ChucK
A 'Strongly Timed' Language for Music Programming

Tillman Weyde
City University London
Overview

- Music Programming Languages
- Design Goals
- Syntax
- Time and Events
- Audio Programming
- On the fly programming
- Examples/Applications
Music Programming Languages

• Support for Composers and Performers
• Interests of Musicians
  – Sound production
  – Sound control
  – Writing music
  – Having music “written” for them
  – Interacting with music
History

• MUSIC N
  – For IBM Mainframes
  – Contains a sequence of sound production instructions producing digital data, then converted to an analog
  – UnitGenerator (UGen): signal producer
  – Patch: a set-up of UGens
  – Orchestra: set of Patches
  – Score (instructions to send to patches)
IBM Mainframe Operation
Modern Successors

• CSound/Super Collider follow the MUSIC N Paradigm
Max/MSP

- Graphical interface following analog synthesizer paradigm, showing data and control flow
Sequencer Software

- Follows the tape machine paradigm, focused on data editing.
ChucK

- Designed by Ge Wang in Princeton (now Stanford)
- Philosophy
  - Programming Language = User Interface
  - Code = Musical Instrument
- Goals
  - Natural expression of musical ideas
  - Flexible control of time
  - Concurrency
Design Principles

• “Strongly Timed”: give easy-to-use and powerful control over timing
• Provide flexibility of a full programming language
• Support live interaction with the runtime system
Language Properties

- Imperative
- Strongly typed
- Runs in a Virtual Machine
- Has functions, objects and classes
Syntax

• ChucK operator '=>'
• Used for assignment ...
  \[42 => \text{int } x;\]
• ... and to create audio connections
  \[
  \text{input => output;}
  \text{input => filter => output;}
  \]
• ... and to supply arguments to a function
  \[
  (\text{a,b}) => \text{Math.min => Math.fabs => x;}
  \text{Math.fabs(Math.min(a,b)) => x}
  \]
Syntax (2)

- Functions
  
  ```chuck
  fun int square(int x) {
      return x*x;
  }
  ```
Music and Time

• Normally programs run as fast as possible:
  `play("note a");`
  `play("note b");`

• In music you often want this
  `play("note a");`
  `wait("one bar");`
  `play("note b");`
Music and Time (2)

• In Java you can use sleep
  
  play("note a");
  Thread.sleep(500);
  play("note b");

• Problems
  – No guaranteed sleep duration
  – No synchronisation of multiple threads
  – Duration of computation
Music and Time (2)

- Alternatively you can use a timer
  
  ```java
  TimerTask task = new TimerTask(){
      public void run(){play("note a");}
  }
  timer.schedule(task, new Date(...));
  ```

- Problems
  - No guaranteed execution time
  - Code distributed over TimerTask objects.
Controlling Time in ChucK

- 'Synchronous' time model
  - based on audio sample clock
- Language elements for time control
- Cooperative multitasking
The “now” variable

- Now represents the current (logical) time
- “chucking” into now means suspending until the new time is reached

```chuck
// print current time
<<< "Hello," , now >>>;
// advance current time
1 :: second => now;
// print new current time
<<< "Time!" , now >>>;
```
The 'shred'

- One thread (like e.g. Java) of operation is called a 'shred' in ChucK
- Starting a program in ChucK is starting a shred
- There can be many shreds in parallel
What Belongs to a Process

• Process state: e.g. new, ready, running, etc.
• Program counter: indicates the next instruction to be executed by the process
• CPU registers: accumulators, stack pointers
• CPU scheduling information: includes process priority, etc.
• Memory-management information: value of the base and limit registers
• Accounting information: includes process #, amount of CPU used, etc.
• I/O status information: includes the I/O allocated to the process, open files, ...
Threads/Sherds are Lightweight Processes

- Threads have individual
  - program counter
  - register set
  - stack space
- A thread shares with its peer threads its
  - code section
  - data section
  - operating-system resources
Thread/'shred' Lifecycle

- States of thread/process in execution run through several states:

```
x => now;
me.yield()
```

```
x => now;
```
'spork'

- Shreds can be started from within the program using the 'spork' command (see 'fork' in Unix)
  
  ```chuck
  fun void do_it(){ ... }
  spork ~ do_it() @=> Shred child;
  ```

- The child shred shares the address space with the parent

- The child shred is stopped when the parent stops
The 'shreduler'

- The scheduler/shreduler manages the execution of threads/shreds
Scheduling Strategy

- Co-operative First-Come, First-Served (FCFS) scheduling
  - No priorities
  - No interruption
  - Shreds are guaranteed to execute at the given logical time (synchronous reactive system)
  - If processing power is not sufficient is not matched, sequence of processing is not changed
Time and Events

- Events allow to wait for something other than a fixed point in time
  - User input
  - Thread signals
  - MIDI messages
  - ...

Event Example

- Processing MIDI Input

```chuck
MidiIn min;
MidiMsg msg;
// open midi receiver, exit on fail
if( !min.open(0) )
    me.exit(); // 'me' is the current shred
while( min => now ){
    // wait on midi event
    while( min.recv( msg ) ){
        // print content
        <<< msg.data1, msg.data2, msg.data3 >>>;
    }
}
```
Audio Programming

- Audio building blocks (Unit Generators, UGens) are provided with ChucK
- Need to be 'patched' together

```chucK
SinOsc osc; // sine wave
440 => osc.freq; // set to 'a'
osc => dac; // route signal to // digital-analog-converter // (= audio output)
500::ms => now; // play 500ms
```
On-the-Fly Programming

- Shreds can be created and removed interactively
- The compiler is integrated in the Virtual Machine, no extra compiling necessary
- MiniAudicle IDE useful, but command line works too
Use Cases

- Composition
- Audio Processing Development
- Live Performance