Songlib: filters

Song Li Buser

Revision Date: November 7, 2011

Filtering notes

It is possible to modify notes using the **songlib** filtering system. It is also possible to design your own filters. The basic filtering strategy is to add filters to an instrument. Notes are then processed by the filters when they are played. Filters are applied in the opposite order they were placed upon upon the instrument's filter stack. As an example, suppose we wish to pass each note of a guitar throw a low-pass filter (to remove any high frequency harmonics), then a diminishing filter (to have the note fade away), and finally a reverb filter (to add richness). We would push filters onto the filter stack in the following order.

```
guitar = readScale(GuitarDir,GuitarPrefix);

pushFilter(guitar,myReverb);
pushFilter(guitar,myDiminish);
pushFilter(guitar,myLowPass);
```

The added filters must have a prototype similar to...

```
void filter(int *,int);
```

...and are called back during filter processing. The added filters are passed the raw audio data (as an integer array) and the length of the array. The filter should then modify the audio data (in place) in the appropriate way.

Since the built-in filters take a varying number of arguments, one must *wrap* them so to accomodate the two argument requirement of added filters. Here is an example, with, *myLowPass* defined as follows:

```
static void
myLowPass(int *data,int length)
    {
    lowPass(data,length,4000,1.414);
    }
```

The *lowPass* function is a one of the built-in filters . It takes two additional arguments beyond the raw data and the lentgh. The *myLowPass* procedure is a wrapper so that the *lowPass* function can be added to the stack of instrument filters.

Filtering occurs when notes are played. Thus if a note is never played, it is never filtered. In additions, notes are only filtered if a non-empty filter stack is in existence at the time the note is played. Finally, notes are (nominally) filtered once and cached. If the filter stack is changed and the note is replayed, the cached version will be used and the note will not be refiltered. If you wish to refilter a note, you must first clear it first via a call similar to:

```
setFilteredNote(instrument,octave,pitch,0);
```

Here are the list of functions for manipulating filters and filtered notes:

```
void pushFilter(int instrument,filter_t filter);

filter_t popFilter(int instrument);
```

The *pushFilter* function adds the given filter to the stack of filters assigned for the given instrument. The *popFilter* function removes the most recently added filter still on the filter stack. The type *filter_t* is defined as:

```
typedef void (*)(int *,int) filter_t
```

One can retrieve filtered notes for use in the *r*-type note playing functions:

```
RRA *getFilteredNote(int instrument,int octave,int pitch);

RRA *setFilteredNote(int instrument,int octave,int pitch, RRA *newest);

RRA *getFilteredNumberedNote(int instrument,int numberedNote);

RRA *setFilteredNumberedNote(int instrument,int numberedNote, RRA *newest);
```

The *getFilteredNote* and *getFilteredNumberedNote* functions retrieve the specified filtered note while the *setFilteredNote* and *setFilteredNumberedNote* update the filtered note.

The *set...* functions return the previous value of the filtered note.

One can also filter notes without playing them by using the following functions:

```
void filter(int instrument,int octave,int pitch);

void nfilter(int instrument,int numberedNote);

void hfilter(int instrument,RRA *h);

void dfilter(int instrument,int *data,int length);
```

Be aware that the *filter* and *nfilter* functions permanantly filter (for the remaining life of the songlib program) the specified clean notes. The *hfliter* function filters the given RRA object while the *dfilter* function filters the given data array. All functions use the filter stack currently associated with the given instrument. If the instrument's filter stack is empty, these functions have no effect.

See also: builtInFilters