A New, Flexible Framework for Audio and Image Synthesis.

James McCartney
3403 Dalton St.
Austin, TX 78745
james@audiosynth.com

Abstract.

A new synthesis framework has been developed which has many interesting features. The following buzzwords briefly describe its features: real-time, controllable, dynamically editable, extensible, portable, lightweight, efficient, embeddable. Unit generators called Units are linked into Graphs which are themselves Units. Unit connections may be changed during execution. Graphs containing cycles are allowed. All Units can execute at full sample rate or at one sample per buffer rate. Graphs can contain subgraphs. Subgraphs can have differing block sizes from their parents. Physical models requiring single sample feedback can be implemented in a Graph with a single sample buffer size and this Graph can be embedded in another Graph having a larger buffer size. FFTs requiring large buffer sizes may similarly be embedded in Graphs with smaller buffer sizes. Units may be reset individually during execution. Modifiers may be applied to any unit while running to control its execution. Modifiers for Pause, Mute, Hold, and Freeze are provided. The engine can synthesize video as well as audio by simply interpreting the samples as a stream of pixel values in a single plane of an image. A number of image specific Units are planned. There is already a large library of Units which are applicable to both audio and video. New Units are added via subclassing. The library is lightweight, implemented in C++ and has a small easy to use interface. It is designed to be able to be embedded easily into other software frameworks. Great attention has been paid to efficiency and clarity. This paper can only give a brief overview and show an example of a plug in for SuperCollider.

To increase efficiency and enable simpler plug ins for version 3 of the SuperCollider synthesis programming language environment, the synthesis engine has been rewritten as a C++ framework. During this development, a number of innovations were incorporated, while retaining previous features that make SuperCollider powerful. The important features retained were: block calculation, dual rate system (control rate and audio rate), dynamic instantiation of all unit generators including long delay lines, variable block size per voice, ability to spawn subgraphs with single sample accuracy sample accurate end time stamps. Nearly all but a few rarely used or obsolete unit generators have been or are in the process of being ported to the new framework.

Unit generators are represented in the framework as instances of the Unit class. Writing a plug in involves providing a constructor and destructor, and overriding a few methods. The constructor is responsible for allocating the Input and Output ports. The InputsChanged() method is called when all the input connections are valid initially and again anytime an input is changed. This method is responsible for setting a pointer to a method for calculating the output samples for the Unit.

The Reset() method is called each time a Unit is started. At the time it is called all inputs are valid. Its responsibility is to initialize any data specific to the Unit and to calculate its first output sample.

In addition one must write one or more functions for calculating the output samples of the Unit. These functions are installed by the InputsChanged() method and are not an override of a virtual function, since there may be any number of them for a given Unit. The reason one might have more than one is to treat as efficiently as possible various combinations of input rates. There are a few means of controlling combinatorial explosion of rates, which will not be dealt with here.

One advantage of the block calculation design over a single sample compiled instrument design is that unit generators can be added, removed, repatched, or have modifiers added to thier behaviour while running. All of these new features were incorporated into the new framework.

All Units may have modifiers added that alter the behaviour of the Unit. These modifiers install their own runtime function into the Unit and save the original so that they may alter the execution of the Unit. Several modifiers are built in, including a fade output level modifier, a modifier which fades the Unit to zero and generates and end time stamp, hold current value while still running, hold current value and pause execution, fade to zero and pause execution.

In addition to variable block size per spawned graph, the new framework permits embedded graphs that have a block size which is an integer multiple or subdivision of the parent. This allows parts of a graph which may require large or single sample buffer sizes to be segregated allowing the rest of the graph to be performed more efficiently.

The synthesis engine framework is independant of the SuperCollider language and environment and can be easily separated to be linked into other programs. The framework includes special memory management routines for allocating memory in real time. Multiple copies of the engine can operate simultaneously in separate threads.

Following is complete working source code for sawtooth oscillator plug in for SuperCollider version 3. Because of inheritance, this Unit can automatically support audio rate and control rate instances, repatching, modifiers and all other features of the SuperCollider engine.

```
//////////////////////////////////////////////////////////////////////////////////////////
// Example SuperCollider unit generator plug in: A Sawtooth oscillator

#include "UnitGlue.h"
#include "SCPlugin.h"
#include "MulAddUnit.h"

#pragma export on
	// export the code fragment's loadPlugIn function to the linker
extern "C" { SCPlugIn* loadPlugIn(void); }
#pragma export off

//////////////////////////////////////////////////////////////////////////////////////////

namespace SCSynth {

class MyLFSaw : public MulAddUnit
		// MulAddUnit is for single output Units that have a mul and add input
{
public:
	MyLFSaw(World* inWorld, int inCalcRate, void* inClientData);

protected:
	virtual void InputsChanged();		// called when inputs change
	virtual void Reset();			// called when Unit is started

private:
	void next_a(int inNumSamples); // run function for audio rate freq input
	void next_k(int inNumSamples); // run function for control rate freq input

	// Samp is a typedef for the output sample type, currently float.
	Samp mPhase, mFreqMul;
	Input mIn[4];			// inputs are: frequency, initial phase, multiply, add
};
```

```
MyLFSaw::MyLFSaw(World* inWorld, int inCalcRate, void* inClientData)
	: MulAddUnit(inWorld, inCalcRate, inClientData), mPhase(0.f)
{
	MakeInputs(4, mIn);			// allocate inputs
	// tell the engine that the phase input is only read upon Reset().
	SetReadRate(1, read_Reset);
}

void MyLFSaw::InputsChanged()
{
	if (InputRate(0) == calc_FullRate) { // choose a run function based on freq input rate
		SetRunFunc((UnitFunc)&MyLFSaw::next_a);	// freq is audio rate
	} else {
		SetRunFunc((UnitFunc)&MyLFSaw::next_k);	// freq is control rate
	}
}

void MyLFSaw::Reset()
{
	MulAddUnit::Reset();			// call inherited Reset method
	mFreqMul = 2.f * SampleDur();		// calculate frequency multiplier
	mPhase = ZIN0(1);			// get the initial phase value

	next_k(1); // calculate first output sample
}

void MyLFSaw::next_a(int inNumSamples)
{
	// freq is audio rate
	Samp *out = ZOUT(0);		// get a pointer to the output buffer
	Samp *freq = ZIN(0);		// get a pointer to the freq input buffer

	Samp freqmul = mFreqMul;	// load instance variables into registers
	Samp phase = mPhase;

	// LOOP, ZXP ZOUT and ZIN are macros that allow recompilation for various architectures
	// while maintaining the most efficient pointer accesses and loop construction.
	LOOP(inNumSamples,
		ZXP(out) = phase;			// write output
		phase += ZXP(freq) * freqmul;		// increment phase
		if (phase >= 1.f) phase -= 2.f;		// wrap output between -1 and +1
		else if (phase <= -1.f) phase += 2.f;
	);
	mPhase = phase;		// restore register to instance variable
	MulAdd(inNumSamples);	// perform multiply & add inputs
}
```

```
void MyLFSaw::next_k(int inNumSamples)
{
	// freq is control rate
	Samp *out = ZOUT(0);			// get a pointer to the output buffer
	Samp freq = ZIN0(0) * mFreqMul;	// calculate freq

	Samp phase = mPhase;		// load instance variable into register

	if (freq >= 0.f) {			// choose a loop based on sign of freq
		LOOP(inNumSamples,
			ZXP(out) = phase;	// write output
			phase += freq;		// increment phase
			if (phase >= 1.f) phase -= 2.f; // wrap output between -1 and +1
		);
	} else {
		LOOP(inNumSamples,
			ZXP(out) = phase;	// write output
			phase += freq;		// increment phase
			if (phase <= -1.f) phase += 2.f; // wrap output between -1 and +1
		);
	}
	mPhase = phase;		// restore register to instance variable
	MulAdd(inNumSamples);	// perform multiply & add inputs
}

} // namespace SCSynth

/////////////////////////////////////////////////////////////////////////////////////////

// define the plug in class
class MyPlugIn : public SCPlugIn
{
public:
	MyPlugIn() {}			// constructor for plug in
	virtual ~MyPlugIn() {}	// destructor for plug in

	// AboutToCompile is called each time the SC class library is compiled.
	virtual void AboutToCompile();
};

void MyPlugIn::AboutToCompile()
{
	// template for installing the Unit in the SC class library
	defineUnitClass<MyLFSaw>();
}

/////////////////////////////////////////////////////////////////////////////////////////

// This function is called when the plug in is loaded into SC.
// It returns an instance of MyPlugIn.
SCPlugIn* loadPlugIn()
{
	return new MyPlugIn();
}
```

References

McCartney, James. 1996. "SuperCollider, a new real time synthesis language." Proceedings ICMC1996, Hong Kong, pp 257-258.

McCartney, James. 1998. "Continued Evolution of the SuperCollider Real Time Synthesis Environment." Proceedings ICMC1998, Ann Arbor Michigan, pp 133-136.