TI Project

Documentation
IEM, Graz 2017

Tim Raspel

Vector Base Amplitude Panning

VST Plugin Implementation

Supervisor
Franz Zotter

September 27, 2017

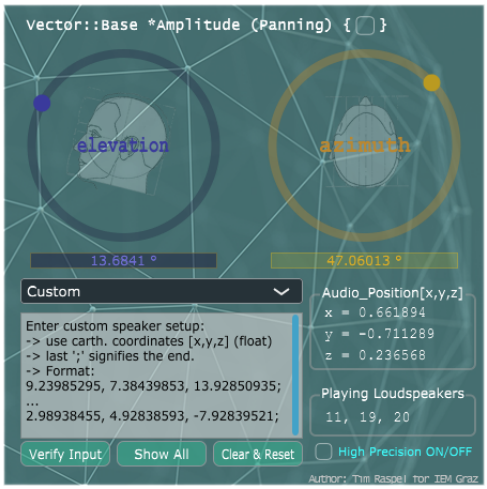
Abstract

This project paper is about implementing VBAP as a VST Plugin. VBAP (vector base amplitude panning) is a method of positioning virtual sound sources on playback setups with an arbitrary number of loudspeakers. The VST Plugin will have its own custom GUI. A primary goal of the project is to establish a properly functioning plugin that allows seamless panning and provides the option of presets, as well as support for custom loudspeaker coordinates. Another explicit goal of the project is to provide a documentation on how one can set up a build environment to develop new plugins or refine existing open-source plugins.

Contents

1 Introduction

This *Toningenieur-Projekt* documentation is principally about marrying two subjects: VBAP & VST. VBAP (vector base amplidute panning) is a method developed by Pulkki [1] for positioning (panning) an audio source on a system of loudspeakers, by extending the concept of panning (commonly used in stereo or surround setups) to arbitrary setups, even three-dimensional with a multitude of loudspeakers. VST (virtual studio technology) is the implementation framework of choice. VST plugins are a standardised, established possibility of adding this new functionality (VBAP) to existing DAW[2] host programs.



1.1 Resulting VST plugin, availability, code

- Plugin name: **Vector Base Amplitude Panning** (VBAP)
- Project period: October 2016 (announcement) — September 2017 (finish)
- Effective time frame: 5 months (research, implementation, documentation)
- Terms of use: **The VBAP plugin is free** (in compliance with licensing)
- Licensing: X11 (see Copyright Notice.txt in project's source and Appendix C!)
- IDE: *Microsoft Visual Studio Community 2017*
- Resources:
  - JUCE/Projucer v5 Personal (www.juce.com) JUCE 5 GPL license
  - Eigen C++ template library (eigen.tuxfamily.org) MPL2 license
  - Convex Hull Algorithm (www.newtonapples.net) GPLv3 license
- Source code: placeholder.net
- Plugin dll: placeholder.net
- Tested DAWs: Reaper 5.40 x64 (Win7)
- Tested Configs: IEM Lehrstudio, IEM Cube

Note: All the VST2 host-plugin communication is handled by JUCE code – JUCE does not use the VST2 headers. Compiling as VST3 requires the Steinberg SDK (www.steinberg.net/en/company/developers.html).

1.2 Structure of the documentation

The following content will acquaint the reader with what VST is, starting from a bird's eye view, eventually descending into details and facts about how to use VST and how to create a plugin (recapturing my own approach on this project) in sections 2 and 3. Useful information for the interested reader can also be found in a condensed Q&A in the appendix (8).
Further on, the VBAP method is described briefly (4). After this theory review, the details of the actual implementation are discussed in section 5. In section 6, precise instructions are given about how the user needs to input custom loudspeaker position coordinates, followed by some information about testing the plugin (7). The documentation ends with a conclusion statement (8). The three appendices are *A: How to make a VST plugin Q&A*, which is a condensed collection of useful information and links about how to get started with VST, *B: plugin manual* is a 1-page poster format for a quick and easy reference, and *C: Copyright and Licenses* is self-explanatory.

2 VST: Virtual Studio Technology

VST is an *interface specification* plus SDK[1] introduced by Steinberg GmbH in 1996. The idea behind it is simple and practical: Steinberg develops a DAW[2] software named Cubase. While constantly updating the program, there are some components that do not change, but are supposed to offer their functionality in all updates yet to come. Such a component is called **plug-in**, and the *specification* defines how the plugging-in works and also which parts of the main program it may influence. By separating plugins from the main program, their content can be shared among different versions of the main program or even other (similar) programs offering this plugin interfacing capability. Additionally, developers outside the progam developing company may create plugins of this kind to independently enhance the capabilities of the program. Section 2.3 presents the details of this concept.

Because of this specification, Cubase quickly gained popularity. Subsequently, VST became an established industry standard for audio plugins.

[1] software developement kit
[2] digital audio workstation. Sequencer program for audio recording & editing

2.1 Applications

VST plugins may be categorised into areas defining their type of functionality:

Effect plugins

Plugins of this type manipulate audio input (provided by the main "host" program), using DSP[3] techniques. Common examples are **delay**, **chorus**, **compressor**, **reverb**, **auto-tune** etc. Quite often, effect plugins try to mimic legacy analogue effect boxes. They show a graphical interface containing knobs, sliders and other sorts of controls for the user to edit the sound in a traditional analogue fashion, while the insides of the plugin simulate the behaviour of the analogue circuit.

Instrument plugins

These plugins are virtual instruments. They emulate the sound of real instruments, generate synthesized sounds or allow working with samples. They are usually controlled by MIDI (substituting the input signal) to generate audio output. To tell them apart from other plugins, they are marked with an "i" in their name: **VSTi**.

Analyser plugins

I came up with this third category, because there are plugins that neither manipulate nor generate audio. They let the audio pass through, analysing it and showing information content. These plugins are useful for advanced metering, frequency diagram illustration etc. Common examples are FFT-Analyser, loudness metering, goniometer and pitch detector (e.g., for guitar tuning).

MIDI plugins

Plugins of this fourth category transfer MIDI data only. MIDI messages are processed and routed to VSTi or hardware devices.

[3] digital signal processing

2.2 Overview: History

1996	VST version 1.0 is released along with Cubase 3.02, containing the first inherent VST plugins (reverb, chorus, stereo echo and auto panner).
1999	VST version 2.0 is released, introducing MIDI support and VSTi. An SDK is included for third-party plugin developement.
2008	VST version 3.0 is released. VSTi now allow audio input, MIDI capabilities are improved.
2011	VST is updated to 3.5, it contains articulation controls and note expression for polyphonic MIDI arrangements.
2013	Steinberg discontinues the maintainance of VST 2.0 SDK

As of today (2017), plugins of VST 2.x standard are still most common, because DAW developers other than Steinberg are reluctant to implement full VST3 support.

2.3 Coding details

Now we are going to plunge into the depths of VST programming. The two different perspectives of host and plugin will be presented briefly in this section. Extensive information can be found in sources [2], [3] and [4].

What exactly is a VST plugin?

"In the widest possible sense a VST plugin is an audio process" [4, p.3]. It cannot operate on its own, it needs a host application taking care of the framework like audio buffering etc.

In terms of source code, a VST plugin is (or better starts off as) a **C++ class**[4] called **AudioEffect** (VST 1.0) or **AudioEffectX** (VST 2.0). The source code is platform-independent, however, having this class obviously is not enough...

First of all, it is an abstract base class. That means that in order to create a new plugin, you need to *derive* your own custom plugin class from that class to meet the VST specification. In this base class, a bunch of methods are declared as *virtual*. The host knows the names of these methods (functions), so it can call them while interacting with the plugin. By deriving your own plugin class, you keep those names, but you may override what the method actually does.

There are methods that send plugin information to the host, like the name, the number of parameters or if it is a VSTi. There are other methods that define the function of the plugin, most importantly *get/setParameter*, *process* and *processReplacing*. The latter two are called by the host all the time, manipulating the audio input stream in real-time. Parameter changes can be accounted for within these functions as well. It might look like this:

[4]user defined type or data structure – declared with keyword *class* – that has data and functions as its members

```
// Apply gain to a stereo input block of audio data and overwrite the output
void MyPlugin::processReplacing(float **inputs, float **outputs, long
    sampleFrames)
{
        float *in1 = inputs[0];
        float *in2 = inputs[1];
        float *out1 = outputs[0];
        float *out2 = outputs[1];
        float gain = getParameter(0);
        while(--sampleFrames >= 0)
        {
                (*out1++) = (*in1++) * gain;     // replacing
                (*out2++) = (*in2++) * gain;
        }
}
```

When you have finished overriding (defining) the required functions, your class may be instantiated (out of the abstract mysterious fog, a concrete object is formed). This is achieved by the host. The host calls the *constructor* of your class when the user loads the plugin into an audio track. When the user hits the play button, the host starts buffering the content on the audio track and calls processReplacing, which, in the example above, changes the volume of the audio.

So far so good. BUT: we are still talking about plain *code* here. The host can't do anything with C++ code lines. It needs to be compiled to something the host can make use of... This *something* is platform-dependent.

- On **Windows** platforms, a VST plugin is a multi-threaded DLL (dynamic link library). In contrast to a static library[5], a DLL can be linked to *during runtime*. The concept of sharing code this way has been conceived in the early days of Windows OS programming, because memory space was precious, and including the same functionality again and again in each compiled system component had to be avoided. Compiling it once, loading it once into dynamic memory and having all other functions access it on runtime still is the core of today's Windows operating systems. There are several downsides to this concept (DLL Hell), but further details will be omitted here.
- On **Mac OS** platforms, a VST plugin is a raw code resource (resource fork) or a bundle. A bundle is a file directory with a defined structure and file extension (.VST), allowing related files to be grouped together as a conceptually single item. Concepts similar to DLL are applying here.

The compiled plugin file is placed in a directory known by the host program. The host maintains a list of all plugins in this directory. It also manages multiple instantiations of the same plugin, as well as saving and loading plugin states when closing and re-opening the audio project.

[5] which is included and linked to at compile time

If you are interested in code details for VST programming (raw basics), consult sources [4] and [2]. For more information about what is going on inside the host program, see [3].

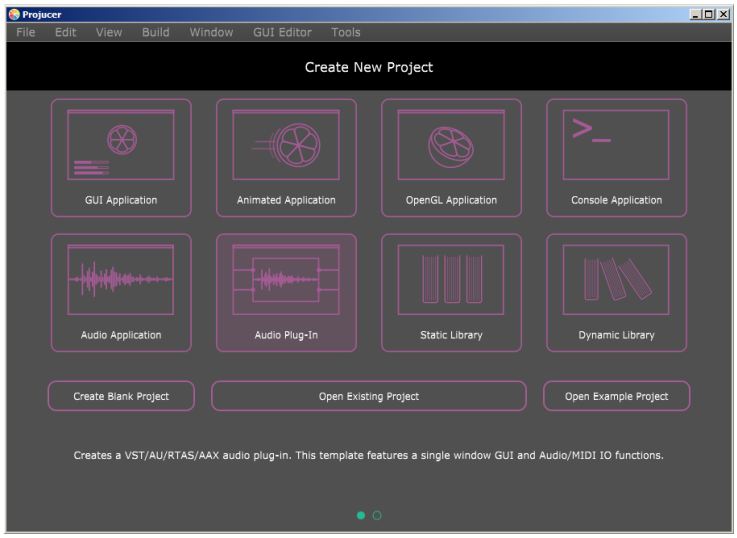
3 VST implementation via JUCE framework

What is JUCE[6]?

> *"JUCE is a partially open-source, cross-platform C++ application framework, used for the development of desktop and mobile applications. JUCE is used in particular for its GUI and plug-in libraries. The aim of JUCE is to allow software to be written such that the same source code will compile and run identically on Windows, Mac OS X and Linux platforms. It supports various development environments and compilers, such as GCC, Xcode, Visual Studio and Code::Blocks."*

All that is needed to start programming a VST plugin is the JUCE library[7] and having installed one of the compatible IDEs listed in the quote above. I developed my plugin on Windows, using Visual Studio 2017.

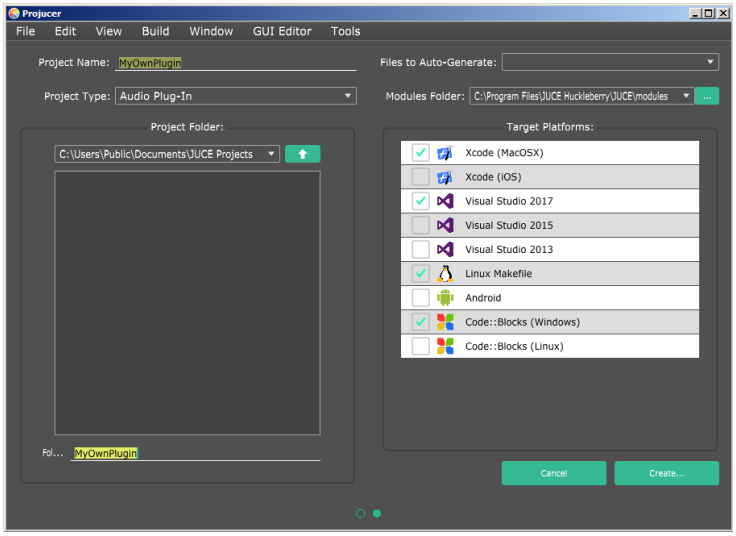
After unpacking JUCE, start the application **Projucer**. It offers you to create a new project:



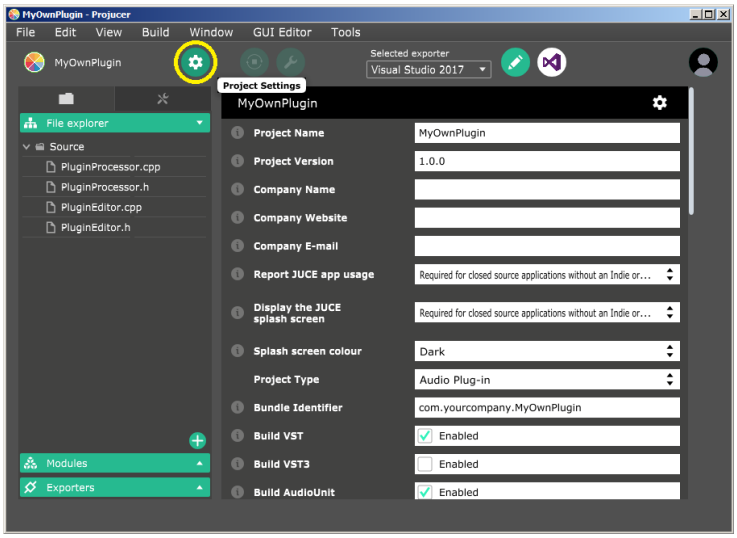
This helper guides you through the process of creating your own plugin template, which is already compilable, which you can use as a starting point when adding your own functionalities. Choosing **Audio Plug-In** will take you to the next windows, where you can select the name for your project (the chosen name will be the name of your plugin

[6] see https://en.wikipedia.org/wiki/JUCE [5. September 2017, 00:21]
[7] available at https://www.juce.com/get-juce/download

class as well, so you want to call it something more meaningful than "MyOwnPlugin", but let's continue with this name for demonstration), moreover, choose the IDE in which you want to compile the plugin later. Projucer will create and manage all relevant files to be opened in the IDE (e.g., in Visual Studio, it will create and update the *solution file*). If you change your mind about which IDE you want to use, you can still choose other platforms later.



Clicking **Create** will complete the process and show you four files that have been created for you. But before taking a look into those, go through the settings first:



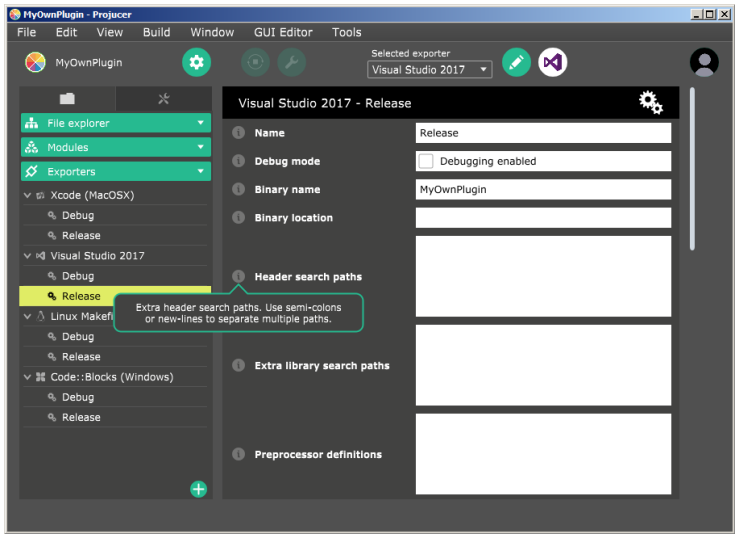
It is important that you enable what you want to build. VST and AU[8] are enabled

[8]Audio Unit, an Apple MacOS plugin format. Core Audio Utility Classes needed!

by default. RTAS[9] and AAX[10] require you to contact Avid in order to get the SDKs for these formats. The *Plugin Name* and *Plugin Manufacturer* entries will show in the DAW's plugin list. The plugin's channel configuration is something quite important: You need to take care about how many input and output channels your plugin accepts. One way to do so is to fix the configuration right here in the settings, e.g., put a $\{1, 1\}$ for 1 input and 1 output. Otherwise you need to take care of the *PreferredChannelConfigurations* section and the *isBusesLayoutSupported*-function in the PluginProcessor.cpp file:

```
#ifndef JucePlugin_PreferredChannelConfigurations
bool MyOwnPluginAudioProcessor::isBusesLayoutSupported (const BusesLayout&
    layouts)
{
  [...]
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    if (layouts.getMainOutputChannelSet() != AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != AudioChannelSet::stereo())
        return false;
  [...]
}
#endif
```

Next on the agenda: The exporters options. You need to add the paths to extra headers or libraries for all your exporters. Because usually, you do not want to copy entire frameworks, SDKs or similars directly into your projects main folder (that would become confusing quickly).



Example: You want to use the *Eigen C++ template library*. You download it and place it in a location where you generally keep your project-external elements. You tell Projucer this location. On compilation, it finds and uses those external files. That way,

[9] Real Time AudioSuite
[10] Avid Audio eXtension

you are not tempted to modify external code (because other projects that depend on that code might become corrupt if you do).

If you scroll down, the options *Optimization* and *Architecture* are well worth a look. You might want to add new configurations for different settings. Right-click on the IDE name and select *Add a new configuration*. By default, there are only two configurations: debug and release. You may want to add options for 32-bit and 64-bit architecture.

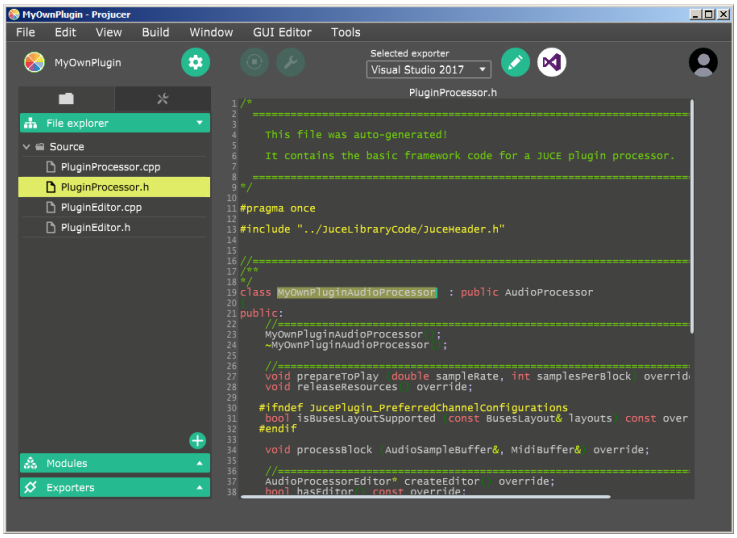
Before going into the code, here are some **very important** notes to remember at all times when working with Projucer and an IDE simultaneously:
Projucer and your code editor both are accessing **the same** code files, and are able to make changes to them. Making a change in Projucer (and saving) will trigger the IDE to reload all files that have changed. Making a change in the IDE will cause Projucer to reload as well (when returning to the Projucer window), but depending on the extent of the modification, a crash is possible. Please take this fact into account so nothing is lost during development.

Key rules in short (from [5] and complemented):

1. Beware of modifying your source files in Projucer and external editor simultaneously
2. Don't ever put code outside of the "editable regions" in managed files (designated by in-line comments), as they are overwritten as soon as Projucer reopens the file in question
3. Don't manage project settings in your development environment (always return to the Projucer and edit settings there!!)
4. JUCE has MANY useful classes. Before building your own XYZ, use the online class documentation to see if there is a solution already

Now we will take a look at the auto-generated code: Our very own custom (well not yet...) plugin class is defined in the header file *PluginProcessor.h*. The class is named after whatever you called your project when first creating it. It inherits from *AudioProcessor* (see after the colon), which is a base class to generally wrap all plugin types you might want to get in the end (like VST, AAX etc.). This makes the code pretty universal!

The first two declarations in the public section are the default constructor and destructor of the class (a basic requirement for classes in C++), followed by functions of the *AudioProcessor* base class, that are to be overridden. Most important for the audio processing will be the *processBlock* function. When compiling for VST format, this function's content (same is true for other function members) will be translated to the *process* and



processReplacing functions defined in the VST specification [4].

The two function declarations at the bottom are quite important. They take care of saving & restoring the plugin's state when the DAW project is saved/loaded:

```
void getStateInformation (MemoryBlock& destData) override;
void setStateInformation (const void* data, int sizeInBytes) override;
```

The implementations of all these functions can be found in *PluginProcessor.cpp*. In the auto-generated version, the absolute minimum is implemented. Although all the function bodies are there, they merely return a due value or don't do anything at all (in case of void-type functions). Exceptions: The constructor, the *acceptsMidi*, *producesMidi* and *isBusesLayoutSupported* functions involve some preprocessor defines (marked by #), which react to options you checked in the Projucer's settings earlier (e.g., the channel configuration).
The only function with a meaningful implementation is *processBlock*. It gets the number of input and output channels, and clears output channels if there are more than inputs. It also provides you with a *writePointer* to the data in the audio buffer, but then doesn't use it. This is quite comfortable though, because you could instantly start writing your DSP code into the following line, like so:

```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    float* channelData = buffer.getWritePointer (channel);
    // ..do something to the data...
    int blocksize = buffer.getNumSamples();
    for (int n = 0; n < blocksize; ++n)
    {
        channelData[n] *= 0.1f;      // a general -20 dB
    }
}
```

However, JUCE framework offers a cleaner way for computing gains (amongst many other useful things). This loop has the same effect as in the previous one:

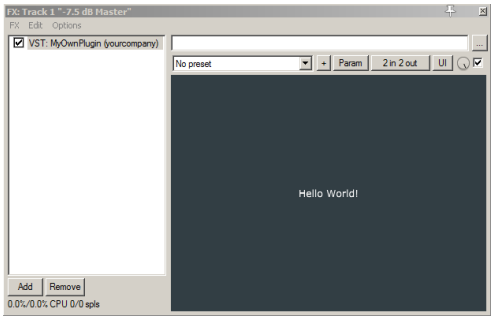
```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
        buffer.applyGain(channel, 0, buffer.getNumSamples(), 0.1f);
}
```

Here are some helpful facts about how the audio buffer works in JUCE, [8] quotes:

> The same buffer is used for both input **and** output.
> When processBlock is called, the buffer contains a number of channels which is at least as great as the maximum number of input and output channels that this filter is using. It will be filled with the filter's input data and should be replaced with the filter's output.
> So for example if your filter has a total of 2 input channels and 4 output channels, then the buffer will contain 4 channels, the first two being filled with the input data. Your filter should read these, do its processing, and replace the contents of all 4 channels with its output.
> Or if your filter has a total of 5 inputs and 2 outputs, the buffer will have 5 channels, all filled with data, and your filter should overwrite the first 2 of these with its output. But be VERY careful not to write anything to the last 3 channels, as these might be mapped to memory that the host assumes is read-only!
> If your plug-in has more than one input or output buses then the buffer passed to the processBlock methods will contain a bundle of all channels of each bus. Use AudiobusLayout::getBusBuffer to obtain an audio buffer for a particular bus. Note that if you have more outputs than inputs, then only those channels that correspond to an input channel are guaranteed to contain sensible data - e.g., in the case of 2 inputs and 4 outputs, the first two channels contain the input, but the last two channels may contain garbage, so you should be careful not to let this pass through without being overwritten or cleared.
> Also note that the buffer may have more channels than are strictly necessary, but you should only read/write from the ones that your filter is supposed to be using.
> The number of samples in these buffers is NOT guaranteed to be the same for every callback, and may be more or less than the estimated value given to prepareToPlay(). Your code must be able to cope with variable-sized blocks, or you're going to get clicks and crashes! Also note that some hosts will occasionally decide to pass a buffer containing zero samples, so make sure that your algorithm can deal with that!
> If the filter is receiving a midi input, then the midiMessages array will be filled with the midi messages for this block. Each message's timestamp will indicate the message's time, as a number of samples from the start of the block. Any messages left in the midi buffer when this method has finished are assumed to be the filter's midi output. This means that your filter should be careful to clear any incoming messages from the array if it doesn't want them to be passed-on.

Be very careful about what you do in this processBlock function callback - it's going to be called by the **audio thread**, so any kind of direct interaction with the UI (thread) is **absolutely out of the question**. If you change a parameter in here and need to tell your UI to update itself, the best way is probably to inherit from a ChangeBroadcaster, let the UI components register as listeners, and then call sendChangeMessage() inside the processBlock() method to send out an asynchronous message. You could also use the AsyncUpdater class in a similar way.

Now, let's turn to something different: The face of the plugin! The two files named *PluginEditor* define what will be seen, when the plugin is compiled and opened in a host program:



Screenshot of the template plugin's GUI, loaded in Reaper

Nothing fancy? Hello World! may not do much, but consider that there already *is* a GUI being created for you! All you need to do is start working on it – and Projucer offers a very handy tool to create GUI components.
Usually, as you click on a GUI cpp file, the Projucers GUI editor opens automatically. The auto-generated *PluginEditor.cpp* however is not compatible. In order to start creating your own GUI, do the following:

1. Either right-click on the source folder in the file explorer or click on the round plus button at the bottom and select **Add New GUI Component**.

2. In your source folder, select the *PluginEditor.cpp* file and overwrite it when asked to. This will replace both editor files and make them compatible with the GUI editor feature.

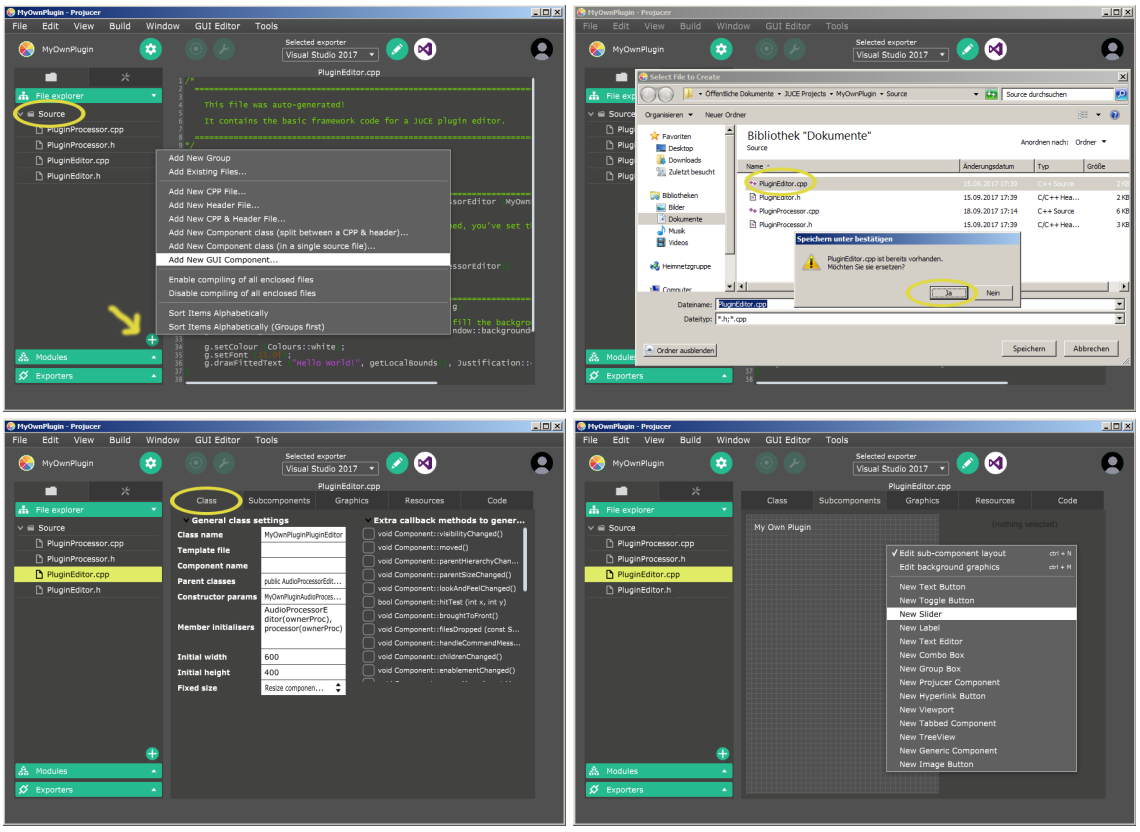
3. When clicking on *PluginEditor.cpp* in the file explorer again, a squared field will pop up, under the *Subcomponents* tab.

4. Select the **Class** tab and edit the following entries:

   - Class name: **MyOwnPlugin**AudioProcessorEditor (or **project name**)
   - Parent classes: public AudioProcessorEditor, public Timer
   - Constructor params: **MyOwnPlugin**AudioProcessor& ownerProc

- Member initialisers: AudioProcessorEditor(ownerProc), processor(ownerProc)

This will connect the overwritten GUI class to the PluginProcessor, because it expects this class name in the auto-generated *createEditor* function. The parent classes are the ones your new GUI will inherit from (AudioProcessorEditor is necessary, the timer may come in handy later). The constructor and initalisers are one mechanism to have a reference of the plugin processor available in the GUI (for sending messages or signaling parameter changes etc.). The pointer to the processor is passed in the *createEditor* function.



5. Set your GUI size and select whether it shall be fixed or resizable. If you choose the latter, you will need to take care of the resize programming!

6. Now you can go to the **Subcomponents** tab again, right-click on the squared area and add new elements.

7. The last tab **Code** shows you what your creations in **Subcomponents** tab look like in writing. If you want to add code manually, do it **in the marked sections only!** Everything you write outside these sections will be erased on re-selecting *PluginEditor.cpp* in the file explorer! The same is true for the *PluginEditor.h*.

8. In *PluginEditor.h*, edit the //[Headers] section to #include "PluginProcessor.h" (this is necessary for the processor reference ownerProc)

Now the plugin provides a proper basis to work with. Some hints on how to continue: Look at the JUCE Tutorials available online, start with **Tutorial: The AudioProcessorValueTreeState class**, which introduces an elegant way of defining and managing adjustable plugin parameters, connecting them to your GUI sliders via attachment and also easily loading and storing them. For further improving the interaction between GUI and audio processor, have a look at **Tutorial: Listeners and broadcasters**. Many more helpful tutorials including code is available online, featuring advanced GUI design, MIDI processing, etc.

4 VBAP: Vector Base Amplitude Panning

In this section, the VBAP method is covered, closely refering to the original source of Pulkki [1].

Audio reproduction began monophonically. One loudspeaker emanated a pointlike sound field. In the 1950s, (two-channel) stereophony became popular. Two loudspeakers emenated a sound field within the space between them (on a horizontal line). Sound field illusion was greatly enhanced hereby, because along this line, multiple sources (e.g. instruments) could be placed for a widened sound perception – while maintaining localisation.

Later approaches stick to horizontal sound fields mostly, but also include holophony and three-dimensional Ambisonics. The idea is to enhance sound reproduction to span not only a plain (like 4.0, 5.1 etc.), but all kinds of three-dimensional spaces. Existing methods stipulate fixed loudspeaker positioning, e.g., Ambisonics loudspeakers are strongly recommended to be placed orthogonally.

VBAP allows loudspeakers to be placed arbitrarily in 2D or 3D. The number of loudspeakers is customisable. The only requirements for the sound reproduction to work properly are:

→ The room is not too reverberant
→ Each loudspeaker is approximately equally far away from the listener

4.1 Basics: 2D Panning

The most common method of 2D amplitude panning is called *intensity panning*. It is an approximative method. Example: Two loudspeakers in a stereo setup play *coherent*[11] signals. The listener perceives a mono source, localised more to the left or right, depending

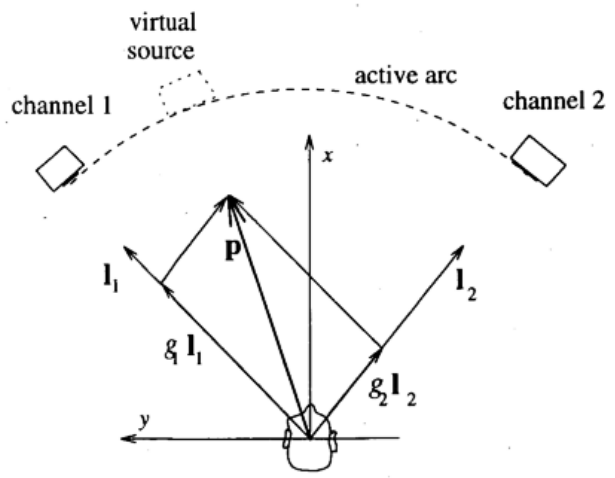
[11] same signal content, different amplitude (or fixed phase-shift)

on how high the loudspeakers signal amplitude is. Gain factors (g_1, g_2) are used to shift the sound source that way. It is then called a *virtual* or *phantom* source. When moving this source once from left to right, its perceived loudness should be about the same during the whole journey. To achieve this, signal power is used over raw signal amplitude:

$$g_1^2 + g_2^2 = C$$

Think of the parameter C (>0) as a general source volume control. Take a look at the image. C represents the length of vector $\mathbf{p}$, which indicates the position of the virtual source. The listener head is in the origin. The vectors $\mathbf{l_1}$ and $\mathbf{l_2}$ have unit-length[12] and point toward loudspeaker 1 and 2. Then, according to vector algebra, $\mathbf{p}$ is a linear combination of the loudspeaker vectors:

$$\mathbf{p} = g_1\mathbf{l_1} + g_2\mathbf{l_2}$$



Stereophonic configuration formulated with vectors [1]

The equation can be re-written in matrix form, which simply means that g_1 and g_2 are bundled up together in a vector called $\mathbf{g}$ and the loudspeaker vectors are packed into a matrix called $\mathbf{L}$:

$$\mathbf{p^T} = \mathbf{g} \cdot \mathbf{L}$$

$$\begin{bmatrix} p_x & p_y \end{bmatrix} = \begin{bmatrix} g_1 & g_2 \end{bmatrix} \cdot \begin{bmatrix} \begin{bmatrix} l_{1x} \\ l_{1y} \end{bmatrix} & \begin{bmatrix} l_{2x} \\ l_{2y} \end{bmatrix} \end{bmatrix}$$

Now, if you as the user want to control where the sound source is, you need to compute this equation "in reverse" to get the gain factors g_1 and g_2. In linear algebra, this is done

[12] length = 1

in shape of a matrix inverse (signified by $^{-1}$ $\rightarrow$ inversion is usually a quite troublesome topic, but problems are naturally preempted in this context[13]). If this inverse exists, the solution looks like this:

$$\mathbf{g} = \mathbf{p^T} \cdot \mathbf{L^{-1}}$$

In a final step, the gain factors need to be normalised to satisfy the first equation (about the consistent signal power), while C is the optional volume control parameter:

$$\mathbf{g^{scaled}} = \frac{\mathbf{g^{calculated}}}{\sqrt{g_1^2 + g_2^2}} \cdot \sqrt{C}$$

As soon as there are more than two loudspeakers (in 2D, e.g., a 5.1 setup), the basic method remains exactly the same, but there will be different **L**-matrices to choose from (in 5.1 there will be 5), depending on between which loudspeaker pair the virtual source is supposed to be. The procedure goes as follows according to [1]:

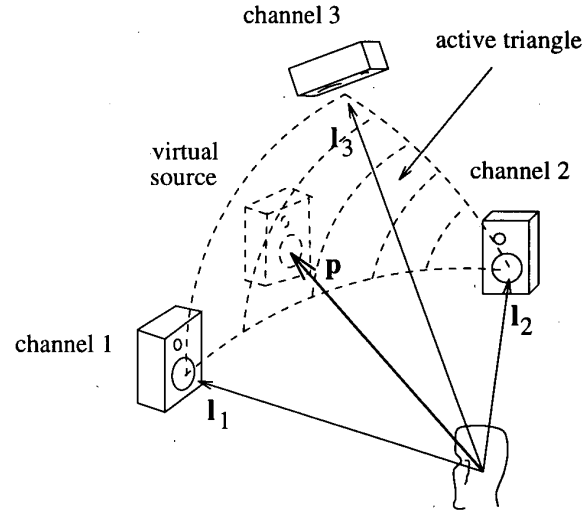
1. Define your direction vector **p**
2. Calculate the (unscaled) gain for *all* possible **L**-matrices
3. There will be exactly one gain pair with *positive values*[14]. This is the one pair corresponding to the two loudspeakers that are supposed to play. The selected area is called *active arc*.
4. As calculating the gain is already part of the previous step, all that is left to do now is the normalisation.
5. If you are moving the source, you need to keep a copy of the old gain, and cross-fade those to the newly calculated gains for a smooth transition.

4.2 Extension to 3D

Proceeding to three dimensions is an easy step. Because the equations of the 2D description hold for 3D as well. The only differences: All vectors have a third entry now, thus the **L**-matrix is 3x3. And the selected area (where the virtual source is located) is not an arc, but a surface segment cut out of a unit sphere (active triangle):

[13] because two loudspeakers being in the exact same location is physically impossible; two loudspeakers facing towards each other is a very undesirable setup; and meeting the equidistance restraint of VBAP, collinearity cannot occur either

[14] for implementation it is recommended to allow for slightly negative values (numeric stability)



Sample config: loudspeaker triangle for 3D panning [1]

Let's extend the 2D equations to 3D (same order as before):

$$g_1^2 + g_2^2 + g_3^2 = C$$

$$\mathbf{p} = g_1\mathbf{l_1} + g_2\mathbf{l_2} + g_3\mathbf{l_3}$$

This equation is even exactly the same, only the dimensions have changed:

$$\mathbf{p^T} = \mathbf{g} \cdot \mathbf{L}$$

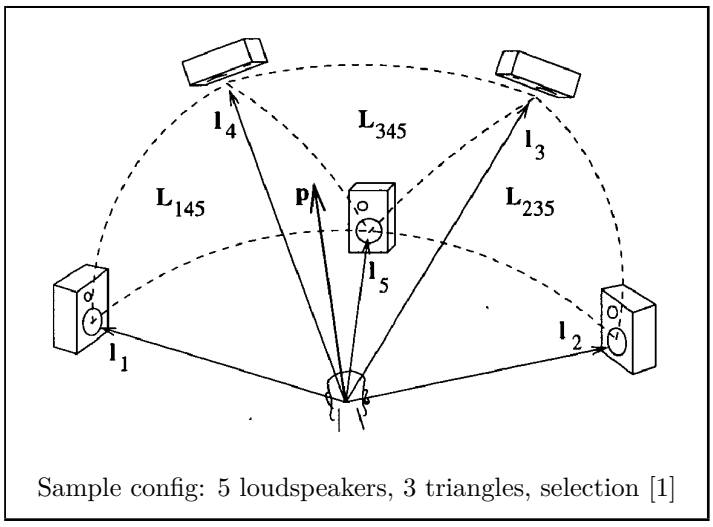
$$\begin{bmatrix} p_x & p_y & p_z \end{bmatrix} = \begin{bmatrix} g_1 & g_2 & g_3 \end{bmatrix} \cdot \begin{bmatrix} \begin{bmatrix} l_{1x} \\ l_{1y} \\ l_{1z} \end{bmatrix} & \begin{bmatrix} l_{2x} \\ l_{2y} \\ l_{2z} \end{bmatrix} & \begin{bmatrix} l_{3x} \\ l_{3y} \\ l_{3z} \end{bmatrix} \end{bmatrix}$$

Thus, the reversed equation doesn't change either:

$$\mathbf{g} = \mathbf{p^T} \cdot \mathbf{L^{-1}}$$

$$\mathbf{g^{scaled}} = \frac{\mathbf{g^{calculated}}}{\sqrt{g_1^2 + g_2^2 + g_3^2}} \cdot \sqrt{C}$$

Just as in 2D – going from stereo to 5.1 – in 3D there will probably be more than 3 loudspeakers (actually an absolute minimum of 4 is required to even form a three-dimensional object). So the *active triangle* amongst many possible triangles (**L**-matrices) must be selected. The procedure is the same as in 2D (compute all possible triplets and select the one with exclusively positive gain).



Sample config: 5 loudspeakers, 3 triangles, selection [1]

Some remarks: If some loudspeakers in the setup do not fulfil the equidistance assumption, they can be compensated for using another individual gain and time-adjustment (e.g., sample delay). The effect of varying distances impacts the sound reproduction the more the smaller the triangle is. This is given either if lots of loudspeakers are used in the setup, or if the setup is not evenly spread with some triangles being really small.

VBAP has three important properties, which are a nice treat, because the method automatically takes care of those special cases:

1. If $\mathbf{p}$ points directly to a loudspeaker (this means $\mathbf{p} = \mathbf{l_i}$), only this particular loudspeaker (i) plays the signal.

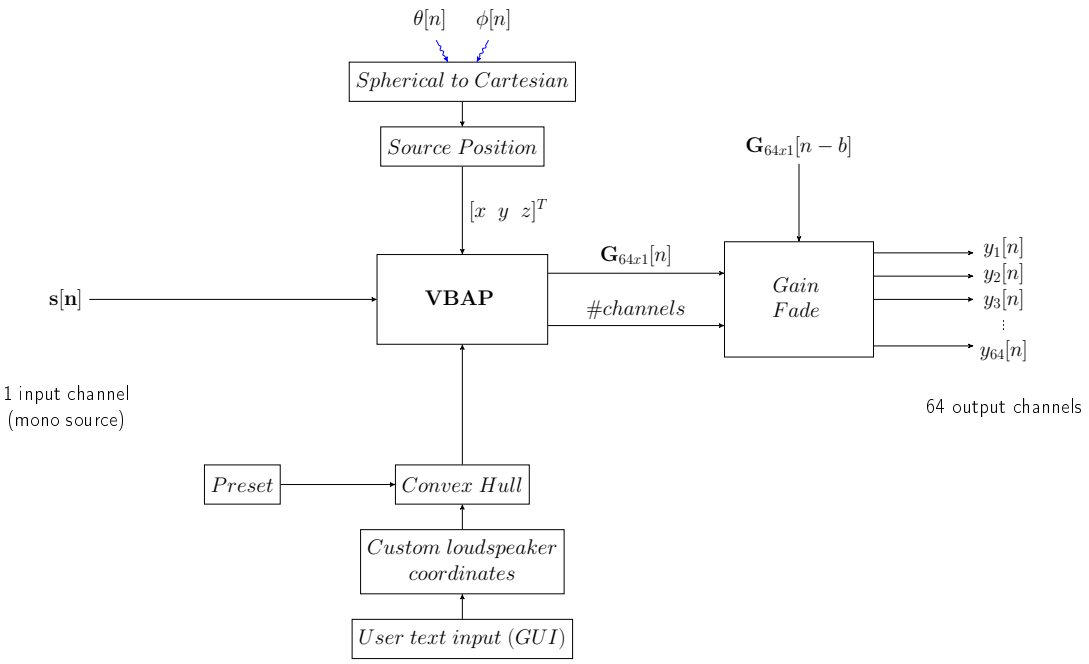
2. If $\mathbf{p}$ points directly to the line connecting two loudspeakers of the triangle, the signal is played only by those two particular loudspeakers, automatically applying the laws of 2D.

3. If $\mathbf{p}$ points directly to the centre of a triangle, then g_1, g_2 and g_3 are equal.

These properties ensure maximal localisation sharpness with the (any) given loudspeaker configuration.

5 VBAP: Implementation as VST plugin

This section will resume at the point where section 3 ended. Specific steps taken to give the JUCE plugin template the function of VBAP as described in the previous section 4 will be presented.



Abstract signal flow graph of the VBAP plugin

The signal flow graph gives an overview of what the implementation does internally. It might be helpful for a better understanding, especially of the advanced sections 5.6 and 5.7.

The input signal **s[n]** is a mono signal (the first channel of the buffer is assumed to be the input). The user controls the VBAP processing by choosing the phantom source position via elevation θ and azimuth angle ϕ during runtime. The user also selects the loudspeaker configuration, either as a preset or a custom one, using the GUI's textbox input (see section 6 for instructions). VBAP computes gains for all channels. 64 channels are the maximum number of channels. The first '*#channels*' channels are multiplied by the gain provided by VBAP (using linear interpolation between gains of the current **G**[n] and the **G**[n-b] of the previously processed block), the remaining channels are cleared of any garbage content (they are silent).

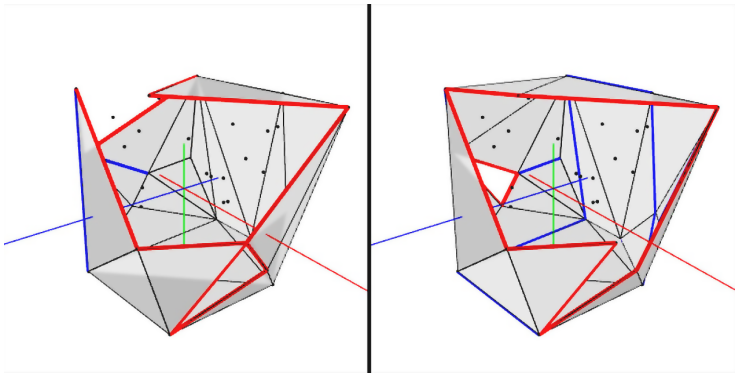
5.1 #include additional code

Natively, C++ offers many nice features. Obviously, not everything is just handed to you in a "predigested ready-to-use way". As for the VBAP method (4), vector and matrix representation of certain values is asked for. C++ has the std::vector template. The

std::vector is a container, storing elements sequentially, taking care of its (dynamic) size on its own. The elements can be any kind of data structure, so with std::vector, you can manage a bunch of classes, or structs, or simple types like int or float.

Mathematical vector and matrix arithmetic is not intended for the std::vector. But since a matrix inversion and several special multiplications are *needed* for the core algorithm, why not look for public license C++ code that already exists and that does what is required? The most versatile possibility is the **Eigen template library**[15]. It provides vector and matrix types of all shapes and sizes, including all basic linear algebra operations you could wish for (normalize, dot, cross, sum, inverse...). There is even a translation table[16] from MATLAB to Eigen (if you are prototyping in MATLAB).

And we need something to manage the loudspeaker configuration of VBAP, a 3D construct similar to the one shown in the most recent image. This kind of construct is called **convex hull**.



Examplary 3D hull algorithm.
Image source: i.vimeocdn.com/video/539916739_1280x960.jpg

Out of a 3D point cloud, a convex hull algorithm finds the outmost points and connects them to form triangular surfaces, wrapping the inner points. The hull consists of information about which points form those surfaces, and/or additional information about the surface (face normal vector) or the connecting border lines. Having a convex hull makes managing the VBAP's "active triangle" much easier, especially for arbitrary loudspeaker configurations. The public C++ code used in this plugin is the **Newton Apple Wrapper**, found in [6].

Since Eigen and Newton Apple Wrapper are not part of the C++ standard libraries, the compiler does not know yet where to find them. Add the path to their directories in the Projucers Exporter setting as described in section 3. Also add the implementation file (.cpp) as an existing file to your project (in the Projucer's *File explorer*).

[15] eigen.tuxfamily.org/index.php?title=Main_Page
[16] eigen.tuxfamily.org/dox/AsciiQuickReference.txt

```
// so far we add to include:
#include <vector>
#include <../Eigen/Dense>
#include "NewtonApple_hull3D.h"

// other standard headers for certain functions:
#include <algorithm>
#include <cmath>
#include <numeric>

// two classes written on my own:
#include "LockedString.h"
#include "VirtualSpeakerClass.h"
```

5.2 Own classes

During plugin development, the issue of threading came up. The JUCE plugin template is destined for multi-threading. One thread is assigned to compute the core DSP functions of the plugin only, while the other thread is concerned with everything happening in the GUI. In this plugin's design it is necessary that core and GUI exchange data. This opens up the possibilitiy of **data race / race conditions**. It is enough for a race condition to occur that a single thread attempts to change the shared data while the rest of the threads can either read or change it. This is undefined behaviour! The content of that shared data is haphazard, depending on which thread accesses first – this is decided at runtime by the thread scheduler, which means the access order could change all the time. In the best case, your data becomes garbage (bad enough!), in the worst case applications using the plugin will crash. However there are methods to avoid this issue.

One of them is using **atomic** variables. This special type of variables ensures that the situation is well-defined if multiple threads race at it. The result of an operation (stored in an atomic variable) however may still cause problems. For example in if-statements depending on that variables value: Another thread changes the value, while the important thread will not execute the if-statement's code, because *in this instant* the condition is not met. But: For basic signaling and error code exchanges between threads, atomics work just fine in case of this specific plugin (see 5.3).

The class **LockedString** implements another method of avoiding race conditions. Since atomics are available only for primitive types int and bool and exchanging whole strings of text between GUI and core is desired, a threadsafe string data type is asked for. The string stored in the LockedString class as a private member can only be read or written using special getter / setter functions called tryWriteLockedString or tryReadLockedString. They impose a **Mutex**[17] on the process of reading or writing the string variable by locking the access for as long as it takes to finish this process. As long as

[17] mutual exclusion

one thread is occupied with that string, no other thread will be allowed to operate on it simultaneously. The JUCE framework provides *CriticalSection* class and *ScopedLock* to achieve this mutex (code example[18]).

The **virtualspeaker** class is much simpler. It represents loudspeakers that are not part of the real configuration, but additional (virtual) ones, which are being created occasionally to improve the implemented VBAP method. It keeps track of how many virtual loudspeakers there are (by ID) and stores the neighbouring real loudspeakers (also by ID). More details will be revealed in the hull computation section 5.6. The two class functions implement the saving and restoring of instances contents (for the plugins get- & setStateInformation functions).

5.3 Parameters

The plugin's main parameters are the **azimuth** and the **elevation** angle, which together define the position of the phantom source. These two are intended to be adjusted (also automated) by the user. There are two slider wheels on the GUI for this purpose. As mentioned in the final paragraph of section 3, the most elegant implementation would be using the **AudioProcessorValueTreeState**. A tutorial is available online on the JUCE website. Here is the VBAP plugin's implementation:

To use an AudioProcessorValueTreeState object, you can store one in your processor class (in PluginProcessor.h): Add AudioProcessorValueTreeState parameters; in the private section. This way, "parameters" will have the same lifetime as the processor.

The AudioProcessorValueTreeState CONSTRUCTOR requires a reference to the AudioProcessor subclass that it will be attached to, and a pointer to an UndoManager object: parameters (*this, nullptr). Add this statement to the initialisation list of the plugin's constructor: VbapAudioProcessor::VbapAudioProcessor() : parameters (*this, nullptr) – as the template's constructor already starts the initialisation list with the BusesProperties and AudioChannelSet, instead of ":" the parameter statement is appended *after* it, separated by "," followed by the constructor function opening bracket "{" (*I explicitly mention this here, because this syntax may lead to huge problems if you choose to fix the channel configuration in the project settings! Then the added parameter statement causes a million compiler errors, because the #ifndef block is ignored and there is no ":" to start the initialisation list!*

[18] www.juce.com/doc/group__juce__core-threads#gacedaa6fb1373c96d2d15e7a617a5cec8

This is how the parameters are set:

```
//==============  Managing plugins parameters  ==============
//
// AudioProcessorValueTree::createAndAddParameter( "ID",
//                                                 "Name",
//                                                 "Label suffix",
//                                                 Range,
//                                                 Default value,
//                                                 Conversion func,
//                                                 Conversion func  )
//
        parameters.createAndAddParameter(
                "polar_angle",
                "PolarAngle",
                String(TRANS("\xb0")),      // degree symbol (U+00B0)
                NormalisableRange<float>(-90.0f, 90.0f),
                0.0f,
                nullptr,
                nullptr);

        parameters.createAndAddParameter(
                "azimuth_angle",
                "AzimuthAngle",
                String(TRANS("\xb0")),
                NormalisableRange<float>(-180.0f, 180.0f),
                0.0f,
                nullptr,
                nullptr);
```

- **ID** should be a unique identifier for this parameter. Think of this as being a variable name; it can contain alphanumeric characters and underscores, but no spaces.
- **Name** is the name that will be displayed on the screen.
- **Label suffix** allows you to specify a suffix (for example "dB" for gain in decibels or "Hz" for frequency parameters).
- **Range** of values that will be represented by the parameter is specified using a NormalisableRange<float> object to set the minimum and maximum values for the parameter. This may also specify a skew-factor to make the transition between minimum and maximum non-linear (see Tutorial: The AudioParameter classes).
- The final two nullptr arguments in this call createAndAddParameter() function are optional conversion functions to convert between the value and the text that you want to represent that value (and vice versa). Specifying a nullptr value as either or both of these arguments uses the default conversion functions (which simply convert a floating point value to a string and a string back to a floating point value respectively).

The final step after configuring the parameters is to initialise the ValueTree within the AudioProcessorValueTreeState. This is stored in the state public member within the AudioProcessorValueTreeState object. This ValueTree object needs an identifier to be valid (which is used as part of the conversion to XML that we will use to save plugin states, see setStateInformation in PluginProcessor.cpp). So directly beneath the previous code (within the plugin's constructor), add

```
        parameters.state = ValueTree( Identifier("VBAPPARMETERDATA") );
```

"VBAPPARMETERDATA" is the name of the XML object storing the state information. At this point the processor is ready to use the AudioProcessorValueTreeState object. However, the two parameters are not yet connected to the GUI sliders. As it is directly related to the ValueStateTree, the parameter-to-slider attachment is shown here: Simply add this thing called SliderAttachment to the GUI component class:

```
// Declarations in PluginEditor.h (or in this specific case TimsGUI.h)
[...]
private:
    //[UserVariables]   -- You can add your own custom ...
    AudioProcessorValueTreeState& valueTreeState;

    ScopedPointer<SliderAttachment> polarAttachment;
    ScopedPointer<SliderAttachment> azimuthAttachment;
    //[/UserVariables]


// Instantiations in PluginEditor.cpp constructor
[...]
// auto-generated slider objects using the GUI Editor feature...
// "polarslider" and "azimuthslider" need to exist prior to attachment
addAndMakeVisible (polarslider = new Slider ("PolarSlider"));
addAndMakeVisible (azimuthslider = new Slider ("AzimuthSlider"));
[...]

//[Constructor] You can add your own custom stuff here..
polarAttachment = new SliderAttachment (valueTreeState, "polar_angle",
                                        *polarslider);
azimuthAttachment = new SliderAttachment (valueTreeState, "azimuth_angle",
                                          *azimuthslider);
//[/Constructor]
```

Now the plugin's parameters are linked to the sliders. The specific values will be needed in core processing. They may be accessed within the processBlock core function via:

```
float current_polar = *parameters.getRawParameterValue("polar_angle");
float current_azimuth = *parameters.getRawParameterValue("azimuth_angle");
```

Then there are all these other plugin-internal variables, that are not visible from the outside. Here is a short list and explanation what they are for:

Eigen::Vector3f srco;	”old” source position
LockedString MessageContainer;	stores the GUI’s TextBox content
LockedString CoordInputContainer;	stores GUI input text for verification
LockedString ActiveLspContainer;	stores the active Lsp IDs for GUI display
Atomic<int> polflag_a;	vital flag to get Azi/Ele right!
Atomic<int> coordflag;	errorflag for user input processing
Atomic<int> precisionflag;	saves the state of the GUI presision button
Atomic<int> selectedslotflag;	saves the state of selected combo box slot
Atomic<int> saveflag;	1: Reaper project was saved, 0: Custom ComboBox may be set once more → this flag must not be saved in XML!
Atomic<int> go_signal;	checked in VbapAudioProcessor::timerCallback() calls refreshLspSetup(), suspends processing!
std::vector<Eigen::Vector3f> originalLspCoord;	stores input coordinates as a first instance (after GUI read in)
std::vector<R3> current_pts;	definite point coordinates + IDs
std::vector<R3> current_pts_ext;	definite point coordinates + IDs (extended hull)
std::vector<R3> current_pts_ext_pre;	copy needed, as NAW resorts current_pts_ext...
std::vector<Tri> current_tris;	holds faces of the hull + IDs + normal vectors
std::vector<Tri> current_tris_ext;	holds faces of the extended hull + IDs + normals
std::vector<Eigen::Vector2i> lspOrder;	(x) NAW ID and (y) original ID
std::vector<Eigen::Vector2i> lspOrder_ext;	(x) extern NAW ID and (y) extern ID
std::vector<float> dim_factor;	special factor for each virtual Lsp (gain weight)
std::vector<int> virtual_IDs;	self-explaining
std::vector<virtualspeaker> virtual_Lsp;	holds the ID of each virtual Lsp and most importantly its neighbours!
Eigen::Matrix<float, 64, 1> G_final;	fixed 64x1 float vector, for ”old” gain values in order to enable gain fading

5.4 Function members

A fair share of functions has been added to the plugin class. They can be categorised into groups depending on what they do:

functions around the hull computation	
computeEverythingHull();	everything concerning the hull computation
isThisVirtual();	quick check if some Lsp is virtual (by ID)
comb(int N, int K);	returns all N-choose-K combinations listed as vector
setCurrentPts();	sets the loudspeaker coordinate configuration
setCurrentPtsExt();	sets the configuration, virtuals being added
setCurrentTris();	computes the (real) convex hull
setCurrentTrisExt();	computes the (extended, virtual) convex hull
getCorrectChannelOrder();	necessary because NAW resorts the Lsps...
getCorrectChannelOrderExt();	again when recalculated including virtuals

verifyGUIinput();	checks GUI input coordinates, gives error messages
refreshLspSetup();	wrapper function, calls verifyGUIinput(), then comuteEverythingHull()
fullResetToIEM();	resets the whole plugin to IEM, in case something went wrong
functions around processBlock	
getSourcePositionUpdate();	calculates the phantom source position (based on current slider state)
calculateHullGain();	implements VBAP method (see 4)
polarRotationSign();	vital function to get Azimuth/Elevation right!
linspace();	MATLAB function "linspace" ported to C++
conversions of all sorts	
convertCarthesianToSpherical();	
convertSphericalToCarthesian();	
R3toVector3f();	R3: struct format used in Newton Apple Wrapper
Vector3ftoR3();	
MultiVector2iToStdVectorInt();	
juceToEigenVector3();	JUCE Vector3D<float> to Eigen::Vector3f
message string generating functions...	(**continued in this column**)
generatePtsInfoMsg();	generatePtsExtInfoMsg();
generateTrisInfoMsg();	generateTrisExtInfoMsg();
generateOrderInfoMsg();	generateOrderExtInfoMsg();
generateLInfoMsg(int i);	generateGfinalInfoMsg();
generateDimfactorInfoMsg();	generateVirtualLspInfoMsg();
showMatrixValuesLin();	generateCustomCoordinatesMsg();
generateCoordinatesMsg();	generateVirtualCoordinatesMsg();
showSelectedMessages();	a wrapper function for all the above
XML for get/setStateInformation	
createXmlState();	turns whole plugin (variables) into an XML element
readXmlState();	restores the entire state of the plugin

The implementation of these functions may be accessed in the project's code provided on the IEM server (GIT).

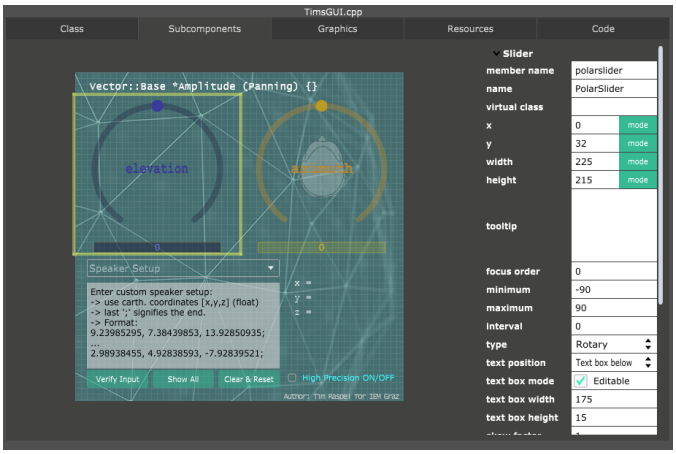
5.5 The GUI

The plugin obtains its own GUI. I start off after overwriting the default "Hello World" GUI of the plugin template (see 3). Components are added using the Projucer's GUI Editor feature. Notice that there are certain elements missing compared to the GUI of the loaded plugin. This is due to the fact that the Projucer can only show elements that are created and maintained in the *Subcomponents* tab. However, some desired behaviour could not be achieved this way. So if modifications need to be made, there are several ways of doing that:

1. If you simply want to change the styles and appearances of the components, you can create your own custom look using the JUCE LookAndFeel class[19].

2. If you want standard components to behave differently, you can make your own class, inheriting from existing components.

3. If it is just a tiny detail that cannot be done in the GUI Editor (and not worth creating an entire class for it), create the component anyway and move the auto-generated code (*Code* tab) to the user-defined areas like //[Constructor_pre] or //[UserPreSize]. It won't show in the *Subcomponents* tab anymore, but it will be visible in the plugin after compilation.



The third method is used for the head image inside the elevation circle and the GroupComponents wrapping the x-y-z position information and the active loudspeaker display on the right bottom side. The head image (just like the background and the azimuth head) is created in the *Graphics* tab, selecting *New Image* and setting the *image source*. In order to assign an image file, add the file in the *Resources* tab, it will then show in the *image source* choices. Beware: All images will be cached in the plugin's final file (dll/bundle). Take quantity and image resolution into account!

The elevation head is supposed to rotate along with the slider knob's movement in a specific way. JUCE has the AffineTransform class which provides a function to rotate images. It is important to set the centre position, around which the image will rotate:

```
headimage->setTransform(AffineTransform::identity.rotated(0.0f,110.0f,130.0f
    ));
headimage->setCentrePosition(110, 130);
```

The GroupComponent elements are moved to the //[Constructor_pre] section simply because they would block the label components displaying x, y, z and the active loudspeaker IDs. This way, they are placed on a layer *below* the displayed information.

[19]Tutorial: www.juce.com/doc/tutorial_look_and_feel_customisation

What is the user allowed to control?

The two main controls are the two sliders adjusting the phantom source position (see 5.3). The GUI intends to visualise this: The slider knobs actually mimic the source position, and the heads inside each slider's cirlce represent the listener. Now, as the knob is moved, the played sound will be perceived at the location indicated by the relative head-knob positioning. Intuitive control is granted to the user (without the need for expensive 3D modelling). The sliders are both fully rotational: seamless arbitrary circular movements become possible this way. Some tweaks have to be made though:

The **azimuth angle** is mathmatically defined in the range $\in [0, 2\pi]$, so a full circle is the appropriate way of representing this range. However, there are two problems: 1. It would be much more intuitive to define the zero-angle right in front of the head's eyes. So a range of $\in [-\pi, +\pi]$ is chosen instead. 2. Coordinate systems are commonly given right-handed orientation: The angle is defined to *increase* on counter-clockwise rotation and *decrease* on clockwise rotation. Translated: Moving the azimuth knob to the left causes the audio to move right and vice versa. The quickest way of fixing this is to invert the sign of the y coordinate when converting the elevation and azimuth to Cartesian coordinates.

The **elevation angle** is mathmatically defined in the range $\in [0, \pi]$, which is sufficient for unique representation in spherical coordinates. In this case as well, the zero-angle shall lie in front of the head's eyes: new range $\in [-\pi/2, +\pi/2]$. Unfortunately, this range spans only half a circle – and the slider components do not allow for jumps – if the desired value went beyond the sliders end position, it would just stop right there. The solution I came up with is stretching the range to a full circle. This way, when going beyond $+\pi$, the value will automatically jump to $-\pi$, ensuring the required smoothness in phantom source positioning. A new issue is caused by this: Moving the elevation knob to a visual 90° results in an actual value of 45° because of the stretching. So if the head is fixed (like in the azimuth case), the relative position will be wrong. The scaling is the reason why the head image is made to rotate. A complicated custom function manages this rotation. It also takes into account that when the azimuth knob is in a position *behind* the azimuth head, it should also be displayed as being behind the elevation head! The same is true vice versa: If the elevation angle makes a jump (surpassing the range), the azimuth slider will jump as well (by 180°).

The **high precision button** activates the phantom source movement feature described in section 5.8. It applies only if the source movement is *very* fast – for instance steep automation jumps. In this case, the standard gain fade is replaced by orthodromic distance interpolation: intermediate source points (an estimated number of 1 to 8) are inserted between the old and the new source position based on how far they are apart – using the shortest path on a spherical surface. This procedure will be computationally more expensive while being more precise in terms of location, however it will perform this extra computation *exclusively* if the source movement within one block of buffered audio

is sufficiently far. In usual applications, this extra feature won't be necessary, so it is recommended to leave it switched off.

The **toggle button** in the GUI's title (between the curled braces) allows the user to switch between VBAP and VBIP. As is known from previous explanations, the 'A' in VBAP stands for 'Amplitude' – because the gain computed by VBAP is used unmodified as described in section 4. The new 'I' stands for 'Intensity' and takes the square root of the VBAP's gain computation before continuing with the normalisation.

The operation of the **menu, textbox & buttons** deserves its own special place. Details will be explained there (6). A short overview: Selecting a menu preset will suspend audio processing and trigger convex hull computation and loudspeaker coordinate setup. When finished, the *Show All* button may display information about the current configuration in the textbox. Selecting the custom menu will display input format instructions in the textbox. The *Verify Input* button is available to be pressed now, it triggers configuration computations when clicked. The textbox will show messages over the process of the computation and reveal possible errors. If something goes wrong, the plugin will reset itself to the first preset. Adding presets is not possible, but the custom configuration is saved *for this particular effect instance* when the audio project is saved in the host program.

The GUI has (is) a **timer** object (introduced when overwriting the default PluginEditor files, see 3). Now its usefulness is examined: **timer** provides the function timerCallback, which is called at regular intervals while the plugin is running. The interval is set in the GUI constructor (in the user-defined section!) by startTimer(50); the value being the time interval in milliseconds. 50 has been chosen because it translates to 20 frames per seconds: the callback function exclusively updates the GUI display (head rotation, coordinates, active loudspeakers and textbox), so this is a reasonable trade-off between wasting computational resources and stuttering visuals.

Remarks: For each instance of sliders, buttons and menues (and other components created in the *Subcomponents* tab) there is a vital function that takes care of what happens if this specific component is activated or used or clicked etc.; and Projucer auto-generates code lines – they just need to be filled with something meaningful. The functions in question are:

- TimsGUI::buttonClicked()
- TimsGUI::comboBoxChanged()
- TimsGUI::sliderValueChanged()
- etc.

Some of the atomic flags introduced in section 5.3 are intended to be set by the GUI, so that the processor is informed of certain changes. The polflag_a is needed for correct

conversion from spherical to Cartesian coordinates involving the concept of the slider angles described earlier. The selectedslotflag saves the currently active menu entry (the plugin processor needs to know this for save & restore). The precisionflag indicates whether the high precision feature is active (info directly for the core processing). And most importantly the go_signal tells the core of the plugin to start re-computing the loudspeaker configuration (immediate asynchronous message exchange between plugin core and GUI! See section 5.2 to learn about the dangers surrounding data exchange between threads).

5.6 Algorithm: Hull computation

Now, the go_signal is given either if a menu preset is selected or if the user has entered his own coordinates and hits the *Verify Input* button. In any case, the following procedure is exactly the same: The core processing of the plugin is suspended (as operating on partially complete configuration data is quite dangerous) and the LockedString type instance called *CoordInputContainer* filled with the requested data (= coordinates of either the presets or the custom user input, both as a *JUCE String*).

The LockedString is the threadsafe method of transfering string data from the GUI thread to the core processing thread. The go_signal makes sure that the string data is processed only after the data write is complete. Making the plugin core realise that it needs to start the new configuration computation is actually the exact same method already applied in the GUI: a timer! The core plugin class also inherits from the JUCE Timer class, and its callback function checks whether the go_signal is set. If it is, the computation process is executed. The interval of this timer however is far more generously spaced, because re-computation is expected to happen very rarely (maybe once or twice when starting a new audio project). So excessive checking is to be avoided at all cost – because this checking will impede the audio thread. On the other hand, the user shouldn't need to wait for too long after having decided on a configuration. The trade-off value of 1 check per second seems reasonable.

As a first step, the refreshLspSetup() function is called from within the timer callback. It is a wrapper function for the entire configuration computation process. These inner functions are executed sequentially:

1. clear() — any and all variables involved in the convex hull computation are cleared for a fresh start

2. verifyGUIinput(...) — is executed, the CoordInputContainer.tryReadLockedString() being read and passed as the argument as to what is supposed to be verified. Within the function, the string argument is parsed and converted into an std::vector<Eigen::Vector3f>. Many things can go wrong as soon as arbitrary user input is involved. This function makes sure that no user input can crash the plugin. In the end, it returns a bool true, if everything went according to plan.

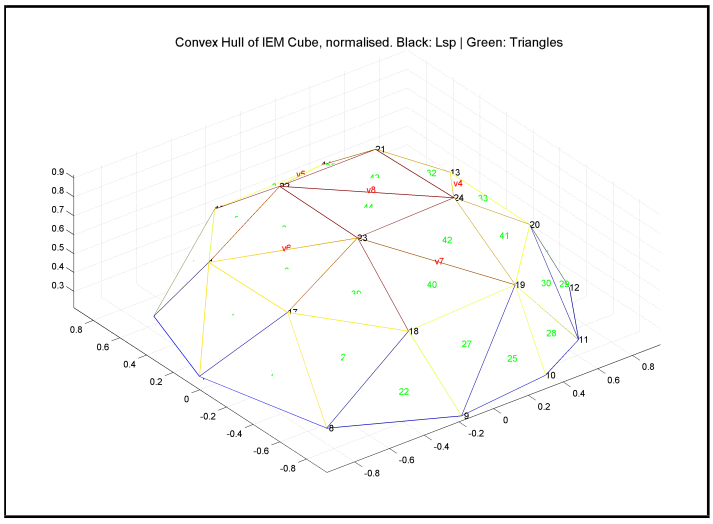
3. This bool is checked to determine, if execution may continue as planned. If not, a failsafe resets the plugin to the default menu preset (so that at least processing can be resumed; otherwise the plugin will never leave the suspended state).

4. If the bool is true, the giant function computeEverythingHull() is executed. During the process, various messages (error or success) will be generated and appended to the content of what the user sees in the GUI textbox. In any case of error occurance, the plugin will be reset to the default menu preset.

5. The convex hull is computed based on the current coordinates (in setCurrentTris(); the *Newton Apple Wrapper Algorithm* is used, see [6]). The result is a list of "Tri" entities (managed by an std::vector), a "Tri" is a data struct defined in the convex hull algorithm. It contains data about each triangular face of the computed convex hull, like the three corner points (representing the loudspeakers in the setup that span those triangles, see illustrations in section 4), the normal vector of the surface and the border lines of the triangle.

6. The convex hull algorithm re-sorts all coordinates before calculating the hull. So in order to actually use the calculation result, at least the original order needs to be known, so the corresponding results can be re-associated. This is done in getCorrectChannelOrder(); which links the "NAW" IDs to the original IDs.

This convex hull would already work. But there are certain problems to be expected in audio reproduction: Although the convex hull *always* consists of triangles, some of these triangles may come to lie in the same plane. For a satisfying audio localisation it is recommended to take these cases into account.



Example: Four points 18, 19, 23 and 24 of the hull lie in the same plane – they span a rectangle, which is tragically divided into two triangles. In case the source is in the middle of the rectangle, only the two diagonally opposite loudspeakers (connected by the common border line) will play – whereas all four corner loudspeakers *should* be active in

this situation. The plugin must thus detect these cases, implement a solution (v7), and translate this solution onto the actual current louspeaker configuration.

7. The surface normals of the triangles are compared. In order to do this and not miss anything, all possible combination pairs are gathered in a huge vector container. Using these combinations, the normals are checked pair-wise if they are parallel. If they are, this particular combination is saved in sameplanes.

8. Only unique combinations are rectangles. If triangle IDs occur multiple times, there are more than 4 corner points in the same plane. This case is assumed to occur only in the bottom-most layer of speakers (ground plane). For this case, there is a special virtual speaker assigned later. If this case occurs anywhere else, it is ignored and left untouched – the default triangles will handle these cases just fine. The sameplanes variable is reduced to only contain combinations representing rectangles.

9. The centre of each rectangle is computed. A virtual loudspeaker is created right there (with a tiny offset towards the outside of the hull as to guarantee it will be part of the new – extended – convex hull). Each virtual loudspeaker has an ID (continuing after the last ID of a real loudspeaker), a dim-factor (see section 5.3), and a list containing the neighbouring real loudspeakers.

10. A special virtual speaker is created at a position below the ground level if every z coordinate > 0 — which is usually the case[20]. This will spread the audio signal among all groundlayer loudspeakers accordingly, as soon as the elevation angle slider claims negative elevation (i.e., depression). A dim-factor of 0.5 is assigned so the audio will fade as the elevation approaches -90°. It is important to note that the listener position *must always be inside* the convex hull for the VBAP algorithm to find active triangles.

11. The new extended convex hull is computed, including the new virtual speakers. Once again, the algorithm re-sorts, so the order needs to be restored (getCorrectChannelOrderExt()).

All variables required for the real-time VBAP computation are set now. The routine returns to the refreshLspSetup() wrapper function and resumes processing, using the new configuration.

[20]because nobody in his right mind would mold speakers into the room's floor... — all joking aside: The listener is always assumed to be in the origin of the coordinate system. But if an elevated listener position is intended (the listener is dangling in the air in the middle of the room for some reason... joking re-enabled!), coordinates with $z < 0$ may occur. Then the special virtual speaker will not be inserted.

5.7 Algorithm: VBAP real-time

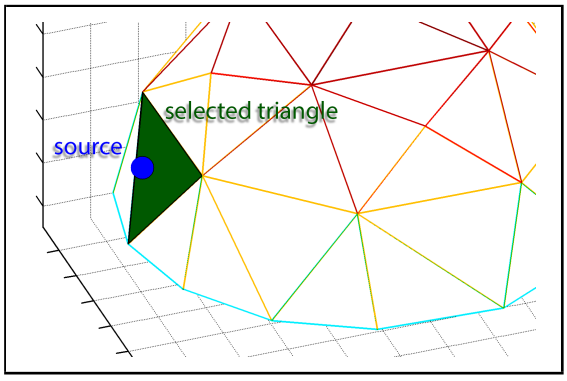
This sections briefly describes the implementation of the VBAP method (see equations in section 4) encompassing the functions prepareToPlay() and processBlock(). The most prominent parameters are the **position of the phantom source** and the **loudspeaker gains** associated with it. VBAP is computed in a block-wise fashion, comparing position data (to determine if a new gain computation is even required) and fading gains. So before the block processing begins, an "old" version of the position as well as the gain vector are necessary. That is why – in prepareToPlay() – the slider values are read once to calculate the old source position srco, and, for this position, calculate the corresponding gain vector G_final.

In processBlock(), the first thing to do is check the host application's channel configuration. VBAP requires exactly one input (monophonic) and outputs depending on the number of loudspeakers. For reasons of lacking time, a sophisticated channel management has been substituted by the simple work-around of requesting the maximum number of channels available in any host program so far (64) and occupying only those needed, silencing the remainder by a zero gain multiplication. The plugin assumes 64 channels to be given. If they aren't, it won't process anything.

There are several more checks in place: If the input buffer is empty, nothing needs to be done, the output will also be empty using a simple clear command. If there is no source movement, nothing new needs to be computed, the old gain is simply applied to the buffer.

If the source did move slightly (or if the precision button is disabled), the gain will be recomputed by the calculateHullGain() function implementing the VBAP method. All zero gain entries will clear the corresponding channel, while non-zero gain values will be crossfaded (old to new gain value, linear) using the JUCE buffer.copyFromWithRamp() function. This will be the most common situation during processBlock.

Else (if the source did move more rapidly or the precision button is enabled), the principle of orthodromic distance will be applied (see next section). But in order to conclude this section, here is a brief description of the **VBAP implementation** in calculateHullGain():



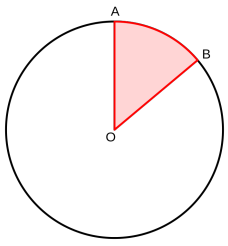
- Current source position vector passed as input argument
- Local gain vector G created (will later be the return value)
- A loop cycles through the entire extended convex hull, searching all triangular surfaces one after the other, picking the VBAP matrix **L** each time and computing the gain triple (vector) **g** (see equations in section 4), until it finds one (*the* one) possessing all positive values > 0. The loop may stop early now.
- If the toggle switch is enabled, the square root of the gain vector is taken element-wise (rendering it an *intensity* panning). Otherwise, the gains are left unmodified.
- Gains are then normalised.
- An if-statement checks whether there is a virtual loudspeaker involved in the selected triangle. If so, the (real existing) neighbours of this virtual are retrieved, and the gain of this virtual is split among all neighbours according to sound intensity law. If not so, normal hull and extended hull are the same in this spot. The correction of the loudspeaker order needs to be applied here, otherwise there will be complete gain chaos (this happened during early testing!). G is filled with those gain values at the indices corresponding to the appropriate channels to which the loudspeakers are routed.
- Finally a gain compensation of a fixed -1 dB is applied (because tests revealed that the method including virtual extensions may cause slight overshoot under extreme circumstances).

As a last step in processBlock, depending on which entry in G_final is non-zero, the active loudspeakers are determined and packed into the dedicated LockedString for thread-safe GUI access and display.

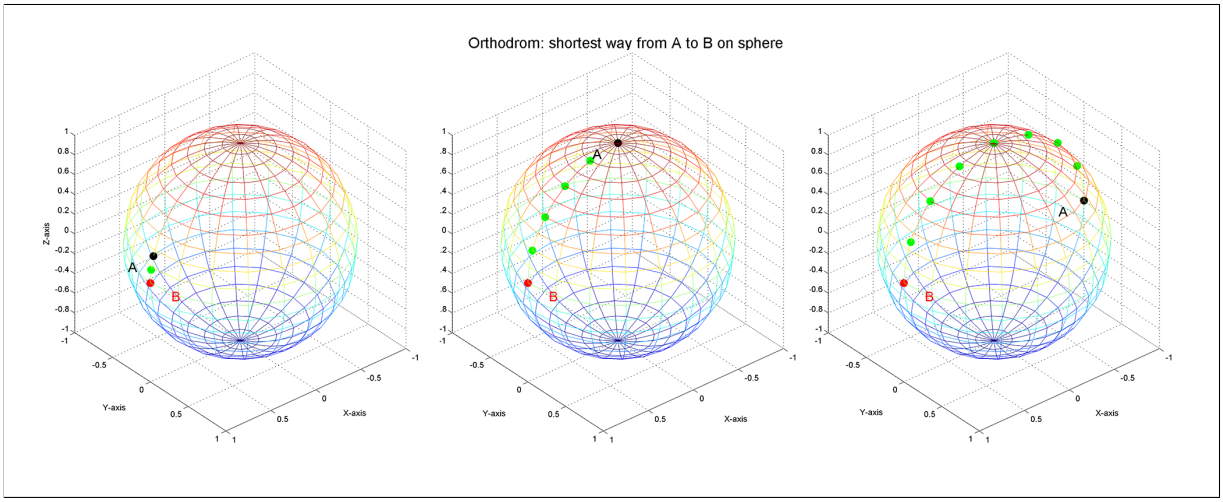
5.8 Algorithm: Orthodromic distances

This algorithm is my own little addition to improve the real-time VBAP. It has been prototyped and testet in MATLAB before implementing it in C++.

Two points (given in spherical coordinates, with elevation θ and azimuth ϕ) are lying on a unit sphere (radius = 1). Then the shortest distance between these two points – taking the path *on* the spheric surface – is defined by the arc segment enclosed by their central angle. It is called orthodromic distance or great-circle distance. Using the spherical law of cosines, it is calculated in the following manner:



$$\begin{aligned} d = \mathbf{arccos}(&cos(\theta_A)cos(\phi_A)cos(\theta_B)cos(\phi_B) + \\ &cos(\theta_A)sin(\phi_A)cos(\theta_B)sin(\phi_B) + \\ &sin(\theta_A)sin(\theta_B)) \end{aligned}$$



The distance influences how many intermediate points are inserted. If processBlock goes for the orthodromic interpolation, it has checked source movement already, which is great enough to justify at least 1 intermediate point. The number of points lies between 1 and 7, defined empirically by:

$$n = round(5 \cdot \sqrt{d} - 1)$$

The spacing is equidistant. The intermediate points are calculated by defining a corresponding equidistant time spacing and evaluating the arc (trajectory) at each step, using a Gram-Schmidt orthonormal basis to get Cartesian coordinates in the result (for compatibility with computeHullGain).

The audio buffer is split into segments in accordance to the time spacing. The VBAP method is applied in each segment separately, resulting in a far more accurate source movement.

6 Instructions: Custom loudspeaker setup

The most pleasant quality of vector base amplitude panning is that it can be applied to literally any loudspeaker configuration. This section clarifies the regulations for correct, acceptable coordinate input in the plugin's GUI. A short version can be found in the appendix plugin manual.

Selecting the **Custom** slot in the preset menu will – in contrast to the other presets – not instantly calculate a configuration, but provide information about the input format in the textbox:

> Enter custom speaker setup:
> → use carth. coordinates [x,y,z] (float)
> → last ';' signifies the end.

→ Format:
9.23985295, 7.38439853, 13.92850935;
...
2.98938455, 4.92838593, -7.92839521;

In order to avoid any misinterpretations of your input, make sure to clear the entire content of the textbox before pasting your own coordinates. It is recommended you stick to the format rules as precisely as possible (although the program takes care of some minor errors). The separating comma between the coordinate entries is absolutely necessary, as is the semicolon concluding every coordinate (the spaces are optional). The parser reads through the whole input string and stops at the very last semicolon (signifying the last coordinate). Each segment in between separators is read in as a floating point variable and stored in a vector container holding the entire coordinate matrix.

The values in those segments may be zero, negative zero, already normalised values, values sharing the same unit of length (e.g., metres, inches, feet, yards...) up to a magnitude of 100, positive or negative sign.

Clicking the **Verify Input** button will trigger the computation. The pasted coordinates will be kept in the textbox contents, messages about the computation progess will be added at the top. In case something is wrong, you will get specific information about it, so you can look at the pasted string again and retry.

Once your coordinates have been accepted, they also become normalised if necessary. Normalisation includes removing a possible z-coordinate offset so that the ground layer of loudspeakers lies approximately in the xy-plane. Virtual speakers will be added if necessary. Hit the **Show All** button to attain all information about the convex hull of your configuration. The coordinates will also be displayed in this information package, so nothing is lost.

If you save your audio project in the DAW host, the plugin's state will be saved as well. It will remember that you selected the **Custom** slot and will restore your input configuration in your next mixing sessions. Should you, however, decide to change the selected slot to one of the presets (or hit the **Clear & Reset** button), the plugin will forget your custom input, and on reselecting the **Custom** slot it will show the format message again. You will need to re-enter your input in this case. It is recommended you keep your own configurations, e.g., in textfiles as a backup.

Last but not least, here is some MATLAB code to help you generate coordinate input (in case the approximate loudspeaker positions in azimuth and elevation are known). The generated coordinates are following the input rules of the plugin and are already normalised:

```
% Enter your custom coordinates as azimuth and elevation
Azi = [0 30 -30 45 -45 90 -90 135 -135 180];
Ele = [0  0   0 45  45  0   0   0    0  55];

Setup = zeros(size(Azi,2),3);

for h = 1:size(Azi,2)
    xs = cosd(Azi(h)).*cosd(Ele(h));
    ys = sind(Azi(h)).*cosd(Ele(h));
    zs = sind(Ele(h));

    Setup(h,:) = [xs ys zs];
end

norms = sqrt(sum(abs(Setup).^2,2));

Setup_normed = repmat(1./norms,1,3) .* Setup;

X = Setup_normed;

for i = 1:size(X,1)
    fprintf('%.8f, %.8f, %.8f;\n',X(i,1),X(i,2),X(i,3));
end
```

To provide a complete example of what a valid user input configuration coordinate set looks like, the output of the above MATLAB code is printed below:

```
1.00000000, 0.00000000, 0.00000000;
0.86602540, 0.50000000, 0.00000000;
0.86602540, -0.50000000, 0.00000000;
0.50000000, 0.50000000, 0.70710678;
0.50000000, -0.50000000, 0.70710678;
-0.00000000, 1.00000000, 0.00000000;
0.00000000, -1.00000000, 0.00000000;
-0.70710678, 0.70710678, 0.00000000;
-0.70710678, -0.70710678, 0.00000000;
-0.57357644, -0.00000000, 0.81915204;
```

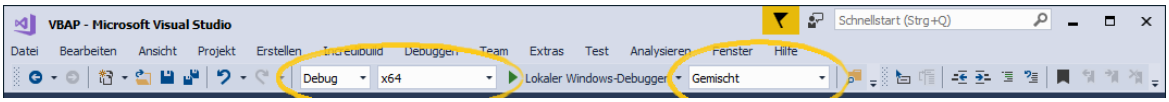
Note: The input values do not need to have as many decimal places as in the example. It is however recommended for them to contain the decimal point, because for a yet unknown reason, if the input is for instance `0, 0, 1;`, it will not be recognized by the verification. Try `0.0, 0.0, 1.0;` instead.

7 Plugin testing

This section hints at how to properly test a plugin. The testing goes way beyond solving compiler errors, because the plugin is a dynamic library built for runtime operation. Having spent lots of time dealing with – and finally understanding meanings of – compiler messages, one feels extremely relieved when the code finally compiles without errors. The result is then pulled into a favourite host application to see how it looks... and **BOOM!!** Your host application crashes and you are desperate and clueless as to what went wrong. And believe me – a lot can go wrong, even if the code compiles flawlessly!

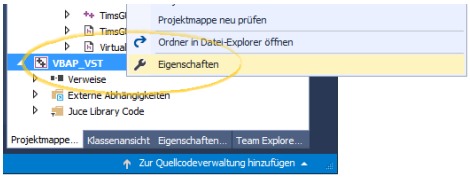
Fortunately, there are ways to debug the plugin during runtime. This is something Projucer didn't take care of for once – you need to set it up yourself. However, it created a debug exporter for you. You simply need to make some changes in the configuration – but this time not within the Projucer, this time in your IDE. I will show the process for Visual Studio 2017. Similar settings can be made in Xcode, CodeBlocks etc.

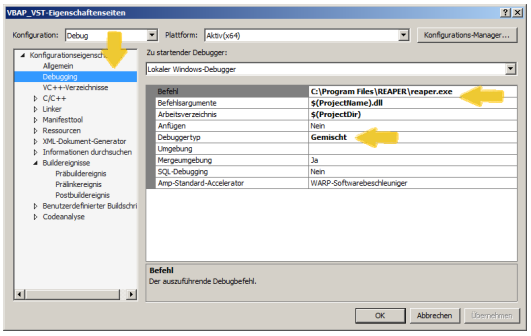
Select **Debug** mode (let platform set itself) and choose **Gemischt** (Mixed) debugging



Mixed mode is important, otherwise the debugger won't notice break points you set in your plugin code as the host application loads and uses it. Now, how do I make the debugger interact with my favourite host application (e.g. Reaper)?

Right-click on the **bold-faced project name** in the solution explorer (the startup project that is being built by default) and select **Properties** (Eigenschaften).

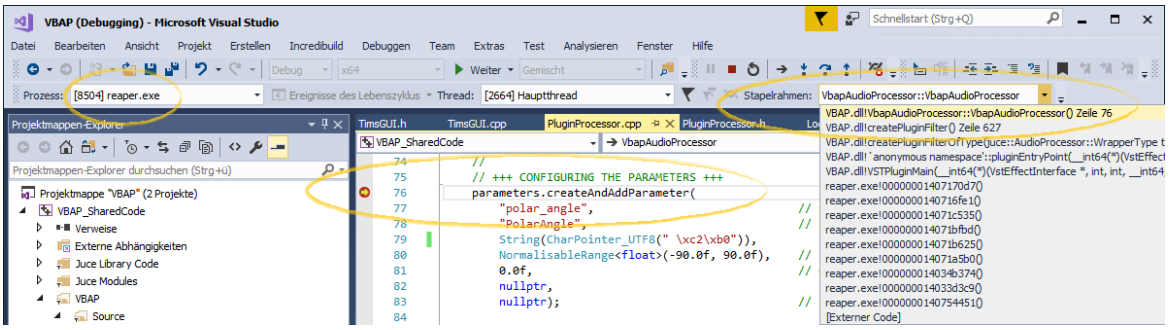




In the configuration settings, go to **Debugging** and modify **Command** (Befehl) to contain the path to the .exe of the desired plugin host application (e.g., Reaper in this case). In **Command arguments** directly below, you need to hand the compiled .dll over to the host → **($ProjectName).dll**. The type of debugger is set to **Mixed** here as well, according to what was set in the first step above.

If you now compile in debug mode and start the (local Windows) debugger, the host application will start automatically – and check for newly added plugins. If you have set a break point within your plugin's constructors code, your IDE will pop up and suspend operation of the host application until you *continue* or step through etc. If the plugin's initial check by the host turns out to be fine, you can go ahead and try to load the plugin into an audio track. This will cause the debugger to become active again. Now you test what your plugin does in live action. If something comes up during runtime, you will usually get assert messages or exceptions. The recent history (**stack frame** = Stapelrahmen) prior to the bug gives you hints about what may have caused the problem.

The screenshot shows an example debug session. A break point is set in the plugin class constructor before the parameter polar_angle is created. Reaper.exe is currently running, but halted due to the break point. The stack frame (Stapelrahmen) shows which function in the code is currently in execution (it's the VBAP plugin constructor) and lists other function calls that happened before, leading to the plugin's constructor function. These may be functions of your own written code, or of code included by external headers (e.g., the JUCE framework, wrapping your code to fit the VST specifications in this example) or machine code of the host application (which obviously is not shown to you as actual code, because has been compiled).



Briefly: An issue came up while debugging, concerning the auto-generated code of the Projucer's GUI Editor feature: As components such as buttons or sliders are added in the *Subcomponents* tap, they are automatically registered as being listened to by the main GUI component (e.g., polarslider->addListener (this);). To avoid bugs, it is recommended to manually un-register those listeners in the user-defined section of the GUI's destructor: //[Destructor_pre] ... polarslider->removeListener(this);

Remark: If you run into asserts, the JUCE code is very well commented in those code areas where programmers are expected to end up. For example, when it comes to string coding, all kinds of platform-dependent issues may occur. Then JUCE comments explain how to achieve the correct symbols avoiding misinterpretations.

8 Conclusion

The plugin is up and running. Looking back, I realise that the code solutions I came up with may not be the most efficient, most readable, most maintainable or most elegant way of design or implementation. More experienced programmers may even throw their hands up in despair. Having attended some lectures and laboratories involving C programming and assembler, this project is my first adventure towards C++. I started familiarising myself with the C++ language basics (advancements since C) by writing tiny tutorial programs.

When I felt confident enough I started reading *Designing Audio Effect Plug-Ins in C++* [7] and did most of the examples discussed in the book, which are focussing on the core DSP processing for the greater part. But the resulting plugins kept conveying the impression to be tied to the associated host program *RackAFX* written by the author, Will Pirkle, additionally the compiled dll files were huge, containing more than 80% unnecessary content in my specific case of application.

So I briefly tried *IPlug*, but since there were no tutorials or guides on how to start or proceed (there is a framework documentation though), so I ran into the brick wall of lacking programming skills.

Things went on much smoother on my next attempt in JUCE. It took quite some time finding my way around the API documentation and JUCE's coding paradigms. In the process, I have learned to very much appreciate what JUCE has to offer, and have finished the VBAP project using this framework. I also have to acknowlegde the most helpful capabilities of Visual Studio 2017 which go way beyond syntax highlighting – and which made me write large portions of code in VS2017 instead of the Projucer (even though I advised against it earlier in this documentation). Superior features like IntelliSense, advanced code search and navigation just make a programmers life so much easier.

This project spanned the time of almost a year from its announcement to its completion; it has been interrupted by unfortunate personal events, superceded by more important tasks and put on hold because of knowledge and skill troughs. But in the end I completed the plugin, learning a great deal about programming in general and C++ plus the VST environment in particular. For all people who want to take on the adventure of VST plugin programming, I gathered information and descriptions in this project documentation (and also in the appendix (8) Q&A) in order to ease their path – so they shall not end up spending a whole year like me ⌣̈

I deem this project to be at least a great personal success and hope that it may be of use to others in the future.

References

[1] V. Pulkki, "Virtual Sound Source Positioning Using Vector Base Amplitude Panning", *Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology, FIN-02015 HUT, Finland*, 1997. [Online]. Available: http://lib.tkk.fi/Diss/2001/isbn9512255324/article1.pdf.

[2] TERAGON, *How to make VST plugins in Visual Studio*, Online, 2012. [Online]. Available: http://teragonaudio.com/article/How-to-make-VST-plugins-in-Visual-Studio.html.

[3] TERAGON, *How to make your own VST host*, Online, 2012. [Online]. Available: http://teragonaudio.com/article/How-to-make-your-own-VST-host.html.

[4] Steinberg, *Virtual Studio Technology Plug-In Specification 2.0 Software Development Kit*, Media Technologies GmbH, 1999. [Online]. Available: http://jvstwrapper.sourceforge.net/vst20spec.pdf.

[5] RedwoodAudio, *JUCE 4.x for VST Plugin Development*, 2012. [Online]. Available: http://www.redwoodaudio.net/Tutorials/juce_for_vst_development__intro.html.

[6] D. A. Sinclair, *A 3D Sweep Hull Algorithm for computing Convex Hulls and Delaunay Triangulation*, Online, 2016. [Online]. Available: http://www.newtonapples.net/paper/NewtonWrapper.pdf.

[7] W. Pirkle, *Designing Audio Effect Plug-Ins in C++*. Taylor & Francis, 2012, ISBN: 9781136699764. [Online]. Available: https://books.google.at/books?id=QddcxHLavrMC.

[8] JUCE, *API documentation*, online, 2017. [Online]. Available: https://www.juce.com/doc/classes.

Appendix

A: How to make a VST plugin? (Q&A)

How to make a VST plugin?

Q&A-style

This document addresses the issue of where to begin creating a self-made VST plugin. When I found myself in this situation, I experienced that it is not at all easy to find hands-on information on how to start, what is required in terms of knowledge and skills, but also in terms of programs and additional components. Where to begin? What important aspects to consider?

Reading this Q&A will give you some ideas (and hopefully answers) about what VST plugin programming is all about and provide you with a pool of material to proceed from.

This serves as an appendix to my *Toningenieurs-Projekt* university-related work at IEM Graz.
Tim Raspel, written in September 2017.

List of Questions

Q1: Which skills are useful if I want to make a VST plugin?

Making VST plugins is an interdisciplinary challenge. Having the idea of "I'd like to make my own plugin now" may be quite challenging and depends on how much you already know. Basically you require knowledge/experience in the following areas:

- **Audio basics:** What is sound (waves, frequencies etc.) and how is it represented digitally (sampling, impulse trains, quantization, buffers)? If you're not familiar in this field, there are many helpful books out there, and even introductory PDF files or lecture slides[1].
- **Programming:** The VST SDK is natively written in C++, so knowledge of C/C++ or similar languages gives you an advantage. If you do *not* know how to program, it might not be the best idea to start off with a VST plugin as a first project. In fact, many people advise against this and recommend learning C/C++ *before* attempting a VST plugin. Literature and tons of tutorials about learning C/C++ are widely available.
- **Math:** Depending heavily on what you are planning on doing in your plugin, having some (or more... or less) knowledge of basic engineering math (like linear algebra, complex analysis) is a key aspect. Filter design requires at least fundamentals in Fourier (and similar) transformations. Most of the time, solid math skills will be helpful, also for designing little functions or algorithms along the way.
- **Digital signal processing:** You need some insight into how digital data is manipulated, what common patterns are and what they do, e.g. FFT, number formats, quantization etc.
- **Audio signal processing:** You will require specific knowledge about digital signal processing applied in the field of audio, e.g., filters, delays, dynamic processing, spatial processing, reverb etc. – depending on what your plugin is supposed to do.

Q2: I am an absolute beginner, are there very easy ways to make a VST plugin?

Yes, there are. But you won't plunge deep into the topic by choosing this approach: applications like *SynthEdit*, *Synthmaker/Flowstone* or *Reaktor* allow for making a complete plugin by graphical design – you can drag&drop elements, connect them, create your functionality this way.

If you want to dig a little deeper, I recommend Will Pirkle's *Designing Audio Effect Plugins in C++*. He supplies a freely downloadable software named *RackAFX*, which is basically a VST host program plus added functionality for your plugin developement. The book has lots of example code, starting with easy tasks like a gain fader, continuing with simple equalisers and advancing with delay effects (e.g., flanger) and dynamic processing. All you need is RackAFX, a compiler of choice (e.g., VisualStudio or CodeBlocks) and this book. You get to look at working C++ code without instantly being forced to start coding on your own. When you've got used to the process, the book gives you some *do-it-on-your-own*-tasks, but provides the solutions as well.
Pirkle has published another book about VST instruments, if you are more interested in that: *Designing Software Synthesizer Plug-Ins in C++*.

[1] http://www.hep.upatras.gr/class/download/psi_epe_sim/1_Basics_DSP_AV_Intro.pdf
https://www.princeton.edu/~cuff/ele201/kulkarni_text/signals.pdf

Q3: I am already a little experienced, are there very easy ways to make a VST plugin?

If you are already experienced in C++ programming in one way or another, there are other nice existing tools – and using those won't require you to start from zero. Frameworks like *JUCE* or *iPlug* provide you with a C++ codebase that already implements many useful things when it comes to VST programming, e.g., plugin templates you can start with and add your own code, or background code that takes care of cross-platform issues for you.

If you are a C++ expert and want to dive in the direct way, read the *Steinberg Virtual Studio Technology Plugin Specification* on how to use the AudioEffectX class.

Q4: How do I proceed when developing a VST plugin? What are the steps?

1. **Vision:** Come up with a unique idea. What is your plugin supposed to do? Is a similar plugin already out there? Or maybe you want to create a plugin for learning purposes? Then choose something simple to start with (usually a gain fader), and if you feel comfortable after that, continue with a low order EQ etc.

2. **Signal processing:** Design the core of your plugin. In case of a gain fader, there needs to be a multiplication, maybe two if you want stereo and so on. As it becomes more complicated, you may want to test what's happening in a nice environment like *MATLAB* and then **translate** it to C++ later. You may need to find and #include some additional C++ libraries along the way, as many *MATLAB* functionalities are high-level and do not have a direct C++ equivalent. Maybe you need an FFT? Or you want to use special vectors in your plugin?

3. **UI design:** Which parameters of your processing is the user allowed to play with when using the plugin? Or a bit advanced: Come up with a comfortable design in terms of interaction and workflow – if you were the user, how would you like your plugin to be? Smooth and easy... Colors, buttons & fancy images should be considered *after* defining the parameter access!

4. **UI developement:** Create your UI. Connect it to the core processing of step 2. Using *JUCE* or *iPlug* definitely cuts corners in this step!

5. **UI refinement:** Now, the elementary stuff should be working. Take some time to shine up your UI (especially when planning on selling it as a product).

6. **Optimization:** Well, the plugin may be working, but some components/processes/elements are slow. Identify those and improve their performance.

7. **Testing:** Is the plugin compatible with different operating systems (Windows XY) and/or different host programs (Ableton, Cubase, Reaper...)? And there is the x86/x64 issue... This step could take quite some time.

8. **Maintenance:** During testing or later use, bugs will appear that need fixing. Also, new ideas or suggestions may pop in on how to improve/enhance the plugin's functionalities. Be prepared to go back to the earlier steps and revisit/modify your work. Think about versioning.

9. **Licensing & release:** Licensing is important for commercial use as well as for public distribution (see Q7)! Maybe you want some kind of copyright protection. There are options like Pace iLok or Steinberg eLicenser.

Q5: What do I need to start programming a C++ VST plugin?

This website[2] answers your question very well. It even describes how to set everything up and gives hints about troubleshooting common problems.

The short answer is: you need

- the Steinberg VST SDK
- an IDE (integrated developement environment) = code editor + compiler
- skills mentioned in Q1

Q6: Programming errors everywhere! What do I do?

A solid C++ reference is always helpful. Check out if your syntax is correct when using certain types, classes, functions, etc. Sometimes you get 43 errors just because you forgot to close a bracket or something.

If you don't know what the error message of the compiler wants to tell you, ask `stackoverflow.com/questions/tagged/c++`. People before you have likely gone through the same ordeal. If you really don't find an answer, post your problem and maybe there's someone who can help you.

In case of VST plugins, there may be errors beyond compile-time, because the plugin is a .dll file. That means that it will compile just fine, but when you load the plugin into your host, the host crashes and you can't tell why because the host's crash dump file is even more cryptic. Modern IDEs (like Visual Studio) have the possibility of running your dll in a debug mode involving the host program. You need to set the path to your host program in the projects debug options. Then just set your debug points in the code, compile a debug version and start the debugger. Your host program will automatically start up and you can step through your code and find the root of all evil.

Also, if you are using additional frameworks or libraries, always consult the documentation. Especially when using the *JUCE API*, you get an excellent description of everything in there.

Q7: May I publish/distribute/sell my VST plugin?

Licensing is quite an important point when juggling with source code. And it becomes an issue as soon as you don't just use your plugin on your personal computer, but start redistributing it in some way.

[2] `http://teragonaudio.com/article/How-to-make-VST-plugins-in-Visual-Studio.html`

You need to take care if there are any restrictions on code snippets you copied from the internet (e.g. GitHub), libraries you included in your project etc. The VST SDK will be a part of your project. Currently, it has a "dual-license", making the SDK available for commercial developers as well as open source use. The licensing may change from time to time, so check what the current license text says[3].

There are basically two types of licensing: closed source and open source. The former restricts the use of source code with the intention of commercial profits. Software developement is an important industrial branch, thus there is an interest in preventing proprietary code from being freely accessible. If you plan on releasing a software product (like a bundle of VST plugins for example), you have to make sure the code you use is properly licensed for your purpose. It must not contain anything licensed under GPL, which takes us to the latter type of licensing: open source.

GPL[4] (general public license) aims at keeping code open source. It gives you the freedom to use the software for any purpose, to modify it in any way you wish, and to share it (including mods) with anybody you want (without breaking any other laws of course, e.g., export laws).
You can mod, mix (including with any other source code), match, run, compile... GPL code any way you want without any obligation – as long as you don't redistribute it. Note that you cannot modify the license itself or sub-license the GPL code.

If you distribute an executable code that contains GPL code: Most often you must give access to ALL the source code as well, including sources not originally under GPL:

- If two codes (interacting closely together) don't have compatible licenses you cannot redistribute them together. Period.
- If your GPL executable is distributed in order to run and interact closely with another code, you must give access to both source codes to the users (in terms of VST: it is a DLL, so yes – linking usually means interacting closely).
- Running your proprietary software written in PhP, on top of Linux and using MySQL does not mandate any redistribution of your proprietary source code.
- The mere distribution of executables alongside each other on a medium (or through a network) doesnt trigger any source-code-distribution contagion; for instance: a Linux distro commonly contains many files under various licenses, sometimes incompatible with each other.

You can distribute GPL code any way you want (including for a fee) as long as getting the source code is not made harder for the user than getting the executable. This means that:

- If users pay for the executable they cannot be charged more for getting the source.
- If a certain procedure has to be respected for accessing the executable then the procedure to access the source cannot be substantially harder, longer, more complicated or expensive etc.
- Access to the source code should be given with enough information/tools/files including the source code of appropriate libraries, so that it is reasonably easy to compile into a working code.

[3] VST 3 SDK Licensing FAQ: https://sdk.steinberg.net/viewtopic.php?t=286
[4] see: blog.milkingthegnu.org

By authoring code under the GPL (including mods) you give away all patent rights that would contradict the terms of the GPL (e.g., you cannot prevent people to use, mod, and share the code as stated by the GPL). If you violate the terms of the GPL in good faith you can be re-established in your rights after showing compliance.

This short and plain description of GPL is included here, because it is the most common license. Of course it is not the only open source license. Some others are *Apache, MIT (Expat/X11), BSD, MPL, CC0*, but there are dozens more[5].

Important note: If source code does not carry a license to give users the four essential freedoms, then – unless it has been explicitly and validly placed in the public domain – it is not free software.

Some developers think that code with no license is automatically in the public domain. This is not true under today's copyright law; rather, all copyrightable works are copyrighted by default. This includes programs. In some countries, users that download code with no license may infringe copyright merely by compiling it or running it. In order for a program to be free, its copyright holders must explicitly grant users the four essential freedoms. The document with which they do so is called a free software license (see the examples above). This is what free software licenses are for.

Q8: Where can I find more information about VST plugin making?*

Generally nice sites to consult:
www.kvraudio.com/forum
www.teragonaudio.com/article/How-to-make-VST-plugins-in-Visual-Studio.html
www.willpirkle.com
www.plugindeveloper.com/tag/iplug
www.martin-finke.de/blog/articles/audio-plugins-001-introduction
www.image-line.com/support/FLHelp/html/plugins/Synthmaker.htm
www.synthedit.com
www.stackoverflow.com/questions/2581025/how-are-vst-plugins-made#2581339
www.asktoby.com/TobyNewman-Dissertation-howtowriteaVSTplugin.pdf
www.redwoodaudio.net/Tutorials/juce_for_vst_development__intro.html
www.quora.com/What-are-the-steps-to-writing-a-VST-plugin?share=1

C++ frameworks:
www.steinberg.net/en/company/developers.html
www.juce.com
www.taletn.com/wdl

PureData VST wrapper:
puredata.info/
github.com/pierreguillot/Camomile

* Sources: This FAQ summarizes information given in some of the listed links for the purpose of having a short and clear overview on how to approach VST plugin programming.

[5] license list + GPL compatibility: https://www.gnu.org/licenses/license-list.en.html

B: VBAP plugin manual

VST Plugin Manual: Vector Base Amplitude Panning

{ Author: Tim Raspel, Supervisor: Franz Zotter } Kunst-Universitaet Graz (IEM)

Description

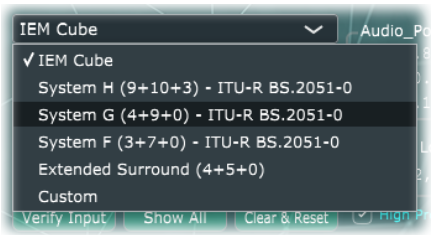
With the **Vector Base Amplitude Panning VST Plugin** you can position a monophonic sound source (1 input channel) on 3D playback setups with an arbitrary number of loudspeakers (max. 64 output channels). The DSP core of the plugin is based on the VBAP method described in [1]. The convex hull computation algorithm used is called *Newton Apple Wrapper* [2]. JUCE framework and Eigen template library are also included.

Setup

Compile the project (VS2017) and copy the resulting VBAP.dll to your local VST Plugin folder (e.g. C:\Program Files (x86)\Steinberg\VstPlugins). If this path is known by your DAW, it will be scanned on startup, showing *Vector Base Amplitude Panning* as a newly found plugin. This plugin has been tested in Reaper. Other host programs may cause previously undiscovered problems.

Presets

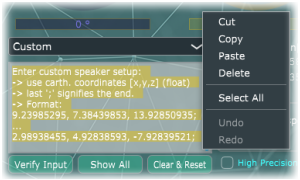
The plugin provides 5 coordinate presets:
IEM Cube contains the speaker coordinates of the institute's main 3D audio performance space. It has 4 top-level, 8 mid-level and 12 ground-level speakers (the LFEs are controlled separately).
System H to **Extended Surround** (System D) are standardised speaker setups defined in ITU-R BS.2051-0 norm. Since it allows tolerance in position, see project documentation for exact values.



(4 + 9 + 0) signifies the count of upper, middle and bottom positions. Bottom is usually assigned to LFEs only. This scheme does not apply to IEM Cube, because it does not account for the additional mid layer.

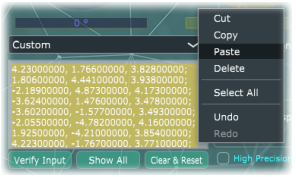
Using Custom Coordinates

Select the **Custom** option in the menu. A helper text gives you specific instructions.



Delete that text, then paste your coordinates (**the order will be used as channel order!**).

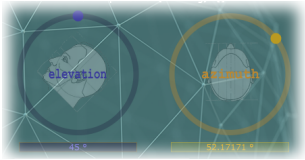
Hit **Verify Input**. The plugin will check your input and keep you updated on the process of the computation.



Your own custom setup will be loaded when you restore a saved project. **Caution**: Once you select **Custom** again (or another preset), your pasted coordinates will be lost!

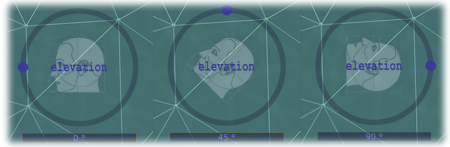
Source Position Controls

The main parameters of this plugin are the elevation and azimuth angle. They may be changed in real-time, they are automatable.

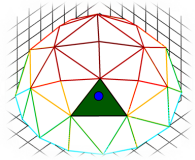


Both blue and yellow dot act as a source position indicator, each relative to its head inside the cirle. Imagine it is *your* head. Use the head's perspective to visually deduce where the source is located. The azimuth head is static. Moving the yellow dot around the circle makes the audio circle around you horizontally.

The elevation is also a full cirle, so you can move the source around you vertically as well. However, the elevation angle is defined from (0°) to ±90°, spanning only *half a circle*. That is why you need to move the blue dot around the circle *2 times* for *one* full vertical audio rotation.



Moving the elevation beyond 90° makes the yellow dot jump, because from the azimuth point of view, the source is now behind the head. If you now move the yellow dot to the front again, the elevation head will react and jump to correct the position relative to the blue dot.



To emphasize what is happening in the background: You are in the centre of a bubble, and the two angles represent *one point* on that bubble's surface. The internally computed *convex hull* matches the chosen loudspeaker coordinates to approximate the bubble's surface. The plugin then finds the loudspeakers closest to that point and assigns them to play the sound (weighted gains, see [1]).

Information Display

The upper box shows the current source position in Cartesian coordinates. The plugin uses **left-handed orientation**. The reason: Moving the yellow slider dot then correctly indicates the source position relative to the head image (which represents the listener). The lower box displays the active loudspeaker's IDs (conform with your channel routing). Update rate is 20 fps.
1 ID = source *at* speaker position
2 IDs = source *on a line* between 2 speakers
3 IDs = source within triangle face of hull (most common)
4 IDs = source within rectangle face of hull
5+ IDs = source near virtual negative-Z-speaker. It distributes gains to the ground level set of speakers.

References

[1] V. Pulkki. *Virtual Sound Source Positioning Using Vector Base Amplitude Panning*. JAES Volume 45 Issue 6 pp. 456-466; June 1997

[2] D. A. Sinclair. *A 3D Sweep Hull Algorithm for computing Convex Hulls and Delaunay Triangulation*. www.newtonapples.net/paper/NewtonWrapper.pdf

Toggle Buttons

This button switches between using raw amplitudes or square-root of amplitudes (intensity) for the panning gain.

Only activate this button if you are planning on moving the source **really fast** (e.g., jumps in automation). The plugin interpolates intermediate points to improve sound movement using the principle of orthodromic distance on the convex hull.

Textbox Buttons

Verify Input	Processes current textbox content; works only if **Custom** is selected
Show All	Displays information about setup, gain, convex hull etc. in textbox
Clear & Reset	Deletes the current setup and loads the default **IEM Cube**

C: Copyright and Licenses

```
Code templates for audio plugins have been generated by the PROJUCER, Copy-
right (c) 2015 - ROLI Ltd. (version: 5.0.2), underlying JUCE Personal license.

Modifications have been made to these templates, using mainly the JUCE API
building blocks, as well as:
- Newton Apple Wrapper Convex Hull algorithm (GPL v3)
- Eigen Vector/Matrix Algebra library (MPL v2)
- Rosetta Code "Combinations" function in C++ (GNU FDL v1.2)

The share of code written by the author is licensed under X11:

Copyright (c) 2017 Tim D. Raspel

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is fur
nished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IM-
PLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNEC-
TION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of the above copyright holder
shall not be used in advertising or otherwise to promote the sale, use or
other dealings in this Software without prior written authorization.
```