Comparison of Software Sound Synthesis Tools

Lance Putnam

5/10/2007

Software Sound Synthesis Tools

In general, one can find two different types of software sound synthesis (SSS) tools: applications and libraries. Applications provide a user interface (graphical, textual, or both), programmability through scripting, and an open "plugin" architecture for synthesis algorithms. Applications generally have a quick learning curve. Applications are sometimes referred to as rapid prototyping tools meaning that they allow one to be very expressive with the tool in a short order of time, but usually at the expense of computational inefficiency. To improve efficiency, synthesis algorithms are written in lower-level languages, such as C or C++, as plugins and loaded dynamically into the program at run-time. From a user's perspective, applications are much easier to learn and use because they have a high-level interface oriented towards rapid development in a specific domain. From a developer's perspective, applications are very difficult to maintain due to dependencies on shifting operating systems and complications in building a graphical/textual user interface. Some existing SSS applications include SuperCollider, Pd, Max/MSP, and ChucK.

Libraries can be described as a library of independent modules written in a computationally efficient language that are programmatically connected to perform a specific task. To use a library effectively, one must be well versed in one of these languages. Libraries are more atomic in the sense that they make no assumptions about how a user will piece components together to solve a specific problem. In this way, they offer maximal flexibility. Also, since libraries are usually written in a more general-purpose and widely used language, they can be uplifted into many different situations, such as writing plugins and special-purpose tools. This makes them more usable from a developer's perspective. Unfortunately, the atomization of functionality in libraries demands more effort in creating higher-level structures. This adversely effects its usability factor from a user perspective. Existing SSS libraries include STK, CSL, Aura, CLAM, and SndObj.

An important consideration to make when choosing an interface for SSS is the programming language(s) used. Many SSS tools are bilingual in their approach having both a language for performing DSP and another for high-level expressive control. C and C++ are almost universally accepted as the languages of choice for performing DSP. This is due to their high degrees of efficiency and flexibility and, in turn, widespread usage and active community following. The choice of a higher-level control language, however, is a much

more complicated issue. A language rift emerges that is largely a result of personal style and multiple ways of approaching and thinking about a complex problem. The two options are to choose an existing scripting language or to write a new one. Surprisingly, the latter is the most popular choice. This may be largely due to a lack of widely accepted scripting languages that can effectively solve synthesis control problems. Some relatively newer languages, such as Python, Ruby, and Lua, are rising to top in this regard and should be kept in consideration. However, a more general problem is that programming languages tend to lose their generality as more specific tasks need to be accomplished through it. There seem to be more fundamental issues regarding control of synthesis algorithms that need to be solved first.

There are five issues that come to mind when considering the overall quality of a SSS tool: ease of use, flexibility, scalability, platform independence, and ease of maintenance. These aspects were chosen since they seem to exhibit a trade-off characteristic. Ease of use concerns the user effort required to do something expressive with the tool. Flexibility is a measure of the expressive potential of a tool, i.e. the degree to which it can solve generic problems. Scalability is a measure of how well the tool works in increasingly complex and demanding systems. Ease of maintenance has to do with how easy it is for a developer to fix problems in the source code and have those changes reflected in the tool. Platform independence rates how much platform-specific code the tool depends on and how readily it can be used on several platforms. This closely impacts the maintainability of the tool, since changes in OSs may directly impact its source code. Table 1 compares the maximum potential of each of the five attributes between SSS applications and libraries.

Table 1: Tool Attribute Potentials

Potential	Application	Library
Ease of Use	High	Low
Flexibility	Medium	High
Scalability	Medium	High
Ease of Maintenance	Low	High
Platform Independence	Low	High

Of course it would be desirable to have a tool that has high potential in

all areas. Libraries have high potential in all areas except ease of use which is very low. Conversely, applications have high ease of use, but suffer in all other areas, especially maintenance and platform independence. It is hard to say whether the added benefits of having a nice textual and/or graphical user interface is worth the sacrifices made with other aspects of the tool. For the time, it seems logical to work with a library and try to make it as easy as possible to use. One promising development in software engineering, Design Patterns, has partially risen out of a need to create simpler interfaces to software. The next section will present and compare several existing SSS libraries.

Summary of Sound Synthesis Libraries

The following libraries were chosen for analysis because they are all cross-platform, open-source and are documented. Aura is the only exception, but its design philosophy is worth looking at.

Aura is a software platform made to facilitate interactive systems that combine audio, graphics, and sensors (Dannenberg 2002, 2004). Aura was designed on the principles of generality across modes of data and scalability to run either as a single-processor or distributed system. An Aura system is divided into three separately threaded zones for graphics, control, and audio. Each zone handles inter-zone communication, memory management, and event scheduling. Objects communicate by putting messages on a FIFO queue of the zone where the object resides. Message passing is implemented using remote procedure calls and therefore is scalable to run over a network. Aura synthesis patches can be constructed and modified at run-time. Using the "instrument editor" synthesis graphs can be constructed graphically using the common boxes and wires approach. Patches can generate C++ code which can then be compiled and linked into the application at run-time.

CLAM is a combined C++ library and rapid prototyping tool for audio and music processing (Amatriain 2002, Arumi 2005). CLAM is meant to be comprehensive and includes signal processing classes, audio and MIDI I/O, XML serialization, algorithm and data visualization and interaction, and multithreading handling. There are three main layers of the infrastructure: a library (Repository), processing nodes (Processing objects), and processing

graph (Network). The Repository holds commonly encountered digital signal processing algorithms that operate on special data containers called ProcessingData objects. ProcessingData objects are divided into several types such as audio, spectrum, musical phrase, and segment. Data streams within Processing objects are divided into sample-synchronous signals (Ports) and asynchronous event signals (Controls). The Network is a data-flow model for constructing directed acyclic graphs of Processing objects operating on data streams. Several GUI components are available, using Qt and FLTK, for building visual interfaces. CLAM also includes a graphical "NetworkEditor" for constructing synthesis graphs.

The CREATE Signal Library (CSL) is a general-purpose C++ software framework for sound synthesis and digital audio signal processing (Pope 2003, 2006). CSL is designed from the ground up to be used in distributed systems, with several CSL programs running as servers on a local-area network, streaming control commands and sample buffers between them. CSL is also designed for "orchestra-scale" synthesis and thus has scalability in mind. Unit generators can be connected to each others inputs in the constructor to easily build synthesis graphs. Data is passed around through Buffer objects which represent single- or multi-channel audio signals. Sound localization is strongly supported with classes for simple panning, VBAP, and Ambisonics. Wavefield synthesis is not supported, but is planned to be in the near future.

SndObj is a C++ library that comprises a series of signal processing and control classes which can be used in a number of signal processing applications (Lazzarini, 2001). SndObj is aimed to be comprehensive in terms of functionality for working with audio and to be modular like analogue synthesizers. Objects are all derived from the same base class which knows about the sample-rate and vector size and has buffers for input and output. A utility class Table provides common tabulated functions, such as harmonics series, windows, and conversions, used in sound synthesis applications.

The Synthesis ToolKit (STK) is a C++ environment for the rapid prototyping of realtime synthesis and audio processing algorithms (Cook 1999). STK was designed with ease of use, extensibility, and pedagogy in mind. STK is unique from other similar libraries in that it is based on scalar, as opposed to vector, processing. This allows for arbitrary single-sample feedback

paths in processing graphs. Unit generators can also process sample vectors.

Comparison of Sound Synthesis Libraries

Table 2 lists what are deemed the fundamental building blocks for sound analysis and synthesis and their availablity in each library. Unfortunately there is no documentation available for Aura, so it has been omitted.

Table 2: Fundamental Synthesis Components

UGen	CLAM	CSL	SndObj	STK
Env (decay)	x	x	x	x
Delay (1-pole)			x	x
Delay (biquad)		x	x	x
Delay (comb)			x	x
Delay (vari)		x	x	x
Noise (white)		x	x	x
Noise (pink)		x		
Osc (sine)	x	x	x	x
Osc (ramp)		x	x	
Sampler		x	x	x
Spectral (DFT)	x	x	x	
Spectral (PVoc)	x		x	

Table 3 compares some of the features of SSS libraries.

From this comparison, it is evident that SndObj has the best support for fundamental synthesis techniques. CSL and STK have an acceptable score and CLAM appears to be quite lacking missing several important generators such as delays and noise. Basic unit generators are crucial to extending the library to include higher-level synthesis techniques. The capability of working in the time-frequency domain is also quite important. CLAM shines in this area seemingly favoring spectral over time-domain processing. SndObj has basic DFT and phase vocoder functionality making it acceptable. In terms of sound localization, CSL is the clear winner with Ambisonics, VBAP, and planned Wavefield synthesis. The other libraries only have simple panning

Table 3: Library Features

Feature	Aura	CLAM	CSL	SndObj	STK
Fundamentals	-	4/12	9/12	11/12	8/12
Spectral	-	Many	DFT	DFT, PVoc	-
Localization	-	Pan	Many	Pan	-
Proc. Rank	-	Vector	Vector	Vector	Scalar, Vector
Graphs?	Yes	Yes	Yes	Yes	No
Distributed	Yes	No	Yes	No	No

techniques. STK has the most sophisticated sample processing design, featuring both scalar and vector ranks. The other libraries support only vector rank processing, but this is sufficient for most general purposes. In terms of building synthesis graphs, CLAM has the most flexible design. CSL and SndObj support simple "pull" models which are suitable for many situations, but may not necessarily scale well. As far as distributed processing, Aura has the best design. This was one of its guiding design principles and as a result the entire system can be distributed transparently. CSL is the only other library that supports distributed processing. However, its model only includes streaming sample frames and not a generalized RPC mechanism as in Aura. There are other important aspects to consider in choosing a SSS library such as data containers, analysis functions, and data mapping, memory, scalar, and vector operations. These were not considered at the time of writing.

Bibliography

[1] Amatriain, X., Arumi, P., Ramirez, M. 2002. "CLAM, yet another library for audio and music processing?" Proceedings of OOPSLA 2002, Seattle, Washington, USA.

[2] Arumi, P. and Amatriain, X. 2005. "CLAM: an object oriented framework for audio and music." Proceedings of 3rd International Linux Audio Conference; Karlsruhe, Germany.

[3] Cook, P. R., Scavone, G. 1999. "The Synthesis ToolKit (STK)." Proceedings of 1999 International Computer Music Conference.

[4] Dannenberg, R.B. 2002. "Aura as a platform for distributed sensing and control." Symposium on Sensing and Input for Media-Centric Systems (SIMS 02), Santa Barbara: University of California Santa Barbara Center for Research in Electronic Art Technology, pp. 49-57.

[5] Dannenberg, R. B. 2004. "Aura II: Making real-time systems safe for music." Proceedings of the 2004 Conference on New Interfaces for Musical Expression (NIME04), Hamamatsu, Japan.

[6] Lazzarini, V. 2001. "Sound processing with the SndObj library: An overview." Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01), Limerick, Ireland.

[7] Pope, S. T., Ramakrishnan, C. 2003. "The CREATE Signal Library (Sizzle): Design, issues, and applications." Proceedings of 2003 International Computer Music Conference.

[8] Pope, S. T., Amatriain, X. Putnam, L., Castellanos, J., Avery, R. 2006. "Metamodels and design patterns in CSL4." Proceedings of 2006 International Computer Music Conference, New Orleans, LA.