Real-Time Implementation and Control of a Classical Guitar Synthesizer in SuperCollider

Mikael Laurson
Sibelius Academy
Center for Music and Technology
P.O.Box 86, 00251 Helsinki, Finland
laurson@siba.fi

Abstract

This paper presents a detailed account on how a model-based state-of-the-art classical guitar synthesizer has been implemented in SuperCollider2 (SC2, McCartney 1998). SC2 is an attractive choice for model-based instruments as it is a general purpose and efficient software synthesizer containing a large set of predefined unit generators. It is also a high-level programming language allowing the user to tailor various control strategies. Besides implementation issues we describe how the model is controlled and how the user can specify accurately various playing styles used in the classical guitar repertoire.

1. Background

The recent advances in modeling techniques of musical instruments have raised two questions of primary importance. First, how should the models be implemented: should one use environments that have an emphasis on physical modeling algorithms (STK, Cook and Scavone 1999); or, should one use large, general purpose synthesis environments (SC2, MSP, Csound). Second, what control strategies should one choose: are existing controllers used by human performers adequate; or, should one design new ones; or, should one have a completely different approach where the control information is generated by the computer. These questions are complex and obviously no definite answers can be found here.

Take for instance the first question. On the one hand, general-purpose synthesis environments are often block oriented and thus do not support easily sample-based calculation often required by some recent non-linear techniques used by model-based instruments (for instance Tolonen et al. 2000). On the other hand, general-purpose synthesis languages may have better support when designing complex instruments. For instance, in our case, when using a commuted string model (Karjalainen et al. 1993, Smith 1993), a huge database of samples and other control data is required in order to get satisfactory results. Thus for our study, as we use a linear string model requiring a lot of data, a general purpose real-time system such as SC2 seems to be a good choice.

The answer to the second question, concerning the control of model-based instruments, depends very much on what specific instrument one attempts to control. As our aim is to control an acoustical guitar model, we chose to use an approach where the complete control information is generated by an enriched notation environment called Expressive Notation Package (ENP, Laurson et al. 1999, Kuuskankare and Laurson 2000). ENP is a PatchWork (PW, Laurson 1996) user library and it is written in Lisp and CLOS. The use of notation in expressive control was motivated by the lack of adequate real-time guitar controllers, familiarity with common music notation and precision of control. ENP resembles commercial notation packages since it requires no textual input. Besides a full set of standard notation facilities, ENP has user-definable extensions - such as instrument specific expressions - that allow efficient description of interpretation.

The system described in this paper consists of three parts: database, control part and string models. In the following we discuss each part in turn (sections 2-4). After this, in section 5, we give examples of how various playing styles used in the classical guitar repertoire can be simulated using the tools that were described in the previous sections.

2. Database

Our model is based on digital waveguide modeling and commuted waveguide synthesis (Smith 1992, Karjalainen et al. 1998). In order to capture the individual characteristics of each string we use as a starting point a large database of pre-processed samples. This database consists of about 80 samples and allows us to use an individual sample for every possible string/fret combination. The database consists also of pre calibrated filter coefficients, pitch values and amplitude scalers. This scheme allows the system to use “neutral“ playing parameters just by using one finger/fret control event.

The automated Matlab-based system for extracting excitation samples and other control information from pre-recorded sound material used for this study is described in Välimäki and Tolonen 1998, and in Erkut et al. 2000. The data produced by the Matlab system was afterwards calibrated by ear. This fairly cumbersome calibration step was necessary mainly for three reasons. First, the analysis tools producing the raw material for the guitar model were not always perfect (although the most recent versions, described in Erkut et al. 2000, seem to produce very good results). Typically this phase included the correction of the decay and timbre characteristics of problematic string/fret combinations. Second, the original recorded sound material included some occasional unevenness and distortion. In order to remove these deficiencies, it was necessary to replace some of the problematic excitation signals with signals that were found from nearby string/fret combinations. Third, the calibration process assumes a single polarization model and thus gives out the parameters for only this simplified model. As a final step the guitar model was tuned and the amplitudes of the excitation signals were normalized.

3. Control Part

Figure 1 gives an overview how the guitar model is controlled. The starting point is a musical excerpt that is prepared in advance by the user in ENP. This score includes besides basic musical information such as pitches and rhythms also instrument specific information such as fingerings, pluck positions, left hand slurs, and so on (for details see Laurson et al. 1999). The score is then translated into MIDI data that in turn controls in real-time the guitar model situated in SC2.

Figure 1: *Control flow of the guitar model.*

The control part in SC2 maps incoming MIDI data to meaningful control information which can be either discrete (pluck events) or continuous. The latter case consists normally of scalers that allow to deviate from the neutral case. All incoming MIDI data is scaled to an appropriate range using the SC2 Plug and MIDIController modules. These modules also perform internally interpolation between incoming MIDI values. This allows to use a fairly low control-rate - typically 10 or 20 ms - for continuous control.

The scheme shown in figure 1 has the advantage that the synthesizer part in SC2 is quite simple and straightforward to implement. The most complex part of the system - i.e. the generation of the control data in ENP - is discussed below in section 5.

4. String Model Part

The guitar model consists of six dual-polarization strings. Sympathetic coupling of the strings is achieved with an inherently stable coupling matrix implementation (for details see Karjalainen et al. 1998). The string models are implemented using the powerful SC2 "multichannel expansion" feature. Operations for several strings can be expressed by a single statement.

The Appendix gives some of the most important parts of the SC2 code used by the system. Due to space limitations the code is not complete: there are no variable declarations, only the start and end parts of the large data-base array are given, the code for the loading of samples and for the feedback matrix is missing, etc. The code is split in 5 sections.

The first section (1) defines a SC2 array that defines the database for each possible string/fret combination where each sub array has the following meaning: [<freq>, <filter coef>, <filter gain>, <sample>, <amp correction>, <pitch correction>].

In section (2) we define the main function that is finally given to the SC2 Synth object when running the instrument model. We start by defining two arrays of buffers used later by the delays (one array for each polarization). Also we give the pitches for the open strings for our model.

Section (3) defines a set of dynamic controllers (using MIDIController and Plug). This part defines the interface for the incoming control information from ENP and the guitar model. Also we initialize the system to appropriate initial values.

In section (4) we define a general function, called 'vrfn', for incoming pluck events. This function returns the excitation signals which will be fed later into the delay lines of the string models. We use here the Voicer module as the pluck events are simulated with MIDI note-on messages. The function first calculates values for pluck position and amplitude. After getting current string and fret numbers, we retrieve the corresponding string/fret sub array from the database. This information is used, in turn, to update several controller values. Finally we calculate the final output for the excitation output. This fairly complex expression is used to scale and comb filter the sample output.

In the final section (5) the code calculates final controller values for delay times, detuning scalers, filter coefficients and filter gains. These are used as arguments when reading delay arrays producing the output of the string models. The output of the first delay array is given to the function 'matMult' which calculates the output of the feedback matrix (used to simulate the sympathetic coupling of the strings). After this the excitation signals are written to both delay buffers. Also, the output of the feedback matrix is written to the second buffer array (the feedback scheme given above, where the output of the feedback matrix - calculated from the first delay array - is fed only to the second delay array, is motivated by the need to keep the system stable). Finally, the output of the delays are panned and mixed resulting in the final output.

5. Simulation of Playing Styles

This section gives some ideas how various playing styles used by a classical guitarist can be simulated from ENP. We start by describing some basic techniques such as simple plucks, fast "replucks" (i.e. repeated plucks), staccato, left-hand slurs and vibrato. Also mapping of other playing styles such as forte, piano, pizzicato and harmonics will be discussed (some of these techniques are also discussed in Erkut et al. 2000).

Sending a key-on event to SC2 simulates a simple pluck event. The system selects from the database the appropriate excitation sample, the main frequency plus some other controller parameters according to the current string (= channel) and fret (= the value of the current 'frets' controller) numbers. The velocity gives the amplitude and the key number the pluck position of the current pluck event. A comb filter simulates the pluck position. The pluck position values are normalized between 0 and 1.0 (thus 0.5 refers to a position at the middle of a string). Just before the string is replucked the gain of the loop filter of the current string is lowered with an envelope for a short time. For fast replucks, i.e. when the current string to be plucked is still ringing, the system sends an extra repluck sample just before the actual pluck excitation. This is done in order to simulate the damping effect of a vibrating string when the right hand fingernail of the player touches the string. This technique is used also when playing in a staccato style where the player damps the strings with the right hand fingers.

Left hand slurring technique is implemented simply by sending a key-on event with a very low velocity value. In this case the gain of the loop filter is not modified.

Additional pitch information is sent from ENP as a scaler using two adjacent MIDI messages (this provides us with a resolution of 14 bits). The maximum depth (or max-depth) of a vibrato to be applied around the current main frequency value is calculated as follows. If the score does not contain for the current note any specific vibrato expressions (i.e. we want only to play a "straight" note), the max-depth value depends on whether the current fret is 0 (i.e. an open string) or not. If it is 0 the max-depth is equal to 0. For higher fret values the max-depth is calculated by adding more and more vibrato (the amount is though always very moderate) as the fret value gets higher. This is done in order to simulate the fact that for higher frets the string is more loose which in turn makes it more difficult for the player to keep the pitch stable. If, however, the score contains vibrato expressions, the vibrato max-depth is calculated depending on the name of the vibrato expression. Vibrato expressions are named with the string "vb" with an extension (a number from 1 to 9) indicating the max-depth of the vibrato. Thus to simulate a very slight vibrato one can use "vb1", a moderate vibrato is given with "vb5", an extreme vibrato with "vb9", and so on. The speed of the vibrato is normally kept static (typically around 5-6 Hz). The overall depth, however, is controlled by an envelope - scaled to the current max-depth value - with an ascending-descending function with two humps, in order to avoid a mechanical effect when applying a vibrato.

Forte vs. piano playing is simulated by changing the gain of the excitation sample. Also the system adjusts the cut-off frequency of a lowpass filter that filters the excitation sample (forte playing has higher cut-off values, piano lower ones). In forte playing the pitch is affected by starting with a slightly sharp pitch which is gradually lowered to the normal pitch value (an alternative way to simulate this idea is discussed in Tolonen et al. 2000).

The pizzicato effect (where the player damps the strings with the right hand) is accomplished by lowering slightly the gain and the cut-off frequency of the loop filter of the current string. Although this produces reasonable results it would probably improve the pizzicato effect if one would use special excitation signals for this purpose.

The harmonics used in the classical guitar repertoire are accomplished so that the player, while plucking a string with the right hand, damps for a short time the string with the left hand. After this the left-hand fingers are lifted rapidly allowing the string to ring freely. This effect produces a very distinct "bell" like sound. The harmonics effect is simulated in ENP by setting the pluck position value so that it matches the current string length. Thus, if we want to produce a harmonic that is one octave higher than the open string (i.e. the player damps the string at the 12th fret) the pluck position value is 0.5. Although we do not simulate the actual complex physical behavior, our approach produces fairly good results.

6. Conclusions

We have presented in this paper how a classical guitar model can be implemented in a general-purpose synthesis environment. We also discussed how various playing techniques are realized using an enriched notation package. Future plans include for instance the improvement of the automated analysis system for extracting excitation signals and control data (some recent developments are reported in Erkut et al. 2000). Also it would be interesting to change the current MIDI control system to a more flexible synthesis protocol such as Open Sound Control (OSC, Wright and Freed 1997).

Acknowledgements

This work has been supported by the Academy of Finland in project "Sounding Score - Modeling of Musical Instruments, Virtual Musical Instruments and their Control".

References

Cook, P. R., and G. P. Scavone. 1999. "*The Synthesis ToolKit (STK)*". In Proc. ICMC'99, pp. 164-166.

Erkut, C., V. Välimäki, M. Karjalainen, and M. Laurson. 2000. "*Extraction of Physical and Expressive Parameters for Model-Based Sound Synthesis of the Classical Guitar*". in AES, the 108th Convention 2000.

Karjalainen, M., V. Välimäki, and Z. Jánosy. 1993. *"Towards high-quality sound synthesis of the guitar and string instruments*". In Proc. ICMC'93, pp. 56–63.

Karjalainen, M., V. Välimäki, and T. Tolonen. 1998. "*Plucked-String Models: From the Karplus-Strong Algorithm to Digital Waveguides and Beyond*". Computer Music J., Vol. 22, No. 3, pp. 17-32.

Kuuskankare, M., and M. Laurson. 2000. "*Expressive Notation Package (ENP), a Tool for Creating Complex Musical Output*". In Proc. Les Journées d'Informatique Musicale, pp. 49-56.

Laurson, M., and J. Duthen. 1989. "*PatchWork, a Graphical Language in PreForm*". In Proc. ICMC'89, pp. 172-175.

Laurson, M. 1996. "*PATCHWORK: A Visual Programming Language and Some Musical Applications*". Doctoral dissertation, Sibelius Academy, Helsinki, Finland.

Laurson, M., J. Hiipakka, C. Erkut, M. Karjalainen, V. Välimäki, and M. Kuuskankare. 1999. "*From Expressive Notation to Model-Based Sound Synthesis: a Case Study of the Acoustic Guitar*". In Proc. ICMC'99, pp. 1-4.

McCartney, J. 1998. "*Continued Evolution of the SuperCollider Real Time Environment*". In Proc. ICMC'98, pp. 133-136.

Smith, J. O. 1992. "*Physical Modeling Using Digital Waveguides*". In *Computer Music J.,* Vol. 16, No. 4, pp. 74–91.

Smith, J. O. 1993. *"Efficient synthesis of stringed musical instruments*" In Proc. ICMC'93, pp. 64–71.

Tolonen, T., V. Välimäki, and M. Karjalainen. 2000. "*Modeling of tension modulation nonlinearity in plucked strings*". In IEEE Trans. Speech and Audio Processing, Vol. 8, No. 3, May, pp. 300-310.

Välimäki, V., and T. Tolonen. 1998. *"Development and Calibration of a Guitar Synthesizer*". J. Audio Eng. Soc., Vol. 46, No. 9, Sept., pp. 766-778.

Wright, M., and A. Freed. 1997. "*Open Sound Control: A New Protocol for Communicating with Sound Synthesizers*". In Proc. ICMC'97*,* pp. 101-104.

Appendix

/// **(1) database** ///

```
generalinfo = #[
[[ 329.628, 0.117, 0.9949, "ex100", 1.1, 1.00221 ],[ 349.228, 0.27, 0.998, "ex101", 1,  .....
,[ 164.814, 0.368, 0.9969, "ex612", 1.0, 1.00039 ]  ]  ];
stramps = 0.1*[2.0,1.5,2.0,1.3,2.5,2.2];
GF.resetLoadedSamples;
GF.loadSamplesList(samples);
nofstrs = 6;
```

/// **(2) main function** ///

```
fn = {arg synth;
buffers  = Array.fill(nofstrs, {Signal.new(Synth.sampleRate * 0.02);});
buffers2 = Array.fill(nofstrs, {Signal.new(Synth.sampleRate * 0.02);});
basefreqs = [64, 59, 55, 50, 45, 40].midicps;
basedels = 1/basefreqs;
```

/// **(3) dynamic controllers: gain, freq, fltcoefscs, frets** ///

```
gctrls = Array.fill(nofstrs, {arg ind; MIDIController.kr(ind+1,7,0,1.0,'linear',0.001);});
gains = Array.fill(nofstrs, {arg ind; var curinfo;
                                curinfo = at(at(generalinfo,ind),0); Plug.kr(at(curinfo,2),0.01);});
pitchHigh = Array.fill(nofstrs, {arg ind; MIDIController.kr(ind+1,1,0,127*128,'linear',0.01);});
pitchLow  = Array.fill(nofstrs, {arg ind; MIDIController.kr(ind+1,2,0,127,'linear',0.01);});
freqs = Array.fill(nofstrs, {arg ind; var curinfo;
                                curinfo = at(at(generalinfo,ind),0); Plug.kr(at(curinfo,0),0.01);});
fcoefs = Array.fill(nofstrs, {arg ind; var curinfo;
                               curinfo = at(at(generalinfo,ind),0); Plug.kr(at(curinfo,1),0.01);});
fltcoefAndgainscs  = Array.fill(nofstrs, {arg ind; MIDIController.kr(ind+1,3,0.0,1.0,'linear',0.01);});
frets  = Array.fill(nofstrs, {arg ind; MIDIController.kr(ind+1,4,0,127,'linear',0.0);});
```

/// **(4) pluck events** ///

```
vrfn =  {arg chan;
        Voicer.ar({ arg voicer, i, synth, deltaTime, channel, note, velocity;
            var plckpos, curstring, output, amp, curfret, curinfo, sample, pitchcorr;
            plckpos = ((note/127)*0.5); amp = ((velocity/127) ** 2);
            curstring = (channel - 1);
            curfret = asInteger(value(source (at(frets,curstring))));
            curinfo = at(at(generalinfo,curstring),curfret);
            source(at(freqs, curstring)) = at(curinfo,0);
            source(at(fcoefs, curstring)) = at(curinfo,1);
            source(at(gains, curstring)) = at(curinfo,2);
            sample = at(curinfo,3);
            pitchcorr = at(curinfo,5);
            output = GF.sampleIns([sample, 1], pitchcorr, 0, nil)*amp;
            output = Line.kr(at(stramps,curstring)*at(curinfo,4),0.001,
                 max(0.001,if(curstring<4,{0.28*(0.13+(amp*0.05))},{0.32*(0.18+(amp*0.05))}))*
                LPF.ar((output- (DelayA.ar(output, 0.05,  at(basedels,curstring)*plckpos,
                if(curstring<4,{0.35},{0.75})))),(0300+(amp*8000)));},1,chan,1); };
excits  = [vrfn.value(1), vrfn.value(2), vrfn.value(3), vrfn.value(4), vrfn.value(5), vrfn.value(6)];
```

/// **(5) string models** ///

```
matLabPitchSc = 10000;
deltimes = 1/(freqs * ((pitchHigh+pitchLow) / matLabPitchSc));
detunes2 = [1.0001,1.0001,1.0001,1.00015,1.00018,1.0002];
finalfcoefs = fcoefs*fltcoefAndgainscs;
finalgains = gains*gctrls*fltcoefAndgainscs;
fltdels = OnePole.ar(TapA.ar(buffers, deltimes),finalfcoefs,finalgains);
fltdels2 = OnePole.ar(TapA.ar(buffers2, deltimes*detunes2),finalfcoefs,finalgains);
feedb1 = LPZ1.ar(value(matMult, fltdels, feedbMatrix));
DelayWr.ar(buffers, excits+fltdels);
DelayWr.ar(buffers2, excits+fltdels2+feedb1);
Mix.ar(Pan2.ar(Mix.ar([fltdels,fltdels2]),2*[ -0.07, -0.05, -0.02, 0.0, 0.02, 0.04 ],2));
};
Synth.play({arg synth;
 synth.blockSize = 56;
  value(fn, synth)});
)
```