VISR Code documentation

Release

The S3A team

October 30, 2018

CHAPTER

ONE

ABOUT

About VISR

About this document

This is the code reference document for the VISR (Versatile Interactive Scene Renderer) software framework. It describes the software architecture and modules, and provides the API documentation for the C++ and Python interfaces. In addition, it details the software development process, including obtaining and compiling the source code.

CHAPTER

TWO

DEVELOPMENT AND CONTRIBUTION

Source Code

Depending on the operating system you have, the following startup guides will be useful for creating a successful build and/or using VISR for your own development. In a new terminal window, run the following:

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

osx

The choice of IDE for MacOS X is Xcode. You can see the full guide on developing with visr here at the .. .. MacOS Setup Guide.

linux

To develop and contribute using VISR with a Linux-based IDE, have a look at the .. .. Linux Setup Guide.

windows

To develop and contribute using VISR with Windows and Visual Studio, have a look at the .. .. Windows Setup Guide.

Raspberry Pi

VISR Git Workflow

This page documents the version control workflow followed by the VISR community when dealing with the development of VISR. If you want to contribute, and/or already have code or bug fixes you would like to see in the official repo, please follow these guidelines to make the life of the devs easier, minimize time needed for review, and ensure speedy and efficient incorporation of your improvements into VISR.

VISR Git Workflow

The git workflow for openFrameworks is based on this article detailing a very good branching model. Read the article first, it's quite complete and nicely presented, no need to replicate everything here.

https://nvie.com/posts/a-successful-git-branching-model/

Continuous Integration

All contributions that are merged together in the remote are tested before they are pushed. This is to ensure what you contribute works and is tested on all operating systems. Sometimes, before you push, you might forget to compile or test you code (we all make these mistakes sometimes), so the cvssp-servers at Surrey University will run certain jobs to ensure that your code compiles and is built correcntly.

Every time a merge request from your feature/name-of-your-feature to the remote/origin/develop occurs, a pipeline is triggered testing what you have done before it is merged. You should receive an email if it does not.

On a successful pipeline, other work is done to automatically update VISR' WebAPI. Make sure you comment what you have written!

Setting up a build environment

Standard build setup

Git

- On Linux git can be installed through the distribution's package manager. On Ubuntu, Raspbian, and Debian-based distribtution, the command is `sudo apt install git`.
- On Mac OS X 10.9 and above, the git command line tool is installed on its first invocation (source: ‘GIT SCM book <)
- On Windows, we suggest the Git for Windows binaries.

Alternatively you can choose from a number of GUIs, for example <SourceTree <https://www.sourcetreeapp.com/>‘_ or GitKraken, or the support in modern IDEs as Microsoft Visual Studio or Apple XCode.

CMake

VISR uses CMake as a portable build tool The minimum required version is CMake 3.1.

CMake can be used as a command line tool but also provides a GUI (cmake-gui) for configuring builds.

- On Linux, install throught he distribution's package manager. On Debian-based systems, the packages are named `cmake` for the command line tools and `cmake-qt-gui` for the graphical user interface.
- On Windows and Mac OS X, download
- If you plan to use the machine for CI (continuous integration tasks), make sure you add it to the system path (Windows: Select “Add CMake to the system PATH for all users”)

Documentation

The VISR documentation is mainly written as ReStructured text documents (using **‘Sphinx <>‘_** to create web pages and PDF documents), whereas the code documentation pulled into these documents is generated through Doxygen.

In order to create the user and API documentation, the following software tools must be installed:

- Doxygen: On Windows and Mac OS X, we recommend downloading binary packages from the Doxygen download page and installing them. On Linux, installing via the distribution-specific package manager (e.g., ‘‘sudo apt install doxygen’‘. Note that the XML output generation needed for subsequent build stages is quite buggy in older Doxygen releases, it might therefore be advisable to upgrade to a recent Doxygen release or to build from source (especially on Linux, where the version provided by the package manager could be rather dated).

- LaTeX: Both the Doxygen documentation and the PDF generation of Sphinx require a LaTeX system to be installed. We recommend the following LaTeX distributions:
  - **Linux: The system-provided LaTeX (installed, e.g., through ‘‘sudo apt install texlive-full” ‘‘sudo apt install texliv**
    - Mac OS X: **‘MacTeX <http://www.tug.org/mactex/‘_**
    - Windows: **‘MikTeX <http://www.miktex.org‘_**

  Note that - unless you install a “full” distribution - LaTeX packages might be missing and would need to be installed as needed. So look out for error messages when running the documentation generation for the first time, and use the distribution’s package manager to install the missing packages as needed.

  After installing LaTeX and running the CMake configuration stage with the option BUILD_DOCUMENTATION ena

  - Windows: Set the CMake variable ‘‘MIKTEX_BINARY_PATH” to the directory containing the MikTeX binaries, e.g., ‘‘C:/Program Files/MiKTeX 2.9/miktex/bin/x64’‘
  - Mac OS X: Set `PDFLATEX_COMPILER` to the result of the shell command `which pdflatex’’. On a recent MacTeX installation, this yields ‘‘/Library/TeX/texbin\pdflatex/`
- Sphinx: Sphinx is typically bundled with the Python distribution you are using and should therefore be installed through the package management system of this distribution.
  - **Linux: use the package manager of the Linux distribution, e.g., `sudo apt install python3-sphinx`.**
    - Anaconda (recommended distribution on Windows and Mac OS X): use `conda install sphinx`.

  Alternatively, the generic Python package manager `pip` can be used.
- **Breathe: Breathe is needed by sphinx to render source code generation for languages other than Python. As Breathe i**
  - **Linux: use the package manager of the Linux distribution, e.g., `sudo apt install python3-breathe`**
    - Anaconda (recommended distribution on Windows and Mac OS X): use `conda install -c conda-forge breathe`. Note that breathe is not contained in the standard package repository but the additional repository conda-forge.

Installer package generation

Installation packages are created using CPack, which is integrated into the CMake build software. On Linux and Mac OS X, no external build software is needed.

Windows

Download and install the NSIS installer system (**‘https://sourceforge.net/projects/nsis/‘_**). It will be automatically recognized if you create the installer packages.

Setup as Gitlab runner

To provide a

CHAPTER

THREE

LIBRARIES

VISR API

Architecture

API reference

Class Component

class `visr::Component`

Base class for processing components. Components may contain ports to exchange data (either audio signal or parameter) with other components or with the exterior. A component may have a parent, that is, a composite component it is contained in. If the parent is null it is a top-level component. Components also have a name, which must be unique within a containing composite component.

Public Functions

Component(SignalFlowContext **const** &*context*, char **const** **componentName*, *CompositeComponent* **parent*)

Constructor, constructs a component.

Parameters

- `context` - Configuration object containing basic execution parameters (such as sampling frequency and period (block length))
- `componentName` - The name of the component. If this component is contained in a higher-level parent component, the name must be unique within that parent component
- `parent` - Pointer to the containing composite component, if there is one. Otherwise, that is, if the present component is at the top level, pass `nullptr`.

Component(SignalFlowContext **const** &*context*, std::string **const** &*componentName*, *CompositeComponent* **parent*)

Constructor. Convenvience function, accepts a standard string instead of a C chararacter pointer.

Parameters

- `context` - Configuration object containing basic execution parameters (such as sampling frequency and period (block length))
- `componentName` - The name of the component. If this component is contained in a higher-level parent component, the name must be unique within that parent component.
- `parent` - Pointer to the containing composite component, if there is one. Otherwise, that is, if the present component is at the top level, pass `nullptr`.

Component(*Component* **const**&)

Deleted copy constructor to avoid copy construction of this and derived classes.

Component(*Component*&&)
Deleted move constructor to avoid moving of this and derived classes.

Component &**operator=**(*Component* **const**&)
Deleted assignment operator to prohibit (copy) assignment of this and derived classes.

Component &**operator=**(*Component*&&)
Deleted assignment operator to prohibit move assignment of this and derived classes.

virtual **~Component**()
Destructor (virtual)

std::string **const** &**name**() const
Return the 'local', non-hierarchical name.

std::string **fullName**() const
Return the full, hierarchical name of the component.

void **status**(StatusMessage::Kind *status*, char **const** **message*)
Signal informational messages or the error conditions. Depending on the value of the `status` parameter, this might result in a message conveyed to the user or abortion of the audio processing.

Parameters

- `status` - The class of the status message
- `message` - An informational message string.

template <typename... *MessageArgs*>
void **status**(StatusMessage::Kind *status*, MessageArgs... *args*)
Signal informational messages or the error conditions where the message string is constructed from an arbitrary sequence of arguments. Depending on the value of the `status` parameter, this might result in a message conveyed to the user or abortion of the audio processing.

Template Parameters

- `MessageArgs` - List of argument types to be printed. Normally they are automatically determined by the compiler, so there is no need to specify them.

Parameters

- `status` - The class of the status message
- `args` - Comma-seprated list of parameters with unspecified types. The main requirement is that all types support an "<<" operator.

bool **isComposite**() const
Query whether this component is atomic (i.e., a piece of code implementing a rendering functionality) or a composite consisting of an interconnection of atomic (or further composite) components.

AudioPortBase &**audioPort**(char **const** **portName*)

AudioPortBase **const** &**audioPort**(char **const** **portName*) const

AudioPortBase &**audioPort**(std::string **const** &*portName*)

AudioPortBase **const** &**audioPort**(std::string **const** &*portName*) const

ParameterPortBase &**parameterPort**(char **const** **portName*)

ParameterPortBase **const** &**parameterPort**(char **const** **portName*) const

ParameterPortBase &**parameterPort**(std::string **const** &*portName*)

ParameterPortBase **const** &**parameterPort**(std::string **const** &*portName*) const

SamplingFrequencyType **samplingFrequency**() const
Return the sampling frequency of the containing signal flow.

std::size_t **period**() const
Return the period of the containing signal processing graph, i.e., the number of samples processed in each invocation of the process function of the derived audio components. This methods can be called at any point of the lifetime of the derived component, i.e., for instance in the constructor.

bool **isTopLevel**() const
Query whether the component is at the top level of a signal flow.

Note

Not needed for user API

impl::ComponentImplementation &**implementation**()
Provide a pointer to an external implementation object. The type of this implementation object is opaque, i.e., not visible from the public VISR API.

Note

This method is not supposed to be called in user code. It is public because it is is used by the VISR runtime system.

impl::ComponentImplementation **const** &**implementation**() const
Provide a pointer to an external implementation object, constant version. The type of this implementation object is opaque, i.e., not visible from the public VISR API.

Note

This method is not supposed to be called in user code. It is public because it is is used by the VISR runtime system.

Public Static Functions

std::string **const** &**nameSeparator**()
Separator used to form hierarchical names.

Protected Functions

Component(impl::ComponentImplementation *_impl_)
Constructor that receives the internal implementation object. This overload has to be called by the other constructors (including those of subclasses) to make sure that the implementation object is instantiated. The motivation for this constructor is to provide different implementation objects for different subclasses.

Class AtomicComponent

class visr::**AtomicComponent**
Base class for atomic components. These components are at the lowest level in the hierarchy and implement runtime functionality as C++ code. Abstract base class, derived classes must override the virtual method *process()*.

Public Functions

AtomicComponent(SignalFlowContext **const** &_context_, char **const** *_name_, *CompositeComponent* *_parent_)
Constructor.

Parameters

- context - a signal flow context structure containing general parameters, e.g., sampling rate and block size of computation.

- `name` - Null-terminated character string containing the name. Name must be unique within the containing composite component (if there is one).
- `parent` - A composite component to contain this atom, If it is a null pointer (the default), then this component is at the top level.

AtomicComponent (*AtomicComponent* **const**&)
Deleted copy constructor to avoid copying.

AtomicComponent (*AtomicComponent*&&)
Deleted move constructor to avoid move construction.

virtual **~AtomicComponent** ()
Destructor (virtual). Atomic components are destined to be instantiated and managed polymorphically, thus requiring virtual destructors.

virtual void **process** () = 0
Pure virtual *process()* function. The overriding methods of base classes are called in regular intervals, each processing a fixed number (`context.period()`) number of samples.

Class CompositeComponent

class `visr::CompositeComponent`
Base class for processing components that are composed of other components (atomic and composite). In this way, processing components can be structured hierarchically. Composite components store the contained sub-components, external audio and parameter ports, and conenctions between the ports.

Unnamed Group

typedef using visr::CompositeComponent::ChannelRange = visr::ChannelRange
Making the types for defining audio connections known inside CompositeConponent and derived classes These are convenience aliases to make the syntax in derived signal flows more concise.

Note

: This also means that we include the ChannelList/ChannelRange definition in this header, as these classes become part the *CompositeComponent* interface.

typedef using visr::CompositeComponent::ChannelList = visr::ChannelList

Public Functions

CompositeComponent (SignalFlowContext **const** &*context*, char **const** **name*, *CompositeComponent* **parent*)
Constructor.

Parameters

- `context` - Reference to a signal flow context object providing basic runtime parameters as period length or sampling frequency.
- `name` - "the name of the component. Used to address the component inside other components and for status reporting.
- `parent` - Reference (pointer) to a parent component if the present object is part of a containing signal flow. If `nullptr` is passed, this component is the top level.

~CompositeComponent ()
Destructor.

std::size_t **numberOfComponents** () const
The number of contained components (not including the composite itself). This method considers only atomic and composite components at the next level, i.e., not recursively.

impl::CompositeComponentImplementation &**implementation**()
Return a reference to the internal data structures holding ports and contained components. From the user point of view, these data structure is opaque and unknown.

impl::CompositeComponentImplementation **const** &**implementation**() const
Return a reference to the internal data structures holding ports and contained components, const version. From the user point of view, these data structure is opaque and unknown.

void **parameterConnection**(char **const** **sendComponent*, char **const** **sendPort*, char **const** **receiveComponent*, char **const** **receivePort*)
Register a connection between parameter ports (both real ports of contained components or external placeholder ports).

Parameters

- `sendComponent` - The name of the component holding the send port (local name, not the fully qualified name). When specifiying an external port of a composite component, use an empty string or `"this"`.
- `sendPort` - The local (not fully qualified) name of the send port.
- `receiveComponent` - The name of the component holding the receive port (local name, not the fully qualified name). When specifiying an external port of a composite component, use an empty string or `"this"`.
- `receivePort` - The local (not fully qualified) name of the receive port.

Exceptions

- `std::invalid_argument` - if a specified component or port does not exist.

void **parameterConnection**(ParameterPortBase &*sender*, ParameterPortBase &*receiver*)
Register a connection between parameter ports (both real ports of contained components or external placeholder ports).

Parameters

- `sender` - Reference to the sendig port (retrieved, for example using *`Component::parameterPort()`*)
- `receiver` - Reference to the sendig port (retrieved, for example using *`Component::parameterPort()`*)

void **audioConnection**(char **const** **sendComponent*, char **const** **sendPort*, ChannelList **const** &*sendIndices*, char **const** **receiveComponent*, char **const** **receivePort*, ChannelList **const** &*receiveIndices*)
Register an audio connection between a sending and a receiving audio port. This overload uses C strings to denote both the names of the components holding the ports and the output ports itself. Lists of channel indices are to be specified for the sending and the receiving port. The sizes of these lists must be identical, and the contained indices must not exceed the width of the send and receive port, respectively. Empty lists for both the send and receive indices are permitted and result in no connection.

See

ChannelList for the syntax to specify the channel index lists.

Parameters

- `sendComponent` - Name of the component holding the sending audio port. If the send port is an external input of this component, use "" or "this"
- `sendPort` - The name of the sending port.
- `sendIndices` - A list of channel indices denoting the send channels of the sending side.
- `receiveComponent` - Name of the component holding the receiving audio port. If the receive port is an external output of the present component, use "" or "this"

- `receivePort` - The name of the receiving port.
- `receiveIndices` - A list of channel indices denoting the receive channels within the receiver port.

Exceptions

- `std::invalid_argument` - if a specified component or port does not exist.

void **audioConnection** (*AudioPortBase* &*sendPort*, ChannelList **const** &*sendIndices*, *AudioPortBase* &*receivePort*, ChannelList **const** &*receiveIndices*)

Register an audio connection between a sending and a receiving audio port. This overload uses audio ports (either directly referencing external in- and output of this components or retrieving ports of contained components using the *Component::audioPort()* method). Lists of channel indices are to be specified for the sending and the receiving port. The sizes of these lists must be identical, and the contained indices must not exceed the width of the send and receive port, respectively. Empty lists for both the send and receive indices are permitted and result in no connection.

See

ChannelList for the syntax to specify the channel index lists.

Parameters

- `sendPort` - The send port object.
- `sendIndices` - A list of channel indices denoting the send channels of the sending side.
- `receivePort` - The receive port object.
- `receiveIndices` - A list of channel indices denoting the receive channels within the receiver port.

void **audioConnection** (*AudioPortBase* &*sendPort*, *AudioPortBase* &*receivePort*)

Register an audio connection between all channels of a sending and a receiving audio port. This overload uses audio ports (either directly referencing external in- and output of this components or retrieving ports of contained components using the *Component::audioPort()* method). It establishes one-to-one connections between the channels of the sender and the receiver.

Parameters

- `sendPort` - The send port object.
- `receivePort` - The receive port object.

Exceptions

- `std::invalid_argument` - if the port widths do not match.

Class AudioPortBase

class `visr::`**AudioPortBase**

Base class for audio ports. Audio ports can form part of the external interface of components and denote start and end points od audio signal connections. An audio port is characterised by a sample type (fundamental integral and floating-point data type as well as complex floating-point types), the width, that is, the number of elementary audio signals represented by this port.

Protected Functions

void ***basePointer**()

Return the data pointer to fir first (technically zeroth) channel. The type of this pointer is char and needs to be casted in derived, typed port classes.

void **const** ***basePointer**() const

Return the data pointer to fir first (technically zeroth) channel, costant versiob The type of this pointer is char and needs to be casted in derived, typed port classes.

Private Members

impl::AudioPortBaseImplementation ***mImpl**
Pointer to the private, opaque implementation object.

Class AudioInputBase

class `visr::`**`AudioInputBase`**
Base class for audio input ports. This base class is not intended to be used by API users. This class itself cannot be instantiated, because it is not associated with a specific sample type. Only derived classes may actually be instantiated.

Template class AudioInputT

template <typename *DataType*>
class `visr::`**`AudioInputT`**
Class template for concrete audio inputs holding samples of a specific type.

Template Parameters

- `DataType` - The sample type used by this audio port type.

Alias class AudioInput

Non-template of the visr::AudioInputT port template specialised for the default sample type.

typedef **`using visr::AudioInput = typedef AudioInputT<SampleType>`**
Alias for audio input ports using the default datatype (typically float)

Builtin component library (rcl)

Runtime library

Purpose

Main classes

class `visr::rrl::`**`AudioSignalFlow`**
Base class for signal flows, i.e., graphs of connected audio components which perform an audio signal processing operation. This base class provides the infrastructure for setting up the graphs and for transferring the input and output samples. For the audio processing, this class provides a callback interface that must be called in regular intervals (i.e., for a fixed number of samples consumed and generated, respectively).

Checking functions

Internal classes and functions

Object model library

Purpose

API reference

namespace `visr::objectmodel`
The documentation for the namespace objectmodel. Detailed description follows here.

Typedefs

typedef `using visr::objectmodel::ObjectId = typedef unsigned int`

typedef `using visr::objectmodel::GroupId = typedef unsigned int`

typedef `using visr::objectmodel::LevelType = typedef float`
Type use for level (gain, volume) settings, linear scale

typedef `using visr::objectmodel::ObjectTypeIntegerRepresentation = typedef std::uint8_`

Enums

enum `ObjectTypeId`
A numeric id to uniquely describe object types.

Values:

`PointSource` = 0
Simple point-like source (monopole)

`PlaneWave` = 1
Straight plane-wave source type

`DiffuseSource` = 2
Totally diffuse source type

`PointSourceWithDiffuseness` = 3
Point-source-like audio object with an addditional “diffuseness” attribute controlling the fraction of the source that is reproduced diffusely.

`ExtendedSource` = 4
Source type with controllable extent, i.e. width and height.

`PointSourceWithReverb` = 5
Point source with reverberation

`PointSourceExtent` = 6
Point source with explicit spatial extent.

`HoaSource` = 7
Higher Order Ambbisonics object, sound field representation based on spherical harmonics

`ChannelObject` = 8
Source type representing a single or multiple channels routed to a set of loudspeaker channels.

Functions

std::string **const** &**objectTypeToString**(*ObjectTypeId type*)
Convert an object type id into its string representation

Parameters

- `type` -

Exceptions

- `std::logic_error` - Happens only in case of an internal inconsistency, i.e., if the type is not found in the lookup table.

ObjectTypeId **stringToObjectType**(std::string **const** &*typeString*) **Return**

The object id of the type correspoonding to the string representation

Parameters

- `typeString` -

Exceptions

- `std::invalid_argument` - If typeStr does not correspond to an existing object type.

Variables

InstantiateObjectFactory **const cInstantiationHelper**
Object which is used to initialise the object factory.

struct InstantiateObjectFactory
A helper class with whole purpose is to register the different object types in the factory.

class ObjectParser
#include <object_parser.hpp>

class ObjectVector
#include <object_vector.hpp> A class representing a set of audio objects of potentially different types.

class PointSourceWithReverb
#include <point_source_with_reverb.hpp> Audio object representing a monopole point source with corresponding object-based reverberation. Derived from PointSource.

namespace python

Functions

void **exportChannelObject**(pybind11::module &*m*)

void **exportDiffuseSource**(pybind11::module &*m*)

void **exportHoaSource**(pybind11::module &*m*)

void **exportObject**(pybind11::module &*m*)

void **exportObjectType**(pybind11::module &*m*)

void **exportObjectVector**(py::module &*m*)

void **exportObjectVector**(pybind11::module &*m*)

void **exportPointSource**(pybind11::module &*m*)

void **exportPointSourceExtent**(pybind11::module &*m*)

void **exportPointSourceWithDiffuseness**(pybind11::module &*m*)

void **exportPointSourceWithReverb**(pybind11::module &*m*)

void **exportPlaneWave**(pybind11::module &*m*)

namespace `test`

Functions

BOOST_AUTO_TEST_CASE(ParsePointSource)

BOOST_AUTO_TEST_CASE(ParsePlaneWave)

BOOST_AUTO_TEST_CASE(UpdateSceneSameIdSameType)

BOOST_AUTO_TEST_CASE(UpdateSceneSameIdDifferentType)

BOOST_AUTO_TEST_CASE(ParseMultiChannelObject)

BOOST_AUTO_TEST_CASE(ParseObjectEq)

BOOST_AUTO_TEST_CASE(ReencodeObjectEq)

BOOST_AUTO_TEST_CASE(ParseChannelObject)

BOOST_AUTO_TEST_CASE(WriteChannelObject)

BOOST_AUTO_TEST_CASE(InstantiatePointSources)

BOOST_AUTO_TEST_CASE(ObjectVectorAssign)

BOOST_AUTO_TEST_CASE(ParsePointSourceWithReverb)

BOOST_AUTO_TEST_CASE(SerialisePointSourceWithReverb)

BOOST_AUTO_TEST_CASE(InstantiateRenderer)

Elementary functions Library

Purpose

The **efl** library provides a common interface for arithmetic functions commonly used in DSP. These are mainly vector and matrix arithmethic functions. These

Numeric container classes

Arithmetic functions

template <**typename** T>
ErrorCode `visr::efl::`**vectorCopy**(T **const** **source*, T **dest*, std::size_t *numElements*, std::size_t *alignment*)