The Playhead

Document version: 0.1

Introduction

The point of this project, program and documentation is to enable the making of an amazing music recording gadget that’s:

- Easy to get started with,
- Has high quality audio,
- Allows for audio-recording and DSP-generated sounds,
- Is extendable – yet sovereign,
- Platform-independent and open-source

There’s a ton of super powerful DAW’s out there. So WHY bother trying to create another one? Well as I see it, I really miss a tool that’s inspiring to work with, has its personal style, with the power of open source, and is always in your pocket. Now Caustic was great. Or almost great.. But that was then. So let’s make something amazing.

Features

Oh there’s a ton, but let’s list a few:

- DSP-units of synths and effects
- Midi-event processors (eventors)
- Stereo processing
- Platform independent
- Built on open-source tools
- Audio recording
- Midi in and OUT!
- Flexible time-signatures, 5/4, 6/8, 7/8, 11/8 etc.
- Real-time effects and audio in for vocoders etc
- Web-based UI

System overview

A program such as a this is a pretty complex piece of software. Building one from a front-end perspective is futile. There’s too many critical aspects that needs to be in place before anything pleasant can be presented to the eye. One of the most fundamental parts that needs to be understood is that in order to have something that’s fun to work with, you want low latency, and in order to get low latency, nothing must interfere when the audio-card requests new data.

Modern CPU’s has multiple cores but the irony is that in order for them to work effectively you need to rely on a task-scheduler that in itself is time-consuming, and any execution passed through different threads (or cores) will easily take more time than you gain. That is - if you’re not careful in your design.

For any newcomers to the C++ world, lets draft an overview of the application in a conceptual way – and what could be a better analogy than – a recording studio!

The Threads Studio

(Maybe this part is just silly… But an introduction to the challenges is needed)

Ok. So you want to do some recording. Off to the recording studio. At the heart of the studio is the room where you do the actual recording but there’s more to it right. You need coffee, cables and electricity. All has to be in place in order for any recording to be successful. It’s time to present two of our characters.

The Studio Runner

The studio runner does a number of tasks such as prepares keyboards and guitars. This (thread) is however not allowed to enter the recording room. Fix coffee and cables. And in the context of our application: Attach midi-keyboards, check if files needs to be written or read, make coffee. And buy coffee. Yes anything that just should be in place but isn’t time-critical as long as it’s in place – that’s assignments for the studio runner.

The PlayerEngine

The player engine is what’s inside the recording studio. Once you start it, well if it isn’t fed with what it wants and needs, it will produce unpleasant sounds. However, as long as you provide what it requires – it can do wonders. Here’s what it’s responsible for:

- Grabbing any incoming audio that should be processed or recorded.
- Processing patterns and timing – checking for any events that should be processed.
- Processing midi-in, generating amazing low-latency playback – even on modest hardware.
- Iterating over “Racks” (kind of like tracks but think Reason) generating stereo audio from recorded audio, synth events and effects.

Then there’s more but let’s stick to the basics right?

(The Racks)

The racks aren’t threads of their own, but controlled by the PlayerEngine. They are the containers of synths and effects. And they have their pattern players. In fact, every rack follow a specific order execution:

PatternData + MidiIn > Eventor1 > Eventor2 > Synth > Effect1 > Effect2 > Emitter

Quite easy right? An event may be in a pattern or coming from a keyboard, it may be enhanced once or twice before reaching the synth, then the generated sound can be modulated by one or two effects, until it finally gets processed by the emitter (aka mixer channel).

- Hey! Isn’t this a perfect landscape for some parallel execution, exercising all those eight cores in the CPU! No, sorry. There’s no time for such debauchery. All low-latency rendering must be kept it in a single thread, any kind of actions that would involve the OS-scheduler or memory management could create lag / clicks / noise in the audio-stream.

The other blokes

Besides the two threads you’ve just met – the StudioRunner and PlayerEngine, there’s some more people running around in the room.

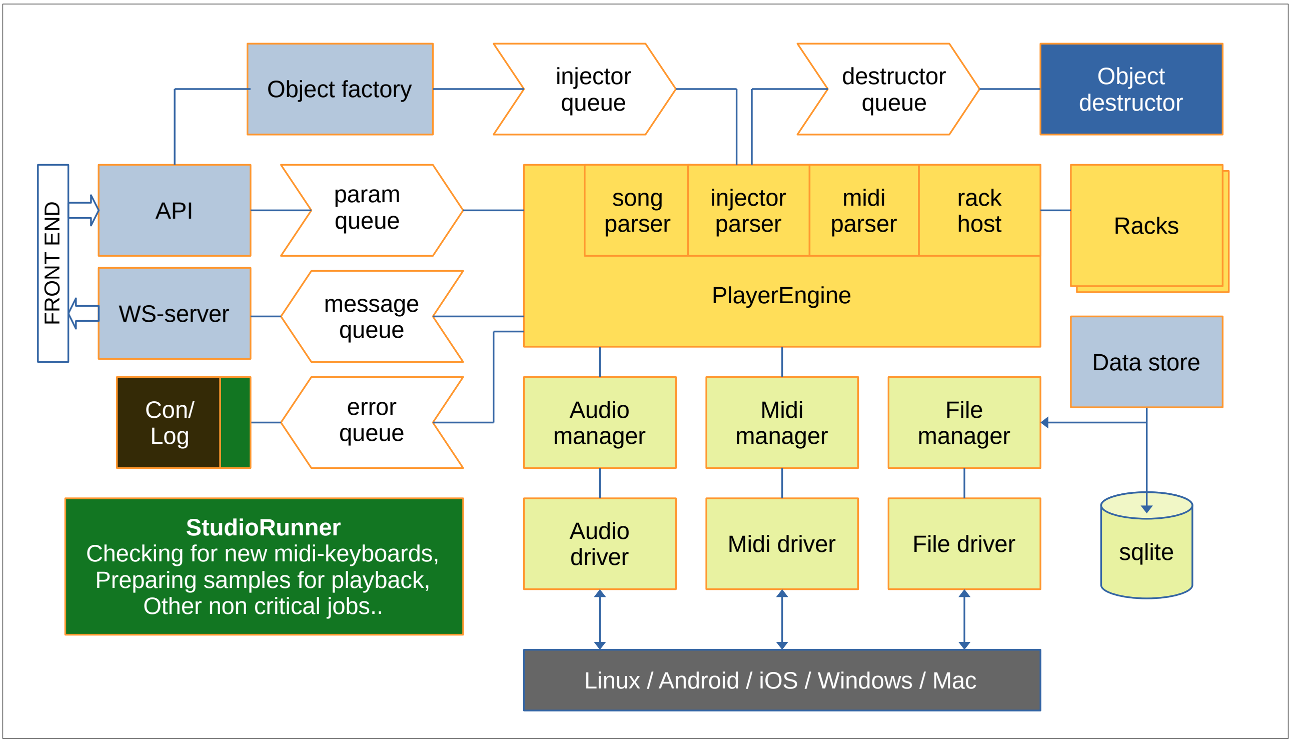
Garbage Guy
He get’s a list of things he should destroy and does so very effectively making sure that old synths that have done their time doesn’t occupy space in the studio.

Message Gal
The Message-Gal has a collection of messages it tries to feed into the player-engine. Actually, its more like you put a message on a plate and when the player-engine feels for digesting it, so it does. Messages may come out too. The message Gal spread these news to the surrondings, over a web-socket server.

Error Earl
When things doesn’t go according to plan, player-engine doesn’t bother to much about it. It just spits out a short message, and it’s up to error-earl to make sure that the error reaches somebody with some sense of responsibility.

Application system diagram

Ok, enough with the studio metaphor, let's look at how this is actually designed:



Different threads represented by different colours

Threads

The different threads are represented with different colours. The PlayerEngine is the central part and every one around is more or less servants.

Queues

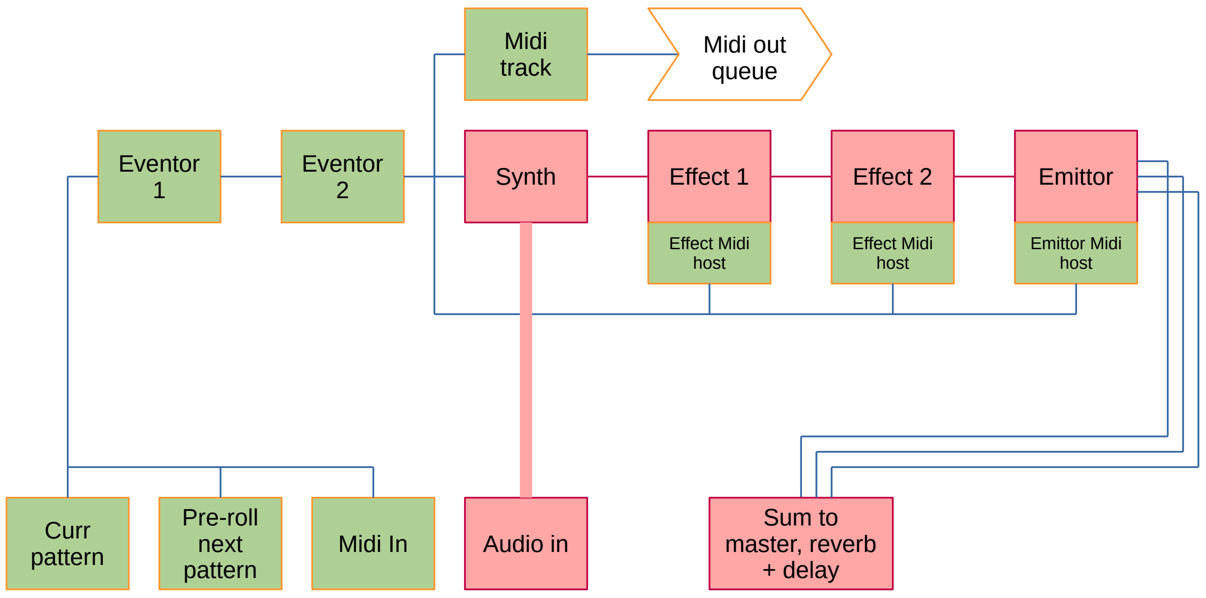
The white arrowed boxes are queues - a central part when working with multi-threaded systems. Mutexes must be avoided, at least from the perspective of PlayerEngine.

Object factories & destructors

The PlayerEngine is way too occupied to have time for memory management, or even worse – storage drives, like loading samples. Everything needs to be prepared outside the audio-thread, and served on a silver-plate in order for the PlayerEngine to digest it. If an object is swapping out another, the old object is thrown on the destructor queue. It's the job of the destructor to do the actual memory cleanup. PlayerEngine won't care about such worldly matters.

Rack system diagram

The rack is where the actual DSP-action happens. This is what it looks like:



Patterns with pre-roll

Pre-rolling in midi-patterns is for intros and stuff. After all, not all starts with one. The midi-in (from pattern-player) has already been filtered to only cover what's relevant for this rack.

Eventors

Eventors may do quanitzation, arpeggiation, all sorts of midi-stuff. Note that they are *after* the pattern-player, thus can't really control what's being recorded to the patterns. More on that later..

Midi out

Midi out is tricky. It has to be in time-sync with what's leaving the audio-card, not what's being processed in the DSP. iOS has great built in support for this, Android doesn't. So only chance to succeed here is to have a queue.

Midi for effects

Effects doesn't understand MIDI. But the host they're sitting in does. Think of the host having robot arms controlling the effect that has no midi-knowledge. It's a design choice making implementation of effects easier, yet allowing the knobs to be adjusted over midi.

Emitter

Emitter is the "channel-mixer", creating a stream for the master bus, the reverb bus, and delay bus.

API

The API is hosted by a web server and uses different protocols for different purposes:

- Static .html and .js files are delivered to frontend from endpoint /fe/*
- Web-socket commnunication for low-latency is communicated over /ws/*
- RPC-JSON performs larger state changes to units, racks or songs.
- REST endpoints are used for various read-only operations.

/fe (GET only)

Not much here to say really. The frontend user-agent (browser) needs some graphics and stuff. It’s loaded from here.

Somehow there needs to be a way to load customized front-end assets, images etc for theming. The idea is that these should reside in the /user-directory. Possibly, a REST get-endpoint could be used to pick the appropriate resource file from either the user-directory or the default asset.

/ws (n/a)

For low-latency operation, such as striking a note and getting a visual feedback (through the audio-server), web-sockets are used.

/rpc (POST only)

One endpoint to rule them alL and really a POST-only endpoint. To simplify interaction during development, a get-method is available too. The POST-format has only one mandatory member - method.

RPC object types

Object types must have a corresponding factory. They are the once carrying out the offline work. Hard workers not receiving much credit. They are: server, *device, project, rack, unit, pattern, song*

server	Control the audio- and RPC-webserver.
device	Device specific settings like prefered audio-interface, buffer size etc
project	Main attributes like tempo, master-tune etc.
rack	Manages setting for a specific rack
unit	Manage settings and parameters for a specific DSP-unit
pattern	Data for a certain pattern (not written)
song	Data for the chaining of patterns into a song (not written). Um – this may be rack based too. So “chain”.

Examples of actual endpoints will be added to the online api reference of the server. With a running copy of ThePlayhead you can reach the api-reference at: http://localhost:18080/fe/api-ref.html

Device management

In order to make ThePlayhead as platform independent as possible and yet powerful, drives has been chosen with care. The drivers needed are Audio, Midi and File-management.

Audio

For audio the PortAudio library is used. Downloading portaudio includes a lot of examples on how to use it but essentialy it about querying the OS about available audio-interfaces and connecting to one of these. PortAudio can be found here: https://github.com/PortAudio/portaudio

AudioManager – the clutch

The AudioManager sits on top of the audio-driver and the idea is that only AudioManager needs to be aware of PortAudio. Everything else in the application is agnostic to what type of driver is being used. The audio-manager also contains a “clutch” that may *disengage the wheels from the motor*, or in our context – disengage incoming audio stream from the PlayerEngine application.

This enables powerful routing of audio streams, for example for benchmarking audio, without having to restart the audio drivers. Likewise, audio drivers can be restarted without causing effects in rest of the application (except a slight pause in the playback of course).

The default audio-device may be setup in the file *device.json* . Note that only one audio device can be connected at a time. There is no support for multiple devices. This also means that a device with only audio-in will be hard to get going.

Sampling frequency

The audio sampling frequency is 48 kHz as this is standard on mobile devices. Currently, not much efforts is being made to allow other sampling frequencies.

Note that the DSP-sampling frequency *may* be halfed to 24 kHz. This feature is to enable heavy processing on thinner devices. All recorded audio will still be recorded in 48 kHz but then all DSP-processing will be made in 24 kHz. Finally, the processed signal is oversampled back to 48 kHz before sent to audio-card. This feature is currently not highest priority but may be later as application complexity increases.

Test scripts

WARNING: Check level of our PC before running these scripts. Take good care of your ears!

The folder /lab contains some test scripts for port-audio. To run them enter the folder and use the shell-script “run”.

```
/lab$ ./run pa_list.cpp       # lists your available audio devices
/lab$ ./run pa_seamless.cpp   # switches between simplex and duplex audio
/lab$ ./run pa_fuzz3.cpp      # real time fuzz effect
```

If you want to experiment with Jack, start JackCtl first and confirm using pa_list above that it’s running. Update device_id accordingly in for example pa_fuzz3.cpp.

Midi

For Midi, the library RtMidi is used, available here: https://github.com/thestk/rtmidi

RtMidi works in team with underlying system. The device.json does NOT contain any information regarding device to select, so it connects to anything – up to five devices simultaneously.

Test scripts

The folder /lab contains a simple test script for midi. You need to prepare for it to work: as before, to them enter the folder and use the shell-script “run”.

```
/lab$ ./run midilog.cpp       # outputs midi messages from devices
```

Now open a **second terminal** and write (Linux only):

```
# this should list your midi keyboard and “RtMidi input client”
/lab$ aconnect -l

# will output something like:
client 0: 'System' [type=kernel]
    0 'Timer           '
      Connecting To: 144:0
    1 'Announce        '
      Connecting To: 144:0
client 14: 'Midi Through' [type=kernel]
    0 'Midi Through Port-0'
      Connecting To: 128:0
client 128: 'RtMidi Input Client' [type=user,pid=145519]
    0 'RtMidi Input    '
      Connected From: 14:0
client 129: 'Virtual Keyboard' [type=user,pid=145457]
    0 'Virtual Keyboard'
client 144: 'PipeWire-System' [type=user,pid=1369]
    0 'input           '
      Connected From: 0:1, 0:0
client 145: 'PipeWire-RT-Event' [type=user,pid=1369]
    0 'input           '

#use id’s accordingly
/lab$ aconnect 129:0 128:0
```

In your first console-window (running midilog.cpp) you should now see midi-messages when pressing notes on your midi-keyboard.

When your done, you may disconnect the midi-routing in second terminal using:

```
/lab$ aconnect -d 129:0 128:0
```

Hopefully you get this working and you can have some fun with it!

MidiManager

Just as in Audio, there’s a MidiManager with the purpose of having the rest of the application agnostic and also improve stability. In ThePlayhead, these managers are supervised by “StudioRunner”. So this is the thread/guy/gal that enables the hotplugging of Midi-devices.

File management

File management shouldn’t be such a big deal right? The dealbreaker here however, is that different file-systems have different rules on how files may be named. Regarding upper- and lowercase, reserved filenames etc. Writing files uncontrolled would also open up for security concerns. For this reason there need to be an isolation layer when writing files, either using index-files resolving names to numeric filenames, or a sqlite database, where smaller “files” are stored.

Currently, the solution for this hasn’t been settled. Using sqlite has its pros and cons. So TBA.

Another file concern: Any sample file that should be played needs to be fully or partially pre-loaded. Also, any audio-in that should be recorded needs buffering before being written to disk. (We’re not limited recording only to RAM here!) But this is really managed by other services of the application, so the main thing here is to provide a platform independent way of storing files with possibly akward names.

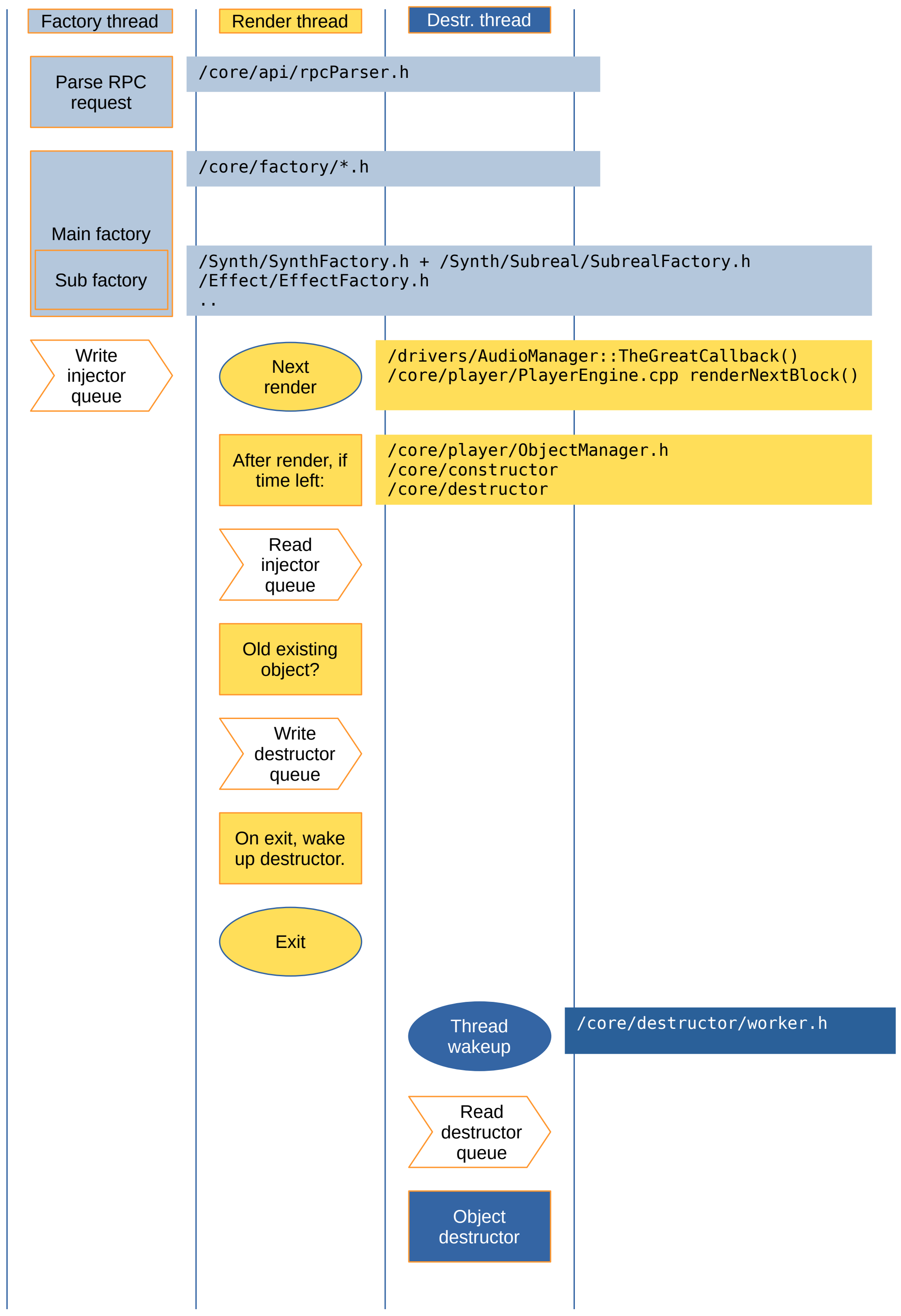
Drivers for mobile platforms

PortAudio supports Linux, Windows and OSX but not Android. We need something different here. Android Audio isn’t great but there’s a project called Oboe, https://github.com/google/oboe . So it’s going to get complicated, but it’s doable – with a little help from my friends right?

Ok, that’s all regarding the drivers. Now go back to the overview at page 4. It’s pretty clear right?

Object construction

To be elaborated, but a quick overview of the process of creating and mouting objects.



Swim-lane diagram over constucting objects and passing them to player engine

Data store

Another quite complex thing is the unserialization of projects and patches. Rough overview:

1) RPC recieves project.load, /core/api/rpcParser.h

2) Request delegated to /core/factory/project.h

3) Data store is called, trying to unserialize the data of the file-name, /core/storage/DataStore.h

4) DataStore gets help from DocumentManager (/core/storage/DocumentManager.h) to read the actual file. Could be in file-system or sqlite or above the clouds. DocMan gets it.

5) With project json file now unserialized into DataStore, back to project, doing the actual object setups. Same factories used by object-constructor is used here.

TO BE IMPROVED. Setup of racks should be delegated, allowing rack-patches and ordinary synth patches. Currently there’s no plan to give eventors and effects patch-support other than using rack-patch.