This document no longer lives here, and may be out of date. Please view or edit it at its new home on GitHub.

Here are brief explanations of how a few things work in the Deluge codebase as of June 2023. This is very minimal documentation, and you'll need to dig deep into the codebase to see how everything works.

These sections also appear as comments in the relevant .h files in the codebase.

By Rohan Hill, sole Deluge developer until open source release in June 2023.

Audio / CPU performance

A huge number of factors influence the performance of a particular Deluge firmware build. Subpar performance will usually be noticed in the form of sounds dropping out in songs which performed better in other firmware versions. Ensuring optimal performance of any code modifications you make, and subsequently their builds, will be a big challenge. But don't sweat it too much - if you've added some cool features which are useful to you or others, maybe slightly lowered audio performance, which is only noticeable in certain circumstances, is a reasonable tradeoff?

The Deluge codebase, since 2021, has been built with GCC 9.2. I (Rohan) compared this with the other 9.x GCC versions, some 10.x ones, and the 6.x version(s) that the Deluge had used earlier. Performance differences were negligible in most cases, and ultimately I settled on GCC 9.2 because it resulted in a built binary which was smaller by a number of kilobytes compared to other versions. GCC 10.x was particularly bad in this regard.

The build process includes LTO - this helps performance a fair bit. And normal O2 optimization. These are both disabled in the HardwareDebug build configuration though, for faster build times and ease of code debugging. If you're using live code uploading via a J-link and want to do some tests for real-world performance, you should enable these for this configuration, at least while doing your tests.

A few other of the standard compiler optimizations are enabled, like –gc-sections to remove unused code from the build. Beyond that, I've experimented with enabling various of the most advanced GCC optimizations, but haven't found any that improve overall performance.

Audio rendering

The Deluge renders its audio in “windows” (or you could more or less say “buffers”), a certain number of samples long. In fact, the Deluge’s audio output buffer is a circular, 128-sample one, whose contents are continuously output to the DAC / codec via I2S (Renesas calls this SSI) at 44,100hz. That buffer is an array called ssiTxBuffer.

Each time the audio routine, AudioEngine::routine(), is called, the code checks where the DMA outputting has gotten up to in ssiTxBuffer, considers where its rendering of the previous “window” had ended at, and so determines how many new samples it may now render and write into ssiTxBuffer without “getting ahead of itself”.

This scheme is primarily useful because it regulates CPU load somewhat, for a slight tradeoff in audio rendering quality (more on that below) by using longer windows for each render. For instance, if the CPU load is very light (as in, not many sounds playing), rendering a window will happen very fast, so when the audio routine is finished and is then called again the next time, only say one or two samples will have been output via DMA to the DAC. This means the next window length is set at just one or two samples. I.e. lighter CPU load means shorter windows.

Often the window length will be rounded to a multiple of 4, because various parts of the rendering code (e.g. oscillator table lookup / interpolation) use Arm NEON optimizations, which are implemented in a way that happens to perform best working on chunks of 4 samples. Also, and taking precedence over that, windows will be shortened to always end at the precise audio sample where an event (e.g. note-on) is to occur on the Deluge’s sequencer.

The window length has strictly speaking no effect on the output or quality of oscillators (including sync, FM, ringmod and wavetable), filters, most effects, sample playback, or timings/jitter from the Deluge’s sequencer. Where it primarily does have a (barely audible) effect is on envelope shapes. The current “stage” of the envelope (i.e. A, D, S or R) is only recalculated at the beginning of each window, so the minimum attack, decay or release time is the length of that window, *unless* that parameter is in fact set to 0, in which case that stage is skipped. So, you can get a 0ms (well, 1 sample) attack time, but depending on the window length (which depends on CPU load), your 0.1ms attack time could theoretically get as long as 2.9ms (128 samples), though it would normally end up quite a bit shorter.

With envelopes (and LFO position actually) only being recalculated at the start of each “window”, you might be wondering whether you’ll get an audible “zipper” or stepped effect as the output of these changes sporadically. The answer is, usually not, because most parameters for which this would be audible (e.g. definitely anything level/volume related) will linearly interpolate their value change throughout the window, usually using a variable named something to do with “increment”. This is done per parameter, rather than per modulation source, essentially for ease / performance. Wavetable position and FM amounts are other important parameters to do this to.

But for many parameters, the stepping is not audible in the first place, so there is no interpolation. This includes filter resonance and oscillator / sample-playback pitch. Especially

sample-playback pitch would be very difficult to interpolate this way because the (unchanging) pitch for the window is used at the start to calculate how many raw audio-samples (i.e. how many bytes) will need to be read from memory to render the (repitched) audio-sample window.

Sounds, Instruments and “Drums”

An **Instrument** is the “Output” of a Clip - the thing which turns the sequence or notes into sound (or MIDI or CV output).

Instruments include Kit, MIDIInstrument, and CVInsttrument. And then there’s SoundInstrument, which is basically a synth - more on that below.

Kits are made up of multiple **Drums**. Even when they are not drum sounds, the class is called Drum, for better or worse. In most instructional material for users, Synthstrom has referred to them often as “items within kits”, or sometimes “rows” or “sounds” where applicable.

Types of Drum are MIDIDrum, GateDrum, and SoundDrum (most often a sample, but we’ll talk more about that).

And then there is the class called **Sound** - which can be either an Instrument or a Drum, in the form of SoundInstrument or SoundDrum respectively. These classes are implemented using “multiple inheritance”, which is sacrilegious to many C++ programmers. I (Rohan) consider it to be a more or less appropriate solution in this case and a few others in the Deluge codebase where it’s used. It’s a little while though since I’ve sat and thought about what the alternatives could be and whether anything else would be appropriate.

Anyway, Sound (which may be named a bit too broadly) basically means a synth or sample, or any combination of the two. And, to reiterate the above, it can exist as a “synth” as the melodic Output of one entire Clip(s), or as just a Drum - one of the many items in a Kit, normally associated with a row of notes.

ModelStacks

This is a system that helps each function keep track of the “things” (objects) it’s dealing with while it runs. These “things” often include the Song, the Clip, the NoteRow - that sort of thing. This was only introduced into the Deluge’s codebase only in 2020 - some functions do not (yet) use it. Its inclusion has been beneficial to the codebase’s ease-of-modification, as well as code tidiness, and probably a very slight performance improvement.

Previously, the Deluge's functions had to be passed these individual "things" as arguments - a function might need to be passed a Clip and an AutoParam, say. However, if I later decided that a function needed additional access - say to the relevant ParamCollection, this could be tiresome to change, since the function's caller might not have this, so its caller would have to pass it through, but that caller might not have it either - etc. Also, all this passing of arguments can't be good for the compiled code's efficiency and RAM / stack / register usage.

Another option would be for each "thing", as stored in memory to include a pointer to its "parent" object. E.g. each Clip would contain a pointer back to the Song, so that any function dealing with the Clip could also find the Song. However, this would be unsatisfactory and inefficient because RAM storage and access would be being used for something which theoretically the code should just be able to "know".

Enter my (Rohan's) own invented solution, "ModelStacks" - a "stack" of the relevant parts of the "model" (objects representing the makeup of a project on the Deluge) which the currently executing functions are dealing with. Things can be "pushed and popped" (though the implementation doesn't quite put it that way) onto and off the ModelStack as needed. Now all that needs to be passed between functions is the pointer to the ModelStack - no other memory or pointers need copying (except in special cases), and no additional arguments need to be passed. The ModelStack typically exists in program stack memory.

For example, suppose a Song needs to call a function on all Clips. The ModelStack begins by containing just the Song. Then as each Clip has its function called, that Clip is set on the ModelStack. And suppose each Clip then needs to call a function on its ParamManager - that's pushed onto the ModelStack too. So now, if the ParamManager, or anything else lower-level, needs access to the Song or Clip, it's right there on the ModelStack. The code now just "knows" what this stuff is, which I consider to be the way it "should" be: a human reading / debugging / understanding the code will know what these higher-up objects are, so why shouldn't the code also have an intrinsic way to "know"?

This is additionally beneficial because, suppose we decide at some future point that there needs to be some new object inserted between Songs and Clips - maybe each Clip now belongs to a ClipGroup. We can now mandate that the addClip() call is only available on a newly implemented ModelStackWithClipGroup, for which having a ClipGroup is now a prerequisite. By simply trying to compile the code, the compiler will generate errors, showing us everywhere that needs to be modified to add a relevant ClipGroup to the ModelStack - still a bit of a task, but far easier as it will only be functions at higher-up levels that need to add the ClipGroup, and then we can just take it for granted that it's there in the ModelStack. The alternative would be having to modify many functions all the way down the "tree" of the object / model structure, to accept a ClipGroup as an argument, so that it can be passed down to the next thing / object.

Another advantage is that error checking can be built into the ModelStack - which may also be easily switched off for certain builds. For example, there are many instances in the Deluge codebase where ModelStackWithTimelineCounter::getTimelineCounter() is called - usually to get the Clip (TimelineCounter is a base class of Clip). We know that the returned TimelineCounter is not allowed to be NULL. Rather than insert error checking into every

instance of such a call to ensure that it wasn't passed a NULL, we can instead have getTimelineCounter() itself perform the check for us and generate an error if need be, all in a single line of code.

One disadvantage is that some simple function calls on a "leaf" / low-level object such as AutoParam now require an entire ModelStack to be built up and provided, even if the function only in fact needed to know about one parent object - e.g. the Clip. However, in practice, I've observed very few cases where ModelStacks get populated unnecessarily - especially as ModelStacks are implemented more widely throughout the codebase, so most functions already have a relevant ModelStack to pass further down the line.

Another potential pitfall - suppose a "leaf" / low-level object - say AutoParam - needs to call a function on its parent ParamCollection. In this sort of case, which is very common too, the ModelStack is passed back upwards in the "tree hierarchy". But now, what if this function in ParamCollection now needs to do something that requires calling a function on each of its AutoParams? If it sets the AutoParam on the ModelStack, then the original AutoParam - to which execution will eventually be returned - is no longer there on the ModelStack, which may break things and we might not realise as we write the code. Ideally, I wish there was a solution where we know that so long as the code compiles, we're not at risk of overwriting anything on the ModelStack that might be needed. I couldn't devise a nice solution to this other than just exercising caution as the programmer. My memory doesn't quite serve me here - I experimented with having functions only accept a const ModelStack* (which is now the case for many of the functions and I can't quite remember why, sorry!) but this somehow did not provide a solution for the code to be immune to the pitfall identified above.

I'm actually not sure how fields like game development deal with similar problems, which they must encounter as e.g. a "world" might contain many "levels", which might also contain many "enemies" - a tree-like structure which the code must have to traverse, like on the Deluge. I tried Googling it, but couldn't find anything about a standard approach to this. Perhaps the each-object-stores-a-pointer-to-its-parent solution, as I mentioned above, is the norm? If you know, I'd be really interested to know!

SD card audio streaming

Audio streaming (for Samples and AudioClips) from the SD card functions by loading and caching Clusters of audio data from the SD card. A formatted card will have a cluster size for the filesystem - often 32kB, but it could be as small as 4kB, or even smaller maybe? The Deluge deals in these Clusters, whatever size they may be for the card, which makes sense because one Cluster always exists in one physical place on the SD card (or any disk), so may be easily loaded in one operation by DMA. Whereas consecutive clusters making up an (audio) file are often placed in completely different physical locations.

For a Sample associated with a Sound or AudioClip, the Deluge keeps the first two Clusters of that file (from its set start-point and subject to reversing) permanently loaded in RAM, so playback of the Sample may begin instantly when the Sound or AudioClip is played. And if the Sample has a loop-start point, it keeps the first two Clusters from that point permanently loaded too.

Then as the Sample plays, the currently-playing Cluster and the next one are kept loaded in RAM. Or rather, as soon as the “play-head” enters a new Cluster, the Deluge immediately enqueues the following Cluster to be loaded from the card ASAP.

And then also, loaded Clusters remain loaded/cached in RAM for as long as possible while that RAM isn’t needed for something more important, so they may be played again without having to reload them from the card. Details on that process below.

Quick note - Cluster objects are also used (in RAM) to store SampleCache data (which caches Sample data post-repitching or post-pitch-shifting), and “percussive” audio data (“perc” for short) which is condensed data for use by the time-stretching algorithm. The reason for these types of data being housed in Cluster objects is largely legacy, but it also is handy because all Cluster objects are made to be the same size in RAM, so “stealing” one will always make the right amount of space for another (see below to see what “stealing” means).

Memory allocation

The Deluge codebase uses a custom memory allocation system, largely necessitated by the fact that the Deluge’s CPU has 3MB ram, plus the Deluge has an external 64MB SDRAM IC, and both of these need to have dynamic memory allocation as part of the same system.

The internal RAM on the CPU is a bit faster, so is allocated first when available. But huge blocks of data like cached Clusters of audio data from the SD card are always placed on the external RAM IC because they would overwhelm the internal RAM too quickly, preventing potentially thousands of small objects which need to be accessed all the time from being placed in that fast internal RAM.

Various objects or pieces of data remain loaded (cached) in RAM even when they are no longer necessarily needed. The main example of this is audio data in Clusters, discussed above. The base class for all such objects is Stealable, and as the name suggests, their memory may usually be “stolen” when needed.

Most Stealables store a “numReasonsToBeLoaded”, which counts how many “things” are requiring that object to be retained in RAM. E.g. a Cluster of audio data would have a “reason” to remain loaded in RAM if it is currently being played back. If that numReasons goes down to 0, then that Stealable object is usually free to have its memory stolen.

Stealables which in fact are eligible to be stolen at a given moment are stored in a queue which prioritises stealing of the audio data which is less likely to be needed, e.g. if it belongs to a Song that’s no longer loaded. But, to avoid overcomplication, this queue is not adhered to in the case where a neighbouring region of memory is chosen for allocation (or itself being stolen) when the allocation requires that the object in question have its memory stolen too in order to make up a large enough allocation.

Works in progress, or stuff in an un-ideal state

In many places, a class with a singular object has been used where a namespace would have made more sense. I (Rohan) was a total C++ noob when I started the Deluge! E.g. MidiEngine is still a class. A few of these, I have converted to namespaces, such as AudioEngine.

Various function calls, such as freezeWithError(), still always go through NumericDriver, even when OLED is used instead.

Various classes, namespaces or files do things seemingly unrelated to their stated purpose (like PlaybackHandler), or don't have a clearly enough defined purpose. E.g. what the heck is View? I promise it used to be an actual thing, in the early days of Deluge development!

In various places, 8-bit or 16-bit ints are used unnecessarily because in the early days of Deluge development I mistakenly believed that using less bits gave better or faster performance. But this is almost never the case and sometimes the reverse is even true! Processors perform best on numbers in their native number of bits - which of course is 32 bits for a 32-bit processor like the Deluge's Renesas RZ/A1L. But, this code can't just be replaced wholesale with the guarantee that it'll function exactly the same - each change would need to be thought about and tested. So I've only changed these sometimes while refactoring parts of the code.

From Aria Burrell: Cross-platform build scripts and supported development in other operating systems and editors are on the horizon. I hope to test and release these changes within a week of the Open Source launch. They may affect some of the workflows discussed in this document.

Quirks

For reasons not exactly known, globally declared instances of classes (so, objects) will not get their constructors called automatically on boot-up as is supposed to happen in C++. This will immediately cause problems, as things don't get initialized. And for classes with virtual functions (i.e. using polymorphism), their vtable won't even be set, causing an instant crash as soon as any virtual function is called on them.

This is why, in Deluge.cpp, every single globally declared object gets manually set up with a "new" statement, like `new (&instrumentClipView) InstrumentClipView;`

See a more technical discussion of that problem here.