VISR User documentation

The S3A project team

Oct 30, 2018

CONTENTS

CHAPTER

ONE

ABOUT

The VISR framework is a collection of software for audio processing that forms the backbone for most of the technology created in S3A. In this extensible software framework, complex audio algorithms can be formed by interconnecting existing building blocks, termed components.

It can be used either interactively in the Python language, in custom applications (for instance in written C++, or integrated into other applications, for instance as DAW plugins or Max/MSP externals. While the VISR provides several renderers and building blocks for spatial and object-based audio, it is nonetheless a generic audio processing framework that can be used in other applications, for example array processing or hearing aid prototypes. The Python integration makes the system accessible, and enables easy algorithm development and prototyping.

CHAPTER

TWO

GETTING STARTED WITH THE VISR FRAMEWORK: OVERVIEW

2.1 Python integration

CHAPTER

THREE

BASIC TUTORIAL

This tutorial explains the first steps for using the VISR framework by creating and running an audio renderer. It is based on the conference paper [].

CHAPTER

FOUR

PEOPLE

CHAPTER

FIVE

GETTING VISR

5.1 Download

The VISR framework can be obtained in diffeent forms. FOr most persons, however, downloading and installing an installer package is the most convenient way to use this framework.

Installation packages can be downloaded from the S3A software download page .

Installation packages are available for the following platforms:

Windows (x86_64) Recent versions (Windows 8 and Windows 10) 64 Bit only

Mac OS X Version 10.11 and above, 64 Bit only

Linux Ubuntu 16.04 LTS and Ubuntu 18.04 LTS, 64 bit

Rasspberry Pi (ARM) Raspbian Stretch, 32 Bit

5.2 Installing VISR

Binary installation packages are the suggested way to uses the VISR framework. A binary installers enables all uses of the framework, including

- Running standalone applications
- Using DAW plugins based on the VISR
- Using the Python interfaces and creating new functionality in Python
- Creating standalone applications and extension libraries in C++

Hint: Building the VISR from source is necessary only in these cases:

- Porting it to a platform where no binary installer exists
- Fixing or changing the internal workings of the framework.

Installation packages are available on the S3A Software download page.

Note: If you plan to use the Python integration of the VISR framework (see *Python integration*), you need to select an installation package matching the Python version you are using, for example **VISR-X.X.X-python36-Windows.exe**.

5.2.1 Windows

The graphical installer is provides as an **.exe** file and provides a dialog-based, component-enables installation. Figure *figure_windows_installer* shows the component selection dialog of the installer. The choices are detailed below in section *Installation components*.

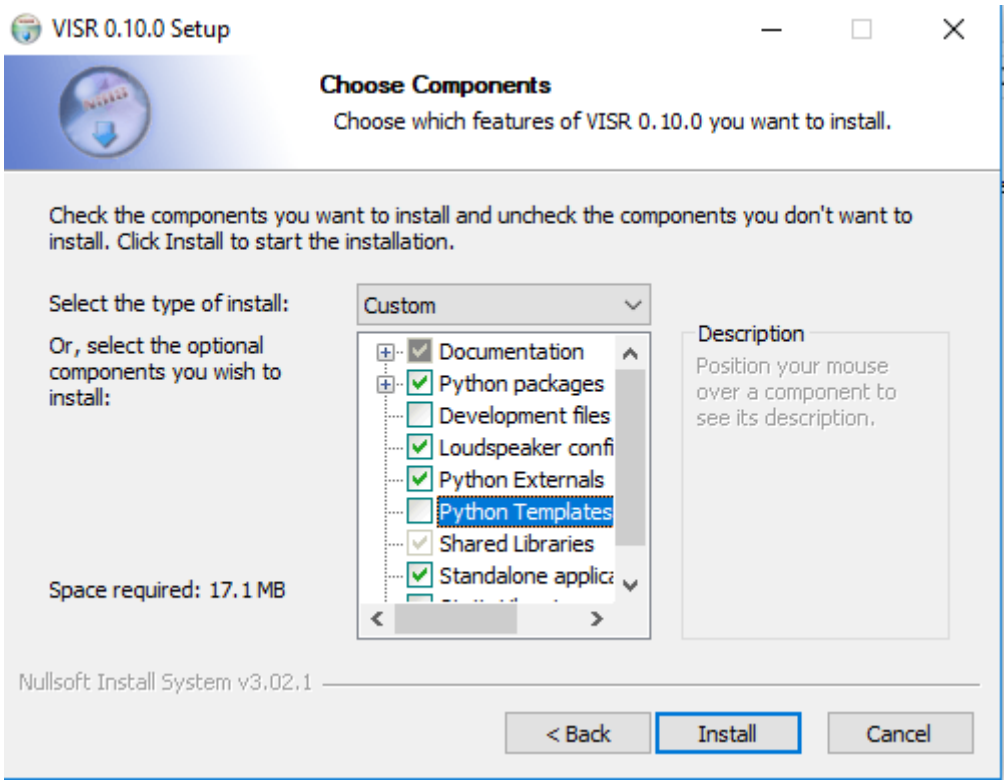

Fig. 1: Graphical Windows installer.

An executable installer (.exe) with a graphical user interface and corresponding uninstall functionality. Supported are 64-bit versions of Windows. If required, install the "Microsoft Visual C++ Redistributable for Visual Studio 2017", package, for example from the Visual C++ downloads page.

On Windows, it is necessary to add the directory containing the VISR libraries (DLLs) as well as the directory containing third-party libraries shipped with the VISR installer to the **PATH** variable. To this end, open the environment variable editor (Settings -> System -> Advanced system settings -> Environment variables). The environment variable on Windows 10 is depicted in figure windows_environment_variables_editor .

Append the value "C:Program FilesVISR-X.X.Xlib;C:Program FilesVISR-X.X.X3rd" if the standard installation location was used. (Note: Replace **X.X.X** with the actual version number of VISR. Depending on your system permissions and whether you VISR shall be used by all users of the computer, you can either set the **PATH** user variable or the **PATH** system variable.

Note: Any applications used to access VISR (for example command line terminals, Python development environments, or DAWs) must be closed and reopened before the changed paths take effect.

Append the path ''<install-directory>/lib" to the path variable, where ''install_diectory" is the directory specified during the installation. For the default path, the setting would be ''c:Program FilesVISR-N.N.Nlib'', where ''N.N.N is replaced by the actual version number. If the **PATH** variable is edited as a string, subsequent paths are separated by semicolons.

Note: Future versions of the installer might adjust the paths automatically. However, as pointed out in NSIS Path manipulation, this needs an extremely cautious implementation to avoid potential damage to users' systems.

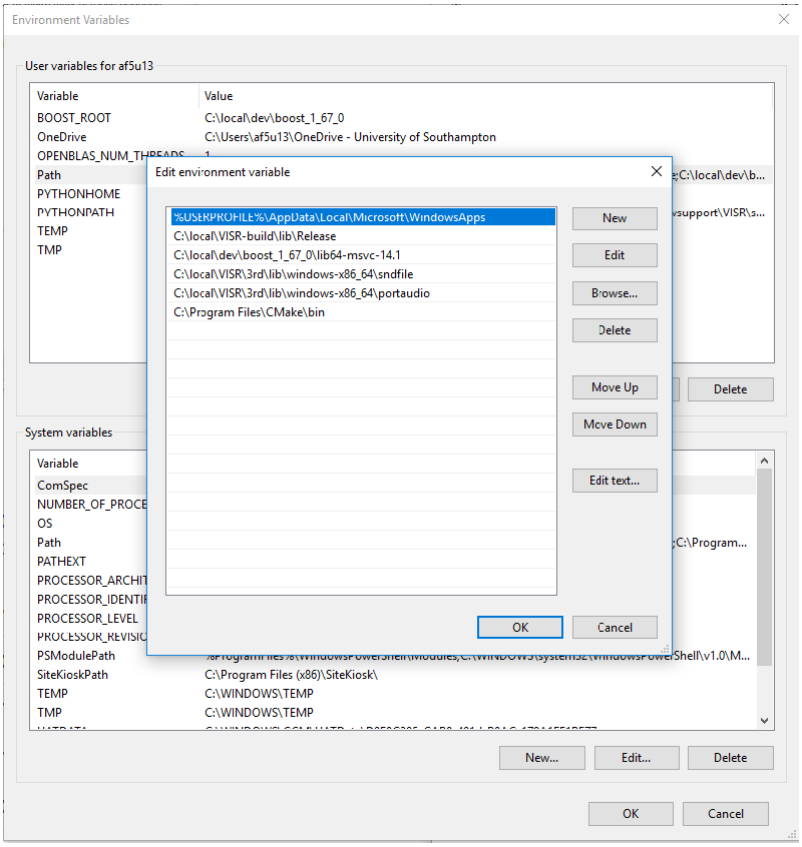

Fig. 2: Environment variable editor on Windows 10.

To use standalone applications (see section *Using standalone applications*), it may be useful to add the **bin/** directory to the user or system path. For the default installation location, add ‘’c:Program FilesVISR-N.N.Nbin” to the **%PATH%** environment variable.

5.2.2 Mac OS X

An installer with a graphical user interface guides through the installation process and allows the selection of optional components. Figure *Component-based installer for Mac OS X.* shows a screenshot of this installer. By default, it installs the VISR into the directory **/Applications/VISR-X.X.X/** where **X.X.X** denotes the version number.

To access the component selection dialog, use the button “Customize” on the “Installation Type” screen (see figure *“Installation type” screen of Mac OS X installer. Use “Customize” to get to the component selection.*)

To use the standalone applications from the command line, the **bin/** subfolder of the installation directory, e.g., **/Applications/VISR-X.X.X/bin**. This can be done, for example, by adding

```
export PATH=$PATH:/Applications/VISR-X.X.X/bin
```

to the file **$HOME/.bash_profile**. However, this works only for running standalone applications from a shell (i.e., a terminal window). If you need this path also from applications that are not started from a shell, we recommend the solution used in section *Configuration*.

5.2.3 Linux

For Linux, installation packages are provided as *.deb* (Debian) packages. At the moment, this package is monolithic, i.e., it contains all components. They are installed via the command

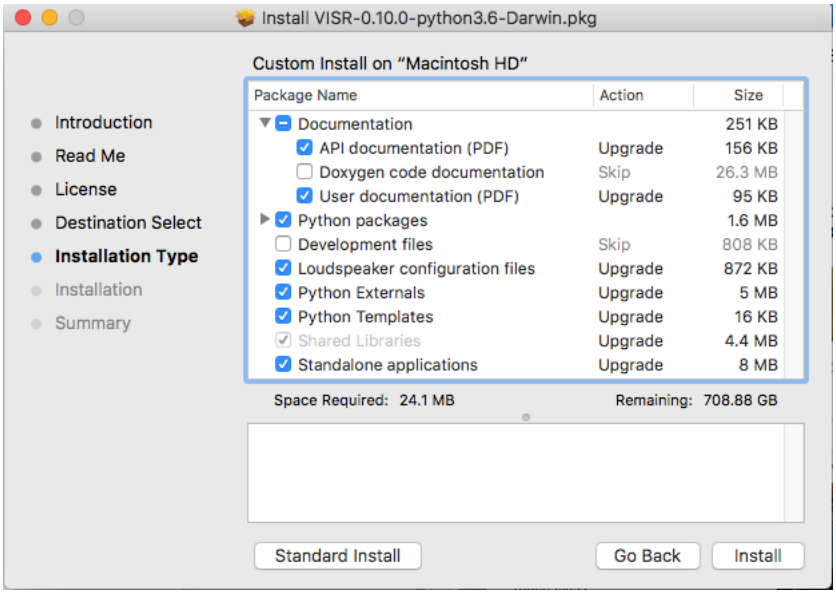

Fig. 3: Component-based installer for Mac OS X.

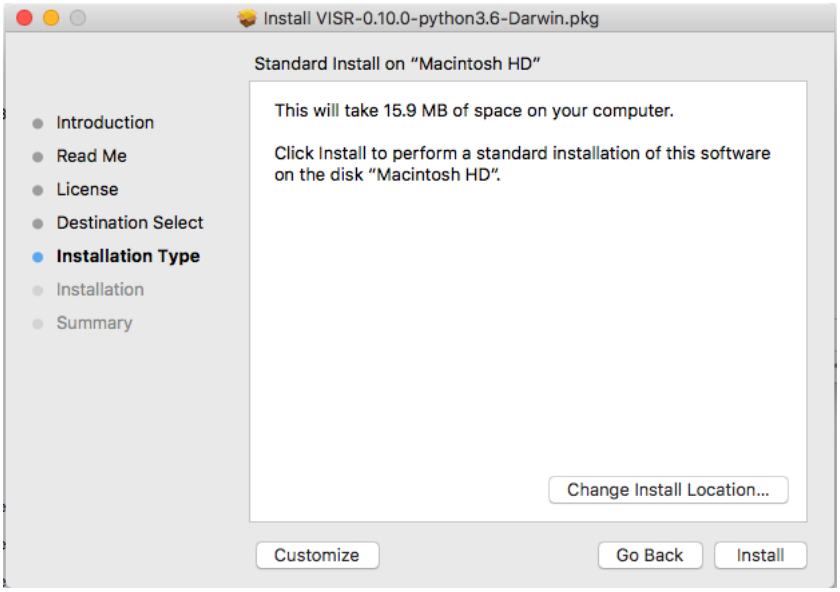

Fig. 4: "Installation type" screen of Mac OS X installer. Use "Customize" to get to the component selection.

```
sudo apt install VISR-<version>.deb
```

If this command reports missing dependencies, these can be installed subsequently with the command

```
sudo apt install --fix-broken
```

After that the framework is ready to use.

5.3 Installation components

With the dialog-based, component-enabled installers, parts of the framework can be chosen depending on the intended use of the framework.

Shared Libraries The core VISR libraries. This component is mandatory and cannot be unselected.

Standalone applications. Renderers and small tools to be run as command-line applications.

Python externals Python modules that give access to the functionality of the framework from Python. Also needed to run applications that use Python internally (e.g., the binaural synthesis toolkit or metadapter-enabled rendering).

Python Packages VISR extensions implemented in Python. This group of components requires the component "Python externals".

Development files Header files and CMake build support - Needed to extend the VISR with components using C++ or use the framework in external C++ applications.

Loudspeaker configurations A set of standard loudspeaker configuration files and additional example files from actual locations.

Python templates A set of commented template files for different types of VISR components.

Documentation User and code reference documentation as PDF documents. The Doxygen code documentation covering the complete source code can be optionally selected. However, the latter documentation is deprecated and will be contained in the code reference documentation in the future.

5.4 Setting up Python

As explained in section *Python integration*, the Python integration is an optional, albeit central, part of the VISR framework that enables a number of its functionalities, for example:

- Using the framework interactively from a Python interpreter.
- Using application that use Python internally, for instance the Binaural Synthesis Toolkit or metadata adaptation processes using the metadapter.
- Creating new signal flows or algorithms in Python.

To use these functionalities, a Python 3 distribution must be installed on the computer, and some configuration steps are required.

5.4.1 Python distribution

Depending on the system, we suggest different Python distributions:

Linux

Use the system-provided Python3 installation.

To install, use the package manager of your distribution, e.g.,

```
sudo apt install python3
```

Windows and Mac OS X

We recommend Anaconda. Please make sure you install the Python3 / 64-Bit variant.

Note: Some Mac OS variants (for example 10.12) come with a pre-installed Python 3 variant in **/Library/Frameworks/Python.framework**. In this case, care must be taken that it does not interferes with the chosen Python distribution. In particular, the **PYTHONHOME** environment variable must be set correctly.

5.4.2 Configuration

Two environment variables must be set to ensure the working of the VISR Python subsystem.

- **PYTHONPATH** This variable is used to add the directory containing the VISR python modules to the system path. To this end, the **python/** subdirectory of the installation folder must be added to **PYTHONPATH**.

  Note that other ways exist to add to the system path, for example

  ```
  import sys
  sys.path.append( '<visr_installation_dir>/python' )
  ```

  However, we recommend setting **PYTHONPATH** and assume this in the examples throughout this document.

PYTHOMHOME This variable is needed to locate the files and libraries of the Python distribution. This is especially important if there are more than one distributions on the system, most often on Mac OS X. Strictly speaking, this variable is required only if VISR Python code is executed from a C++ application, for instance some DAW plugins, **python_runner** standalone application (section ??), or the **visr_renderer** with metadata processing enabled. (see section *VISR object-based loudspeaker renderer*).

This variable has to be set to the root directory of the Python distribution, i.e., one level of hierarchy above the **bin/** folder conatining the Python interpreter. Depending on the platform and the distribution, the correct value might be:

Windows with Anaconda C:ProgramDataAnaconda3

Mac OS X with Anaconda $HOME/anaconda3/

Linux /usr

It is necessary to check whether these settings match with your directory layout.

If the Python distribution provides a **python-config** or **python3-config** binary, the command

```
python-config --prefix
```

or

```
python3-config --prefix
```

can be used to retrieve the required value for **PYTHONHOME** On Linux, setting **PYTHONHOME** is not necessary in most cases, because there is only the system-provided Python installation available.

Depending on the operating system, these variables can be set as follows:

OPENBLAS_NUM_THREADS It is advisable, in many cases, to set the value of this environment variable to 1. It controls how **numpy** numerical algebra functions are distributed to multiple CPU cores. **numpy** is used by the VISR Python integration as well as in many Python-based VISR components performing mathematical or DSP operations. For the matrix/vector sizes typically encountered in our code, the overhead for distributing the work over multiple cores typically exceeds the potential gains. Multithreading is disabled by setting the maximum number of cores (or threads) to 1:

```
OPENBLAS_NUM_THREADS = 1
```

This setting is optional. However, if you encounter excessive loads, for example a constant 100% load in the real-time thread, this setting can help to resolve the problem.

Linux Append the lines .. code-block:: bash

export PYTHONPATH=$PYTHONPATH:/usr/share/visr/python export OPENBLAS_NUM_THREADS=1

to **$HOME/.profile**.

Windows Add **PYTHONPATH** entries either as a user or system variable as described in *Windows* section. The corrects setting are (assuming the default installation directory and the Anaconda distribution):

PYTHONPATH=c:Program FilesVISR-X.X.Xpython PYTHONHOME=c:ProgramDataAnaconda3 OPENBLAS_NUM_THREADS=1

Note that if there is already a **PYTHONPATH** variable, the recommended value should be appended, using a semicolon as a separator.

Mac OS X In order to set the environment variables system-wide, without requiring that the applications in question is started from a shell, (e.g., a command-line terminal), we recommend a custom **launchd** property list file, as detailed, e.g., in this StackExchange thread.

To this end, create a **VISR-X.X.X.plist** file with this contents

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
↪DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>Label</key>
 <string>my.startup</string>
 <key>ProgramArguments</key>
 <array>
   <string>sh</string>
   <string>-c</string>
   <string>
   launchctl setenv PYTHONPATH /Applications/VISR-X.X.X/python
   launchctl setenv OPENBLAS_NUM_THREADS 1
   launchctl setenv PYTHONHOME ${HOME}/anaconda3
   </string>
 </array>
 <key>RunAtLoad</key>
 <true/>
</dict>
</plist>
```

By convention, these files are stored in **/Users/<loginname>/Library/LaunchAgents/**. To activate the settings, call

```
launchctl load <path-to-file>/VISR-X.X.X.plist
```

To take effect, all applications using these settings (e.g., terminals, Python interpreters, DAWs) must be quit and reopened.

These settings are preserved if the machine is restarted. To deactivate them, the property list file must be unloaded:

```
launchctl unload <path-to-file>/VISR-X.X.X.plist
```

If you made changes to the settings, you have to perform the **unload** command followed by a **load**.

For convenience, the installers create a pre-configured **VISR-X.X.X.plist** file in the **etc** subdirectory of the installation directory (e.g., **/Applications/VISR-X.X.X/etc/VISR-X.X.X.plist**). This file can be either loaded directly or copied to the **LaunchAgents/** directory first. Please check the values in this file first and adjust them accordingly.

5.5 Verifying the installation

We suggest some basic tests to verify that the VISR framework has been correctly installed and configured.

5.5.1 Testing a standalone application

This test is to ensure that that the installation is successful, and that the VISR shared libraries can be located and are compatible with the system. When using the component-enabled installers, the component `Standalone applications` must have been selected in order to perform this check.

In a terminal (Linux shell, Mac OS Terminal application, Windows command line **cmd**), execute this command:

```
<visr-installation-dir>/bin/matrix_convolver --version
```

For the different platforms, the full commands are (assuming the default installation directory) Windows

```
  "c:\Program Files\VISR-X.X.X\bin\matrix_convolver.exe" --version

Note that the quotes are necessary to cope with the space in the path.
```

Mac OS X

```
/Applications/VISR-X.X.X/bin/matrix_convolver --version
```

Linux

```
/usr/bin/matrix_convolver --version
```

If you added the **bin/** directory as described above, calling

```
matrix_convolver --version
```

is sufficient.

In any case, the call should generate a statement like

```
VISR Matrix convolver utility  0.10.0
```

If there is an error message about a missing shared library (or DLL), you should consult the respective section about installation. In particular this applies Windows, where the **PATH** variable needs to be set accordingly.

Testing the interactive Python integration

This test ensures that the VISR framework can be used interactively from Python interpreters.

First start a Python 3 interpreter (for example **python** or **ipython**). Depending on the system, the binaries might be called **python3** or **ipython3**, respectively It must be the interpreter of the Python distribution you intend to use (e.g., Anaconda).

In the interpreter, try to import the **visr** modules

```
import visr
```

This command should return without an error message. In this case, you can check whether the module is loaded from the correct location:

```
getattr( visr, '__file__' )
```

The directory of the resulting file path should be **<visr-installation-dir>/python**. For example, on Windows this returns **C:Program FilesVISR 0.10.0\python\visr.pyd**.

5.6 Source Code

Alternatively, the VISR framework can be installed and build from source code. It is hosted at the GitLab repository https://gitlab.eps.surrey.ac.uk:s3a/VISR.git

To retrieve the source code, clone the repository with

```
git clone https://gitlab.eps.surrey.ac.uk:s3a/VISR.git
```

Setting up a build environment, including the required software tools, and compiling the source code is detailed in the VISR API documentation.

5.7 Support and help

Sopport for installing and using the VISR is available through several ways.

First, you should check the FAQ section of the website (TODO: Insert link here)

Second, the mailing list (insert link to the registration page of the 3a-software list here).

Third, problems and supected bugs can be reported on (insert link to issues page of GitLab repository / later GitHub repo).

CHAPTER

SIX

VISR PRINCIPLES

CHAPTER

SEVEN

USING VISR

7.1 Using VISR standalone renderers

7.1.1 Using standalone applications

The VISR framework provides a number of standalone real-time rendering applications for some of its audio processing functionality.

If a component-aware installer is used (see Section *Installation components*), then the component "Standalone applications" has to be selected during installation.

The standalone applications are started as command line applications, and configured through a number of command line options or a configuration file.

Common options

All standalone applications provided with the VISR provide a common set of command line options:

–version or -v Returns a short description of the tool and its version information.

–help or -h Returns a list of supported command line options with brief descriptions.

–option-file <filename> or @<filename> Pass a configuration file containing a set of command line options to the applications. This options allows to store and share complex sets of command line options, and to overcome potential command line length limitations.

A typical option file has the format

```
-i 2
-o 2
-f 48000
-c "/usr/share/visr/config/generic/stereo.xml"
```

where,by convention, one option is stored per line.

–sampling-frequency or -f The sampling frequency to be used for rendering, as an integer value in Hz. Typically optional. If not given, a default value (e.g., 48000 Hz) will be used.

–period or -p The period, or blocksize, or buffersize to be used by the audio interface.

In most cases, the period should be a power of 2, e.g., 64, 128, 256, 512, ..., 4096. Lower values mean lower audio latency, but typically higher system load and higher susceptibility to audio underruns.

Typically an optional argument. If not given, a default value (e.g., 1024) is used.

–audio-backend or -D Specify the audio interface library to be used.

This option is mandatory.

The audio interfaces depend on the operating system and the configuration of the user's system. The most common options are **"PortAudio"** (all platforms) and **"Jack"** (Linux and Mac OS X). Note that additional

libraries (or backends) can be available for a specific platform, and new backends might be added in the future.

-audio-ifc-options A string to provide additional options to the audio interface.

This is an optional argument, and its content is interface-specific.

By convention, the existing audio interfaces expect JSON (JavaScript Object Notation) strings for the backend-configuration.

To pass JSON strings, the whole string should be enclosed in single or double quotes, and the quotes required by JSON must be escaped with a backslash. For example, the option might be used in this way:

```
visr_renderer ... -audio-ifc-options='{ \"hostapi\": \"WASAPI\" }'
```

Section *Interface-specific audio options* below explains the options for the currently supported audio interfaces.

-audio-ifc-option-file Provide a interface-specific option string within a file.

This can be used to avoid re-specifying complex options strings, to author them in a structured way, and to store and share them.

In addition, it avoids the quoting and escaping tricks needed on the command line. For example, the option shown above could be specified in a file **portaudio_options.cfg** as

```
{
  "hostapi": "WASAPI"
}
```

and passed as

```
visr_renderer ... -audio-ifc-option-file=portaudio_options.cfg
```

Note: The options **–audio-ifc-options** and **–audio-ifc-option-file** are mutually exclusive, that means other none or one of them can be provided.

VISR object-based loudspeaker renderer

These renderers facilitate object-based rendering to arbitrary loudspeaker setups. They use the VISR audio object model and the corresponding JSON format described in Section *Predefined object-based rendering primitives and renderers*.

Note that there are two binaries for loudspeaker rendering: **visr_renderer** and **baseline_renderer**. The provision of these separate binaries has technical reasons - mainly their dependency on a compatible and configured Python installation, as explained below.

The two binaries provided are:

visr_renderer This is the full object-based renderer, including a powerful metadata adaptation engine for intelligent object-based rendering - the Metadapter - implemented in Python. This metadapter is integrated into the rendering binary as an optional part, and is used if the option **–metadapter-config** is specified. The binary itself, however, needs a Python istallation to start at all, irrespective whether this option is set.

baseline_renderer This is the legacy object-based loudspeaker renderer. At the time being, it provides the same functionality as the **visr_renderer**, but without the optional integrated metadapter component. In this way, the binary is independent of a Python distribution on the user's computer.

In general, we recommend to use **visr_renderer** if possible, and to use **baseline_renderer** on systems where the Python features of the VISR framework are not available.

The command line arguments supported by the **visr_renderer** application are:

```
$> visr_renderer.exe --help
-h [ --help ]                    Show help and usage information.
-v [ --version ]                 Display version information.
--option-file arg                Load options from a file. Can also be used
                                 with syntax "@<filename>".
-D [ --audio-backend ] arg       The audio backend.
-f [ --sampling-frequency ] arg Sampling frequency [Hz]
-p [ --period ] arg              Period (blocklength) [Number of samples per
                                 audio block]
-c [ --array-config ] arg        Loudspeaker array configuration file
-i [ --input-channels ] arg      Number of input channels for audio object
                                 signal
-o [ --output-channels ] arg     Number of audio output channels
-e [ --object-eq-sections ] arg Number of eq (biquad) section processed for
                                 each object signal.
--reverb-config arg              JSON string to configure the object-based
                                 reverberation part, empty string (default) to
                                 disable reverb.
--tracking arg                   Enable adaptation of the panning using visual
                                 tracking. Accepts the position of the tracker
                                 in JSON format"{ "port": <UDP port number>,
                                 "position": {"x": <x in m>, "y": <y im m>,
                                 "z": <z in m> }, "rotation": { "rotX": rX,
                                 "rotY": rY, "rotZ": rZ } }" .
-r [ --scene-port ] arg          UDP port for receiving object metadata
-m [ --metadapter-config ] arg   Metadapter configuration file. Requires a
                                 build with Python support. If empty, no
                                 metadata adaptation is performed.
--low-frequency-panning          Activates frequency-dependent panning gains
                                 and normalisation
--audio-ifc-options arg          Audio interface optional configuration
--audio-ifc-option-file arg      Audio interface optional configuration file
```

The arguments for the **baseline_renderer** application are identical, except that the `--metadapter-config` option is not supported as explained above.

`--audio-backend` or `-D` The audio interface library to be used. See section :ref:using_standalone_renderers_common_options.

`--audio-ifc-options`: Audio-interface specific options, section :ref:using_standalone_renderers_common_options.

`--audio-ifc-option-file`: Audio-interface specific options, section :ref:using_standalone_renderers_common_options.

`--sampling-frequency` or `-f`: Sampling frequency in Hz. Default: 48000 Hz. See section :ref:using_standalone_renderers_common_options.

`--period` or `-p`: The number of samples processed in one iteration of the renderer. Should be a power of 2 (64,128,...,4096,...) . Default: 1024 samples. See section :ref:using_standalone_renderers_common_options.

`--array-config` or `-c`: File path to the loudspeaker configuration file. Path might be relative to the current working directory. Mandatory argument. The XML file format is described below in Section *Loudspeaker configuration file format*.

`--input-channels` or `-i`: The number of audio input channels. This corresponds to the number of single-waveform objects the renderer will process. Mandatory argument. A (case-insensitive) file extension of c .xml triggers the use of the XML format for parsing.

`--output-channels` or `-o`: The number of output channels the renderer will put write to. If not given, the number of output channels is determined from the largest logical channel number in the array configuration.

`--object-eq-sections`: The number of EQs (biquad sections) that can be specified for each object audio signal.

Default value: o, which deactivate EQ filtering for objects.

--low-frequency-panning: Switches the loudspeaker panning between standard VBAP and a dual-frequency approach with separate low- and high-frequency panning rules.

Admissible values are `true` and `false`. The default value is `false`, corresponding to the standard VBAP algorithm.

--reverb-config: A set of options for the integrated reverberation engine for the RSAO (`PointsourceWithReverb`) object (see section *Object-Based Reverberation*). To be passed as a JSON string. The supported options are:

numReverbObjects: The number of RSAO objects that can be rendered simultaneously. These objects may have arbitrary object ids, and they are automatically allocated to the computational resources avalable.

To be provided as a nonnegative integer number The default value is 0, which means that the reverberation rendering is effectively disabled.

lateReverbFilterLength: Specify the length of the late reverberation filters, in seconds.

Provided as a floating-point value, in seconds. Default value is zero, which results in the shortest reverb filter length that can be processed by the renderer, typically one sample.

lateReverbDecorrelationFilters: Specifies a multichannel WAV file containing a set of decorrelation filters, one per loudspeaker output. The number of channels must be equal or greater than the number of loudspeakers, channels that exceed the number of loudspeakers are not used.

To be provided as a full file path. The default value is empty, which means that zero-valued filters are used, which effectively disables the late reverb.

discreteReflectionsPerObject: The maximum number of discrete reflections that can be rendered for a single RSAO object.

Given as a nonnegative integer number. The default value is 0, which means that no discrete reflections are supported.

maxDiscreteReflectionDelay: The maximum discrete reflection delay supported. This allows a for tradeoff between the computational resources, i.e., memory required by the renderer and a realistic upper limit for discrete reflection delays.

To be provided as a floating-point number in seconds. Default value is 1.0, i.e., one second.

lateReverbFilterUpdatesPerPeriod Optional argument for limiting the number of filter updates in realtime rendering. This is to avoid processing load peaks, which might lead to audio underruns, if multiple RSAO objects are changed simultaneously. The argument specifies the maximum number of objects for whom the late reverb filter is calculated withon one period (audio buffer). If there are more pending changes than thix number, the updates are spread over multiple periods. This is a tradeoff between peak load and the timing accuracy and synchronity of late reverb updates.

Optional value, default value is 1, meaning at most one update per period

An example configuration is:

```
--reverb-config='{ \"numReverbObjects\": 5, \"lateReverbFilterLength\": 4.0,
          \"lateReverbDecorrelationFilters\": "/home/af5u13/tmp/decorr.wav\",
          \"discreteReflectionsPerObject\": 10 }'
```

--tracking Activates the listener-tracked VBAP reproduction, which adjust both the VBAP gains as well as the final loudspeaker gains and delays according to the listener position. It takes a non-empty string argument containing a JSON message of the format: `{ "port": <UDP port number>, "position": {"x": <x in m>, "y": <y im m>, "z": <z in m> }, "rotation": { "rotX": rX, "rotY": rY, "rotZ": rZ } }`. The values are defined as follows:

ID	Description	Unit	Default
port	UDP port number	unsigned int	8888
position.x	x position of the tracker	m	2.08
position.y	y position of the tracker	m	0.0
position.z	z position of the tracker	m	0.0
rotation.rotX	rotation the tracker about the x axis, i.e., y-z plane	degree	0.0
rotation.rotY	rotation the tracker about the y axis, i.e., z-x plane	degree	0.0
rotation.rotZ	rotation the tracker about the z axis, i.e., x-y plane	degree	180

Note: The option parsing for `--tracking` not supported yet, default values are used invariably. To activate tracking, you need to specify the `--tracking` option with an arbitrary parameter (even `--tracking=false` would activate the tracking.

`--scene-port` The UDP network port which receives the scene data in the VISR JSON object format.

`--metadapter-config` An optional Metadapter configuration file in XML format, provided as a full path to the file. If specified, the received metadata are passed through a sequence of metadata adaptation steps that are specified in the configuration file. If not given., metadata adaptation is not performed, and objects are directly passed to the audio renderer.

This option is not supported by the **baseline_renderer** application.

Loudspeaker configuration file format

The loudspeaker configuration has to be specified in an XML file. It is used primarily for the loudspeaker renderers.

An example is given below.

```
<panningConfiguration>
  <loudspeaker id="M+000" channel="1" eq="highpass">
    <cart x="1.0" y="0.0" z="0"/>
  </loudspeaker>
  <loudspeaker id="M-030" channel="2" eq="highpass">
    <polar az="-30.0" el="0.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="M+030" channel="3" eq="highpass">
    <polar az="30.0" el="0.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="M-110" channel="4" eq="highpass">
    <polar az="-110.0" el="0.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="M+110" channel="5" eq="highpass">
    <polar az="110.0" el="0.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="U-030" channel="6" eq="highpass">
    <polar az="-30.0" el="30.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="U+030" channel="7" eq="highpass">
    <polar az="30.0" el="30.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="U-110" channel="8" eq="highpass">
    <polar az="-110.0" el="30.0" r="1.0"/>
  </loudspeaker>
  <loudspeaker id="U+110" channel="9" eq="highpass">
    <polar az="110.0" el="30.0" r="1.0"/>
  </loudspeaker>
  <virtualspeaker id="VoS">
```

(continues on next page)

(continued from previous page)

```
    <polar az="0.0" el="-90.0" r="1.0"/>
    <route lspId="M+000" gainDB="-13.9794"/>
    <route lspId="M+030" gainDB="-13.9794"/>
    <route lspId="M-030" gainDB="-13.9794"/>
    <route lspId="M+110" gainDB="-13.9794"/>
    <route lspId="M-110" gainDB="-13.9794"/>
  </virtualspeaker>
  <triplet l1="VoS" l2="M+110" l3="M-110"/>
  <triplet l1="M-030" l2="VoS" l3="M-110"/>
  <triplet l1="M-030" l2="VoS" l3="M+000"/>
  <triplet l1="M-030" l2="U-030" l3="M+000"/>
  <triplet l1="M+030" l2="VoS" l3="M+000"/>
  <triplet l1="M+030" l2="VoS" l3="M+110"/>
  <triplet l1="U+030" l2="U-030" l3="M+000"/>
  <triplet l1="U+030" l2="M+030" l3="M+000"/>
  <triplet l1="U-110" l2="M-030" l3="U-030"/>
  <triplet l1="U-110" l2="M-030" l3="M-110"/>
  <triplet l1="U+110" l2="U-110" l3="M-110"/>
  <triplet l1="U+110" l2="M+110" l3="M-110"/>
  <triplet l1="U+030" l2="U-110" l3="U-030"/>
  <triplet l1="U+030" l2="U+110" l3="U-110"/>
  <triplet l1="U+030" l2="U+110" l3="M+110"/>
  <triplet l1="U+030" l2="M+030" l3="M+110"/>
  <subwoofer assignedLoudspeakers="M+000, M-030, M+030, M-110, M+110, U-030, U+030,
↪ U-110, U+110"
          channel="10" delay="0" eq="lowpass" gainDB="0"
          weights="1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0"
          />
  <outputEqConfiguration numberOfBiquads="1" type="iir">
      <filterSpec name="lowpass">
         <biquad a1="-1.9688283" a2="0.96907117" b0="6.0729856e-05" b1="0.
↪00012145971" b2="6.0729856e-05"/>
      </filterSpec>
      <filterSpec name="highpass">
         <biquad a1="-1.9688283" a2="0.96907117" b0="-0.98447486" b1="1.9689497"␣
↪b2="-0.98447486"/>
      </filterSpec>
  </outputEqConfiguration>
</panningConfiguration>
```

Format description

The root node of the XML file is `<panningConfiguration>`. This root element supports the folloring optional attributes: * `isInfinite` Whether the loudspeakers are regardes as point sources located on the unit sphere (`false`) or as plane waves, corresponding to an infinite distance (`true`). The default value is `false`. * `dimension` Whether the setup is considered as a 2-dimensional configuration (value `2`) or as three-dimensional (`3`, thedefault). In the 2D case, the array is considered in the x-y plane , and the `z` or `el` attributes of the loudspeaker positions are not evaluated. In this case, the triplet specifications consist of two indices only (technically they are pairs, not triplets).

Within the `<panningConfiguration>` root element, the following elements are supported:

`<loudspeaker>` Represents a reproduction loudspeaker. The position is encoded either in a `<cart>` node representing the cartesian coordinates in the `x`, `y` and `z` attributes (floating point values in meter), or a `<polar>` node with the attributes `az` and `el` (azimuth and elevation, both in degree) and `r` (radius, in meter).

The `<loudspeaker>` nodes allows for number of attributes:

- `id` A mandatory, non-empty string identification for the loudspeaker, which must be unique across all `<loudspeaker>` and `<virtualspeaker>` (see below) elements. Permitted are alpha-numeric

characters, numbers, and the characters "@&()+/:_-". ID strings are case-sensitive.

- `channel` The output channel number (sound card channel) for this loudspeaker. Logical channel indices start from 1. Each channel must be assigned at most once over the set of all loudspeaker and subwoofers of the setup..
- `gainDB` or `gain` Additional gain adjustment for this loudspeaker, either in linear scale or in dB (floating-point values. The default value is 1.0 or 0 dB. `gainDB` or `gain` are mutually exclusive.
- `delay` Delay adjustment to be applied to this loudspeaker as a floating-point value in seconds. The default value is 0.0).
- `eq` An optional output equalisation filter to be applied for this loudspeaker. Specified as a non-empty string that needs to match an `filterSpec` element in the `outputEqConfiguration` element (see below). If not given, no EQ is applied to for this loudspeaker.

<virtualspeaker> An additional vertex added to the triangulation that does not correspond to a physical loudspeaker. Consist of a numerical `id` attribute and a position specified either as a `<cart>` or a `<polar>` node (see `<loudspeaker>` specification).

The `<virtualspeaker>` node provides the following configuration options:

- A mandatory, nonempty and unique attribute `id` that follows the same rules as for the `<loudspeaker>` elements.
- A number of `route` sub-elements that specify how the energy from this virtual loudspeaker is routed to real loudspeakers. The `route` element has the following attributes: * `lspId`: The ID of an existing real loudspeaker. * `gainDB`: A scaling factor with which the gain of the virtual loudspeaker is distributed to the real loudspeaker.

  In the above example, the routing specification is given by

```
 <virtualspeaker id="VoS">
  <polar az="0.0" el="-90.0" r="1.0"/>
  <route lspId="M+000" gainDB="-13.9794"/>
  <route lspId="M+030" gainDB="-13.9794"/>
  <route lspId="M-030" gainDB="-13.9794"/>
  <route lspId="M+110" gainDB="-13.9794"/>
  <route lspId="M-110" gainDB="-13.9794"/>
</virtualspeaker>
```

  That means that the energy of the virtual speaker `"vos"` is routed to five surrounding speakers, with a scaling factor of 13.97 dB each.

`<subwoofer>` Specify a subwoofer channel. In the current implementation, the loudspeaker are weighted and mixed into an arbitray number of subwoofer channels. The attributes are:

- `assignedLoudspeakers` The loudspeaker signals (given as a sequence of logical loudspeaker IDs) that contribute to the subwoofer signals. Given as comma-separated list of loudspeaker index or loudspeaker ranges. Index sequences are similar to Matlab array definitions, except that thes commas separating the parts of the sequence are compulsory.

  Complex example:

```
assignedLoudspeakers = "1, 3,4,5:7, 2, 8:-3:1"
```

- `weights` Optional weights (linear scale) that scale the contributions of the assigned speakers to the subwoofer signal. Given as a sequence of comma-separated linear-scale gain values, Matlab ranges are also allowed. The number of elements must match the `assignedLoudspeakers` index list. Optional value, the default option assigns 1.0 for all assigned loudspeakers. Example: "0:0.2:1.0, 1, 1, 1:-0.2:0".
- `gainDB` or `gain` Additional gain adjustment for this subwoofer, either in linear scale or in dB (floating-point valus, default 1.0 / 0 dB). Applied on top of the `weight` attributes to the summed subwoofer signal. See the `<loudspeaker>` specification.

- `delay` Delay adjustment for this (floating-point value in seconds, default 0.0). See the `<loudspeaker>` specification.

`<triplet>` Loudspeaker triplet specified by the attributes `l1`, `l2`, and `l3`. The values of `l1`, `l2`, and `l3` must correspond to IDs of existing real or virtual loudspeakers. In case of a 2D setup, only `l1` and `l2` are evaluated.

`outputEqConfiguration` This optional element must occur at most once. It provides a global specification for equalisation filters for loudspeakers and subwoofers.

```
<outputEqConfiguration  type="iir" numberOfBiquads="1">
  <filterSpec name="lowpass">
    <biquad a1="-1.9688283" a2="0.96907117" b0="6.0729856e-05" b1="0.
↪00012145971" b2="6.0729856e-05"/>
  </filterSpec>
  <filterSpec name="highpass">
    <biquad a1="-1.9688283" a2="0.96907117" b0="-0.98447486" b1="1.9689497" b2=
↪"-0.98447486"/>
  </filterSpec>
</outputEqConfiguration>
```

The attributes are:

- `type`: The type of the output filters. At the moment, only IIR filters provide as second-order sections (biquads) are supported. Thus, the value `"iir"` must be set.
- `numberOfBiquads`: This value is specific to the `"iir"` filter type.

The filters are described in `filterSpec` elements. These are identifed by a `name` attribute, which must be an non-empty string unique across all `filterSpec` elements. For the type `iir`, a `filterSpec` element consists of at most `numberOfBiquad` nodes of type `biquad`, which represent the coefficients of one second-order IIR (biquad) section. This is done through the attributes `a1`, `a2`, `b0`, `b1`, `b2` that represent the coefficients of the normalised transfer function

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

Interface-specific audio options

This section described the audio-interface-specific options that can be passed through the `--audio-ifc-options` or `--audio-ifc-option-file` arguments.

PortAudio interface

The interface-specific options for the PortAudio interface are to be provided as a JSON file, for example:

```
{
 "sampleformat": "...",
 "interleaved": "...",
 "hostapi" : "..."
}
```

Note: When used on the command line using the `--audio-ifc-options` argument, apply the quotation and escaping as described in Section *Common options*.

The following options are supported for the PortAudio interface:

sampleformat Specifies the PortAudio sample format. Possible values are:

- `signedInt8Bit`

- `unsignedInt8Bit`
- `signedInt16Bit`
- `unsignedInt16Bit`
- `signedInt24Bit`
- `unsignedInt24Bit`
- `signedInt32Bit`
- `unsignedInt32Bit`
- `float32Bit`.

interleaved: Enable/disable interleaved mode, possible values are `true, false`.

hostapi: Used to specify PortAudio backend audio interface. Possible values are:

- `default`: This activates the default backend
- `WASAPI` : Supported OS: Windows.
- `ASIO` : Supported OS: Windows.
- `WDMKS`: Supported OS: Windows.
- `DirectSound` : Supported OS: Windows.
- `CoreAudio` : Supported OS: MacOs.
- `ALSA` : Supported OS: Linux.
- `JACK` : Supported OSs: MacOs, Linux.

PortAudio aupports a number of other APIs. However, they are outdated or refer to obsolete platforms and therefore should not be used: - `SoundManager` (MacOs) - `OSS` (Linux) - `AL` - `BeOS` - `AudioScienceHPI` (Linux)

This configuration is an example of usage of PortAudio, with Jack audio interface as backend.

```
{
  "sampleformat": "float32Bit",
  "interleaved": "false",
  "hostapi" : "JACK"
}
```

Jack audio interface

The following options can be provided when using Jack as our top level component's Audio Interface:

`clientname`: Jack Client name for our top level component.

`servername`: Jack Server name. If not provided, the default Jack server is used.

`autoconnect`: Globally enable/disable the automatic connection of ports. Admissible values are `true` and `false`. This setting can be overridden specifically for capture and playback ports in the port configuration section described below.

`portconfig`: Subset of options regarding the configuration and connection of Jack Ports, see following section.

Port Configuration

The port configuration section allows to individually set properties for the capture, i.e., input, and the playback, i.e., output, ports of an application.

`capture`: Specifies that the following options regard the top level component's capture ports only

- `autoconnect` : Enable/disable auto connection to an external jack client's input ports, possible values are `true, false`
- `port`: Jack ports specification
  - `basename`: Common name for all top level component's capture ports
  - `indices`: list of port numbers to append to top level component's capture port name. It is possible to use Matlab's colon operator to express a list of numbers in a compact fashion (es."0:4" means appending numbers 0 to 3 to port names)
  - **`externalport`: Specification of an external jack client to connect to if `autoconnect` is enabled.**
    - `client`: Name of an external jack client to use as input for our top level component (es. "system")
    - `portname`: Common name for all external jack client input ports
    - `indices`: List of port numbers that together with :code:' portname' describe existing external jack client input ports. It is possible to use Matlab's colon operator to express a list of numbers.

`playback`: Specifies that the following options regard the top level component's playback ports only.

- `autoconnect` : Enable/disable auto connection to an external jack client's output ports, possible values are `true, false`
- **`port`: Jack ports specification**
  - `basename`: Common name for all top level component's playback ports
  - `indices`: list of port numbers to append to top level component's playback port name. It is possible to use Matlab's colon operator to express a list of numbers in a compact fashion (es."0:4" means appending numbers 0 to 4 to port names)
  - **`externalport`: Specification of an external jack client to connect to if `autoconnect` is enabled.**
    - `client`: Name of an external jack client to use as output for our top level component (es. "system")
    - `portname`: Common name for all external jack client output ports
    - `indices`: List of port numbers that together with :code:' portname' describe existing external jack client output ports. It is possible to use Matlab's colon operator to express a list of numbers.

Simple Example

This configuration example shows how to auto-connect the Jack input and output ports of an application to the default jack client (`system`), specifying which range of ports to connect.

```
{
  "clientname": "BaseRenderer",
  "autoconnect" : "true",
  "portconfig":
  {
    "capture":
    {
      "port":
      [{ "externalport" : {"indices": "1:4"} }]
    },
    "playback":
    {
      "port":
```

(continues on next page)

(continued from previous page)

```
      [{ "externalport" : {"indices": "5:8"} }]
    }
  }
}
```

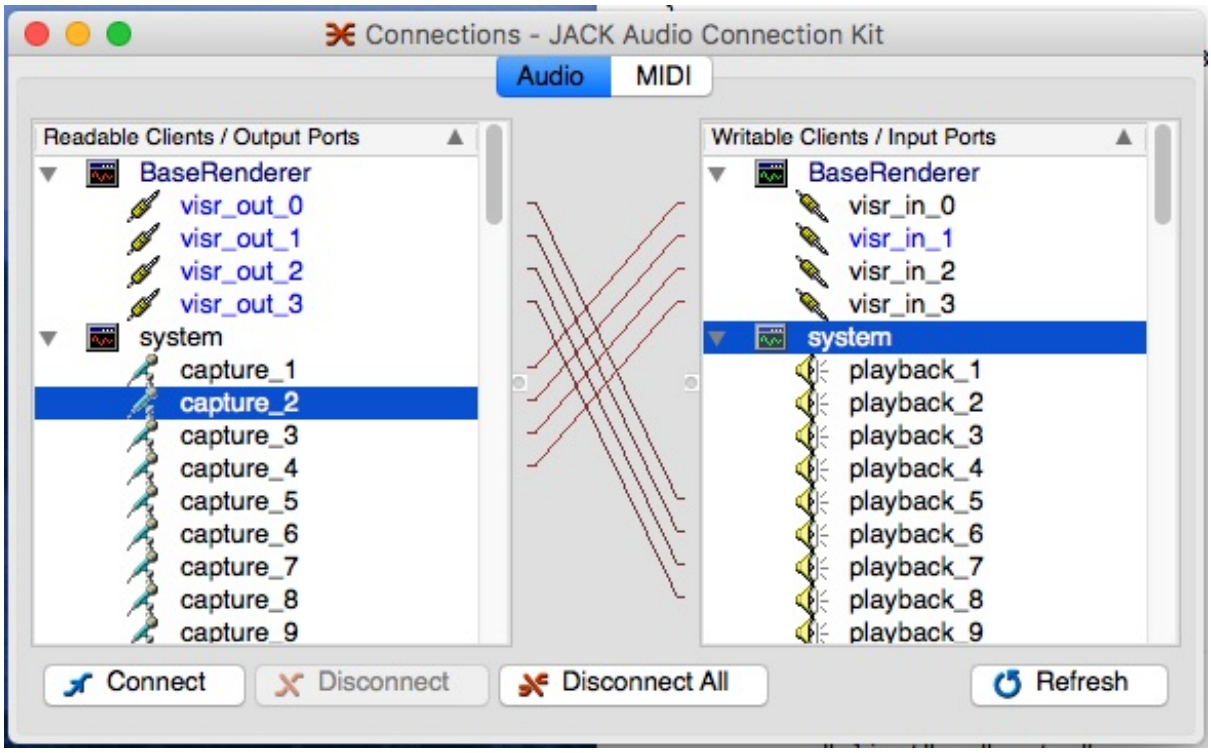

Fig. 1: Jack audio complex configuration example.

Complex Example

Follow a more complex example where auto-connection of ports is performed specifying different jack clients and the ranges of ports to be connected are described both for the top level component and for external clients.

```
{
  "clientname": "VisrRenderer",
  "servername": "",
  "autoconnect" : "true",
  "portconfig":
  {
    "capture":
    {
      "autoconnect" : "true",
      "port":
      [
        {
         "basename" : "Baseinput_" ,
         "indices": "0:1",
         "externalport" :
          {
            "client" : "REAPER",
            "portname": "out",
            "indices": "1:2"
          }
```

(continues on next page)

(continued from previous page)

```
        },
        {
         "basename" : "Baseinput_" ,
         "indices": "2:3",
         "externalport" :
          {
            "indices": "4:5"
          }
         }
      ]
    },
    "playback":
    {
      "autoconnect" : "true",
      "port":
      [{
        "basename" : "Baseoutput_" ,
        "indices": "0:1",
        "externalport" :
         {
          "client" : "system",
          "portname": "playback_",
          "indices": "4:5"
         }
       }]
    }
  }
}
```

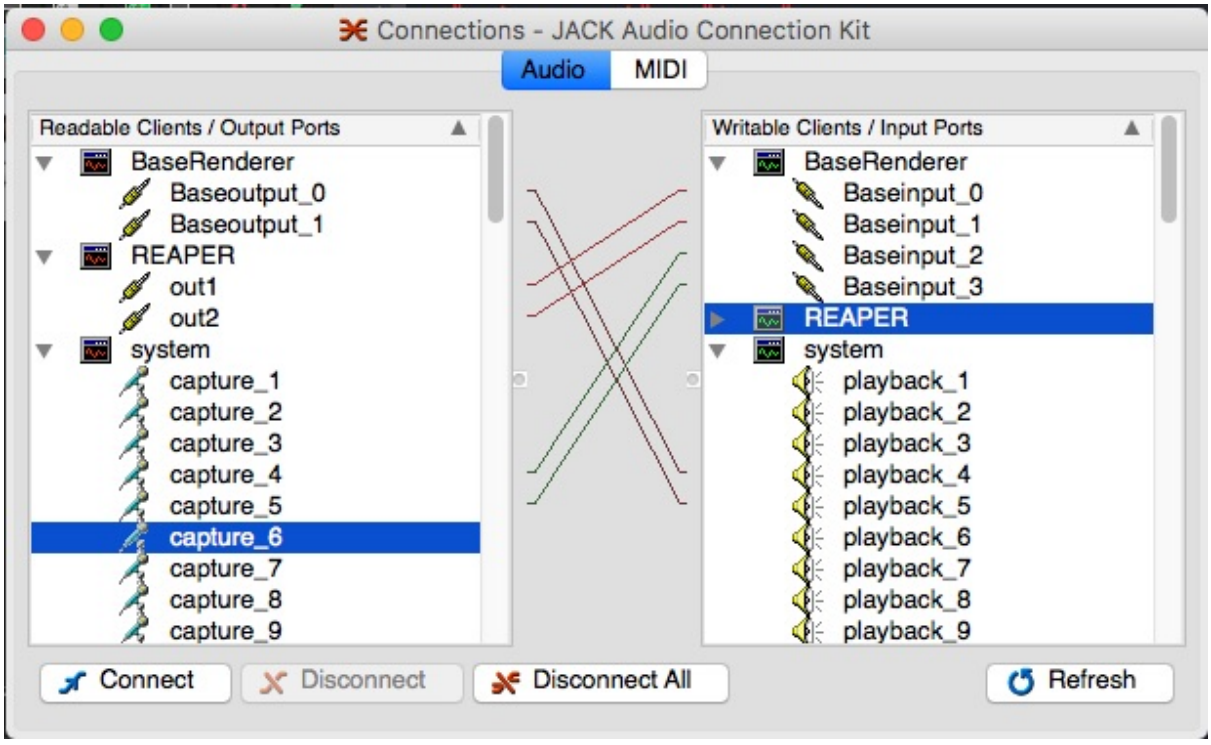

Fig. 2: Jack audio complex configuration example.

7.2 Using VISR with Python

7.3 Using VISR audio workstation plugins

7.4 Using Max/MSP externals

CHAPTER

EIGHT

EXTENDING VISR

In this part we describe how to use the VISR framework to implement new functionality, i.e., functionality that is not contained in the existing components or standalone renderers. This part is basically an extended version of the tutorial presented in

8.1 Creating signal flows from existing components in Python

8.2 Writing atomic functionality in Python

8.3 Implementing atomic components in C++

8.4 Creating composite components in C++

CHAPTER

NINE

OBJECT-BASED AUDIO WITH VISR

9.1 Overview

Although the VISR framework is deliberately application-agnostic, it is well-suited for working with spatial and object-based audio.

9.2 The VISR object model

9.3 Predefined object-based rendering primitives and renderers

9.4 Object-Based Reverberation

CHAPTER

TEN

VISR COMPONENT REFERENCE

CHAPTER

ELEVEN

OLD CONTENTS

11.1 Examples

11.2 Tutorials

The contents of these files will be removed or moded to other parts of the documentation.