PyAL Documentation

Release 0.2.0

Marcus von Appen

May 15, 2017

Contents

PyAL is a wrapper around the OpenAL library and as such similar to the discontinued PyOpenAL project. In contrast to PyOpenAL, it has no licensing restrictions, nor does it rely on C code, but uses `ctypes` instead.

CHAPTER 1

Contents

Installing PyAL

This section provides an overview and guidance for installing PyAL on various target platforms.

Prerequisites

PyAL relies on some 3rd party packages to be fully usable and to provide you full access to all of its features.

You must have at least one of the following Python versions installed:

- Python 2.7, 3.1+ (http://www.python.org)
- PyPy 1.8.0+ (http://www.pypy.org)
- IronPython 2.7.3+ (http://www.ironpython.net)

Other Python versions or Python implementations might work, but are (currently) not officially tested or supported by the PyAL distribution.

You must have OpenAL installed. OpenAL-compatible libraries might have shipped with your sound drivers already. Otherwise it is recommended to obtain them from your sound card manufacturer or from http://www.openal.org or http://kcat.strangesoft.net/openal.html.

Installation

You can use either the python way of installing the package or the make command using the Makefile on POSIX-compatible platforms, such as Linux or BSD, or the make.bat batch file on Windows platforms.

Simply type

```
python setup.py install
```

for the traditional python way or

```
make install
```

for using the Makefile or make.bat. Both will try to perform a default installation with as many features as possible.

Trying out

You also can test out PyAL without actually installing it. You just need to set up your `PYTHONPATH` to point to the location of the source distribution package. On Windows-based platforms, you might use something like

```
set PYTHONPATH=C:\path\to\pyal\:$PYTHONPATH
```

to define the `PYTHONPATH` on a command shell. On Linux/Unix, use

```
export PYTHONPATH=/path/to/pyal:$PYTHONPATH
```

For bourne shell compatibles or

```
setenv PYTHONPATH /path/to/pyal:$PYTHONPATH
```

for C shell compatibles. You can omit the *:$PYTHONPATH*', if you did not use it so far and if your environment settings do not define it.

Note: If you are using IronPython, use `IRONPYTHONPATH` instead of `PYTHONPATH`.

Notes on Mercurial usage

The Mercurial version of PyAL is not intended to be used in a production environment. Interfaces may change from one checkin to another, methods, classes or modules can be broken and so on. If you want more reliable code, please refer to the official releases.

Integrating PyAL

PyAL consists of two modules, *`openal`*, which is a plain 1:1 API wrapper around the OpenAL 1.1 specification, and *`openal.audio`*, which contains some high-level audio classes and helper functions, which use the OpenAL wrapper.

Both modules are implemented in a way that shall make it easy for you to integrate and deploy them with your own software projects. You can rely on PyAL as third-party package, so that the user needs to install it before he can use your software. Alternatively, you can just copy both modules into your project, shipping them within your own project bundle.

Importing

The *`openal`* module relies on an external OpenAL library which it can access for creating the wrapper functions. This means that the user needs to have OpenAL installed or that you ship an OpenAL library with your project.

If the user has an OpenAL library installed on the target system, the `ctypes` hooks of *`openal`* try to find it in the OS-specific standard locations via `ctypes.util.find_library()`. If you are going to ship your own OpenAL library with the project or can not rely on the standard mechanism of `ctypes`, it is also possible to set the environment variable `PYAL_DLL_PATH`, which shall point to the directory of the OpenAL library.

Note: `PYAL_DLL_PATH` is preferred over the standard mechanism. That said, if the module finds a OpenAL library in `PYAL_DLL_PATH`, it will try to use that one in the first place, before using any OpenAL library installed on the target system.

Let's assume, you ship your own library *OpenAL.dll* within your project location *fancy_project/third_party*. You can set the environment variable `PYAL_DLL_PATH` before starting Python.

```
# Win32 platforms
set PYAL_DLL_PATH=C:\path\to\fancy_project\third_party

# Unix/Posix-alike environments - bourne shells export
PYAL_DLL_PATH=/path/to/fancy_project/third_party

# Unix/Posix-alike environments - C shells setenv PYAL_DLL_PATH
/path/to/fancy_project/third_party
```

You also can set the environment variable within Python using `os.environ`.

```
os.environ["PYAL_DLL_PATH"] = "C:\\path\\to\\fancy_project\\third_party"
os.environ["PYAL_DLL_PATH"] = "/path/to/fancy_project/third_party"
```

Note: If you aim to integrate *openal* directly into your software and do not want or are not allowed to change the environment variables, you can also change the `os.getenv("PYAL_DLL_PATH")` query within the *openal.py* file to point to the directory, in which you keep the DLL.

Direct OpenAL interaction

openal is a simple (really, really simple) wrapper around the bindings offered by the OpenAL 1.1 specification. Each constant, type and function defined by the standard can be found within *openal*. There are no additional object structures, safety nets or whatever else, so that you can transfer code written using *openal* easily to any other platform in a 1:1 manner.

A brief example in C code:

```
#include <AL/al.h>
#include <AL/alc.h>

int main(int argc, char *argv[]) {
    ALuint source;
    ALCdevice *device;
    ALCcontext *context;

    device = alcOpenDevice(NULL);
    if (device == NULL)
    {
        ALenum error = alcGetError();
        /* do something with the error */
        return -1;
    }
    /* Omit error checking */
    context = alcCreateContext(device, NULL);
    alcMakeContextCurrent(context);
```

```
    /* Do more things */
    alGenSources(1, &source);
    alSourcef(source, AL_PITCH, 1);
    alSourcef(source, AL_GAIN, 1);
    alSource3f(source, AL_POSITION, 10, 0, 0);
    alSource3f(source, AL_VELOCITY, 0, 0, 0);
    alSourcei(source, AL_LOOPING, 1);

    alDeleteSources(1, &source);
    alcDestroyContext(context);
    alcCloseDevice(device);
    return 0;
}
```

Doing the same in Python:

```
from openal import al, alc    # imports all relevant AL and ALC functions

def main():
    source = al.ALuint()
    device = alc.alcOpenDevice(None)
    if not device:
        error = alc.alcGetError()
        # do something with the error, which is a ctypes value
        return -1
    # Omit error checking
    context = alc.alcCreateContext(device, None)
    alc.alcMakeContextCurrent(context)

    # Do more things
    al.alGenSources(1, source)
    al.alSourcef(source, al.AL_PITCH, 1)
    al.alSourcef(source, al.AL_GAIN, 1)
    al.alSource3f(source, al.AL_POSITION, 10, 0, 0)
    al.alSource3f(source, al.AL_VELOCITY, 0, 0, 0)
    al.alSourcei(source, al.AL_LOOPING, 1)

    al.alDeleteSources(1, source)
    alc.alcDestroyContext(context)
    alc.alcCloseDevice(device)
    return 0

if __name__ == "__main__":
    raise SystemExit(main())
```

This does not feel very pythonic, does it? As initially said, `openal` is a really simple, really thin wrapper around the OpenAL functions. If you want a more advanced access to 3D positional audio, you might want to read on about `openal.audio`.

openal.audio - advanced sound support

`openal.audio` is a set of advanced, pythonic classes for 3D positional audio support via the OpenAL standard. It utilises `openal`, but hides all the `ctypes` related, sequential programming workflow from you. It is designed to be non-invasive within a component-based application.

At least three classes need to be used for playing back audio data. *SoundSink* handles the audio device connection and controls the overall playback mechanisms. The *SoundSource* represents an in-application object that emits sounds and a *SoundData* contains the PCM audio data to be played.

Device handling

To actually play back sound or to stream sound to a third-party system (e.g. a sound server or file), an audio output device needs to be opened. It usually allows the software to access the audio hardware via the operating system, so that audio data can be recorded or played back.

```
>>> sink = SoundSink()          # Open the default audio output device
>>> sink = SoundSink("oss")     # Open the OSS audio output device
>>> sink = SoundSink("winmm") # Open the Windows MM audio output device
...
```

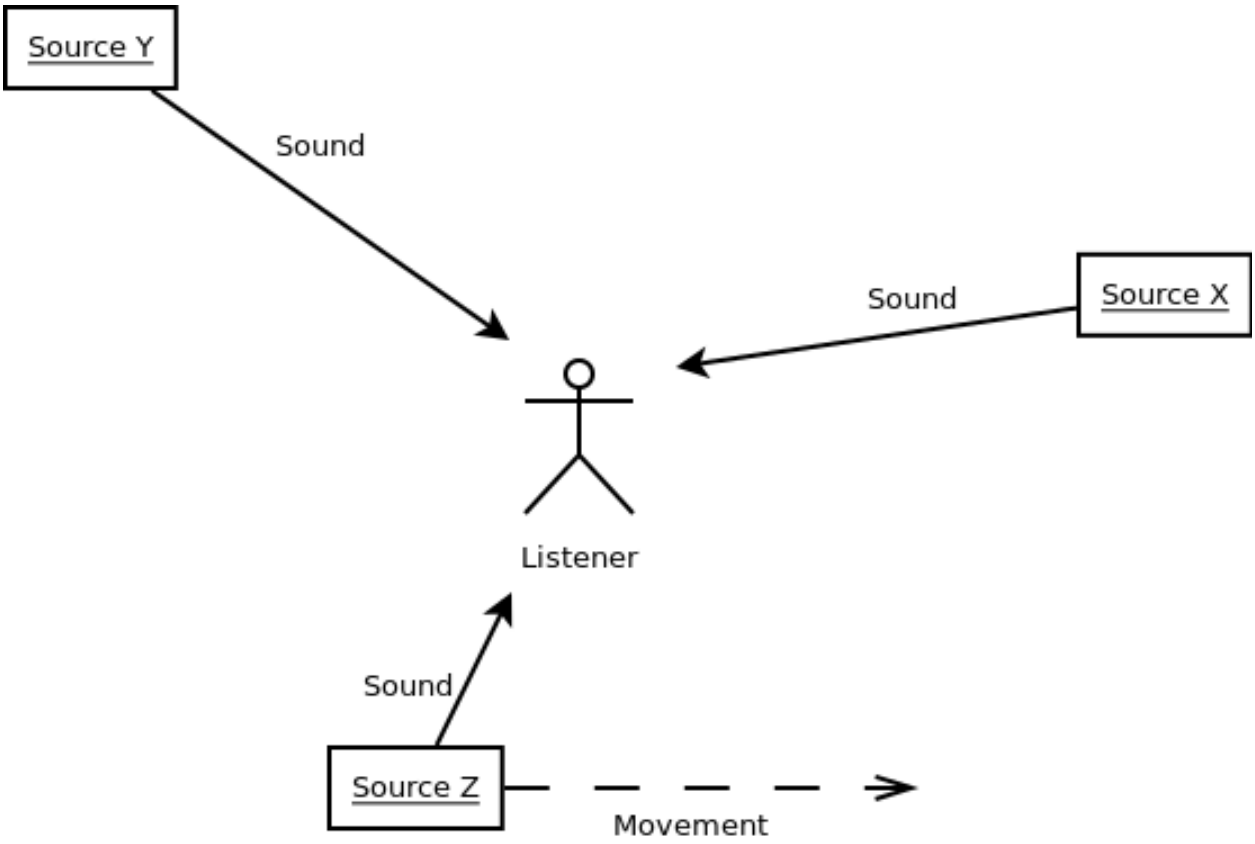
Note: Depending on what to accomplish and what kind of quality for audio output to have, you might want to use a specific audio output device to be passed as argument to the *SoundSink* constructor.

It is possible to create multiple *SoundSink* instances for the same device. OpenAL specifies an additional device-dependent execution context, so that multiple contexts (with e.g. different settings) can be used on one device. Likewise, multiple *SoundSink* objects can use the same device, while each of them uses its own execution context.

Note: Several OpenAL functions perform context-specific operations. If you mix function calls from *openal* with the *openal.audio* module, you should ensure that the correct *SoundSink* is activated via *SoundSink.activate()*.

Placing the listener

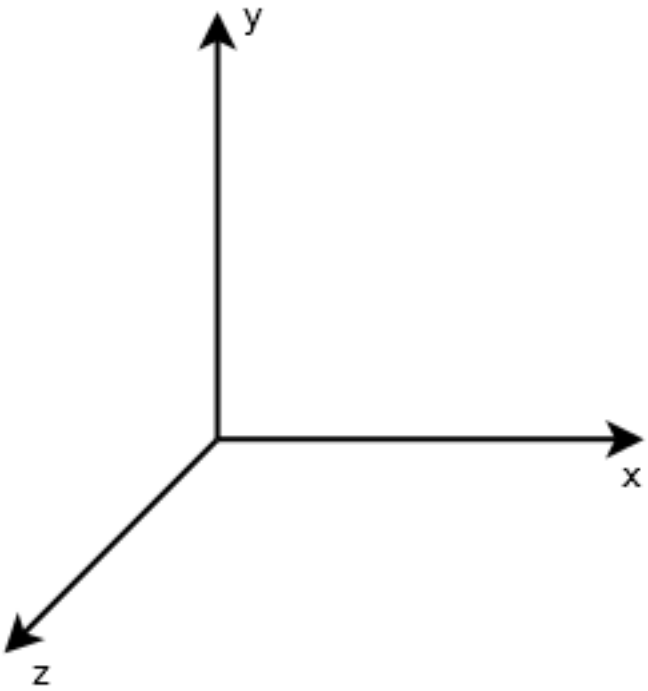
The OpenAL standard supports 3D positional audio, so that a source of sound can be placed anywhere relative to the listener (the user of the application or some in-application avatar).



The image above shows a listener surrounded by three sources of sound. Two are in front of them, while one is behind the listener, moving from left to right.

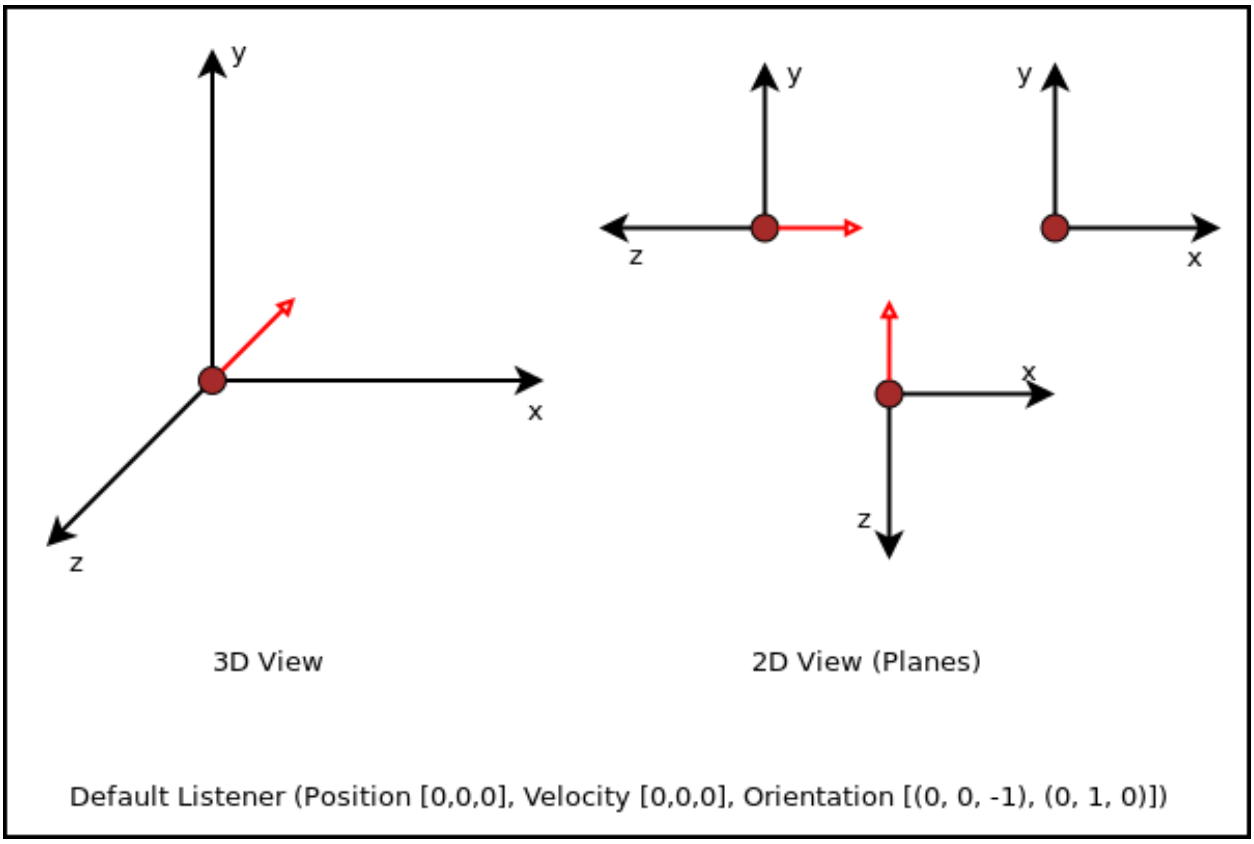
OpenAL only knows about a single listener at each time. Each *SoundSink* can manage its own listener, which represents the user or in-application avatar. As such, it represents the 'pick-up' point of sounds.

Placing and moving the listener (as well as sound sources in OpenAL) is done in a RHS coordinate system. That said, the horizontal extent of your monitor represents the x-axis, the vertical the y-axis and the visual line between your eyes and the monitor surface reprensents the z-axis.



It is crucial to understand how placing and moving sound sources and the listener will influence the audio experience. By default, the listener for each individual *SoundSink* is placed at the center of the coordinate system, `(0, 0, 0)`. It does not move and looks along the z-axis "into" the monitor (most likely the same direction you are looking at right now).

```
>>> listener = SoundListener()
>>> listener.position = (0, 0, 0)
>>> listener.velocity = (0, 0, 0)
>>> listener.orientation = (0, 0, -1, 0, 1, 0)
...
```

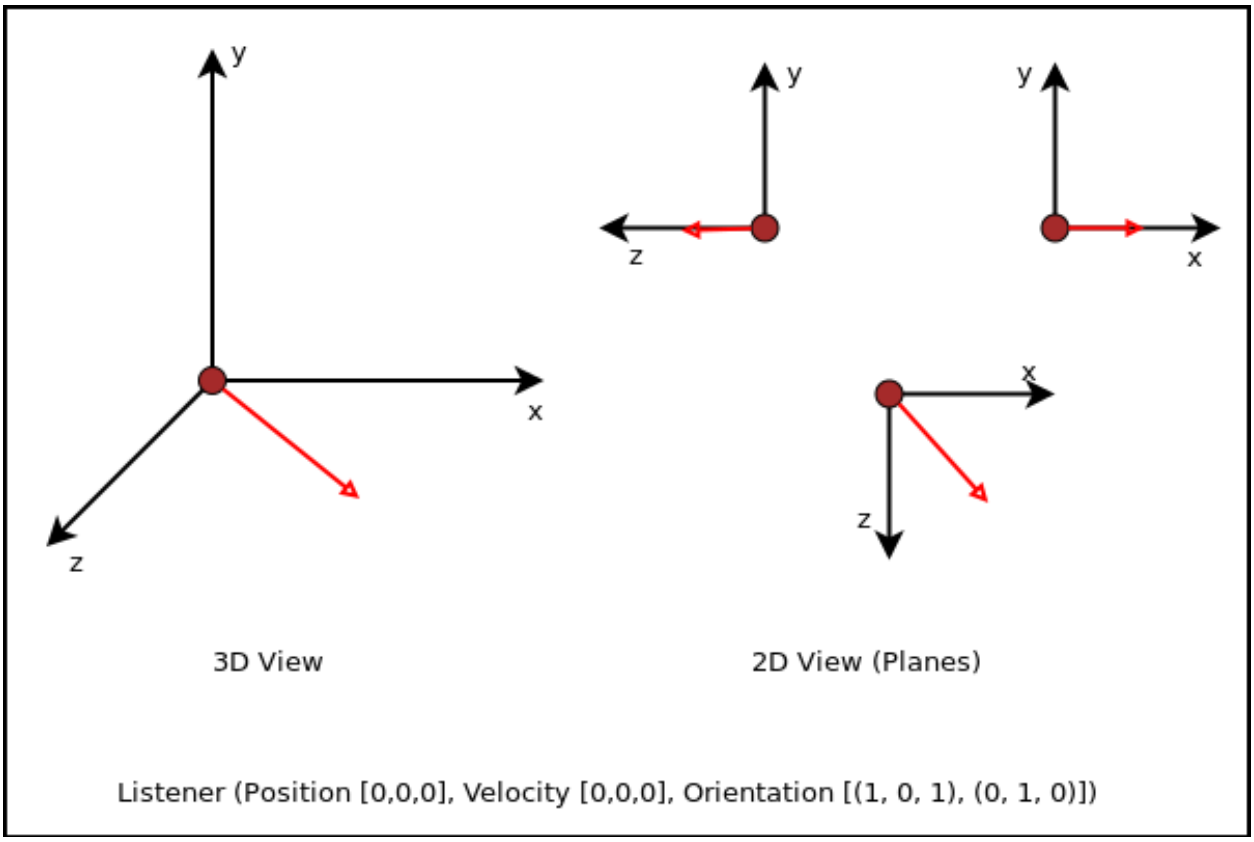


While the *SoundListener.position* and *SoundListener.velocity* are quite obvious in their doing, namely giving the listener a (initial) position and movement, *SoundListener.orientation* denotes the direction the listener "looks at". The orientation consists of two components, the general direction the listener is headed at and rotation. Both are expressed as 3-value tuples for the x-, y- and z-axis of the coordinate system.

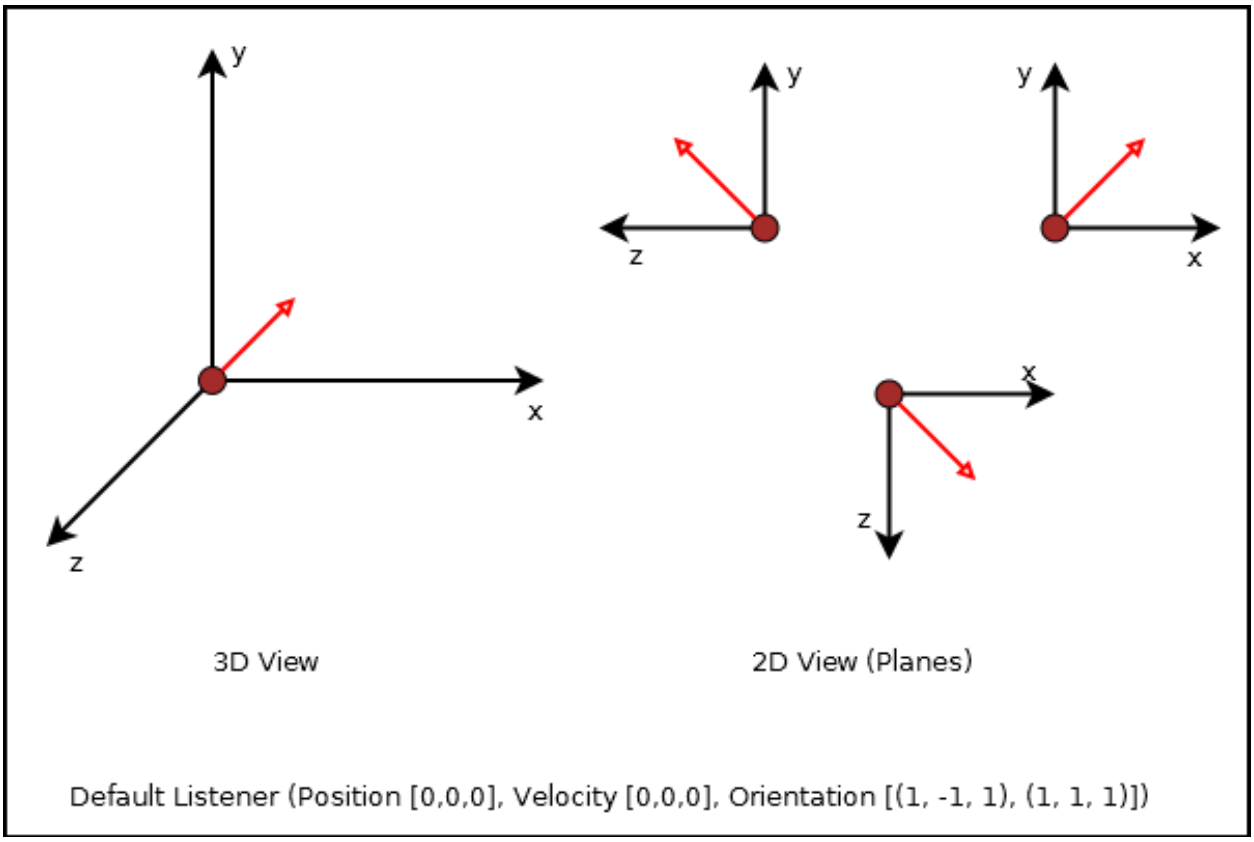
```
>>> listener.orientation = (0, 0, -1,   0, 1, 0)
>>> #                       ^^^^^^^^   ^^^^^^^
>>> #                       direction  rotation
```

Changing the first 3 values will influence the direction, the listener looks at.

```
>>> listener.orientation = (1, 0, 1, 0, 1, 0)
```



Changing the last 3 values will influence the rotation of the looking direction.



The orientation defines a orthogonal listening direction, so that any sounds the user (or avatar) hears, are processed correctly. If you imagine a car driving by on your right side, while you are looking straight ahead (parallel to the car's driving direction), you will hear the car on your right side (with your right ear receiving the most noise). If you look on the street, following the car with your eyes and head, the listening experience will differ (since both ears of you receive the noise in nearly the same way).

Note: Setting the orientation in OpenAL is somehat similar ot OpenGL's `gluLookAt` function, which adjusts the view direction. You might want to take a look at http://www.glprogramming.com/red/chapter03.html#name2 for further details about that.

Creating sound sources

A *SoundSource* represents an object that can emit sounds. It can be any kind of object and allows you to play any sound, you put into it. In an application you can enable objects to emit sounds, by binding a *SoundSource* to them.:

```
>>> source = SoundSource()
```

Todo

more details

Creating and playing sounds

To create and play sounds you use *SoundData* objects, which contain the raw PCM data to be played. To play the sound, the *SoundData* needs to be queued on a *SoundSource*, which provides all the necessary information about the volume, the position relative to the listener and so on.

```
>>> wavsound = load_wav_file("vroom.wav")
```

There are some helper functions, which create *SoundData* objects from audio files. If you have a raw PCM data buffer, you can create a *SoundData* from it directly.

```
>>> rawsound = SoundData(pcmbuf, size_of_buf, channels, bitrate, frequency_in_hz)
```

Queueing the loaded sound is done via the *SoundSource.queue()* method, which appends the sound to the source for processing and playback.

```
>>> wavsound = load_wav_file("vroom.wav")
>>> source.queue(wavsound)
```

You just need to inform the *SoundSink* about the *SoundSource* afterwards, so that it knows that a new sound emitter is available.

```
>>> soundsink.play(source)
```

When you add other sounds to play to the source, they will be picked up automatically for playback, as long as the *SoundSource* is not paused or ran out of something to play.

API

class openal.audio.**OpenALError**([*msg=None*[, *alcdevice=None*]])

An OpenAL specific exception class. If a new *OpenALError* is created and no *msg* is provided, the message will be set a mapped value of `openal.al.alGetError()`. If an `openal.alc.ALCdevice` is provided as *alcdevice*, `openal.alc.alcGetError()` will be used instead of `openal.al.alGetError()`.

class openal.audio.**SoundData**(*data=None*, *channels=None*, *bitrate=None*, *size=None*, *frequency=None*, *dformat=None*)

The *SoundData* consists of a PCM audio data buffer, the audio frequency and additional format information to allow easy buffering through OpenAL.

channels

The channel count for the sound data.

bitrate

The bitrate of the sound data.

size

The buffer size in bytes.

frequency

The sound frequency in Hz.

data

The buffered audio data.

class openal.audio.**SoundListener**(*position=[0, 0, 0], velocity=[0, 0, 0], orientation=[0, 0, -1, 0, 1, 0]*)

A listener object within the 3D audio space.

orientation
The listening orientation as 6-value list.

position
The listener position as 3-value list.

velocity
The movement velocity as 3-value list.

gain
The relative sound volume (perceiptive for the listener).

changed
Indicates, if an attribute has been changed.

class openal.audio.**SoundSource**(*gain=1.0, pitch=1.0, position=[0, 0, 0], velocity=[0, 0, 0]*)
An object within the application world, which can emit sounds.

gain
The volume gain of the source.

pitch
The pitch of the source.

position
The (initial) position of the source as 3-value tuple in a x-y-z coordinate system.

velocity
The velocity of the source as 3-value tuple in a x-y-z coordinate system.

queue(*sounddata : SoundData*) → None
Adds a *SoundData* audio buffer to the source's processing and playback queue.

class openal.audio.**SoundSink**(*device=None*)
Audio playback system.

The SoundSink handles audio output for sound sources. It connects to an audio output device and manages the source settings, their buffer queues and the playback of them.

device
The used OpenAL openal.alc.ALCdevice.

context
The used openal.alc.ALCcontext.

activate() → None
Activates the *SoundSink*, marking its *context* as the currently active one.

Subsequent OpenAL operations are done in the context of the SoundSink's bindings.

set_listener(*listener : SoundListener*) → None
Sets the listener position for the *SoundSink*.

Note: This implicitly activates the *SoundSink*.

---

process_source(*source : SoundSource*) → None
Processes a single *SoundSource*.

Note: This does *not* activate the *SoundSink*. If another *SoundSink* is active, chances are good that the source is processed in that *SoundSink*.

---

process (*world*, *components*) → None
Processes *SoundSource* components, according to their `SoundSource.request`

Note: This implicitly activates the *SoundSink*.

openal.loaders - loading sounds

Todo

Outline

API

`openal.loaders.`**load_file** (*fname : string*) → SoundData
Loads an audio file into a `SoundData` object.

`openal.loaders.`**load_stream** (*source : object*) → SoundData
Not implemented yet.

`openal.loaders.`**load_wav_file** (*fname : string*) → SoundData
Loads a WAV audio file into a `SoundData` object.

Release News

This describes the latest changes between the PyAL releases.

0.2.0

Released on 2013-XX-XX.

- Nothing yet

0.1.0

Released on 2013-04-21.

- Initial Release

Further readings:

Todo list for PyAL

- proper unit tests
- more examples

License

```
This software is distributed under the Public Domain.

In cases, where the law prohibits the recognition of Public Domain software,
this software can be licensed under the zlib lincese as stated below:

Copyright (C) 2012-2013 Marcus von Appen <marcus@sysfault.org>

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Documentation TODOs

Todo

more details

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pyal/checkouts/latest/doc/audio.rst, line 130.)

Todo

Outline

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pyal/checkouts/latest/doc/loaders.rst, line 7.)

Last generated on: May 15, 2017

Python Module Index

o

Index