PRESENTING ‘COSMOSF’ AS A CASE STUDY OF AUDIO APPLICATION DESIGN IN OPENFRAMEWORKS

Sinan Bökesoy
Doğan Bey Sok. 34097
Büyükada-Istanbul

ABSTRACT

Since the introduction of the open source toolkits for multimedia interaction programming, artists were encouraged to start to develop their tools in C++. The C++ language has considerable advantages in performance gain; and frameworks such as Openframeworks¹ and Cinder² are C++ programming toolkits allowing us to encounter an interesting combination of artistic approach and programmer’s perspective/design. Cosmosf[1] is a stochastic sound synthesis engine, which does integrate a bottom-up sonic organization into a top-down organizing event generation system with a complex modulation routing and recursive audio structure. Cosmosf is made within Openframeworks. This paper introduces how its structural components are integrated within OF. The purpose is to share this experience with the computer music community as a case study.

1. INTRODUCTION

1.1. Cosmosf: advanced stochastic synthesizer

Cosmosf¹ is a real-time dynamic stochastic synthesis engine, which does generate sonic textures, where discrete sonic events of certain density are distributed in a time space with their onset time and duration parameter calculated with stochastic/deterministic functions. Each macro event defines the duration of a meso-space, and the sub micro events are distributed inside it. The overall goal is to achieve control on each event space and perform the process of change on the appropriate operation level. The user can intervene with the system in real-time on different time scales by inputting a sound source or accessing different type of synthesis/modulation generators by controlling the parameters for the sonic event distribution. (Figure 2)

Figure 2. Event generation (top-down) and audio routing (bottom-up) in Cosmosf

It has a recursive structure with an audio feedback loop, offering emergent sonic behavior within a hierarchy of multiple time scales. The output of the system is fed back again to the input, as the micro-event audio data.

1.2. Audio Coding in Openframeworks

One can find the existing C++ frameworks being relatively poor when considering the audio programming features offered in their bundles compared to MaxMSP⁴. Cosound⁵ etc. However with some existing C++ Synth Tool kits or various open source DSP code snippets, 3rd party expansions are always possible. For instance Cosmosf DSP code is based on the functions provided by Maximilian C++ Synth Tool Kit [2], an MIT licenced library. The of2Maxim add-on for OF is simply wrapping the Maximillian C++ Toolkit.

Openframeworks[3] gives the opportunity of easily accessing the system audio by defining an object of fSoundStream class and by setting the basic parameters bufferSize, nChannels and sampleRate to initiate it:

soundStream.set difíc(2, 0, sampleRate, initialBufferSize, 4);

On Figure 3, the communication between the code segments organized as C++ classes.

The update() method updates all the program variables in interaction with the user interface and calls the necessary update routines such as the event distribution mechanism again in accordance to the parameter input of the user. The LFO’s and stochastic LineGen’s are generated inside the audioRequest[] method and are operating with micro and meso event audio processing classes CosmosCell and CosmoCellM. The stochastic function generator ClassStochF is called by the event generation mechanism and by the modulation sources.

3. C++ TRANSLATION OF THE STRUCTURE

3.1. Event distribution process

void testApp::audioRequested(int*output, int bufferSize, int nChannels) { for(int i = 0; i < bufferSize; i++)

The nChannels variable defines the number of audio channels in the pipe. The duty of your code inside the for loop is to generate the amount of sample audio as set in bufferSize parameter (usually set to a multiple of 256). For instance, if your setup is for stereo audio, your audio information is assigned to the *output pointer via:

output[0*nChannels] = (for left channel audio sample);
output[1*nChannels + 1] = (for right channel audio sample);

Therefore the for loop fills an output array with its size equals to 2*bufferSize for the stereo audio format.

2. DEFINITION OF THE COMPONENTS

Cosmosf has following program routines organized in classes of C++ language to maintain different processes inside the application. For instance;

CellLDcpp / CellLD.h : for generating the micro event data with stochastic functions.
CellLDMeso.cpp / CellLDMeso.h : for generating the meso event data with stochastic functions.
ClassStochF.cpp / ClassStochF.h : for generating the stochastic function calculations.
CosmosCell.cpp / CosmosCell.h : for generating the micro event audio data.
CosmoCellM.cpp / CosmosCellM.h : for generating the meso event audio data.
LFO.cpp / LFO.h : Low frequency modulation sources program code.
LinearGen.cpp / LinearGen.h : Stochastic linear modulation generators in Xenakis way³.

Figure 3. The communication between the code segments organized as C++ classes.

On Figure 4, the Cosmosf cycle is a circular field, where all the events are visualized and positioned with their onset and duration time references shown on concentric circles. The full circle signifies the cycle length like the clock of the day.

The micro and meso space parameters are assigned with the user interface elements and then the stochastic functions calculate the onset and duration of each event in these spaces. For instance the CellLD class, which generates the micro events inside a meso event has the following significant methods for its operation:

void CellLD::setup(int Length, int density, int OnsetOffset, int MicroDensity) {}
PRESENTING ‘COSMOSF’ AS A CASE STUDY OF AUDIO APPLICATION DESIGN IN OPENFRAMEWORKS

Sinan Bökesoy
Doğan Bey Sok. 34971
Büyükada-Istanbul

ABSTRACT

Since the introduction of the open source toolkits for multimedia interaction programming, artists were encouraged to start to develop their tools in C++. The C++ language has considerable advantages in performance gain; and frameworks such as Openframeworks¹ and Cinder² are C++ programming toolkits allowing us to encounter an interesting combination of artistic approach and programmer’s perspective/design. Cosmosƒ¹ is a stochastic sound synthesis engine, which does integrate a bottom-up sonic organization into a top-down organizing event generation system with a complex modulation routing and recursive audio structure. Cosmosƒ is made within Openframeworks. This paper introduces how its structural components are integrated within OF. The purpose is to share this experience with the computer music community as a case study.

1. INTRODUCTION

1.1. Cosmosƒ: advanced stochastic synthesizer

Cosmosƒ¹ is a real-time dynamic stochastic synthesis engine, which does generate sonic textures, where discrete sonic events of certain density are distributed in a time space with their onset time and duration parameter calculated with stochastic/deterministic functions. Each macro event defines the duration of a meso-space, and the sub micro events are distributed inside it. The overall goal is to achieve control on each event space and perform the process of change on the appropriate operation level. The user can intervene with the system in real-time on different time scales by inputting a sound source or accessing different type of synthesis/modulation generators by controlling the parameters for the sonic event distribution. (Figure 2)

2. DEFINITION OF THE COMPONENTS

Cosmosƒ has following program routines organized in classes of C++ language to maintain different processes inside the application. For instance;

CellLD.cpp / CellLD.h : for generating the micro event data with stochastic functions.

CellLDMeso.cpp / CellLDMeso.h : for generating the meso event data with stochastic functions.

ClassStochF.cpp / ClassStochF.h : for generating the stochastic function calculations.

CosmosCell.cpp / CosmosCell.h : for generating the micro event audio data.

CosmosCellM.cpp / CosmosCellM.h : for generating the meso event audio data.

LFO.cpp / LFO.h : Low frequency modulation sources program code.

LinearGen.cpp / LinearGen.h : Stochastic linear modulation generators in Xenakian way⁴.

1 Iannis Xenakis has gathered his pioneering ideas of stochastic music in “Formalized Music: Thought and Mathematics in Composition” 1971.

On Figure 3, the communication between the code segments organized as C++ classes.

The update() method updates all the program variables in interaction with the user interface and calls the necessary update routines such as the event distribution mechanism again in accordance to the parameter input of the user. The LFO’s and stochastic LineGen’s are generated inside the audioRequest() method and are operating with micro and meso event audio processing classes CosmosCell and CosmosCellM. The stochastic function generator ClassStochF is called by the event generation mechanism and by the modulation sources.

3. C++ TRANSLATION OF THE STRUCTURE

3.1. Event distribution process

The user interface elements and then the stochastic functions calculate the onset and duration of each event in these spaces. For instance the CellLD class, which generates the micro events inside a meso event has the following significant methods for its operation;

void CellLD::setup(int Length, int density, int OnsetOffset, int MicroDensMod) {}
The phasor points to the absolute time since the beginning of each cycle. The phasor frequency is indeed the inverse of 1 Cosmos cycle duration. If the current cycle length is defined with the floating variable Loop then the Mpos variable in the below expression gives us the current phasor point indicating the absolute time in the cycle.

\[
\text{Mpos} = \text{mainCounter} \times \text{Loop} + 1, \text{Loop} + 1;
\]

When the mainCounter reaches the end of the cycle, new meso events and micro events are distributed for the next Cosmos cycle.

if (Mpos >= Loop + 1) / (if it is in the end of the loop, regenerate the cell distribution, CellMesoGenTri[0] // generate the meso-cell distribution CellMicroGenTri[0] // generate the micro-cell distribution)

3.4. Audio rendering in Cosmos main program

After realizing the event distribution for the new cycle, the program transfers this data to the micro and meso event audio classes. This happens in cascade for loop’s. First the AudioMicros class instances, which generate the audio for the micro-events, will be updated. The parameters are:

- Feedback audio data (the address pointer to the feed variable). It is calculated as;

\[
\text{feed} = (\text{outputs[0]}+\text{outputs[1]}) \times \text{Fdbamount} + \text{sample};
\]

This expression tells us, that the stereo output of Cosmos is being summed and multiplied with the feedback amount value set by the user and then added with the livelnput sample data maintained at dataBN.

- The phasor value pointing to the absolute time index.
- Various micro-cell parameters as floating point and integer arrays (only the addresses are passed).
- Synthesis parameters and LFO (6 LFOs for a micro event) parameter arrays.
- Pointers to the Cosmos output buffers bufData and bufDataB (two buffers which records the output of Cosmos in turn).

\[
\text{for} \ (\text{int} \ i = 0; \ i < \text{numCell.CellDensity}; \ i++) \ { \ \\
\text{sumLo} = \text{sumLo} + \text{AudioMicros[i][j].update(&feed[2], &point[0], &CellParam[0], &CellParam[0], &Grnparam[0], &Filtparam[0], &LFOValues[0], &bufferData[0], &bufData[0]);}
\}
\]

\[
\text{MesoSum} = \text{AudioMesoS[i][j].update(&sumLo, &point[0], &CellParam[0], &CellParam[0], &Grnparam[0], &Filtparam[0], &LFOValues[0], &bufferData[0], &bufData[0]);}
\]

3.3. Scheduling of the events

Cosmos uses the phasor object, a member of the ofxMaxim add-on and is defined in the ofxMaxOsc class.

The audio data generated at the various sections of Cosmos are assigned to pointer variables, which are allocated during each initialization phase of the classes (Figure 5). Therefore not the memory content but the memory address to the start of the relevant data will be passed in between sections. This memory chunk will be freed when the class object will be destroyed after its use. Otherwise memory leakage will happen, and in brief the application might crash.

The next significant code in the CosmosCellM update method is the if-then conditional loop, which compares the current phasor value with the onset time of the micro event inside the cycle. When they match, then the micro event DSP routine starts and generates the event audio until the end of the relevant micro event.

if (ulong(point) >= posLo & ulong(point) < posLo+posM + 1) {
\text{DSP code;}
\}

The posLo variable here is the event start time and the posM is the event duration with the variable type ulong.

There are various DSP routines and waveform generating functions inside the DSP code. For example to playback a loaded sample from the buffer with certain speed and start offset, we use the ofMapMaxSample object from the ofxMaxim add-on with quadratic interpolation.

The code expression below is for the calculation of the Sample Start point considering the values from modulation sources like LineGen and LFO. Likewise if Cosmos is not in recursive buffer playing mode; the beats play method, being a member of the ofxMaxSample class, plays the sample with the Speedindex value from the sample buffer dataBuf:

\[
\text{convX} = \text{sampleRate}/\text{sizeBN}(1-\text{sizeBN}\text{sampleRate});
\]

\[
\text{d} = \text{beatsX} / \text{Speedindex} \times \text{sampleRate}/\text{sizeBN}(1-\text{sizeBN} \text{sampleRate});
\]

Then the sumLo variable will be initialized with value 0 before the outer for loop closes. Now the generated audio can be assigned to the output variable, which carries the data to the system audio driver.

Cosmos fills 2 audio buffers, bufData and bufDataB with this output and they can be used in order to be reassigned as the input for the micro events. (Figure 5)

3.5. Generating audio events

Now we will have an inside look to the AudioMicros and the AudioMesoS Class. These objects not just generate the audio but also the modulation sources for the audio signals in audio rate inside this audio routine.

\[
\text{double} \text{CosmosCellM: updated(double } \text{Audio, double* poi, float* CellParam[0], float* GrnParam, float* Filtparam[0], double* LFOValues, double* bufData, double* bufDataB)}
\]

- Audio is a pointer to the audio assigned for the current micro event input instance coming from the direct feedback connection of the Cosmos output.
- Pointer to the phasor value giving the current absolute time inside the Cosmos cycle.
- Pointers to the parameter arrays imported from the user interface via the main program.
- DSP section parameters.
- Pointer to the LFO values calculated in the main program.
- Pointers to the recursive buffer content.

Despite the difficulty of expressing/explaining such complex code structures, which is generally the case with C++, we had a simple overlook to the significant design features implemented in Cosmos as a case study of audio application development in Openframeworks. More references of such these will encourage the composers to develop their own tools in such platforms.

4. CONCLUSION

5. REFERENCES

The phasor points to the absolute time since the beginning of each cycle. The phasor frequency is indeed the inverse of 1 Cosmosf cycle duration. If the current cycle length is defined with the floating variable Loop then the Maxpoint variable in the below expression gives us the current phasor point indicating the absolute time in the cycle.

\[
\text{Maxpoint} = \text{mainCounter}.phasor(1000\text{Ls}/\text{Loop})\lfloor 1/\text{Loop}\rfloor;
\]

When the Maxpoint reaches the end of the cycle, new meso events and micro events are distributed for the next Cosmosf cycle.

\[
\text{if} (\text{Maxpoint} \geqslant \text{Loop}) \{ \text{// if it is end of the loop, }
\]

regenerate the cell distribution. \( \text{CellMesoGenTri[]}() \) generate the meso event distribution \( \text{CellMicroGenTri[]}() \) generate the microcell distribution

3.4. Audio rendering in Cosmosf main program

After realizing the event distribution for the new cycle, the program transfers this data to the micro and meso event audio classes. This happens in cascade for loop’s. First the Audiomicroclass instances, which generate the audio for the micro-events, will be updated. The parameters are:

- the feedback audio (data the pointer to the feed variable). It is calculated as;

\[
\text{feed} = (\text{output}[0] + \text{output}[1]) * \text{numBumount} + \text{sample};
\]

This expression tells us, that the stereo output of Cosmosf is being summed and multiplied with the feedback amount value set by the user and then added with the input sample data maintained at dataBN.

- The phasor pointing to the absolute time index.
- Various micro-cell parameters as floating point and integer arrays (only the addresses are passed).
- Synthesis parameters and LFO (6 LFOs for a micro event) parameter arrays.
- Pointers to the Cosmosf output buffers dataBufl and dataBufl with this output and they can be used in order to be reassigned as the input for the micro events. (Figure 5)

3.5. Generating audio events

Now we will have an inside look to the AudiomicroS class instance returns the generated audio data to the variable sumLo. When there are overlapping micro events inside a meso event, they will be summed together as indicated within the expression;

\[
\text{sumLo} = \text{sumLo} + \text{AudiomicroS}[];
\]

Each AudiomicroS class instance returns the generated audio data to the variable sumLo. When there are overlapping micro events inside a meso event, they will be summed together as indicated within the expression;

\[
\text{sumLo} = \text{sumLo} + \text{AudiomicroS}[];
\]

When the inner loop closes, each micro space audio data will be passed inside the update method of the Audiomeso class instance. The meso events generate panning on their audio lines; hence the output of each Audiomeso class instance becomes a double sized array called MesoSum[]. When there are overlapping meso events, they will be summed to the array sum[] as indicated within these lines;

\[
\text{sum}[0] = \text{sum}[0] + \text{MesoSum}[0];
\]

\[
\text{sum}[1] = \text{sum}[1] + \text{MesoSum}[1];
\]

Then the sumLo variable will be initialized with value 0 before the outer loop closes. Now the generated audio can be assigned to the *output variable, which carries the data to the system audio driver.

Cosmosf fills 2 audio buffers, dataBufl and dataBufl with this output and they can be used in order to be reassigned as the input for the micro events. (Figure 5)

\[
\text{double} \text{CosmosfM:upadate} = \text{double} * \text{audio, double* poi, float* \text{CellParam}, int* \text{CellParam}, float* \text{Gumpar, float* Filtrparam, double* LFOvalues, double* bufData, double* bufDataB});
\]

- *Audio is a pointer to the audio assigned for the current micro event input instance coming from the direct feedback connection of the Cosmosf output.
- Pointers to the phasor value giving the current absolute time inside the Cosmosf cycle.
- Pointers to the parameter arrays imported from the user interface via the main program.
- DSP section parameters.
- Pointer to the LFO values calculated in the main program.
- Pointers to the recursive buffer content.

The next significant code in the CosmosfM update method is the if-then conditional loop, which compares the current phasor point with the start offset of the micro event inside the cycle. When they match, then the micro event DSP routine starts and generates the event audio until the end of the relevant micro event.

if (ulong(point) >= posA && ulong(point) < posL + posS + 1) {
    \text{DSP code} \};

The posA variable here is the event start time and the posL is the event duration with the variable type ulong.

There are various DSP routines and waveform generating functions inside the DSP code. For example to playback a loaded sample from the buffer with certain speed and start offset, we use the ofSampleMaxSample object from the ofSampleMax addon with quadratic interpolation.

The code expression below is for the calculation of the Sample Start point considering the values from modulation sources like LineGen and LFO. Likewise if Cosmosf is not in recursive buffer playing mode, the beats play method, being a member of the ofSampleMaxSample class, plays the sample with the Speedindex value from the sample buffer dataBufl:

\[
\text{double} \text{X} = \text{sampleRate} / \text{sizeBN} -(\text{ulong} X * \text{sizeBN}) \%
\]

The expression \text{Speedindex} = \text{sampleRate} / \text{sizeBN} \%
The sampleRateX calculated the revised playback rate value according to the demanded start offset point considering the sampleRate and the size of the sample as given with the sizeBN variable. Hence according the code, the playback rate decreases when the performed chunk size of the sample data decreases too.

4. CONCLUSION

Despite the difficulty of expressing/explaining such complex code structures, which is generally the case with C++, we had a simple overlook to the significant design features implemented in Cosmosf as a case study of audio application development in Openframeworks. More references of such these will encourage the composers to develop their own tools in such platforms.

5. REFERENCES

