Conductor Follower Documentation

Sakari Bergen

December 13, 2012

Contents

1 Overview **2**

2 Setting up the Environment **2**
2.1 The Synthesis Engine 2
2.2 The VST Host 2
2.3 OpenNI 3

3 Running Conductor Follower **3**
3.1 Setting up the Plugin Host and Synthesis Engine 3
3.2 The Graphical User Interface 3
3.3 The Gestural Interface 5
3.4 Problems Loading Presets or Running Multiple Instances . . . 5
3.5 Vienna Symphonic Library Recommendations 5
3.6 A Note on MIDI Files 6

4 The Configuration Files **6**
4.1 The Score Definition File 7
4.2 The Instrument Definition File 7
4.3 The Beat Pattern Definition File 8

5 Compiling Conductor Follower **8**
5.1 Requirements 9
5.2 Building on Visual Studio 9
5.3 Building on Other Compilers 9

6 Adding Support for Different Interfaces **9**
6.1 The Plugin Interface 10
6.2 The Score Format 10
6.3 The Motion Tracking Interface 10

1 Overview

Conductor Follower is a VST plugin that follows conducting gestures using a *Kinect*[1], and processes MIDI events based on the gestural input. The system requires a score to be provided as a MIDI file, and additionally, a number of configuration files.

2 Setting up the Environment

In order to work, Conductor Follower requires:

- A sample-based synthesis engine
- A VST host
- A working OpenNI[2] installation
- Some MIDI scores
- Properly set up configuration files:
  - A score definition for each MIDI score
  - An instrument definition file for the synthesis environment
  - A beat pattern definition file

The synthesis engine, VST host and OpenNI are discussed in this section, while the MIDI scores and configuration files are discussed in the next.

2.1 The Synthesis Engine

During the development of Conductor Follower, only the *Vienna Symphonic Library* Suite (VSL)[3] was used. However, any sample-based VST-compatible synthesis engine that supports switching patches via keyswitch events should work. In order to make the system work properly, an instrument definition file must be provided. The available patches per instrument, and the respective keyswitches are defined in this file. Instrument definition files are discussed in detail in section 4.2.

2.2 The VST Host

As Conductor Follower is a plugin, it obviously needs a host to run. Unfortunately VSL does not support running MIDI-generating VST plugins directly,

[1]Any OpenNI compatible device may be used, but if it runs at anything but 30 Hz, you will need to make some modifications to the system.

[2]`http://openni.org/`

[3]`http://www.vsl.co.at/`

so with VSL, a separate VST host is required. Other synthesis engines might support hosting such plugins, and in such a case, it is recommended to host the plugin directly in the synthesis engine. In theory, any VST host should work with Conductor Follower.

During development, the Juce[4] audio plugin host was used. The source code for the audio plugin host can be found in the `extras/audio plugin host` directory of the Juce source package. Since the host can be self-built and run in a debugger, it makes debugging the plugin a lot easier.

2.3 OpenNI

In order to work, Conductor Follower requires a functioning install of OpenNI. If you happen to receive Conductor Follower in binary form, a compatible version of OpenNI should be provided with it. Otherwise, you will need to install OpenNI as part of fulfilling the build requirements.

3 Running Conductor Follower

Running Conductor Follower requires some setting up, after which the graphical and gestural user interfaces are used to control the system.

3.1 Setting up the Plugin Host and Synthesis Engine

In order to get the system to work, you need to load the Conductor Follower plugin and your synthesis engine in the host. The plugin has an audio input and output, which were added only because some hosts did not support plugins with zero audio outputs and inputs. However, these ports do not do anything and may be left unconnected. The only port which needs to be connected, is the is the MIDI output port, which should be connected to the synthesis engine.

3.2 The Graphical User Interface

The GUI of the plugin contains a set of adjustments and status indicators. The basic set initially visible contains the following items:

State	The current state: "Waiting for wave gesture", "Waiting for start gesture", "Rolling", or "Stopped". These should be self explanatory.

[4] `http://www.rawmaterialsoftware.com/juce.php`

Score definition file	File selector for the score definition, as discussed in section 4.1.
Restart score	Pressing this button will cause the score to be rewound, and the system will wait for either the start or wave gesture, depending on its current state.
Listen to score	Plays back the score from the beginning. Press *Restart score* to start conducting again (from the beginning).
Show more options	Shows the options described below.

When pressing the *Show more options* button, a set of status indicators and options appear. The status indicators show relative values, the bar ranging from minimum value to maximum value. The status indicator are based on the tempo and movement features, as described in the thesis:

Speed	The current tempo relative to the transcribed tempo.
Motion length	The windowed trajectory length of the hand.
Velocity dynamic range	The windowed standard deviation of the absolute velocity.
Jerk peak	The filtered and windowed peak of jerk (derivative of acceleration).

The extra options presented are:

Expression sensitivity	The amount of expressive features applied to the score. At 0 the expressive features of conducting have zero effect on the output, and the patch selection is solely based on the note lengths and velocities in the score. At 100, the maximum amount of expression is applied.
Catchup fraction	The percentage of position difference to be corrected at each detected beat. Lower values react more slowly, but are more musical.
Tempo filter cutoff time	The time over which tempo is averaged over. Lower values will follow tempo changes faster, while they will also be more sensitive to misclassifications. Larger values are more stable, but will follow tempo changes slower, and might cause misclassifications during fast tempo changes.
Beat anomaly penalty	A relative value for how much penalty is given for extra or missing beats in beat pattern classification. A high value will follow clear conducting more reliably, but if the motion tracking component detects

	extra beats, or skips beats, large values will cause classification errors.
Tempo change following	The percentage of tempo changes to apply when a score contains a tempo change.

3.3 The Gestural Interface

The gestural interface only follows one hand, usually the right hand. The most critical gestures to learn are the wave gesture, the start gesture, and the beat gestures. The rest of the gestural interface can be discovered via experimenting.

The Wave gesture is required for the OpenNI hand tracking to find your hand. Simply wave to the motion tracker as if saying bye-bye, until you see a red trace following your hand in the visualization. The wave has to have a large enough amplitude and speed.

The start gesture requires holding your hand steady for a while, and after that doing an downward and upward motion. That is steady – down – up. The initial tempo is taken from the length of this gesture.

After the piece has started, the tempo is beat with the vertical movement of the hand, movement mostly horizontal movement, and very little vertical movement will not work as expected.

3.4 Problems Loading Presets or Running Multiple Instances

Due to thread safety issues in OpenNI, running multiple instances of Conductor Follower in the same process is not stable. This includes the case of loading a VST host preset including Conductor Follower, if Conductor Follower is already loaded in the host. You can, however, run two different VST hosts, and Conductor Follower should work fine in both, due to the hosts being different processes. The workaround to the preset problem is to close Conductor Follower or the host each time a preset is loaded. Based on every crash observed so far, this is solely a OpenNI problem, and might be fixed just by using a later version of OpenNI.

3.5 Vienna Symphonic Library Recommendations

In order to save time when loading samples, it is recommended to start one or more instances of *Vienna Ensemble Server*. Once the samples are loaded in the server instance, loading them in the VSL plugin is a multitude faster.

3.6 A Note on MIDI Files

The quality of the MIDI files used as scores carries a huge importance on the functional quality of the system. In addition to overall quality, it is especially important to note the following points:

- The score must be in proper tempo and meter. That is, notes must be properly aligned along bars and beats. Many MIDI files lack proper tempo and time signature information.
- There must be one MIDI channel per instrument. One common problem is putting all the strings on one ”string ensemble” track.
- Tempo changes can be problematic. If the conductor does not follow the tempo changes exactly, the changes may cause issues.

4 The Configuration Files

The configuration files all use a JSON-like format. The format looks like this:

```
object_name {
  // This is a one line comment
  property_name: "string value",
  another_property: 0.85,
  a_list_property: [
   list_element {
     property: 1
   },
   /* This is a
      multi-line
      comment
   */
   list_element {
     property: 2
   }
  ]
}
```

The format should be pretty self explanatory. Whitespace does not carry meaning, and most delimiting commas are optional.

Score positions values are in the format `<bar>|<beat>`, e.g. `5|1.5`.

4.1 The Score Definition File

The score definition file includes the following properties

`name`	A human readable name for the score.
`midi_file`	The path of the related midi file.
`instrument_file`	The path of the related instrument definition file.
`beat_pattern_file`	The path of the related beat pattern file.
`tracks`	A list of track to instrument mappings.
`events`	A list of score events.

The `tracks` list contains `track` objects, which have the following properties:

`name`	A human readable name for the track.
`instrument`	The name of the instrument, which must match that in the instrument definition file.

The `events` list currently only supports `tempo_sensitivity` objects, which have the following properties:

`position`	The position of the event (`<bar>|<beat>`).
`instrument`	The tempo sensitivity, $[0.0, 1.0]$.

4.2 The Instrument Definition File

The instrument definition file contains a top-level unnamed list of `instrument` objects. Each instrument contains the following properties:

`name`	A human readable name for the instrument
`shortest_note_threshold`	The length in seconds, that equals a relative length of zero.
`longest_note_threshold`	The length in seconds, that equals a relative length of one.
`channels`	A list of MIDI channels (integers) this instrument is available on.
`patches`	A list of `patch` objects, describing the available patches.

Each `patch` object contains the following properties:

`name`	The name of the instrument (must be unique).
`keyswitch`	The key for switching to this patch.
`length`	The relative maximum length of notes, $[0.0, 1.0]$.

`attack`	The amount of attack in the patch, $[0.0, 1.0]$.
`weight`	The amount of weight in the patch, $[0.0, 1.0]$.

Keyswitches are in the format `<note><octave>`, where note is the capital name of the note, e.g. `C#`, only including neutral and sharp notes (no flats). The octave is the octave as an integer. Thus, valid examples of keyswitch values are `C1`, `C#1` and `C12`. Invalid values include `1C`, `Db1` and `c12`.

4.3 The Beat Pattern Definition File

The beat pattern definition file contains a top-level unnamed list of `beat_pattern` objects. Each beat pattern contains the following properties:

`meter`	The time signature this pattern is associated with.
`beats`	A list of beats.

The meter is in the format `<count>/<division>`, where both values are integers. Valid values include e.g. `4/4` and `6/8`. The `beats` list contains `beat` objects, each including a single property `time`, which indicates the time of the beat as a floating point value of beats into the bar. A period is used as the decimal point. Valid values for `time` are thus e.g. `1` and `1.5` .

5 Compiling Conductor Follower

The source code is currently available at `http://github.com/sbergen/ConductorFollower`. In addition, you will need to install a number of libraries. Ready project files are available only for *Visual Studio 2010*, but the code should be trivially portable to other systems also.

The project is divided into the following sub-projects:

cf	Common Utilities
cfTests	Unit tests for the above
CFPlugin	The VST Plugin
Data	Configuration file parsers
DataTests	Unit tests for the above
MotionTracker	The motion tracking component
MotionTrackerTests	Unit tests for the above
ScoreFollower	Score following component
ScoreFollowerTests	Unit tests for the above
PatchMapper	Patch mapping functionality
Visualizer	Visualizer component used in the GUI

5.1 Requirements

The following libraries are required for building Conductor Follower:

Boost:	`http://www.boost.org/` Version 1.51 has been tested.
OpenNI:	`http://openni.org/` Version 1.5.2.23 has been tested.
Juce:	`http://www.rawmaterialsoftware.com/juce.php` Version 2.0 has been tested.

5.2 Building on Visual Studio

The git repository includes project files for VS2010. Currently only the 64-bit build targets are properly set up, but copying over the configuration to the 32-bit targets should be rather trivial. In order for the project files to work, the following environment variables need to be set:

BOOST_INCLUDE	The path to the boost headers.
BOOST_LIB	The path to the boost binary libraries.
OPEN_NI_INCLUDE	The path to the OpenNI headers.
OPEN_NI_LIB64	The path to the OpenNI 64-bit binary libraries.

The *CFPlugin* project is the plugin, and it should automatically build all its dependencies. However, it is recommended to also build all the unit test projects, which will be automatically run after building.

5.3 Building on Other Compilers

No support for other compilers is currently present, but since the code was written bearing compatibility in mind, it should be trivial to port the project to any C++11 compatible compiler. The required libraries should support at least OS X and Linux in addition to Windows. It is recommended to use Visual Studio to look at the project structure if build scripts for other platforms are to be made, but the structure should be easy to figure out via trial and error even without using VS.

6 Adding Support for Different Interfaces

Some parts of the system were designed to be easily modifiable to work with alternative libraries. These include the plugin interface (currently VST), the score format (currently MIDI), and the motion tracking interface (currently OpenNI). All class names mentioned in this section omit the common namespace *cf*.

6.1 The Plugin Interface

Juce already supports at least AudioUnit, RTAS, NPAPI, ActiveX and LADSPA. Porting the code to work with these formats should be easy. However, if something not supported by Juce is needed, the porting should be rather easy even then.

The main functionality of the plugin is platform agnostic, and is used via the `ScoreFollower::Follower` interface. The actual usage requires only about 20 lines of code (currently in the function `CfpluginAudioProcessor::processBlock()` in the *CFPlugin* project).

The UI for the plugin operates on two classes: `ScoreFollower::FollowerOptions` and `ScoreFollower::FollowerStatus`. The current implementation uses some C++ template metaprogramming tricks provided by *Boost.Fusion* to automatically generate the UI during compile time, but making a UI that manually uses these structures should be trivial – just take a look at how the structures are used in the *ScoreFollower* project.

6.2 The Score Format

The Following interfaces are used by the *ScoreFollower* to read the score and operate on the score events:

- `ScoreFollower::ScoreReader`
- `ScoreFollower::TrackReader`
- `ScoreFollower::ScoreEvent`

These should all be rather trivial to implement on any other score format, given a good library that supports operating on the data.

6.3 The Motion Tracking Interface

The motion tracking interface, which currently only tacks one hand, is defined in `MotionTracker::HandTracker`. The *OpenNI* implementation is respectively in `MotionTracker::OpenNIHandTracker`, and may be used as an example implementation.

The Kinect depth sensor runs at 30 Hz. If some other device with a differing frame rate were to be used, some parameters would need adjusting. Some of the frame rate dependent variables are in the class `MotionTracker::`

`FrameRateDependent`, but several variables got also left out. These variables should be tracked down and added to the aforementioned class, and some logic should be made for selecting the right set of variables.