libgig 0.6.0

by Christian Schoenebeck <cuse@users.sourceforge.net>

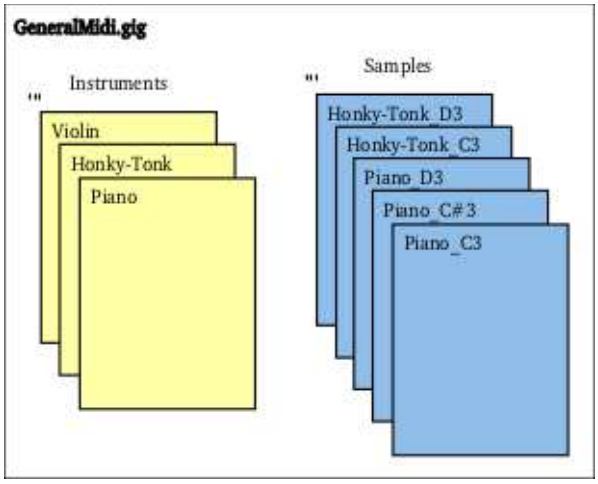
Preface

This is only a quick introduction to libgig. Have a look at the UML diagram and the API documentation for detailed descriptions. I will add more informations here if I find the time for it. Let me know if you like to add some sentences to this documentation!

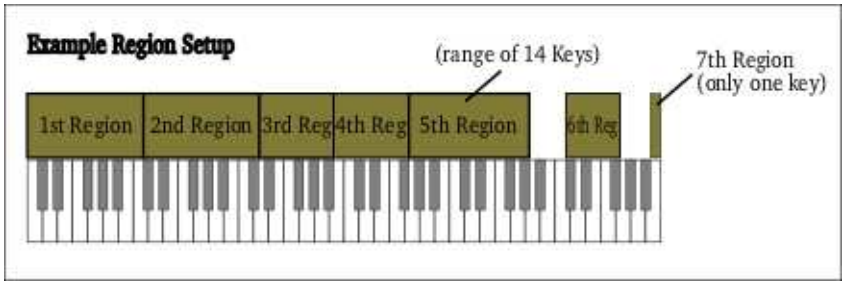
Format Overview

The Gig format is based on the DLS (Downloadable Sounds) file format, but few parts of this open format are left. Many things are dropped, some were added.

A Gig file consists of Samples and Instruments. The latter provide the abstract articulation data how and of course which sample(s) should be played. It's possible that not all samples are used by one of the defined instrumenst. That's why you can access the list of samples independently of the instruments with libgig.



Instruments consist of two levels: Regions and Dimension Regions (the latter are also referred as Subregions or Cases, in libgig however they're called Dimension Regions). But first the Regions:



In DLS format, regions are allowed to overlap on the keyboard and can also have velocity ranges defined and can thus be used for velocity splits. For the Gig format the developers

completely dropped that; Regions are not allowed to overlap on the keyboard and they're just defined by their key range. That takes a bit of flexibility, but on the other hand it makes it easier to get the corresponding region that belongs to a triggered key (if there is one actually defined for the triggered key). And as the region is just identified by the key position, the new allocated voice will have that single one region to the end of it's life, so it won't ever change due to a modulation source (e.g. MIDI controller) or time. Regions themselves though provide almost no articulation data. You have to identify one of it's Dimension Regions that fits to the current situation, which is dependent to 0 - 32 influences, called Dimensions. Usually a Dimension is a MIDI controller, but can also be an audio channel, a layer (for layering instruments), key release, key velocity, key pressure or the pressed key number (position on the keyboard).

Dimension Regions are therefore something like subregions of a region. Dimension Regions contain almost all articulation data for Gig instruments and also the reference to a sample (there is also some kind of global sample reference stored in Regions, but I wonder about it's sense and I'm not sure if it's used by the Gigasampler engine anyway, as every Region has at least one Dimension Region, I recommend always to use the sample reference stored in the Dimension Region instead of the ones stored in the Regions). The amount of Dimension Regions depends on the number of Dimensions which are defined for a Region (don't mix the term “Dimension” and “Dimension Region”). A Region has exactly either 1, 2, 4, 8, 16 or 32 Dimension Regions, so is always a power of two. There can be 0 – 5 dimensions defined for each region independantly.

Each dimension has a specified number of splits. A split is also called “Bit”. You will see where this term came from later when I'll show you how to calculate the sought dimension region. If a dimension has 0 Bits, the dimension has one zone, thus no split of the dimension's value range. If a dimension has 1 Bit, it has 2 zones, 2 Bit = 4 zones, 3 Bit = 8 zones, 4 Bit = 16 zones, 5 Bit = 32 zones. The latter is also the maximum of possible zones.

The amount of actual dimension regions calculates by multiplying the numbers of all dimension zones, thus by:

```
DimensionRegions = Dimension0 Zones * Dimension1 Zones * Dimension2 Zones *
                   Dimension3 Zones * Dimension4 Zones
```

Example: If we have only one velocity dimension with 4 zones, we have only 4 dimension regions. If we would have a 2nd dimension modulation wheel with 8 zones we'd have `4 * 8 = 32` dimension regions.

Each dimension region has it's own articulation data. So to identify the correct dimension region for the current case, you have to take the current MIDI controller values of all defined dimension into account or which zone is selected by that. The sought dimension region (index) now results by the following Horner-scheme:

```
DimensionRegion = (((Dim4Zone * Dim3Bits + Dim3Zone)
                               * Dim2Bits + Dim2Zone)
                               * Dim1Bits + Dim1Zone)
                               * Dim0Bits + Dim0Zone
```

So maybe you now got the idea where the term “Bit” came from when referring to a split.

Usually all dimension regions from the same region have the same sample reference. The exception here is the velocity dimension which is also allowed to be linked to different samples, same applies to the layer dimension.

Now we got the rough thing, so we come how you actully use the library in your engine to play Gigs correctly (and don't worry about such calculations mentioned above, those are already embedded into the library).

Using the library in your engine

When receiving a note on signal you get the region by just doing:

```
gig::Region* pRegion = pInstrument->GetRegion(MIDIKeyNumber);
```

If there is no region defined for the triggered key, the method returns `NULL` and you just ignore the note on signal in your code. Otherwise you now have to identify the corresponding dimension region. For that you first have to look which dimensions are defined:

```
pRegion->pDimensionDefinitions[DimensionNr];
```

And then you just put the current controller values for the defined dimensions into the following method:

```
gig::DimensionRegion* pDimRgn =
      pRegion->GetDimensionRegionByValue(Dim4Val,Dim3Val, ..., Dim0Val);
```

and you finally have the articulation data for the current situation. The first call (`GetRegion()`) you only have to invoke once; only when a key was triggered. The 2nd call though (`GetDimensionRegionByValue()`), you have to repeat whenever one of the defined dimension controller values has changed.

The dimension region contains the pointer to the sample to play. You have two possibilities to play the sample with libgig: the easiest one is to load the whole sample into RAM. You can do that with:

```
buffer_t buf = pSample->LoadSampleData();
```

The `buffer_t` structure contains the size of the allocated buffer with the sample data in bytes and a pointer to the beginning of the data. Then all you have to do is to use that pointer and copy the data to your audio output buffer. When you don't need the sample anymore, release the sample buffer to free the memory again:

```
pSample->ReleaseSampleData();
```

Drawback of the this approach of course is that it consumes a lot of memory. Instead of loading the whole sample into RAM you can also load only a short part of the beginning of the sample into RAM:

```
gig::buffer_t buf = pSample->LoadSampleData(numberOfSamplesToCache);
```

and then using the `SetPos()` and `Read()` methods of the Sample object to stream the remainder of the sample directly from disk:

```
pSample->SetPos(new_pos); // if you have to jump to another position
pSample->Read(pYourOutputBuffer, numberOfSamplesToReadOn);
```

If the sample is compressed, the Read() method automatically decompresses the sample on the fly while it's copying the data to your given output buffer (same applies to the `LoadSampleData()` method of course).

That's it, have fun!