YSE Manual

Mute (http://mutecode.com)

Revision: August 21, 2015

Contents

Copyright Notice

The YSE sound engine is licensed under the Eclipse license. Without going into details, this means YSE can be used for commercial as well as open source projects. If in doubt, consult the Eclipse license.

Warning!

Parts of YSE are implemented using the Juce audio engine. (In particular Juce is used to support audio formats, input- and output devices.) Juce comes with a dual license option: GNU Public License for open source projects, and a commercial license for commercial projects. If you use YSE in a commercial product, you are free to do so but you should still get a commercial Juce license.

Thank you!

Mute likes to thank the Juce team for their splendid library. Juce enables us to focus on the functionality YSE, instead of having to worry about audio formats and devices. Another big help was 'The Audio Programming Book', edited by Richard Boulanger and Victor Lazzarini, which contains a vast resource of audio code examples. We also studied the Pure Data source code a lot. A big thank you goes to Miller Puckette for this wonderful piece of software.

Part I

The Basics

CHAPTER 1

Getting Started

1.1 Installation

1.1.1 Binary Release

Binary releases are provided on the Github project page. They contain the YSE header files, compiled libraries (both static and dynamic) and a few code examples. As always, they are probably the easiest way to get to know YSE.

1.1.1.1 Windows

YSE is compiled with Microsoft Visual Studio 2013. Due to the nature of c++, this means you will have to use Visual Studio for your own project as well. Because a lot of c++11 features are used, you need VS2012 or newer.

Download YSE and unpack the zip file. The folder 'demo' contains quite a few code examples. Examining the source code of these examples is a good idea if you're starting to work with YSE. All the examples can be compiled with the supplied script `generateDemoWin`, as long as you have visual studio installed. The script contains a call to `vcvarsall.bat`. If you didn't install Visual Studio at its default location, you will need to alter this path.

Adding YSe to your own project required that you:

1. Add the include directory to your project.

2. Add the `yse.hpp` header file to your project source files.
3. Add `libyse32.lib` to your project.
4. If you use the dll library, the easiest method is to pick an output directory and copy the provided dll in there, alongside your executable.

INFO

64 bit versions of the library are also provided. They are untested, but should work fine.

1.1.1.2 MacOS

Download YSE and unpack the zip file. The compiled libraries have to be placed in `\usr\local\lib`. This is especially important for the dynamic library. The folder 'demo' contains quite a few code examples. Examining the source code of these examples is a good idea if you're starting to work with YSE. All the examples can be compiled with the supplied script `generateDemoMac`. *(For information on linking yse on the command line, examine this script.)*

If you're used to programming on a mac, using YSE in your own projects should be straightforward. The following steps are needed:

1. Add the include directory to your project.
2. Add the `yse.hpp` header file to your project source files.
3. Add `libyse.dylib` (dynamic) or `libyse.a` (static) to your project.
4. If you decide to use the static version of YSE, please add the following frameworks to your project: Cocoa, IOKit, AudioToolbox, QuartzCore, DiscRecording, CoreMIDI, CoreAudio & Accelerate.

1.1.1.3 Android

Note: this manual is in heavy development right now, along with providing binaries for all platforms. I will add information for all platforms while working on the binary releases. Please check back in a few days. (august 21, 2015. Hopefully this paragraph is gone soon!)

1.2 Important Concepts

1.2.1 Functors

Some classes in YSE are probably needed in every program using the engine. I've decided to implement them as 'functors'. Internally, a functor is something like this:

```
YSE::system & YSE::System() {
  static system s;
  return s;
}
```

You should not try to create your own object from the system class. YSE uses this functor internally as well. We think this approach has some advantages:

- Code fragments are easier to read because the names for base objects (i.e. functors) are always the same.
- Contrary to global objects, the object in a functor is only created if called.
- You don't have to keep track whether or not an object is created. You can just use it whenever needed. It might be created already, internally, or it will be created when you're trying to use it. Both will work the same.
- Because the functor returns a reference to the object, it can be used just like a real object, except for the braces.

Other functor objects are:

```
YSE::System();
YSE::Listener();
YSE::Reverb();
YSE::Log();
YSE::IO();

// common sound channels
YSE::ChannelMaster();
YSE::ChannelGui();
YSE::ChannelFX();
YSE::ChannelMusic();
YSE::ChannelAmbient();
YSE::ChannelVoice();
```

1.2.2 Namespaces

Basic classes are always in the namespace `YSE`. Some classes are bit more complex though. To ensure that YSE can be used lock-free, they consist out of an interface class, an implementation class, a manager class and a message class. All of these are within their own namespace, but a typedef is provided for each of these classes so that you don't have to worry about this.

The reason this approach is mentioned here, at the very beginning of the manual, is that the doxygen reference manual does not 'see' these typedef's. So while in the documentation you may find the sound class defined as `YSE::SOUND::interfaceObject` you can use `YSE::sound` in your code. The same goes for other complex classes like `YSE::CHANNEL::interfaceObject`, which can be used as `YSE::channel`. *(Actually, when you want to use a class called 'interfaceObject' you will always be able to call it by it's parent namespace in lowercase.)*

Other namespaces are used for the more advanced capabilities of YSE: DSP, MIDI and MUSIC.

CHAPTER 2

Playing Sound

2.1 Initialise the Engine

Before you use any YSE objects, you will have to initialize the engine with:

```
YSE::System().init();
```

Once you've started the sound system, its update function should be called frequently. For responsive audio, this should be about 10 to 20 times a second. *(When using YSE as a virtual synth, you'll want to increase this number.)* If you're using YSE together with a game engine, a good place would be inside the game update loop. If your program doesn't have such a loop, you should consider a timer of some sort.

```
YSE::System().update();
```

When you're about to shutdown your program, call the close function to properly stop audio output:

```
YSE::System().close();
```

The close function also ensures that internal threads are stopped properly. It's really important to call this before exiting your program.

2.2 Create a sound

A sound object can be created from the class `YSE::sound`, which is actually a typedef for `YSE::SOUND::interfaceObject`. Constructing this object actually doesn't do much. This has the advantage that sound objects can be directly included inside classes or even at a global level, without using pointer constructions. Once the sound system is initialised, you can 'create' the sound, which means that a sound file or generator is loaded in memory.

Only after creating a sound, it can be played. Some properties of the sound can be passed to the create function, but behaviour can be altered later on.

```
// example of simple sound playing

// A sound can exist before initialising because it is just an interface.
YSE::sound mySound;

// Before actually creating the sound, the system MUST be initialised.
YSE::System().init();

// Now the sound can be created and played
mySound.create("sound.ogg").play();

// or create on a subchannel, looping:
mySound.create("sound.ogg", YSE::System().channelFX(), true);
mySound.play();
```

Most set functions in yse can be chained. So

```
mySound.setRelative(true);
mySound.setDoppler(false);
mySound.setPosition(YSE::Vec(1,1,1));
mySound.play();
```

is equal to:

```
mySound.setRelative(true).setDoppler(false).setPosition(YSE::Vec(1,1,1)).play();
```

Sound objects are very flexible. I won't go over all features in here. Please refer to the doxygen manual for more information.

2.3 About loading sound files

So what happens when you instruct a sound object to create itself? You pass the name of a sound file to the sound object, but what happens then?

Internally, YSE will check whether or not this file is already in memory. If it is, an extra pointer to this file will be created. You do not have to worry about the same file being loaded twice.

Memory is also released when no longer needed. If the last pointer to a file is destructed, YSE will keep the file in memory for about 30 seconds. If after that period no new sound object has requested access to the sound file, it will be released from memory.

When a file is requested and not in memory, it will be loaded in a separate thread. It can take a short while to load the file, depending on it's size and your CPU speed. Your software will continue while YSE loads the file in memory. You can even define properties for this sound object, or instruct it to start playing. These instructions will internally be delayed until the file is done loading.

2.4 Virtualization

YSE does not set a limit to the number of sounds you can play simultaneously. In a scene with a lot of sound sources this means you can create and play every sound object when you create it's visual counterpart. But that doesn't mean it makes sense to really play all those sounds. This is why YSE will virtualize some of them. By default, only 50 sounds will really be played. YSE will sort all sounds according to their proximity and volume, then play the ones on top of the list. The other sounds will probably not be heard anyway, so there's no need to render them.

This default can be altered with:

```
YSE::System().maxSounds(Int value);
```

It is also possible to disable virtualization completely for certain channels:

```
YSE::System().ChannelMusic().setVirtual(false);
```

Chapter 3

Channels

3.1 using and adding channels

When you create a sound and you do not specify a channel, it will be added to the master channel (`YSE::ChannelGlobal()`). A few other channels do exist by default:

```
YSE::ChannelGui();
YSE::ChannelFX();
YSE::ChannelAmbient();
YSE::ChannelVoice();
YSE::ChannelMusic();
```

You can also create your own channels, should you want to. If you do, the channel will be created as a child of the master channel, unless you specify another channel as the parent. Just like a sound, a channel is a typedef to `YSE::CHANNEL::interfaceObject`. Initialization of a custom channel object happens when you call its `create` function. This means it can also be constructed before `System().init()` is called. The create function should be called after `System().init()`.

Info

When a channel is removed, all its child channels and sounds are moved to its parent. This can be done without any interuption in the sound system, but if there is a large difference in output volumes of the channels, there might be a short click in the output sound.

Note that sounds and channels can also be moved from one channel to another manually.

3.2 Performance

Channels have several purposes. First, a few properties like volume can be adjusted to all sounds in the channel at once. Read the doxygen docs for more information about channel functions.

Another purpose is more hidden, but actually very important. When the audio callback function is called, it instructs all channels to calculate the audio they'd like to output at that moment. Every channel does this in its own thread. If the device you're using has multiple cores, as most devices do nowadays, YSE takes advantage of all available cores to calculate its audio.

With this in mind, it is crucial to spread out sounds over multiple channels for the best performance. (If you only use YSE to play a few sounds at a time, don't bother. It's mostly useful for 20 or more simultanious sounds.)

Another thing for which channels are important, are reverb calculations. This topic is discussed in another chapter.

CHAPTER 4

3D versus 2D

4.1 3D

By default, YSE will place a sound in a virtual 3D field. Its ouput will be rendered to one or more speakers in a stereo or surround setup. This is calculated internally, so you only have to update a sound's position to move it to a new spot.

Another way to change the position of all sounds in the stereo field is by moving the Listener object. By default, the Listener object is at position 0x0y0z, pointing forward. You can change the listener's position and orientation.

When you use YSE in a 3D game environment, the usual approach is to update the listener's position on every update. YSE will automatically calculate the current velocity of the listener and, when sounds have doppler enabled, apply a pitch shift to the sound accordingly.

```
YSE::Listener().SetPosition(YSE::Vec(0,0,0));
YSE::Listener().SetRotation(YSE::Vec(0,0,0));
```

4.2 2D

What if you do not want to position your audio in 3D? YSE is a bit different compared to other sound engines in that EVERY sound has a position. But this position can be absolute (typical 3D) or relative to the listener. A sound has an absolute position by default. To make it relative:

```
mySound.setRelative(true);
```

A relative sound at position 0 will always be at the position of the listener. In YSE, it is the closest you get to a '2D sound'. This has the extra advantage that mono sounds in a 2D environment can still be panned out quite easily by using their position.

4.3 Multichannel sound

Multichannel sounds can be used just the same way as mono sounds. They can be at an absolute or a relative position. The function setSpread(x) controls the angle, i.e. how much space there is between the channels in the virtual environment.

For example, if you play a stereo sound, with a spread of 90 degrees, and positioned at 0, the left channel will sound at -45% and the left channel wil sound at 45%.

If you play a 4 channel sound with the same spread of 90 degrees, the first channel will play at -45%, the second at -15%, the third at 15% and the forth at 45%.

Multichannel sounds can also be at absolute positions. In this case the angle of the channels will scale according to the position and angle of the sound itself and the listener.

Directional sound (mono or multichannel) is not implemented yet, but will follow later.

CHAPTER 5

Other Basic Classes

In this chapter we will review the other basic functionality of YSE. Most of these classes don't need much information. We think the header files should explain themselves.

5.1 Devices

You can use `YSE::System().getDevices()` to retrieve a list of all available audio devices. For related functions, check the header file. After engine initialization it is possible to switch to another output device. More information on a particular device can be retrieved from the `YSE::device` objects that are returned by this function.

To change the active device, you need to create an `YSE::deviceSetup` object, like this:

```
const std::vector<YSE::device> & list = YSE::System().getDevices();
YSE::deviceSetup setup;
setup.setOutput(list[3]); //assuming there are at least 4 devices
YSE::System().closeCurrentDevice();
YSE::System().openDevice(setup);
```

The `openDevice` function also takes an optional argument to specify channel output. By default the device will be used as a stereo device if possible. For surround configurations you can provide one of the folowing enumeration values as an argument:

```
enum CHANNEL_TYPE {
  CT_AUTO, // will pick stereo when possible
  CT_MONO,
  CT_STEREO,
```

```
  CT_QUAD,
  CT_51,
  CT_51SIDE,
  CT_61,
  CT_71,
  CT_CUSTOM, // custom type, you need to set speaker positions yourself
};
```

Info

The demo code also contains an example of how to work with audio devices.

5.2 Sound Occlusion

The sound occlusion implementation in YSE is mainly targeted at game developers, although it might be of some use to sound artists too.

Most game developers (if working with some sort of spacial representation) already have a simplified version of their virtual environment by means of a PhysX environment (or Bullets on Mac). These libraries offer functions for collision detection which are also useable for sound occlusion.

To implement sound occlusion with YSE, you can provide a function which takes two positions as arguments, and returns a float. YSE will call this function to check if a sound should be occluded or not. Partial occlusion is also possible:

```
float occlusionCallback(const YSE::Vec \& source, const YSE::Vec \& listener) {
  // your code goes here
}

// enable occlusion globally
YSE::System().occlusionCallback(occlusionCallback);

// turn occlusion on for the sounds you'd like to be affected
YSE::sound mySound;
mySound.create("afile.wav").setOcclusion(true);
```

The code in the callback itself depends on what you would like to do, but it would probably be a raycast between the first and the second position. If any objects are in that line, the sound should be occluded. If you assign different materials to your objects, you can easily decide how much of the sound gets through.

5.3 Reverb

YSE has a global reverb effect which can be enabled with:

```
YSE::System().getGlobalReverb().setActive(true);
YSE::System().getGlobalReverb().setPreset(YSE::REVERB_GENERIC);
YSE::ChannelMaster().attachReverb();
```

You do not have to attach this reverb to the master channel though. It can be on any channel you want. The reverb will be applied to the channel itself and all underlying channels. *(You can always move channels around to get a setup that suits you.)*

The following presets are available:

```
enum REVERB_PRESET {
  REVERB_OFF,
  REVERB_GENERIC,
  REVERB_PADDED,
  REVERB_ROOM,
  REVERB_BATHROOM,
  REVERB_STONEROOM,
  REVERB_LARGEROOM,
  REVERB_HALL,
  REVERB_CAVE,
  REVERB_SEWERPIPE,
  REVERB_UNDERWATER,
};
```

Aside from these presets, it is also possible to create a custom reverb.

5.3.1 Multiple Reverbs

Reverb calculations take a lot of CPU. Although it is possible to create multiple reverb objects, you should try to avoid this. As an alternative, you can use the `YSE::reverb` class. This class is not a real reverb, but a reverb configuration. Each reverb configuration can have its own values and can be assigned a position,a range and a rollOff range. When a listener moves into this range, the global reverb will take the values of this configuration into account to apply the reverb effect. This way, there's still only one reverb to calculate, but it is quite easy to create different accoustics bound to certain locations.

The downside of this approach is that the reverb is always dependant on the position of the listener. It cannot take into account which sound sources are located within the range of the reverb.

Info

The demo code shows a good example of these local reverbs.

Part II

DSP

Chapter 6

DSP Objects

6.1 Filter Objects

YSE offers 2 main uses for DSP: sound generation and filtering. Filter objects should be subclasses of YSE::DSP::dspObject. This is a virtual base class for which you should implement at least the process function. This function will be called during the audio rendering, which is done in a separate thread.

The process function is where all the sound manipulation happens. You get a reference to a multichannel audio buffer, which can be modified as to apply the filter. The comments in the following example should explain.

```
// all dsp filters must be based on the class YSE::DSP::dspObject
class lowPassFilter : public dspObject {
public:
  lowPassFilter() : parmFrequency(1000.f) {}
  virtual ~lowPassFilter() {};

  lowPassFilter & frequency(Flt value) {
    parmFrequency.store(value);
    return *this;
  }

  Flt frequency() { return parmFrequency; }

  virtual void create() {
    // The create function will be called by the parent
    // class when needed. For this class a lowPass object
    // has to be created.
    lp.reset(new lowPass);
```

```
  }

  // The engine will call this function while parsing audio.
  // Make sure it's fast!
  virtual void process(MULTICHANNELBUFFER & buffer) {
    // Let the parent class decide if it's needed
    // to call create
    createIfNeeded();

    // update the filter frequency
    (*lp).setFrequency(parmFrequency);

    // Whe can't be sure how many channels there are,
    // so loop through all of them
    for (UInt i = 0; i < buffer.size(); i++) {
      // apply lowpass filter on buffer
      buffer[i] = (*lp)(buffer[i]);
    }
  }

private:
  // because parmFrequency can be used by both the frontend and
  // the audio callback thread, it must be an atomic float
  aFlt parmFrequency;

  // When YSE is used as a dynamic library, you want to avoid crossing heap
  // boundaries. For this reason the internal lowPass object is defined as
  // a pointer.
  std::shared_ptr<lowPass> lp;
};
```

If variables are used both in the process function and in functions that will be called from the main thread, it is up to you to make them thread safe. The easiest way to do this is by using std::atomic, or by use of the already provided atomic versions of basic variables: aInt, aBool, aFlt and aVec. Locking mechanisms can also be used but should be avoided whenever possible.

Another thing you should be wary of is memory allocation when working with the dynamic YSE library. The problem lies in the fact that a dll has its own heap. If you create an object of your class in your program and you would use a lowPass object instead of a pointer, the lowPass object would be created on the stack. No problem so far... But if your object is dynamically created, it will be on the application's heap. This means it cannot be used by the dll and your application will crash.

The bottomline is: when you use the static library or you are very sure all dsp objects are created on the stack, go ahead and use objects. Otherwise, use pointers.

Filter objects can be inserted in sounds and channels. They can also be chained.

```
YSE::sound sound;
// .. create sound first ..
sound.setDSP(myFirstFilter).setDSP(mySecondFilter);
```

6.2 Source Objects

Source objects are quite related to filter objects, but are used to generate sound. For this reason the process function is a bit different. The audio buffer will not be passed as an argument, but exists within the class. It is up to you to fill it.

The process function does have an SOUND_INTENT as an argument. This indicates if the engine wants to start, pause or stop the sound.

Another function you will have to implement is the frequency function. This function will be called by the engine to adjust the frequency of your sound source. It is mainly used to create a doppler effect when needed. You should not used it for basic frequency changes.

Info

Example `demo07_dsp_source.cpp` shows you the basic code for a DSP source.

Sounds can be created with a DSP source object instead of a sample:

```
myDSPSource source;
YSE::sound sound;
sound.create(source).play();
```

CHAPTER 7

DSP Primitives

YSE provides a lot of primitive dsp functions and classes. These can be used to build more complex filters and generators without having to start from scratch. Most primitives do their DSP thing in the operator() function. This means you can use them quite easily:

```
// object definition somewhere in your class
YSE::DSP::sine mySine;

// assign sinewave in the process function
buffer = mySine(440);
```

Info

It is our mission to provide as much DSP primitives as we can, but it will take some time. Already there some classes which are not discussed in this manual such as fourier, ADSRenvelope and wavetable. For a complete overview of these classes, consult the header files or open the code examples.

7.1 YSE::DSP::buffer

A buffer contains a block of audio, and by default uses the standard buffersize which is used for audio rendering. They are the foundation of nearly every other DSP class. Basic operators like +,-,/ and * can also be used with buffers.

A buffer contains float values which represent your audio samples. If you need access to these values to modify them directly, you can get a pointer to the first value with the function

`getPtr()`. The current size of the buffer can be retrieved with the function `getLength()`. This is rarely needed though, unless you're working on the engine itself.

Because the buffer class is used everywhere in the engine, we decided to keep it as small as possible. The more complex functions are implemented in derived classes, like drawableBuffer (lets you draw lines inside a buffer), fileBuffer (lets you load and save a buffer directly to disk) and wavetable (which is a filebuffer with extra functions to create fourier tables).

7.2 Oscillators

Standard oscillators are Sine, Cosine, Saw and Noise. They can be used to render an audiobuffer according to a given frequency. Sine and Saw can also take another audiobuffer as input. In this case, the buffer is supposed to contain the frequency for every sample.

Another class in this group is vcf, which is actually a filter but put together with oscillators because of the way it works internally.

7.3 Filters

The supplied filter objects are lowpass, highpass, bandpass and a biQuad filter. The frequency for these filters can be set with setFrequency. The operator() argument is an mono audiobuffer.

The bandpass filter also has a Q value. The biQuad filter is a bit harder to deal with, because the filter itself is more complicated. If it doesn't seem to make much sense, it is probably time to read a bit about biQuad filters first. A nice website for visualizing biquad parameters, is http://www.earlevel.com/main/2010/12/20/biquad-calculator/

The last class in the filter category is a sample & hold class.

7.4 Delay

Delay keeps an internal delay line of variable length. You can read from a delay at any given position, and copy its contents to another buffer. A simple use for this would be an echo effect, but it is also the foundation for a granulator.

7.5 Ramp

Ramps can be used to change a value over time. The resulting audio buffer will not contain a real audio signal, but can be applied to other audio buffers.

```
// in class definition:
YSE::DSP::sine sine;
YSE::DSP::ramp ramp;

// some trigger to set up the ramp
if(trigger) {
  ramp.set(1, 100); // go from 0 to one over 100 milliseconds
}

ramp.update();
buffer = sine(440);
buffer *= ramp();
```

A simplified version of ramp is lint (linear interpolator). The internal value also moves towards its target, but is applied on a float instead of an audio buffer.

These classes aside, a few functions also can be applied to audiobuffers, as long as the required manipulation takes most the length of one buffer:

```
void FastFadeIn(buffer& s, UInt length);
void FastFadeOut(buffer& s, UInt length);
void ChangeGain(buffer& s, Flt currentGain, Flt newGain, UInt length);
```

7.6 Math

To ease the development of DSP objects, YSE offers quite a few mathematical classes. These classes all have their internal buffer and treat incoming signals as constants. These classes are available:

clip clips audio signal at low and high value

rSqrt reciprocal square root of all signal values

sqrt square root of all signal values

wrap calculates difference between a signal and the first exceeding integer

pow the rest is obvious, right?

exp

log

abs

Part III

Making Music

Chapter 8

Virtual synth's

Info

The remaining chapters of this manual describe a few ideas that are not really consolidated and are very incomplete. You're welcome to play with them and encouraged to write feedback on the forum, but they will probably change a lot in future versions of YSE.

To create a synthesizer in YSE, you need either a dspSourceObject or a samplerConfig object. The first one enables you to create your own dspSourceObject (as seen in the previous part of this manual) and have a synth create voices with this object as the sound generator. The second option enables you to use a sound file and create a 'sampler'. For best results, use a `.wav` file as input. Compressed audio is not ideal because it will be played at altered speeds.

After creating a synth object, it must be assigned to a sound to be played. This enables you to have a synth object play just like any other 3D sound. It can be given a position, can be affected by doppler or any other effects, assigned to channels and have reverb applied.

The example code `demo12_synth.cpp` uses a synth object with a sampler configuration assigned to channel one, and a dspSourceObject to channel two. It is used to play random notes. It is also possible to have a synth play a midi file, as in example `demo13_midi_file.cpp`.

CHAPTER 9

Audio Manipulation

DSP filters allow you to alter sound in realtime. But sometimes it can also be useful to alter sound files directly. This can be done by loading a soundfile into a fileBuffer object. Using DSP primitives, the sound can be altered on the spot. Sound objects also accept buffers as a sound source, which allows you to play the file while modifying it. There are no safeguards here, but floats are supposed to be atomic in every modern c++ implementation. Since the sound only reads from the buffer, it should be safe to write to the buffer from within your application as long as you don't change its length.

Example `demo13_AudioFileManipulation.cpp` demonstrates this.

CHAPTER 10

Player

Another concept we're working on is an algorithmic player class. The idea is to have a player play on a virtual synth. The player class doesn't accepts note (midi) input, but can be 'directed'. You can set its minimum and maximum velocity, note durations, the motifs and chords it can use, and a lot more. And all these values can be changed over time. The current implementation is only a proof of concept, but still fun to play with.

Examples 15 and 16 should give you an idea of the possibilities.

CHAPTER 11

Wavetables and ADSR envelopes

To enable more complex virtual synthesizers, we've added wavetables and ADSR envelopes. In fact, our envelope system is based on the ADSR principle, but is much more flexible. Every part of the envelope (Attack, Sustain and Release) can contain as much values as you like, enabling quite complex envelopes.

It is also possible to extract an envelope from an audio file and apply that to another audio file or a DSP source. Combined with the basic wavetable class, this should be enough to emulate some famous '80s instruments like the Yamaha DX series.