Goldsmiths Department of Computing
Internal Report #4
Jan 2010

MAXIMILIAN: A CROSS PLATFORM C++ AUDIO SYNTHESIS LIBRARY FOR ARTISTS LEARNING TO PROGRAM

Mick Grierson

Department of Computing

Goldsmiths

University of London

London SE14 6NW, UK.

ABSTRACT

Maximilian is a free, open source, MIT licensed C++ audio synthesis library built on top of RtAudio, designed to be cross platform and simple to use. The syntax and program structure have been designed to mirror the popular Java-based 'Processing' environment. Complex DSP operations have been masked as much as possible to facilitate the use of the library by artists and creatives who are learning to program for the first time. The library provides classes for standard waveforms, envelopes, sample playback, filters with resonance, and delay lines. In addition, equal power stereo, quadraphonic and 8-channel ambisonic support is included. The library can be used on its own, or in combination with other tools such as openFrameworks, or the Steinberg VST SDK.

INTRODUCTION

Maximilian is a freely available audio synthesis library written in C++. Its only dependency is Gary P. Scavone's RtAudio 4 library [4][5], providing a framework for compatibility across Windows, Linux and OS X. Maximilian has been designed to simplify the process of learning computer music, digital signal processing and synthesis in C++. Furthermore, it is intended that this library be used to introduce audio programming to artists who have less experience with textual programming languages.

Design Approach and Rationale

The structure and syntax of the library has been designed along similar lines to the popular Java based programming environment for artists, *Processing*.[6] This is for a number of reasons. Maximilian has been specifically developed for the Goldsmiths Creative Computing program – a degree program that fuses interdisciplinary arts and science approaches. Students on this and related courses (including our Music Computing joint degree program) come from a variety of backgrounds. Although often both technically and aesthetically gifted, they are not typical computer science students. In year one, they become very familiar with *Processing* as part of their introduction to programming. Students learn basic audio programming in Java / Processing through the use of Ollie Bown's excellent Beads library[1] (Beads plugs a significant gap, as owing to systemic problems involving the JVM garbage collector and audio buffering, Java, and therefore Processing suffers from a distinct lack of genuinely usable audio classes). This teaching method has proved very successful, impacting significantly on student learning. Maximilian continues this approach, allowing students to progress from Java to C++, whilst maintaining a familiar syntax and structure.

RELATED WORK

The popular C++ creative coding toolkit, openFrameworks (OF), also uses RtAudio 4. As Maximilian has been released using an open source, MIT license, it might be considered favourable to other existing solutions for open source creative audio application development in C++, including FMOD (currently used in OF) and the Synthesis Tool Kit (STK) [2][3]. Also, as Maximilian is written in C++, it can be easily used to develop software using a variety of different audio programming toolkits, including the iPhone and Steinberg VST SDK's. This makes it more widely portable than other text-based computer music platforms, and although it is not intended as a competitor to Super Collider, Chuck[7], PD, STK or MaxMSP, it will be very useful in learning/teaching creative and third party software development to both undergraduates and postgraduates studying arts, computation and creative coding. In addition, there are no significant restrictions with respect to the commercial use of Maximilian, allowing students to generate income from their own software projects.

THE MAXIMILIAN API

Maximilian consists of two files: maximilian.h and maximilian.cpp. In order to compile software using Maximilian, RtAudio.h and RtError.h must be included. Both the RtAudio main function and callback function reside in maximilian.cpp, alongside Maximilian's function definitions. Users are encouraged to write their code into a separate main.cpp file. This file should contain three

things. First, it should specify the sampler rate, buffer size and number of channels in the following manner.

```
extern int channels=2;
extern int buffersize=256;
extern int samplerate=44100;
```

At present, Maximilian uses the default sound device. If the user wishes to utilise an external sound device, this can be specified by editing the main function in maximilian.cpp, or alternatively making the external device the default. The number of channels selected can not exceed the maximum number supported by the default device. In addition, the sample rate must be supported by the default device. Buffer size should be a power of 2, and will vary from machine to machine.

Secondly, main.cpp should contain the function

```
void setup() {
(user initialisations)
}
```

These initialisations are user-specific and will most often relate to the initial conditions of the user's declarations. At present, users are encouraged to declare objects, arrays, data etc. at the beginning of the file.

Finally, main.cpp must have the below function:

```
void play(double *output) {
(user code)
}
```

This function is used by the callback function in maximilian.cpp to populate a buffer of size n(BufferSize) with double data. The callback function then inserts this data into the buffer in an interleaved format relative to the number of channels declared above. This is not very optimised at the moment but seems to work surprisingly well.

After completing these steps, all a programmer need do in order to create sound is instantiate some objects, utilise their member functions to create data, perform elementary mathematical operations on the data, then write the data to the output.

CLASSES

At present, maximilian.h contains 6 main classes: osc, envelope, delayline, filter, sampler and mix (for panning). It is anticipated that this collection will be expanded over the coming months.

The available classes contain the following member functions. The osc class is a generic oscillator object, capable of producing phasors, sawtooth waves, sine waves, cosine waves, square waves, a sine oscillator bank of up to 100 sines, and noise. These functions are overloaded – in particular the phasor function. The phasor is a central general purpose object, and can produce a continuous signal between any two values (specified as a pair of doubles), providing a means to control any number of object types or functions, In addition, the phasor function of the osc object can be used to index arrays of musical information, or buffer data, which in turn can be used to control new objects. The phasor function is defined below.

```
double osc::phasor(double frequency,
double startphase, double endphase) {
  output=phase;
  if (phase<startphase) {
       phase=startphase;
  }
  if ( phase >= endphase ) phase =
startphase;
  phase += ((endphase-
startphase)/(samplerate/frequency));
  return(output);
}
```

The envelope class at present has two main functions, `line` and `trigger`. The line object takes an array as input and produces a sample accurate ramp with up to 1000 segments. The input array must have two values for each segment – target value, and (double) duration in milliseconds. In this way, the line function is similar to the line~ object in PD and MaxMSP. However, the `trigger` function allows the envelope to be triggered from any point along the ramp. In addition, `envelope::line()` is powerful and fast enough to be used for waveshaping and sample playback.

```
double envelope::line(int
numberofsegments,double
segments[1000]) {
  period=1./(segments[valindex+1]*0.
004);
  nextval=segments[valindex+2];
  currentval=segments[valindex];
  if (currentval-amplitude >
0.0000001 && valindex <
numberofsegments) {
       amplitude += ((currentval-
startval)/(samplerate/period));
  } else if (currentval-amplitude <
-0.0000001 && valindex <
numberofsegments) {
       amplitude -= (((currentval-
startval)*(-1))/(samplerate/period));
  } else if (valindex
>numberofsegments-1) {
```

```
        valindex=numberofsegments-2;
    } else {
        valindex=valindex+2;
        startval=currentval;
    }
    output=amplitude;
    return(output);
}
```

The delay line is a simple buffer object with a variable size buffer. The buffer size can be altered at signal rate, making this object very flexible for the creation of a number of effects, including delay, flange and chorus etc. In addition, the object can be used as a string for elementary physical modelling. When combined with a filter (detailed below), the delay line is even more powerful, and can be used to create elementary reverbs. The main delay line function is called dl:

```
double delayline::dl(double input,
int size, double feedback)  {
   if ( phase >=size ) phase = 0;
   memory[phase]=(memory[phase]*feedb
ack)+(input*feedback)*0.5;
   phase+=1;
   output=memory[phase+1];
   return(output);
}
```

The filter object contains a number of useful filters, including low pass, high pass, low pass with resonance, high pass with resonance, and bandpass. These greatly expand the power and flexibility of the library. The elementary lowpass filters are as elementary and as fast as possible. Below is the design for the low pass and high pass functions, followed by the resonant low pass filter function. Importantly, the standard lowpass and highpass filters take a cutoff frequency input scaled between 0. and 1., whereas the resonant filter functions (such as `filter::lores()`) take a frequency value in Hz, and a resonance value between 1 and 100 respectively. In part, credit is due to Olli Niemitalo for the resonant filter design.

```
double filter::lopass(double input,
double cutoff) {
   output=outputs[0] + cutoff*(input-
outputs[0]);
   outputs[0]=output;
   return(output);
}

double filter::hipass(double input,
double cutoff) {
   output=input-(outputs[0] +
cutoff*(input-outputs[0]));
   outputs[0]=output;
   return(output);
}


double filter::lores(double
input,double cutoff1, double
resonance) {
   cutoff=cutoff1*0.5;
   if (cutoff>(samplerate*0.25))
cutoff=(samplerate*0.25);
   if (resonance<1.) resonance = 1.;
   z=cos(TWOPI*cutoff/samplerate);
   c=2-2*z;
   double r=(sqrt(2)*sqrt(-pow((z-
1),3))+resonance*(z-
1))/(resonance*(z-1));
   x=x+(input-y)*c;
   y=y+x;
   x=x*r;
   output=y;
   return(output);
}
```

The mix class provides functions for equal power panning in stereo, quad or eight channel ambisonic scenarios. In each case, it is important that the sound hardware attached to the machine is capable of utilising the specified number of outputs. These functions are declared (double *) and take three arguments: the input signal, the output array (with either 2,4 or 8 elements), and the desired position in either one, two or three dimensions. The quad function is displayed below. The eight channel ambisonic version of this function is used in the newly created Goldsmiths Digital Studios Audiovisual Cognition and Interaction Lab, (designed by this researcher).

```
double *mix::quad(double input,double
four[4],double x,double y) {
   if (x>1) x=1;
   if (x<0) x=0;
   if (y>1) y=1;
   if (y<0) y=0;
   four[0]=input*sqrt((1-x)*y);
   four[1]=input*sqrt((1-x)*(1-y));
   four[3]=input*sqrt(x*y);
   four[4]=input*sqrt(x*(1-y));
   return(four);
}
```

Importantly, the mix class is not as simple to use as the other classes. The function returns an array which then must be written back to the appropriate output thus:

```
   output[0]=four[0];
   output[1]=four[1];
```

```
    output[2]=four[2];
    output[3]=four[3];
```

At present, it appears that this is the most complex piece of code a user will have to write. Users need to take care to ensure that `extern int channels` is initialised with the appropriate number of channels.

PROGRAMMING EXAMPLE

```
#include "maximilian.h"/*here we go*/

double track1[2],track2[2];/*These are
stereo busses*/
double
filtered,tune,delayed,mixed,ramp,noise,
pan;/*don't really need this*/
int morebeats,lastmorebeats;/*some
counters*/
double
melody[14]={600,0,0,650,0,0,400,0,0,425
,0,300,0,340};/*some data*/
osc a,b,c,d,e;/*some oscillator
objects*/
delayline delay;/*a delay line*/
filter myfilter;/*a filter*/
mix track1pan,track2pan;/*two pan
objects*/

extern int channels=2;/*your compiler
may complain*/
extern int buffersize=256;
extern int samplerate=44100;

void setup() {
 /*nothing required here this time.
Useful for initing envelopes*/
}

void play(double *output) {
    morebeats=((int)
a.phasor(0.25,0,16));/*here's a counter
based on a phasor*/
    tune=b.saw(melody[morebeats]*0.25)
;
    ramp=c.phasor(0.25,1,2048);
    pan=d.phasor(0.125);
    delayed=delay.dl(tune, ramp,
0.7)*1.;
    mixed=((delayed*0.3)*0.5);
    noise=e.noise()*0.25;
    filtered=myfilter.lores(noise,ramp
+1*5,1);
    track1pan.stereo(filtered,
moreoutputs, pan);
    track2pan.stereo(mixed, outputs,
1-pan);
    output[0]=track1[0]+track2[0];
    output[1]=track1[1]+track2[1];
}
```

FUTURE WORK

Maximilian will be used to teach year two and three undergraduate DSP and creative software development as part of the Goldsmiths Creative Computing Degree program. In addition, it will be developed further through the addition of a number of other important classes. These will hopefully include a granular sampler class, FFT classes with a phase vocoder function, onset detection, feature extraction, network timing for synchronising over udp, and pattern libraries for composition. It is hoped that Maximilian will prove useful to the openFrameworks community. It is also anticipated that Maximilian can be effectively optimised in the not too distance future so that it may more readily be deployed in commercial scenarious, including paid performances. Beyond this, it is hoped that Maximilian will be embraced by the open source, FLOSS and DIY audio and creative arts community so that it can have a more fulfilling life of its own.

REFERENCES

[1] O. Bown, “Ecosystem Models for Real-Time Generative Music: A Methodology and Framework”, in *Proceedings of the International Computer Music Conference*, Montreal, Canada, 2009.

[2] Cook, P. and G. Scavone (1999). The Synthesis ToolKit (STK). In *Proceedings of the International Computer Music Conference*, Beijing.

[3] Cook, P. R. (2002). Real Sound Synthesis for Interactive Applications. A K Peters, Ltd.

[4] Scavone, G. and P. R. Cook (2005). RTMidi, RTAudio, and a Synthesis Toolkit (STK) UPdate. In Proceedings of the In- ternational Computer Music Conference, Barcelona, Spain.

[5] Scavone, G. P. (2002). RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output. In Proceedings of the In- ternational Computer Music Conference, Go¨borg, Sweden, pp. 196–199. ICMA.

[6] Shiffman, D (2009). Learning Processing: A beginners guide to programming Images, Animation and Interaction, Morgan Kaufmann Series in Computer Graphics.

[7] Wang, G. and P. R. Cook (2003). ChucK: A Concurrent, On- the-fly Audio Programming Language. In Proceedings of the International Computer Music Conference (ICMC),, Singapure.