OpenAL++ - An object oriented toolkit for real-time spatial sound

A master thesis by
Tomas Hämälä
at VRlab, Umeå

11th February 2002

Abstract

In this report, the development of an object oriented toolkit for real-time spatial sound is presented. The theory behind spatial sound and speech recognition is presented. Earlier toolkits were studied, and presented. A portable toolkit, called OpenAL++, was developed. The Teleface project was studied and an outline for integration with OpenAL++ was made.

Contents

List of Figures

1 Introduction

OpenAL++ is a part of the Avatar project which in turn is a part of the Vista project[HOL00]. The goal of the Avatar project is to develop an advanced toolkit for simulating human behaviour in virtual environments. The toolkit will contain modules for artificial intelligence and character animation. OpenAL++ will add sound to the project.

Vista - short for Visual Interactive Simulation Tools and Applications - is a research program that is partly funded by the European Union. Other projects in Vista are real-time physics with haptic feedback and cluster based graphics.

1.1 Scope

Sound is an essential part of everyday life for most people. It gives important information about our surroundings, enhancing the visual sensations from our eyes and giving us awareness of things that we cannot see. This naturally means that sound is also important when we try to simulate the real world[1]. Adding sound to a simulation will of course enhance the experience, but more importantly it can give the same kind of information as sounds do in the real world. For sounds to effectively give this information, they need to have a position in the simulated world and be affected by properties such as distance. This kind of sound is called *spatial sound*.

Making sound spatial can be quite complex, so the preferred method would be to use an existing SDK[2], and there exists several such toolkits. At VRlab, we have several requirements for the SDK we will use for our projects. Obviously it must be real-time (due to the interactive nature of many of our applications), it should be open-source (so that it can be easily extended), it must be portable (at least between Windows and Linux) and it should not be too low-level (that is, it should be easy to use). As the SDK is to be used together with different graphics (and physics etc.) SDK:s, another requirement is that it is not part of a tightly integrated library containing other parts than (spatial) sound. Many existing SDK:s were studied (see section 2), but nothing suitable was found. Hence OpenAL++; our own SDK for spatial sound, built on OpenAL (see section 5.1 for more information on OpenAL).

As mentioned above, spatial sound can be used to give hints about the environment to the user. As such it can be used in real-time, interactive

[1]For the importance of sound in virtual environments, see [LAR01]

[2]Software Development Kit - a library of functions etc.

applications that take place in a virtual environments, like games and VR[3] applications.

1.2 Problem specification

The purpose of this project is to develop an object oriented SDK for handling real-time spatial sound. The SDK should be portable, easy to use, well documented and easily extensible.

Among the capabilities should be:

- Creation of sound sources, and setting their positions as well as other attributes.
- Playing sounds through the sources. These sounds could be pre-recorded or streamed through a microphone or through network sockets.
- Have some kind of mechanism to enhance scalability, in the sense of being able to play many sound sources simultaneously.

In a later stage, techniques from the Teleface project[AGE99] could be integrated with the developed SDK. Teleface is a way to extract facial expressions from sound data - something that would fit in to a virtual world with avatars.

1.3 The report

The remaining chapters of this report are about OpenAL++; its past, present and future. *Previous work* describes a number of earlier spatial sound API:s and SDK:s, and the reasons for not choosing them. It also has sections for OpenAL and PortAudio, the sound SDK:s used. The chapter *Theory of spatial sound* explains the underlying physics, terms and formulas used when working with spatial sound, and *Theory of speech recognition* will talk about techniques for analyzing speech. *Implementation* is a description of the work done and *Results and conclusions* is about how well the goals were met and discusses what else worked well or not so well. Finally, *Future developments* is about possible future additions to OpenAL++.

[3]Virtual Reality

2 Previous work

Real-time simulation of spatial sound is a large research area, and several API[4]:s and SDK:s already exist. For different reasons they were not used, and these reasons are described in this chapter. The last two sections will describe OpenAL and PortAudio; the two sound SDK:s that were used.

2.1 Low level toolkits

DirectX is a popular SDK made by Microsoft[MIC01]. It has sound components for handling spatial sound. However, apart from the fact that it is only available for Windows, it is of quite low level. In fact, it is used as the underlying layer for OpenAL (below) in Windows. Another disadvantage of DirectX is that it is closed source.

SoundRegistry is a thin wrapper[5] for OpenAL[BRÄ01]. It is almost as low level as OpenAL, which means that it is no significant advantage to use it over OpenAL. SoundRegistry is open source.

The SoundTerrain API is a SDK for the MacOS[OLL01]. It is quite similar to OpenAL, but has some added features. The most important of these are that when there are not enough resources for all the sounds to play, SoundTerrain will automatically choose which sounds to play (depending on proximity and loudness), and that the user can set the maximum amount of CPU time SoundTerrain is allowed to use. The latter can be used to limit the amount of time taken by the sound simulation, so that other (time-critical) parts of an application can finish in time.

OpenAL is a low level, open source audio library[LOK00]. It is very portable, but still in development, which means that not everything works on the supported platforms. It is this SDK that was chosen as a core for our SDK, hence the name OpenAL++. More information on OpenAL is available in section 5.1.

2.2 High level toolkits

VESS (Virtual Environment Software Sandbox) is a SDK for virtual environments. It has support for graphics, audio and collision detection, as well as several kinds of input and output devices[UCF01]. The SDK is also quite

[4]Application Programmers Interface, the same thing as SDK

[5]A wrapper is a layer around an underlying library. It is used to hide some of the details of the library and add functionality, thereby bringing it up to a higher level. A "thin wrapper" means that the wrapper does not do much more work than the underlying library.

well documented. VESS has a good API for sound, but it is to integrated with the rest of the VESS package. What we wanted was something that could easily be combined with different graphics (and physics etc.) API:s, like OpenSceneGraph[OSF01] and Vortex[CRI01], for instance.

SGI:s Cosmo3D has the same problem; it is an integrated graphics/sound API. Furthermore it is no longer officially supported by SGI[ECK98].

A company called Aureal had an object oriented API called A3D[AUR00]. It implemented many very good features - HRTF (see chapter 3), wavetracing[6], volumetric sound etc. The drawbacks are that it is only implemented for Windows, and that it is not supported anymore: Aureal went out of business and was bought by Creative, who do not seem to want to proceed the development of A3D.

2.3 Summary of previous work

SDK	Level	Portability	Stand-alone	Open Source
DirectX	Low	Windows only	No	No
SoundRegistry	Low	As OpenAL	Yes	Yes
SoundTerrain	Low	MacOS only	Yes	Yes
OpenAL	Low	Most platforms	Yes	Yes
VESS	High	Unix/Linux	No	Yes
Cosmo3D	High	NT, SGI	No	Yes
A3D	High	Windows only	Yes	No

As can be seen in the table, OpenAL satisfies most of our requirements, which is why it was chosen as the core for our SDK.

[6] A method of "following" the sound waves through space. Using this, objects that are between the sound source and the listener can be found. Any such sources would be said to *occlude* the source, and the sound would be muted. Sound reflections on different surfaces could also be calculated.

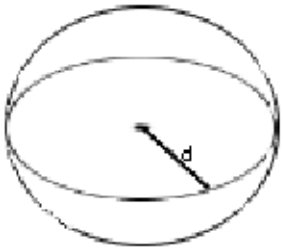


Figure 1: Energy travels in all directions from a point in space. At the distance d, the total area is $4 * \pi * d^2$.

3 Theory of spatial sound

This chapter will explain the underlying physics of spatial audio, and also different ways to do the audio simulation. For basic physics like the following, many sources exist. One good is [HEN01].

3.1 The basic theories

An object that vibrates creates waves that travel through the air. These waves are actually compressions and rarefactions of air, and can also be seen as energy. The amount of energy that is emitted per unit of time is called *power*. It makes more sense to measure how much energy that is in one spot at a specified time, than to measure the total amount of energy, and therefore *intensity* is used. Intensity is the amount of energy through an area per unit of time, that is power per area unit. When something emits a sound, the energy spreads out in all directions (not counting reflections), which means that the further away from the object one gets, the larger the total area will be. This in turn, means that the intensity decreases as the distance to the source of the energy (in this case sound) increases. The area at a certain distance from a point in space is that of a sphere, that is $4 * \pi * d^2$, where d is the distance (see figure 1). Because the intensity is the energy through an area per unit of time, it is inversely proportional to the area. This means that the intensity is inversely proportional to the distance squared. This is known as the inverse square law of physics.

The inverse square law can be used to derive the formula for attenuation[7]

[7]How much a sound is muted depending on distance

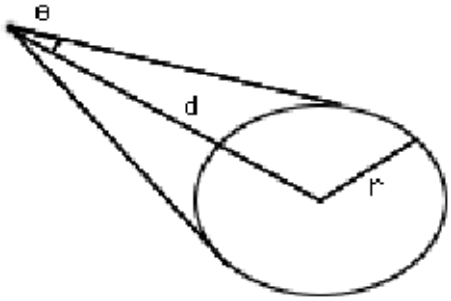


Figure 2: If energy only travels in one direction, the area at distance d is proportional to the area of a sphere with radius d. This means that the area is also proportional to d^2.

in OpenAL (section 5.1). Just taking the basic law would give us:

$$I_{attenuated} = \frac{I_{original}}{d^2} \tag{1}$$

Where I is intensity, and d is distance. However, for practical reasons, OpenAL uses gain (or amplitude) instead of intensity, and intensity is proportional to the square of the gain. This gives us $G^2_{attenuated} = \frac{G^2_{original}}{d^2}$, which can be simplified to $G_{attenuated} = \frac{G_{original}}{d}$.

The formula still has the problem that it is a division by zero if d is zero. Because we know that d (the distance) cannot be negative, all that has to be done to avoid this is to add one to the denominator.

$$G_{attenuated} = \frac{G_{original}}{1 + d} \tag{2}$$

To gain a little more control, two parameters are added: Rolloff factor (F) and reference distance (R). Reference distance is the distance at which the listener will experience $G_{original}$ and Rolloff factor is a factor to increase or decrease attenuation (see [LOK00]).

$$G_{attenuated} = \frac{G_{original}}{1 + F * \frac{d-R}{R}} \tag{3}$$

The above reasoning also works even if energy just propagates in one direction, because the area is proportional to the distance squared even in this case (see figure 2).

Figure 3: The distance from the sound to R is shorter than the distance to L. Hence, the sound will be played at a louder volume in the right channel, giving the illusion the sound is to the right.

So if one now calculated the attenuation based on the distance between the listener and the source, one would get a sense of distance in the simulation: A nearby source would be louder than a far-away. What still would be missing is a sense of direction. To achieve this, two points in space could be used instead of the single point the listener occupies (see figure 3). One point, called L, is placed a short distance to the left of the listener and another, called R, is placed the same distance to the right of the listener. This, of course, demands that the listener has an orientation in space (instead of just a position). Then attenuation is calculated separately for R and L. Sound is then played in the left speaker with the attenuation of L and in the right speaker with the attenuation of R. Doing this means we get a left/right directional sense. To get a sense for whether a sound is in front of or behind, four speakers or a different method than simple attenuation (like HRTF, see below) would have to be used.

The difference in intensity for a sound when reaching one ear compared to the other is called Interaural Intensity Difference (IID). The other main clue that humans use to locate sounds, is called Interaural Time Difference (ITD). The differences in distance from the sound source to a listeners ears means that the sound will reach one ear slightly before the other. The difference in time can easily be calculated, as long as the speed of sound in the medium in question is known (see [WEN92] for information on IID and ITD). IID and ITD do not explain how it is possible to hear whether a sound source is in front of or behind a listener, or how differences in elevation can be heard. To solve these problems - at least to some degree - other methods have to be used.

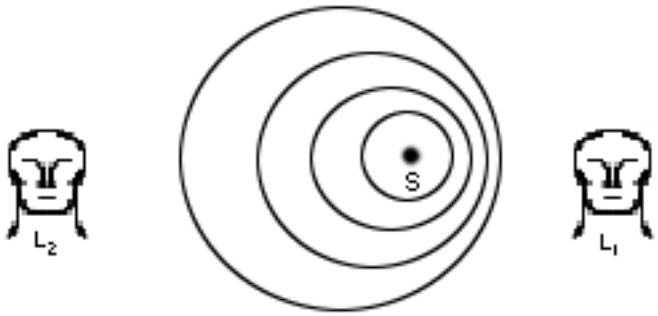


Figure 4: A sound source (S) moves towards a listener (L_1) and away from another (L_2). L_1 will perceive an apparent frequency that is higher than the real frequency of S, and L_2 will perceive an apparent frequency that is lower.

3.2 Head Related Transfer Functions

The use of Head Related Transfer Functions, or HRTF, is a slightly more complex method that is not used in OpenAL. It was developed by NASA in conjunction with Aureal, and much research has been done by those and other companies (see, for example, [WEN92]). The differences in the sound that reaches the left and right ear, respectively, are used by the brain to localize the source of the sound. A first step in calculating how the sound will appear when the source is in different directions from the listener is to place a microphone in each ear of a subject. Then a sound is played at different positions around the subject, and recorded in the microphones. By analyzing the differences in the played and recorded sound, filters can be created that can simulate how the sound will be affected for sources in different directions. These filters, the HRTF, are then applied in real-time to sounds to spatialize them.

The HRTF are actually different for different people, due to differently shaped ears and heads, but non-personal HRTF:s seem to work - although not as well as personal ones. More details on this can be found in [WEN92].

3.3 The Doppler effect

Another phenomenon of sound that is quite common for spatial sound toolkits to simulate, is the Doppler effect. It is an apparent shift in frequency, and appears when a sound source moves relative to a listener or vice versa. If the movement is such that the distance gets smaller, the apparent frequency is higher than the real frequency, and if the movement is the other way,

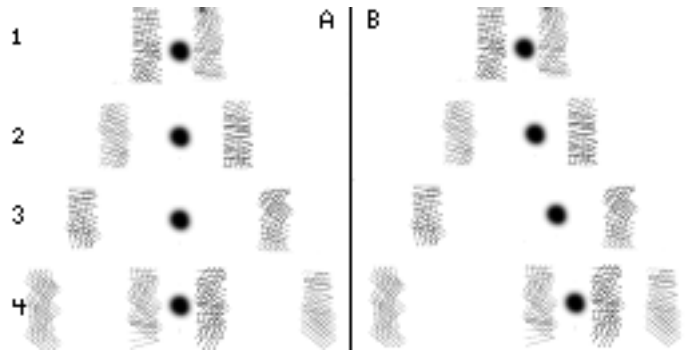


Figure 5: The source in A is stationary, and the source in B is moving to the left. They both have the same frequency: 1/3 vibrations per time step. Time starts with the uppermost row, and goes down one time step per frame. The compressions created by the sound are shown as they propagate through space. As can be seen, the compressions to the right of the moving source are closer to each other than the compressions to the left of the source. Therefore, the apparent frequency is higher to the right of the moving source than to the left of it.

the apparent frequency is lower than the real frequency (see figure 4). The following reasoning explains why this happens:

A sound source will create a number of compressions equal to its frequency per second[8]. If the sound source is stationary, the distance will be the same between any two compressions; this distance is known as the *wave length*. If the sound source is moving, on the other hand, the wave length will be different in different directions: Because the source is moving after the compressions in one direction and away from them in the opposite direction, the wave length will be shorter in the first and longer in the latter (see figure 5). A shorter wave length is the same as a higher frequency. The relationship between wave length and frequency is $f = \frac{v}{\lambda}$, where f is the frequency, v is the speed of sound and λ is the wave length.

There following formulas describe the Doppler effect. In them f' is the apparent frequency, f is the real frequency, s is the speed of sound, s_s is the speed of the source and s_l is the speed of the listener. Both s_s and s_l are speeds along the line defined by the positions of the source and the listener.

$$f' = f * \frac{s}{s \pm s_s} \tag{4}$$

[8]To simplify the discussion, a simple, sinusoidal sound wave of constant frequency is used here.

and

$$f' = f * \frac{s \pm s_l}{s} \tag{5}$$

Equation 4 is for a moving source, and the $\pm s_s$ is s_s if the source is moving away from the listener and $-s_s$ if it is moving towards the listener. If the listener is moving, equation 5 is used. In this case $\pm s_l$ is positive when moving towards source, and negative otherwise.

If, instead of using the speed, a signed value would be used, the $\pm$ would not be necessary. This can be accomplished by projecting the velocity vector(s) onto the vector $\bar{V} = S - L$ (where S is the position of the source and L is the position of the listener). The projected vector is called $\bar{V}'$, and the signed value that is needed is v in $\bar{V} = v * \bar{V}'$. If this signed value is called v_s for the source and v_l for the listener, we get (with equations 4 and 5, respectively):

$$f' = f * \frac{s}{s + v_s} \tag{6}$$

and

$$f' = f * \frac{s - v_l}{s} \tag{7}$$

The two formulas can be combined to handle cases when both the source and the listener move. This formula still works even if source and/or listener is not moving:

$$f' = f * \frac{s}{s + v_s} * \frac{s - v_l}{s} = f * \frac{s - v_l}{s + v_s} \tag{8}$$

In OpenAL, this formula is used, with the addition of a factor (D) that can be used to exaggerate or deemphasize the Doppler effect (see [LOK00]):

$$f' = D * f * \frac{s - v_l}{s + v_s} \tag{9}$$

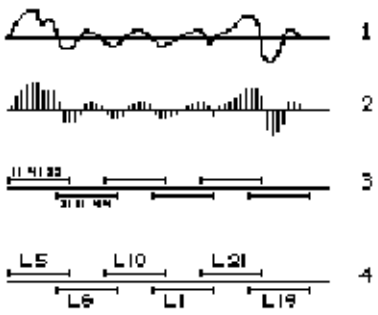


Figure 6: An analog signal (1) is sampled to a digital signal (2). The signal is divided into (overlapping) frames, and the most important features of each frame is stored (3). The reason for the overlap is to avoid loss of features on the boundaries. Then each feature is labeled based on its features (4).

4 Theory of speech recognition

What will be presented here is not so much a way of recognizing what is said as a way of recognizing the phonemes that compose the speech. Recognizing the phonemes is (usually) the first step in computing what is said, though. A *phoneme* is the smallest part of speech, it is one of the about 40-50 sounds that build up our languages. Of course, not all phonemes appear in all languages. Having the phonemes for a sentence could of course be used for speech recognition, but also for making a synthetic face move so it appears that it is uttering the sentence. But more on this later.

Doing any kind of analysis directly on the signal (that is the sound) can be quite complex, simply because the sheer amount of data. Because of this, the signal is usually divided into frames of about 10 milliseconds each. The n most important features of each frame are picked out, and then these are quantized. Quantization means that the n-dimensional space that the vector of features can be seen as points in, is divided into regions and labeled. All points in the same region gets the same label. These labels can be seen as a compressed version of the signal, containing only those features of the original signal that are important for our purposes. We call these labels L1, L2, .. , Lm, where m is the number of regions (see figure 6).

To get from a signal to its phonemes, we need a way to calculate $P(s|p)$, where s is a signal and p a phoneme. This should be read as "the probability that p gives s." That is, we have the signal (or rather a part of it) and want to know the probability that it corresponds to a particular phoneme. Knowing the probabilities for all phonemes means that it is "only" a matter

of assigning the most probable phoneme to the signal. To find it, *Hidden Markov Models* can be used.

In [RUS95] the following is said about *Markov Models*:

> In general, a Markov Model is a way of describing a process that goes through a series of states. The model describes all the possible paths through the state space and assigns a probability to each one. The probability of transitioning from the current state to another one depends only on the current state, not on any prior part of the path.

Hidden Markov Models (HMM) are the same, except for that each state has probabilities for every possible output, not just the state changes, and the same output can appear in more than one state. For every phoneme that is to be recognized, one HMM is needed. To find out what phoneme a sequence of quantization labels corresponds to, the following steps need to be done:

1. For every path that gives the sequence:
2. Multiply the probability of the path with the probability that the path generates the sequence.
3. For every HMM:
4. Sum the probabilities calculated above belonging to the HMM.

What is being calculated here is actually $P(s|p)$, which means that the HMM with the highest probability corresponds to the most likely phoneme. See figure 7 for an example. In the Teleface project[AGE99] this method was used to find the phonemes, and then map them to facial expressions of a virtual face. Another method that was used in Teleface is Artificial Neural Networks (ANN), below.

To find the parameters for the HMM:s, some kind of training method- like the forward-backward algorithm - is used. This will not be discussed further here.

Artificial Neural Networks are based on how the brain processes information. They consist of a number of units (or artificial neurons) that take a number of inputs, process them and produce a single output. The processing takes place in a function that, based on learned parameters, combine the input into an output. The parameters are learned through a process of sending input to the neurons and giving feedback on the output. If the output is not right, the feedback should cause the function to change its behaviour. This has many uses, but in this case it was trained to map the audio input to parameters for describing facial expressions (see section 8).

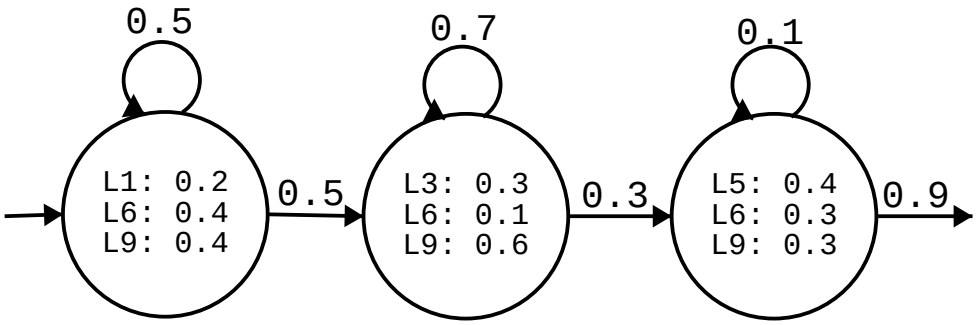


Figure 7: Given that p is the phoneme corresponding to the HMM, above, and that the signal s is quantized as q=[L1,L6,L6,L5]. There are three possible paths that generate that sequence. The probability that q corresponds to p as $P(q|p) = (0.2 * 0.4 * 0.1 * 0.4) * (0.5 * 0.5 * 0.3 * 0.9) + (0.2 * 0.1 * 0.1 * 0.4) * (0.5 * 0.7 * 0.3 * 0.9) + (0.2 * 0.1 * 0.3 * 0.4) * (0.5 * 0.3 * 0.1 * 0.9) = 0.000324$

5 Audio libraries used

The audio libraries used in OpenAL++ are presented in this section.

5.1 OpenAL

OpenAL - or the Open Audio Library - is an open, cross-platform API for interactive, spatial audio. The companies involved, at the time of writing, are Creative Labs and Loki Entertainment Software. The primary audience for OpenAL is game and application developers that use portable standards, like the graphics API OpenGL, for their applications.

The API is designed to have a similar look to OpenGL (in coding style and conventions). As it is primarily made to generate audio in a simulated 3D space, it does not support functions such as panning. The following table summarizes the capabilities of OpenAL:

OpenAL can handle:	OpenAL cannot handle:
Distance based attenuation	Occlusions
Directional sounds	Reflections
Reverb (on some platforms)	ITD
Doppler effects	HRTF
Pitch changes	Volumetric sound
	Direct control over sound channels

See chapter 3 for more information on the above terms.

OpenAL has five main components:

- A context for the sound simulation.
- A device to output sound on. This is associated to the context, and every context must have a device.
- A listener. This is a static object, so there is one listener in every context. This has both position and orientation in the virtual environment, as well as other parameters such as velocity.
- A number of sources. These have position, direction and other parameters such as velocity and pitch. Sources are created on demand.
- A number of buffers. Each one of these is associated with a number of sources. Buffers are created when needed.

See figure 8 for a conceptual picture of OpenAL.

Before using any other parts of OpenAL, a context and its device must be allocated, and when sound simulation is done, the device and context should

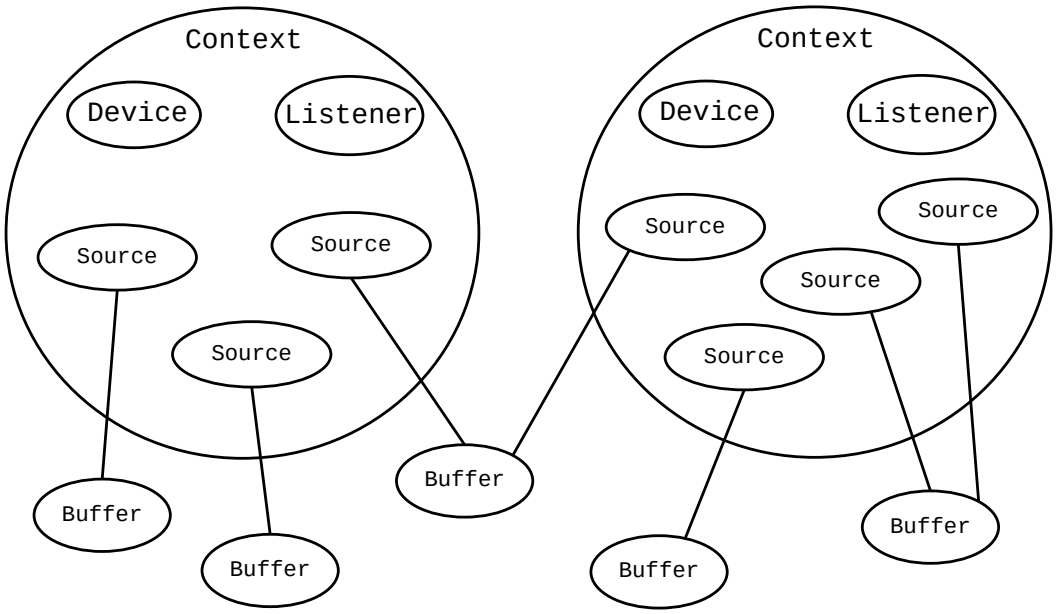


Figure 8: As can be seen from this conceptual picture of OpenAL, several contexts can exist (though only one can be active at a time). Every context has a listener and a device. A number of sources can also exist in a context. Buffers are not context specific and can be shared between contexts as well as sources.

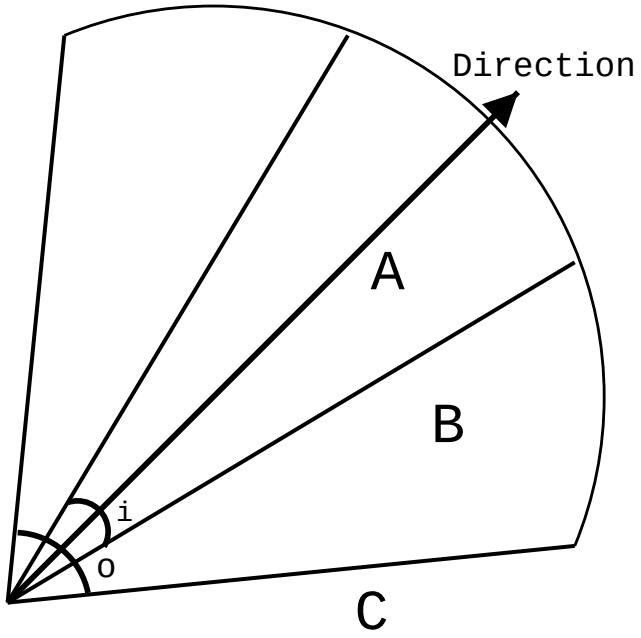


Figure 9: Example of a directional source in OpenAL. i is the inner cone angle and o is the outer cone angle. In area A, the source is at full gain and in area C it is at outer cone gain. In area B the gain is interpolated between full gain and outer cone gain.

be freed. These steps can be handled by utility functions in the ALUT (AL Utility Library), but then some control is lost. Using ALUT means that it is not possible to set refresh rate, output frequency and whether the audio context should be synchronous (with graphics) or not. It also means that explicit control over devices is lost, so the default device will be used, with default settings.

ALUT can also be used to load files in WAV format[9], which can then be loaded into buffers. The buffers are then associated with sources. The sources have a number of attributes, like position and velocity, which will affect how it will sound when played. The source attributes can be found in table 1.

As mentioned above, every context has a listener. Listener attributes can be found in table 2. Buffers only have the attributes frequency, size and data - with the obvious uses.

Perhaps the most important part of OpenAL's simulation of sound is

[9]A popular sound storage format

Attribute	Comment
Position	This can be absolute or relative to the listener
Velocity	In vector form. Used for Doppler calculations
Direction	For directional sources (see fig. 9)
Inner cone angle	For directional sources
Outer cone angle	For directional sources
Outer cone gain	For directional sources
Pitch	
Gain	
Max gain	Can be used to clamp the gain (after attenuation)
Min gain	Can be used to clamp the gain (after attenuation)
Rolloff factor	Used in attenuation calculations
Reference distance	Used in attenuation calculations
Looping	

Table 1: Source attributes in OpenAL

Attribute	Comment
Position	Absolute
Velocity	As for sources
Orientation	Up and at vectors (orthogonal to each other)
Gain	The "global" gain.

Table 2: Listener attributes in OpenAL

that it does *attenuation*. This means that it calculates how much a sound should be muted because of the distance between the source and the listener. Combine this with different calculations for two (or more) channels based on the orientation on the listener, and you get a (primitive) way of simulating distance and direction to sounds. OpenAL's way of calculating attenuation is based on the inverse square law of physics (see chapter 3 *Theory of spatial sound*, equation 3).

5.2 PortAudio

PortAudio is a portable toolkit for audio. It does not have any support for spatial sound, but rather supports more basic audio input and output. Unlike the very rudimentary audio capture support of OpenAL, PortAudio has good support for querying available input devices and their capabilities. For that reason PortAudio was selected for use in the input device part of OpenAL++. However, because of the very modular design of OpenAL++, it could easily be adapted to not use PortAudio if better input support is implemented in OpenAL.

6 Implementation

The development of OpenAL++ started with studies on existing audio toolkits (OpenAL, VESS, SoundRegistry, SoundTerrain, DirectX audio components, A3D, Cosmo3D audio components). This was an attempt to learn from earlier work - both mistakes and successes in design. After this the design phase started. Over three weeks were committed to design, during which the class diagram was revised several times, and tests were made on paper. To do the diagrams, I used Dia[LAR00] to begin with, but later moved to Rational Rose[RAT02] to get code generation capabilities.

Several design choices had to be made before reaching the final stage. The most important were:

- The decision to keep the logical structure - Listener, Source, Buffer and Environment - of OpenAL.
- Sources and other objects in the simulated world could have been stored in a scene graph structure[10], but we decided not to do this. The primary advantages of having a scene graph are that occlusions and reflections would be easier to implement and that integration with graphics might be simpler. However, because OpenAL does not support it, we chose not to implement any kind of occlusion. Also, matrices are the best way to store the transformations in a scene graph, but OpenAL does not use them.
- To achieve better scalability (in number of sounds played simultaneously), GroupSource was added. An alternative would have been to have some kind of built in system for selecting which sources to play.

During the work I found a few things that did not work too well. These were:

- The design of the classes for streaming sound. This was a result of a few things: I was not certain at the time of design on how to do streaming with OpenAL (as OpenAL is still in development, there were a couple of alternatives). I also did not know how to handle input devices; as was mentioned above, OpenAL does not have much support for audio input, and I had initially some problems with getting OpenAL to work

[10] A hierarchical way of structuring three-dimensional data, usually graphics. A scene graph has nodes of several types; primarily transformations and geometries. To render the scene, the graph is traversed, doing transformations and drawing geometries as the corresponding nodes are visited. This technique can also be extended to physics and spatial sound.

with PortAudio. Because of this, some changes in this part of the design had to be made during implementation.

- The current version of OpenAL++ is not totally portable. This is because of the differences in how the (supposedly portable) OpenAL, ALUT and PortAudio libraries work on different platforms. So everything in OpenAL++ does not work the same in Linux and Windows - the main difference is in the streaming functions (streaming through network sockets on Windows might give a distorted sound, while streaming from a microphone might not work in Linux at all). Also, no testing has been done on other platforms than these two.

The main thing that worked well was the overall design. Much work was put into designing - over three weeks were spent on this, and much time on studies of other API:s - and the implementation did not begin until the major part of the design was done. This led to a fast implementation of the basic functionality of OpenAL++, with very few problems (except those noted above).

During the work on OpenAL++ a few bugs in were found OpenAL - bugs which had to be fixed to get all the wanted functionality. They have been reported and fixed, but there is still some work to do to get to a stable, totally portable OpenAL.

6.1 Structure of OpenAL++

OpenAL++ is built using three other libraries: OpenAL, CommonC++[SUG00] and PortAudio[BUR01]. OpenAL is, of course, the foundation and what gives most of the sound capabilities of OpenAL++. PortAudio takes care of sound capture - input from different devices - which is one thing OpenAL does not handle well. CommonC++ is used to get portable support for threads and sockets.

An object oriented approach was used to build OpenAL++. It uses virtual base classes to take care of initialization. This means that little or no explicit initialization is necessary, with the exception of reverb. The reason reverb needs explicit initialization is partly that it is an extension, and therefore should not be initialized unless it is to be used, and partly because it is not available on all platforms.

The classes a programmer will use follow the same structure as OpenAL: The sound scene is divided into Listener, AudioEnvironment, Source(s) and SoundData[11]. In OpenAL++ it is also possible to create several Listeners

[11] corresponding to OpenAL buffers

and quickly switch between them.

The AudioEnvironment class is used for setting global parameters - like Doppler settings and global volume. The Listener class takes care of listener position and orientation. Note that none of these have to be instantiated; if they are not, default values are set for them.

A sound source can be either a Source - for playing one sound - or a GroupSource - for mixing several sounds together. The GroupSource class can be used to enhance scalability; on some platforms - like Windows - the maximum number of sources is dependent on the hardware and usually is something like 32. Using GroupSource, several sounds - preferably such that have static positions in space relative to each other - can be mixed into one source, thereby freeing resources to other sounds.

SoundData can be of two different types: Sample and Stream. Sample is just an ordinary sound sample, loaded from disc. Stream is some kind of streamed sound, either NetStream - a stream through a socket - or InputDevice - some kind of input device, a microphone for example. To accomplish streaming, double buffering is used. The following algorithm describes the process:

1. Queue two buffers in the source.
2. Wait for new data to arrive.
3. Wait for OpenAL to process one of the buffers.
4. Unqueue the processed buffer.
5. Fill the buffer with the new data.
6. Queue the buffer.
7. Goto 2.

As long as a suitable buffer size (depending on sampling rate etc.) is chosen, no glitches in sound should be heard[12]. This is because two buffers are used, so one buffer can be updated while the other one is playing. Every stream has its own thread for doing the above process (see figure 10).

A NetStream uses a UDP socket for the sound and, optionally, a TCP socket for control messages. The control messages are, at the moment, only sample frequency, message size and a message for stopping communication. They are entirely one way, from the sender of the audio stream to the receiver of it.

[12]Notice that some implementations of OpenAL might have some problems with glitches between queued buffers

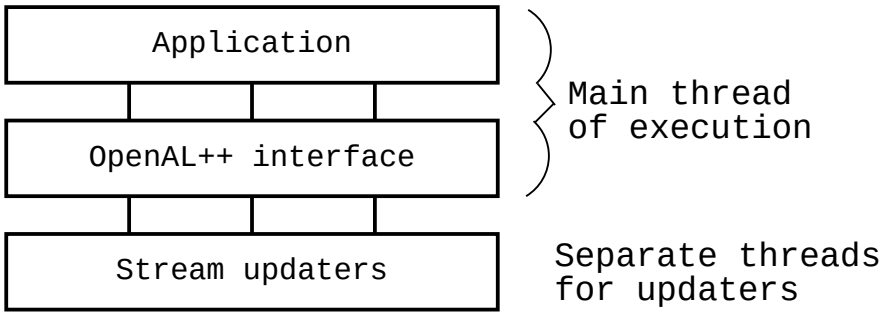


Figure 10: OpenAL++ uses separate threads for updating streams. These are handled invisibly to the user of the SDK.

7 Results and Conclusions

The major goals of OpenAL++ were scalability, portability and ease of use. It should also be well documented. These goals were reached in varying degrees. The project certainly is well documented, as Doxygen[VAN02] was used as part of the implementation process. It is also very easy to use: a simple scene can be set up with a few lines of code, with no initialization or exit code. More complex things, like streaming data from a microphone or through sockets, can also be achieved easily.

The goal of portability was not a total success - although the basic functionality should be the same in Windows and Linux, some things might differ, because of the different ways OpenAL works on different platforms. The scalability aspect might not be handled as good as one would want it to. Group sources certainly enhance scalability, but they demand some work from the user. Furthermore, group sources only help to reduce the load on the system by reducing the number of playing sources - they do not reduce the number of allocated sources. This can be a problem on platforms where the number of allocated sources is very limited. An automatic way to select which sources to simulate properly would certainly be easier for the user, but would be hard to implement because how the selection process should work depends on the application. Also, such a selection mechanism would be better implemented in a lower level (that is in OpenAL).

8 Future developments

At a later stage, techniques developed in the Teleface project of KTH could be added to OpenAL++. The methods of the Teleface project are used to analyse an audio stream - interpreted as speech - and get the corresponding lip movements from it. This analysis is done by using either Hidden Markov Models or Artificial Neural Networks.

This could be very useful in a virtual environment. For example a participant in the world could use a microphone to talk to others in the environment. OpenAL++ would make his voice come from the position of his avatar, and Teleface technology could make the lips of his avatar move in a convincing way. This would, apart from being more realistic, be good for the hearing impaired. According to studies (on hearing impaired people) reported in [AGE99], intelligibility was increased from 33.7% for audio only to 54.0% with an artificial face with lip movements.

There is, however, at least one problem with using this with OpenAL++: OpenAL does not have any direct way to query the position that is being played at the moment, nor what the sound data looks like at that position. One possible solution to this would be to have very small segments of sound data that are queued. OpenAL could then be queried about how many segments have been played, and thus one would know which segment is currently being played. Assuming the programmer has saved the queued data somewhere, he could find the right segment in that data and use it for the analysis. As this should be independent of attenuation, the original data can be used. As long as no pitch changes are applied to the sound, this should work fine, and even in that case, it would not be very hard to (at least) approximate the changes.

9 Acknowledgments

The author wishes to thank

The people on the OpenAL mailing list, for fast and helpful answers, especially Joe Valenzuela - the maintainer of the Linux version of OpenAL.

And, of course, my supervisor Anders Backman.

References

[AGE99] Agelfors, Eva <eva@speech.kth.se> et. al : Synthetic visual speech driven from auditory speech, 1999

[AUR00] Aureal Inc.: A3D 3.0 API Reference Guide (2000)

[BRÄ01] Brändle, Christian <christian.braendle@fhs-hagenberg.ac.at>, Bailer, Werner <werner.bailer@fhs-hagenberg.ac.at>: SoundRegistry, 2001-11-12 <http://www.fhs-hagenberg.ac.at/staff/haller/openal/SoundRegistryC.zip> (2002-01-10)

[BUR01] Burk, Phil <philburk@softsynth.com>: PortAudio - Portable Audio Library, 2001-11-29 <http://www.portaudio.com/> (2001-12-13)

[CRI01] Critical Mass Labs: Vortex, 2001-05-15 <http://www.cm-labs.com/products/vortex/> (2002-01-14)

[ECK98] Eckel, George: Cosmo 3D Programmer's Guide, 1998 <http://techpubs.sgi.com/library/manuals/3000/007-3445-002/pdf/007-3445-002.pdf> (2002-01-15)

[HEN01] Henderson, Tom <thenderson@glenbrook.k12.il.us>: The Physics Classroom - Sound Waves and Music, 2001-02-19 <http://www.glenbrook.k12.il.us/gbssci/phys/Class/sound/soundtoc.html> (2001-12-01)

[HOL00] Holmlund, Kenneth <kenneth@hpc2n.umu.se>: Vista, 2000-05-15 <http://www.vrlab.umu.se/forskning/vista.shtml> (2001-12-01)

[LAR00] Larsson, Alexander <alla@lysator.liu.se>: Dia a drawing program, 2000-02-29 <http://www.lysator.liu.se/ alla/dia/dia.html> (2002-01-10)

[LAR01] Larsson, Pontus <pontus.larsson@ta.chalmers.se>, Västfjäll, Daniel and Kleiner, Mendel: Do we really live in a silent world? - The (mis)use of audio in virtual enviroments, 2001-10-04 <http://vrlcb.sm.chalmers.se/conference/pdfs/21%20larsson.pdf> (2002-01-15)

[LOK00] Loki Entertainment Software <info@openal.org>: OpenAL — Open Source Audio Library, 2000-11-04 <http://www.openal.org/home/> (2002-01-10)

[MIC01] Microsoft corporation: About DirectX - Microsoft DirectX, 2001-11-01 <http://www.microsoft.com/directx/homeuser/aboutdx.asp> (2002-01-10)

[OLL01] Ollman, Ian <iano@cco.caltech.edu>: SoundTerrain, 2001-01-01 <http://alienorb.com/SoundTerrain/> (2001-12-19)

[OSF01] Osfield, Robert <robert@openscenegraph.com>: Open Scene Graph, 2001-12-30 <http://www.openscenegraph.org/> (2002-01-14)

[RAT02] Rational Software: Rational Rose, 2002 <http://www.rational.com/products/rose> (2002-01-13)

[RUS95] Russell, Stuart and Norvig, Peter: Artificial Intelligence - A modern approach, chapter 24.7 Speech Recognition, 1995, Prentice hall

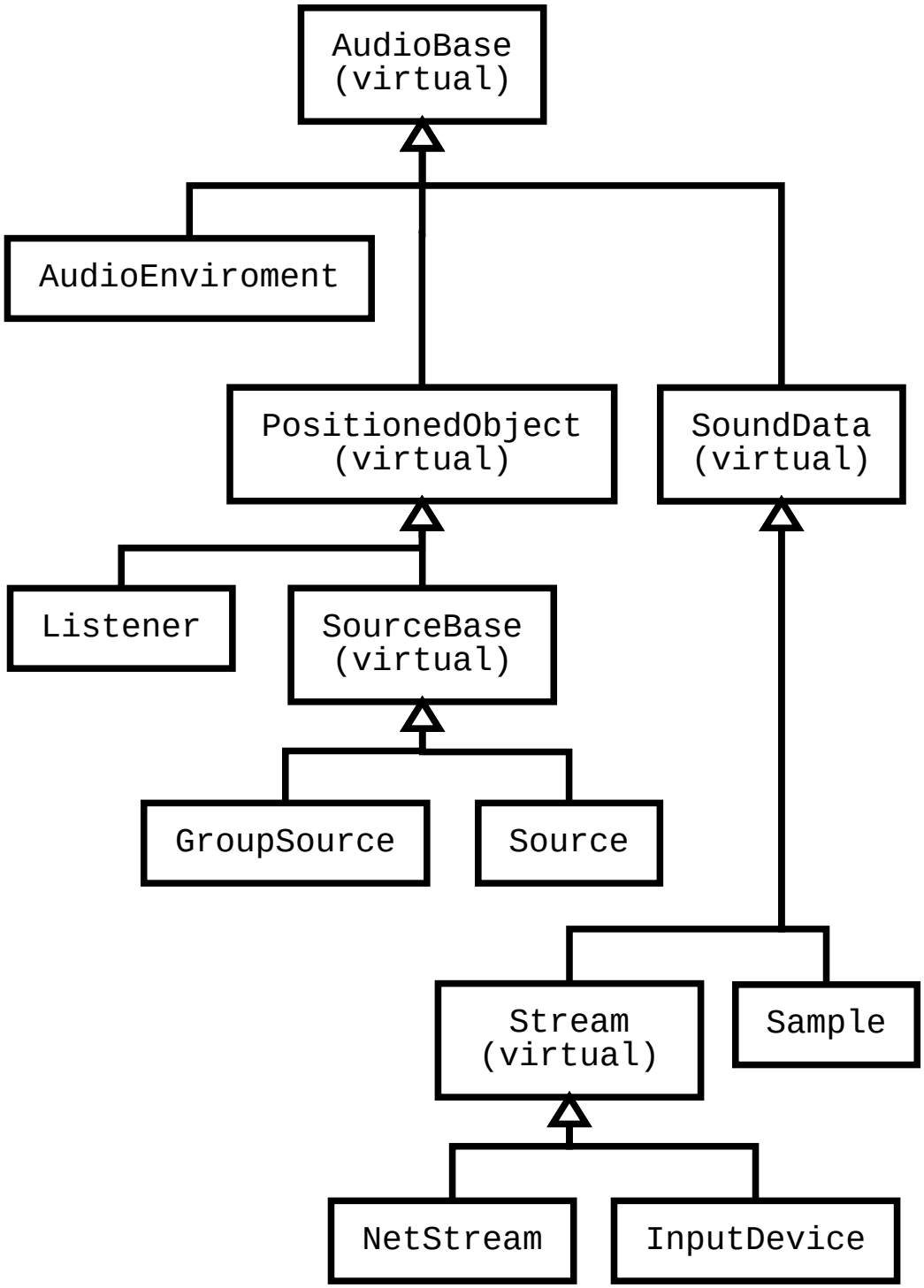
[SUG00] Sugar, David <dyfet@ostel.com>: Common C++ - A GNU Portable Application Framework, 2000-04-30 <http://cplusplus.sourceforge.net/> (2001-12-13)

[UCF01] University of Central Florida: Virtual Environment Software Sandbox, 2001-08-01 <http://vess.ist.ucf.edu/> (2002-01-10)

[VAN02] van Heesch, Dimitri <dimitri@stack.nl>: Doxygen, 2002-01-05 <http://www.doxygen.org/> (2002-01-10)

[WEN92] Wenzel, Elizabeth M. <bwenzel@mail.arc.nasa.gov>: Localization in Virtual Acoustic Displays, Presence, volume 1, number 1, pp. 80-107, 1992

A Design


AudioBase
(virtual)
AudioEnviroment
PositionedObject
(virtual)
SoundData
(virtual)
Listener
SourceBase
(virtual)
GroupSource
Source
Stream
(virtual)
Sample
NetStream
InputDevice

AudioBase Virtual base class. Takes care of initialization and shutdown. *(page B-3)*

AudioEnvironment Class for global parameters, such as volume, Doppler factor etc. *(page B-4)*

PositionedObject Virtual base class for listeners and sources. *(page B-8)*

Listener Class for listeners. *(page B-10)*

SourceBase Base class for sources (GroupSource and Source). *(page B-13)*

GroupSource Class for mixing together several sources into one. *(page B-23)*

Source Class for sources. *(page B-25)*

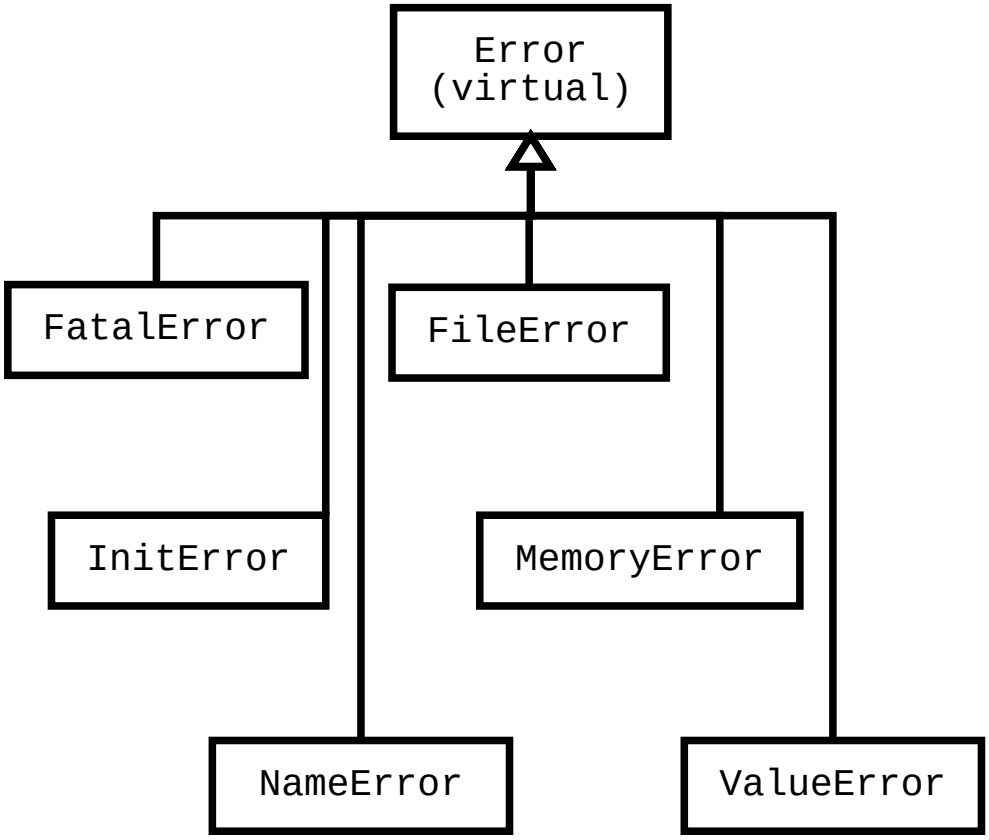
SoundData Base class for sounds. *(page B-29)*

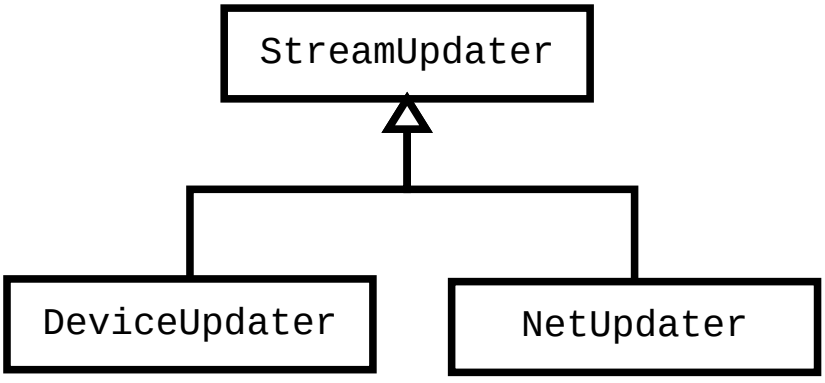
Sample Class for loading sampled sound files. *(page B-30)*

Stream Base class for streamed sounds. *(page B-31)*

NetStream Class for sounds streamed through network sockets. *(page B-34)*

InputDevice Class for sounds streamed from an input device (a microphone for example). *(page B-33)*


Error
(virtual)
FatalError
FileError
InitError
MemoryError
NameError
ValueError


StreamUpdater
DeviceUpdater
NetUpdater

Error Base class for errors that are thrown by OpenAL++. *(page B-35)*

FatalError Error caused by bugs (in OpenAL++ or one of the libraries it uses), corrupted memory etc. *(page B-36)*

FileError Caused by wrong permissions, missing files etc. *(page B-37)*

InitError Caused by trying to do things without proper initialization, or failure in initialization. *(page B-38)*

MemoryError Caused by insufficient memory etc. *(page B-38)*

NameError Caused by invalid OpenAL identifiers. *(page B-39)*

ValueError Caused by values out of range etc. *(page B-40)*

–

StreamUpdater Threaded base class for updating streams.

DeviceUpdater Updater for devices (like microphones).

NetUpdater Updater for streaming through network sockets.

B Documentation

Large parts of this documentation is based on the latex code automatically generated by Doxygen.

B.1 Requirements

To use the OpenAL++ SDK, three libraries are needed:

- OpenAL. This can be downloaded from at <http://www.openal.org>. It can also be acquired through CVS or downloaded as precompiled binaries; for information on this, check the mentioned URL.
- CommonC++. At least version 1.9 is needed.
  Check <http://cplusplus.sourceforge.net/> for information on downloading.
- PortAudio. For downloads, check <http://www.portaudio.com/>. Note that it is possible to compile OpenAL++ without this, if audio capture is not needed. The files inputdevice.* and deviceupdater.* should then be left out of the compile.

B.2 Installation

Unfortunately, no automatic installation exists at this moment. In a Unix environment (like Linux), the following steps should be done:

1. Download the OpenAL++ source package. Uncompress it if necessary.
2. Enter its directory (alpp).
3. Type make.
4. The library files are in ./lib. Copy these to the appropriate place (for example: *cp lib/lib* /usr/local/lib*).
5. The include files should be put in a place that is in your include path (for example:
   mkdir /usr/local/include/alpp; cp include/.h /usr/local/include/alpp*).

It should then be possible to compile an openalpp program with the flag -lopenalpp.

In windows, do the first two steps above. Then open openalpp.dsw (in the alpp directory) with Visual C++, and build the library. The lib-file will

appear under Debug or Release, depending on which mode it was compiled in. Copy this to the appropriate place, and do the same with the include files (under include).

B.3 OpenAL++ Hierarchical index

Only classes that can be used by the SDK user will be documented here[13].

[13] virtual base classes for these are also documented

B.4 openalpp::AudioBase Class Reference

Base class for environment, listener and source classes.

`#include <audiobase.h>`

Protected Methods

- **AudioBase** (int frequency=-1, int refresh=-1, int synchronous=-1) throw (InitError)

  Constructor.

- virtual ∼**AudioBase** ()

  Destructor.

Static Protected Attributes

- bool **reverbinitiated_**

  Flag for whether reverb has been initiated.

- void(∗ **alReverbScale**)(ALuint sid, ALfloat param)

  Set reverb scale.

- void(∗ **alReverbDelay**)(ALuint sid, ALfloat param)

  Set reverb delay.

B.4.1 Detailed Description

Base class for environment, listener and source classes.

Takes care of initialisation/shutdown of anything necessary (e.g. ALut)

Constructor & Destructor Documentation

openalpp::AudioBase::AudioBase (int *frequency* = -1, int *refresh* = -1, int *synchronous* = -1) throw (InitError) `[protected]`

Constructor.

Parameters:

frequency is the output frequency, in Hz.

refresh is the refresh rate, in Hz.

synchronous is a flag for syncronous context. Values <0 indicates that the default should be used.

Member Data Documentation

void(∗ openalpp::AudioBase::alReverbDelay)(ALuint sid, ALfloat param) `[static, protected]`

Set reverb delay.

This pointer will be set by AudioEnvironment::InitiateReverb()

Parameters:

sid is the OpenAL name for the source

param is the reverb delay. Range [0.0,2.0].

void(∗ openalpp::AudioBase::alReverbScale)(ALuint sid, ALfloat param) `[static, protected]`

Set reverb scale.

This pointer will be set by AudioEnvironment::InitiateReverb()

Parameters:

sid is the OpenAL name for the source

param is the reverb scale. Range [0.0,1.0].

bool openalpp::AudioBase::reverbinitiated_ `[static, protected]`

Flag for whether reverb has been initiated.

Reverb can be initiated with AudioEnvironment::InitiateReverb() The documentation for this class was generated from the following file:

- **audiobase.h**

B.5 openalpp::AudioEnvironment Class Reference

Class for setting global parameters.

`#include <audioenvironment.h>`

Public Methods

- **AudioEnvironment** () throw (InitError)

  Constructor.

- **AudioEnvironment** (int frequency, int refresh, bool synchronous) throw (InitError)

  Constructor.

- **AudioEnvironment** (int frequency, int refresh=-1) throw (InitError)

  Constructor.

- void **SetSoundSpeed** (float speed) throw (ValueError,FatalError)

  Sets the speed of sound in the environment.

- float **GetSoundSpeed** () throw (FatalError)

  Get the speed of sound in the environment.

- void **SetDopplerFactor** (float factor) throw (ValueError,FatalError)

  Sets the Doppler factor.

- float **GetDopplerFactor** () throw (FatalError)

  Gets the Doppler factor.

- void **SetGain** (float gain)

  Sets global gain (volume).

- float **GetGain** () throw (FatalError)

  Gets the global gain.

- void **SetDistanceModel** (**DistanceModel** model) throw (FatalError)

  Sets the distance model used in attenuation calculations.

- **DistanceModel GetDistanceModel** () throw (FatalError)

Gets the distance model used in attenuation calculations.

- void **InitiateReverb** () throw (InitError)

  Initiates Loki's reverb implementation.

B.5.1 Detailed Description

Class for setting global parameters.

This doesn't have to be instantiated if one does not need to set global parameters.

Constructor & Destructor Documentation

openalpp::AudioEnvironment::AudioEnvironment (int *frequency*, int *refresh*, bool *synchronous*) throw (InitError)

Constructor.

The parameters are ignored if this isn't the first object to be instatiated of the **AudioBase** (p. 3) descendants.

Parameters:

frequency is the playing frequency of the environment (in Hz)

refresh is the refresh rate of the environment (in Hz)

synchronous is true if the environment is synchronous

openalpp::AudioEnvironment::AudioEnvironment (int *frequency*, int *refresh* = -1) throw (bf InitError)

Constructor.

The parameters are ignored if this isn't the first object to be instatiated of the **AudioBase** (p. 3) descendants.

Parameters:

frequency is the playing frequency of the environment (in Hz)

refresh is the refresh rate of the environment (in Hz)

Member Function Documentation

DistanceModel openalpp::AudioEnvironment::GetDistanceModel () throw (FatalError)

Gets the distance model used in attenuation calculations.

Returns:

the model.

float openalpp::AudioEnvironment::GetDopplerFactor () throw (FatalError)
Gets the Doppler factor.

Returns:
Doppler factor.

float openalpp::AudioEnvironment::GetGain () throw (FatalError)
Gets the global gain.

Returns:
global gain

float openalpp::AudioEnvironment::GetSoundSpeed () throw (FatalError)
Get the speed of sound in the environment.

Returns:
speed of sound in length units per second.

void openalpp::AudioEnvironment::SetDistanceModel (DistanceModel *model*) throw (FatalError)
Sets the distance model used in attenuation calculations.

Parameters:
model is one of: None, InverseDistance, InverseDistanceClamped.

void openalpp::AudioEnvironment::SetDopplerFactor (float *factor*) throw (ValueError, FatalError)
Sets the Doppler factor.
This can be used to exaggerate, deemphasize or completely turn off the doppler effect.

Parameters:
factor has a default value of one.

void openalpp::AudioEnvironment::SetGain (float *gain*)
Sets global gain (volume).
The volume a source will be played at will be multiplied by this _after_ the attenuation calculations. Note: In todays's implementation on Linux, gain is clamped to [0.0,1.0]. This will be changed in future releases of OpenAL.

Parameters:
gain is the gain [0.0,...

void openalpp::AudioEnvironment::SetSoundSpeed (float *speed*) throw (ValueError,FatalError)

Sets the speed of sound in the environment.
This is used in doppler calculations.

Parameters:

speed is the speed of sound in length units per second.

The documentation for this class was generated from the following file:

- **audioenvironment.h**

B.6 openalpp::PositionedObject Class Reference

Virtual base class for positioned objects.
`#include <positionedobject.h>`

Public Methods

- virtual void **SetPosition** (float x, float y, float z)=0

  Set position.

- virtual void **GetPosition** (float &x, float &y, float &z) const=0

  Get position.

- virtual void **SetVelocity** (float vx, float vy, float vz)=0

  Set velocity.

- virtual void **GetVelocity** (float &vx, float &vy, float &vz) const=0

  Get velocity.

B.6.1 Detailed Description

Virtual base class for positioned objects.
(I.e. listeners and sources).

Member Function Documentation

virtual void openalpp::PositionedObject::GetPosition (float & *x*, float & *y*, float & em z) const `[pure virtual]`

Get position.

Parameters:

x x coordinate.

y y coordinate.

z z coordinate.

Reimplemented in **openalpp::Listener** (p. 11), and **openalpp::SourceBase** (p. 16). **virtual void openalpp::PositionedObject::GetVelocity (float & *vx*, float & *vy*, float & *vz*) const** `[pure virtual]`

Get velocity.

Parameters:

vx x member of velocity vector.

vy y member of velocity vector.

vz z member of velocity vector.

Reimplemented in **openalpp::Listener** (p. 11), and **openalpp::SourceBase** (p. 16). **virtual void openalpp::PositionedObject::SetPosition (float *x*, float *y*, float *z*)** `[pure virtual]`

Set position.

Parameters:

x x coordinate.

y y coordinate.

z z coordinate.

Reimplemented in **openalpp::Listener** (p. 11), and **openalpp::SourceBase** (p. 16). **virtual void openalpp::PositionedObject::SetVelocity (float *vx*, float *vy*, float *vz*)** `[pure virtual]`

Set velocity.

Parameters:

vx x member of velocity vector.

vy y member of velocity vector.

vz z member of velocity vector.

Reimplemented in **openalpp::Listener** (p. 11), and **openalpp::SourceBase** (p. 16).

The documentation for this class was generated from the following file:

- **positionedobject.h**

B.7 openalpp::Listener Class Reference

Class for listeners.

`#include <listener.h>`

Public Methods

- **Listener** ()

  Constructor.

- ~**Listener** ()

  Destructor.

- **Listener** (const Listener &listener)

  Copy constructor.

- **Listener** (float x, float y, float z, float directionx, float directiony, float directionz, float upx, float upy, float upz)

  Constructor.

- **Listener** (float x, float y, float z)

  Constructor.

- void **Select** ()

  Select this listener.

- bool **IsSelected** ()

  Check if this listener is currently selected.

- void **SetOrientation** (float directionx, float directiony, float directionz, float upx, float upy, float upz)

  Set the current orientation of this listener.

- void **GetOrientation** (float &directionx, float &directiony, float &directionz, float &upx, float &upy, float &upz) const

  Get the current orientation of this listener.

- Listener & **operator=** (const Listener &listener)

Assignment operator.

- void **SetPosition** (float x, float y, float z)
  Inherited from **PositionedObject** (p. 8).
- void **GetPosition** (float &x, float &y, float &z) const
  Inherited from **PositionedObject** (p. 8).
- void **SetVelocity** (float vx, float vy, float vz)
  Inherited from **PositionedObject** (p. 8).
- void **GetVelocity** (float &vx, float &vy, float &vz) const
  Inherited from **PositionedObject** (p. 8).

B.7.1 Detailed Description

Class for listeners.

Constructor & Destructor Documentation

openalpp::Listener::Listener ()
Constructor.
Creates the listener at the default position. **openalpp::Listener::Listener (float *x*, float *y*, float *z*, float *directionx*, float *directiony*, float *directionz*, float *upx*, float *upy*, float *upz*)**
Constructor.
Creates the listener at the specified position and orientation.

Parameters:

x x coordinate

y y coordinate

z z coordinate

directionx x value of the direction vector

directiony y value of the direction vector

directionz z value of the direction vector

upx x value of the up vector

upy y value of the up vector

upz z value of the up vector

openalpp::Listener::Listener (float *x*, float *y*, float *z*)
Constructor.
Creates the listener at the specified position.

Parameters:

x x coordinate

y y coordinate

z z coordinate

Member Function Documentation

void openalpp::Listener::GetOrientation (float & *directionx*, float & *directiony*, float & *directionz*, float & *upx*, float & *upy*, float & *upz*) const
Get the current orientation of this listener.

Parameters:

directionx x value of the direction vector

directiony y value of the direction vector

directionz z value of the direction vector

upx x value of the up vector

upy y value of the up vector

upz z value of the up vector

bool openalpp::Listener::IsSelected ()
Check if this listener is currently selected.

Returns:
true if this listener is selected, false otherwise.

void openalpp::Listener::SetOrientation (float *directionx*, float *directiony*, float *dir ectionz*, float *upx*, float *upy*, float *upz*) Set the current orientation of this listener.

Parameters:

directionx x value of the direction vector

directiony y value of the direction vector

directionz z value of the direction vector

upx x value of the up vector

upy y value of the up vector

upz z value of the up vector

Listener& openalpp::Listener::operator= (const Listener & *listener*)
Assignment operator.

Parameters:
listener is the object to make a copy of.

The documentation for this class was generated from the following file:

- **listener.h**

B.8 openalpp::SourceBase Class Reference

Base class for sources.
`#include <sourcebase.h>`

Public Methods

- void **Play** ()

  Play the source.

- void **Pause** ()

  Pause the source.

- void **Stop** ()

  Stop the source.

- void **Rewind** ()

  Rewind the source.

- **SourceState GetState** () const

  Get the current state.

- void **SetLooping** (bool loop=true)

  Turn on/off looping.

- bool **IsLooping** () const

  Check whether the source is looping.

- void **SetDirection** (float directionx, float directiony, float directionz)

  Sets the direction of the source.

- void **GetDirection** (float &directionx, float &directiony, float &directionz) const

  Gets the direction of the source.

- void **MakeOmniDirectional** ()

  Makes the source omni-directional.

- void **SetSoundCone** (float innerangle, float outerangle=360.0, float outergain=0.0)

  Sets the sound cone parameters for a directional sound source.

- void **GetSoundCone** (float &innerangle, float &outerangle, float &outergain) const

  Gets the sound cone parameters.

- void **SetGain** (float gain)

  Sets gain (volume).

- float **GetGain** () const

  Gets the gain (volume).

- void **SetMinMaxGain** (float min=0.0, float max=1.0)

  Sets maximum and minimum gain this source will be played at.

- void **GetMinMaxGain** (float &min, float &max) const

  Gets maximum and minumum gain.

- void **SetAmbient** (bool ambient=true)

  Makes the source ambient (or makes it stop being ambient).

- bool **IsAmbient** () const

Check if the source is ambient.

- void **SetReferenceDistance** (float distance=1.0)
  Sets the reference distance for this source.
- float **GetReferenceDistance** () const
  Gets the reference distance.
- void **SetMaxDistance** (float distance)
  Sets the maximum distance.
- float **GetMaxDistance** () const
  Gets the maximum distance.
- void **SetRolloffFactor** (float factor=1.0)
  Sets the roll-off factor.
- float **GetRolloffFactor** () const
  Gets the roll-off factor.
- void **SetPitch** (float pitch=1.0)
  Sets the pitch.
- float **GetPitch** () const
  Gets the pitch.
- void **SetReverbScale** (float scale) throw (InitError,ValueError)
  Set reverb scale for this source.
- void **SetReverbDelay** (float delay) throw (InitError,ValueError)
  Set reverb delay for this source.
- float **GetReverbDelay** () throw (InitError)
  Get reverb delay for this source.
- float **GetReverbScale** () throw (InitError)
  Get reverb scale for this source.

- ALuint **Link** (const SourceBase &source) throw (MemoryError)

  Link this source to another.

- void **Unlink** (const SourceBase &source) throw (NameError)

  Unlink this source from another.

- void **Unlink** (const ALuint name) throw (NameError)

  Unlink this source from another.

- void **UnlinkAll** ()

  Unlink all sources from this.

- ALuint **GetAlSource** () const

  Returns the OpenAL name of the source.

- void **SetPosition** (float x, float y, float z)

  Inherited from **PositionedObject** (p. 8).

- void **GetPosition** (float &x, float &y, float &z) const

  Inherited from **PositionedObject** (p. 8).

- void **SetVelocity** (float vx, float vy, float vz)

  Inherited from **PositionedObject** (p. 8).

- void **GetVelocity** (float &vx, float &vy, float &vz) const

  Inherited from **PositionedObject** (p. 8).

- SourceBase & **operator=** (const SourceBase &sourcebase)

  Assignment operator.

- **~SourceBase** ()

  Destructor.

Protected Methods

- **SourceBase** () throw (MemoryError,NameError)

  Constructor.

- **SourceBase** (float x, float y, float z) throw (MemoryError,NameError)

  Constructor.

- **SourceBase** (const SourceBase &sourcebase)

  Copy constructor.

Protected Attributes

- ALuint **sourcename_**

  OpenAL name for this source.

B.8.1 Detailed Description

Base class for sources.

This class holds functions for playing, setting position etc. However, it cannot be instantiated, instead a source of either type (**GroupSource** (p. 23) or **Source** (p. 25)) should be created.

Constructor & Destructor Documentation

openalpp::SourceBase::SourceBase (float *x*, float *y*, float *z*) throw (MemoryError,NameError) `[protected]`

Constructor.

Parameters:

x x coordinate.

y y coordinate.

z z coordinate.

Member Function Documentation

ALuint openalpp::SourceBase::GetAlSource () const
Returns the OpenAL name of the source.
Can be used to directly modify the source with OpenAL functions.

Returns:
Identifier for the source.

void openalpp::SourceBase::GetDirection (float & *directionx*, float & *directiony*, float & *directionz*) const
Gets the direction of the source.

Parameters:
direction x x value of the direction vector.

direction y y value of the direction vector.

direction z z value of the direction vector.

float openalpp::SourceBase::GetGain () const
Gets the gain (volume).

Returns:
gain.

float openalpp::SourceBase::GetMaxDistance () const
Gets the maximum distance.

Returns:
maximum distance.

void openalpp::SourceBase::GetMinMaxGain (float & *min*, float & *max*) const
Gets maximum and minumum gain.

Parameters:
min is minimum gain.

max is maximum gain.

float openalpp::SourceBase::GetPitch () const
Gets the pitch.

Returns:
pitch.

float openalpp::SourceBase::GetReferenceDistance () const
Gets the reference distance.

Returns:
reference distance.

float openalpp::SourceBase::GetReverbDelay () throw (InitError)
Get reverb delay for this source.

Returns:
the delay.

float openalpp::SourceBase::GetReverbScale () throw (InitError)
Get reverb scale for this source.

Returns:
the scale.

float openalpp::SourceBase::GetRolloffFactor () const
Gets the roll-off factor.

Returns:
rolloff factor.

void openalpp::SourceBase::GetSoundCone (float & *innerangle*, float & *outerangle*, floa t & *outergain*) const
Gets the sound cone parameters.

Parameters:
innerangle specifies the inner cone.

outerangle specifies the outer cone.

outergain specifies the gain outside the outer cone.

SourceState openalpp::SourceBase::GetState () const
Get the current state.

Returns:
one of Initial,Playing,Paused,Stopped

bool openalpp::SourceBase::IsAmbient () const
Check if the source is ambient.

Returns:
true if the source is ambient, false otherwise.

bool openalpp::SourceBase::IsLooping () const
Check whether the source is looping.

Returns:
true if it's looping, false otherwise.

ALuint openalpp::SourceBase::Link (const SourceBase & *source*) throw (MemoryError)
Link this source to another.
This causes calls to **Play**() (p. 13), **Pause**() (p. 13), **Stop**() (p. 13) and **Rewind**() (p. 13) (on this source) t o be applied to all sources this has been linked to, synchronously.

Parameters:
source is the source to link to.

Returns:
identifier for the linked source. This is also the OpenAL name for it.

void openalpp::SourceBase::MakeOmniDirectional ()
Makes the source omni-directional.
The same effect can be achieved by calling SetDirection(0,0,0) **void openalpp::SourceBase::SetAmbient (bool *ambient* = true)**
Makes the source ambient (or makes it stop being ambient).
This function _will_ change the source's position, direction and roll-off factor.

Parameters:
ambient is true if the source should be ambient, false otherwise.

void openalpp::SourceBase::SetDirection (float *directionx*, float *directiony*, float *directionz*)
Sets the direction of the source.

Parameters:
direction x x value of the direction vector.

direction y y value of the direction vector.

direction z z value of the direction vector.

void openalpp::SourceBase::SetGain (float *gain*)
Sets gain (volume).
The volume a source will be played at will be multiplied by this _after_ the attenuation calculations. Note: In todays's implementation on Linux, gain is clamped to [0.0,1.0]. This will be changed in future releases of OpenAL.

Parameters:
gain is the gain [0.0,...

void openalpp::SourceBase::SetLooping (bool *loop* = true)
Turn on/off looping.

Parameters:
loop is true if the source should loop, false otherwise.

void openalpp::SourceBase::SetMaxDistance (float *distance*)
Sets the maximum distance.
This is used in attenuation calculations, if the distance model is Inverse-DistanceClamped.

Parameters:
distance is the maximum distance.

void openalpp::SourceBase::SetMinMaxGain (float *min* = 0.0, float *max* = 1.0)
Sets maximum and minimum gain this source will be played at.
I.e. the gain will be clamped to these values.

Parameters:
min is minimum gain.

max is maximum gain.

void openalpp::SourceBase::SetPitch (float *pitch* = 1.0)
Sets the pitch.
1.0 is normal. Each reduction by 50% equals a reduction by one octave.

Parameters:
pitch is the pitch (0.0,1.0].

void openalpp::SourceBase::SetReferenceDistance (float *distance* = 1.0)
Sets the reference distance for this source.
The reference distance is used in attenuation calculations.

Parameters:
distance is the reference distance.

void openalpp::SourceBase::SetReverbDelay (float *delay*) throw (Init-Error,Value -Error) Set reverb delay for this source.
AudioEnvironment::InitiateReverb() must be called before using this. This is how many seconds back in time the echo will be.

Parameters:

delay is the delay [0.0-2.0] in seconds.

void openalpp::SourceBase::SetReverbScale (float *scale*) throw (Init-Error,Value -Error)

Set reverb scale for this source.

This is simply the scale of the "echo." AudioEnvironment::InitiateReverb() must be called before this.

Parameters:

scale is the reverb scale [0.0-1.0].

void openalpp::SourceBase::SetRolloffFactor (float *factor* = 1.0)

Sets the roll-off factor.

This is used in distance attenuation calculations.

Parameters:

factor is the rolloff factor.

void openalpp::SourceBase::SetSoundCone (float *innerangle*, float *outerangle* = 360.0, fl oat *outergain* = 0.0)

Sets the sound cone parameters for a directional sound source.

This function has no effect on omni-directional sources. Two cones, with the top at the source, and turned the same direction as the source, are defined by this. Inside the inner cone (specified by innerangle), sound will be played at full volume (attenuated by distance), and outside the outer cone (specified b y outerangle) sound will be played at the volume specified by outergain. Between these areas, the sound volume will be interpolated between normal gain and outergain.

Parameters:

innerangle specifies the inner cone.

outerangle specifies the outer cone.

outergain specifies the gain outside the outer cone.

void openalpp::SourceBase::Unlink (const ALuint *name*) throw (Name-Error)

Unlink this source from another.

Parameters:

name is the name of the source to unlink.

void openalpp::SourceBase::Unlink (const SourceBase & *source*) throw (NameError)

Unlink this source from another.

Parameters:

source is the source to unlink.

The documentation for this class was generated from the following file:

- **sourcebase.h**

B.9 openalpp::GroupSource Class Reference

Class for group sources.

`#include <groupsource.h>`

Public Methods

- **GroupSource** (float x=0.0, float y=0.0, float z=0.0) throw (NameError)

  Constructor.

- void **Play** () throw (InitError,FileError)

  Same as **SourceBase::Play** (p. 13)*, except that this mixes the sources in the group if it haven't been done yet.*

- void **MixSources** (unsigned int frequency=22050) throw (InitError,FileError,FatalError,ValueError,MemoryError)

  Mix all added sources into one.

- ALuint **IncludeSource** (**Source** *source) throw (ValueError)

  Includes a source in the group.

- void **ExcludeSource** (const **Source** &source) throw (NameError)

  Removes a source from the group.

- void **ExcludeSource** (ALuint source) throw (NameError)

  Removes a source from the group.

- **GroupSource** (const GroupSource &groupsource)

  Copy constructor.

- ∼**GroupSource** ()

Destructor.

- GroupSource & **operator=** (const GroupSource &groupsource)

  Assignment operator.

B.9.1 Detailed Description

Class for group sources.

Used for mixing together several sources _before_ they are played. This can be used to play more sounds with less processing power. Of course the prob lem is that the sources cannot be moved separately.

Constructor & Destructor Documentation

openalpp::GroupSource::GroupSource (float *x* = 0.0, float *y* = 0.0, float *z* = 0.0) t hrow (NameError)

Constructor.

Creates the group source at the specified position.

Parameters:

x x coordinate.

y y coordinate.

z z coordinate.

Member Function Documentation

void openalpp::GroupSource::ExcludeSource (ALuint *source*) throw (NameError)

Removes a source from the group.

This will of course require the remaining sources to be mixed again.

Parameters:

source is the identifier of the source to exclude.

void openalpp::GroupSource::ExcludeSource (const Source & *source*) throw (NameError)

Removes a source from the group.

This will of course require the remaining sources to be mixed again.

Parameters:

source is the source to exclude.

ALuint openalpp::GroupSource::IncludeSource (Source * *source*) throw (ValueEr ror) Includes a source in the group.

Returns an identifier that can be used as an argument to **Exclude-Source**() (p. 24). This identifier is also the OpenAL name for the included source.

Parameters:

source is (a pointer to) the source to include.

Returns:

identifier for the source.

void openalpp::GroupSource::MixSources (unsigned int *frequency* = 22050) throw (InitError,FileError,FatalError,ValueError,Memory-Error)

Mix all added sources into one.

This function will be called by **Play**() (p. 23), if sources have been included since the last time MixSamples() was called, so if you want the source to start playing as fast as possible after the **Play**() (p. 23)-call, **Mix-Sources**() (p. 25) should be called separately

Parameters:

frequency is the frequency that will be used when mixing.

The documentation for this class was generated from the following file:

- **groupsource.h**

B.10 openalpp::Source Class Reference

Class for "normal" sources.

`#include <source.h>`

Public Methods

- **Source** (float x=0.0, float y=0.0, float z=0.0)

  Constructor.

- **Source** (const char *filename, float x=0.0, float y=0.0, float z=0.0)

  Constructor.

- **Source** (const **Sample** &buffer, float x=0.0, float y=0.0, float z=0.0)

  Constructor.

- **Source** (const **Stream** &stream, float x=0.0, float y=0.0, float z=0.0)

  Constructor.

- **Source** (const Source &source)

  Copy constructor.

- ∼**Source** ()

  Destructor.

- void **SetSound** (const char ∗filename)

  Create a buffer for the source and load a file into it.

- void **SetSound** (const **Sample** &buffer)

  Sets a new buffer for the source.

- void **SetSound** (const **Stream** &stream)

  Sets a new (streamed) buffer for the source.

- const **SoundData** & **GetSound** () const

  Gets the buffer associated with the source.

- void **Play** (const char ∗filename)

  Play a file on the source.

- void **Play** (const **Sample** &buffer)

  Play a buffer on the source.

- void **Play** (const **Stream** &stream)

  Play a stream on the source.

- void **Play** ()

  Play this source.

- void **Stop** ()

  Stop this source.

- bool **IsStreaming** ()

  Check if the source is streaming.

- Source & **operator=** (const Source &source)

  Assignment operator.

B.10.1 Detailed Description

Class for "normal" sources.
This is used for standard OpenAL sources, whether streaming or not.

Constructor & Destructor Documentation

openalpp::Source::Source (float *x* = 0.0, float *y* = 0.0, float *z* = 0.0)
Constructor.
Creates the source with the specified position.

Parameters:

x x coordinate.

y y coordinate.

z z coordinate.

openalpp::Source::Source (const char * *filename*, float *x* = 0.0, float *y* = 0.0, flo at *z* = 0.0)
Constructor.
Creates the source and a buffer with the specified file.

Parameters:

filename is the name of the file.

openalpp::Source::Source (const Sample & *buffer*, float *x* = 0.0, float *y* = 0.0, f loat *z* = 0.0)
Constructor. Creates the source with the specified buffer.

Parameters:

buffer is the buffer to use.

openalpp::Source::Source (const Stream & *stream*, float *x* = 0.0, float *y* = 0.0, f loat *z* = 0.0)
Constructor.
Creates the source with the specified stream.

Parameters:

stream is the stream to use.

Member Function Documentation

const SoundData& openalpp::Source::GetSound () const
Gets the buffer associated with the source.

Returns:
the buffer.

bool openalpp::Source::IsStreaming ()
Check if the source is streaming.

Returns:
true if the source is streaming, false otherwise.

void openalpp::Source::Play ()
Play this source.
This is only here, because the above Play(...) hides **SourceBase::Play**() (p. 13) Reimplemented from **openalpp::SourceBase** (p. 13).

void openalpp::Source::Play (const Stream & *stream*)
Play a stream on the source.
This will change the source's buffer.

Parameters:
stream is the stream to play.

void openalpp::Source::Play (const Sample & *buffer*)
Play a buffer on the source.
This will change the source's buffer.

Parameters:
buffer is the buffer to play.

void openalpp::Source::Play (const char * *filename*)
Play a file on the source.
This will change the source's buffer.

Parameters:
filename is the name of the file to play.

void openalpp::Source::SetSound (const Stream & *stream*)
Sets a new (streamed) buffer for the source.
The source should _not_ be playing when doing this.

Parameters:
stream is the new buffer.

void openalpp::Source::SetSound (const Sample & *buffer*)

Sets a new buffer for the source.
The source should _not_ be playing when doing this.

Parameters:

buffer is the new buffer.

void openalpp::Source::SetSound (const char * *filename*)

Create a buffer for the source and load a file into it.
The source should _not_ be playing when doing this.

Parameters:

filename is the name of the file.

void openalpp::Source::Stop ()

Stop this source.
This is needed here for streaming sources...
Reimplemented from **openalpp::SourceBase** (p. 13).
The documentation for this class was generated from the following file:

- **source.h**

B.11 openalpp::SoundData Class Reference

Base class for sound data.
`#include <sounddata.h>`

Public Methods

- ALuint **GetAlBuffer** () const

  Get the OpenAL name for the buffer.

- **SoundData** () throw (NameError,InitError)

  Constructor.

- **SoundData** (const SoundData &sounddata)

  Copy constructor.

- **~SoundData** ()

  Destructor.

- SoundData & **operator=** (const SoundData &sounddata)

  Assignment operator.

Protected Attributes

- **SoundBuffer * buffer_**

  Handles generation and deletion of OpenAL buffers correctly.

- ALuint **buffername_**

  OpenAL name for the buffer.

B.11.1 Detailed Description

Base class for sound data.

Member Function Documentation

ALuint openalpp::SoundData::GetAlBuffer () const
Get the OpenAL name for the buffer.

Returns:
the OpenAL name.

The documentation for this class was generated from the following file:

- **sounddata.h**

B.12 openalpp::Sample Class Reference

Class for loading sampled files.
`#include <sample.h>`

Public Methods

- **Sample** (const char *filename) throw (FileError)

  Constructor.

- **Sample** (const Sample &sample)

  Copy constructor.

- std::string **GetFileName** () const

  Get file name of loaded file.

- Sample & **operator=** (const Sample &sample)

  Assignment operator.

B.12.1 Detailed Description

Class for loading sampled files.

Constructor & Destructor Documentation

openalpp::Sample::Sample (const char * *filename*) throw (FileError)
Constructor.

Parameters:
filename is name of file to load.

Member Function Documentation

std::string openalpp::Sample::GetFileName () const
Get file name of loaded file.

Returns:
file name.

The documentation for this class was generated from the following file:

- **sample.h**

B.13 openalpp::Stream Class Reference

Base class for **NetStream** (p. 34) and **InputDevice** (p. 33) .
`#include <stream.h>`

Public Methods

- **Stream** () throw (NameError)

  Default constructor.

- **Stream** (const Stream &stream)

  Copy constructor.

- Stream & **operator=** (const Stream &stream)

Assignment operator.

- ~**Stream** ()

  Destructor.

- void **Record** (ALuint sourcename)

  Start recording.

- void **Stop** (ALuint sourcename)

  Stop recording.

Protected Attributes

- SoundBuffer * **buffer2_**

  For double-buffering of sounds.

- **StreamUpdater** * **updater_**

B.13.1 Detailed Description

Base class for **NetStream** (p. 34) and **InputDevice** (p. 33) .
Used for audio streams.

Member Function Documentation

void openalpp::Stream::Record (ALuint *sourcename*)
Start recording.
I.e. start copying data to buffers.

Parameters:
sourcename is the (OpenAL) name of the source.

void openalpp::Stream::Stop (ALuint *sourcename*)
Stop recording.

Parameters:
sourcename is the (OpenAL) name of the source.

The documentation for this class was generated from the following file:

- **stream.h**

B.14 openalpp::InputDevice Class Reference

Class for handling input devices, like microphones.

`#include <inputdevice.h>`

Public Methods

- **InputDevice** ()

  Constructor.

- **InputDevice** (int device, unsigned int samplerate, unsigned int buffersize=1024, **SampleFormat** format=Mono16)

  Constructor.

- **InputDevice** (const InputDevice &input)

  Copy constructor.

- InputDevice & **operator=** (const InputDevice &input)

  Assignment operator.

- ~**InputDevice** ()

  Destructor.

B.14.1 Detailed Description

Class for handling input devices, like microphones.

Constructor & Destructor Documentation

openalpp::InputDevice::InputDevice (int *device*, unsigned int *samplerate*, unsigned int *buffersize* = 1024, SampleFormat *format* = Mono16)

Constructor.

Parameters:

device is the device to open. -1 for default input.

samplerate is the desired sample rate.

buffersize is the desired buffer size.

format is the desired sample format.

The documentation for this class was generated from the following file:

- **inputdevice.h**

B.15 openalpp::NetStream Class Reference

Class for handling streams through sockets.
`#include <netstream.h>`

Public Methods

- **NetStream** (ost::UDPSocket *socket, ost::TCPStream *controlsocket=NULL)

  Constructor.

- **NetStream** (ost::UDPSocket *socket, **SampleFormat** format, unsigned int frequency, unsigned int packetsize)

  Constructor.

- **NetStream** (const NetStream &stream)

  Copy constructor.

- ~**NetStream** ()

  Destructor.

- NetStream & **operator=** (const NetStream &stream)

  Assignment operator.

B.15.1 Detailed Description

Class for handling streams through sockets.

Preliminary tests indicate that packets smaller than ca 1 kb should not be used (tests were done with Mono8, 11025 Hz).

Constructor & Destructor Documentation

openalpp::NetStream::NetStream (ost::UDPSocket * *socket*, ost::TCPStream * *controlsocket* = NULL)

Constructor.

Parameters:

socket is the socket to stream data through.

controlsocket is an (optional) TCPStream that can be used to send information about the stream. The constructor will begin with trying to read SampleForm at, frequency, and packetsize. The sender can also use the control socket to send "EXIT" when it's run out of data to send. If this parame ter is not given, defaults will be used (format=Mono8, frequency=11025, packetsize=1024).

openalpp::NetStream::NetStream (ost::UDPSocket * *socket*, SampleFormat *format* , unsigned int *frequency*, unsigned int *packetsize*)

Constructor.

Parameters:

socket is the socket to stream data through.

format is the format the data will be in.

frequency is the frequency of the sound.

packetsize is the size of the packets the sound will be sent in.

The documentation for this class was generated from the following file:

- **netstream.h**

B.16 openalpp::Error Class Reference

Error (p. 35) class for throwing.

`#include <error.h>`

Public Methods

- **Error** ()

  Constructor.

- **Error** (const char *description)

  Constructor.

- **Error** (const Error &error)

  Copy constructor.

- std::ostream & **Put** (std::ostream &stream) const

  Function used for printing.

Protected Attributes

- std::string **errorstring_**

  A description of the error.

B.16.1 Detailed Description

Error (p. 35) class for throwing.

The descendants of this class are different error types, and the exact error can be displayed by using "cout << error;" where error is an instance of **Error** (p. 35) (or one of its descendants)

Constructor & Destructor Documentation

openalpp::Error::Error () `[inline]`

Constructor.

Will use a default error message. **openalpp::Error::Error (const char * *description*)** `[inline]`

Constructor.

Parameters:

description is error message to use.

Member Function Documentation

std::ostream& openalpp::Error::Put (std::ostream & *stream*) const

Function used for printing.

Parameters:

stream is stream to print to

Returns:

the stream with the error message appended.

The documentation for this class was generated from the following file:

- **error.h**

B.17 openalpp::FatalError Class Reference

Fatal error.

`#include <error.h>`

Public Methods

- **FatalError** (const char *description)

  Constructor.

B.17.1 Detailed Description

Fatal error.
Caused by error in implementation, corrupted memory etc.

Constructor & Destructor Documentation

openalpp::FatalError::FatalError (const char * *description*) `[inline]`
Constructor.

Parameters:
description is error message to use.

The documentation for this class was generated from the following file:

- **error.h**

B.18 openalpp::FileError Class Reference

File error.
`#include <error.h>`

Public Methods

- **FileError** (const char *description)

  Constructor.

B.18.1 Detailed Description

File error.
Caused by wrong file permissions, missing files etc.

Constructor & Destructor Documentation

openalpp::FileError::FileError (const char * *description*) `[inline]`
Constructor.

Parameters:
description is error message to use.

The documentation for this class was generated from the following file:

- **error.h**

B.19 openalpp::InitError Class Reference

Init error.
`#include <error.h>`

Public Methods

- **InitError** (const char *description)
  Constructor.

B.19.1 Detailed Description

Init error.
Caused by trying to do actions without proper initialization.

Constructor & Destructor Documentation

openalpp::InitError::InitError (const char * *description*) `[inline]`
Constructor.

Parameters:
description is error message to use.

The documentation for this class was generated from the following file:

- **error.h**

B.20 openalpp::MemoryError Class Reference

Memory error.
`#include <error.h>`

Public Methods

- **MemoryError** (const char *description)

  Constructor.

B.20.1 Detailed Description

Memory error.
Caused by insufficient memory etc.

Constructor & Destructor Documentation

openalpp::MemoryError::MemoryError (const char * *description*) `[inline]`
Constructor.

Parameters:
description is error message to use.

The documentation for this class was generated from the following file:

- **error.h**

B.21 openalpp::NameError Class Reference

Name error.
`#include <error.h>`

Public Methods

- **NameError** (const char *description)

  Constructor.

B.21.1 Detailed Description

Name error.
Caused by invalid (OpenAL) names.

Constructor & Destructor Documentation

openalpp::NameError::NameError (const char * *description*) `[inline]`
Constructor.

Parameters:
description is error message to use.

The documentation for this class was generated from the following file:

- **error.h**

B.22 openalpp::ValueError Class Reference

Value error.
`#include <error.h>`

Public Methods

- **ValueError** (const char *description)
  Constructor.

B.22.1 Detailed Description

Value error.
Caused by values out of range etc.

Constructor & Destructor Documentation

openalpp::ValueError::ValueError (const char * *description*) `[inline]`
Constructor.

Parameters:
description is error message to use.

The documentation for this class was generated from the following file:

- **error.h**

C A simple example

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

int main() {
  // Create a source, and load the file example.wav into it.
  Source source(``example.wav'');

  // Move the source to be in front of, and a little to the
  // right of the listener.
  source.SetPosition(5,0,-15);

  // Play the loaded sound in the source.
  source.Play();

  sleep(5);

  // Create another source.
  Source source2;

  // Move it to be to the left of the listener.
  source2.SetPosition(-10,0,0);

  // Play the file example2.wav in the source.
  source2.Play(``example2.wav'');

  sleep(5);

  return 0;
}
```