Luppp - A real-time audio looping program

Harry VAN HAAREN,
University of Limerick,
Ireland

harryhaaren@gmail.com
http://harryhaaren.blogspot.com
https://github.com/harryhaaren/Luppp

Abstract

Luppp is a live performance tool that allows the playback of audio while applying effects to it in real time. It aims to make creating and modifying audio as easy as possible for an artist in a live situation.

At its core lies the JACK Audio Connection Kit for access to audio and MIDI, usable effects include LADSPA, LV2 and custom Luppp effects. There is a GUI built using Gtkmm, however it is encouraged to use a hardware controller for faster and more intuitive interaction.

Keywords

Live performance, real-time looping, audio production

1 Introduction

Luppp is the "Luppp Untuitive Personal Performing Program", it's a tool to be used in a live scenario to creatively produce music. The fact that it is to be used live imposes certain requirements like low latency input-output and real-time effect rendering.

This paper gives an overview of the structure of the audio processing engine in Luppp and how it delegates tasks to different threads, as well as how communication between the threads is achieved. The loading of effects is discussed, as is the user interface and its design.

2 Engine

2.1 Dependencies

The core engine depends on a number of different libraries:

- JACK [1]
- Gtkmm [2]
- libsndfile [3]
- Fluidsynth [4]
- libconfig [5]
- SUIL [6]
- LILV [7]

2.2 Overview of components

The core of the program consists of a processing engine. Its function is to process all input events and produce its output within a certain time frame. Its primary inputs are MIDI messages and events from the GUI, while its primary output is audio data that is written to the JACK ports. It also facilitates the display of data on screen in the user interface. This involves transporting data from the real-time engine thread to the user interface thread to be displayed.

2.3 Engine Events

The `EngineEvent` class is a class that represents any action that the engine can perform. For each action there is a function to set the right parameters in the class, while reading data from the class is done through the use of public data members.

2.4 Events and ring-buffers

To communicate between the various threads in the engine in a lock-free manner ring-buffers are used. Pointers to instances of the `EngineEvent` class are passed through the ring-buffers, which allows the use of these events from a real-time thread. The GUI thread creates a pool of `EngineEvent` instances while the real-time JACK thread can pull event pointers from this queue, sending the messages without having to allocate them. This causes minimal overhead in the real-time thread while a background thread occasionally polls the ring-buffer for write space, and fills it with new instances.

2.5 State store

The `StateStore` class holds the state of all other components of the engine. It is a centralized location for all data like `AudioBuffers`, `AudioSinks`, `BufferAudioSourceStates` etc. The real-time thread takes `EngineEvents`, and writes the data contained within it to the `StateStore`. This state data is later requested by each engine component when it is required to do its processing.

The requesting of state data is done using unique identification numbers for each state instance. When an instance of a class that needs a state is created, it automatically increments its own ID number, while also adding a new state with its own ID number to the `StateStore`. It can now request the state with its own ID number from `StateStore` and it will receive its own state data.

2.6 Audio processing

The `AudioTrack` is the main structure in audio processing, with the track's `AudioSource` providing a channel of mono audio, a list of `Effects` being applied to this signal, and finally the `AudioSink` passes the produced data to the `Mixer` class. The `Mixer` then takes each track's output, applies master effects and writes the data to the master output ports.

Meta data is available to each element inside an `AudioTrack` so `AudioSource`, `Effect` and `AudioSink` class instances can make informed decisions how to process the next block of audio based on the engines current state.

2.6.1 Audio sources

The `AudioSource` base class is defined as the start of the processing chain. It is subclassed to provide varying functionality: The `BufferAudioSource` reads samples from an `AudioBuffer`, and then writes them to the `AudioTrack`'s buffer. In the case of instrument sources like Fluidsynth or a LV2 synth, it gathers incoming MIDI data from JACK or a playing MIDI clip and generates samples and writes them to the `AudioTrack`'s buffer.

2.6.2 Effects and plugins

The `Effect` base class is defined as a class that requests its state from the StateStore using its own ID, sets its new parameters, and performs some processing on an `AudioTrack`'s buffer. Currently there is no latency compensation done for any of the effects, however the effects that are available have been extensively tested to comply both with the real time requirements of the program, as well as not inducing unacceptable latency to the track.

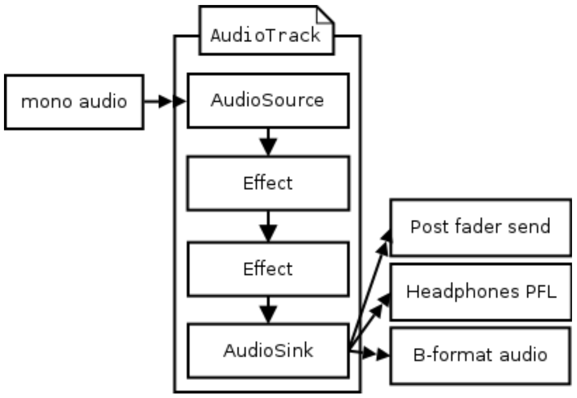

Figure 1: `AudioTrack` block diagram

This base class can be subclassed to host any kind of audio effect, the only constraint is that the number of input samples must match the number of output samples. LADSPA[8] and LV2[9] effects are implemented, however theoretically any type of plugin format could be supported.

2.6.3 Audio sinks

The `AudioSink` class is the base class that represents the end of the processing chain. Every track has an `OutputAudioSink`, whose purpose it is to mix the mono input samples into ambisonic B-format, while also copying the samples to the post-fade send buffer and headphones pre-fade listen buffers.

2.6.4 Headphones PFL

Each track has headphones pre-fade listen functionality, which allows the user to preview tracks before actually mixing them into the master output. This functionality is achieved by writing the mono signal that is passed to the `AudioSink` to the headphones buffer in `Mixer`. After each track has been processed, `Mixer` will write the headphones buffer to the JACK port.

2.6.5 Post-fade Sends

Each track has a post-fade send. All tracks are summed together into a single buffer, which is then written to a JACK audio port. This JACK port can be connected to arbitrary JACK clients to add effects to the signal. Although a mono signal is sent, the return ports are B-format to support scenarios where an ambisonic effect is used. The returned signal is then mixed into the master bus.

2.7 Real-time recording

In order to record an arbitrary length of audio an arbitrary length array is needed to store it, and due to real-time constraints we cannot create one such array in the real-time thread. This problem was solved by writing all incoming audio to a ring-buffer, and allowing the GUI thread to create the actual buffers that are later used in the engine. The buffer is passed to the real-time thread using an event and ring-buffer as described earlier. This allows the user to record any length of audio in real-time without any non-real-time operations occurring.

3 User Interface

The user interface for Luppp is geared towards live use, and hence it makes as little use of popup dialogs and extra windows as possible. The main view is laid out with tracks oriented vertically, and song-parts horizontally.

3.1 Clip view

The main part of the GUI is dedicated to the clip view, which selects the currently playing `AudioBuffer`. To load a buffer into a clip we right click on the desired clip, and we will be presented with a file-chooser dialog. It is also possible to drag-and-drop samples onto the clip using the side-pane file browser. Recording audio from a JACK audio input port is also possible, just record enable the track and click on the clip.

3.2 Track view

The lower part of the window shows the state of the currently selected track. Its `AudioSource` is on the left while any effects are listed to the right. The main parameters of effect can be modified using the mouse, by click-and-drag in the graph area. The X and Y axis of the graph are mapped to the two most used parameters of that effect type.

3.3 Master track

The master track provides feedback as to what scene is currently playing, the rotation and elevation values of the B-format ambisonic output and headphones volume levels.

3.4 Side-pane browser

On the left of the UI there is a browser that allows the previewing of samples, instruments and effects. Drag-and-drop operations from the samples to the clips is supported, as is dropping effects and instruments onto the track view. The browser filters its contents based on file extension, samples must have a .wav extension. The Instrument and Effect lists are hard coded.

4 Meta data files

To inform the engine how long an audio loop is in musical beats, meta data files are used. The files are read by libconfig, and can provide information like the length of the loop in beats, or the key of a loop. This gives the engine a chance to process the audio in a musically meaningful way, like stretching drum loops to match the tempo of the currently playing material.

4.1 Sample meta data

When loading a sample, Luppp attempts to open a file called `lupppSamplePack.cfg` from the same directory as the sample is located. If this file exists, we iterate trough all information in the file, and check if any of the metadata is relevant to the sample that we want to load. If meta data exists about the sample, we set properties on the `AudioBuffer` instance that the sample will get loaded into. These properties can be its number of beats (in the musical sense), or what musical key its in, etc.

Example `lupppSamplePack.cfg` file:

```
luppp :
{
  samplePack:
  {
    name = "SamplePackName";
    numSamples = 1;

    s0:
    {
      name = "sampleName.wav"
      numBeats = 16;
    }
  }
}
```

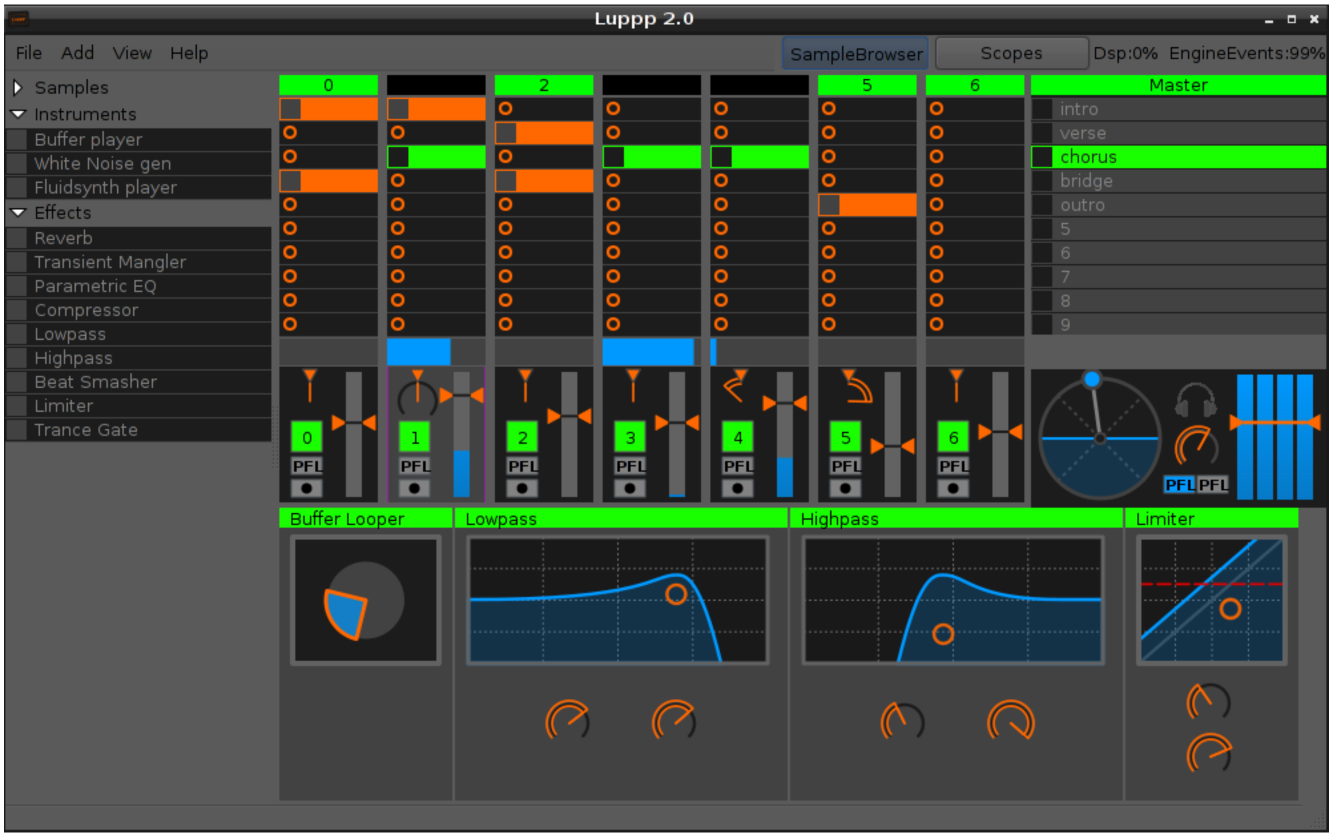

Figure 2: Luppp user interface showing the clip selector, track effects and file browser.

5 Future work

In future I would like to implement more features for Luppp, particularly around easier access to core Luppp functionality from controllers. Also on the todo list is MIDI sequencing functionality which will allow a faster workflow while creating new musical ideas.

Some improvements can be made in performance, and multi-threading the actual audio processing path in the engine is a possibility.

5.1 MIDI sequencing and editing

It is intended to support the looping of MIDI clips in a similar fashion to Seq24. MIDI clips will be shown in the user interface just like audio clips, and clicking on one will select it to be shown in the MIDI editor.

The MIDI editor will be as streamlined as possible, with minimal features and as fast a workflow as is possible. Some initial sketches of possible workflows have been made, however this functionality is still in its planning stages.

5.2 Immutable Engine

For performance reasons it is planned to make `AudioBuffer` instances immutable, so that they can be used in the real-time thread as well as shown in the GUI without expensive copying. Plotting the audio clips in the GUI will become an easier task, as we can safely access the `AudioBuffers` from multiple threads due to its immutability.

To modify an `AudioBuffer` a new one will be created in the GUI thread, and that new instance will be swapped into engine. This modified buffer will keep the same unique ID number, so that all other parts of the engine will automatically use to the updated buffer.

5.3 Sidechain compression

In order to fully support sidechain compression between tracks, it will be necessary to change the direction of sample requesting in an `AudioTrack`. This is due to the fact that for sidechain compression audio data from another track is needed, and hence a method to request the processing of another `AudioTrack` than the current one is needed.

If each `SidechainCompressor` instance has a pointer to a `SidechainSource`, it can request the source to produce the needed samples, and then continue to process its own `AudioTrack`'s audio based on the sidechain buffer of samples. This also involves implementing a system whereby each `Effect` or `AudioSource` can be marked as finished processing, so that we avoid re-calculating parts of a track that has already been processed because of the call to the `SidechainSource`.

Essentially this makes the `SidechainSource` a cache for audio samples, which can be requested to process the next block of audio either from the `AudioTrack` that is it part of, or another `AudioTrack` which needs this `SidechainSource`'s output to complete its own processing.

5.4 Additional effect classes

Initial work to host Pure Data patches or CSound instruments as `AudioSource`s or `Effect`s has went well, although some design is needed with regards to real-time constraints of the effects and loading files. Another issue that needs to be addressed is that non real-time safe operations might be carried out by user defined patches.

VAMP plugins [10] will hopefully be supported in the future, to allow the user analyze existing material, and work with a higher level of control over the music. If more meta data can be provided to the Luppp engine, more high level features can be implemented as more data exists for use during the processing cycle. I feel this is quite an important aspect of Luppp to develop, as it gives the user more scope for creativity and inspiration instead of hindering the workflow with mathematical details of the processing that is going on behind the GUI.

6 Conclusions

While Luppp is currently in a usable state, there are some known issues that must be dealt with. I would like to rethink the method of communication between the `Engine` and the `StateStore` for the user interface. The way that the user interface widgets are stored in the gui class can also be much improved. Some design issues like the concept of tracks and how ID number related to tracks could also be upgraded, as currently some functionality is hindered by how the ID's are set.

7 Acknowledgments

I would like to thank the entire LAD community for all that I have learnt through reading their code, asking questions on the LAD mailing list as well as on IRC.

Thanks also to all those who have helped me in any way while I have been working on Luppp for providing suggestions, testing buggy code and just chatting about the project in general.

8 References

[1] JACK Audio Connection Kit
`http://www.jackaudio.org`

[2] Gtkmm, official C++ interface for the popular GUI library GTK+
`http://www.gtkmm.org`

[3] libsndfile, C library for reading and writing files containing sampled sound
`http://www.mega-nerd.com/libsndfile`

[4] Fluidsynth, a real-time software synthesizer based on the SoundFont 2 specifications
`www.fluidsynth.org/`

[5] libconfig, a simple library for processing structured configuration files
`http://www.hyperrealm.com/libconfig`

[6] SUIL, a lightweight C library for loading and wrapping LV2 plugin UIs
`http://drobilla.net/software/suil`

[7] LILV, a library to make the use of LV2 plugins as simple as possible
`http://drobilla.net/software/lilv`

[8] LADSPA, Linux Audio Developer's Simple Plugin API
`http://www.ladspa.org`

[9] LV2, a plugin standard for audio systems
`http://lv2plug.in/trac/`

[10] VAMP, an audio processing plugin system for plugins that extract descriptive information from audio data
`http://vamp-plugins.org/`