Loop-based Composition with RTcmix and Hula

John Gibson
Center for Electronic and Computer Music
Indiana University School of Music
johgibso@indiana.edu

Abstract

Hula is a composition tool designed for the creation of looping textures. Written in Python, and interfacing with the RTcmix synthesis and processing package, Hula supports sampling, synthesis and effects within an object-oriented scripting environment. This paper briefly introduces RTcmix, focusing on its looping and granulating abilities, and then presents an overview of Hula syntax and features. Some thoughts about future extensions of Hula follow.

1 Introduction

One of the more intriguing aspects of electronic music today is the cross-fertilization of ideas between composers trained in the Western classical tradition and musicians involved in the various DJ and dance music sub-cultures. "Classical" composers like me, drawn to the music of Autechre, Photek and Squarepusher, are thinking of ways to incorporate the loop-based ideas prevalent in this music into our own pieces.

Commercial programs such as Sonic Foundry's Acid let you combine and loop canned sound files with a friendly graphical user interface. Acid performs real-time pitch-shifting and time-scaling to match the pitches of samples and to coordinate their tempi. But the program assumes that you want to do these things — that you want to make a conventional rock song or dance track.

I take a different approach. Using RTcmix, a program that I develop with other composer-programmers,[1] I build a system that generates repetitive textures algorithmically. The system allows for great flexibility in the coordination, selection and processing of samples, and also includes synthesis capabilities.

RTcmix has its roots in the open source computer music programs derived from work at Bell Labs in the 1960's. Recent extensions to RTcmix allow us to use two rich scripting languages, Perl and Python, to control synthesis and processing. Python in particular takes an object-oriented approach that can model high-level musical ideas well.

In this paper I give a brief introduction to RTcmix, focusing on its looping and granulating capabilities. Then I present Hula, a Python program that harnesses the power of RTcmix to offer a loop-based scripting environment. Hula lets a composer develop complex textures that comprise multiple layers, each looping at independent rates and periods. Scripts are written in Python, so they have full access to Python language features. Currently, Hula is non-interactive and text-based, but an interactive mode for live performance and a graphical interface are under consideration. Hula, like RTcmix, runs on Linux and Mac OS X platforms.[2]

2 RTcmix Loops and Grains

RTcmix is a package of plug-in *instruments*, which synthesize or process sound; utility functions that create tables, open sound files, generate random numbers, and so on; a scheduler that determines when to play each instrument; and a parser to read scores. Instruments are written in C++, so extending RTcmix is possible. (RTcmix is distributed with over 60 instruments.) Instruments can be connected using a system of internal buses, rather like the *insert* effects in Pro Tools and similar programs.

Composers typically use RTcmix in one of two modes: *interactive* or *non-interactive*. In interactive mode, RTcmix listens for incoming instructions and responds more or less immediately, by playing a note or performing some other task. (A *note* is the sound generated by one instance of an instrument, regardless of whether this comes across as one note in the conventional musical sense.) In this mode, RTcmix can be used as the synthesis engine for a graphical user interface program or a real-time performance system. In non-interactive mode, the composer gives RTcmix a text *score*, which describes actions to take and when to take them. RTcmix parses the score, schedules notes, and then plays the notes in real time or writes the resulting sound to a file.

[1] The principal authors of RTcmix are Brad Garton, John Gibson, Douglas Scott and David Topper, with major contributions from Mara Helmuth, Luke Dubois, Ivica Bukvic and others. RTcmix was derived from Cmix, written by Paul Lansky. Download the RTcmix source code at `ftp://presto.music.virginia.edu/pub/rtcmix`. Additional information is available at `http://music.columbia.edu/cmc/RTcmix`.

[2] Hula is available at `http://mypage.iu.edu/~johgibso/hula.htm`. It requires a working installation of RTcmix with the Python parser enabled.

```
rtinput("mysound.aif")           // open input sound file
filedur = DUR()                  // get file duration
totdur = 12                      // total output duration (seconds)
increment = 0.25                 // one note every 1/4 second
notedur = increment * 1.2        // notes will overlap slightly
maxgain = -3; mingain = -18      // max and min volume, in decibels

// simple linear ramp amplitude envelope; applies to each note
setline(0, 0,  0.001, 1,  notedur - 0.001, 1,  notedur, 1)

// loop for <totdur> seconds; a note plays every <increment> seconds
for (start = 0; start < totdur; start = start + increment) {

   // irand() returns a random number within the specified range;
   // <inskip> is the duration to skip when reading from the input file
   inskip = irand(0, filedur - notedur)

   gain = irand(mingain, maxgain)
   amp = ampdb(gain)             // convert dB to amp multiplier
   pan = random()                // random number in range [0,1]

   STEREO(start, inskip, notedur, amp, pan)     // play the note
}
```

Figure 1: Simple RTcmix looping score, written in the MinC scripting language

For many years, RTcmix was limited to a single scripting language: MinC (for "MinC is not C"), a simple C-like interpreter written by Lars Graf in the mid 1980's. A MinC script is a series of statements that perform computation, define loops and conditions, and call functions, including the RTcmix instruments.

Constructing loops easily is a strength of RTcmix. Figure 1 shows a score fragment written in MinC. This score opens a sound file and plays notes using the STEREO instrument, which is a basic mixer with static pan capability. The notes are drawn from random spots in the sound file, and each note has a random amplitude and pan location. The result is a stream of different-sounding notes, but with all notes having the same duration and all separated by the same time interval.

MinC makes it easy to organize your work using variables, and it includes a modest facility for creating and accessing lists of values. So it would be possible to extend the example above to make STEREO read from several different sound files at once, or to make the attack rhythm follow a series of values stored in a list, rather than playing a steady stream.

The score above could form the basis of a full-featured granulator. Each call to STEREO would play one grain. You could think of the *increment* variable as a way of expressing the grain rate. You might want to provide dynamic control over this, by making the rate follow a curve. The same could be applied to grain (i.e., *note*) duration. You might want to include a way to randomize the start times of successive grains, so as to offer asynchronous, as well as synchronous, granulation. You could make use of the TRANS instrument, which transposes its input, to provide randomly transposed grains. Finally, you could construct a way to traverse the input file at a particular rate, rather than just grabbing random bits of sound from all over the file.

The problem is that to do a very complete job of a score like this, you would run up against some major limitations of MinC: there are no user-defined functions, there is no way to refer to code modules living in other files, handling lists is rather awkward, and so on. In short, doing anything very complicated in MinC can be painful.

For this reason, in 2000-2001 I added two alternative interpreters to RTcmix: Perl and Python, both popular, well-supported, general-purpose scripting languages. These greatly enhance the score-writing process. Using them with RTcmix is no different from using MinC — you just write your score with any text editor and feed it to the appropriate version of RTcmix. These extension languages open up vast new territory for exploring algorithmic composition, and they make possible the construction of large, scriptable applications that remain tightly integrated with the RTcmix core.

3 Doing the Hula

Shortly after working with the Python parser for RTcmix, I realized that my gangly, unmanageable MinC scripts could

be turned into a generalized application, which would allow me to make complex loop-based scores much more easily and more flexibly than is possible with MinC. That application is Hula, a Python program that provides a powerful scripting interface to RTcmix.

Others have recognized the value of Python for music systems. For example, Dannenberg (2002) modeled his Serpent language on Python. But Hula uses Python more narrowly, as a way to specify higher-level looping constructs, not as a way to design the networks of unit generators that create audio streams. For Hula, the latter is done inside RTcmix instruments, which are accessible to instrument designers only via C++. This makes Hula less flexible with respect to sound rendering than other scriptable computer music systems, such as Csound (Boulanger 2000) and SuperCollider (McCartney 2000), but also less complicated for beginning users, who do not have to worry about such lower-level details when learning to make loop-based music with the system.

A typical Hula script is shorter and simpler than the MinC equivalent. For example, Figure 1 could be written in Hula as Figure 2. Central to Hula is the notion of an *attack pattern*.

```
from randskip import *
file = "mysound.aif"
attacks = (0, 0.25, 0.5, 0.75, 1)
dur = 0.25 * 1.2      # note duration
transpose = gain = pan = seed = 0
skipper = RandSkip(attacks, dur, gain, pan,
                        transpose, seed, file)
skipper.set_gain_range(-18, -3)
skipper.auto_mutate_gains()
skipper.auto_mutate_pans()
skipper.auto_mutate_inskips()
skipper.play(0, 12)   # play pattern 12 times
```

Figure 2: Hula equivalent of Figure 1

This is simply a list of attack points, expressed in quarter-note beats, and terminated by the total number of beats in the pattern. So the attack pattern in Figure 2, stored in the *attacks* variable, consists of four sixteenth notes spanning exactly one beat — the notes start at times 0, 0.25, 0.5 and 0.75. The attack pattern for "I been workin' on the railroad" would be (0, 1.75, 2, 2.75, 3, 3.75, 4, 6, 8), assuming the short notes are sung as sixteenths. This pattern has eight notes of varying lengths; the total pattern is eight beats long. So an attack pattern does not have to be a stream of equal note values. Making a MinC script equally flexible would require extra code, some of which would need to handle the representation of time in beats, rather than seconds.

After specifying an attack pattern, you create a Hula *player object* of some type. RandSkip is a type of player that can skip randomly around the input sound file while reading portions of it. (There are many other player types, a few of which I discuss below.) As you might expect, you can have many players working together, with their own attack patterns and other properties, and they may have independent tempo maps. In the Hula score above, we create one RandSkip object, arbitrarily called "skipper." Then we set various properties of skipper, such as the gain range (in decibels), by calling *object methods*. Finally, we tell skipper to play its pattern twelve times. Hula assumes that you want to create a pattern and hear it repeat.

Our score plays sound from disk without any transposition. But by supplying a list of values in the *transpose* variable, as opposed to a single value of zero, we can ask the player to transpose each note of the pattern differently. For example, if the input file were a clearly pitched sound, the following attack pattern and transpose list would play the opening of "Happy Birthday" (disregarding its meter).

```
attacks = (0, 0.75, 1, 2, 3, 4, 6)
transpose = (0, 0, 0.02, 0, 0.05, 0.04)
```

Transposition values (and pitch values for other Hula player objects), are specified in *octave-point-pitch-class* notation.[3] Similarly, we could request different note durations, pan locations and gains for each note in the pattern.

The power of Hula derives mainly from the variety of player properties available to the script author. Some of these are designed to make successive iterations of a pattern sound different. The various *auto-mutate methods* allow you to randomize, within a range that you can specify, parameters such as gain, pan, duration, pitch, inskip, attack time smear, etc. Also, there are ways to have the attack pattern itself change while looping. The *auto_mutate_attacks* method randomly constructs, for each loop iteration, a new attack pattern having the same number of notes as the original pattern, and using a grid formed from the smallest note value in that pattern. The *auto_shuffle_attacks* method scrambles the series of durations represented by the original pattern. We could generate obscure variations on the "Happy Birthday" rhythm by including the following lines in our Hula script.

```
skipper.set_attacks(0, 0.75, 1, 2, 3, 4, 6)
skipper.auto_mutate_attacks()
```

This randomly redistributes the lone dotted eighth and sixteenth notes within the pattern, while retaining the number of attacks and the pitch sequence.

Using other Hula player objects is similar. The OneShotPitch object is a one-shot sampler that selects sound files based on their proximity to the pitches you specify. The sound files must be in a single directory that also contains an index file listing the pitch of each sample. Figure 3 (next page) is a Hula score for two instruments, bass and ride cymbal, performing typical jazz rhythmic patterns. The cymbal player uses RandSkip and behaves as in Figure 2, except that

[3] That is, the value to the left of the decimal point gives the octave, and the value to the right of the decimal point gives the pitch class, or semitone, relative to C as zero.

he doesn't skip randomly in the input file, because he doesn't use *auto_mutate_inskips*. For the bass player, we specify a collection of pitches and tell her, via the *auto_mutate_pitches* method, to select randomly from them when playing her rhythmic pattern. OneShotPitch chooses samples from the specified directory according to how closely they match the pitch for the current note. You could vary the randomly selected pitches by changing the value of the *seed* variable, which initializes the bass player's random number generator. If you wanted, you could give each player its own tempo map, instead of requesting a global tempo of 220 BPM.

```
from randskip import *
from oneshotpitch import *

set_audio()         # set up audio converter
set_all_tempos(220)        # for all players

# create bass player
attacks = (0, 1, 2, 2.66, 3, 4)
dur = 1.2                  # note duration
pan = 0.7
pitch = gain = 0
dir = "/snd/bass/uprights/"
bass = OneShotPitch(attacks, dur, gain, pan,
                            pitch, seed, dir)
bass.set_pitch_collection(
   6.00, 6.02, 6.03, 6.05, 6.07, 6.10, 7.00)
bass.auto_mutate_pitches()

# create cymbal player
attacks = (0, 1, 1.66, 2, 3, 3.66, 4)
dur = 1.8
pan = 0.3
file = "/snd/cymbals/rides/ride1.aif"
cymbal = RandSkip(attacks, dur, gain, pan,
                            pitch, seed, file)

# set up random gain changes
bass.set_gain_range(-15, -3)
bass.auto_mutate_gains()
cymbal.set_gain_range(-18, -9)
cymbal.auto_mutate_gains()

# play pattern ten times
cymbal.play(0, 10)
bass.play(0, 10)
```

Figure 3: Hula Jazz Jam

Some Hula player objects are designed to process audio generated by other objects. Let's say we wanted to add an effect to the cymbal part, such as a shelving equalizer to brighten the sound. We simply tell the cymbal player to send his output to a bus.

```
cymbal.set_out_bus("bus 0-1")
```

Then we create an equalizer to read from that bus.

```
from eq import *
dur = get_total_beats(attacks, 10)
eq = EQ(0, dur)
eq.set_eq_type("high shelf")
eq.set_cutoff(2500)   # in Hertz
eq.set_eqgain(6)      # shelf height in dB
eq.play()
```

Since the equalizer doesn't play a rhythmic pattern, we have to tell it how many beats of sound to expect at its input. The *get_total_beats* utility function takes an attack pattern and the number of loop iterations (ten in this case) and returns the number of beats required. There are many such utility functions in Hula.

Perhaps the most complex Hula player object is Granulate, which offers extensive control over the granular sampling process. Figure 4 is a Hula score fragment that granulates an input sound file. It lets you adjust the rate and direction of traversal through the file, as well as grain density and various forms of *jitter*, or random variation of a parameter. It allows you to create harmonies by transposing grains to fit a *transposition collection*. Some methods accept either constants or a line or curve specification.

```
attacks = (0, 5, 10, 15)  # three notes
dur = 12                  # same for each note
gain = -9
pan = (0.5, 0.1, 0.9)
transp = (-0.07, -0.05, -0.03)
seed = 1
file = "applepie.aif"
granny = Granulate(attacks, dur, gain, pan,
                           transp, seed, file)
granny.set_transp_collection(0, .01, .05, .07)
granny.set_transp_jitter(0.07)
granny.set_grain_dur(0.2)
granny.set_traversal("line", 0, 0.001, 1, 1)
granny.set_density("curve", 0, 200, 1, 1, 50)
granny.set_grain_envelope("hanning")
granny.set_pan_jitter(0.1)
granny.play()
```

Figure 4: Hula Granulation

Since a Hula score is written in Python, it's easy to extend it using features of the Python language. For example, you can use Python lists and control flow statements to create more complex scores. You can even invent Python objects that combine multiple players into a "super player." You then could make libraries of such players and import them into your scores.

4 Future Directions

Of course, all these scores are non-interactive. While Hula is playing a loop or granulating a sound, you have no influence over it. So the Hula composition process involves writing a score, listening to the score, fixing the score, listening again, and so on. Wouldn't it be nice to make the process interactive in some way, at least as an option? A new version of RTcmix, currently under development, will enable the specification of various interactive control sources — such as MIDI messages, sensor inputs or GUI events — within a Python script. Hula could readily take advantage of this functionality to allow interactive control over sound rendering. But this won't solve the problem of interactive control over higher-level looping note patterns. Addressing this deficiency could entail a dramatically different way of connecting Hula to RTcmix. On the other hand, creating a graphical interface for Hula that substitutes curve-drawing and button-pressing for score-typing, prior to submission of the score, would not present any serious problems. The trick would be to design such an interface so that the flexibility offered by the Python language is still available to advanced users.

References

Boulanger, R. (Ed.) (2000). *The Csound Book*. Cambridge, Massachusetts: MIT Press.

Dannenberg, R. B. (2002). A language for interactive audio applications. In *Proceedings of the International Computer Music Conference*. International Computer Music Association.

McCartney, J. (2000). A new, flexible framework for audio and image synthesis. In *Proceedings of the International Computer Music Conference*. International Computer Music Association.