Notes on ChucK for Music

Chapter 1
Music Basics

1.1 Introduction

You should probably start with the ChucK Tutorial. ChucK Tutorial

This tutorial explains how to make a certain type of music in a certain way. Since ChucK is almost a general purpose language, there are many such ways to make beautiful music using it. As a work in progress, this document encodes groups of techniques, and invites other approaches and perspectives.

Please send comments to graham.coleman at upf.edu.

1.2 Melodies and Scales

ChucK rocks. So you want to lay down some chords, melodies, etc? Play all night in your {country, reggae, synthpop, hiphop, laptop, forró} band? Let's do some of that.

By giving you sample-level control within the language, ChucK allows you to be original with your synthesis algorithms. Since we're out for chords, we'll be unoriginal for the moment. Fortunately ChucK has baked-in STK, a collection of realtime synthesis algorithms. Programming Guide: Unit Generators

```
//connect a plucked string to the soundcard out
StifKarp inst => dac;
```

Instruments usually input raw frequencies, but since we're using western scales, we need a basic way to convert those into frequencies. std.mtof (Midi-to-Frequency) converts the midi key numbers (read: 88 keyboard keys) into frequencies acceptable for use.

```
std.mtof( 60 ) => inst.freq; //set the note to middle-C
inst.noteOn( 0.5 ); //play a note at half volume
1::second => now; //compute audio for 1 second
```

Go ahead and execute the code to see if it works.

If you are using command-line chuck:

Type the code into a source file, save in, and run *chuck tut01.ck* (implied file name) at the prompt.

If you are using the audicle:

Start the Audicle. The shrEditor face (the code editor) should come up by default. Start a new buffer with Ctrl-N, and type in your source code. Execute the code by clicking the (S) bubble (for spork) or typing Ctrl-L. You can kill the new shred by clicking on the numbered bubble.

ChucK will play a middle C. Next, we'll play a sequence.

```
[0,2,3,1,4,2,6,3,4,4] @=> int mel[]; //sequence data

for (0=>int i; ; i++) { //infinite loop
  std.mtof( 48 + mel[i%mel.cap()] ) => inst.freq; //set the note
  inst.noteOn( 0.5 ); //play a note at half volume
  300::ms => now; //compute audio for 0.3 sec
}
```

Combine this code with the instrument declaration code and run it. The 'for' loop will loop through the sequence data. The index 'i' grows indefinitely, but we wrap it to the size of mel[] with the mod operator.

This melody is 'chromatic'. It selects black or white keys from the keyboard indiscriminantly, which is why it doesn't sound like any particular scale. To play something more conventional, we can encode a major scale.

```
[0,2,3,1,4,2,6,3,4,4] @=> int mel[]; //sequence data
[0,2,4,5,7,9,11,12] @=> int major[]; //major scale

for (0=>int i; ; i++) { //infinite loop
  std.mtof( 48 + major[mel[i%mel.cap()]] ) => inst.freq; //set the note
  inst.noteOn( 0.5 ); //play a note at half volume
```

```
  300::ms => now; //compute audio for 0.3 sec
}
```

major encodes the semitones within the scale. From the 1st note of the scale, we have a whole step to the next note (2 half steps), when another whole step, then a half step, etc. To play the scale, we simply select a scale degree (1st, 2nd, 3rd...) and use it to index the scale. We add this to an offset (48). Since 48 is a C, this will play notes from the C scale.

1.3 Flexible Scales

This works, but we can only select degrees within one octave of the scale. We'd prefer for the scale matrix to wrap indefinitely upwards. For example, if you generated a melody that asks for the 8th scale degree, you'd like it to pick a note from the next octave of the scale. This scale would look like this:

[0,2,4,5,7,9,11, 12,14,16,17,19,21,23, 24,26,28,29,...]

We can simulate this behavior with a function:

```
fun int scale(int a, int sc[]) {
  sc.cap() => int n; //number of degrees in scale
  a/n => int o; //octave being requested, number of wraps
  a%n => a; //wrap the note within first octave

  if ( a<0 ) { //cover the negative border case
    a + 12 => a;
    o - 1 => o;
  }

  //each octave contributes 12 semitones, plus the scale
  return o*12 + sc[a];
}
```

Now use this function to simulate indexing into the scale.

```
[0,2,3,1,4,2,6,3,4,4] @=> int mel[]; //sequence data
[0,2,4,5,7,9,11,12] @=> int major[]; //major scale

for (0=>int i; ; i++) { //infinite loop
  std.mtof( 48 + scale( mel[i%mel.cap()], major )) => inst.freq; //set the note
  inst.noteOn( 0.5 ); //play a note at half volume
  300::ms => now; //compute audio for 0.3 sec
}
```

To play from a different scale (G major) we can just change the offset:

```
std.mtof( 3*12 + 7 //3rd octave, 7 semitones from C
  + scale( mel[i%mel.cap()], major )) => inst.freq; //set the note
```

To play a harmony, simply add a constant to the melodic contour:

```
std.mtof( 3*12 + 7 //3rd octave, 7 semitones from C
  + scale( mel[i%mel.cap()] +5, major )) => inst.freq; //set the note
```

If you are using Audicle, try spawning off several copies of this with different melodic offsets. There we go, almost musical!

To speed improvisation, I abstracted the functionality into a class Scale. Also, there are common scales such as Major (sc.maj), Minor (sc.min), and a few more exotic ones: scale.ck

To use classes from outside a source file, you'll have to bring them into the VM as well. Currently, there isn't an automatic way of doing this (like the Java classpath) but hopefully it will be easier in the future.

If you are using the command-line chuck:

Send both the Scale class and your source to the vm: *chuck scale.ck tut05.ck*.

If you are using the audicle:

Open the audicle and the shrEditor comes up by default. Press Ctrl-F to open file and select the scale.ck file. Ctrl-L

or hit the (S) bubble to execute the class (this compiles the class into the VM for this session). Next, start a new buffer with Ctrl-N, type in your code, and spork it off.

```
Scale sc; //include this at the top

std.mtof( 3*12 + 7 //3rd octave, 7 semitones from C
  + sc.scale( mel[i%mel.cap()] +5, sc.gypsy )) => inst.freq; //set the note
```

This ends our lesson on scales.

1.4 Timing

The Timing section from the language spec explains the basics really well. Language Specification: Time

One strategy for synchronizing a bunch of shreds to a musical structure (like a chord progression) involves using the following two operations, from the spec:

A. (synchronize to period):

```
// synchronize to period of .5 second
.5::second => dur T;
T - (now % T) => now;
```

Synchronize to period computes the offset from even multiples of a beat, and advances time by that amount. 'now' time is a global phenomenon, so if several shreds do this independently, they will end up in lockstep, at least temporarily.

This example demonstrates beat synchronization with a simple ascending line. Supply an instrument declaration, instantiate Scale, and send a few copies of this to the VM, using either 'chuck + <file>' or Audicle:

```
1::minute/140/2 => dur T; //140 bpm eighth notes
T - (now % T) => now; //sync to beat

for (0=>int i; ; i++) { //infinite loop
  sc.scale( (i%10)*2, sc.min ) => int note; //even notes of the scale
  std.mtof( 3*12 + 10 + note ) => inst.freq; //set note
  inst.noteOn( 0.5 ); //play a note

  T => now; //compute one beat of audio
}
```

Using this strategy, we can get shreds sporked at different times to line up on (or against) the beat. Since this is a relatively cheap operation, we can even do this quite frequently, for example if we perform random or irregular operations on time.

B. (division of **now**):

```
// time / dur yields number
(now / T) $ int => int n; //get the integer part
math.fmod( (now / T), 1.0 ) => float f; //fraction part
```

We can use A to get the orchestra members to act precisely in concert. We'll use B to help them keep count, and to provide a wider shared context between them. Insert this above the loop in the previous example.

```
(now / T) $ int => int n; //guess the beat

for (n=>int i; ; i++) { //loop knows the beat
```

Now each shred has access to a globally aware beat. We can also divide this uniformly into beats, measures, and sections:

```
i%nbeats => int b; //which beat of nbeats
i/nbeats%nmeas => int m; //which measure
i/nbeats/nmeas => int s; //which section or rep
```

This will provide the groundwork for that foundation of pop music, the chord progression.

1.5 Timing Shortcuts

We can add these basic timing operations to a class for brevity's sake: tg.ck

```
fun void sync() { //sync to beat
  beat - (now % beat) => now;
}

fun void sync(dur T) { //sync to arbitrary
  T - (now % T) => now;
}

//get global beat
fun int guess() {
  return (now / beat) $ int;
}
```

Some instance data for the particular grid and a setter:

```
dur beat;
dur meas;
dur sect;

int nbeat;
int nmeas;

fun void set(dur mybeat, int nb, int nm) {
  mybeat => beat;
  nb => nbeat;
  beat*nbeat => meas;
  nm => nmeas;
  meas*nmeas => sect;
}
```

And some shortcuts for the mod rhythms:

```
fun int b(int r) {
  return (r%nbeat);
}

fun int m(int r) {
  return (r/nbeat%nmeas);
}

fun int s(int r) {
  return (r/nbeat/nmeas);
}
```

That allows us to systematize our timing operations:

```
TimeGrid tg;

//140 bpm 8th notes, 8 per measure, 10 meas / section
tg.set(1::minute/140/2, 8, 10);

tg.sync(); //sync to beat

while( true ) {
  tg.guess() => int i; //get the global beat

  //other stuff

  tg.beat => now; //advance time one beat
}
```

1.6 Other Timing Approaches

Our approach uses no truly global data-it merely derives context from the already global **now**. Another approach

uses Events in Global data to synchronize separate shreds.

We first define a new class with a static reference to an Event object (or an array of them). Next, we spork a shred that initializes the events, along with an infinite loop that fires the events at intervals of your choosing. Finally, each musical shred chucks the event(s) to now in order to lock with the beat.

1.7 Chords and Arpeggios

We've introduced most of the techniques we'll need. Let's start with a bass line. Begin a file with some declarations:

```
Scale sc; TimeGrid tg;

StifKarp inst => dac; //plucked string

tg.set( 1::minute/140/2, 8, 8 ); //140::bpm, 8 beats, 8 measures
```

The simplest bass lines play just the bassnote of the current chord. This chord progression will be diatonic, meaning that we'll choose the bass notes (and all other chord tones) strictly from notes in a single scale. Add a data sequence for the progression:

```
[0, 3, 4, 1, 5, 4, 3, 3] @=> int bass[];
```

Go ahead and sync() so all shreds will start on the nearest beat:

```
tg.sync();
```

Now we can use the global beat indicator to derive the current measure. We can do this once if we perform only very-time-regular operations, but we can also just do it whenever:

```
while( true ) { //infinite loop
  tg.guess() => int i; //global beat indicator
  tg.m(i) => int m; //measure indicator

  sc.scale( bass[m], sc.maj ) => int note; //select the bassnote
  std.mtof( 3*12 + 7 + note ) => inst.freq; //7 semitones from C is G

  inst.noteOn( 0.7 ); //play a note

  tg.beat => now; //advance by one beat
}
```

Play this code. As you hear, it repeats the bass note for each beat and then changes with the measures.

Here's the trick. Western harmony is really simple-most chords are based on triads. To play a triad, just play the bass note (the 1st), two scale tones up (the 3rd), and two more scale tones up (the 5th). Since we're using zero based indices, these become 0, 2, and 4. If we wish to go to higher octaves, we should choose:

[0,2,4, 7,9,11, 14,16,18, ...]

This should look familiar. In fact, it's the same kind of sequence we generated to filter scale tones. Keep in mind we might want to extend arbitrary sets of scale tones, either for certain chords (like 7th chords: [0,2,4,6]), or for generating melodies. Typically a melody that is 'in a key' includes mostly chord tones (1st, 3rd, and 5th) and other non-chord or passing tones. This function will look almost exactly like scale: [1]

```
//oct is the size of an octave in scale degrees, usually 7
fun int arp(int a, int oct, int deg[]) {
  deg.cap() => int n; //number of arp degrees
  a/n => int o; //number of octaves up/down
  a%n => a; //after subtracting the octaves

  if ( a<0 ) { //the border case
    a + n => a;
    o - 1 => o;
  }

  return o*oct + deg[a];
}
```

Let's apply this to our bassline code to produce a variation.

```
arp( b/2, 7, [0,2,4] ) => int a; //the arp/melodic contour
sc.scale( a+bass[m], sc.maj ) => int note; //select the bassnote
std.mtof( 3*12 + 7 + note ) => inst.freq; //7 semitones from C is G
```

The first line generates a simple contour (b/2: [0,0,1,1,2,2,3,3]) and converts it into (diatonic) scale tones with arp(). The second adds the measure-dependent bass and selects scale tones from the chromatic scale (in C). The last adds a 3-octave offset and 7 semitones to transpose the key (to G).

Arp functionality is also provided in the Scale class. scale.ck

1.8 Arp Variations

When the bass moves, there is a jump in the melody corresponding to the number of scale tones difference. If you don't want jumps in it (which are find for vamps, but maybe jarring for a melody) you have to find a way to subtract out the jumps. Scale.iarp is an attempt at doing that. The implementation right now is a little hackish, but it should improve.

1.9 Extras

fade.ck

1.10 More

... standard variation techniques ...

Next Chapter

Footnotes:

[1]This function is called arp for Arpeggiation-a term denoting playing the tones of a chord separately. If we add some passing tones into the sequence, we can model melodies as well.

File translated from TEX by TTH, version 3.78.
On 26 Oct 2007, 18:41.