter].
```

```
                    GetSpeakerCoefficient(X)) +
                (tempBFormat[Y] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(Y)) +
                (tempBFormat[Z] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(Z));
            }
            // Store speaker feeds into temporary buffers. Each speaker is
            // stored twice, once for each ear.
            if(bufferCount < 512)
            {
                switch(currentPreset)
                {
                    case quad:
                        StoreSignalsFor2D4(audioData, j);
                        break;
                    case octagon:
                        StoreSignalsFor2D8(audioData, j);
                        break;
                    case threeD8:
                        StoreSignalsFor3D8(audioData, j);
                        break;
                    case threeD16:
                        StoreSignalsFor3D16(audioData, j);
                        break;
                }

                bufferCount++;
            }
            // Once 512 samples are collected.
            if(bufferCount == 512)
            {
                switch(currentPreset)
                {
                    case quad:
                        Process2D4ToBinaural(audioData);
                        break;
                    case octagon:
                        Process2D8ToBinaural(audioData);
                        break;
                    case threeD8:
                        Process3D8ToBinaural(audioData);
                        break;
                    case threeD16:
                        Process3D16ToBinaural(audioData);
                        break;
                }
                // Reset sample counter.
                bufferCount = 0;

            }
            // Zero pad the the buffers of ouputs that will not be used.
            for(int speakerCounter = speakersInPreset; speakerCounter <
                kNumberOfSpeakers; speakerCounter++)
            {
                audioData[speakerCounter][j] = 0.0;
            }
        }
    }
    // For second order processing.
    if(ambisonicProcessingOrder == 2)
    {
        // For all the frames.
        for(UInt32 j = 0; j < iFrames; j++)
```

```
{
    // Store the values of the B-Format inputs for that frame.
    for(int a = 0; a < kNumberOfInputs; a++)
    {
        tempBFormat[a] = audioData[a][j];
    }
    // Derive the output for all the speakers in the preset and
    // store it in the buffer.
    for(int speakerCounter = 0; speakerCounter < speakersInPreset;
        speakerCounter++)
    {
        audioData[speakerCounter][j] = (float)
        (tempBFormat[W] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(W)) +
        (tempBFormat[X] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(X)) +
        (tempBFormat[Y] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(Y)) +
        (tempBFormat[Z] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(Z)) +
        (tempBFormat[R] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(R)) +
        (tempBFormat[S] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(S)) +
        (tempBFormat[T] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(T)) +
        (tempBFormat[U] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(U)) +
        (tempBFormat[V] *
            speakerPointer[speakerCounter].
            GetSpeakerCoefficient(V));
    }
    // Store speaker feeds into temporary buffers. Each speaker is
    // stored twice, once for each ear.
    if(bufferCount < 512)
    {
        switch(currentPreset)
        {
            case quad:
                StoreSignalsFor2D4(audioData, j);
                break;
            case octagon:
                StoreSignalsFor2D8(audioData, j);
                break;
            case threeD8:
                StoreSignalsFor3D8(audioData, j);
                break;
            case threeD16:
                StoreSignalsFor3D16(audioData, j);
                break;
        }

        bufferCount++;
    }
    // Once 512 samples are collected.
    if(bufferCount == 512)
    {
```

```
                switch(currentPreset)
                {
                    case quad:
                        Process2D4ToBinaural(audioData);
                        break;
                    case octagon:
                        Process2D8ToBinaural(audioData);
                        break;
                    case threeD8:
                        Process3D8ToBinaural(audioData);
                        break;
                    case threeD16:
                        Process3D16ToBinaural(audioData);
                        break;
                }
                // Reset sample counter.
                bufferCount = 0;

            }
            // Zero pad the the buffers of ouputs that will not be used.
            for(int speakerCounter = speakersInPreset; speakerCounter <
                kNumberOfSpeakers; speakerCounter++)
            {
                audioData[speakerCounter][j] = 0.0;
            }
        }
    }
    // For third order processing.
    if(ambisonicProcessingOrder == 3)
    {
        // For all the frames.
        for(UInt32 j = 0; j < iFrames; j++)
        {
            // Store the values of the B-Format inputs for that frame.
            for(int a = 0; a < kNumberOfInputs; a++)
            {
                tempBFormat[a] = audioData[a][j];
            }
            // Derive the output for all the speakers in the preset and
            // store it in the buffer.
            for(int speakerCounter = 0; speakerCounter < speakersInPreset;
                speakerCounter++)
            {
                audioData[speakerCounter][j] = (float)
                (tempBFormat[W] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(W)) +
                (tempBFormat[X] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(X)) +
                (tempBFormat[Y] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(Y)) +
                (tempBFormat[Z] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(Z)) +
                (tempBFormat[R] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(R)) +
                (tempBFormat[S] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(S)) +
                (tempBFormat[T] *
                    speakerPointer[speakerCounter].
                    GetSpeakerCoefficient(T)) +
```

```
            (tempBFormat[U] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(U)) +
            (tempBFormat[V] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(V)) +
            (tempBFormat[K] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(K)) +
            (tempBFormat[L] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(L)) +
            (tempBFormat[M] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(M)) +
            (tempBFormat[N] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(N)) +
            (tempBFormat[O] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(O)) +
            (tempBFormat[P] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(P)) +
            (tempBFormat[Q] *
                speakerPointer[speakerCounter].
                GetSpeakerCoefficient(Q));
        }
        // Store speaker feeds into temporary buffers. Each speaker is
        // stored twice, once for each ear.
        if(bufferCount < 512)
        {
            switch(currentPreset)
            {
                case quad:
                    StoreSignalsFor2D4(audioData, j);
                    break;
                case octagon:
                    StoreSignalsFor2D8(audioData, j);
                    break;
                case threeD8:
                    StoreSignalsFor3D8(audioData, j);
                    break;
                case threeD16:
                    StoreSignalsFor3D16(audioData, j);
                    break;
            }

            bufferCount++;
        }
        // Once 512 samples are collected.
        if(bufferCount == 512)
        {
            switch(currentPreset)
            {
                case quad:
                    Process2D4ToBinaural(audioData);
                    break;
                case octagon:
                    Process2D8ToBinaural(audioData);
                    break;
                case threeD8:
                    Process3D8ToBinaural(audioData);
                    break;
                case threeD16:
```

```
                        Process3D16ToBinaural(audioData);
                        break;
                }
                // Reset sample counter.
                bufferCount = 0;

            }
            // Zero pad the the buffers of ouputs that will not be used.
            for(int speakerCounter = speakersInPreset; speakerCounter <
                kNumberOfSpeakers; speakerCounter++)
            {
                audioData[speakerCounter][j] = 0.0;
            }
        }
    }

    for(int i = 0; i < kNumberOfOutputs; i++)
    {
        outBufferList.mBuffers[i].mData = audioData[i];
    }

    return noErr;
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::StoreSignalsFor2D4
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::StoreSignalsFor2D4(float* fPtrAudioData[], UInt32 frame)
{
    // Middle speaker row, left ear
    dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][bufferCount] =
        fPtrAudioData[0][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][bufferCount] =
        fPtrAudioData[1][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][bufferCount] =
        fPtrAudioData[2][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][bufferCount] =
        fPtrAudioData[3][frame];
    // Middle speaker row, right ear
    dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][bufferCount]
        = fPtrAudioData[0][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][bufferCount]
        = fPtrAudioData[1][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][bufferCount]
        = fPtrAudioData[2][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][bufferCount]
        = fPtrAudioData[3][frame];
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::StoreSignalsFor2D8
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::StoreSignalsFor2D8(float* fPtrAudioData[], UInt32 frame)
{
    // Middle speaker row, left ear
    dPtrVirtualSpeakerTempBuffer[middle][clockwise000][left][bufferCount] =
        (double) fPtrAudioData[0][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][bufferCount] =
        (double) fPtrAudioData[1][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise090][left][bufferCount] =
        (double) fPtrAudioData[2][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][bufferCount] =
        (double) fPtrAudioData[3][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise180][left][bufferCount] =
        (double) fPtrAudioData[4][frame];
```

```
    dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][bufferCount] =
        (double) fPtrAudioData[5][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise270][left][bufferCount] =
        (double) fPtrAudioData[6][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][bufferCount] =
        (double) fPtrAudioData[7][frame];
    // Middle speaker row, right ear
    dPtrVirtualSpeakerTempBuffer[middle][clockwise000][right][bufferCount]
        = (double) fPtrAudioData[0][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][bufferCount]
        = (double) fPtrAudioData[1][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise090][right][bufferCount]
        = (double) fPtrAudioData[2][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][bufferCount]
        = (double) fPtrAudioData[3][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise180][right][bufferCount]
        = (double) fPtrAudioData[4][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][bufferCount]
        = (double) fPtrAudioData[5][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise270][right][bufferCount]
        = (double) fPtrAudioData[6][frame];
    dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][bufferCount]
        = (double) fPtrAudioData[7][frame];
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::StoreSignalsFor3D8
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::StoreSignalsFor3D8(float* fPtrAudioData[], UInt32 frame)
{
    // Upper speaker row, left ear
    dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][bufferCount] =
        fPtrAudioData[0][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][bufferCount] =
        fPtrAudioData[1][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][bufferCount] =
        fPtrAudioData[2][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][bufferCount] =
        fPtrAudioData[3][frame];
    // Lower speaker row, left ear
    dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][bufferCount] =
        fPtrAudioData[4][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][bufferCount] =
        fPtrAudioData[5][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][bufferCount] =
        fPtrAudioData[6][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][bufferCount] =
        fPtrAudioData[7][frame];
    // Upper speaker row, right ear
    dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][bufferCount] =
        fPtrAudioData[0][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][bufferCount] =
        fPtrAudioData[1][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][bufferCount] =
        fPtrAudioData[2][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][bufferCount] =
        fPtrAudioData[3][frame];
    // Lower speaker row, right ear
    dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][bufferCount] =
        fPtrAudioData[4][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][bufferCount] =
        fPtrAudioData[5][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][bufferCount] =
        fPtrAudioData[6][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][bufferCount] =
```

```
        fPtrAudioData[7][frame];
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::StoreSignalsFor3D16
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::StoreSignalsFor3D16(float* fPtrAudioData[], UInt32 frame)
{
    // Upper speaker row, left ear
    dPtrVirtualSpeakerTempBuffer[upper][clockwise000][left][bufferCount] =
        (double) fPtrAudioData[0][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][bufferCount] =
        (double) fPtrAudioData[1][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise090][left][bufferCount] =
        (double) fPtrAudioData[2][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][bufferCount] =
        (double) fPtrAudioData[3][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise180][left][bufferCount] =
        (double) fPtrAudioData[4][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][bufferCount] =
        (double) fPtrAudioData[5][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise270][left][bufferCount] =
        (double) fPtrAudioData[6][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][bufferCount] =
        (double) fPtrAudioData[7][frame];
    // Lower speaker row, left ear
    dPtrVirtualSpeakerTempBuffer[lower][clockwise000][left][bufferCount] =
        (double) fPtrAudioData[8][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][bufferCount] =
        (double) fPtrAudioData[9][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise090][left][bufferCount] =
        (double) fPtrAudioData[10][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][bufferCount] =
        (double) fPtrAudioData[11][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise180][left][bufferCount] =
        (double) fPtrAudioData[12][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][bufferCount] =
        (double) fPtrAudioData[13][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise270][left][bufferCount] =
        (double) fPtrAudioData[14][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][bufferCount] =
        (double) fPtrAudioData[15][frame];
    // Upper speaker row, right ear
    dPtrVirtualSpeakerTempBuffer[upper][clockwise000][right][bufferCount] =
        (double) fPtrAudioData[0][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][bufferCount] =
        (double) fPtrAudioData[1][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise090][right][bufferCount] =
        (double) fPtrAudioData[2][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][bufferCount] =
        (double) fPtrAudioData[3][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise180][right][bufferCount] =
        (double) fPtrAudioData[4][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][bufferCount] =
        (double) fPtrAudioData[5][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise270][right][bufferCount] =
        (double) fPtrAudioData[6][frame];
    dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][bufferCount] =
        (double) fPtrAudioData[7][frame];
    // Lower speaker row, right ear
    dPtrVirtualSpeakerTempBuffer[lower][clockwise000][right][bufferCount] =
        (double) fPtrAudioData[8][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][bufferCount] =
        (double) fPtrAudioData[9][frame];
    dPtrVirtualSpeakerTempBuffer[lower][clockwise090][right][bufferCount] =
```

```
            (double) fPtrAudioData[10][frame];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][bufferCount] =
            (double) fPtrAudioData[11][frame];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise180][right][bufferCount] =
            (double) fPtrAudioData[12][frame];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][bufferCount] =
            (double) fPtrAudioData[13][frame];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise270][right][bufferCount] =
            (double) fPtrAudioData[14][frame];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][bufferCount] =
            (double) fPtrAudioData[15][frame];
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::Process2D4ToBinaural
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::Process2D4ToBinaural(float* fPtrAudioData[])
{
    // Zero pad the remaining half of the temporary speaker buffers.
    for(int a = 512; a < 1024; a++)
    {
        dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][a] = 0.0;
    }
    /* Copy the temporary speaker buffers for the left ear,
    into the Double Complex Split objects and convert to odd-even format.*/
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left], 2,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left], 2,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left], 2,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left], 2,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], 1, nOver2);
    /* Copy the temporary speaker buffers for the right ear,
    into the Double Complex Split objects and convert to odd-even format.*/
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right], 2,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right], 2,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right], 2,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right], 2,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right], 1, nOver2);
    /* Convert left ear Double Split Complex objects to the frequency
domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
```

```
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], stride,
log2n, kFFTDirection_Forward);
    /* Convert right ear Double Split Complex objects to the frequency
domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right], stride,
log2n, kFFTDirection_Forward);
    // Multiply the left ear feeds with the corresponding HRTFs.
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], stride,
nOver2, 1);
    // Multiply the right ear feeds with the corresponding HRTFs.
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right], stride,
nOver2, 1);
    /* Convert left ear Double Split Complex objects to the time domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], stride,
log2n, kFFTDirection_Inverse);
```

```
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], stride,
log2n, kFFTDirection_Inverse);
    /* Convert right ear Double Split Complex objects to the time domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right], stride,
log2n, kFFTDirection_Inverse);
    // Scale the left ear signals stored in Double Split Complex objects.
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise045][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise045][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise045][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise045][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise135][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise135][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise135][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise135][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise225][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise225][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise225][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise225][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise315][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise315][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise315][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise315][left].imagp, 1,
nOver2 );
    // Scale the right ear signals stored in Double Split Complex objects.
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise045][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise045][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise045][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise045][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise135][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise135][right].realp, 1,
```

```
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise135][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise135][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise225][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise225][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise225][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise225][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise315][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise315][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise315][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise315][right].imagp, 1,
nOver2 );
    /* Convert the left ear temporary speaker Double Complex Split objects,
    from odd-even format to normal ordering, and copy to normal array. */
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left], 2, nOver2);
    /* Convert the right ear temporary speaker Double Complex Split
objects,
    from odd-even format to normal ordering, and copy to normal array. */
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right], 2, nOver2);
    // Add the overlap buffers to the output signals.
    for(int a = 0; a < overlapSize; a++)
    {
        dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][a] +=
dOverLap[middle][clockwise045][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][a] +=
dOverLap[middle][clockwise135][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][a] +=
dOverLap[middle][clockwise225][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][a] +=
dOverLap[middle][clockwise315][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][a] +=
dOverLap[middle][clockwise045][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][a] +=
dOverLap[middle][clockwise135][right][a];
```

```
        dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][a] +=
dOverLap[middle][clockwise225][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][a] +=
dOverLap[middle][clockwise315][right][a];
    }
    // Update overlap buffers.
    for(int a = 512; a < 1023; a++)
    {
        dOverLap[middle][clockwise045][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][a];
        dOverLap[middle][clockwise135][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][a];
        dOverLap[middle][clockwise225][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][a];
        dOverLap[middle][clockwise315][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][a];
        dOverLap[middle][clockwise045][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][a];
        dOverLap[middle][clockwise135][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][a];
        dOverLap[middle][clockwise225][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][a];
        dOverLap[middle][clockwise315][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][a];
    }
    // Derive left and right ear signals.
    for(int a = 0; a < 512; a++)
    {
        /* Add the left ear signals, divide them by the ammount of signals
        and multiply the total by the headphone amplitude. */
        fPtrAudioData[left][a] = (float) ((
            dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][a] ) /
4) * fHeadphoneAmplitude;
        /* Add the right ear signals, divide them by the ammount of signals
        and multiply the total by the headphone amplitude. */
        fPtrAudioData[right][a] = (float) ((
            dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][a] )
/ 4) * fHeadphoneAmplitude;
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::Process2D8ToBinaural
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::Process2D8ToBinaural(float* fPtrAudioData[])
{
    // Zero pad the remaining half of the temporary speaker buffers.
    for(int a = 512; a < 1024; a++)
    {
        dPtrVirtualSpeakerTempBuffer[middle][clockwise000][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise090][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise180][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise270][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise000][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][a] = 0.0;
```

```
            dPtrVirtualSpeakerTempBuffer[middle][clockwise090][right][a] = 0.0;
            dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][a] = 0.0;
            dPtrVirtualSpeakerTempBuffer[middle][clockwise180][right][a] = 0.0;
            dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][a] = 0.0;
            dPtrVirtualSpeakerTempBuffer[middle][clockwise270][right][a] = 0.0;
            dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][a] = 0.0;
        }
        /* Copy the temporary speaker buffers for the left ear,
        into the Double Complex Split objects and convert to odd-even format.*/
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise000][left], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise000][left], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise090][left], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise090][left], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise180][left], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise180][left], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise270][left], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise270][left], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], 1, nOver2);
        /* Copy the temporary speaker buffers for the right ear,
        into the Double Complex Split objects and convert to odd-even format.*/
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise000][right], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise000][right], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise045][right], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise090][right], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise090][right], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise135][right], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise180][right], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise180][right], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise225][right], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise270][right], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise270][right], 1, nOver2);
        ctozD((DOUBLE_COMPLEX *)
    dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right], 2,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise315][right], 1, nOver2);
        /* Convert left ear Double Split Complex objects to the frequency
    domain,
        storing them in the same Double Split Complex object. */
        fft_zripD(fft1024Setup,
    &virtualSpeakerDoubleComplexSplit[middle][clockwise000][left], stride,
    log2n, kFFTDirection_Forward);
```

```
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise090][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise180][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise270][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], stride,
log2n, kFFTDirection_Forward);
    /* Convert right ear Double Split Complex objects to the frequency
domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise000][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise090][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise180][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise270][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right], stride,
log2n, kFFTDirection_Forward);
    // Multiply the left ear feeds with the corresponding HRTFs.
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise000][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise000][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise000][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise090][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise090][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise090][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], stride,
nOver2, 1);
```

```
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise180][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise180][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise180][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise270][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise270][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise270][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left],
stride, &earHRTFDoubleComplexSplit[middle][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], stride,
nOver2, 1);
    // Multiply the right ear feeds with the corresponding HRTFs.
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise000][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise000][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise000][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise090][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise270][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise090][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise180][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise180][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise180][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise270][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise090][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise270][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right],
stride, &earHRTFDoubleComplexSplit[middle][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right], stride,
nOver2, 1);
    /* Convert left ear Double Split Complex objects to the time domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise000][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise090][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise180][left], stride,
```

```
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise270][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], stride,
log2n, kFFTDirection_Inverse);
    /* Convert right ear Double Split Complex objects to the time domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise000][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise090][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise180][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise270][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right], stride,
log2n, kFFTDirection_Inverse);
    // Scale the left ear signals stored in Double Split Complex objects.
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise000][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise000][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise000][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise000][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise045][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise045][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise045][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise045][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise090][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise090][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise090][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise090][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise135][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise135][left].realp, 1,
nOver2 );
```

```
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise135][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise135][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise180][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise180][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise180][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise180][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise225][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise225][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise225][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise225][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise270][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise270][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise270][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise270][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise315][left].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise315][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise315][left].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise315][left].imagp, 1,
nOver2 );
    // Scale the right ear signals stored in Double Split Complex objects.
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise000][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise000][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise000][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise000][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise045][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise045][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise045][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise045][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise090][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise090][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise090][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise090][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise135][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise135][right].realp, 1,
nOver2 );
```

```
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise135][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise135][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise180][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise180][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise180][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise180][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise225][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise225][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise225][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise225][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise270][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise270][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise270][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise270][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise315][right].re
alp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise315][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[middle][clockwise315][right].im
agp, 1, &scale,
virtualSpeakerDoubleComplexSplit[middle][clockwise315][right].imagp, 1,
nOver2 );
    /* Convert the left ear temporary speaker Double Complex Split objects,
    from odd-even format to normal ordering, and copy to normal array. */
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise000][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise000][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise045][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise090][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise090][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise135][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise180][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise180][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise225][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise270][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise270][left], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise315][left], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left], 2, nOver2);
    /* Convert the right ear temporary speaker Double Complex Split
objects,
    from odd-even format to normal ordering, and copy to normal array. */
```

```
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise000][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise000][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise045][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise090][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise090][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise135][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise180][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise180][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise225][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise270][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise270][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[middle][clockwise315][right],
1, (DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right], 2, nOver2);
    // Add the overlap buffers to the output signals.
    for(int a = 0; a < overlapSize; a++)
    {
        dPtrVirtualSpeakerTempBuffer[middle][clockwise000][left][a] +=
dOverLap[middle][clockwise000][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][a] +=
dOverLap[middle][clockwise045][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise090][left][a] +=
dOverLap[middle][clockwise090][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][a] +=
dOverLap[middle][clockwise135][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise180][left][a] +=
dOverLap[middle][clockwise180][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][a] +=
dOverLap[middle][clockwise225][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise270][left][a] +=
dOverLap[middle][clockwise270][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][a] +=
dOverLap[middle][clockwise315][left][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise000][right][a] +=
dOverLap[middle][clockwise000][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][a] +=
dOverLap[middle][clockwise045][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise090][right][a] +=
dOverLap[middle][clockwise090][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][a] +=
dOverLap[middle][clockwise135][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise180][right][a] +=
dOverLap[middle][clockwise180][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][a] +=
dOverLap[middle][clockwise225][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise270][right][a] +=
dOverLap[middle][clockwise270][right][a];
        dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][a] +=
dOverLap[middle][clockwise315][right][a];
    }
    // Update overlap buffers.
    for(int a = 512; a < 1023; a++)
    {
        dOverLap[middle][clockwise000][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise000][left][a];
```

```
        dOverLap[middle][clockwise045][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][a];
        dOverLap[middle][clockwise090][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise090][left][a];
        dOverLap[middle][clockwise135][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][a];
        dOverLap[middle][clockwise180][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise180][left][a];
        dOverLap[middle][clockwise225][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][a];
        dOverLap[middle][clockwise270][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise270][left][a];
        dOverLap[middle][clockwise315][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][a];
        dOverLap[middle][clockwise000][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise000][right][a];
        dOverLap[middle][clockwise045][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][a];
        dOverLap[middle][clockwise090][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise090][right][a];
        dOverLap[middle][clockwise135][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][a];
        dOverLap[middle][clockwise180][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise180][right][a];
        dOverLap[middle][clockwise225][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][a];
        dOverLap[middle][clockwise270][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise270][right][a];
        dOverLap[middle][clockwise315][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][a];
    }
    for(int a = 0; a < 512; a++)
    {
        /* Add the left ear signals, divide them by the ammount of signals
        and multiply the total by the headphone amplitude. */
        fPtrAudioData[left][a] = (float) ((
            dPtrVirtualSpeakerTempBuffer[middle][clockwise000][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise045][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise090][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise135][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise180][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise225][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise270][left][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise315][left][a] ) /
8) * fHeadphoneAmplitude;
        /* Add the right ear signals, divide them by the ammount of signals
        and multiply the total by the headphone amplitude. */
        fPtrAudioData[right][a] = (float) ((
            dPtrVirtualSpeakerTempBuffer[middle][clockwise000][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise045][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise090][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise135][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise180][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise225][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise270][right][a] +
            dPtrVirtualSpeakerTempBuffer[middle][clockwise315][right][a] )
/ 8) * fHeadphoneAmplitude;
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::Process3D8ToBinaural
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::Process3D8ToBinaural(float* fPtrAudioData[])
{
```

```
    // Zero pad the remaining half of the temporary speaker buffers.
    for(int a = 512; a < 1024; a++)
    {
        dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][a] = 0.0;
    }
    /* Copy the temporary speaker buffers for the left ear,
    into the Double Complex Split objects and convert to odd-even format.*/
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], 1, nOver2);
    /* Copy the temporary speaker buffers for the right ear,
    into the Double Complex Split objects and convert to odd-even format.*/
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right], 2,
```

```
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], 1, nOver2);
    /* Convert left ear Double Split Complex objects to the frequency
domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], stride,
log2n, kFFTDirection_Forward);
    /* Convert right ear Double Split Complex objects to the frequency
domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], stride,
log2n, kFFTDirection_Forward);
    // Multiply the left ear feeds with the corresponding HRTFs.
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], stride,
```

```
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], stride,
nOver2, 1);
    // Multiply the right ear feeds with the corresponding HRTFs.
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], stride,
nOver2, 1);
    /* Convert left ear Double Split Complex objects to the time domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
```

```
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], stride,
log2n, kFFTDirection_Inverse);
    /* Convert right ear Double Split Complex objects to the time domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], stride,
log2n, kFFTDirection_Inverse);
    // Scale the left ear signals stored in Double Split Complex objects.
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise045][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise045][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise045][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise045][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise135][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise135][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise135][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise135][left].imagp, 1,
```

```
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise225][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise225][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise225][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise225][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise315][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise315][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise315][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise315][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise045][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise045][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise045][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise045][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise135][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise135][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise135][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise135][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise225][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise225][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise225][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise225][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise315][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise315][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise315][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise315][left].imagp, 1,
nOver2 );
    // Scale the right ear signals stored in Double Split Complex objects.
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise045][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise045][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise045][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise045][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise135][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise135][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise135][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise135][right].imagp, 1,
```

```
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise225][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise225][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise225][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise225][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise315][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise315][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise315][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise315][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise045][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise045][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise045][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise045][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise135][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise135][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise135][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise135][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise225][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise225][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise225][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise225][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise315][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise315][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise315][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise315][right].imagp, 1,
nOver2 );
    /* Convert the left ear temporary speaker Double Complex Split objects,
    from odd-even format to normal ordering, and copy to normal array. */
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left],
```

```
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left],
2, nOver2);
    /* Convert the right ear temporary speaker Double Complex Split
objects,
    from odd-even format to normal ordering, and copy to normal array. */
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right], 2, nOver2);
    // Add the overlap buffers to the output signals.
    for(int a = 0; a < overlapSize; a++)
    {
        dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][a] +=
dOverLap[upper][clockwise045][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][a] +=
dOverLap[upper][clockwise135][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][a] +=
dOverLap[upper][clockwise225][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][a] +=
dOverLap[upper][clockwise315][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][a] +=
dOverLap[lower][clockwise045][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][a] +=
dOverLap[lower][clockwise135][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][a] +=
dOverLap[lower][clockwise225][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][a] +=
dOverLap[lower][clockwise315][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][a] +=
dOverLap[upper][clockwise045][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][a] +=
dOverLap[upper][clockwise135][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][a] +=
dOverLap[upper][clockwise225][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][a] +=
dOverLap[upper][clockwise315][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][a] +=
```

```
dOverLap[lower][clockwise045][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][a] +=
dOverLap[lower][clockwise135][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][a] +=
dOverLap[lower][clockwise225][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][a] +=
dOverLap[lower][clockwise315][right][a];
    }
    // Update overlap buffers.
    for(int a = 512; a < 1023; a++)
    {
        dOverLap[upper][clockwise045][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][a];
        dOverLap[upper][clockwise135][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][a];
        dOverLap[upper][clockwise225][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][a];
        dOverLap[upper][clockwise315][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][a];
        dOverLap[lower][clockwise045][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][a];
        dOverLap[lower][clockwise135][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][a];
        dOverLap[lower][clockwise225][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][a];
        dOverLap[lower][clockwise315][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][a];
        dOverLap[upper][clockwise045][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][a];
        dOverLap[upper][clockwise135][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][a];
        dOverLap[upper][clockwise225][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][a];
        dOverLap[upper][clockwise315][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][a];
        dOverLap[lower][clockwise045][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][a];
        dOverLap[lower][clockwise135][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][a];
        dOverLap[lower][clockwise225][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][a];
        dOverLap[lower][clockwise315][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][a];
    }
    // Derive left and right ear signals.
    for(int a = 0; a < 512; a++)
    {
        /* Add the left ear signals, divide them by the ammount of signals
        and multiply the total by the headphone amplitude. */
        fPtrAudioData[left][a] = (float) ((
            dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][a] ) /
8) * fHeadphoneAmplitude;
        /* Add the right ear signals, divide them by the ammount of signals
        and multiply the total by the headphone amplitude. */
        fPtrAudioData[right][a] = (float) ((
            dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][a] +
```

```
            dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][a] ) /
8) * fHeadphoneAmplitude;
    }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  BBinaural::Process3D16ToBinaural
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void BBinaural::Process3D16ToBinaural(float* fPtrAudioData[])
{
    // Zero pad the remaining half of the temporary speaker buffers.
    for(int a = 512; a < 1024; a++)
    {
        dPtrVirtualSpeakerTempBuffer[upper][clockwise000][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise090][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise180][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise270][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise000][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise090][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise180][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise270][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise000][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise090][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise180][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise270][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise000][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise090][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise180][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise270][right][a] = 0.0;
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][a] = 0.0;
    }
    /* Copy the temporary speaker buffers for the left ear,
    into the Double Complex Split objects and convert to odd-even format.*/
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise000][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise000][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise090][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise090][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
```

```
dPtrVirtualSpeakerTempBuffer[upper][clockwise180][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise180][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise270][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise270][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise000][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise000][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise090][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise090][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise180][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise180][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise270][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise270][left], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], 1, nOver2);
    /* Copy the temporary speaker buffers for the right ear,
    into the Double Complex Split objects and convert to odd-even format.*/
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise000][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise000][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise090][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise090][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise180][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise180][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise270][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise270][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right], 2,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise000][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise000][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
```

```
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise090][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise090][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise180][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise180][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise270][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise270][right], 1, nOver2);
    ctozD((DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right], 2,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], 1, nOver2);
    /* Convert left ear Double Split Complex objects to the frequency
domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise000][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise090][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise180][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise270][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise000][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise090][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise180][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], stride,
log2n, kFFTDirection_Forward);
```

```
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise270][left], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], stride,
log2n, kFFTDirection_Forward);
    /* Convert right ear Double Split Complex objects to the frequency
domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise000][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise090][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise180][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise270][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise000][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise090][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise180][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise270][right], stride,
log2n, kFFTDirection_Forward);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], stride,
log2n, kFFTDirection_Forward);
    // Multiply the left ear feeds with the corresponding HRTFs.
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise000][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise000][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise000][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], stride,
```

```
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise090][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise090][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise090][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise180][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise180][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise180][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise270][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise270][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise270][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left],
stride, &earHRTFDoubleComplexSplit[upper][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise000][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise000][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise000][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise090][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise090][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise090][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise180][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise180][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise180][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise270][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise270][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise270][left], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left],
stride, &earHRTFDoubleComplexSplit[lower][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], stride,
nOver2, 1);
    // Multiply the right ear feeds with the corresponding HRTFs.
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise000][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise000][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise000][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], stride,
```

```
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise090][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise270][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise090][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise180][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise180][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise180][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise270][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise090][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise270][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right],
stride, &earHRTFDoubleComplexSplit[upper][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise000][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise000][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise000][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise315][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise090][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise270][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise090][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise225][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise180][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise180][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise180][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise135][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise270][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise090][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise270][right], stride,
nOver2, 1);
    zvmulD(&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right],
stride, &earHRTFDoubleComplexSplit[lower][clockwise045][earType], stride,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], stride,
nOver2, 1);
    /* Convert left ear Double Split Complex objects to the time domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise000][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], stride,
log2n, kFFTDirection_Inverse);
```

```
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise090][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise180][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise270][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise000][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise090][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise180][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise270][left], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], stride,
log2n, kFFTDirection_Inverse);
    /* Convert right ear Double Split Complex objects to the time domain,
    storing them in the same Double Split Complex object. */
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise000][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise090][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise180][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise270][right], stride,
log2n, kFFTDirection_Inverse);
```

```
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise000][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise090][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise180][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise270][right], stride,
log2n, kFFTDirection_Inverse);
    fft_zripD(fft1024Setup,
&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], stride,
log2n, kFFTDirection_Inverse);
    // Scale the left ear signals stored in Double Split Complex objects.
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise000][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise000][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise000][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise000][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise045][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise045][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise045][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise045][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise090][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise090][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise090][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise090][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise135][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise135][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise135][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise135][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise180][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise180][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise180][left].imag
```

```
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise180][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise225][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise225][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise225][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise225][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise270][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise270][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise270][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise270][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise315][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise315][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise315][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise315][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise000][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise000][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise000][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise000][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise045][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise045][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise045][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise045][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise090][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise090][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise090][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise090][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise135][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise135][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise135][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise135][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise180][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise180][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise180][left].imag
p, 1, &scale,
```

```
virtualSpeakerDoubleComplexSplit[lower][clockwise180][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise225][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise225][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise225][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise225][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise270][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise270][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise270][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise270][left].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise315][left].real
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise315][left].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise315][left].imag
p, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise315][left].imagp, 1,
nOver2 );
    // Scale the right ear signals stored in Double Split Complex objects.
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise000][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise000][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise000][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise000][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise045][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise045][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise045][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise045][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise090][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise090][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise090][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise090][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise135][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise135][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise135][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise135][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise180][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise180][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise180][right].ima
gp, 1, &scale,
```

```
virtualSpeakerDoubleComplexSplit[upper][clockwise180][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise225][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise225][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise225][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise225][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise270][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise270][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise270][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise270][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise315][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise315][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[upper][clockwise315][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[upper][clockwise315][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise000][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise000][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise000][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise000][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise045][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise045][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise045][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise045][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise090][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise090][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise090][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise090][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise135][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise135][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise135][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise135][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise180][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise180][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise180][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise180][right].imagp, 1,
```

```
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise225][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise225][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise225][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise225][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise270][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise270][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise270][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise270][right].imagp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise315][right].rea
lp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise315][right].realp, 1,
nOver2 );
    vsmulD(virtualSpeakerDoubleComplexSplit[lower][clockwise315][right].ima
gp, 1, &scale,
virtualSpeakerDoubleComplexSplit[lower][clockwise315][right].imagp, 1,
nOver2 );
    /* Convert the left ear temporary speaker Double Complex Split objects,
    from odd-even format to normal ordering, and copy to normal array. */
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise000][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise000][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise045][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise090][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise090][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise135][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise180][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise180][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise225][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise270][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise270][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise315][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise000][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise000][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise045][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise090][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise090][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise135][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise180][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise180][left],
```

```
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise225][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise270][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise270][left],
2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise315][left], 1,
(DOUBLE_COMPLEX *) dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left],
2, nOver2);
    /* Convert the right ear temporary speaker Double Complex Split
objects,
    from odd-even format to normal ordering, and copy to normal array. */
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise000][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise000][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise045][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise090][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise090][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise135][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise180][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise180][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise225][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise270][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise270][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[upper][clockwise315][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise000][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise000][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise045][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise090][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise090][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise135][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise180][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise180][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise225][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise270][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise270][right], 2, nOver2);
    ztocD(&virtualSpeakerDoubleComplexSplit[lower][clockwise315][right], 1,
(DOUBLE_COMPLEX *)
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right], 2, nOver2);
    // Add the overlap buffers to the output signals.
    for(int a = 0; a < overlapSize; a++)
    {
        dPtrVirtualSpeakerTempBuffer[upper][clockwise000][left][a] +=
```

```
dOverLap[upper][clockwise000][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][a] +=
dOverLap[upper][clockwise045][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise090][left][a] +=
dOverLap[upper][clockwise090][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][a] +=
dOverLap[upper][clockwise135][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise180][left][a] +=
dOverLap[upper][clockwise180][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][a] +=
dOverLap[upper][clockwise225][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise270][left][a] +=
dOverLap[upper][clockwise270][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][a] +=
dOverLap[upper][clockwise315][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise000][left][a] +=
dOverLap[lower][clockwise000][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][a] +=
dOverLap[lower][clockwise045][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise090][left][a] +=
dOverLap[lower][clockwise090][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][a] +=
dOverLap[lower][clockwise135][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise180][left][a] +=
dOverLap[lower][clockwise180][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][a] +=
dOverLap[lower][clockwise225][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise270][left][a] +=
dOverLap[lower][clockwise270][left][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][a] +=
dOverLap[lower][clockwise315][left][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise000][right][a] +=
dOverLap[upper][clockwise000][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][a] +=
dOverLap[upper][clockwise045][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise090][right][a] +=
dOverLap[upper][clockwise090][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][a] +=
dOverLap[upper][clockwise135][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise180][right][a] +=
dOverLap[upper][clockwise180][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][a] +=
dOverLap[upper][clockwise225][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise270][right][a] +=
dOverLap[upper][clockwise270][right][a];
        dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][a] +=
dOverLap[upper][clockwise315][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise000][right][a] +=
dOverLap[lower][clockwise000][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][a] +=
dOverLap[lower][clockwise045][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise090][right][a] +=
dOverLap[lower][clockwise090][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][a] +=
dOverLap[lower][clockwise135][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise180][right][a] +=
dOverLap[lower][clockwise180][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][a] +=
dOverLap[lower][clockwise225][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise270][right][a] +=
dOverLap[lower][clockwise270][right][a];
        dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][a] +=
dOverLap[lower][clockwise315][right][a];
    }
    // Update overlap buffers.
```

```
    for(int a = 512; a < 1023; a++)
    {
        dOverLap[upper][clockwise000][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise000][left][a];
        dOverLap[upper][clockwise045][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][a];
        dOverLap[upper][clockwise090][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise090][left][a];
        dOverLap[upper][clockwise135][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][a];
        dOverLap[upper][clockwise180][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise180][left][a];
        dOverLap[upper][clockwise225][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][a];
        dOverLap[upper][clockwise270][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise270][left][a];
        dOverLap[upper][clockwise315][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][a];
        dOverLap[lower][clockwise000][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise000][left][a];
        dOverLap[lower][clockwise045][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][a];
        dOverLap[lower][clockwise090][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise090][left][a];
        dOverLap[lower][clockwise135][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][a];
        dOverLap[lower][clockwise180][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise180][left][a];
        dOverLap[lower][clockwise225][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][a];
        dOverLap[lower][clockwise270][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise270][left][a];
        dOverLap[lower][clockwise315][left][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][a];
        dOverLap[upper][clockwise000][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise000][right][a];
        dOverLap[upper][clockwise045][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][a];
        dOverLap[upper][clockwise090][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise090][right][a];
        dOverLap[upper][clockwise135][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][a];
        dOverLap[upper][clockwise180][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise180][right][a];
        dOverLap[upper][clockwise225][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][a];
        dOverLap[upper][clockwise270][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise270][right][a];
        dOverLap[upper][clockwise315][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][a];
        dOverLap[lower][clockwise000][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise000][right][a];
        dOverLap[lower][clockwise045][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][a];
        dOverLap[lower][clockwise090][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise090][right][a];
        dOverLap[lower][clockwise135][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][a];
        dOverLap[lower][clockwise180][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise180][right][a];
        dOverLap[lower][clockwise225][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][a];
        dOverLap[lower][clockwise270][right][a - 512] =
dPtrVirtualSpeakerTempBuffer[lower][clockwise270][right][a];
        dOverLap[lower][clockwise315][right][a - 512] =
```

```
dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][a];
    }
    for(int a = 0; a < 512; a++)
    {
        /* Add the left ear signals, divide them by the ammount of signals
        and multiply the total by the headphone amplitude. */
        fPtrAudioData[left][a] = (float) ((
            dPtrVirtualSpeakerTempBuffer[upper][clockwise000][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise045][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise090][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise135][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise180][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise225][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise270][left][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise315][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise000][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise045][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise090][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise135][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise180][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise225][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise270][left][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise315][left][a] ) /
8) * fHeadphoneAmplitude;
        /* Add the right ear signals, divide them by the ammount of signals
        and multiply the total by the headphone amplitude. */
        fPtrAudioData[right][a] = (float) ((
            dPtrVirtualSpeakerTempBuffer[upper][clockwise000][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise045][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise090][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise135][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise180][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise225][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise270][right][a] +
            dPtrVirtualSpeakerTempBuffer[upper][clockwise315][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise000][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise045][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise090][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise135][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise180][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise225][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise270][right][a] +
            dPtrVirtualSpeakerTempBuffer[lower][clockwise315][right][a] ) /
8) * fHeadphoneAmplitude;
    }
}
```