CBPSC: CORPUS-BASED PROCESSING FOR SUPERCOLLIDER

Thomas Stoll
Buffalo, NY, USA
tms@corpora-sonorus.com

ABSTRACT

Corpus-based Processing for SuperCollider (CBPSC) is a new set of software tools written in the powerful SuperCollider language. The design of these tools takes much inspiration from similar software and from Music Information Retrieval generally. CBPSC is a set of classes and other code that can be used for various purposes, including the building and deployment of databases of tagged audio data. It is the use of this metadata that enables intuitive and intelligent control over musical processes using the associated sound segements. The paper covers many of the features of the core classes, as well as providing some simple examples of its use. As a conclusion, several principles are listed that have informed its design and will continue to influence its continued development.

1. INTRODUCTION

Utilizing aspects of Music Information Retrieval (M.I.R.) and concatenative synthesis, CBPSC combines aspects of both specialized disciplines into a set of tools enabling creative use and intuitive control of the production of sound. CBPSC represents one of the latest iterations of the database-backed audio processing software concept. While M.I.R.-type operations can be integrated into the software framework and concatenative synthesis is among the subset of applications that can be performed within CBPSC, the main innovation of CBPSC is the general set of data structures and functionality that it exposes to the user. CBPSCs intended user is the composer, producer, or performer who uses the software to define and structure a performance or procedure containing elements whose interrelationships may range across a continuum from fixed to free.

2. WHAT IS CBPSC?

CBPSC is a collection of SuperCollider [2] classes. Each core class is organized around certain functionality: database composition, partitioning, search, etc. Further classes contain algorithms for agent-based traversal, Markov-based traversal, state machines, etc. Still more files contain graphical and other interfaces. The user creates code specific to the execution of particular functions, processes, or pieces using these interfaces and basic building blocks. At the core of every corpus-backed operation or system is the CorpusDB class, which over several months has solidified into a class whose data structures represent a database of sound data linked to metadata and operations utilizing such information.

CBPSC is related to earlier versions of corpus-based processing objects for MaxMSP named CBP. That Max 5 version (see [5] for a discussion of some aspects of CBP) is in turn based on CataRT [3]. This SuperCollider version is intended to be a robust, extensible, scriptable, and useful tool for sound design, algorithmic composition, live computer music processing, and/or audio analysis. Already, music compositions, improvisations and demos exist, created by the author, that demonstrate various aspects.

Reflecting a decision made by the author, CBPSC is not specifically geared towards any one technique or artistic process, e.g. concatenative synthesis. CBPSC is intended to be a general toolkit for creating and using database-backed musical applications. However, it is quite easy to create a complete concatenative sampling application within the CBPSC framework. CBPSC is not specifically intended to be used for M.I.R., a research activity concerned with defining and measuring similarities among audio data and/or metadata derived from audio [4]. M.I.R. takes a somewhat broad, in terms of time scales, sizes of corpora, etc., view of sound and sound materials, emphasizing more exhaustive searches of large amounts of data. CBPSC takes much inspiration from M.I.R., and offers the potential for both the creative use of M.I.R. techniques and concepts as well as a framework to support at least a subset of all possible M.I.R. applications.

One other important qualifying remark is necessary: CBPSC's database-centric model implies the inclusion of a structured collection of data. A usage scenario that does not benefit from the use of a backing database of analysis information or that does not track large amounts of data has little use for the structures inherent to CBPSC. Finally, and practically, CBPSC requires at least some programming, which can be a barrier to accessibility in and of itself.

3. CBPSC IN DEPTH

As mentioned above, the code that makes up CBPSC can be broken into four categories: core classes, interface classes, control classes, and functional code that makes use of these classes. The functional bits can themselves be quite extensive. Both iterative functions that form corpora and Routines that deploy audio material based on linked metadata utilize functions and data structures in the core and

control classes, usually with some kind of graphical interface providing feedback to the human operator. Again, every CBPSC application uses an instance of CorpusDB class (or its subclass). See Table 1 for a listing of the main classes.

CorpusDB	The main core class; compulsory.
CorpusSearch	Search functionality.
PartitionedCorpusDB	Main class for partitioned corpora.
PartitionedCorpusSearch	Search functionality for partitioned corpora.
UnitSpace	A general G.U.I. for two-dimensional graphs.
CorpusUnitViewer	A G.U.I. for visualizing units of a CorpusDB.

Table 1. A listing of the core classes of CBPSC at version 0.1.4.

3.1. Using CBPSC

Uses fit into two general patterns: database composition and deployment of resources served from a database. In order to deploy sound samples, one must first compose such a database. Deployment generally includes any procedures that extract metadata from the corpus (or several corpora) and use it to arrange, manipulate, process, or otherwise deploy the sounds in the database associated with that metadata. This process encompasses calls to the core database classes that fetch metadata and data and search functions that use that data. Algorithms that use these hooks do the actual work of rendering the audio pointed to by that metadata.

In the course of developing this database–or better, corpus–of sounds over several sessions, save-and-restore functionality is mandated. Additional processes may be incorporated such that they are run independently of one another. To that end, all classes utilize the XML-writing and reading functions provided by the core database class.

3.2. CorpusDB

The first step in setting up any corpus-based system is instantiating a CorpusDB class. The arguments to the class creation methods are an anchor string and a variable referring to the audio server. [1]

```
~myCorpus = CorpusDB.new(thePath,
        aServerReference);
```

Several functions are used in analysis and segmentation into units:

[1]In SuperCollider, variables beginning with a tilde ( ) have, for most purposes, global scope, and therefore important object references like the CorpusDB object will be assigned to such global variables.

```
~myCorpus.addSoundFile(".../sound.aif");
~myCorpus.analyzeSoundFile(".../sound.aif");
~myCorpus.addSoundFileUnit([0,1000]);
~myCorpus.segmentSoundFiles;
```

Temporal boundaries for units are defined in calls to "addSoundFileUnit". Raw metadata generated by the analysis function is averaged over those segments to compose metadata. Furthermore, there are two functions that handle saving corpora as XML files:

```
~myCorpus.exportCorpusToXML...
~myCorpus.importCorpusFromXML...
```

The guts of the CorpusDB class, as well as the other core classes, contain many more functions than those outlined here. Many of these functions serve the purpose of mapping sound files to different forms of metadata. Figure 2 shows the main keys and values in the Dictionary, a list which reveals a bit about the structure of the database. Each grouping of data with its top-level key represents the rough equivalent of a table in a database. Access is provided through calls like:

```
~myCorpus[\someTable][someKey]...
```

dtable	A Dictionary of integer/descriptor pairs.
sftable	Path string/sound file metadata pairs.
sfmap	Integer index/sound file path string pairs.
sfgmap	Sound file group mappings.
sfutable	Metadata tagged with sound file mappings and relative indices.
cutable	Metadata tagged with corpus-wide unit indices.

Table 2. A listing of the core classes of CBPSC at version 0.1.4.

3.3. CorpusSearch

In order for metadata to be available for search or discovery, it is helpful to utilize a dedicated search class. The CorpusSearch class uses internally a KDTree, itself an implementation for Supercollider of a k-dimensional tree[7], a powerful search technique. The basic search functionality is wrapped in a class:

```
~myCorpusSearch = CorpusSearch.new(~myCorpus);
~myCorpusSearch.buildTree(nil, [0,1,2,3],
        normFlag: true);
```

The search tree is built from an entire associated corpus (the first argument may be a reference to a modified corpus). The second argument, a list, specifies which columns of the metadata table (descriptors) to use, and a flag signals whether or not to normalize the metadata before incorporation into the search tree. Normalization neutralizes the effect of using categories of metadata whose values cover unequal ranges.

After building a tree, a user simply searches for the nearest node to a particular point. Using a two-dimensional normalized search tree, the following is a valid search request that uses a radius to limit the scope of search:

```
~myCorpusSearch.findNearest([0.18,0.56],
        1.0);
```

More complex search schemes may be developed from these simple methods (see PartitionedCorpusSearch, below). In addition to nearest-to-point searches in a given descriptor space, the nodes of the KDTree may be extracted and used to compare distances between nodes, distances from the center of mass of all the nodes, etc.

3.4. Partitioned Corpora and Search

In some cases, one may wish to divide or partition a corpus into two or more sub-corpora. While there are several possible ways of accomplishing this, CBPSC includes two subclasses, PartitionedCorpusDB and PartitionedCorpusSearch, that provide method variants that handle partitioned data. Rather than providing methods that overload non-partitioned methods, new methods work alongside the old. This way, a method using a partitioned corpus can still access the full corpus if need be.

An array containing the descriptor column to query for making the split and a hash table are passed as arguments when creating a PartitionedCorpusSearch instance. In the following example, the value 1 corresponds to the sound file grouping descriptor and the hash table (second argument) maps two partition indexes to six values in the aforementioned descriptor column.

```
~partitioning = [1, Dictionary[0 -> (0..3),
        1 -> (4..5)]];
```

This partitioning is passed as an argument to the call creating a new PartitionedCorpusDB. The subsequent call to create a PartitionedCorpusSearch uses the latest partitioning to have been stored in the associated CorpusDB.

3.5. A G.U.I. Widget/Class: CorpusUnitViewer

The CorpusUnitViewer class supports a simple interface to allow for drawing a two-dimensional depiction of a corpus. It is based on a more general superclass UnitSpace, a class based on graphing discrete points as small on-screen shapes. UnitSpace is based on the existing ParaSpace class [1] with several important modifications. There are several subclasses still in the experimental/prototype stage that also inherit from the basic UnitSpace class; these classes support dragging and zooming points on the basic UnitSpace graph. The dragging functionality has been turned off in the CorpusUnitSpace class, but interaction is still possible for a pointer-based search of the graph.

3.6. Custom Functions

Outside of these core functions is code that performs specific actions and tracks the relationships among data and

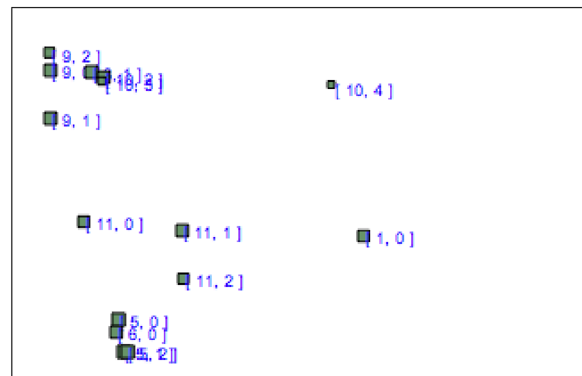

Figure 1. A CorpusUnitViewer. Units are marked-up with some identifying metadata.

metadata in CBPSC applications. A piece that has a CorpusDB instance at its center must be "built out" to include other data structures, some subroutines or functions, a custom G.U.I., perhaps some algorithmic-composition methods, and other code to create a functioning piece of sound art. It is this code, which must often be created anew for each piece or application, that harnesses the general strengths of the CBPSC system. An extremely simple example of some code that plays all the member units of a corpus is shown in Figure 2. As functionality is recreated for multiple pieces, one must consider collecting tasks that can be logically related into a class, and pulling whatever classes that can be sufficiently generalized into the core classes of CorpusDB.

```
~playAll = Routine.new({
    ~someCorpus[\cutable].do({ |unit|
        Synth.new(\unitPlayer, [\uid, unit[0]]);
        unit[6] * 0.001.wait;
    });
}).play;
```

Figure 2. Some simple custom code that plays all the units of a corpus, one after the other. The variable "unit" is an array with all the metadata associated with that unit.

4. UNITS AND PERSISTENCE THROUGH XML FILE STORAGE

As important to CBPSC as the database metaphor is the basic concept of the unit. A unit is a segment of a sound file defined by specific temporal endpoints. Within a corpus, unit tables contain rows of units. Information in a unit row beyond that which defines its bounds is descriptive in nature, consisting of metadata gathered during analysis of the raw audio or attached by an interactive process. Furthermore, the unit is an abstract entity that can be represented in different ways based on its measured features. Please see the author's earlier paper [6] for a much deeper discussion of units and other concepts on which this software is based.

As mentioned above, the information gathered through analysis and database composition can be saved to XML

files and made persistent. The XML files have a defined structure that enables import and export functions to reconstitute a corpus properly in memory. The structure includes a listing of all the descriptors (Figure 3), the descriptive metadata that define the units for that corpus.

```
<heading name="DMAP">
        <descrid name="unitID">0</descrid>
        <descrid name="sfgrpID">1</descrid>
        <descrid name="sfileID">2</descrid>
        <descrid name="sfRelID">3</descrid>
        <descrid name="onset">4</descrid>
        <descrid name="duration">5</descrid>
        <descrid name="tartiniFreq">6</descrid>
        <descrid name="tartiniHasFreq">7</descrid>
        <descrid name="power">8</descrid>
        <descrid name="flatness">9</descrid>
        <descrid name="centroid">10</descrid>
        <descrid name="zerox">11</descrid>
        <descrid name="flux">12</descrid>
        <descrid name="rolloff">13</descrid>
        <descrid name="slope">14</descrid>
        <descrid name="spread">15</descrid>
        <descrid name="crest">16</descrid>
</heading>
```

Figure 3. The list of descriptors that appears in each metadata file.

This list of descriptors maps directly to the columns of the unit tables. Figure 4 shows a single typical unit. In this case, the unit is known as a corpus unit, because it is defined within the scope of one whole corpus. Many units may map to a single sound file, but if they have identical temporal boundaries but differing data, they should be considered inaccurate and/or poorly defined.

```
|    <corpusunit sfid="0" relid="0">0 0 0 0 0
555.87301587302 258 0.529 0.02619 0.1532 2040 0.9732
0.3689 5275 -0.017 0.1418 0.4353 0.845791 0.354833
0.0230932 0.00574027 0.130343 0.113005 0.129433
0.166138 0.188011 0.261277 0.255017 0.231954 0.22819
0.236639 0.246733 0.252072 0.243705 0.19832 0.198587
0.191352 0.195163 0.252803 0.296006 0.291954
129.96878039068"</corpusunit>
```

Figure 4. The descriptive metadata for a single unit. In addition to the listed descriptors, the data includes a series of Mel-frequencies Cepstral Coefficients derived during analysis.

5. SUMMARY OF GOALS

This paper outlines much of the core functionality of this implementation of corpus-based processing. As other implementations of this sort of software are different, it is important to emphasize the main features of this version, as these features drive development.

CBPSC is created with the composer or producer in mind: one who wishes to harness both the power of an organized database of sound resources and who wants to work with that data on a conceptual level that includes the ability to organize and reorganize that data in myriad ways. Thus, the reliance on SuperCollider for its rich set of data structures, client-server architecture, and flexible language. CBPSC also has to be powerful and efficient enough for practical use (including real time considerations) and general enough to both not impose too much structure on the user and be easily extensible. The barriers to access (choice of SuperCollider as an environment, required knowledge of programming and design principles, etc.) are such that it makes sense to release CBPSC with an open source license. Users are encouraged to contribute changes and ideas for revisions back to the core project, or fork CBPSC for their own customization. CBPSC is ready to incorporate interfaces to existing and emerging technologies. In addition, this system should be able to exchange data with other systems, where practical, and be able to support any protocols that may emerge as as standards.

6. MOVING FORWARD

These considerations will be used to push development forward. The author continues to use this code to create music, meanwhile implementing new features, fixing bugs, and researching ways to improve the software. As the set of techniques and ideas surrounding intelligent musical processing grows, it is hoped that the potential for creative use has outlets. CBPSC aims to be one of these outlets. It is already a compositional tool for its author, and it is hoped that others find it useful.

CBPSC is available under a GPL license at www.corporasonorus.com.

7. REFERENCES

[1] T. Magnusson, http://www.ixi-software.net/content/download/ParaSpace.zip, 2006, accessed January 24, 2011.

[2] J. McCartney, "Supercollider: A new real time synthesis language," in *Proceedings of the International Computer Music Conference*, Hong Kong, 1996.

[3] D. Schwarz, G. Beller, B. Verbrugghe, and S. Britton, "Real-time corpus-based concatenative synthesis with catart," in *Proceedings of the COST-G6 Conference on Digital Audio Effects (DAFx)*, Montreal, Canada, 2006, pp. 279–282.

[4] "Definition of the field," http://smcnetwork.org/roadmap/definition, SMC, 2007, accessed January 24, 2011.

[5] T. Stoll, "Beyond concatenation: Some ideas for the creative use of corpus-based sonic material," in *Proceedings of the International Computer Music Conference*, Montreal, 2009.

[6] ——, "Abstraction in a unit-based audio processing system," in *Proceedings of the International Computer Music Conference*, New York, 2010.

[7] http://en.wikipedia.org/wiki/Kd_tree, wikipedia.org, 2011, accessed May 11, 2011.