Agents in ChucK: A Timely Programming Experience

Michael Spicer
School of Media and Info-Communications Technology, Singapore Polytechnic
mspicer@sp.edu.sg

Abstract

An agent based interactive composing system, developed in the ChucK environment is described. The system makes use of simple autonomous agents to achieve high level musical goals, specified by a human performer in real time. The agents autonomously work to achieve the goals at the note level and at the synthesis level. A primary motivation for this project was to get an intuitive feel of the effectiveness in working in an environment optimized for “live coding” to improve my productivity.

1 Introduction

In late 2005, I had reached a point where I felt I needed to take stock of my situation. I had been working on agent based interactive composition systems for 5 years, and was getting less and less time to develop software systems. I felt that I needed to improve the balance of software development time verses music output. I had just reached a point in the development a new system (SPAA) that it was usable and was considering options as to how to proceed. I am quite a fast C++ programmer, but there is a lot of DSP coding that needed to be either "found" or programmed from scratch. The SPAA system also had performance trouble. In order to get many agents running at once, the system will need to be distributed over a number of computers, probably using OSC, which would require more work.

At the same time, I found it intriguing that the "Live Coding" performance was starting to catch on in some circles, and reasoned that the use of this sort of programming environment should be much more productive than my C++/Java environments. I decided to temporarily halt development on the next phase of SPAA and explore the idea of implementing a simple agent based composition system in a live coding development environment.

2 Live coding options

The main live coding environments that I considered were SuperCollider and ChucK. Initially I thought that SuperColloider would be the obvious choice, but then I discovered that the latest version of ChucK had become object oriented so I decided to investigate both. I began to to work through some tutorials in SuperColloder, and quickly discovered that, although it was object oriented, its' Smalltalk “accent” did not neatly fit into my C++ style of thinking. SuperCollider worked at a higher level of abstraction, too much "magic" seemed to happen behind the scenes for my taste, and I felt I would need to start to learn a new programming style from scratch. (I should say that the SuperCollider tutorials were also quite good, and they very clearly showed how different the programming style was). I then tried ChucK, and after working through the tutorials, and found that it very easily fit my C++/Java centric programming style. I found it was designed to be very similar to Java. It did take a while to get used to the ChucK operator (=>) and the “strongly timed”, inherently multithreaded (or should I say multishredded?) approach, but the tutorials and the example code included in the package (and online) gave me all the clues I needed to leverage my Java experience.

After playing around with the live coding approach that the tutorials promoted, I started to create a few classes. It quickly became apparent that it would be very simple to modify my existing Java objects (such as wavetable oscillators, contour generators and probability distribution generators) to run in ChucK. To a large extent, the process was:

- Paste the Java class definition into the ChucK file.
- Replace all the assignment operators (=) with =>, as well as move it to the other end of the line.
- Modify any for loops in a similar way.
- Add the key word fun to the front of each method definition.

The biggest change was in way instances these objects were used in control loops of the program.

3 Overall System design

The basic agent I decided to implement is an adaptation of the agent design used in my previous systems. The idea was to create an ensemble of agents, acting as virtual musical performers. The composer initializes each agent, and in performance, provides a set of high level musical goals. The agents collectively modify their internal states so as to achieve the high level goals. The goals are set using MIDI continuous controller messages. (Unlike my other systems, this system has no user interface.) The agents
normally cooperate to achieve the goals, but they may occasionally ignore them or deliberately avoid them, to add an extra element of uncertainty.

The thing that really separated the agents in this ChucK based system from the previous systems is that the agents have a much more micro level control of the sound synthesis. The previous systems worked at the level of MIDI message, or on large buffers of live audio input. In this system each agent has control of the parameters of a simple FM synthesizer (Chowning 1973), and the agents (mainly) cooperate to create a sound world that mirrors the goals specified by the user. An agent can be divided into two sections: the control section and the synthesis section.

2.1 The synthesis section of the agent

Each agent has its own simple FM synthesizer, with its own reverb and running in stereo. I choose to use FM due to the variety of sounds it can produce predictably by modifying very simple parameters. The synthesizer just a simple extension of the ChucK vibrato tutorial!

The signal flow is quite clearly reflected in the code, once you understand:
- The ChucK operator indicates the audio signal flow.
- The modulating oscillator needs to be connected to the “blackhole” if it is not connected to the dac.
- The modulation oscillator cannot be connected to the carriers’ frequency control directly. You need to get its’ value using the last() method and patch to the carrier frequency input using the ChucK operator.

The code runs in its own thread and is updated every sample (i.e. at the audio rate) and it obviously does run a lot slower than the equivalent in C++, but was very fast and easy to implement. Hopefully the ChucK runtime will become faster soon.

2.2 The control section of the agent

The agents state is updated every 100 milliseconds (the K rate). It updates the various parameters so as to achieve the users’ current goals. The basic note level parameters controlled by each agent are pitch and duration and work using the same approach used with the AALIVE (Spicer, Tan, and Tan 2003) and AALIVENET (Spicer 2004) programs (the two Wavetable Oscillator / Sample and Hold / Finite State Machine design). In fact, I used the process described above to modify the Java code from AALIVENET to produce the core of the agent control section in just one evening.

I decided to quantize both of these parameters so as make the system easy to improvise with using traditional instruments, so the pitches are mapped to equal tempered semitone (at A 440) and the timing is quantized to various multiples of 16th notes. These can easily be modified as required with a few lines of code.

The synthesis parameters controlled autonomously by the agents are:
The amount of high frequencies presents (the modulation index),
How harmonic/inharmonic the sound is (determined by the C:M ratio).
The amplitude envelope.
The dry signal/reverb Mix.

In addition the user has direct control over the overall loudness and textural density
The structure of the agent is shown below.
The agent consists of a number of arrays containing the shape contours of the various parameters. The agents cycle through this shape every 30 seconds or so (as specified by the user at compile time). These values are applied to the agents’ synthesizer. Every 100 milliseconds, the average value of each array is calculated, and then the average value of all three agents is calculated. These values are compared to the target values, set by the human performer, and an error for each parameter is determined. The agent then applies small bias to each array, so as to gradually reduce the errors (a type of gradient descent learning). In this way, the agents converge on a musical output that corresponds to the users goals.

2 Trio: The Composition

Over the period of building the system, I experimented with some of its various possibilities and the shape of first piece emerged. Using my 1.5 GHz Macintosh Powerbook, I could only run three agents without using too much CPU, so my first ChucK piece became a piece for three agents i.e. a trio. 8 MIDI continuous controllers (faders, mod wheel and expression pedal) are used to set the human performers goals. Each wavetable in all the agents were initialized with a fractal line generation routine. A typical example of the resulting contour is shown below.

Some contours of some parameters were scaled and appropriately biased, based on compositional needs. For example, the pitch contour for each of the agents is first normalized and then scaled to cover one third of the agents pitch range. Then a bias of 0.33 is added to one agent, and a bias of 0.66 for another, so all three agents will initially be set to cover different registers.

The large scale structure of the piece is formed by the human performer setting a sequence of goal states, in real time, using the MIDI controller. After much exploration and fine tuning, so as to get a feel of how the agents respond when they transition from particular goal states, a general succession of goal states was determined. After using the system for some private performances to some colleagues, I decided to turn off the agents’ ability to be uncooperative. Even with the agents mainly cooperating, it sometimes got a bit too unpredictable for me. I put this down to the fact that there were only three voices in the texture, so everything is a bit exposed. With more agents, the occasional agent being disruptive should just add “healthy dissent” to keep things interesting.

3 Conclusion

In a relatively short time, I managed to learn enough ChucK programming to build a reasonably complex interactive system (although I am still not an expert live coder!). The idea of choosing this particular “live coding” environment to improve my productivity paid off, due to speed with which I could try things out, and the fact that I could leverage my C++ and Java programming experience. Being able to quickly translate code from one environment to the other was a big advantage for me. I would recommend any experienced C++/Java programmer interested in live coding to at least try ChucK for this reason.
However, ChucK is not without its’ shortcomings. It is still in development, and runs a bit slow. In the future, hopefully ChucK will be faster, so that more complex systems can be built. In the meantime, there are various networking approaches (Wang et al. 2006) that could distribute the computing load.

The command line programming experience for this project was a bit of a “blast from the past”, reminding me of my early UNIX days. I did not try to use “The Audicle” programming environment for ChucK, as I gathered it was a bit too premature. I found that I missed the luxury of having an interactive debugger and an integrated development environment. Hopefully “The Audicle” will provide better programming experience eventually. Even so, I did enjoy programming this project with ChucK, even with the command line interface.

Overall, the project was a positive experience. In addition to learning a new programming language and try a small amount “live coding” (in rehearsal), I did manage to explore the possibilities the agents working the synthesis level, and build a moderately complex object based system, in about 400 lines of code. Unfortunately, I don’t think ChucK executes fast enough to realize the ideas I have for the SPAA system, so I’ll have to stick to C++ for that, but I will not be giving up ChucK development. I think it will be very helpful for those small one off projects that regularly crop up, will greatly help me manage my time!

4 Acknowledgments

Thanks to Ge Wang, Perry Cook and the rest of the ChucK team for building Chuck.

References


