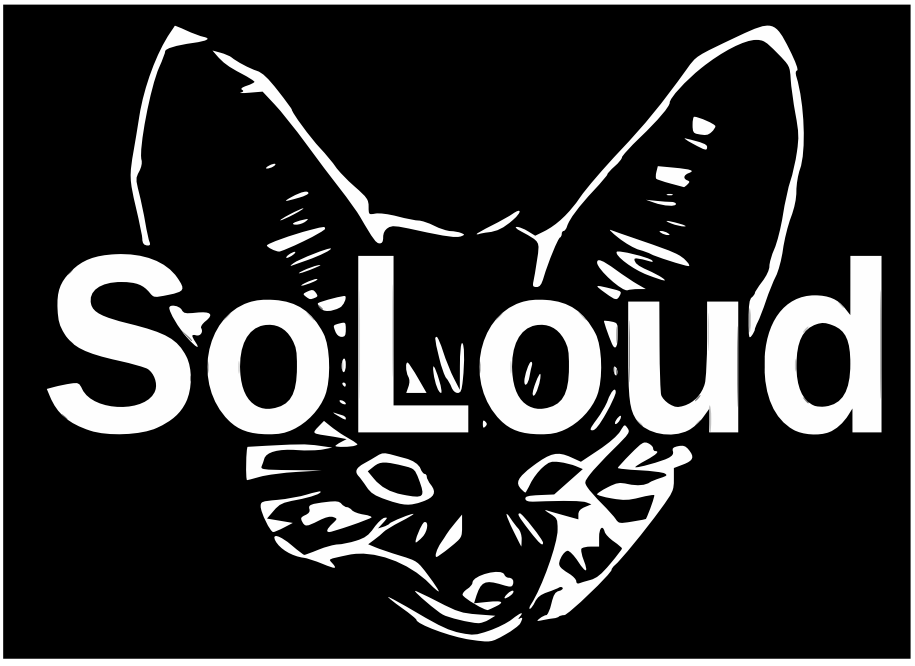
SoLoud Audio Engine

Jari Komppa

May 21, 2014



Contents

1 Introduction

SoLoud is an easy to use, free, portable c/c++ audio engine for games.

1.1 How Easy?

The engine has been designed to make simple things easy, while not making harder things impossible. Here's a code snippet that initializes the library, loads a sample and plays it:

```
// Declare some variables
SoLoud::Soloud soloud; // Engine core
SoLoud::Wav sample;    // One sample

// Initialize SoLoud (automatic back-end selection)
soloud.init();

sample.load("pew_pew.wav"); // Load a wave file
soloud.play(sample);        // Play it
```

The primary form of use the interface is designed for is "fire and forget" audio. In many games, most of the time you don't need to modify a sound's parameters on the fly - you just find an event, like an explosion, and trigger a sound effect. SoLoud handles the rest.

If you need to alter some aspect of the sound after the fact, the "play" function returns a handle you can use. For example:

```
int handle = soloud.play(sample);         // Play the sound
soloud.setVolume(handle, 0.5f);           // Set volume; 1.0f is "normal"
soloud.setPan(handle, -0.2f);             // Set pan; -1 is left, 1 is right
soloud.setRelativePlaySpeed(handle, 0.9f);// Play a bit slower; 1.0f is normal
```

If the sound doesn't exist anymore (either it's ended or you've played so many sounds at once it's channel has been taken over by some other sound), the handle is still safe to use - it just doesn't do anything.

There's also a pure "C" version of the whole API which can even be used from non-c languages by using SoLoud as an DLL.

1.2 How Free?

SoLoud is released under the ZLib/LibPNG license. That means, among other things, that:

- You can use it in free or commercial applications as much as you want.
- You can modify it. (But you don't need to).
- You don't need to give the changes back. (But you can).
- You don't need to release the source code. (But you can).
- You don't need to add a splash screen. (But you can).
- You don't need to mention it in your printed manual. (But you can).

Basically the only things the license forbids are suing me, or claiming that you made SoLoud. If you redistribute the source code, the license needs to be there. But not with the binaries.

Parts of the SoLoud package were not made by me, and those either have a similar license, or more permissive (such as Unlicense, CC0, WTFPL or Public Domain).

1.3 How Powerful?

While SoLoud's usage has been designed to be very easy, it's still packed with powerful functionality. Some of the features include:

- Multiple voices, playing different or even the same sound multiple times on top of each other.
- Adjustable play speed, volume and pan.
- Faders for all of the attributes (fade out for 2 seconds, then stop, for instance).
- Filter interface and ready filters for low/high pass, echo, etc for real-time modification of audio.
- Mixing busses for grouping of audio into different uses and adjusting their attributes in one go.
- Gapless looping.
- Playing several ogg streams at once.
- Sound effects synthesizer.
- Modplug library capable of playing various multi-channel music formats (including mod, s3m, it, xm, mid, abc).
- Easy cleanup.

1.4 There's a Catch, Right?

SoLoud quite probably doesn't have all the features you'd find in a commercial library like FMOD or WWISE. There's no artist tools or engine integration. There's no 3d audio. Output is, currently, limited to stereo.

It quite probably isn't as fast. As of this writing, it has no specialized assembler optimizations, for any platform.

It definitely doesn't come with the support you get from a commercial library.

If you're planning to make a multi-million budgeted console game, this library is (probably) not for you. Feel free to try it though =)

2 Legal

SoLoud, like everything else, stands on the shoulders of giants; however, care has been taken to only incorporate source code that is under liberal licenses, namely ZLib/LibPNG, CC0 or public domain, or similar, like WTFPL or Unlicense, where you don't need to include mention of the code in your documentation or splash screens or any such nonsense.

2.1 SoLoud Proper

SoLoud proper is licensed under the ZLib/LibPNG license. The code is a clean-room implementation with no outside sources used.

```
SoLoud audio engine
Copyright (c) 2013 Jari Komppa

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.
```

2.2 OGG Support

The OGG support in the Wav and WavStream sound sources is based on stb_vorbis by Sean Barrett, and it's in the public domain. You can find more information (and latest version) at http://nothings.org/stb_vorbis/

2.3 Speech Synthesizer

The speech synth is based on rsynth by the late Nick Ing-Simmons (et al). He described the legal status as:

```
    This is a text to speech system produced by
    integrating various pieces of code and tables
    of data, which are all (I believe) in the
    public domain.
```

Since then, the rsynth source code has passed legal checks by several open source organizations, so it "should" be pretty safe.

The primary copyright claims seem to have to do with text-to-speech dictionary use, which I've removed completely.

I've done some serious refactoring, clean-up and feature removal on the source, as all I need is "a" free, simple speech synth, not a "good" speech synth. Since I've removed a bunch of stuff, this is probably safer public domain release than the original.

I'm placing my changes in public domain as well, or if that's not acceptable for you, then CC0: http://creativecommons.org/publicdomain/zero/1.0/.

The SoLoud interface files (soloud_speech.*) are under the same ZLib/LibPNG license as the other SoLoud bits.

2.4 Fast Fourier Transform (FFT)

FFT calculation is provided by a fairly simple implementation by Stephan M. Bernsee, under the Wide Open License:

```
COPYRIGHT 1996 Stephan M. Bernsee <smb [AT] dspdimension [DOT] com>

                        The Wide Open License (WOL)

Permission to use, copy, modify, distribute and sell this software and its
documentation for any purpose is hereby granted without fee, provided that
the above copyright notice and this license appear in all source copies.
THIS SOFTWARE IS PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY OF
ANY KIND. See http://www.dspguru.com/wol.htm for more information.
```

2.5 Sfxr

The sfxr sound effects synthesizer is by Tomas Pettersson, re-licensed under zlib/libpng license by permission.

```
Copyright (c) 2014 Jari Komppa
Based on code (c) by Tomas Pettersson, re-licensed under zlib by permission

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.
```

2.6 Libmodplug

The branch of libmodplug that is used in SoLoud was declared public domain. Authors include:

- Olivier Lapicque - olivierl@jps.net
- Markus Fick - webmaster@mark-f.de
- Adam Goode - adam@evdebs.org
- Jake Stine - air@divent.org
- Peter Grootswagers - pgrootswagers@planet.nl
- Marco Trillo - toad@arsystel.com
- Kenton Varda - temporal@gauge3d.org

with some fixes modifications by Jari Komppa, to work with SoLoud.

3 Quick Start

3.1 Download SoLoud

First, you need to download SoLoud sources. You can find the downloads on the `http://soloud-audio.com/download.html` page.

3.2 Add files to your project

You can go the lazy way and just add all of the sources to your project, or you can copy the things you need to a single directory and include those. You'll need the core files, and quite likely the wav files. If you need the speech synth, include those, too.

There may be a pre-built static library that you can use. You can use that, too. Note that the Windows DLL only exports the "C" API, which may not be what you want.

There's also a premake4 script if you want to go that way.

3.3 Include files

In order to use a certain feature of SoLoud, you need to include its include file. You might have, for instance:

```
#include "soloud.h"
#include "soloud_wav.h"
```

3.4 Variables

You need at least the SoLoud engine core, and one or more of the sound source variables. If you're using five different sound effect wav files, you need five SoLoud::Wav objects. You can play one object any number of times, even on top of each other.

Where to place these is up to you. Globals work, as do allocation from heap, including in a class as members, etc.

```
SoLoud::Soloud gSoloud;
SoLoud::Wav gWave;
```

3.5 Initialize SoLoud

In your application, once you have your framework up (for instance after your SDL_Init call), include a call to initialize SoLoud.

```
gSoloud.init();
```

The call has a bunch of optional parameters if you'd rather pick the replay back-end and its parameters yourself; the default should work for most cases.

3.6 Set up sound sources

This step varies from one sound source to another, but basically you'll load your wave files here.

```
gWave.load("pew_pew.wav");
```

3.7 Play sounds

Now you're ready to play the sounds. Place playing commands wherever you need sound to be played.

```
gSoloud.play(gWave);
```

Note that you can play the same sound several times, and it doesn't cut itself off.

3.8 Take control of the sound

You can adjust various things about the sound you're playing if you take the handle.

```
int x = gSoloud.play(gWave);
gSoloud.setPan(x, -0.2f);
```

Read the soloud.h header file (or this documentation) for further things you can do.

3.9 Cleanup

After you've done, remember to clean up. If you don't, the audio thread may do stupid things while the application is shutting down.

```
gSoloud.deinit();
```

3.10 Enjoy

And you're done!

4 Premake

SoLoud comes with a premake4 script. If you want to build SoLoud as static library, instead of including the source files in your project, this can be handy.

Premake can be downloaded from http://industriousone.com/premake.

Unfortunately, premake4 cannot magically figure out where your libraries may be installed, so you may have to edit the premake4.lua file. The lines to edit can be found at the very beginning of the file, with the following defaults:

```
local sdl_root       = "/libraries/sdl"
local portmidi_root  = "/libraries/portmidi"
local dxsdk_root     = "C:/Program␣Files␣(x86)/Microsoft␣..."
local portaudio_root = "/libraries/portaudio"
local openal_root    = "/libraries/openal"
```

You will most likely want to edit at least the sdl_root variable. After your edits, you can run premake4 to generate makefiles or the IDE project files of your preference, such as:

```
premake4 vs2010
```

The current version (4.3) supports codeblocks, codelite, vs2002, vs2003, vs2005, vs2008, vs2010, xcode3 and gnu makefiles (gmake). New version with at least vs2012 support is coming soon (as of this writing).

If you wish to use portmidi with the piano example, run premake with an additional parameter:

```
premake4 --with-portmidi vs2010
```

If you wish to include the xaudio2 back-end, use the parameter:

```
premake4 --with-xaudio2 vs2010
```

The back-end is not included by default due to the requirement of xaudio2.lib.

To include libmodplug, yet another parameter is needed:

```
premake4 --with-libmodplug vs2010
```

5 Concepts

5.1 Back end

SoLoud itself “only” performs audio mixing and some resource handling. For it to be useful, it needs one or more sound source and a back end. Some other audio systems use the term ‘sink’ for the back-ends. Examples of back-ends would be winmm, portaudio, wasapi and SDL audio. SoLoud comes with several back-ends, and is designed to make back-ends relatively easy to implement.

Different back-ends have different characteristics, such as how much latency they introduce.

5.2 Channel

One audio stream can contain one or more channels. Typical audio sources are either mono (containing one channel) or stereo (containing two channels), but surround sound audio sources may practically have any number of channels.

In module music (such as mod, s3m, xm, it), “channel” means one of the concurrent sounds played, regardless of speaker configuration. Confusing, yes.

5.3 Stream

SoLoud can play audio from several sound sources at once (or, in fact, several times from the same sound source at the same time). Each of these sound instances is a “stream”. The number of concurrent streams is limited, as having unlimited streams would cause performance issues, as well as lead to unnecessary clipping.

The default number of concurrent streams - maximum number of “voices” - is 64, but this can be adjusted via a defined constant in the soloud.h file. The hard maximum number is 4096, but if more are required, SoLoud can be modified to support more. But seriously, if you need more than 4096 sounds at once, you’re probably going to make some serious changes in any case.

If all channels are already playing and the application requests another sound to play, SoLoud finds the oldest sound and kills it. Since this may be your background music, you can protect channels from being killed by using the soloud.setProtect call.

5.4 Clipping

Audio hardware always has a limited dynamic range. If you think of a signed 16-bit variable, for instance, you can only store values from -32k to +32k in it; if you try to put values outside this range in, things tend to break. Same goes for audio.

SoLoud handles all audio as floats, but performs clipping before passing the samples out, so all values are in the -1..1 range. There’s two ways SoLoud can perform the clipping; the most straightforward is simply to set all values outside this range to the border value, or alternatively a roundoff calculation can be performed, which “compresses” the loud sounds. The more quiet sounds are largely unchanged, while the loud end gets less precision. The roundoff clipper is used by default.

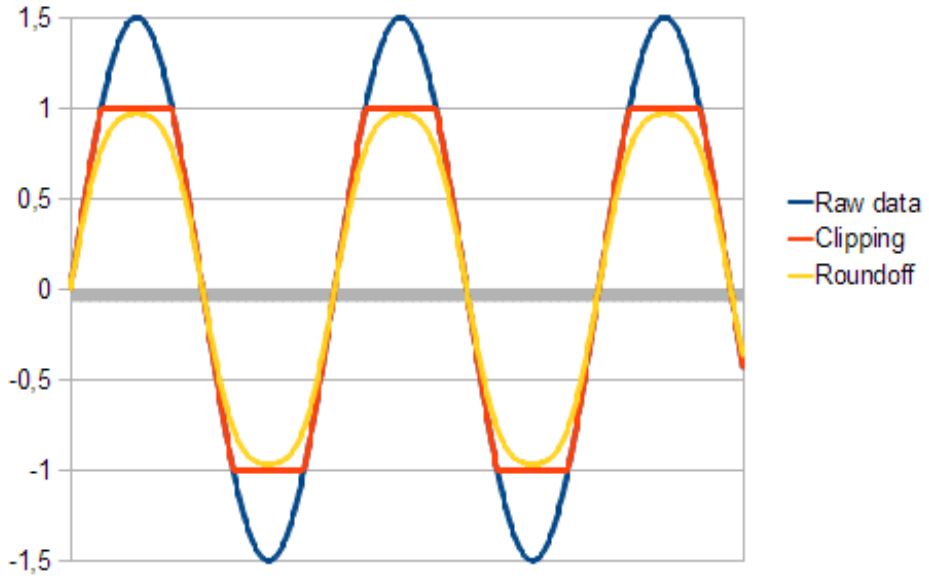


Figure 5.1: Results of different clippers

The roundoff clipper does, however, alter the signal and thus “damages” the sound. A more proper way of doing things would be to use the basic clipper and adjust the global volume to avoid clipping. The roundoff clipper, however, is easier to use.

5.5 Sample

The real world has continuous signals, which would require infinite amount of storage to store (unless you can figure out some kind of complicated mathematical formula that represents the signal). So, we store discrete samples of signals instead. These samples have traditionally been 8, 16 or 24 bit, but high-end audio is tending towards floating point samples.

SoLoud also uses floating point samples internally. First and foremost, it makes everything much simpler, and second, modern computing devices have become fast enough that this is not really a performance issue anymore.

Floating point samples also take more space than, for instance, 16 bit samples, but memory and storage sizes have also grown enough to make this a feasible approach. Nothing stops the audio sources from keeping data in a more “compressed” format and performing on-the-fly conversion to float, if memory requirements are a concern.

5.6 Sample Rate

The sample rate represents the number of samples used, per second. Typical sample rates are 8000Hz, 22050Hz, 44100Hz and 48000Hz. Higher the sample rates mean clearer sound, but also bigger files, more memory and higher processing power requirements.

5.7 Hz

Hertz, SI unit of frequency. 1Hz means “once per second”, 10Hz means “10 times per second”, and 192kHz means “192000 times per second”.

5.8 Play Speed

In addition to a base sample rate, which represents the “normal” playing speed, SoLoud includes a “relative play speed” option. This simply changes the sample rate. However, if you replace your sounds with something that has a different “base” sample rate, using the relative play speed will retain the effect of playing the sound slower (and lower) or faster (and higher).

5.9 Relative Play Speed

SoLoud lets you change the relative play speed of samples. Please note that asking for a higher relative play speed is more expensive than a lower one.

5.10 Resampling

SoLoud has to perform resampling when mixing. In an ideal case, all of the sources and the destination sample rate are the same, and no resampling is needed, but this is often not true.

Currently, SoLoud supports “point sample” resampling, which means it simply skips or repeats samples as needed, as well as “linear interpolation”, which calculates linear interpolation of samples.

Higher quality resamplers are planned.

5.11 Pan

Where the sound is coming from in the stereo sound, ranging from left speaker only to right speaker only. SoLoud uses an algorithm to calculate the left/right channel volume so that the overall volume is retained across the field. You can also set the left/right volumes directly, if needed.

5.12 Handle

SoLoud uses throwaway handles to control sounds. The handle is an integer, and internally tracks the channel and sound id, as well as an “uniqueness” value.

If you try to use a handle after the sound it represents has stopped, the operation is quietly discarded (or if you’re requesting information, some kind of generic value is returned). You can also query the validity of a handle.

5.13 Sound Source and Instance

SoLoud uses two kinds of classes for the sounds. Sound sources contain all the information related to the sound in question, such as wave sample data, while sound instances contain information about an “instance” of the sound.

As an analogue, if you think of an old vinyl record, the sound source is the record, and you can put as many playheads - the instances - on the record. All of the playheads can also move at different speeds, output to a different pan position and volume, as well as different filter settings.

5.14 Latency

Audio latency generally means the time it takes from triggering a sound to the sound actually coming out of the speakers. The smaller the latency, the better.

Unfortunately, there's always some latency. The primary source of latency (that a programmer can have any control over) is the size of audio buffer. Generally speaking, the smaller the buffer, the lower the latency, but at the same time, the smaller the buffer, the more likely the system hits buffer underruns (ie, the play head marches on but there's no data ready to be played) and the sound breaks down horribly.

Assuming there's no other sources of latency (and there quite likely is), with 2048 sample buffer and 44100Hz playback, the latency is around 46 milliseconds, which is tolerable in most cases. A 100ms latency is already easily noticeable.

5.15 Filter

Audio streams can also be modified on the fly for various effects. Typical uses are different environmental effects such as echoes or reverb, or low pass (bassy sound) / high pass (tinny sound) filters, but basically any kind of modification can be done; the primary limitations are processor power, imagination, and developer's skill in digital signal processing.

SoLoud lets you hook several filters to a single audio stream, as well as to the global audio output.

5.16 Mixing Bus

In addition to mixing audio streams together at the "global" level, SoLoud includes mixing busses which let you mix together groups of audio streams. These serve several purposes.

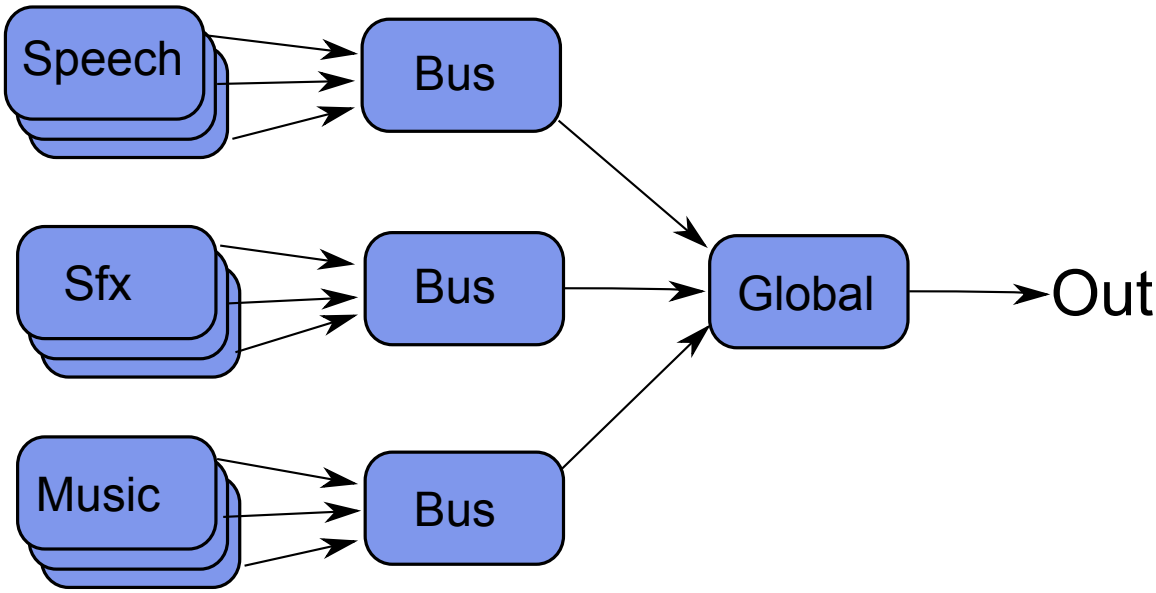


Figure 5.2: Mix busses concept

The most typical use would be to let the user change the volume of different kinds of audio sources - music, sound effects, speech. In this case, you would have one mixing bus for each of

these audio source groups, and simply change the volume on the mixing bus, instead of hunting down every sound separately.

When using environmental effects filters, you most likely won't want the background music to get filtered; the easiest way to handle this is to apply the filters to the mixing bus that plays the sound effects. This will also save on processing power, as you don't need to apply the environmental audio filters on every sound effect separately.

It's also possible that you have some very complex audio sources, such as racing cars. In this case it makes sense to place all the audio streams that play from one car into a mixing bus, and then adjust the panning (or, eventually, 3d position) of the mixing bus.

6 Frequently Asked Questions

6.1 What does it play?

Currently, SoLoud includes support for uncompressed 8 and 16 bit RIFF Wav files, as well as Ogg Vorbis files. Both of these only support a limited feature set of said formats, so you may experience some issues with strange files.

Additionally, SoLoud comes with a speech synthesizer and a retro sound effect synthesizer Sfxr.

Finally, SoLoud includes libmodplug, through which it can play 669, abc, amf, ams, dbm, dmf, dsm, far, it, j2b, mdl, med, mid, mod, mt2, mtm, okt, pat, psm, ptm, s3m, stm, ult, umx, xm, as well as wider support for wav files than the stand-alone wav audio source. (Due to the size of libmodplug, SoLoud can be compiled without it).

The interface for audio sources is relatively simple, so new formats and noise generators, as well as audio filters, can be made.

An example sin/saw/triangle/square generator is also available, as part of the "piano" example.

6.2 What dependencies does it have?

There's no external library dependencies (apart from stdlib). However, to get audio out of your speakers, a back-end is needed. Back-ends that currently exist include SDL, windows multimedia, oss and portaudio, and SoLoud has been designed so that making new back-ends would be as painless as possible.

6.3 Is there a DLL / C-Interface?

Yes! This DLL can be used from non-c++ environments through the "C" interface.

6.4 What's the animal in the logo?

A fennec fox. Google it. They're cute!

6.5 Is there a mailing list?

There's a google group, at `http://groups.google.com/d/forum/soloud`

Main development occurs on GitHub, at `https://github.com/jarikomppa/soloud` and the issue tracker is in use.

Finally, there's #soloud on ircnet, if you want to pop by.

6.6 Are these real questions?

Surprisingly, yes.

7 Examples

SoLoud package comes with a few simple examples. These can be found under the 'demos' directory. Pre-built binaries for Windows can also be found in the 'bin' directory.

7.1 simplest

The simplest example initializes SoLoud, and uses the speech synthesizer to play some sound. Once the sound has finished, the application cleans up and quits.

This example also uses SoLoud's cross-platform thread library to sleep while waiting for the sound to end.

7.2 multimusic

The multimusic example loads two OGG music loops as well as sound effects. You can use the keyboard keys 1 through 0 for various effects:

Key	Effect
1	Play random sfxr "explosion" preset
2	Play random sfxr "blip" preset
3	Play random sfxr "coin" preset
4	Play random sfxr "hurt" preset
5	Play random sfxr "jump" preset
6	Play random sfxr "laser" preset
7	Fade music 1 in and music 2 out
8	Fade music 2 in and music 1 out
9	Fade music relative play speed way down
0	Fade music relative play speed to normal

7.3 piano

This example is a simple implementation of a playable instrument. The example also includes a simple waveform generator (soloud_basicwave.cpp/h), which can produce square, saw, sine and triangle waves. If compiled to use portmidi, you can also use a midi keyboard to drive the example.

Key(s)	Effect
1234..	Play notes ("black keys")
qwer..	Play notes ("white keys")
asdf..	Select waveform

zxcv..	Selects filters.
l	Lo-fi filter

Speech synthesizer and on-screen text describe what different keys do when pressed. Have fun experimenting!

7.4 mixbusses

The mixbusses example demonstrates the use of mixing busses. You can use “qw”, “as” and “zx” keys to adjust volume of different busses.

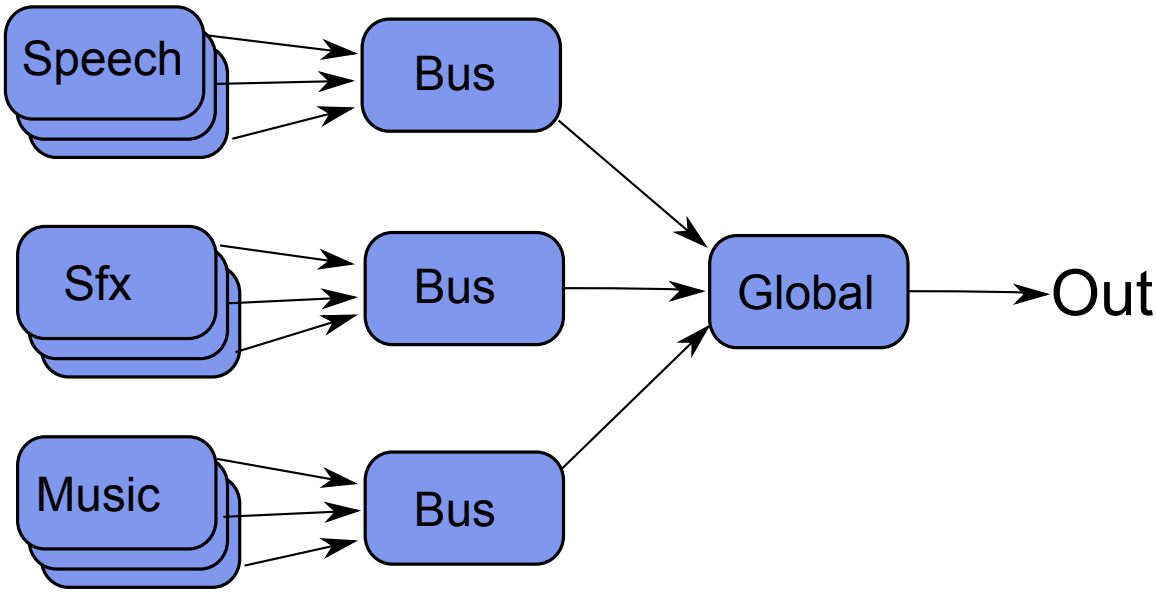


Figure 7.1: Mix busses concept

In case of this example, only one “music” and one “sfx” is played, but the idea is still the same.

7.5 env

The env demo is a non-interactive demo of how SoLoud could be used to play environmental audio.

8 “C”-api / DLL

In order to support non-c++ environments, SoLoud also has a “C” API.

All of the existing interfaces can be used via the “C” API, but features that require extending SoLoud are not available.

8.1 Codegen

The API is automatically generated from the c++ sources via the codegen tool that is part of the SoLoud sources. In most cases you won’t need to use the codegen yourself.

The codegen tool parses the SoLoud headers and generates the needed headers and wrapper cpp code, as well as the DLL .def file.

8.2 API

The “C” API mirrors the c++ API.

If the c++ API functions have default parameters, two functions are generated: one without the default parameters, and one with. The one where you can change the default parameters is post-fixed Ex, such as Soloud_init and Soloud_initEx.

As an example, here’s a simple example in the C++ api:

```
SoLoud::Soloud soloud;
SoLoud::Speech speech;

speech.setText("Hello␣c++␣api");

soloud.init(SoLoud::Soloud::CLIP_ROUNDOFF |
            SoLoud::Soloud::ENABLE_VISUALIZATION)

soloud.setGlobalVolume(4);
soloud.play(speech);

// ...

soloud.deinit();
```

Converted to the “C” API, this becomes:

```
Soloud *soloud = Soloud_create();
Speech *speech = Speech_create();

Speech_setText(speech, "Hello␣c-api");

Soloud_initEx(soloud, SOLOUD_CLIP_ROUNDOFF | SOLOUD_ENABLE_VISUALIZATION,
              SOLOUD_AUTO, SOLOUD_AUTO, SOLOUD_AUTO);

Soloud_setGlobalVolume(soloud, 4);
Soloud_play(soloud, speech);
```

```
// ...

Soloud_deinit(soloud);

Speech_destroy(speech);
Soloud_destroy(soloud);
```

9 Core: Basics

9.1 SoLoud::Soloud Object

In order to use SoLoud, you have to create a SoLoud::Soloud object. The object must be cleaned up or destroyed before your back-end is shut down; the safest way to do this is to call soloud.deinit() manually before terminating.

The object may be global, member variable, or even a local variable, it can be allocated from the heap or the stack, as long as the above demand is met. If the back-end gets destroyed before the back-end clean-up call is made, the result is undefined. As in, bad. Most likely, a crash. Blue screens in Windows are not out of the question.

```
SoLoud::Soloud *soloud = new SoLoud::Soloud; // object created
soloud->init();             // back-end initialization
...
soloud->deinit();           // clean-up
delete soloud;              // this cleans up too
```

Seriously: remember to call the cleanup function. The SoLoud object destructor also calls the cleanup function, but if you perform your application's tear-down in an unpredictable order (such as having the SoLoud object be a global variable), the back-end may end up trying to use resources that are no longer available. So, it's best to call the cleanup function manually.

9.2 Soloud.play()

The play function can be used to start playing a sound source. The function has more than one parameter, with typical default values set to most of them.

```
int play(AudioSource &aSound,
         float aVolume = 1.0f, // Full volume
         float aPan = 0.0f,    // Centered
         int aPaused = 0,      // Not paused
         int aBus = 0);        // Primary bus
```

Unless you know what you're doing, leave the aBus parameter to zero.

The play function returns a channel handle which can be used to adjust the parameters of the sound while it's playing. The most common parameters can be set with the play function parameters, but for more complex processing you may want to start the sound paused, adjust the parameters, and then un-pause it.

```
int h = soloud.play(sound, 1, 0, 1);  // start paused
soloud.setRelativePlaySpeed(h, 0.8f); // change a parameter
soloud.setPause(h, 0);                // unpause
```

9.3 Soloud.seek()

You can seek to a specific time in the sound with the seek function. Note that the seek operation may be rather heavy, and some audio sources will not support seeking backwards at all.

```
int h = soloud.play(sound, 1, 0, 1); // start paused
soloud.seek(h, 3.8f);                // seek to 3.8 seconds
soloud.setPause(h, 0);               // unpause
```

9.4 Soloud.stop()

The stop function can be used to stop a sound.

```
soloud.stop(h); // Silence!
```

9.5 Soloud.stopAll()

The stop function can be used to stop all sounds. Note that this will also stop the protected sounds.

```
soloud.stopAll(); // Total silence!
```

9.6 Soloud.stopSound()

The stop function can be used to stop all sounds that were started through a certain sound source. Will also stop protected sounds.

```
soloud.stopSound(duck); // silence all the ducks
```

9.7 Soloud.setGlobalVolume() / Soloud.getGlobalVolume()

These functions can be used to get and set the global volume. The volume is applied before clipping. Lowering the global volume is one way to combat clipping artifacts.

```
float v = soloud.getGlobalVolume(); // get the current global volume
soloud.setGlobalVolume(v * 0.5f);   // halve it
```

Note that the volume is not limited to 0..1 range. Negative values may result in strange behavior, while huge values will likely cause distortion.

9.8 Soloud.setPostClipScaler() / Soloud.getPostClipScaler()

These functions can be used to get and set the post-clip scaler. The scaler is applied after clipping. Sometimes lowering the post-clip result sound volume may be beneficial. For instance, recording video with some video capture software results in distorted sound if the volume is too high.

```
float v = soloud.getPostClipScaler(); // get the current post-clip scaler
soloud.setPostClipScaler(v * 0.5f);   // halve it
```

Note that the scale is not limited to 0..1 range. Negative values may result in strange behavior, while huge values will likely cause distortion.

10 Core: Attributes

10.1 Soloud.getVolume() / Soloud.setVolume()

These functions can be used to get and set a sound's current volume setting.

```
float v = soloud.getVolume(h); // Get current volume
soloud.setVolume(h, v * 2);    // Double it
```

Note that the volume is the "volume setting", and the actual volume will depend on the sound source. Namely, a whisper will most likely be more quiet than a scream, even if both are played at the same volume setting.

If an invalid handle is given to getVolume, it will return 0.

10.2 Soloud.getPan() / Soloud.setPan()

These functions can be used to get and set a sound's current pan setting.

```
float v = soloud.getPan(h); // Get current pan
soloud.setPan(h, v - 0.1);  // Little bit to the left
```

The range of the pan values is -1 to 1, where -1 is left, 0 is middle and and 1 is right. Setting value outside this range may cause undefined behavior.

SoLoud calculates the left/right volumes from the pan to keep a constant volume; to set the volumes directly, use setPanAbsolute.

If an invalid handle is given to getPan, it will return 0.

10.3 Soloud.setPanAbsolute()

These function can be used to set the left/right volumes directly.

```
soloud.setPanAbsolute(h, 1, 1); // full blast
```

Note that this does not affect the value returned by getPan.

If an invalid handle is given to getPan, it will return 0.

10.4 Soloud.getSamplerate() / Soloud.setSamplerate()

These functions can be used to get and set a sound's base sample rate.

```
float v = soloud.getSamplerate(h); // Get the base sample rate
soloud.setSamplerate(h, v * 2);    // Double it
```

Setting the value to 0 will cause undefined behavior, likely a crash.

To adjust the play speed, while leaving the base sample rate alone, use setRelativePlaySpeed instead.

If an invalid handle is given to getSamplerate, it will return 0.

10.5 Soloud.getRelativePlaySpeed() / Soloud.setRelativePlaySpeed()

These functions can be used to get and set a sound's relative play speed.

```
float v = soloud.getRelativePlaySpeed(h); // Get relative play speed
soloud.setRelativePlaySpeed(h, v * 0.5f); // Halve it
```

Setting the value to 0 will cause undefined behavior, likely a crash.

Change the relative play speed of a sample. This changes the effective sample rate while leaving the base sample rate alone.

Note that playing a sound at a higher sample rate will require SoLoud to request more samples from the sound source, which will require more memory and more processing power. Playing at a slower sample rate is cheaper.

If an invalid handle is given to getRelativePlaySpeed, it will return 1.

10.6 Soloud.getProtectChannel() / Soloud.setProtectChannel()

These functions can be used to get and set a sound's protection state.

```
int v = soloud.getProtectChannel(h);   // Get the protection state
if (v) soloud.setProtectChannel(h, 0); // Disable if protected
```

Normally, if you try to play more sounds than there are channels, SoLoud will kill off the oldest playing sound to make room. This will most likely be your background music. This can be worked around by protecting the sound.

If all sounds are protected, the result will be undefined.

If an invalid handle is given to getProtectChannel, it will return 0.

10.7 Soloud.getPause() / Soloud.setPause()

The setPause function can be used to pause, or unpause, a sound.

```
if (soloud.getPause(h)) hum_silently();
soloud.setPause(h, 0); // resumes playback
```

Note that even if a sound is paused, its channel may be taken over. Trying to resume a sound that's no longer in a channel doesn't do anything.

If the handle is invalid, the getPause will return 0.

10.8 Soloud.setPauseAll()

The setPauseAll function can be used to pause, or unpause, all sounds.

```
soloud.setPauseAll(h, 0); // resumes playback of all channels
```

Note that this function will overwrite the pause state of all channels at once. If your game uses this to pause/unpause the sound while the game is paused, do note that it will also pause/unpause any sounds that you may have paused/unpaused separately.

10.9 Soloud.setFilterParameter()

Sets a parameter for a live instance of a filter. The filter must support changing of live parameters; otherwise this call does nothing.

```
soloud.setFilterParameter(h, 3, FILTER::CUTOFF, 1000);
// set h's 3rd filter's "cutoff" value to 1000
```

10.10 Soloud.getFilterParameter()

Gets a parameter from a live instance of a filter. The filter must support changing of live parameters; otherwise this call returns zero.

```
float v = soloud.getFilterParameter(h,3,FILTER::CUTOFF);
// get h's 3rd filter's "cutoff" value
```

11 Core: Faders

11.1 Overview

Faders are a convenient way of performing some common audio tasks without having to add complex code into your application.

The most common use for the faders is to fade audio in or out, adding nice touches and polish.

Let's say you're exiting a bar and entering the street.

```
soloud.fadeVolume(bar_ambience, 0, 2); // fade bar out in 2 seconds
soloud.scheduleStop(bar_ambience, 2);  // stop the bar ambience after fadeout
street_ambience = soloud.play(cars, 0);// start street ambience at 0 volume
soloud.setProtectChannel(street_ambience, 1); // protect it
soloud.fadeVolume(street_ambience, 1, 1.5f); // fade street in in 1.5
```

Or let's say you're quiting your game.

```
soloud.fadeGlobalVolume(0, 1); // Fade out global volume in 1 second
```

The faders are only evaluated once per mix function call - in other words, whenever the back end requests samples from SoLoud, which is likely to be in chunks of 20-100ms, which is smoothly enough for most uses.

The exception is volume (which includes panning), which gets interpolated on per-sample basis to avoid artifacts.

The starting value for most faders is the current value.

11.2 Soloud.fadeVolume()

Smoothly change a channel's volume over specified time.

```
soloud.fadeVolume(orchestra, 1, 60); // The orchestra creeps in for a minute
```

The fader is disabled if you change the channel's volume with setVolume()

11.3 Soloud.fadePan()

Smoothly change a channel's pan setting over specified time.

```
soloud.setPan(racecar, -1); // set start value
soloud.fadePan(racecar, 1, 0.5); // Swoosh!
```

The fader is disabled if you change the channel's panning with setPan() or setPanAbsolute()

11.4 Soloud.fadeRelativePlaySpeed()

Smoothly change a channel's relative play speed over specified time.

```
soloud.fadeRelativePlaySpeed(hal, 0.1, 6); // Hal's message slows down
```

The fader is disabled if you change the channel's play speed with setRelativePlaySpeed()

11.5 Soloud.fadeGlobalVolume()

Smoothly change the global volume over specified time.

```
soloud.fadeGlobalVolume(0, 2); // Fade everything out in 2 seconds
```

The fader is disabled if you change the global volume with setGlobalVolume()

11.6 Soloud.schedulePause()

After specified time, pause the channel

```
soloud.fadeVolume(jukebox, 0, 2); // Fade out the music in 2 seconds
soloud.schedulePause(jukebox, 2);    // Pause the music after 2 seconds
```

The scheduler is disabled if you set the pause state with setPause() or setPauseAll().

11.7 Soloud.scheduleStop()

After specified time, stop the channel

```
soloud.fadeVolume(applause, 0, 10); // Fade out the cheers for 10 seconds
soloud.scheduleStop(applause, 10);     // Stop the sound after 10 seconds
```

There's no way (currently) to disable this scheduler.

11.8 Soloud.oscillateVolume()

Set fader to oscillate the volume at specified frequency.

```
soloud.oscillateVolume(murmur, 0, 0.2, 5); // murmur comes and goes
```

The fader is disabled if you change the channel's volume with setVolume()

11.9 Soloud.oscillatePan()

Set fader to oscillate the panning at specified frequency.

```
soloud.oscillatePan(ambulance, -1, 1, 10); // Round and round it goes
```

The fader is disabled if you change the channel's panning with setPan() or setPanAbsolute()

11.10 Soloud.oscillateRelativePlaySpeed()

Set fader to oscillate the relative play speed at specified frequency.

```
soloud.oscillateRelativePlaySpeed(vinyl, 0.9, 1.1, 3); // Wobbly record
```

The fader is disabled if you change the channel's play speed with setRelativePlaySpeed()

11.11 Soloud.oscillateGlobalVolume()

Set fader to oscillate the global volume at specified frequency.

```
soloud.oscillateGlobalVolume(0.5, 1.0, 0.2); // Go crazy
```

The fader is disabled if you change the global volume with setGlobalVolume()

11.12 Soloud.fadeFilterParameter()

Fades a parameter on a live instance of a filter. The filter must support changing of live parameters; otherwise this call does nothing.

```
soloud.fadeFilterParameter(h,3,FILTER::CUTOFF,1000,1);
// Fades h's 3rd filter CUTOFF to 1000 in 1 second
```

11.13 Soloud.oscillateFilterParameter()

Oscillates a parameter on a live instance of a filter. The filter must support changing of live parameters; otherwise this call does nothing.

```
soloud.setFilterParameter(h,3,FILTER::CUTOFF,500,1000,2);
// Oscillates the h's 3rd filter's CUTOFF between 500 and 1000
```

12 Core: Misc

12.1 Soloud.getStreamTime()

The getStreamTime function can be used to get the current play position, in seconds.

```
float t = soloud.getStreamTime(h); // get time
if (t == hammertime) hammer();
```

Note that due to being a floating point value, playing a long stream may cause precision problems, and eventually cause the “time” to stop. This will happen in about 6 days. The precision problems will start somewhat earlier.

Also note that the granularity is likely to be rather high (possibly around 45ms), so using this as the sole clock source for animation will lead to rather low framerate (possibly around 20Hz). To fix this, either use some other clock source and only sync with the stream time occasionally, or use some kind of low-pass filter, such as..

```
mytime = (mytime * 9 + soloud.getStreamTime(h)) / 10;
```

While not perfect, that’s way better than using the stream time directly.

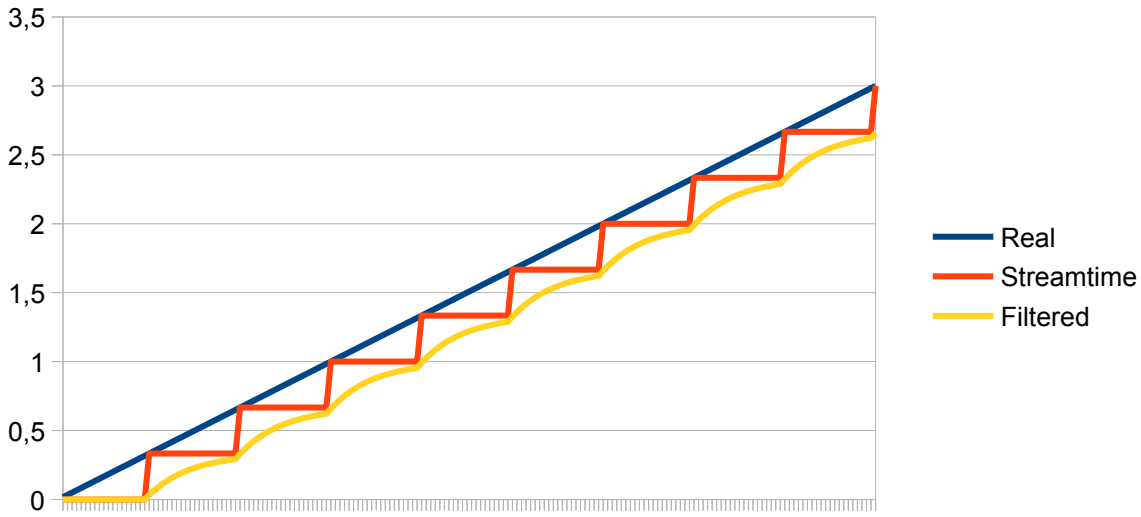


Figure 12.1: Low-pass filtered time values

12.2 Soloud.isValidChannelHandle()

The isValidChannelHandle function can be used to check if a handle is still valid.

```
if (!soloud.isValidChannelHandle(h)) delete foobar;
```

If the handle is invalid, the isValidChannelHandle will return 0.

12.3 Soloud.getActiveVoiceCount()

Returns the number of concurrent sounds that are playing at the moment.

```
if (soloud.getActiveVoiceCount() == 0) enjoy_the_silence();
```

If the handle is invalid, the getActiveVoiceCount will return 0.

12.4 Soloud.setGlobalFilter()

Sets, or clears, the global filter.

```
soloud.setGlobalFilter(0, &echochamber); // set first filter
```

Setting the global filter to NULL will clear the global filter. The default maximum number of global filters active is 4, but this can be changed in a global constant in soloud.h (and rebuilding SoLoud).

12.5 Soloud.calcFFT()

Calculates FFT of the currently playing sound (post-clipping) and returns a pointer to the result.

```
float * fft = soloud.calcFFT();
int i;
for (i = 0; i < 256; i++)
  drawline(0, i, fft[i] * 32, i);
```

The FFT data has 256 floats, from low to high frequencies.

SoLoud performs a mono mix of the audio, passes it to FFT, and then calculates the magnitude of the complex numbers for application to use. For more advanced FFT use, SoLoud code changes are needed.

The returned pointer points at a buffer that's always around, but the data is only updated when calcFFT() is called.

For the FFT to work, you also need to initialize SoLoud with the Soloud::ENABLE_VISUALIZATION flag. Otherwise the source data for the FFT calculation will not be gathered.

12.6 Soloud.getWave()

Gets 256 samples of the currently playing sound (post-clipping) and returns a pointer to the result.

```
float * wav = soloud.getWave();
int i;
for (i = 0; i < 256; i++)
  drawline(0, i, wav[i] * 32, i);
```

The returned pointer points at a buffer that's always around, but the data is only updated when getWave() is called. The data is the same that is used to generate visualization FFT data.

For this function to work properly, you also need to initialize SoLoud with the Soloud::ENABLE_VISUALIZATION flag. Otherwise the source data will not be gathered, and the result is undefined (probably zero).

12.7 Soloud.getVersion()

Returns the version of the SoLoud library. Same as SOLOUD_VERSION macro. Mostly useful when using the DLL, to check the DLL's library version.

13 SoLoud::AudioSource

All audio sources share some common functions. Some of the functionality depends on the audio source itself; it may be that some parameter does not make sense for a certain audio source, or it may be that it has not been implemented for other reasons.

For example, if you stream a live radio station, looping does not make much sense.

13.1 AudioSource.setLooping()

This function can be used to set a sample to play on repeat, instead of just playing once.

```
amenbreak.setLooping(1); // let the beat play on
```

Note that some audio sources may not implement this behavior.

13.2 AudioSource.setFilter()

This function can be used to set or clear the filters that should be applied to the sounds generated via this audio source.

```
speech.setFilter(0, blackmailer); // Disguise the speech
```

Setting the filter to NULL will clear the filter. This will not affect already playing sounds. By default, up to four filters can be applied. This value can be changed through a constant in the soloud.h file.

13.3 AudioSource.setSingleInstance()

This function can be used to tell SoLoud that only one instance of this sound may be played at the same time.

```
menuselect.setSingleInstance(1); // Only play it once, Sam
```

14 SoLoud::Wav

The SoLoud::Wav class represents a wave sound effect. The source files may be in 8 or 16 bit raw RIFF WAV files, or compressed Ogg Vorbis files.

The sounds are loaded and converted to float samples, which means that every second of a 44100Hz stereo sound takes about 350kB of memory. The good side is, after loading, these samples are very lightweight, as their processing is mostly just a memory copy.

For lengthy samples like background music, you may want to use SoLoud::WavStream instead.

14.1 Wav.load()

The wav loader takes just one parameter, the file name:

```
void load(const char *aFilename); // File to load
```

If loading fails, the sample will be silent.

```
SoLoud::Wav boom;
boom.load("boom.wav");
```

If the loading function is called while there are instances playing, the result is undefined (most likely a crash).

14.2 Wav.loadMem()

Alternate way of loading samples is to read from a memory buffer.

```
void loadMem(unsigned char *aMem, int aLength); // Sample to load
```

If loading fails, the sample will be silent.

```
SoLoud::Wav boom;
boom.loadMem(boomMemoryResource, boomMemoryResourceLength);
```

If the loading function is called while there are instances playing, the result is undefined (most likely a crash).

15 SoLoud::WavStream

The SoLoud::WavStream class represents a wave sound effect that is streamed off disk while it's playing. The source files may be in 8 or 16 bit raw RIFF WAV files, or compressed Ogg Vorbis files.

The sounds are loaded in pieces while they are playing, which takes more processing power than playing samples from memory, but they require much less memory.

For short or often used samples, you may want to use SoLoud::Wav instead.

15.1 WavStream.load()

The wav loader takes just one parameter, the file name:

```
void load(const char *aFilename); // File to load
```

If loading fails, the sample will be silent.

```
SoLoud::WavStream muzak;
muzak.load("elevator.ogg");
```

If the loading function is called while there are instances playing, the result is undefined (most likely a crash).

16 SoLoud::Speech

The SoLoud::Speech class implements a simple Klatt-style formant speech synthesizer. It's barely legible, not really human-like, but it's free, and it's here.

Adjusting the speech synthesizer's output with audio filters should allow for various voices, which, along with subtitles, will let you add voice to your games cheaply.

For more serious use, feel free to study the source code and play with the various internal parameters, as well as apply various filters to the sound.

For legal notes, please see the license page.

16.1 Speech.setText()

The setText function can be used to set the text to be spoken.

```
SoLoud::Speech sp;
sp.setText("Hello␣world.␣␣You␣will␣be␣assimilated.");
```

If the setText function is called while speech is playing, SoLoud stops any playing instances to avoid crashing.

17 SoLoud::Sfxr

The SoLoud::Sfxr is a “retro” sound effect synthesizer based on the original Sfxr by Tomas Pettersson.

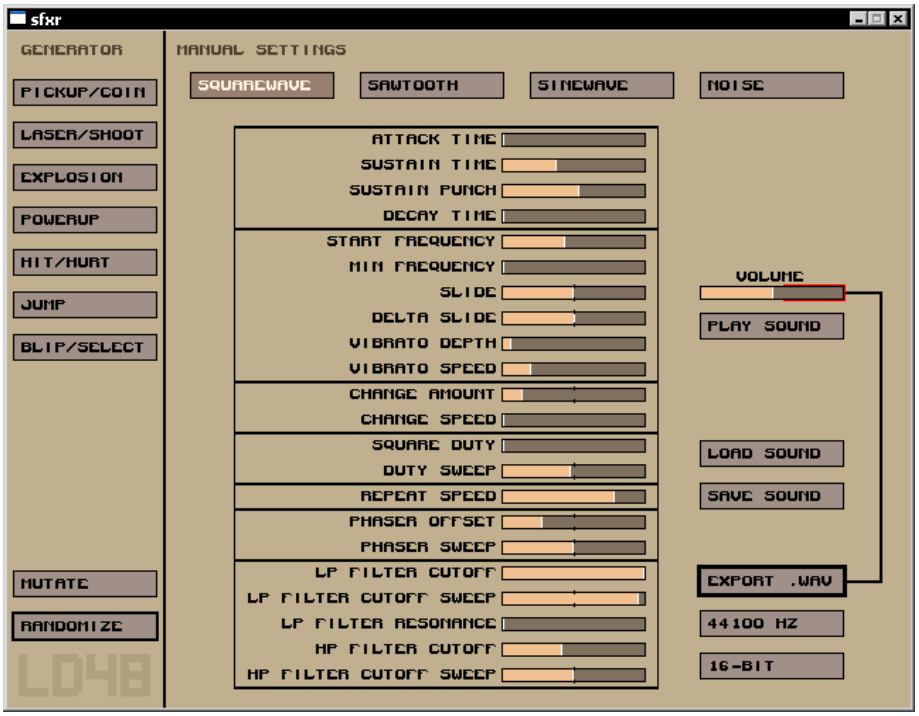


Figure 17.1: Sfxr interface

The original sfxr tool was designed to easily generate sound effects for Ludum Dare 48h games. SoLoud includes the same engine built in, so you can (should you wish) make every coin, explosion etc. sound different.

17.1 Sfxr.loadPreset()

You can simply tell Sfxr to use one of the presets (COIN, LASER, EXPLOSION, POWERUP, HURT, JUMP, BLIP). Each of the presets has several random components, so you can get virtually unlimited variations of each. (Not all variants sound good, though).

```
void loadPreset(int aPresetNo, int aRandSeed); // Preset to load
```

If loading fails, the effect will be silent.

```
SoLoud::Sfxr coin;
coin.loadPreset(Sfxr::COIN, 3247);
```

17.2 Sfxr.loadParams()

Effect parameters can also be loaded from a configuration file saved from the sfxr tool.

```
int loadParams(const char *aFilename); // File to load
```

If loading fails, the return is non-zero.

```
SoLoud::Sfxr boom;
boom.loadParams("boom.sfx");
```

18 SoLoud::Modplug

The SoLoud::Modplug is a module-playing engine, capable of replaying wide variety of multi-channel music (669, abc, amf, ams, dbm, dmf, dsm, far, it, j2b, mdl, med, mid, mod, mt2, mtm, okt, pat, psm, ptm, s3m, stm, ult, umx, xm). It also loads wav files, and may support wider support for wav files than the stand-alone wav audio source.

Due to its size, it's possible to compile SoLoud without the modplug support.

The midi formats (.mid and .abc) require a library of instruments (patches) to be available. One free set can be downloaded from the SoLoud downloads page. By default, the patches are loaded from pat/ directory.

18.1 Modplug.load()

You tell modplug to load a file with the load function:

```
int load(const char *aFilename); // File to load
```

If loading fails, the return is non-zero.

```
SoLoud::Modplug spacedeb;
spacedeb.load("spacedeb.mod");
```

19 Creating New Audio Sources

SoLoud is relatively easy to extend by creating new sound sources. Each sound source consists of two parts: an audio source class, and an audio instance class.

Studying the existing audio sources' source code, in addition to this chapter, will be helpful in creating new ones.

19.1 AudioSource class

```
class Example : public AudioSource
{
public:
  virtual AudioInstance *createInstance();
};
```

The only mandatory member of an audio source is the createInstance function.

The audio source class is meant to contain all and any data that represents the sound in general and can be reused by the instances; for instance, with wave files, the wave data is stored with the audio source, while audio instances just read the data.

Note that there's no setLooping() function - that's inherited from AudioSource, and sets the SHOULD_LOOP flag.

The audio source is also responsible for setting the mChannels and mBaseSamplerate values. These values get copied to all of the instances of this audio source.

19.2 AudioSource.createInstance()

The createInstance function typically creates and returns its counterpart, the audio instance. Usually it also gives a pointer to itself to the audio instance.

19.3 AudioSourceInstance class

```
class ExampleInstance : public AudioSourceInstance
{
public:
  virtual void getAudio(float *aBuffer, int aSamples);
  virtual int hasEnded();
  virtual void seek(float aSeconds, float *mScratch, int mScratchSize);
  virtual int rewind();
};
```

The getAudio and hasEnded methods are mandatory. Seek and rewind are optional.

The audio instance is meant as the "play head" for a sound source. Most of the data should be in the audio source, while audio instance may contain more logic.

19.4 AudioSourceInstance.getAudio()

SoLoud requests samples from the sound instance using the getAudio function. If the instance generates more than one channel (i.e, stereo sound), the expected sample data first has the first channel samples, then second channel samples, etc.

So, if 1024 samples are requested from a stereo audio source, the first 1024 floats should be for the first channel, and the next 1024 samples should be for the second channel.

The getAudio function is also responsible for handling looping, if the audio source supports it. See the implementations of existing sound sources for more details.

If the audio source runs out of data, the rest of the buffer should be set to zero.

19.5 AudioSourceInstance.hasEnded()

After mixing, SoLoud asks all audio instances whether they have ended, and if they have, it will free the object and free the channel. Supporting looping will likely affect the implementation of this function.

19.6 AudioSourceInstance.seek()

Optionally, you can implement a seek function. The base implementation will simply request (and discard) samples from the sound source until the desired position has been reached; for many sound sources, a smarter way exists.

19.7 AudioSourceInstance.rewind()

To enable the base implementation of seek to seek backwards from the current play position, sound source may implement the rewind function. In most cases the rewind is easier to implement than actual smart seeking.

20 SoLoud::Bus

The mixing busses are a special case of an audio stream. They are a kind of audio stream that plays other audio streams. Mixing bus can also play other mixing busses. Like any other audio stream, mixing bus has volume, panning and filters.

Only one instance of a mixing bus can play at the same time, however; trying to play the same bus several times stops the earlier instance.

While a mixing bus doesn't generate audio by itself, playing it counts against the maximum number of concurrent streams.

Mixing busses are protected by default (i.e, won't stop playing if maximum number of concurrent streams is reached).

To play a stream through the mixing bus, use the bus play() command.

```
int bushandle = gSoloud.play(gBus); // Play the bus
gSoloud.setVolume(bushandle, 0.5f); // Set bus volume

int fxhandle = gBus.play(gSoundEffect); // Play sound effect through bus
gSoloud.setVolume(fxhandle, 0.5f); // set sound effect volume
```

21 SoLoud::Filter

Filters can be used to modify the sound some way. Typical uses for a filter are to create environmental effects, like echo, or to modify the way the speech synthesizer sounds like.

Like audio sources, filters are implemented with two classes; Filter and FilterInstance. These are, however, typically much simpler than those derived from the AudioSource and AudioInstance classes.

21.1 Filter class

```
class Example : public Filter
{
public:
  virtual FilterInstance *createInstance();
};
```

As with audio sources, the only required function is the createInstance().

21.2 FilterInstance class

```
class ExampleInstance : public FilterInstance
{
public:
   virtual void initParams(int aNumParams);

       virtual void updateParams(float aTime);

  virtual void filter(
    float *aBuffer,     int aSamples,
    int aChannels,      float aSamplerate,
    float aTime);

  virtual void filterChannel(
    float *aBuffer,     int aSamples,
    float aSamplerate,  float aTime,
    int aChannel,       int aChannels);

  virtual float getFilterParameter(
    int aAttributeId);

  virtual void setFilterParameter(
    int aAttributeId,   float aValue);

  virtual void fadeFilterParameter(
    int aAttributeId,   float aTo,
    float aTime,        float aStartTime);

  virtual void oscillateFilterParameter(
    int aAttributeId,   float aFrom,
    float aTo,          float aTime,
    float aStartTime);
};
```

The filter instance has no mandatory functions, but you may want to implement either filter() or filterChannel() to do some actual work.

21.3 FilterInstance.initParams

You should call this in the constructor of your filter instance, with the number of parameters your filter has. By convention, the first parameter should be the wet/dry parameter, where value 1 outputs fully filtered and 0 completely original sound.

21.4 FilterInstance.updateParams

You should call this function in your filter or filterChannel functions to update fader values.

The mNumParams member contains the parameter count.

The mParamChanged member is bit-encoded field showing which parameters have changed. If you want to know whether parameter 3 has changed, for instance, you could do:

```
mParamChanged = 0;
updateParams(aTime);
if (mParamChanged & (1 << 3)) // param 3 changed
```

Finally, mParam array contains the parameter values, and mParamFader array contains the faders for the parameters.

21.5 FilterInstance.filter()

The filter() function is the main workhorse of a filter. It gets a buffer of samples, channel count, samplerate and current stream time, and is expected to overwrite the samples with filtered ones.

If channel count is not one, the layout of the buffer is such that the first channel's samples come first, followed by the second channel's samples, etc.

So if dealing with stereo samples, aBuffer first has aSamples floats for the first channel, followed by aSamples floats for the second channel.

The default implementation calls filterChannel for every channel in the buffer.

21.6 FilterInstance.filterChannel()

Most filters are simpler to write on a channel-by-channel basis, so that they only deal with mono samples. In this case, you may want to use the filterChannel() function instead. The default implementation of filter() calls the filterChannel() for every channel in the source.

21.7 FilterInstance.getFilterParameter()

This function is needed to support the changing of live filter parameters. The default implementation uses the mParam array.

Unless you do something unexpected, you shouldn't need to touch this function.

21.8 FilterInstance.setFilterParameter()

This function is needed to support the changing of live filter parameters. The default implementation uses the mParam array.

Unless you do something unexpected, you shouldn't need to touch this function.

21.9 FilterInstance.fadeFilterParameter()

This function is needed to support the changing of live filter parameters. The default implementation uses the mParamFader array.

Unless you do something unexpected, you shouldn't need to touch this function.

21.10 FilterInstance.oscillateFilterParameter()

This function is needed to support the changing of live filter parameters. The default implementation uses the mParamFader array.

Unless you do something unexpected, you shouldn't need to touch this function.

22 SoLoud::BiquadResonantFilter

The biquad resonant filter is a surprisingly cheap way to implement low and high pass filters, as well as some kind of band bass filter.

The implementation in SoLoud is based on “Using the Biquad Resonant Filter”, Phil Burk, Game Programming Gems 3, p. 606.

The filter has three parameters - sample rate, cutoff frequency and resonance. These can also be adjusted on live streams, for instance to fade the low pass filter cutoff frequency for a outdoors/indoors transition effect.

The resonance parameter adjusts the sharpness (or bandwidth) of the cutoff.

```
// Set up low-pass filter
gBQRFilter.setParams(SoLoud::BiquadResonantFilter::LOWPASS, 44100, 500, 2);
// Set the filter as the second filter of the bus
gBus.setFilter(1, &gBQRFilter);
```

It’s also possible to set, fade or oscillate the parameters of a “live” filter

```
gSoloud.fadeFilterParameter(
  gMusicHandle, // Sound handle
  0,            // First filter
  SoLoud::BiquadResonantFilter::FREQUENCY, // What to adjust
  2000,         // Target value
  3);           // Time in seconds
```

Currently, four parameters can be adjusted:

Parameter	Description
WET	Filter’s wet signal; 1.0f for fully filtered, 0.0f for original, 0.5f for half and half.
SAMPLERATE	Filter’s samplerate parameter
FREQUENCY	Filter’s cutoff frequency
RESONANCE	Filter’s resonance - higher means sharper cutoff

23 SoLoud::EchoFilter

The echo filter in SoLoud is a very simple one. When the sound starts to play, the echo filter allocates a buffer to contain the echo samples, and loops through this until the sound ends.

The filter does not support changing of parameters on the fly, nor does it take changing of relative play speed into account.

There are two parameters - delay and decay. Delay is the time in seconds until the echo, and decay is multiplier for the echo. If the multiplier is outside the [0..1[range, the results are unpredictable.

```
// Set up echo filter
gEchoFilter.setParams(0.5f, 0.5f);
// Set the filter as the first filter of the bus
gBus.setFilter(0, &gEchoFilter);
```

24 SoLoud::FFTFilter

The FFT filter is a simple voice-breaking filter that uses FFT and inverse FFT.

The filter exists mainly to adjust the speech synthesizer's voice in strange ways. It can also be used as basis for other FFT-based filters.

The filter does not support changing of parameters on the fly, nor does it take changing of relative play speed into account.

There are three parameters, shift, combine and scale. Finding usable results from the filter can be done mainly through trial and error. The combine tells the filter how to combine the wet and dry signals - OVER uses wet signal directly, SUBTRACT subtracts the wet signal from the dry, and MULTIPLY multiplies them together.

Scale exists because the resulting volume level can be all over the place.

```
// Set up echo filter
gFFTFilter.setParams(-15, FFTFilter::SUBTRACT, 0.002f);
// Set the filter as the first filter of the speech
gSpeech.setFilter(0, &gFFTFilter);
```

25 SoLoud::LofiFilter

The lofi filter is a signal degrading filter. You can adjust both the bit depth and the sample rate of the output, and these parameters can also be adjusted (and even faded) on the fly.

```
// Set up low-pass filter
gLofiFilter.setParams(8000, 5);
// Set the filter as the first filter of the bus
gBus.setFilter(0, &gLofiFilter);
```

It's also possible to set, fade or oscillate the parameters of a "live" filter

```
gSoloud.fadeFilterParameter(
  gMusicHandle, // Sound handle
  0,            // First filter
  SoLoud::LofiFilter::BITDEPTH, // What to adjust
  2,            // Target value
  3);           // Time in seconds
```

Currently, four parameters can be adjusted:

Parameter	Description
WET	Filter's wet signal; 1.0f for fully filtered, 0.0f for original, 0.5f for half and half.
SAMPLERATE	Filter's samplerate parameter
BITDEPTH	Filter's bit-depth parameter

26 Back-ends

SoLoud needs a back-end to play audio out. SoLoud ships with a bunch of back-ends with various levels of stability and latency. Creating new back-ends is relatively simple.

SoLoud speaks with the back-end with only a couple of functions, in addition to the optional mutex function pointers.

Studying the existing back-end implementations' source code, in addition to this page, will help creating new ones.

26.1 Soloud.postinit()

The back-end should call Soloud.postinit() once it knows what it can do.

```
void postinit(int aSamplerate, // Sample rate, in Hz
              int aBufferSize, // Buffer size, in samples
              int aFlags);     // Flags
```

The channels and flags most likely come directly from the application, while sample rate and buffer size may depend on how the back-end does things. The buffer size should be the maximum number of samples the back-end requests on one call. Making it bigger doesn't affect latency, but causes SoLoud to create larger than necessary internal mixing buffers.

26.2 Soloud.mix()

The back-end can call the mix function to request a number of stereo samples from SoLoud. The samples will be in float format, and the back-end is responsible for converting them to the desired output format.

```
void mix(float *aBuffer, // Destination buffer
         int aSamples);  // Number of requested stereo samples
```

If the number of samples exceeds the buffer size set at init, the result is undefined (most likely a crash).

26.3 Soloud.mBackendData

This void pointer is free for the back-end to use in any way it wants. It may be a convenient place to store any buffers and other information it needs to keep around.

26.4 Soloud.mLockMutexFunc / Soloud.mUnlockMutexFunc

These function pointers point to functions which should lock and unlock a mutex. If they are left as NULL, they will not be called.

If they're not implemented, SoLoud will not be thread safe. This means that some shared resources, such as the channel data, may be accessed by several threads at the same time. In the worst case one thread may delete an object while another is accessing it.

26.5 Soloud.mMutex

Pointer to mutex data. The pointer is also passed to the lock/unlock mutex functions as a parameter.

26.6 Soloud.mBackendCleanupFunc

This function pointer is used by SoLoud to signal the back-end to perform cleanup; stop any threads, free any resources, etc. If NULL, not called, but may result in resource leaks and quite possibly crashes.

26.7 Different back-ends

This is a non-exhaustive list of back-ends and notes regarding them.

- SDL
  - Most tested, primary development platform
  - Cross-platform
  - Low latency
- PortAudio
  - Cross-platform
  - Low latency
- Windows multimedia
  - Simplest back-end for Windows-only programs
- oss (/dev/dsp)
  - Simplest back-end for Linux-only programs
  - Experimental
- OpenAL
  - Experimental
  - High latency; if this is your only option, you're probably better off using OpenAL directly.
- WASAPI
  - Experimental
- XAudio2
  - Experimental