A Csound Tutorial

Michael Gogins
michael.gogins@gmail.com

December 20, 2009

Contents

List of Figures

1. Introduction

In the words of its author, Barry Vercoe, Csound [1] is a "sound processing language." Technically speaking, Csound is a general-purpose, user-programmable software synthesis system (SWSS). Like most SWSS, Csound uses Max Mathews' original 1957 unit generator design [2]. However, Csound was the first SWSS to be written in the C programming language [3]. Being written in C, which is the most efficient and most portable high-level language, and also very widely used, has ensured Csound's survival and growth.

Vercoe wrote Csound at the Massachusetts Institute of Technology in 1984. Ever since then, Csound has received contributions from researchers, programmers, and musicians all over the world. Csound runs on Unix, Linux, Windows, the Macintosh, and other operating systems. Csound can be extended by writing plugin unit generators, and Csound itself runs as a VST plugin. Csound can be programmed in C, C++, Java, Lisp, Lua, and Python. Csound is taught in a number of leading universities and conservatories. Books have been written on how to use it [4, 5, 6]. Csound can be compiled to use double-precision floating point audio samples for the highest sound quality.

In short, Csound must be considered one of the most powerful musical instruments ever created.

Csound is, perhaps, harder to use than such competing programmable synthesizers as SuperCollider [7], Max [8], or Reaktor [9]. One difficulty is that Csound was written a generation ago as a Unix application, and is controlled by dozens of arcane command-line options (although, precisely because it *is* older, Csound runs faster and has more unit generators). Another difficulty is that Csound lacks some convenient features of other high-level programming languages.

Still, once you learn a few things, Csound is not really so hard to use. The sound processing language turns out to be simple, the documentation is good, Csound always tries to tell you what it is doing (or why it is not doing what you told it)... and the power begins to sing.

The purpose of this tutorial is to teach those often neglected first few things. There are three introductory sections, one each for Windows (Section 2.1), Linux (Section 2.2), and Apple computers (Section 2.3), that lead you, step by step, through obtaining, installing, configuring, and running Csound (also see the Csound Reference Manual [10]). Then follow chapters on writing your own orchestras and scores (Chapter 3), using CsoundVST as a VST plugin in a studio sequencer (Chapter 4), and writing Python scripts to do algorithmic composition using the Csound application programming interface (API) (Chapter 5). Finally, there is a list of software required to use the extra features of Csound (Appendix A), a list of other helper applications and languages for Csound (Appendix B), and some advice on how to achieve good sound quality with Csound (Appendix C).

2. Getting Started

This chapter contains the same information — how to obtain, install, configure, and run Csound — repeated for each of the main personal computer operating systems in use today: Windows, Linux, and Macintosh OS X.

2.1. On Windows

2.1.1. Obtaining Csound

Windows installers for Csound can be obtained from at least two Web sites: SourceForge, at `http://csound.sourceforge.net`; and my blog, at `http://michael-gogins.com/?page_id=59`.

SourceForge hosts the general Csound project, including source code and binary packages for a variety of operating systems. The Windows installers at SourceForge, however, are not permitted to contain CsoundVST or the vst4cs opcodes because they depend on the proprietary Steinberg VST STK, which is not compatible with free or open source software licenses. To download from SourceForge, use your Web browser to go to the **Main Download page** link. On the download page, click on the link to the **csound5** package. You will see a list of releases. At the time this was written, the most recent downloadable version of Csound for Windows is 5.11. You will see a **csound5.11** link on the page. Click on that, and it will expand to show `Csound5.11-win32-d.exe` and `Csound5.11-win32-f.exe`. Both of these programs are Windows installers for Csound 5.11 (Figure 2.1). For later versions and releases, substitute the actual version number for 5.11 in the links and filenames.

`Csound5.11-win32-f.exe` installs a version of Csound that has been compiled to use 32-bit floating point numbers internally to represent audio samples. As a result, it runs about 15% faster than `Csound5.11-win32-d.exe`. `Csound5.11-win32-d.exe` installs a more complete version of Csound, which has been compiled to use 64-bit floating point numbers for audio samples, so that it is a slightly more accurate synthesizer than `Csound5.11-win32-f.exe` [11].

If you are putting on live shows using Csound with complex instruments, and need extra efficiency, download `Csound5.11-win32-f.exe`. Otherwise, you will be better off with `Csound5.11-win32-d.exe`. But if you want to use CsoundVST or vst4cs, download the installer from `http://michael-gogins.com/?page_id=59`. The rest of this tutorial, which covers CsoundVST, assumes you have done this.

2.1.2. Installing Csound

Csound comes with a number of extra features that require other software to work. These extras and their requirements are listed in Appendix A. *Please note: if you*

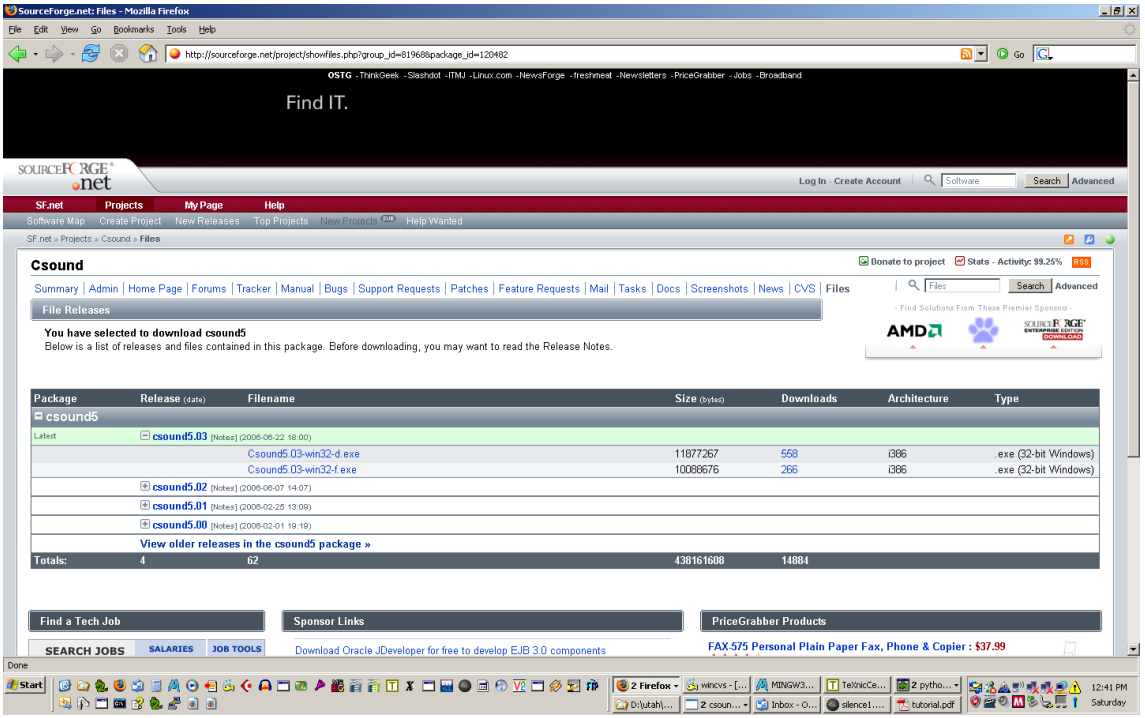

Figure 2.1.: **Download Page**

do not *install any of this other software, the standard features of Csound will still work!*

To install Csound, simply run the installer. It will display a dialog box (Figure 2.2).

Figure 2.2.: **Windows Installer**

Click on the **Next >** button to proceed. You should now see the Csound license agreement (Figure 2.3). You must click on the **I Agree** button to indicate your acceptance of the Csound license before you can install Csound.

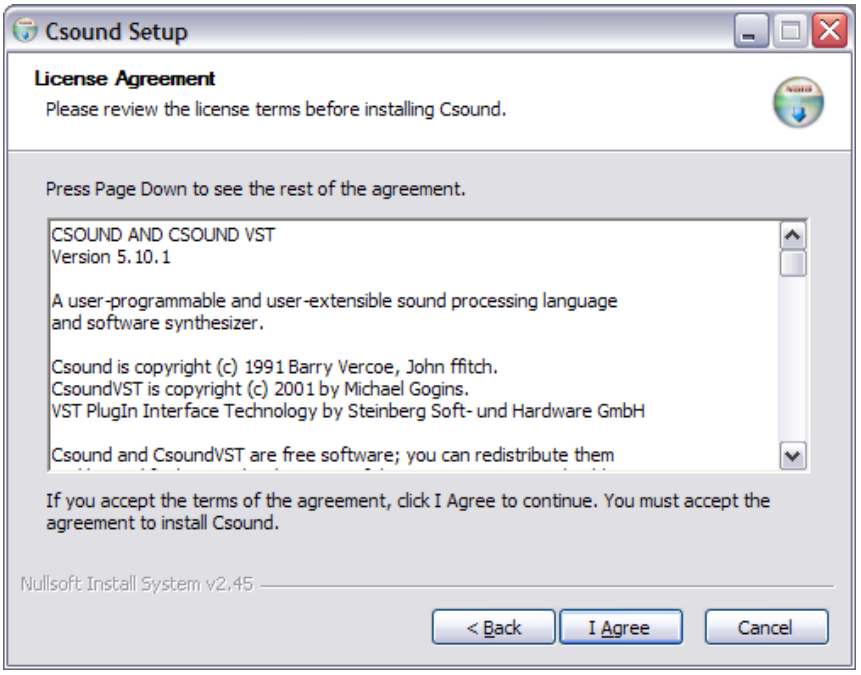

Figure 2.3.: **Csound License**

Tell the installer where to put Csound. Although the default location is the standard Windows `Program Files` directory, Csound may actually work better if you install it in a directory without any spaces in the pathname, such as `C:\Csound` (Figure 2.4).

Tell Csound where to put the Windows **Start Menu** folder for Csound. You can skip this step if you want, but I recommend that you accept the default location (Figure 2.9).

Next, the installer gives you a choice of components to install (see Figure 2.6. To simply run Csound, you need only the "core" installation. However, to complete this tutorial including the section on scripting Csound, you should also select the documentation and the Python features (which require that you first install Python 2.6). A reasonable set of components for completing all sections of this tutorial would include:

- Csound
  - Csound engine, opcodes, and drivers
  - Utilities
  - Documentation
    - Csound Reference Manual
    - A Csound Tutorial

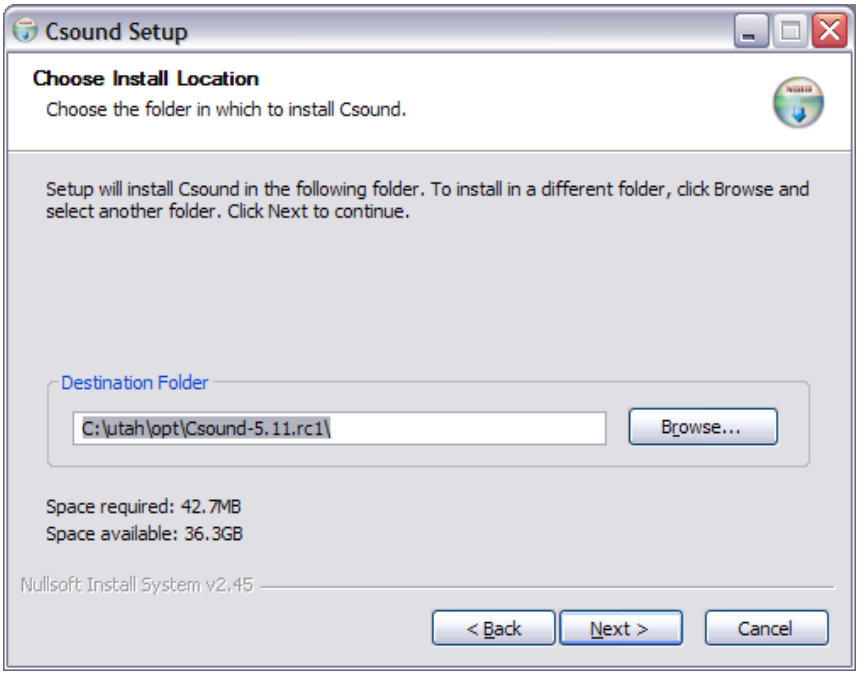

Figure 2.4.: **Csound Location**

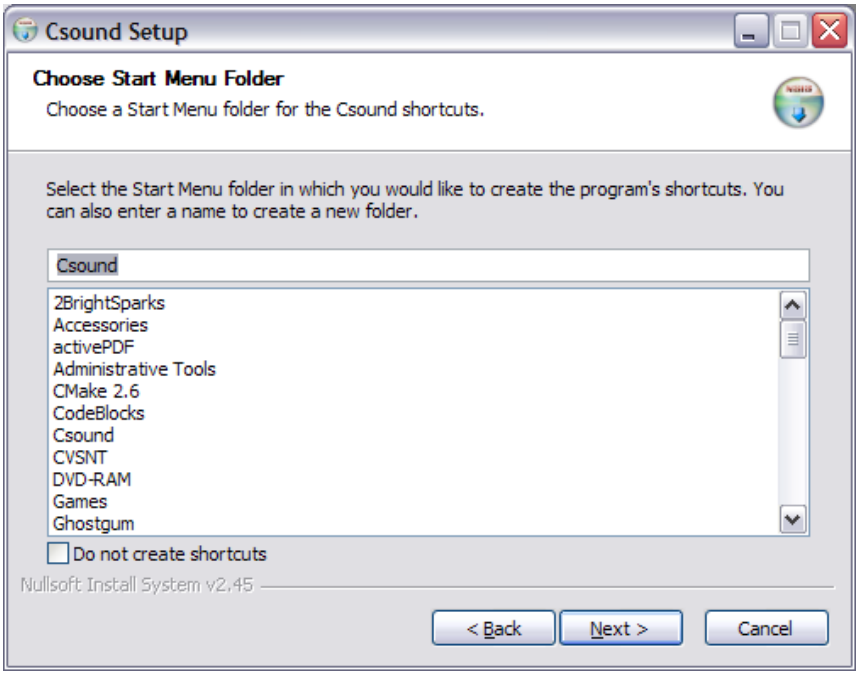

Figure 2.5.: **Csound Menu Location**

- Front ends
  - QuteCsound
  - CsoundVST
- Interfaces
  - Python
    - Python opcodes
    - csnd: Python interface to Csound
    - CsoundAC: Python interface to Csound algorithmic composition

If you want to use Lua for scripting Csound, or use Csound as Java library, or with Lisp, you should select the relevant features. It does no harm to select a complete installation.

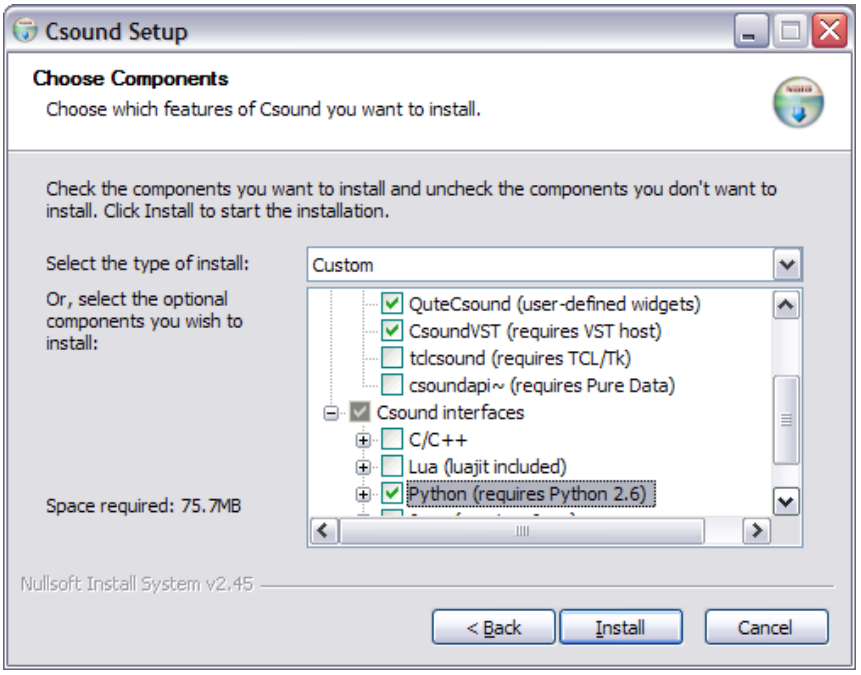

Figure 2.6.: **Csound Components**

Click on the **Install** button. The installer will now unpack and install Csound in your selected location (Figure 2.7). When the installer has finished, you should see the message shown in Figure 2.8. Open the Windows **Start Menu**, where you should find a **Csound** submenu containing various Csound programs and documentation (Figure 2.5).

Configuring and Running Csound on the Command Line

Csound is capable of state-of-the-art audio quality, equal to or better than the best recording gear. For more discussion of how to achieve this quality, see Appendix C. The short piece you are about to render has been modified to render at high

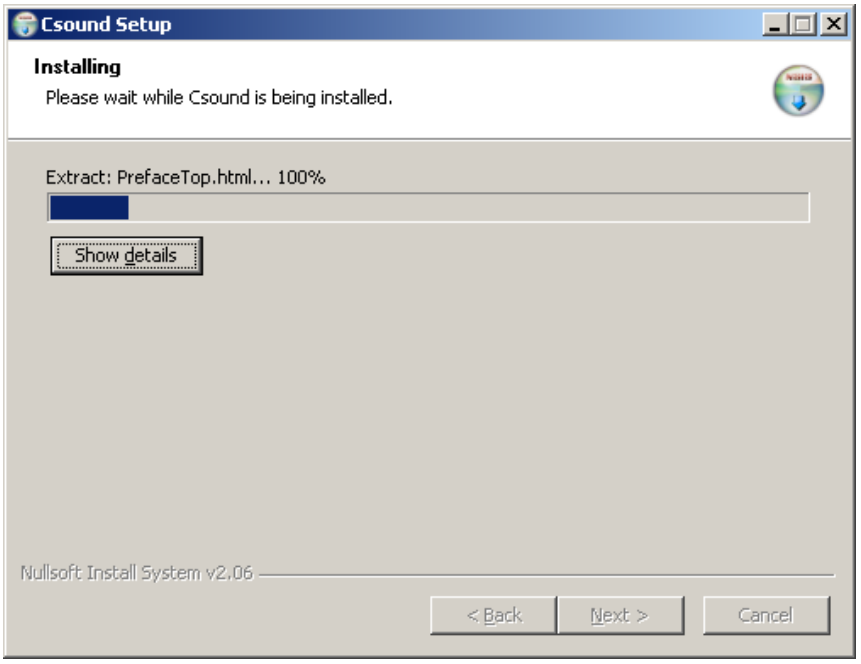

Figure 2.7.: **Csound Installing**

Figure 2.8.: **Installation Completed**

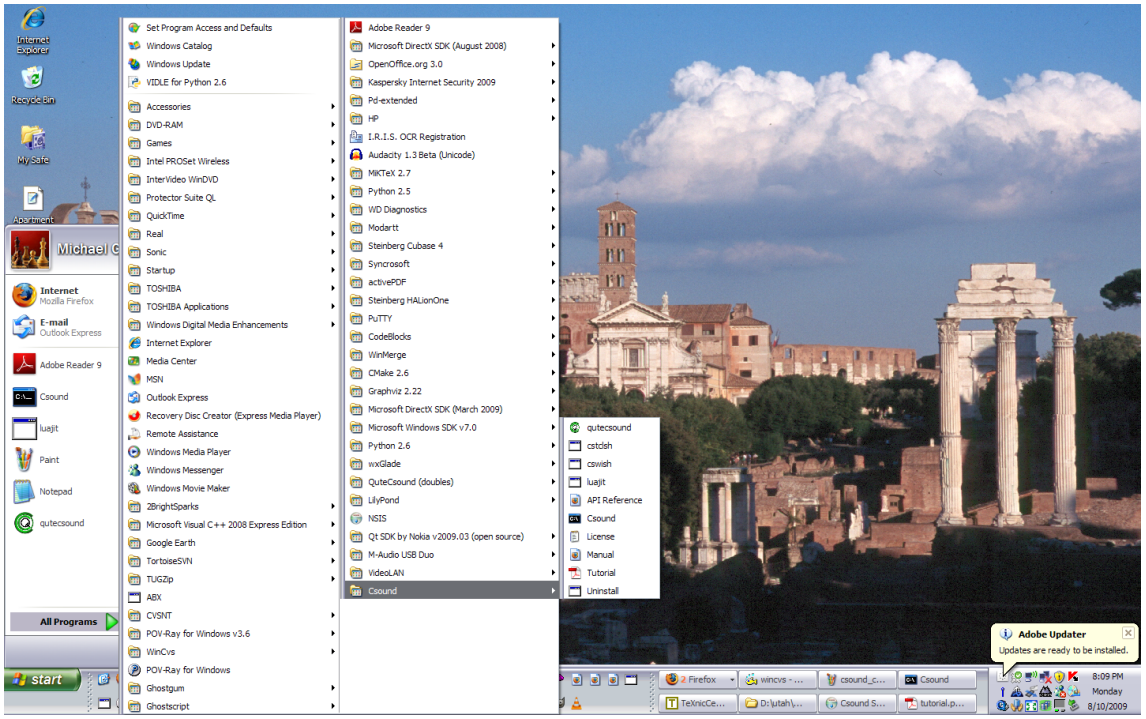

Figure 2.9.: **Csound Start Menu**

resolution, so it should serve as something of a demonstration of what Csound can do.

As you may have gathered, there *many* ways of running Csound. The two ways we are concerned with here are the original way, as a command-line program,[1] and as a GUI program. We will run the piece both ways.

This section assumes that you have installed Csound in the `C:\Csound` directory. In the following, replace this with your actual installation directory. Using a text editor (not a word processor!)[2], take a look at the `C:\Csound\.csoundrc` file. This file provides default command-line options that take effect each time you run Csound, unless you provide another value for the option. As installed, it reads:

[1] What is the *command line*? Every operating system has one. It is a "console window" that has a prompt where the user can type in text commands. On Windows, you can open the console by going to the **Start** menu, selecting the **Run** item, typing `cmd` in the **Open:** field, and clicking the **OK** button. When you see the prompt, type `dir` and press the **ENTER** key as an example of executing a command.

[2] Do *not* use the default text editor on Windows, which is Notepad! Csound files typically have Unix line endings (linefeed only), whereas Notepad only works properly with Windows line endings (linefeed plus carriage return). I recommend that you install and use SciTE [12], a general-purpose text editor for which you can get Csound orchestra language syntax coloring. You can obtain Csound API and orchestra language syntax coloring properties from `http://solipse.free.fr/Api_&_csound.properties/csound.api` and `http://solipse.free.fr/Api_&_csound.properties/csound.properties`, respectively. Then in your global options file, around line 539 add a new line `Csound|orc||\` and around line 611 add a new line `import csound`. Line numbers are very approximate, but you should see similar statements for other languages in the correct locations. You can even run Csound from SciTE. If you *must* use an existing Windows program, use WordPad, not Notepad, and be sure to save your work as a plain text file with the proper filename extension.

```
-d -m135 -H0 -s -W -o dac -+rtaudio=pa -b 128 -B 2048 --expression-opt
```

The meaning of these options is as follows:

`-d` Do not show graphs of function tables.

`-m135` Print informational messages about audio amplitude, audio samples out of range, warnings, and errors, using color codes.

`-H0` Do not print a heartbeat at each kperiod.[3]

`-s` Use 16-bit short integers for audio samples.

`-W` Use the standard Microsoft WAV format for soundfiles.

`-o dac` Send real-time audio output to your computer's default audio interface (i.e., digital-to-audio converter).

`-+rtaudio=pa` Use the PortAudio driver for real-time audio (works on Windows, Linux and Apple).

`-b 128` The number of audio sample frames[4] in Csound's software buffer.

`-B 2048` The number of audio sample frames in the audio interface's hardware buffer. This should be a small (e.g. 2 to 10) integral multiple of `-b`.

`-expression-opt` Tell the Csound orchestra language compiler to optimize arithmetic and logic expressions.

For the complete meaning of all Csound options, see the reference manual [10]. The above options should work for real-time audio output on all operating systems and computers. For now, there is no need to change these options, but later you may wish to modify them according to what you learn about your computer and audio interface. The layers of buffering in Csound work as follows:

1. Every `ksmps` sample frames, Csound reads audio from the `spin` buffer into the `in` family of opcodes; gets score events from the score, MIDI, and other real-time control queues and dispatches those events to instrument instances; writes audio from the `out` family of opcodes to the `spout` buffer; and copies the `spout` buffer to the "software" or `-b` buffer. Consequently, `ksmps` determines the minimum granularity of event and audio processing.

2. Every `-b` sample frames, Csound copies the "software" or `-b` buffer to the "hardware" or `-B` buffer. If `-b` is a multiple of `ksmps`, then if Csound is late producing a `spout` buffer, the `-b` buffer contains enough audio to give Csound a chance to catch up during the next `ksmps`.

[3] A *kperiod* is one Csound control sample, during which Csound computes 1 or more audio sample frames. By computing anywhere from 10 to a hundred or so sample frames per kperiod, Csound can run much more efficiently.

[4] An audio sample is one number. An audio sample frame consists one number for each channel of an audio signal. When people say "sample rate," they usually mean "sample frame rate."

3. Every `-B` sample frames, the sound card plays the "hardware" or `-B` buffer. If `-B` is a multiple of `-b`, then if Csound is late producing a `-b` buffer, the `-B` buffer still contains enough audio so that the sound card can keep playing while Csound catches up during the next `-b` period. Consequently, `-B` determines the minimum latency of audio input and output.

Csound configuration is affected by a number of environment variables,[5] which are all documented in the Csound manual [10].

To run Csound, select the **Csound** command from the Csound Start menu. This will open a console window in the Csound bin directory and run Csound to display its command-line options (see Figure 2.10). Alternatively, you can open a console window from the (Windows **Start** menu, **Run** item, type `cmd` into the **Open:** field, press the **ENTER** key). Type `C: [ENTER]` (`[ENTER]` means press the `ENTER` key) or whatever the drive is where you installed Csound). Type `cd \Csound [ENTER]` to navigate to the `Csound` directory. Type `csound [ENTER]` to run Csound. To see even more options, type `csound --help [ENTER]`.

Now, type `csound examples\xanadu-high-resolution.csd [ENTER]`. The `.csd` file contains in plain text, like all `.csd` files, a Csound score, a Csound orchestra for rendering the score, and command-line options in the `<CsOptions>` tag to control the rendering. The meanings of the options for this piece are as follows:

`-R` Rewrite the header of the output soundfile periodically, so that if you stop Csound in mid-performance, or it crashes, you should still be able to hear as much of the soundfile as was written before Csound stopped.

`-W` Use the standard Microsoft WAV soundfile format.

`-Z` Dither the signal just before writing to the output. Dither is noise that is applied to the signal in order to mask and hide other noise.

`-f` Use floating-point numbers to represent audio samples. Float samples have the greatest dynamic range and precision.

`-o xanadu.wav` Output to a soundfile named `xanadu.wav`.

The messages (Figure 2.11) list the instruments that Csound has compiled, e.g. `instr 1`, `instr 2`, and so on (more on this later), then some other information about how Csound has compiled the orchestra and score in the `.csd` file, then the name of the output soundfile. Then come messages indicating the progress and status of rendering, e.g. `new alloc for instr 1:` indicates that a new instance of instrument 1 has been created to satisfy the demands of the score. Messages starting with `B`, e.g.

[5] An *environment variable* is a string in the form `NAME=value` that the user sets, and the operating system remembers and passes along to programs when they start. The program can look up the value that has been assigned to the variable in order to locate directories and files, set numbers, and so on. The proper way to set environment variables depends on your version of Windows. On Windows XP, go to the **Start Menu**, **Settings** item, **Control Panel** item, **System** icon, **Advanced** tab, **Environment Variables** button to bring up a dialog box where you can create, edit, or delete persistent environment variables.

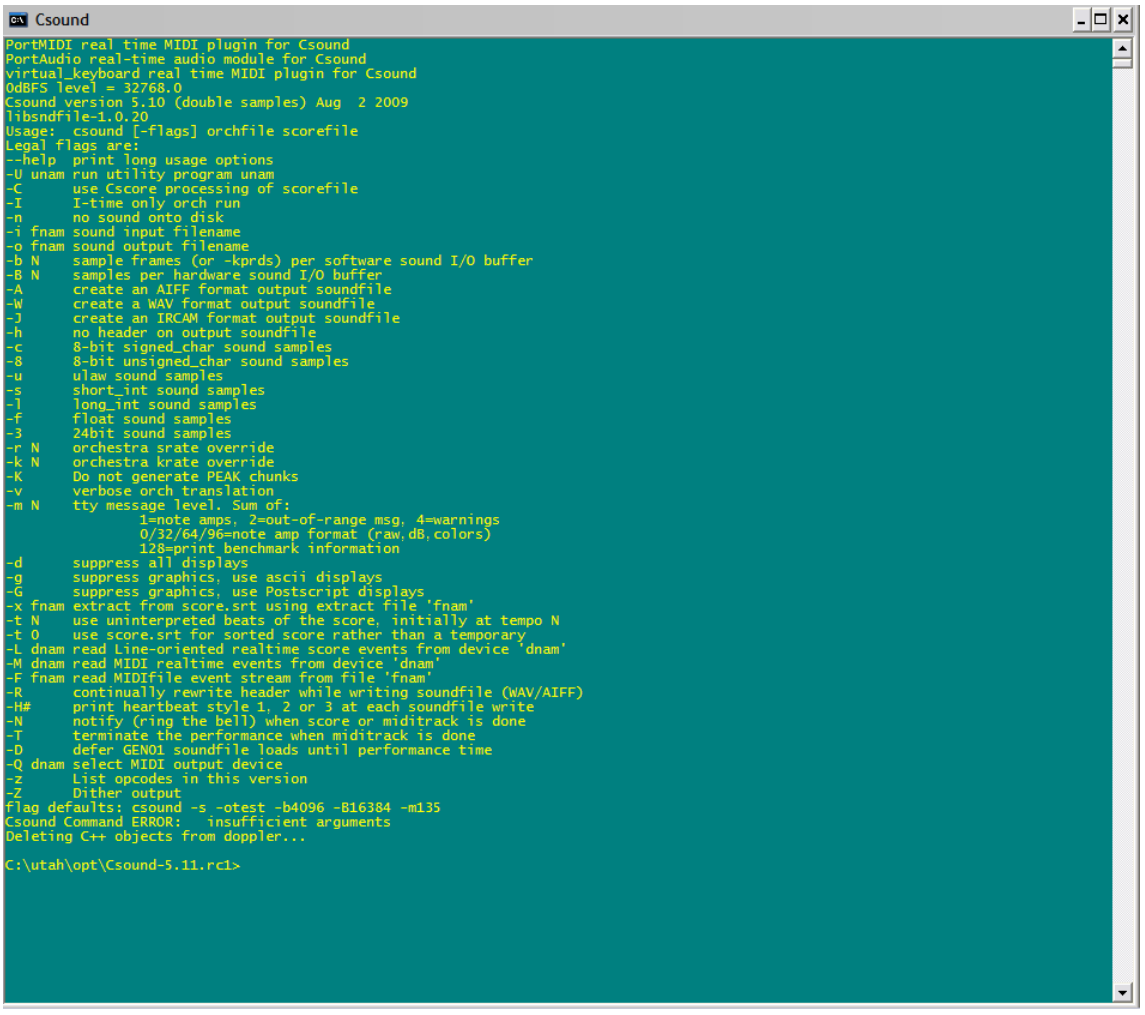

Figure 2.10.: **Csound Console**

```
B 15.500 .. 22.500 T 22.500 TT 22.500 M:    9286.3    9200.8
B 22.500 .. 22.600 T 22.600 TT 22.600 M:    5744.3    6443.3
B 22.600 .. 22.700 T 22.700 TT 22.700 M:    7632.9    7294.0
B 22.700 .. 22.800 T 22.800 TT 22.800 M:    8855.0    7862.5
B 22.800 .. 22.900 T 22.900 TT 22.900 M:    8845.9    7613.5
B 22.900 .. 23.000 T 23.000 TT 23.000 M:    8541.2    7858.1
```

indicate blocks of synthesis, including the time within a marked section of the score `T`, the total time for the whole score `TT`, and the mean amplitude `M` of the signal in each channel of the audio output during that time. These amplitudes are critical, for Csound can easily produce a signal that is so loud it clips. Every time this happens, Csound prints a warning message. A new block begins for each new score event.

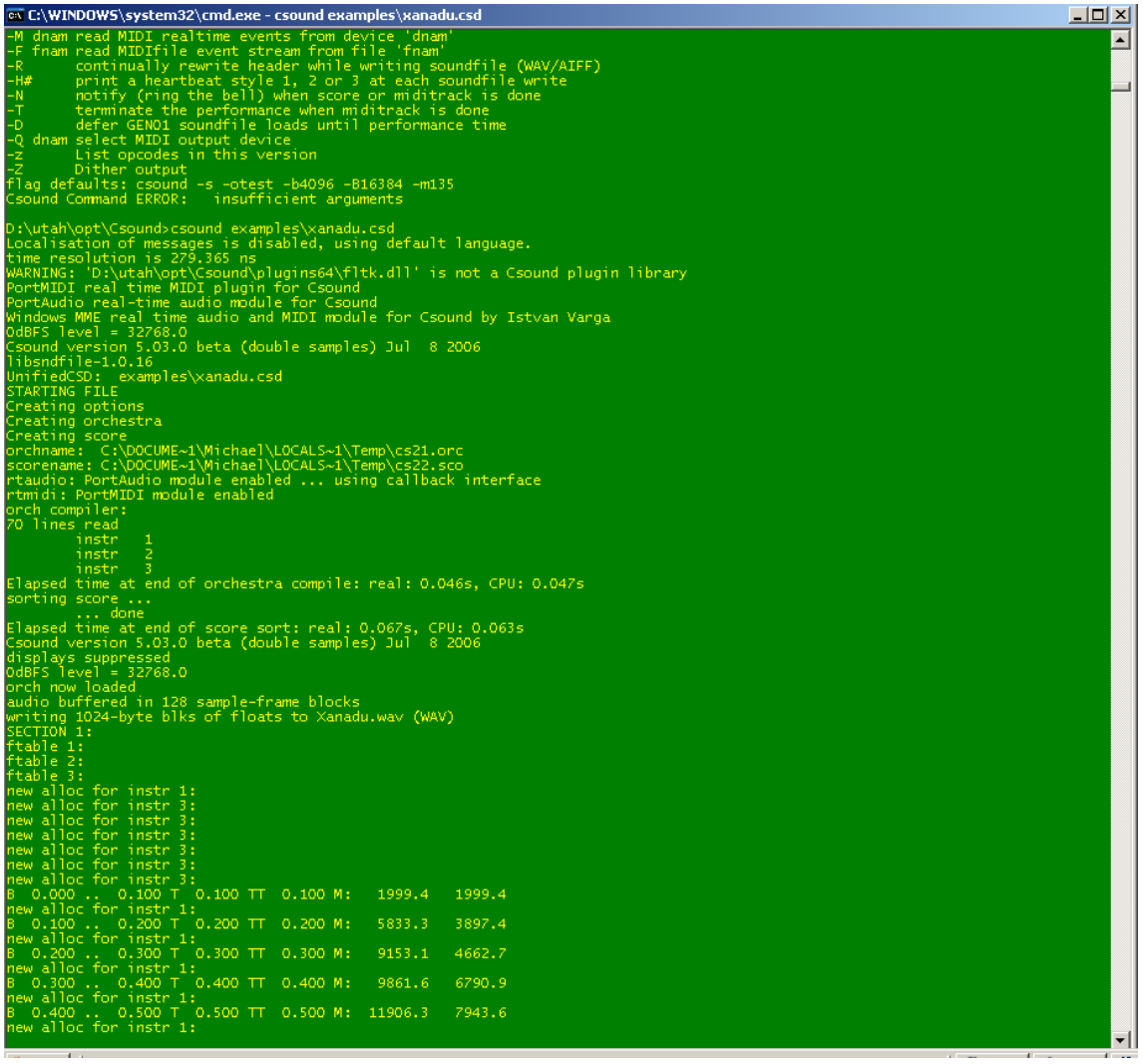

Figure 2.11.: **Command-Line Rendering**

There are various ways to now actually *hear* the piece. All installations of Windows feature the Windows media player, which can play high-resolution soundfiles, and which is usually accessible on the Windows task bar. Open the media player, and use the **File** menu, **Open** command to navigate to the Csound directory and

open the xanadu.wav file. You can now play the piece, although of course it will sound much better if you have an audio interface running into monitor speakers or good home stereo speakers. The piece may also sound good through headphones plugged directly into your computer, though that will depend on the quality of your computer's audio systems — newer computers have much better sound. Media Center PCs may even have high-resolution audio built in.

Configuring and Running QuteCsound

To run QuteCsound, select the **QuteCsound** command from the **Csound** Start menu. You should see something like Figure 2.12.

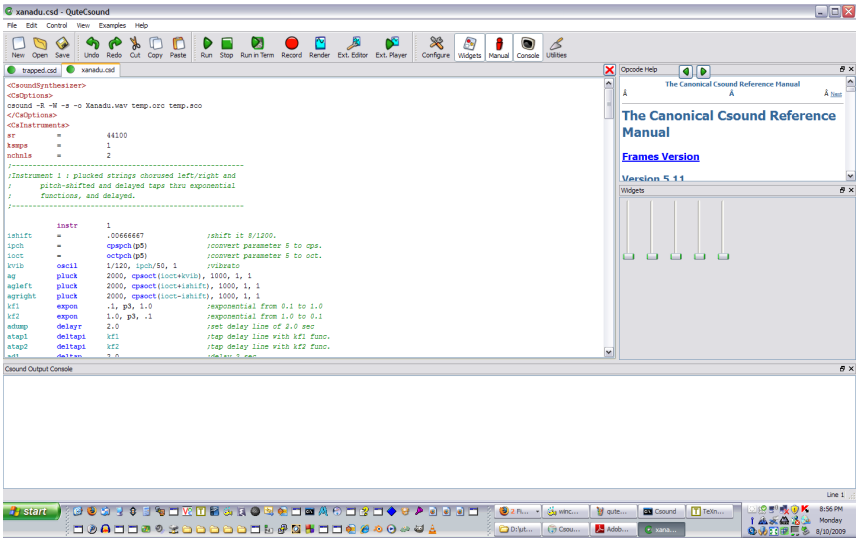

Figure 2.12.: QuteCsound.

Please be aware that configuring audio and MIDI on any computer can involve trial and error, reconfiguration of the operating system and/or device drivers, visits to online user forums and mailing lists, and patience. Using Csound can make this procedure even more tedious. If you are patient and methodical, you will almost certainly be able to achieve reliable, low-latency results with excellent audio quality. If you are familiar with other software synthesizers, you will soon be able to judge for yourself whether Csound has anything to offer in the sound department.

Note also that QuteCsound can be configured either using its own configuration dialogs, or through the Csound options section of the .csd file. In this tutorial, I focus on the QuteCsound configuration dialogs, because they make it easier to remember what you are doing.

All QuteCsound configuration examples in this tutorial are based on my own practice, using the Roland Edirol UA-25EX USB audio/MIDI interface on Windows XP and on Linux (Ubuntu or Eeebuntu). I chose this interface because it is small, bus-powered, and works reasonably well on both Windows and Linux. If you are using this device on Windows, make sure that the **ADVANCED DRIVER** switch on the back panel is set to ON and that the **SAMPLE RATE** switch is set to 48.

This is only the combination of settings for the UA-25EX that enables simultaneous audio input and output with MIDI.

Click on the **Configure** button to open the **Configuration** dialog, and select the **Run** tab. Configure the dialog as shown in Figure 2.13. The purpose of the settings is as follows:

- **Buffer Size (-b)** `100` is a small size that produces low latency. It also is the same as the sampling rate divided by the control rate (`ksmps`).
- **HW Buffer Size (-B)** `200` is a small integral multiple of the buffer size.
- **Additional command line flags** `-midi-key=4 -midi-velocity=5` routes MIDI key numbers to pfield 4 and MIDI velocity numbers to pfield 5. This enables the use of the same Csound instruments both for rendering to file, and for real-time MIDI performance. The other settings of `-r 48000 -k 480` set Csound's sampling rate to 48000 (the same as the UA-25EX) and control rate to 480. This combination of settings should produce an input latency on the order of 2 milliseconds. This is about the fastest possible. If you experience problems, double the buffer size and halve the control rate. If you still experience problems, double again. The `-R -W` parameters cause Csound to periodically update the soundfile header during performance so that the output soundfile will still be playable if the rendering is stopped partway through the score, and to use the standard Windows WAV output soundfile format.
- In the **File (offline render)** group:
  - Checking **Use QuteCsound options** and **Ignore CsOptions** means that QuteCsound will ignore the Csound options from the `.csd` file and use only the options from this dialog when rendering to a soundfile.
  - **File type** `WAVE` makes Csound use the industry-standard Microsoft WAV format for the output soundfile.
  - **Sample format** `32 bit` makes Csound use 32-bit floating-point samples in the output soundfile. This is extremely-high precision audio and is becoming standard in studio work.
  - Setting **Output Filename** enables QuteCsound to launch external applications to edit or play the Csound output soundfile.
- In the **Realtime Play** group:
  - Checking **Use QuteCsound options** and **Ignore CsOptions** means that QuteCsound will ignore the Csound options from the `.csd` file and use only the options from this dialog when rendering in real time.
  - **RT Audio Module** `portaudio` selects the PortAudio driver, which is advisable on Windows in order to use ASIO for low latency.
  - **Input device (-i)** `adc6` selects the ASIO input driver for the UA-25EX. This value was selected by clicking on the button with three dots to the right of the field. Select the appropriate driver for your own computer.

- **Output device (-o)** `dac6` selects the ASIO output driver for the UA-25EX. This value was selected by clicking on the button with three dots to the right of the field. Select the appropriate driver for your own computer.
- **RT MIDI module** `winmm` selects the Windows multimedia driver for MIDI output. This usually works better than the PortMidi driver on Windows. This value was selected by clicking on the button with three dots to the right of the field. Select the appropriate driver for your own computer.
- **Input device (-M)** `0` selects the UA-25EX MIDI in driver. This value was selected by clicking on the button with three dots to the right of the field. Select the appropriate driver for your own computer.

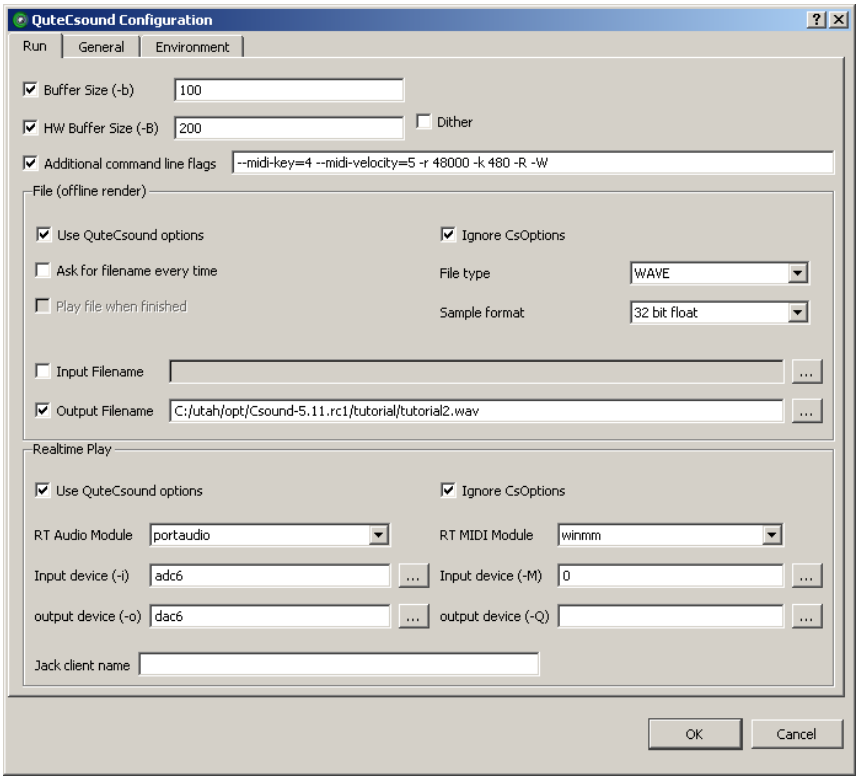

Figure 2.13.: **Configuring `QuteCsound` (Run)**

Select the **General** tab and configure it as shown in Figure 2.14. The purpose of the settings is as follows:

- In the **Editor** and **Console** groups, the font size of 9 displays more text while still being readable. Set an appropriate value for your own display.
- **Run Utilities and Render using Csound API** enables QuteCsound to run its embedded instance of Csound, which affords tighter integration and allows Csound messages to print in QuteCsound's console panel.

Now select the **Environment** tab and configure it as shown in Figure 2.15. The purpose of the settings is as follows.

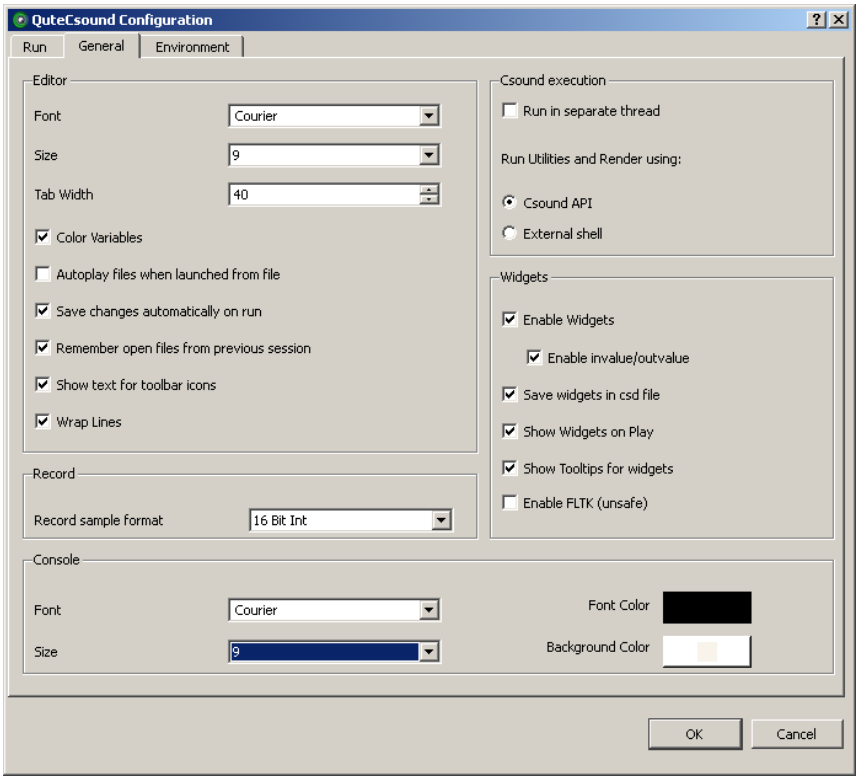

Figure 2.14.: **Configuring `QuteCsound` (General)**

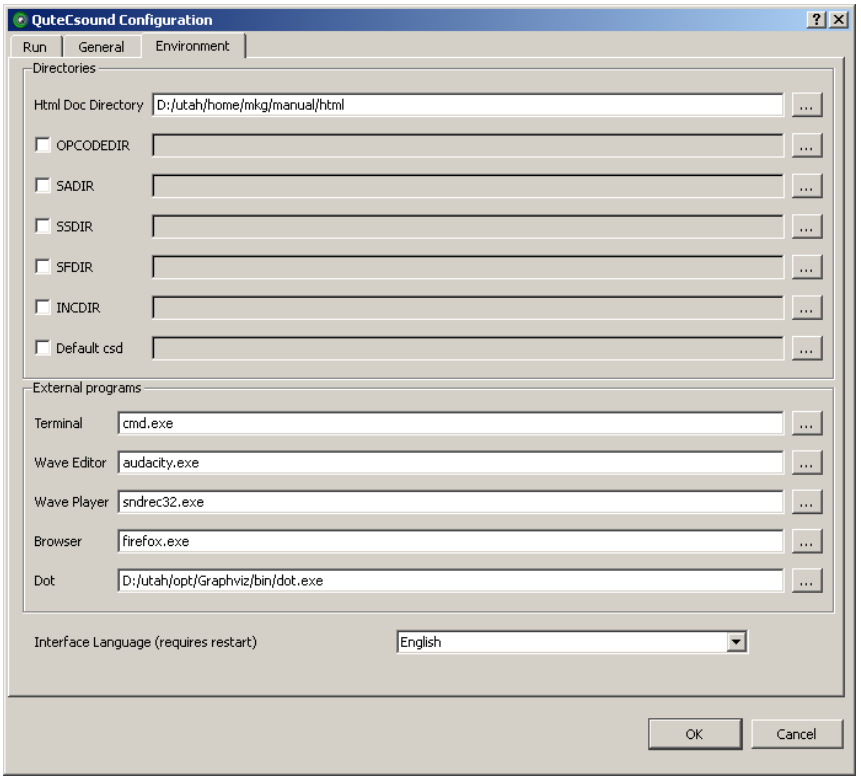

Figure 2.15.: **Configuring `QuteCsound` (Environment)**

- **Html Doc Directory** records the directory containing of the Csound reference manual HTML pages. Use the button with three dots to the right of the field to find this directory on your computer.

- **Wave Editor** records the complete pathname of an external soundfile editor that QuteCsound can use to display, edit, or play Csound output soundfiles. Use the button with three dots to the right of the field to find the appropriate program on your computer.

- **Wave Player** records the complete pathname of an external media player that QuteCsound can use to play Csound output soundfiles. Use the button with three dots to the right of the field to find the appropriate program on your computer.

- `Dot` records the complete pathname of the Graphviz `dot.exe` program, which QuteCsound can use to print useful flow charts of Csound orchestra code. Use the button with three dots to the right of the field to find this program on your computer.

To render ***Xanadu*** using QuteCsound, use the **File** menu, **Open File...** dialog to navigate to the Csound `examples` directory and load the `xanadu-high-resolution.csd` file. You should see something like Figure 2.16. Note that, if properly configured, the Csound Reference Manual is open in a panel in the upper right corner, and can be browsed to obtain help. Not the only that, but if you place the cursor on an opcode name in the editor and press `Shift-F1`, the manual will jump to the documentation for that opcode (Figure 2.17).

Now click on the **Render** button, to render the piece to a soundfile. You should see something like Figure 2.18.

As the piece renders, the **Csound output console** panel prints messages from Csound about the progress of rendering. You can stop rendering at any time by click on the **Stop** button. After stopping, you can restart.

When the piece has finished rendering, you can hear it by clicking on the **Edit** button for the **Output file** field, which, if you have configured an audio editor for `QuteCsound`, will open the editor with the output soundfile already loaded and ready to play or edit (Figure 2.19).

2.1.3. Real-Time MIDI Performance

Real-time MIDI performance means playing Csound as a live MIDI synthesizer. Your computer must have an audio interface connected to headphones or speakers, your computer must also have a MIDI interface, and you must plug the **MIDI out** port of your MIDI keyboard or other controller into the **MIDI In** port of your MIDI interface.

You start Csound with an orchestra that is designed for real-time MIDI performance, you play your controller, Csound renders what you play as you play it, and you hear the audio output from your speakers or headphones. Use QuteCsound's **File** menu, **Open** dialog to open the `CsoundAC.csd` file from the Csound examples

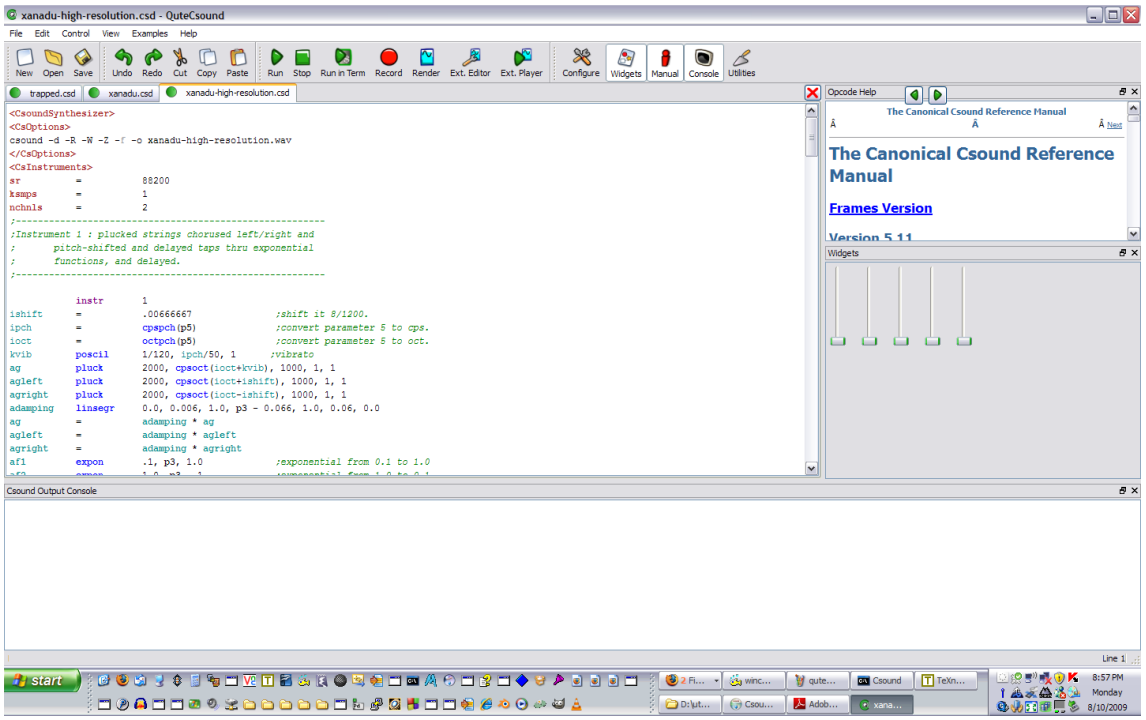

Figure 2.16.: **QuteCsound with *Xanadu***

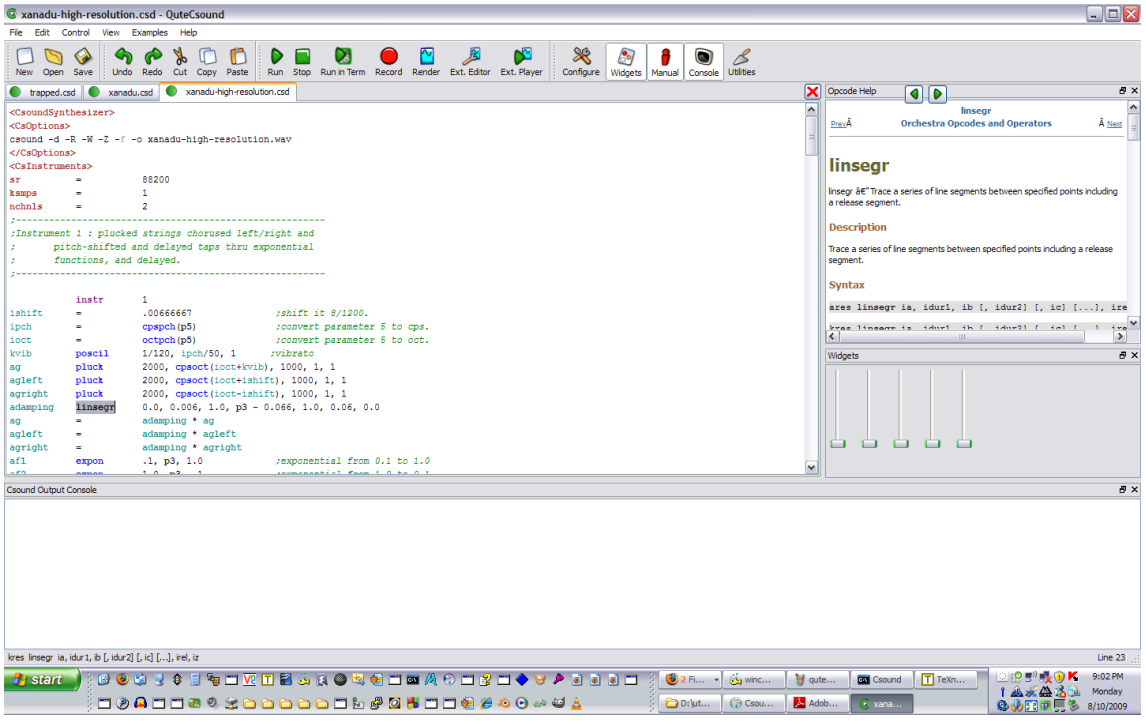

Figure 2.17.: **Contex-Sensitive Opcode Help**

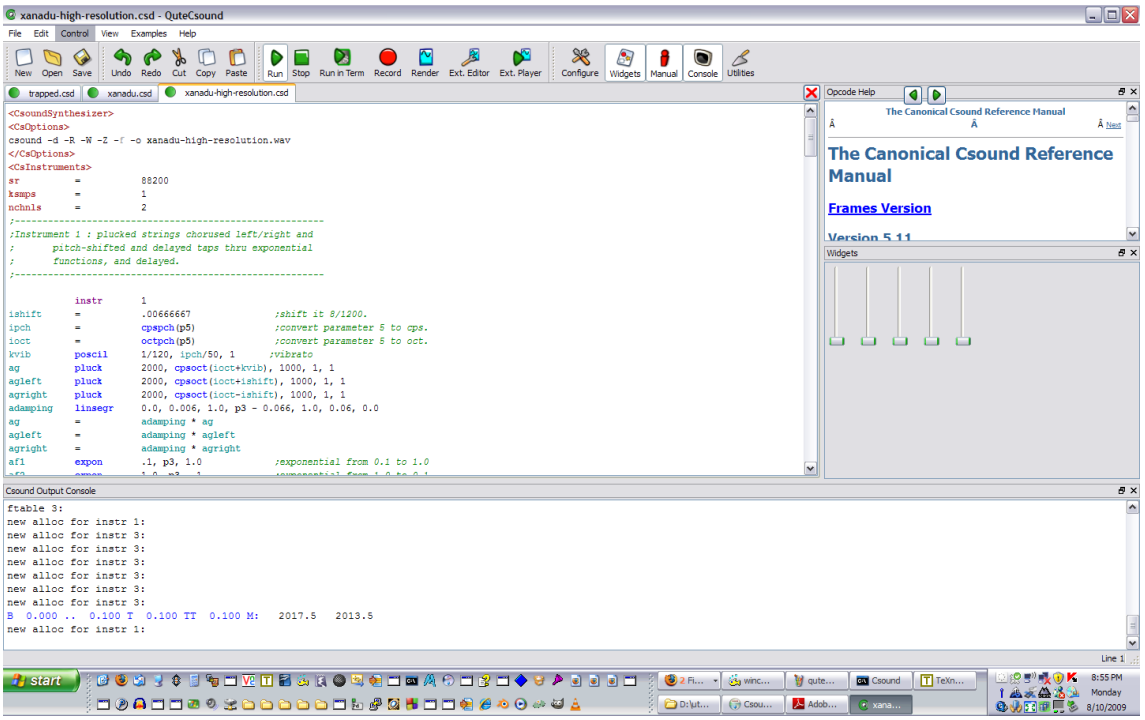

Figure 2.18.: **QuteCsound Rendering**

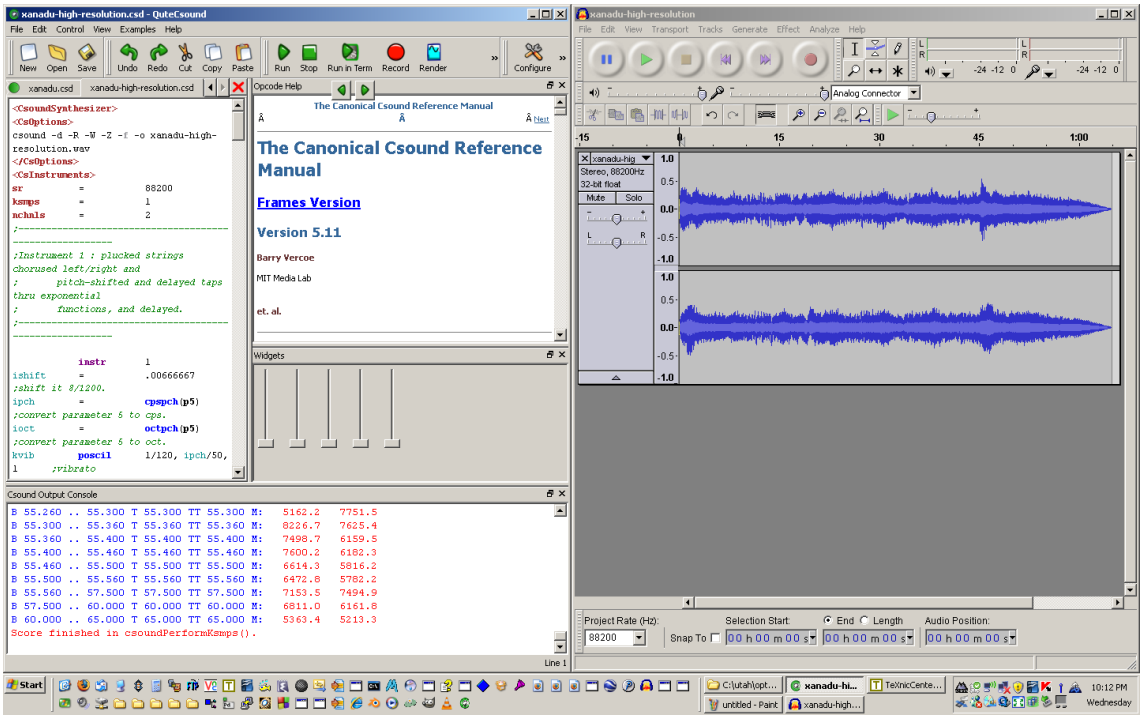

Figure 2.19.: **Edit/Play Output Soundfile**

directory. This is a large Csound orchestra in which all instruments are designed to be usable not only for rendering to a soundfile, but also for real-time performance with MIDI control.

In the Csound editor, edit line 88 or thereabouts to specify which instrument you want to play on MIDI channel 0 using the `massign` opcode. The following lines select instrument 19, a high-quality flute emulation adapted from code by Lee Zakian:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; A S S I G N   M I D I   C H A N N E L S   T O   I N S T R U M E N T S
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        massign             0, 19
                        massign             1, 19
```

The following explains how to do real-time MIDI performance on my notebook computer, running Windows XP Media Center Edition, using a Roland Edirol UA-25EX USB audio/MIDI interface (I chose this unit because it works reasonably well on both Windows and Linux). The configuration is as given above.

Click on the **Run** button. You should hear *nothing*, because the `<CsScore>` tag in the `.csd` file contains no notes — you will be playing the notes in. If you do hear anything, you have a problem! Click on the **Stop** button to end performance and reconfigure.

Now, play a few notes on your keyboard or other MIDI controller. You *should* hear something now. More specifically, you should hear a flute sound. If you don't hear anything, or if you do hear something but it sounds wrong, you have a problem. Click on the **Stop** button.

The most likely problem is that the selected audio or MIDI drivers that you selected are not suitable. Go back to the `QuteCsound` **Configuration** dialog, **Run** tab and change the driver selections. If that doesn't help, try larger buffers and a smaller control rate.

When all is configured, if you have a reasonably new personal computer with a reasonably up to date version of Windows, you will hear what you are playing within a few milliseconds of when you play it. Since your reaction time is probably around 20 milliseconds, and even the best keyboard players are only accurate within about 5 milliseconds, that is fast enough to seem almost instantaneous. You should see something like Figure 2.20.

You can stop rendering at any time by clicking on the **Stop** button, and after stopping you can restart.

To hear different Csound instruments, change the `massign` statement to other instrument numbers, or change the MIDI channel assignment of your MIDI controller, and start Csound again.

2.2. On Linux

To be completed.

2.3. On Apple

To be completed.

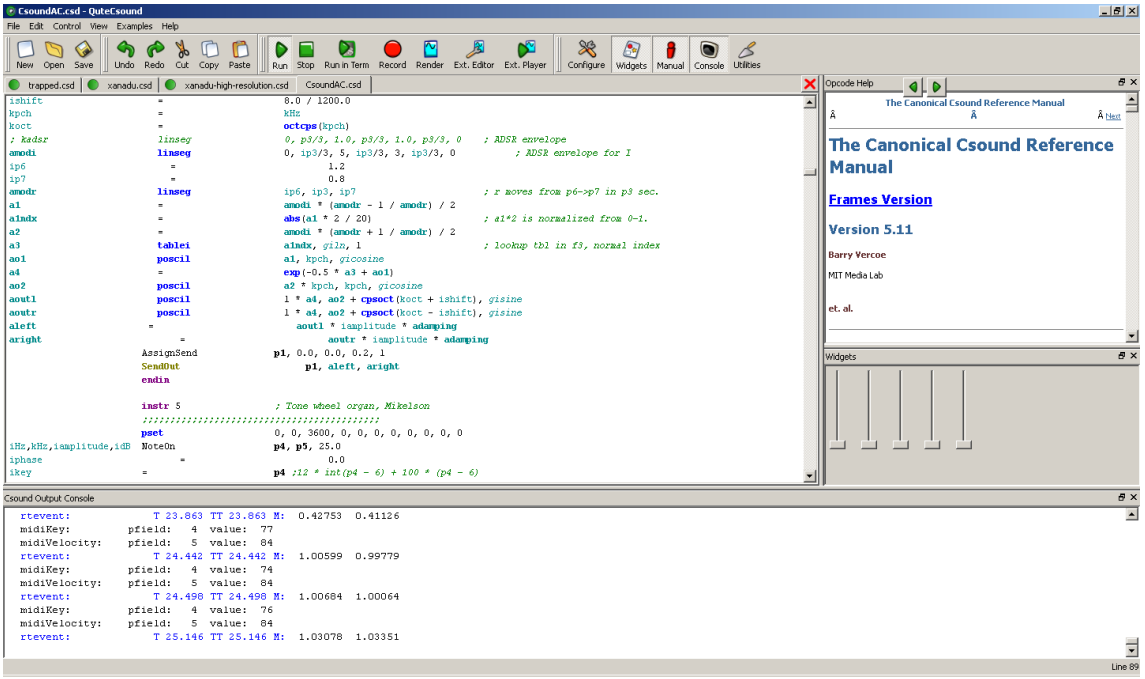

Figure 2.20.: **MIDI Performance**

3. Writing Orchestras and Scores

The chapter starts with two short sections on how software synthesizers in general, and Csound in particular, work. You can skip these sections if you are not interested. There follows a section on writing the simplest possible instrument, and making it sound better and better through a sequence of increasingly refined versions.

3.1. Signal Flow Graphs

Almost all software synthesizers run as a set of unit generators (*opcodes*, in Csound terminology) that are connected so that the outputs of some units feed into the inputs of other units. It is very similar to a modular electronic synthesizer, such as a Moog synthesizer, in which small electronic units are patched together with cords. In software engineering, this kind of wiring diagram is called a *synchronous signal flow graph*. Oscillators, filters, modulators, envelope generators, and even arithmetic operators and functions are all unit generators.

In Csound, each `instr` or instrument block in the orchestra code is one signal flow graph. The inputs to an instrument consist of any number of *pfields* (standing for *parameter fields*), which come from `i` statements in the score, or from real-time events:

`p1` Always represents instrument number, which can be an integer or a fraction. Score events with fractional numbers are considered to be "tied" in the sense that after an instrument instance is initialized, a new score event with the same fractional number is sent to the already running instrument instance, which skips its initialization run. This produces a very good approximation of a slur tying two notes in music notation.

`p2` Always represents the time that the score event begins, although this time can be in seconds or, if the score contains a `t` (tempo) statement, in musical beats.

`p3` Always represents the duration of the score event, in seconds or in beats; if -1, the event will continue indefinitely. Note that instruments can modify the value of their own `p3` fields.

`p4...pN` Higher pfields have user-defined meanings. However, throughout this tutorial, `p4` represents pitch as MIDI key number,[1] and `p5` represents loudness as MIDI velocity number.[2]

[1] MIDI key number represents pitch in semitones, ranging from 0 to 127, with middle C = 60. In Csound, fractional MIDI key numbers can be used to represent non-equally-tempered pitches.

[2] MIDI velocity number represents loudness in a roughly logarithmic scale, ranging from 0 to 127, with *mezzo-forte* being perhaps 100.

Each Csound opcode is one unit generator, and is written as one line of text. Assignment statements, logical operators, and arithmetic operators are also implemented, when the orchestra file is compiled, as unit generators.

Opcodes accept zero more input arguments, and output zero or more return values. The output of an instrument block is sent to output using various opcodes, usually `outs` or `outc`. Since these opcodes have no outputs inside the instrument, they are root nodes of the graph (of course, a graph may have more than one root node).

In Csound, variables and opcodes are active at different *rates*:

i-rate Initialization rate — scalar variables whose names begin with the letter `i`, and whose values are fixed when an instrument instance is initialized, and never change after that.

k-rate Control rate — scalar variables whose names begin with the letter `k`, and whose values can change at the control rate.

a-rate Audio rate – vector variables whose names begin with the letter `a`, and whose values can change at the sample frame rate. Obviously, all input and output audio signals must be represented in `a`-rate variables.

3.2. How Csound Works

3.2.1. Csound Files

Listing 3.1 shows a very simple `.csd` file, which creates the simplest possible instrument and plays one note on it. The code is however extensively commented.

Listing 3.1: **Simple Orchestra**

```
<CsoundSynthesizer>
<CsOptions>
-R -W -f -o tutorial.wav
</CsOptions>
<CsInstruments>
; Sample frames per second.
sr              =               88200
; Number of sample frames per control period.
ksmps           =               1
; Number of audio output channels.
nchnls          =               2
; Amplitude of 0 decibels full scale (maximum amplitude).
0dbfs           =               32767

; Instrument number one -- very simple.
                instr           1
; Pfield 4 contains pitch as MIDI key number.
ikey            =               p4
; Pfield 5 contains loudness as MIDI velocity number.
ivelocity       =               p5
; Translate MIDI key to linear octave.
ioctave         =               ikey / 12 + 3
; Translate linear octave to cycles per second.
ifrequency      =               cpsoct(ioctave)
; Translate MIDI velocity to decibels full scale.
idb             =               ivelocity / 127 * 84
; Translate decibels to output amplitude.
iamplitude      =               ampdb(idb)
; Generate a band-limited sawtooth wave.
```

```
aout            vco2            iamplitude, ifrequency
; Send the output to both channels
                outs            aout, aout
                endin
</CsInstruments>
<CsScore>
i 1 1 5 60 100
</CsScore>
</CsoundSynthesizer>
```

A `.csd` file is a kind of XML file, containing sections marked off by tags. The `<CsOptions>` tag contains command-line options, the `<CsInstruments>` tag contains the Csound orchestra, which in turns contains a header and one or more instrument definition blocks, and the `<CsScore>` tag contains zero or more `f` statements (for generating function tables) and `i` statements (for sending notes to instruments). Any line beginning with a semicolon is a comment and is not interpreted.

In order to make instrument definitions easier to read, many people follow the convention of writing each opcode line in 3 widely and evenly spaced columns, with the output variables flush left, the opcode itself in the middle (remember that `=` is an opcode), and the input parameters on the right. I also think it is easier to read if comments go above lines, not at the right of lines.

3.2.2. Performance Loop

When Csound reads the `.csd` file, this is what happens:

1. Csound loads any plugins in the `OPCODEDIR` (for 32 bit sample Csound) or `OPCODEDIR64` (for 64 bit sample Csound) directory.

2. Csound reads its input files. If the input is a `.csd` file, Csound creates a temporary orchestra (`.orc`) file from the `<CsInstruments>` tag of the `.csd` file, and a temporary score (`.sco`) file from the `<CsScore>` tag of the `.csd` file.

3. Csound parses its command-line options, which can come from various sources (in order of increasing precedence):
   a) Csound's internal defaults.
   b) A `.csoundrc` file in the user's home directory, or the directory specified by the `CSOUNDRC` environment variable.
   c) A `.csoundrc` file in the current directory.
   d) The `<CsOptions>` tag in the .csd file.
   e) The command line.

4. Csound loads and enables any plugin modules required for audio or MIDI input or output.

5. Csound reads the orchestra file, and sets the sample frame rate, kperiod size, and number of audio output channels from the `sr`, `ksmps`, and `nchnls` statements, respectively, in the orchestra header.

6. Csound parses the `instr` blocks in the orchestra file, and compiles each `instr` block into an instrument template, which contains storage for input fields, and two linked lists of opcode templates. One list is for initializing an instrument instance, and the other list is for operating the instance.

7. Csound reads the score file, translates tempo statements, score sections, macros, continuation and increment operators, and so on, and sorts the results to produce a sorted, time-warped, compiled score file.

8. Csound actually performs the compiled score with the compiled instrument templates:
   a) Csound runs down the initialization list for any global instruments or opcodes, and calls each opcode's initialization function (e.g., to load SoundFonts).
   b) Csound checks to see if any real-time events or score events are pending, or if performance has finished. If `f` statements are pending, Csound goes to step 8c. If `i` statements are pending, Csound goes to step 8d. If performance is finished, Csound goes to step 8f.
   c) Csound allocates memory for any pending `f` statements, and initializes the function table; this can involve computing a mathematical curve, or loading a soundfile or a table of data from the disk.
   d) Csound looks for an inactive instrument instance for each pending `i` statement. If an inactive instance is found, Csound activates it. If there is no inactive instance, Csound creates a new instance by copying the instrument template (and its associated lists of opcode templates). Csound fills in the instance's pfields from the `i` statement. Csound then runs down the instance's initialization list, and calls each opcode's initialization function.
   e) Csound performs one kperiod. Csound runs down the list of instrument instances. For each active instance, Csound runs down the instances's operation list, and calls each opcode's operation function. Inside the operation function, if there are any `a`-rate variables, an inner loop must run for `ksmps` sample frames to compute each element of the vector. If the current time has passed the sum of `p2` and `p3`, or if an instrument has turned itself off, Csound deactivates the instance. When all the instances have been run, Csound sends the audio output buffer to the output soundfile or device. Csound then goes back to step 8b.
   f) Csound calls a deinitialization function in each plugin, closes any device plugins, deallocates instrument instances, and resets itself for another performance (or exits).

3.3. Writing Your First Piece

Use a text editor to create a `.csd` file named `tutorial2.csd`, which should contain only the empty tags:

Listing 3.2: **Empty `.csd` File**

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
```

Now fill in the tags one at a time. If you are going to run the piece using `QuteCsound`, you do not need to fill in the `<CsOptions>` tag. It may be a good idea, however, to put in some reasonable default options:

```
<CsOptions>
-W -f -R -o tutorial2.wav
</CsOptions>
```

Create the orchestra header for a sample frame rate of 88200, a control sample rate of 1, and stereo channels (i.e. for a high-resolution stereo soundfile):

```
<CsInstruments>
sr              =               88200
ksmps           =               1
nchnls          =               2
</CsInstruments>
```

Add a global `ftgen` opcode to generate a global function containing a high-resolution sine wave. The number of the wavetable is stored in the global `gisine` variable. The pfields mean:

1. Function number (0 means automatically generate the number).

2. Time at which the function table will be created (0 means the beginning of performance).

3. Size of the table. The bigger the table, the less noise in the signal. 65536 is 2 to the 16th power, which produces a low-noise signal; increasing the size by 1 means that interpolating oscillators that require a power of 2 size have one element past the end of the table to use for interpolation (a *guard point*).

4. The `GEN` function used to generate the table; `GEN 10` generates a series of harmonic partials.

5. Further arguments depend on the `GEN` function. For `GEN` 10, the single pfield 1 means generate the first partial with amplitude 1, and no other partials — i.e. a sine wave.

```
<CsInstruments>
sr              =               88200
ksmps           =               1
nchnls          =               2

gisine          ftgen           0, 0, 65537,    10,     1
</CsInstruments>
```

3.3.1. Simple Sine Wave

Add an empty instrument definition for instrument number 1. Instrument definitions begin with the keyword `instr` and the instrument number, and end with the keyword `endin`.

```
<CsInstruments>
sr                =               88200
ksmps             =               1
nchnls            =               2

gisine            ftgen           0, 0, 65537,    10,     1

                  instr           1
                  endin
</CsInstruments>
```

In the instrument definition, create `i`-rate variables to receive MIDI key number and velocity number from pfields 4 and 5:

```
                  instr           1
ikey              =               p4
ivelocity         =               p5
                  endin
```

Translate the MIDI key number in semitones with middle C = 60 to linear octaves with middle C = 8, and translate the MIDI velocity number to range from 0 to 84 (roughly the dynamic range in decibels of a compact disc):

```
                  instr           1
ikey              =               p4
ivelocity         =               p5
ioctave           =               ikey / 12 + 3
idb               =               ivelocity / 127 * 84
                  endin
```

Translate the octave and decibels to Csound's native units, which are cycles per second and amplitude:

```
                  instr           1
ikey              =               p4
ivelocity         =               p5
ioctave           =               ikey / 12 + 3
idb               =               ivelocity / 127 * 84
ifrequency        =               cpsoct(ioctave)
iamplitude        =               ampdb(idb)
                  endin
```

Add a signal generator, in this case a precision wavetable oscillator for producing a sine wave from our global table:

```
                  instr           1
ikey              =               p4
ivelocity         =               p5
ioctave           =               ikey / 12 + 3
idb               =               ivelocity / 127 * 84
ifrequency        =               cpsoct(ioctave)
iamplitude        =               ampdb(idb)
asignal           poscil          iamplitude, ifrequency, gisine
                  endin
```

Send the signal you have generated to each channel of the stereo output:

```
                  instr           1
ikey              =               p4
ivelocity         =               p5
ioctave           =               ikey / 12 + 3
idb               =               ivelocity / 127 * 84
```

```
ifrequency      =               cpsoct(ioctave)
iamplitude      =               ampdb(idb)
asignal         poscil          iamplitude, ifrequency, gisine
                outs            asignal, asignal
                endin
```

Your new instrument takes 5 pfields:

1. Instrument number.
2. Time in seconds.
3. Duration in seconds.
4. MIDI key number.
5. MIDI velocity.

Create an `i` statement to play a middle C note at *mezzo-forte* on this instrument at time 1 second for 3 seconds:

```
<CsScore>
i 1 1   3 60 100
</CsScore>
```

Your piece is now ready to perform (Listing 3.3).

Listing 3.3: **Instrument Definition**

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
sr              =               88200
ksmps           =               1
nchnls          =               2

gisine          ftgen           0, 0, 65537,    10,     1

                instr           1
ikey            =               p4
ivelocity       =               p5
ioctave         =               ikey / 12 + 3
idb             =               ivelocity / 127 * 84
ifrequency      =               cpsoct(ioctave)
iamplitude      =               ampdb(idb)
asignal         poscil          iamplitude, ifrequency, gisine
                outs            asignal, asignal
                endin
</CsInstruments>
<CsScore>
i 1 1   3 60 100
</CsScore>
</CsoundSynthesizer>
```

Run `QuteCsound`. Use the **File** menu, **Open...** dialog to open your `tutorial2.csd` file. Use the **Configure** dialog, **Run** page and type `tutorial2.wav` in the **Output Filename** field. Click on the **Render** button to render the piece. When the rendering has completed, click on the **Ext. Editor** button to hear the piece (Figure 3.1).

Well, it's not a very interesting piece! And typing in note statements becomes extremely tedious, even for a simple piece like *Three Blind Mice*. Of course, people

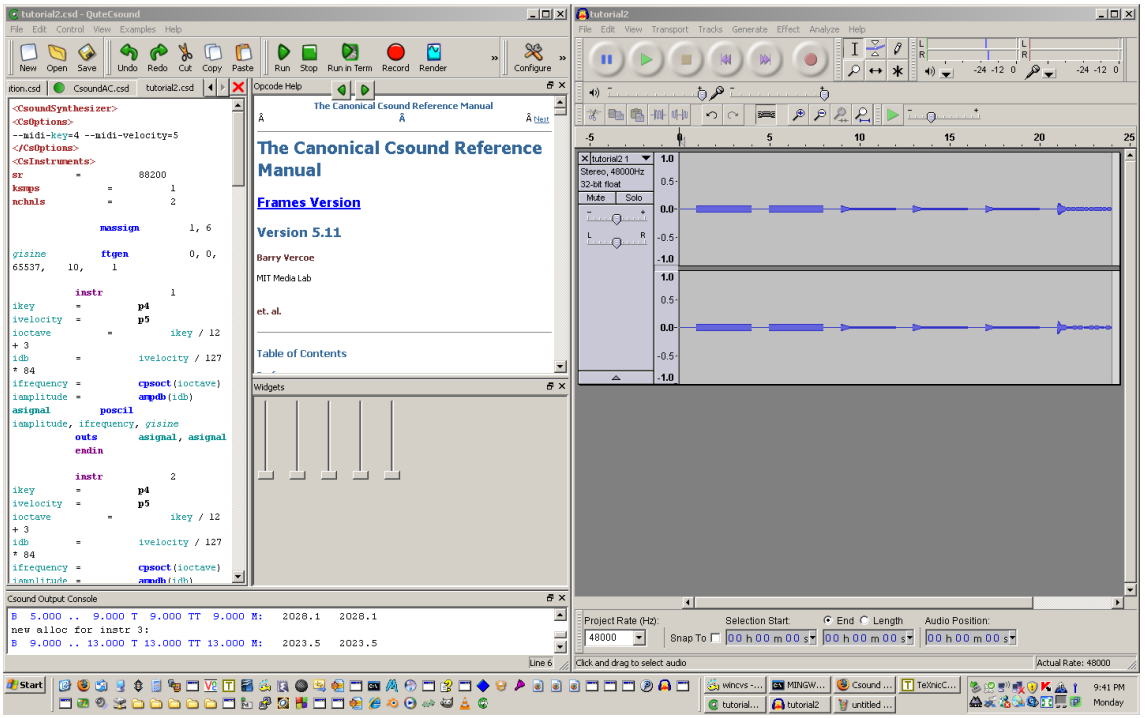

Figure 3.1.: `tutorial2.csd`

who actually use Csound to make music either write programs to generate scores, or they use a MIDI sequencer or notation software, or they play live. Here, however, we will focus only on improving the sound of the instrument.

3.3.2. Simple Sine Wave, De-Clicked

The most obvious problem right now is that the sound begins and ends with an obnoxious click. This is caused by the sharp discontinuity in the signal when the note abruptly turns on and abruptly turns off. This can be fixed by adding a damping envelope to tail off the clicks. In fact, every Csound instrument, with rare exceptions, should have such a damping envelope. Make a copy of your instrument, and number it 2, and add a `linsegr` opcode to tail off the clicks. It is a good idea to add the attack and release times to p3, just in case you have a very short note.

```
                instr           2
ikey            =               p4
ivelocity       =               p5
ioctave         =               ikey / 12 + 3
idb             =               ivelocity / 127 * 84
ifrequency      =               cpsoct(ioctave)
iamplitude      =               ampdb(idb)
asignal         poscil          iamplitude, ifrequency, gisine
iattack         =               0.0015
irelease        =               0.002
isustain        =               p3
p3              =               iattack + isustain + irelease
adamping        linsegr         0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
asignal         =               asignal * adamping
                outs            asignal, asignal
                endin
```

Also add a note to test the new instrument. The `^+4` in pfield 2 means to add 4 to pfield 2 of the previous i statement. We do this to create a second of silence between each test note. For all subsequent modifications, in the same way, we will make a copy of the previous instrument and add a new test note to play it.

```
<CsScore>
i 1 1   3 60 100
i 2 ^+4 3 60 100
</CsScore>
```

Again, render and listen. The note now starts and ends abruptly but without clicks, which is what we want. Of course, the sound is still boring.

3.3.3. Simple Sine Wave, De-Clicked, ADSR Envelope

Let's add a real envelope to give some shape to the sound. Use the `mxadsr` opcode to add an attack, decay, sustain, release (ADSR) envelope with exponentially rising and falling segments (this is musically one of the commonest types of envelope).

```
                instr           3
ikey            =               p4
ivelocity       =               p5
ioctave         =               ikey / 12 + 3
idb             =               ivelocity / 127 * 84
ifrequency      =               cpsoct(ioctave)
iamplitude      =               ampdb(idb)
ienvattack      =               0.004
ienvdecay       =               0.5
ienvlevel       =               0.25
ienvrelease     =               0.05
aenvelope       mxadsr          ienvattack, ienvdecay, ienvlevel, ienvrelease
asignal         poscil          iamplitude, ifrequency, gisine
asignal         =               asignal * aenvelope
iattack         =               0.0015
irelease        =               0.002
isustain        =               p3
p3              =               iattack + isustain + irelease
adamping        linsegr         0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
asignal         =               asignal * adamping
                outs            asignal, asignal
                endin
```

Better, but a sine wave is too plain.

3.3.4. Frequency Modulation, De-Clicked, ADSR Envelope

Add some basic frequency modulation to thicken up the sound. Use another `poscil` opcode to modulate the frequency of the signal generating oscillator. This has the effect of generating additional harmonics in the signal, whose content is controlled by both the amplitude and the frequency of the modulation.

```
                instr           4
ikey            =               p4
ivelocity       =               p5
ioctave         =               ikey / 12 + 3
idb             =               ivelocity / 127 * 84
ifrequency      =               cpsoct(ioctave)
iamplitude      =               ampdb(idb)
ienvattack      =               0.004
ienvdecay       =               0.5
ienvlevel       =               0.25
ienvrelease     =               0.05
```

```
aenvelope       mxadsr          ienvattack, ienvdecay, ienvlevel, ienvrelease
amodulator      poscil          800.0, ifrequency * 7.00, gisine
asignal         poscil          iamplitude, ifrequency + amodulator, gisine
asignal         =               asignal * aenvelope
iattack         =               0.0015
irelease        =               0.002
isustain        =               p3
p3              =               iattack + isustain + irelease
adamping        linsegr         0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
asignal         =               asignal * adamping
                outs            asignal, asignal
                endin
```

The sound is thicker, but it's not changing much as it sounds. Real musical sounds tend to vary subtly all the time.

3.3.5. Frequency Modulation, De-Clicked, ADSR Envelope, Time-Varying Modulation

Take a step in this direction by using the ADSR envelope to modulate not only the signal amplitude, but also the amount of frequency modulation. The only difference is to multiply the `amodulator` variable by the `aenvelope` variable.

```
                instr           5
ikey            =               p4
ivelocity       =               p5
ioctave         =               ikey / 12 + 3
idb             =               ivelocity / 127 * 84
ifrequency      =               cpsoct(ioctave)
iamplitude      =               ampdb(idb)
ienvattack      =               0.004
ienvdecay       =               0.5
ienvlevel       =               0.25
ienvrelease     =               0.05
aenvelope       mxadsr          ienvattack, ienvdecay, ienvlevel, ienvrelease
amodulator      poscil          800.0, ifrequency * 7.00, gisine
asignal         poscil          iamplitude, ifrequency + amodulator * aenvelope,
    gisine
asignal         =               asignal * aenvelope
iattack         =               0.0015
irelease        =               0.002
isustain        =               p3
p3              =               iattack + isustain + irelease
adamping        linsegr         0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
asignal         =               asignal * adamping
                outs            asignal, asignal
                endin
```

The sound is now almost usable. In fact, in some contexts, it probably *is* usable. In general, the more notes are playing, the simpler the actual sounds should be, and the fewer notes are playing, the more complex the individual notes should be. This sound would probably be usable in a busy texture. But suppose we are hearing only a few notes at a time?

3.3.6. Frequency Modulation, De-Clicked, ADSR Envelope, Time-Varying Modulation, Stereo Phasing

Add some delay lines with modulation of the delay times in opposing phase. This will create a moving texture that will shift from one side of the sound stage to the other. Apply the de-clicking envelope to the signal written to the delay line as well.

```
                instr           6
ikey            =               p4
ivelocity       =               p5
ioctave         =               ikey / 12 + 3
idb             =               ivelocity / 127 * 84
ifrequency      =               cpsoct(ioctave)
iamplitude      =               ampdb(idb)
ienvattack      =               0.004
ienvdecay       =               0.5
ienvlevel       =               0.25
ienvrelease     =               0.05
aenvelope       mxadsr          ienvattack, ienvdecay, ienvlevel, ienvrelease
amodulator      poscil          800.0, ifrequency * 7.00, gisine
asignal         poscil          iamplitude, ifrequency + amodulator * aenvelope,
    gisine
asignal         =               asignal * aenvelope
iattack         =               0.0015
irelease        =               0.002
isustain        =               p3
p3              =               iattack + isustain + irelease
adamping        linsegr         0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
krtapmod        poscil          0.002, 1.1, gisine, 0
kltapmod        poscil          0.003, 1, gisine, 0.5
adump           delayr          1.0
ad1             deltapi         0.025 + kltapmod
ad2             deltapi         0.026 + krtapmod
                delayw          asignal * adamping
aleft           =               asignal + ad1
aright          =               asignal + ad2
                outs            aleft * adamping, aright * adamping
                endin
```

3.3.7. MIDI Performance

You can easily modify your patch in order to play it live with a MIDI controller (Figure 3.2).

1. Add an `massign 1, 6` statement in the orchestra header, to send MIDI channel 1 to Csound instrument 6.

2. Add a `pset` statement to instrument 6 to set default values for all 5 of your pfields, so that instrument instances triggered by live MIDI events will receive values (otherwise, warning messages about p4 and p5 not being legal for MIDI will print). Such default values can be useful if you use score pfields to set sound-generating parameters in your instruments. In this case, they can all be zeros.

3. You may wish to delete all `i` statements from the `<CsScore>` tag. If you do so, you must add an `f 0 3600` statement, to tell Csound to render without score events for 3600 seconds (of course 3600 can be any value).

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
sr              =               88200
ksmps           =               1
nchnls          =               2

                massign         1, 6
```

```
gisine          ftgen           0, 0, 65537,    10,     1


                instr           6
                pset            0, 0, 0, 0, 0
ikey            =               p4
ivelocity       =               p5
ioctave         =               ikey / 12 + 3
idb             =               ivelocity / 127 * 84
ifrequency      =               cpsoct(ioctave)
iamplitude      =               ampdb(idb)
ienvattack      =               0.004
ienvdecay       =               0.5
ienvlevel       =               0.25
ienvrelease     =               0.05
aenvelope       mxadsr          ienvattack, ienvdecay, ienvlevel, ienvrelease
amodulator      poscil          800.0, ifrequency * 7.00, gisine
asignal         poscil          iamplitude, ifrequency + amodulator * aenvelope,
    gisine
asignal         =               asignal * aenvelope
iattack         =               0.0015
irelease        =               0.002
isustain        =               p3
p3              =               iattack + isustain + irelease
adamping        linsegr         0.0, iattack, 1.0, isustain, 1.0, irelease, 0.0
krtapmod        poscil          0.002, 1.1, gisine, 0
kltapmod        poscil          0.003, 1, gisine, 0.5
adump           delayr          1.0
ad1             deltapi         0.025 + kltapmod
ad2             deltapi         0.026 + krtapmod
                delayw          asignal * adamping
aleft           =               asignal + ad1
aright          =               asignal + ad2
                outs            aleft * adamping, aright * adamping
                endin
</CsInstruments>
<CsScore>
i 1 1   3 60 100
i 2 ^+4 3 60 100
i 3 ^+4 3 60 100
i 4 ^+4 3 60 100
i 5 ^+4 3 60 100
i 6 ^+4 3 60 100
</CsScore>
</CsoundSynthesizer>
```

Click on the **Run** button, play your MIDI controller, and you should see something like Figure 3.2.

If you plan to do both off-line rendering and live performance, you may wish to standardize some aspects of your instrument definitions:

1. Always use the new MIDI routing flags such as `--midi-key` for MIDI input, not the older MIDI opcodes such as `notnum`, or even the more recent MIDI interoperability opcodes such as `midinoteonkey`.

2. Always put function table statements in the orchestra header, not in the score; in other words, use `ftgen` instead of `f` statements. If you put function tables in the score, you won't be able to just throw out a score to use an orchestra in live performance.

3. Always specify pitch as MIDI key number.

4. Always specify loudness as MIDI velocity.

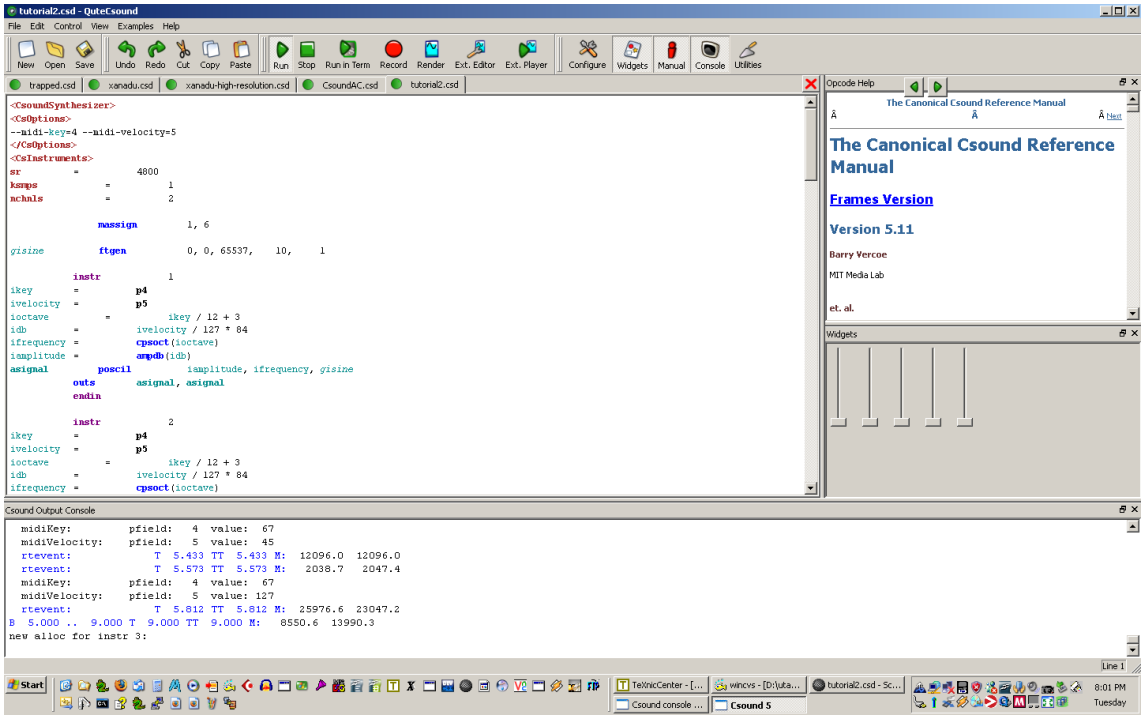

Figure 3.2.: **Playing `tutorial2.csd` Live**

5. In fact, always use the same set of standard pfields in all your instruments — you may add additional pfields to set sound generating parameters that are specific to an instrument.

6. Always use a `pset` statement to give a default value to each pfield, even if it is 0.

7. Always use a releasing envelope generator so that notes will end gracefully during live performance. The names of all the releasing envelope opcodes end with `r`.

8. Worry about sound quality first, and efficiency second. If your computer is having trouble keeping up, you can take a good sounding orchestra and figure out where to substitute more efficient opcodes (`oscil` for `poscil`, for example) much more easily than you can make an orchestra written for efficiency sound good.

This chapter is only a superficial introduction to a very deep topic — some excellent books have been written on it [4, 5, 6].

4. Using CsoundVST

CsoundVST is an extended version of Csound that provides the ability to run as a VST instrument (VSTi) or effect plugin in hosts such as Cubase [13]. One reason for doing this is to write pieces for Csound in standard music notation. Another reason is that Csound instruments can easily sound better than most other VSTis, although some Csound orchestras use a lot of CPU time. This chapter shows how to use Csound as a programmable VSTi in Cubase SX 3. Other audio sequencers should follow similar procedures.

4.1. Configuring CsoundVST

The following assumes that you have installed Csound from a version of the Windows installers that includes CsoundVST (the current version on SourceForge does not).

To configure Cubase for CsoundVST, run Cubase. Use the **Devices** menu, **Plugin information** dialog, **VST Plug-ins** tab. In the **Shared VST Plug-ins folders** field type the path to `CsoundVST.dll`, again `;C:\Csound\bin`. Then click on the **Add** button to append the CsoundVST path to the existing Cubase plugin path (Figure 4.1).

Verify that CsoundVST is now available as follows. Quit Cubase and start it again, and use the **Devices** menu, **VST Instruments** dialog to select CsoundVST as a VSTi. After a brief delay for loading, you should see something like Figure 4.2.

If you don't see this, see Footnote 5 about environment variables. Look for a variable named `PYTHONPATH`. If it exists, append `;C:\Csound\bin` to its value. If does not exist, create it with the value `;C:\Csound\bin`. Then try again.

4.2. Using CsoundVST

In order to use CsoundVST:

1. Begin a Cubase song, and create a MIDI track in it.
2. Create an instance of CsoundVST.
3. Load a Csound orchestra in CsoundVST.
4. Configure the orchestra for VST input and output.
5. Compile the orchestra.
6. Select CsoundVSt as an output for the MIDI track.

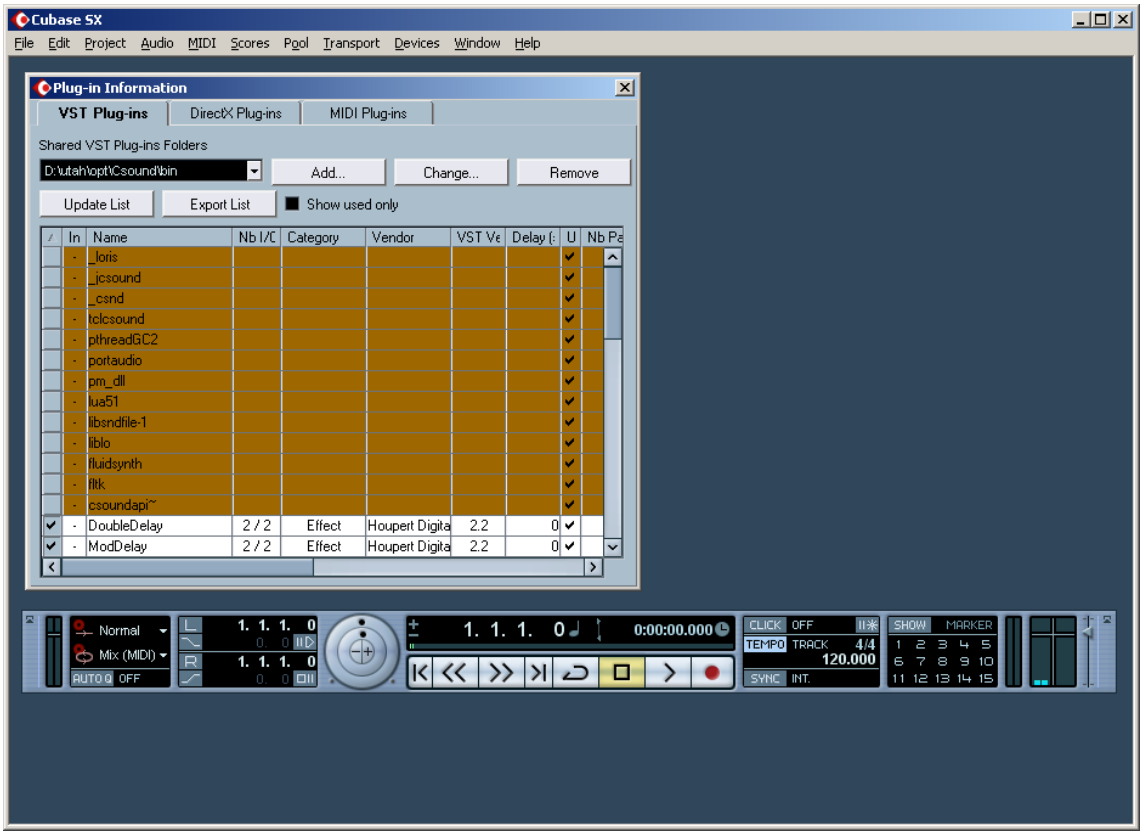

Figure 4.1.: **CsoundVST Plugin Path**

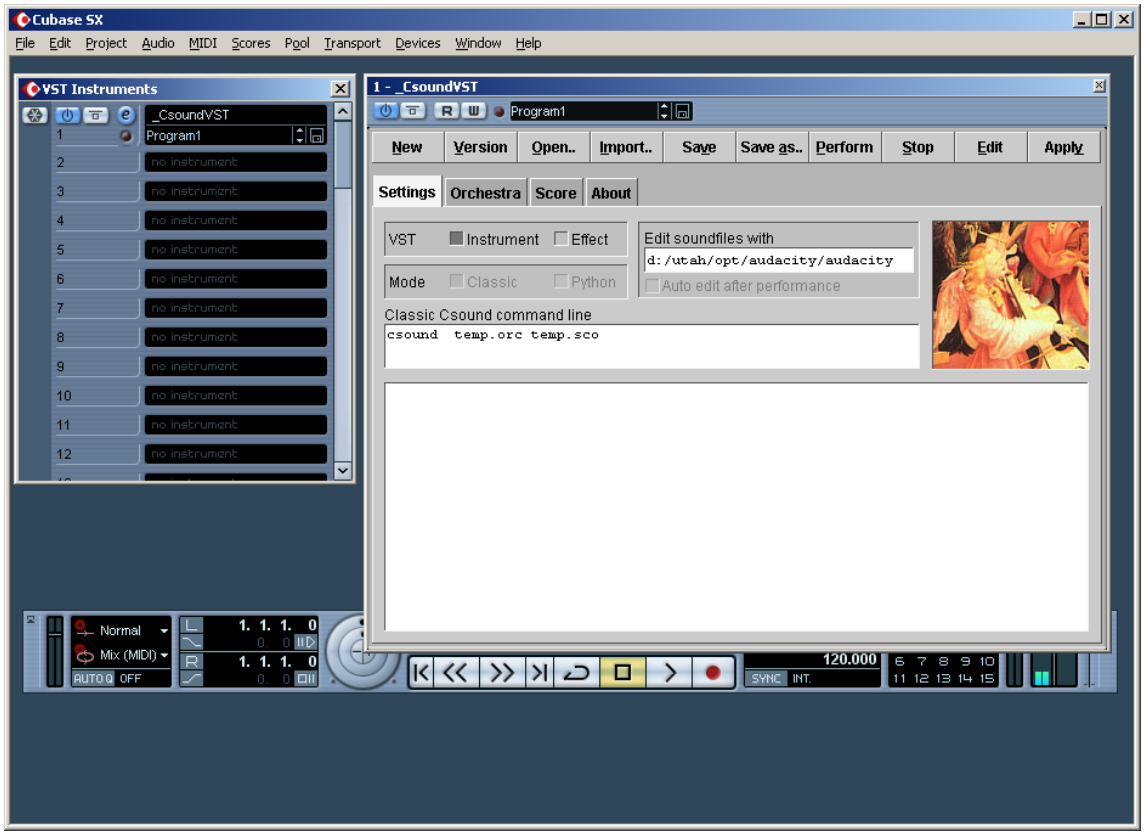

Figure 4.2.: **CsoundVST Loaded**

7. Assign your track's MIDI channel to a Csound instrument number in your orchestra (in fact, you can create any number of tracks assigned to CsoundVST, and you can also create multiple instances of CsoundVST).

8. Enter some music — by playing notes in from a MIDI controller, by importing a MIDI file, by using the piano roll editor, or by writing music notation.

Alternatively, with Cubase 4 and later, you can create an Instrument track, which combines the VST instrument and the MIDI track into one track; the Csound orchestra would be loaded and configured in the same way as above.

These steps can be carried out as follows.

4.2.1. Create a Cubase Song

Run Cubase, and use the **File** menu, **New** item to create a new empty project (Figure 4.3).

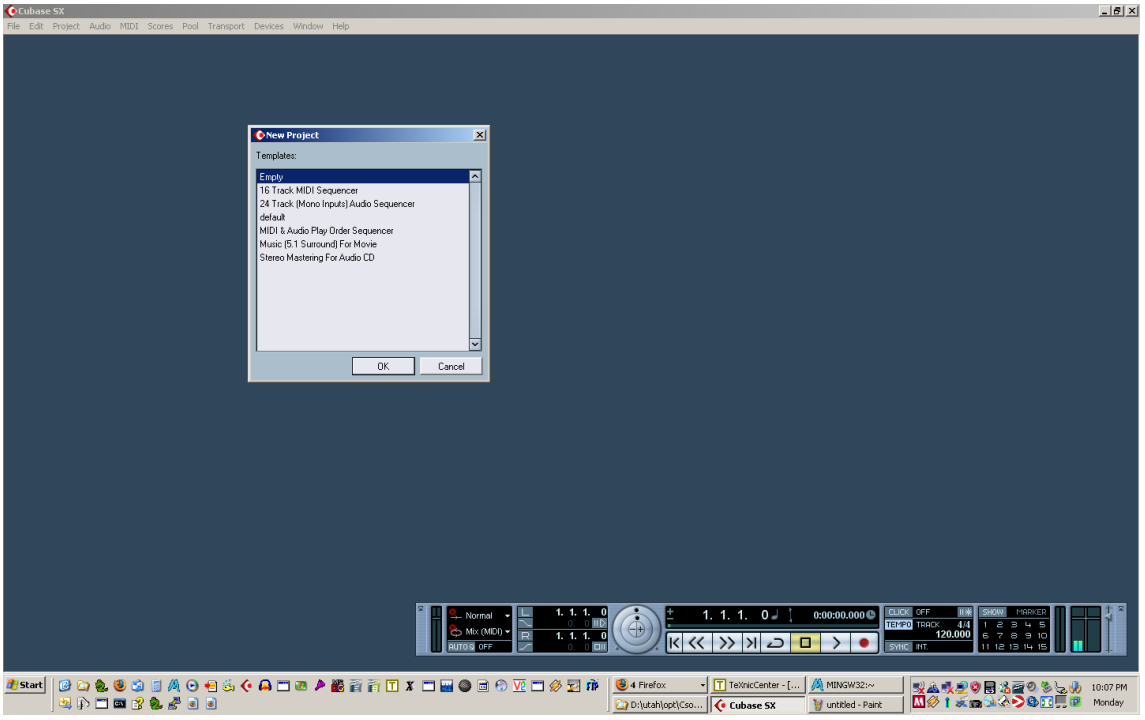

Figure 4.3.: **Creating a New Project**

Use the Project menu to add a new MIDI Track to your song (Figure 4.4).

4.2.2. Create an Instance of CsoundVST

Use the **Devices** menu, **VST Instruments** dialog to create a new instance of CsoundVST. Right-click the mouse on an empty field to bring up a context menu listing available VSTis, and select **CsoundVST**. You should now see the CsoundVST GUI in Cubase (Figure 4.5). Make sure that the **Instrument** checkbox is enabled; if not, enable it, then click on the **Apply** button to save your preference.

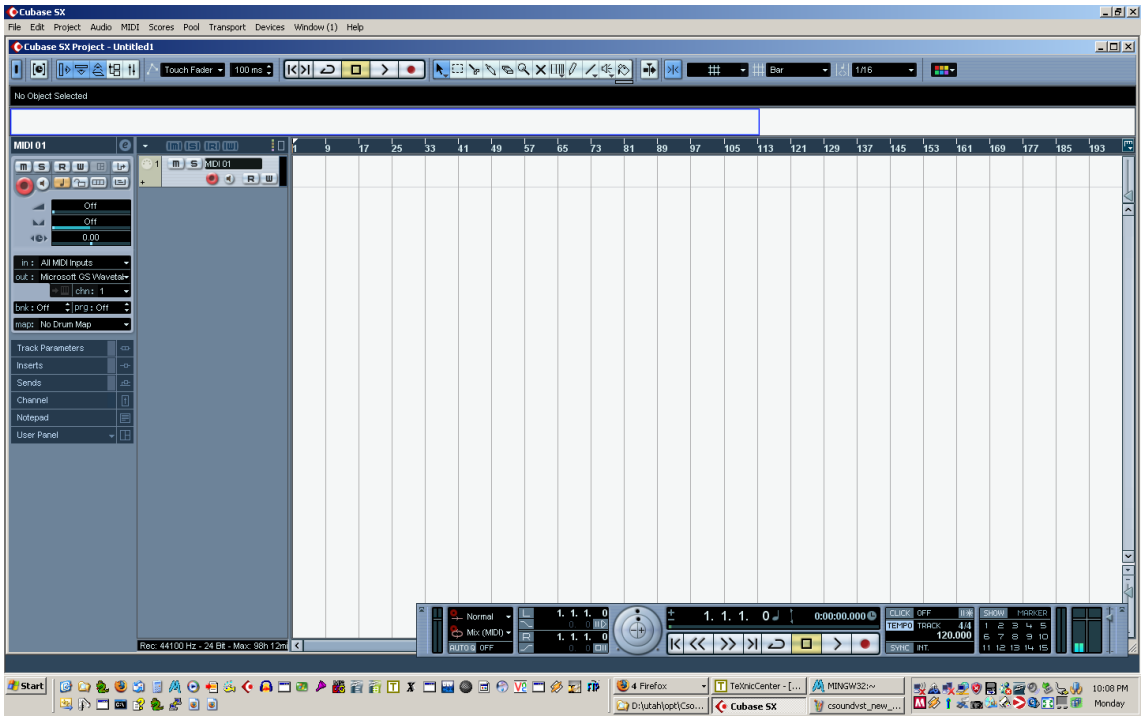

Figure 4.4.: **Creating a New Track**

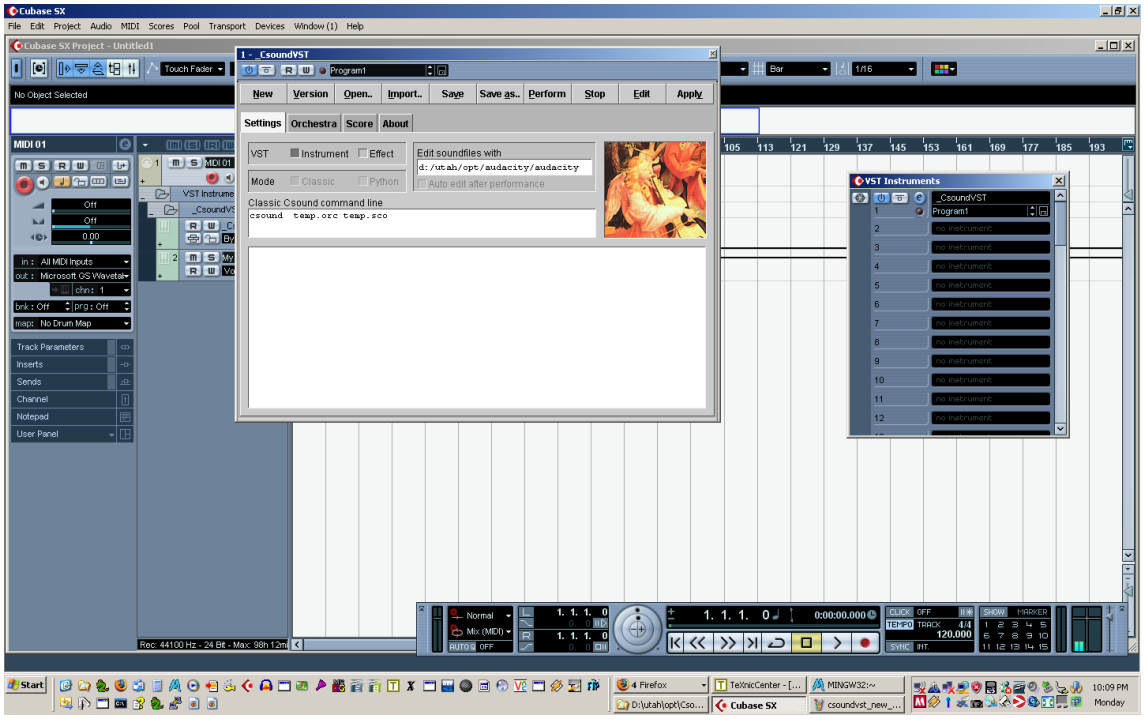

Figure 4.5.: **Creating a New Instance of CsoundVST**

4.2.3. Load a Csound Orchestra

Click on CsoundVST's **Open** button, navigate to `C:\Csound\examples` directory, and open the `CsoundVST-nomixer-flags.csd` file, which contains a prewritten orchestra of sample Csound instruments for VST plugin use (Figure 4.6; for some reason, the Mixer opcodes don't seem to work in CsoundVST). You can click on the **Orchestra** tab to look at or edit the code.

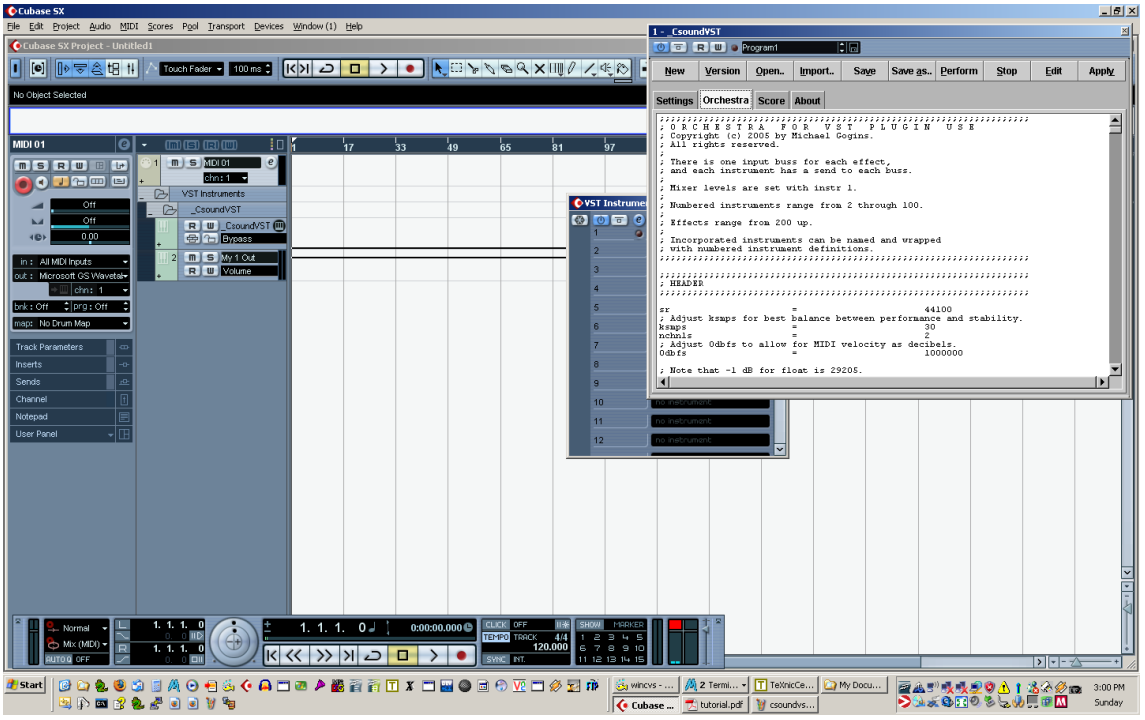

Figure 4.6.: **Loading an Orchestra**

4.2.4. Configure the Orchestra for VST

Click on CsoundVST's **Settings** tab, and configure the orchestra you have loaded to work inside a VST plugin by typing the following options in the **Classic Csound command line** text field.

```
csound -m3 -f -h -+rtmidi=null -M0 --midi-key-oct=4 --midi-velocity=5 -d -n temp.
    orc temp.sco
```

The meanings of each option are as follows. Each setting that is *required* for VST performance is indicated.

`csound` In CsoundVST, the Csound command must be entered just as if you were executing this command line *on* the command line.

`-m3` Display Csound messages to level 3: amplitude messages and signal out of range warnings.

`-f` Output floating-point samples.

`-h` Do not output a soundfile header (which might sound like a click), since Csound's audio output is going straight into Cubase.

`-+rtmidi=null` *Required.* Use a "dummy" MIDI driver. CsoundVST's code inserts the parts of a MIDI driver into Csound that CsoundVST requires to receive MIDI from the VST host.

`-M0` *Required.* Receive MIDI from port 0 (again, this is a "dummy" that simply enables Csound to receive MIDI events).

`-midi-key-oct=4` *Required.* Send MIDI note on message key numbers as linear octaves to pfield 4 of the Csound instruments in the orchestra.

`-midi-velocity=5` *Required.* Send MIDI velocity key numbers to pfield 5 of the Csound instruments in the orchestra.

`-d` *Required.* Display no graphs of wavetables.

`-n` *Required.* Do not send any audio output to actual audio devices or soundfiles — CsoundVST copies audio straight out of the internal buffers of Csound into the host buffers.

`temp.orc` *Required.* CsoundVST stores the Csound orchestra internally in its VST patch. But to perform the score, Csound must automatically export the orchestra using this filename.

`temp.sco` *Required.* CsoundVST stores the Csound score internally in its VST patch. But to perform the score, Csound must automatically export the score using this filename.

When you have created your options, you must make sure that your edits are saved with the Cubase **File** menu, **Save** command.

4.2.5. Compile the Orchestra

Before you can play an orchestra, it must be compiled. In Cubase, you activate a VST plugin by clicking on the **on/off** button (it will light up) that is found on the upper left hand corner of the VST instrument GUI, or also on the MIDI track channel settings. You de-activate the plugin by clicking again on the **on/off** button (it will go dark). When CsoundVST is activated, it exports its stored orchestra and score, compiles them, and performs them; they are then ready to receive MIDI input from Cubase. When CsoundVST is de-activated, it stops performing.

Note: when Cubase loads a song containing CsoundVST, Cubase will automatically activate CsoundVST. This can cause a delay as the orchestra compiles.

As the orchestra compiles, which normally takes a second or so, Csound will print informational messages to the message text area at the bottom of the **Settings** tab. When the messages stop scrolling, compilation is complete (Figure 4.7).

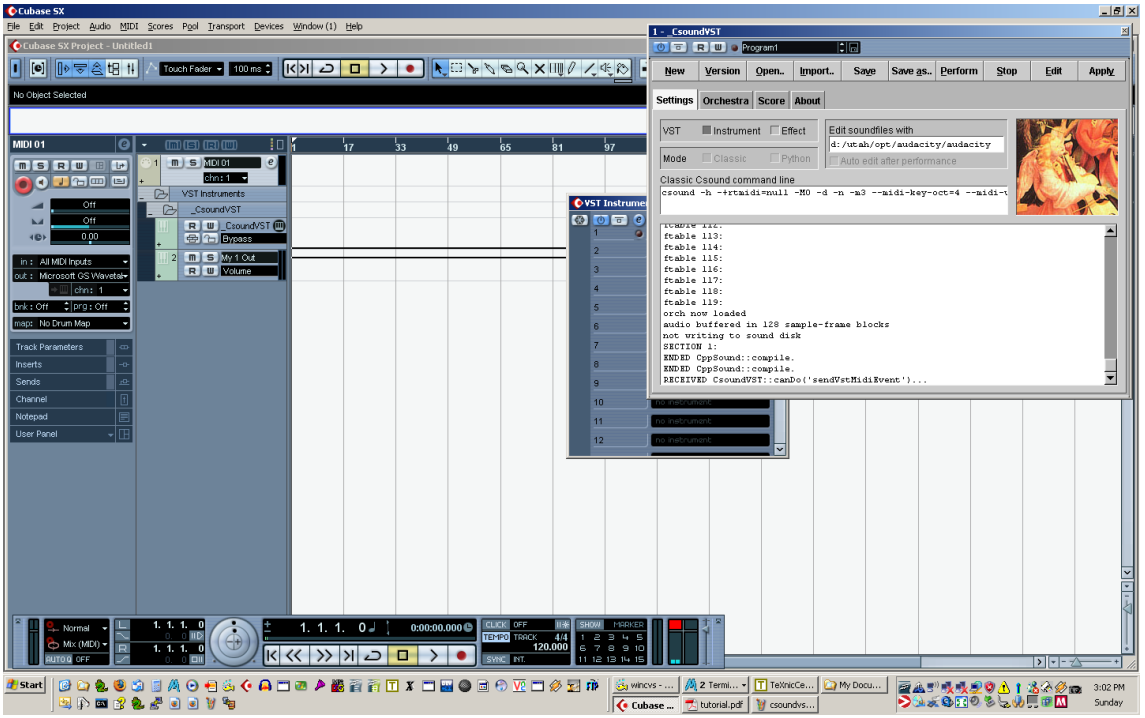

Figure 4.7.: **Compiled Orchestra**

4.2.6. Track Setup

Before you can actually get any sound out of CsoundVST, you must select it as an output in your MIDI track. In the **Track** panel, **out** field, use the left mouse button to pop up a list of available outputs. If it has been activated, `CsoundVST` should be one of these. Select it.

4.2.7. MIDI Channel Setup

Now, assign your MIDI track's channel number. The orchestra contains many more than 16 instruments, but you can assign MIDI channels to instruments numbered higher than 16 by using the `massign` statement in the Csound orchestra header.

Create a part in your MIDI track, set up a loop for the part, use the **Track** panel's **chn** field to assign your track's MIDI channel to a number between 1 and 16, inclusive.

Start recording, and play some notes on your MIDI controller. If notes begin appearing in your part, you know your MIDI interface is working. You may hear nothing at all, or you may hear a loud distorted sound. Use the VST instrument volume control to adjust the gain, if necessary (Figure 4.8). If you still hear nothing, check the Csound messages pane, and re-activate Csound if necessary.

If you make any changes to the Csound orchestra, be sure you use the Cubase **File** menu, **Save** command to save your edits. These edits are saved inside the Cubase song (`.cpr`) file, not to the Csound orchestra that you originally exported, although you can re-export the `.csd` file if you wish.

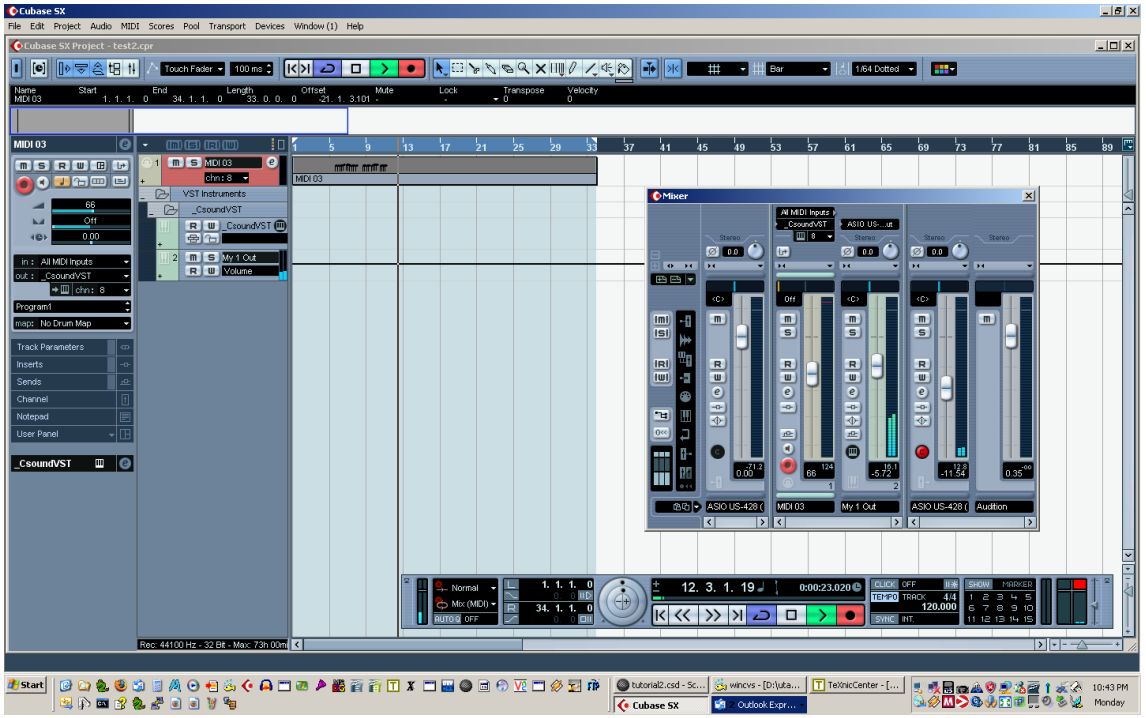

Figure 4.8.: **Channel Setup**

4.2.8. Write Some Music

There are of course many ways to write music with CsoundVST in Cubase, or any other VST-enabled audio sequencer or notation software. You can play in tracks, write music notation, type in event lists, and so on. Figure 4.9 shows CsoundVST rendering a Buxtehude fugue that has been imported from a public domain MIDI file. Note that a single instance of Csound is being used to render all 6 tracks, each of which may play 1, 2, or more voices. Each track is assigned to a different MIDI channel, which in turn is assigned to a different instrument number in the `CsoundVST.csd` orchestra.

Although in this piece the CPU load (as shown by the leftmost vertical meter on the transport bar) is light, it is easy to create instruments and effects in Csound that use a lot of CPU cycles. In such cases, you can use Cubase's own off-line rendering facility, or you can render one track at a time by soloing it and freezing it.

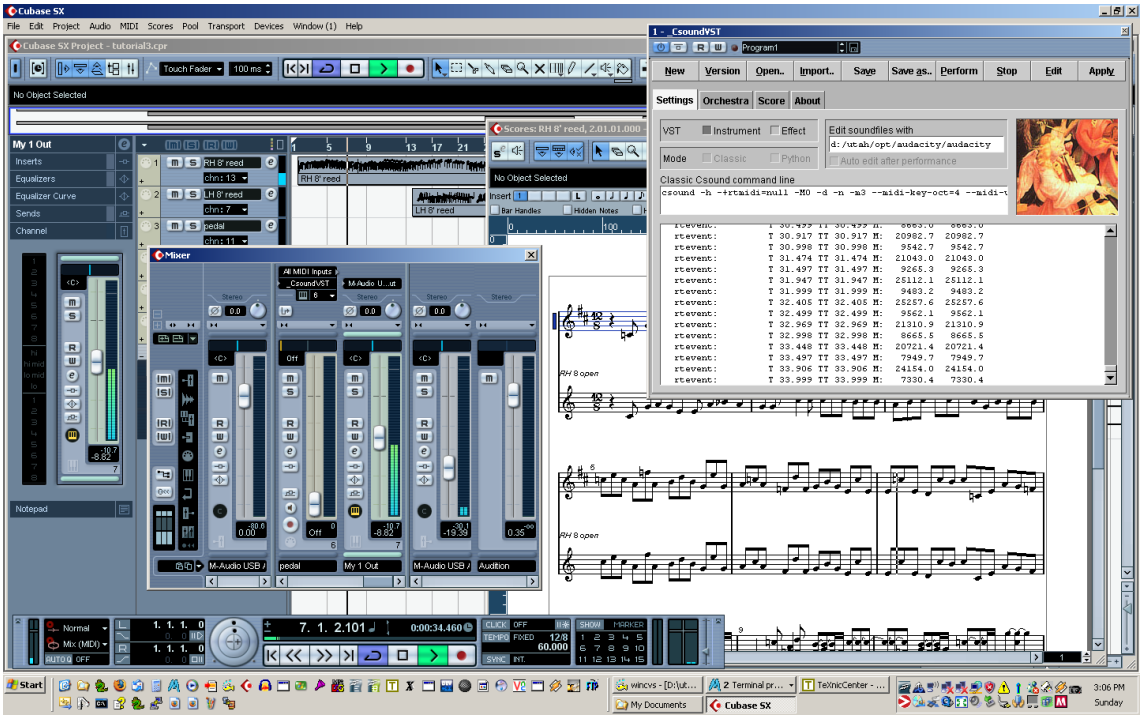

Figure 4.9.: **Scoring with Csound**

5. Python Scripting

There may be thousands or even hundreds of thousands of notes in a single piece of music. Consequently, most musicians do not compose pieces for Csound by typing in one note at a time.

The commonest way of writing Csound scores is to write programs to generate scores. This is called *generative music* or *algorithmic composition*. This, again, is a very deep subject [14, 15, 16, 17].

Of course, if you are the kind of a composer who hears music in his or her head and you just need to get the notes you hear into Csound, you can use Sibelius, export a MIDI file, and have Csound perform your MIDI file using the `--midifile` option:

```
csound --midi-key=4 --midi-velocity=5 --midifile mypiece.mid -RWZfo myrendering.wav
```

On the other hand, if you are such a composer *and* you have some facility with programming, it is probably just as easy to write snippets of code to generate runs, chord progressions, minimalist-style cells, and so on. In other words, a programming language is just another form of music notation. For some purposes, code is a better form of notation. More significantly, composing by programming opens up vast new musical possibilities:

- You can compose things that transcend the limits of your imagination.
- If you have composed something that transcends the limits of your imagination but you don't like it, you can change the code until you do like it — sometimes.
- With recursive or fractal algorithms, a single change in the code can have global effects on the piece, at every level of structure at the same time.
- You can compose things that are too tedious to notate by hand, or too precise for performers to play.

In my view, this is the outstanding reason to use Csound — it is an ideal orchestra for algorithmic composition.

Score generators have been written in many languages. But some languages can operate Csound directly. At the time of writing, these include C [3], C++ [18], Java [19], Lisp [20], Lua [21], and Python [22]. This chapter is about using Python.

Python is an open source, dynamic, high-level, object-oriented programming language with some features of functional programming. Python is widely used, and there is a huge number of libraries available for it, including libraries for scientific computing that turn out to be very useful for computer music. Of all the languages I have used, both in my career as a programmer and in my career as an algorithmic composer, Python has been by far the easiest and most productive language to learn and to use.

The remainder of this chapter assumes that you have at least some experience with Python. If not, running through the Python tutorial at the beginning of the Python manual should be enough to get you started [23].

Csound comes with not just one but two Python interfaces:

`csnd` This is a Python interface to the complete Csound API, also including facilities for loading Csound `.csd`, `.orc`, and `.sco` files, and for building up `.sco` files in memory one statement at a time — very useful for score generation.

`CsoundAC` Includes everything in `csnd`, plus my Silence system [24] for algorithmic composition based on *music graphs*, which represent scores as hierarchical structures in somewhat the same way that a ray tracer represents a visual image as a hierarchical *scene graph*.

This tutorial uses `csnd`. First we use it simply to run an existing piece — the `tutorial2.csd` piece from Chapter 3. Then we use Python to generate a piece using a *Koch curve*, in which each segment of a curve is replaced by a generator curve [25]. We use an existing Csound orchestra to render the piece we have generated. Finally, we experiment with changing the parameters of the compositional algorithm.

5.1. Running Csound from Python

1. Run Idle, the Python editor that comes with Python.
2. Create a Python file, `tutorial4.py`.
3. Import `csnd`. To verify that the import succeeded, print a directory of the `csnd` module, which should list all the API functions and constants in the module.
4. Create an instance of `csnd.CppCsound`, which is the Python interface to the high-level Csound C++ class that has facilities for managing Csound files, as well as the rest of the standard Csound API.
5. Enable Python to print Csound messages by calling `csound.setPythonMessageCallback()`.
6. Load the `tutorial2.csd` piece.
7. Set the Csound command-line options. Note that the command must be completely spelled out, as if you were entering it on the command line, including `csound` and the names of the `.orc` and `.sco` files.
8. Print out the loaded and modified `.csd` file by calling `print csound.getText()`.
9. Render the piece by calling `csound.perform()`. You should see the Csound messages printing out in the Idle **Python Shell** window.

This is illustrated in Listing 5.1 and Figure 5.1.

Listing 5.1: **Running Csound with Python**

```
# Import the Csound API extension module.

import csnd

# Print a directory of its attributes
# (variables, functions, and classes)
# to verify that it was properly imported.

print dir(csnd)

# Create an instance of Csound.

csound = csnd.CppSound()

# Enable Csound to print console messages
# to the Python console.

csound.setPythonMessageCallback()

# Load the tutorial2 piece created earlier.

csound.load('tutorial2.csd')

# Set the Csound command for off-line rendering.

csound.setCommand('csound -RWfo tutorial4.py.wav temp.orc temp.sco')

# Print the complete .csd file.

print csound.getCSD()

# Export the .orc and .sco file for performance.

csound.exportForPerformance()

# Actually run the performance.

csound.perform()
```

5.2. Generating a Score

In Csound, a score is basically a list of `i` statements, each with its own list of pfields. This tutorial has always used the same layout of pfields. This has advantages for algorithmic composition. It makes it easy to build up scores algorithmically.

A sample piece is shown in Listing 5.2. To understand what is happening, read the comments in the code.

Some of the important points are as follows. The score generator is written as a Python class, and an instance of Csound is created as a class member. After generating the score, the code appends an `e` (end) statement to the score, which turns off the reverb instrument and other effects that are running on the `CsoundVST.csd` orchestra's mixer buss with indefinite durations. The code tests to see if it is running as `__main__`, in which case a score is generated (as in this case), or whether it is running because it was imported by another module, in which case no score is generated. The other module can then initialize the generator, derive other classes from it, and otherwise use `tutorial5.py` as a class library.

Listing 5.2: **Koch Curve Score Generator**

```
import csnd
```

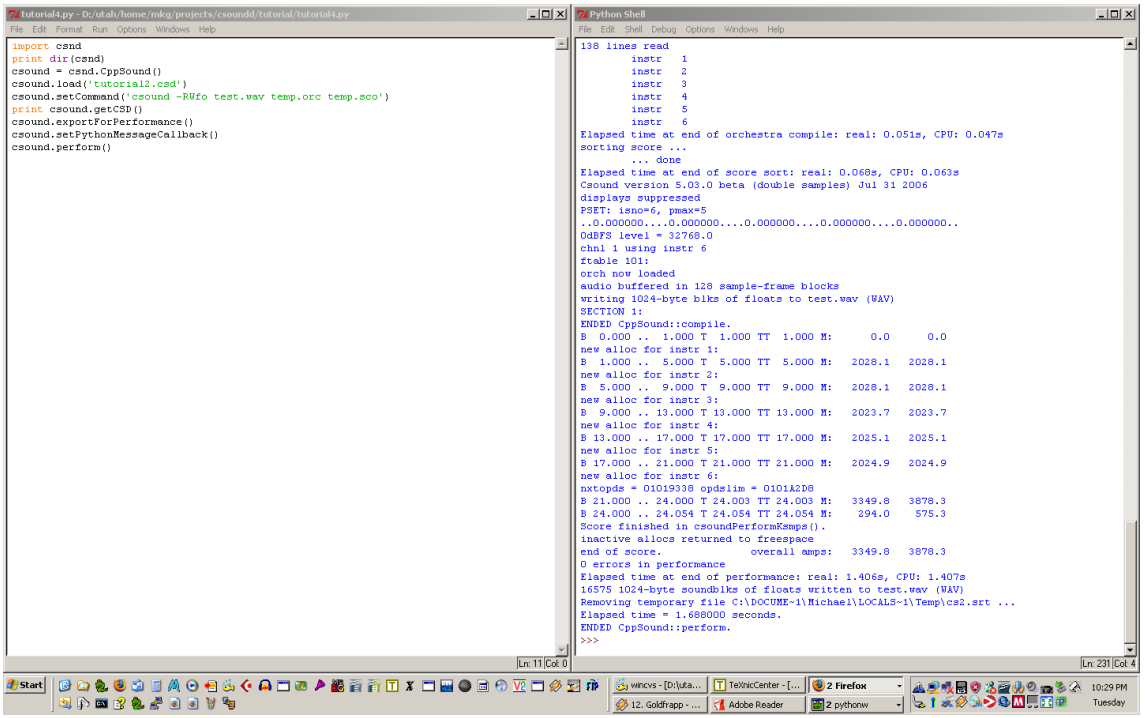

Figure 5.1.: **Running Csound with Python in Idle**

```
# Class to represent transforming a note
# by modifying an implicit initial note,
# creating a duration, adding or subtracting pitch,
# adding or subtracting loudness.

class Transform(object):
        def __init__(self, duration, deltaKey, deltaVelocity):
                self.duration = duration
                self.deltaKey = deltaKey
                self.deltaVelocity = deltaVelocity
                self.normalizedDuration = 1.0

# Class for generating a piece using a long initial note
# and a set of transforms, recursively layering atop generated notes.

class Generator(object):
        def __init__(self):
                # Create an instance of CppSound for rendering.
                self.csound = csnd.CppSound()
                self.csound.setPythonMessageCallback()
                self.csound.load('../examples/CsoundVST.csd')
                # To contain a list of transforms.
                self.transforms = []
                # Assign instruments to levels (level:instrument)
                self.arrangement = {0:12, 1:4, 2:21, 3:7, 4:37}
                # Assign gains to levels (level:gain)
                self.gains = {0:1.5, 1:1.25, 2:1, 3:1, 4:1}
                # Assign pans to levels (level:pan)
                self.pans = {0:0, 1:-.75, 2:.75, 3:-.5, 4:.5, 5:-.25, 6:.25}
        def addTransform(self, deltaTime, deltaKey, deltaVelocity):
                self.transforms.append(Transform(deltaTime, deltaKey, deltaVelocity
                    ))
                self.normalize()
        def normalize(self):
                sum = 0.0
                for transform in self.transforms:
```

```
                        sum = sum + transform.duration
                for transform in self.transforms:
                        transform.normalizedDuration = transform.duration / sum
        # Generate a score in the form of a Koch curve.
        # Each note in a generating motive
        # will get a smaller copy of the motive nested atop it,
        # and so on.
        def generate(self, level, levels, initialTime, totalDuration, initialKey,
            initialVelocity):
                # If the bottom level has already been reached,
                # return without further recursion.
                if level >= levels:
                        return
                time = initialTime
                key = initialKey
                velocity = initialVelocity
                for transform in self.transforms:
                        instrument = self.arrangement[level]
                        duration = totalDuration * transform.normalizedDuration
                        key = key + transform.deltaKey
                        velocity = velocity + (transform.deltaVelocity * self.gains
                            [level])
                        phase = 0
                        pan = self.pans[level]
                        print "%2d: %2d %9.3f %9.3f %9.3f %9.3f %9.3f %9.3f" % (
                            level, instrument, time, duration, key, velocity, phase
                            , pan)
                        self.csound.addNote(instrument, time, duration, key,
                            velocity, pan)
                        # Recurse to the next level.
                        self.generate(level + 1, levels, time, duration, key,
                            velocity)
                        time = time + duration
        # Render the generated score.
        def render(self):
                # Ends indefinitely playing effects on the mixer buss.
                self.csound.addScoreLine("e 2")
                # Print the generated score for diagnostic purposes.
                print self.csound.getScore()
                # High-resolution rendering.
                self.csound.setCommand('csound -R -W -Z -f -r 88200 -k 88200 -o
                    tutorial5.py.wav temp.orc temp.sco')
                self.csound.exportForPerformance()
                self.csound.perform()
                self.csound.removeScore()

# If running stand-alone, generate a piece;
# if imported by another module, do not generate a piece
# (enables the Generator class to be used as a library).

if __name__ == '__main__':

        # Create a generator with four notes
        # in the same interval relationship as B, A, C, H,
        # i.e. Bb, A, C, B,
        # i.e. 0, -1, +3, -1,
        # offset by a tritone.

        generator = Generator()
        generator.addTransform(10, 6 + 0,  0)
        generator.addTransform( 8,   - 1,  3)
        generator.addTransform( 6,   + 3, -2)
        generator.addTransform(12,   - 1,  0)

        # Generate a 5 minute piece.
        generator.generate(0, 3, 0, 300, 38, 84)
        generator.render()
```

Now run the piece. I find that SciTE [12] actually makes a better environment for

Python programming with Csound than Idle (as long as I don't have to do source-level debugging, which SciTE doesn't support), because if you kill Csound while it is running from Idle, Csound often keeps running anyway as a zombie process, whereas if you kill Csound while it is running from SciTE, it really dies and you can start it again. You can use the **Tools** menu, **Go** command to run Python on the currently edited `.py` file, and you can use the **Tools** menu, **Stop Executing** command to stop Python. Figure 5.2 shows SciTE running the **tutorial5.py** piece.

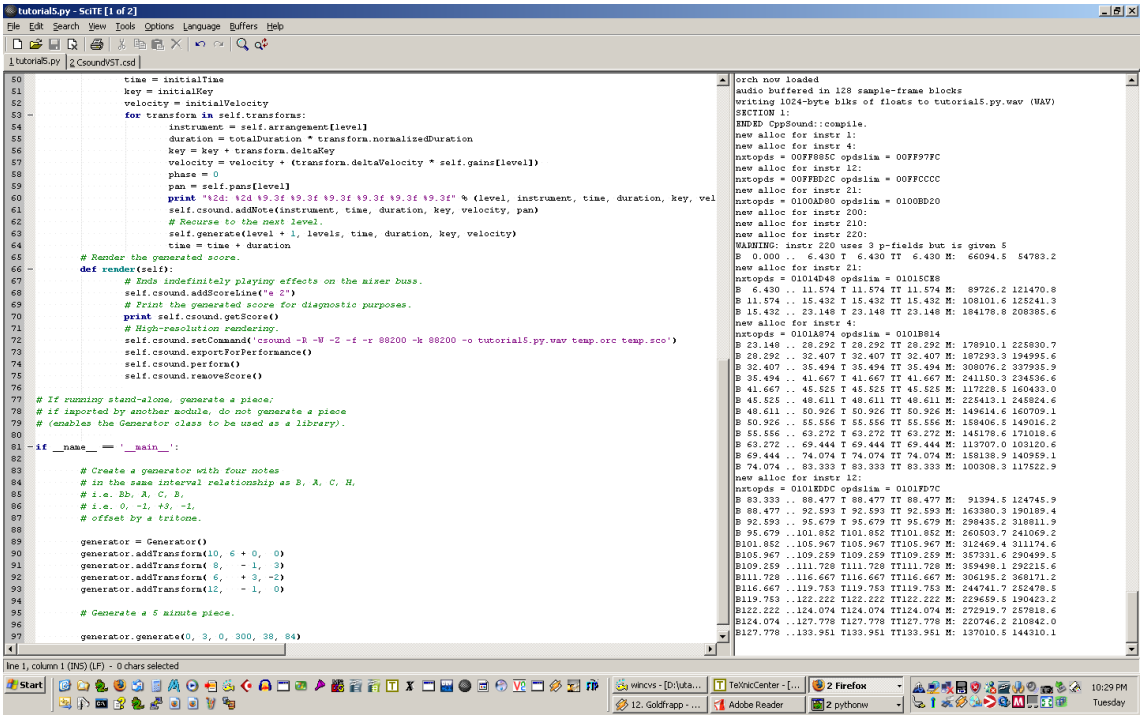

Figure 5.2.: **Running Csound with Python in SciTE**

Note also that you can load a Csound orchestra file (`CsoundVST.csd` in this case) into SciTE at the same time as you are editing or running a Python script.

5.3. Varying the Parameters

Once you have rendered this piece, you can experiment with changing the numbers inside the generators, adding and removing segments from the generators, trying more layers, and so on.

For example, try just the following changes: change the number of levels from 3 to 4, and change the second transform's MIDI key movement from -1 to +1. You will see what two small changes do the overall structure of the piece.

A. Extra Features and Their Requirements

If you wish to use any of these extra features, you should install the other required software *first* according to its standard instructions.

ATS opcodes ATS is a library of C and Lisp functions for spectral Analysis, Transformation, and Synthesis of sound based on a sinusoidal plus critical-band noise model. A modeled sound in ATS can be sculpted using a variety of transformation functions. The ATS opcodes in Csound use these transformations, but to use the opcodes, you must install ATS and analyze some sounds [26].

`csoundapi~` is an external enabling Csound to run inside Pure Data, another SWSS. To use it, you must install Pure Data [27].

tclcsound is a GUI front end for Csound that use the Tcl/Tk scripting environment. To use tclcsound, you must install Tcl/Tk [28].

VST hosting opcodes enable Csound to use external VST plugins as opcodes. To use them, of course, you must acquire them.

Java API To use this, you must install the Java software development kit (SDK) [19].

Lisp API To use this, you must install the Lisp programming language [20].

Lua API To use this, you do not need to install the Lua programming language — it comes in the Windows installer as `luajit.exe`! But, if you do use the Lua API, you may to install various Lua libraries and helpers that you can find starting at `http://www.lua.org`.

Python API to use this, you must install the Python programming language, specifically version 2.4 [22].

B. Helper Applications

The following is a highly selective subset of the various applications that the Csound community has found helpful for working with Csound. All are cross-platform and should work, at a minimum, on both Windows and Linux. All are freely available, open source applications.

B.1. Audio Editors

You can play Csound files using the media player that comes with your operating system, but a dedicated audio editor is much more useful. It will enable you to see your soundfiles, edit out clicks, normalize amplitudes, and more.

B.1.1. Audacity

Audacity [29] is the most powerful freely available, cross-platform audio editor. Get it.

B.2. Text Editors

You can edit Csound scores and orchestras with a word processor, but you should find a real programmer's editor much more useful. Each of the following has add-ons for working with Csound files.

B.2.1. Emacs

Emacs [30] has been widely used as a programmer's editor for decades. It has various Csound environments.

B.2.2. SciTE

Not as powerful as Emacs, more user-friendly than vi. SciTE [12] is the editor that I most often use with Csound. You can get a Csound syntax coloring package for SciTE, which can run both Csound and Python from its own shell.

B.3. Composing Environments

A variety of specialized music composition environments have been developed, either specifically for Csound, or that can work with Csound. These are mainly intended for art music and algorithmic composition.

B.3.1. athenaCL

Christopher Ariza's athenaCL [31] is a powerful Python-based composing environment that is designed to work with Csound, and which has incorporated within itself many facilities from other earlier composition software. It is designed to be used as an interactive command-line shell, but can also be used as a Python class library.

B.3.2. Blue

Steven Yi's Blue [32], written in Java, provides a visual composing environment for Csound based on time lines. Blue can also run Python scripts.

B.3.3. CsoundAC

CsoundAC, by me, contains a set of classes that implement my idea of music graphs [24]. It is the only composing environment that is distributed with Csound. The Windows installer for Csound comes with `luajit`, a runtime interpreter for the Lua language that has a built-in just-in-time compiler. So, if you want to compose by writing Lua code, you don't need to install anything! But if you want to use CsoundAC from Python, you must install Python before you install Csound. You can write CsoundAC programs using either a Python development environment, or from a Python-aware text editor. I use either the default Python IDE (IDLE), or SciTE, as my main composing environment.

B.3.4. Common Music

Rick Taube's Common Music [33] is a very powerful Lisp-based programming language dedicated to algorithmic composition. It contains facilities for automatically generating Csound scores.

B.3.5. Pure Data

Miller Puckette's Pure Data [27] is, itself, a widely used SWSS. However, it is also used as a composing environment, and it contains a `csoundapi~` external that can receive events from Pure Data, route them to Csound using the Csound API, and feed audio or events from Csound back into Pure Data.

B.4. Programming Languages

The following programming languages can use Csound through the Csound API. Such languages are especially useful for algorithmic composition.

B.4.1. C/C++

C [3] and C++ [18] are still the standard programming languages for "systems programming," i.e. writing the fastest, most complex, and most demanding software.

Most operating systems are written in C, and most commercial applications are written in C++ or C. You can use Csound as a "synthesis engine" in your own C and C++ applications by using the Csound C or C++ APIs, and linking with the Csound library.

B.4.2. Java

Java [19] is another widely used language. It is only about a third as efficient as C or C++, but it is somewhat easier to program. The Csound API has a Java interface.

B.4.3. Lisp

Lisp [20] is the second-oldest (after FORTRAN) high-level programming language. It is particularly noteworthy for being the implementation language for Common Music, an excellent algorithmic composition system that is designed to work with Csound.

B.4.4. Lua

Lua [21] is a lightweight, interpreted high-level language. As it is relatively new, it features a good balance of features from earlier languages. On Windows, there is a just-in-time compiler for Lua that can run Lua programs as fast as compiled Java code (i.e., about 1/3 as fast as C or C++). The Windows installers for Csound actually installs not only the Lua interface to the Csound API, but also the Lua just-in-time compiler itself.

B.4.5. Python

Python [22] is my favorite programming language for working with Csound. I find it is easier to read and write than other programming languages, and it has very extensive libraries, e.g. for scientific computing and for computer graphics.

Although Python is an interpreted language and therefore does not run fast, Python can call into precompiled extensions written in C or other efficient languages. The Csound API's Python interfaces are themselves examples of such extensions.

C. Audio Quality

Currently, studio recording is done to stereo or surround sound (5.1 or 7.1) on computers, hard disk recorders, or professional digital audio tape (DAT) recorders to 24-bit or floating-point samples at a rate of 48,000, 88,200, 96,000 or even 192,000 sample frames per second. This is "high-resolution audio." At this time, the only digital consumer electronics formats that can reproduce high-resolution audio are DVD-A and SACD.

CD-quality audio is of distinctly lower resolution: stereo sound with 16 bit integer samples at 44,100 samples per second.

High-resolution audio, on good speakers or earphones, sounds airy, present, spacious, and undistorted. CD-quality audio, by contrast, can sound flat, shrill, harsh, and flat or boxed in. Usually, this is the result of cumulative mistakes made in this less forgiving medium – CDs actually are precise enough to reproduce most of what we hear. Therefore, CDs made by experts can sound very good indeed, except for their more limited dynamic range and less detailed quiet sounds. Normally, it takes educated ears to hear these differences.

Vinyl records of high quality are not directly comparable to digital recordings. They have their own virtues and flaws. They are more detailed, airy, and spacious than CDs, but can have harmonic distortion, rumbling, hiss, and crackling. In general, well-made records, especially if pressed from direct metal masters, are roughly equal to high-resolution audio in aesthetic quality, even if they are not really as precise.

If you are not used to high-resolution audio, you will need to educate your hearing before you can achieve it (or even hear it). Develop your ears by listening critically to outstanding work on flat, deep, high-resolution audio systems, e.g. real studio monitor speakers or good headphones, at loud but not overwhelming volume in a quiet, sound-adsorbent environment. Listen to your own work in direct comparison. Learn to be objective and to set your own feelings aside, and to hear what others say about your work without becoming defensive.

Listen to live orchestral and chamber music, and big-band jazz, from good ensembles, in good halls, from good seats. This is the gold standard for sound – even the best high-resolution audio can't touch it. Also listen to outstanding recordings of orchestral, chamber, piano, rock, folk, jazz, New Age, film music (and again, film music) and of course computer music. For computer music, listen to academic computer music, EA, "dance music", mods and demos, and even chip tunes. Each of these genres has something valuable to say about audio beauty and music production quality that is relevant to computer music.

Csound is eminently capable of high-resolution audio. It can render to *any* number of channels, at *any* sampling rate, using floating-point samples. Csound also contains high-quality software implementations of all the effects applied by mastering

engineers. Therefore, Csound is as good or better than the best studio gear.

If you have a professional or semi-professional audio interface on your computer, you can play high-resolution soundfiles made with Csound (although you will not hear their full dynamic range unless you have professional gear).

Specific technical advice in decreasing order of importance (all this assumes you don't care how long it takes to render a piece, only if it sounds good):

1. Some of the sounds made by Csound have no counterpart in other kinds of music. They may contain excessive high frequencies, aliasing distortion, or other kinds of noise. On the other hand, the sounds can be of extreme clarity and precision — *hyper-real*. You need to be constantly aware of what your sounds *actually sound like*.

2. Always render to floating-point soundfiles at 88,200 samples per second. You can translate them to 24 bits or to CD quality later if you want, but having the extra precision and dynamic range is vital. There is no audible difference in quality between 88,200 and 96,000 samples per second, but 88,200 can translated to CD quality by direct downsampling, whereas 96,000 requires fancy filtering and lots of time.

3. If you use sampled sounds, use the best samples you can possibly find. Pay if you must!

4. Also if you use sampled sounds, beware of their own ambience clashing with any reverberation or other ambience you set up using Csound. Samples may also have unwanted noise — it may be possible to de-noise them (Csound has facilities for doing this too).

5. Use a "de-clicking" envelope to wrap all your instrument final output signals.

6. Watch out for aliasing, which can make sounds buzzy or harsh, in frequency modulation and wavetable oscillators. Aliasing happens when the signal contains frequencies above half the sampling rate (the Nyquist frequency), so that under digital sampling they reflect or fold back under the Nyquist frequency. For so-called "analog" sounds with simple waveforms such as square or sawtooth waves, use non-aliasing opcodes such as `vco` or `vco2`. You do not need to worry about aliasing with plain sine or cosine waves.

7. For final renderings, always render with `ksmps=1`.

8. Use a-rate variables for envelopes and, in general, wherever opcodes permit. This enables decent results with `ksmps=100` or so.

9. Use only the most precise interpolating oscillators, such as `poscil` or `poscil3`.

10. For wavetable oscillators, the larger the wavetable, the less noisy the signal; 65537 is not too big.

11. Be vigilant for artifacts and noise introduced by various digital signal processing algorithms, especially echoes in reverberation. Don't over-use effects – this is a very common error that you can fix by listening to good examples of studio and live recording.

12. Experiment with some modest compression, e.g. by using the `compress` or `dam` opcodes.

13. Use the 64-bit sample version of Csound.

14. Do not use the dither (`-Z` option). There is no need to dither if you are rendering to a floating-point medium. Furthermore, the dither option in Csound currently does nothing. If you do need to render to a fixed-point (i.e. integer) medium, such as CD audio, keep your work in high-resolution format as long as possible. When the piece is completely finished, use a soundfile editor or a conversion program with a dither effect to convert the soundfile to the fixed-point format.

Bibliography

[1] Barry Vercoe, John ffitch, Istvan Varga, Michael Gogins, et al. Csound. http://csound.sourceforge.net. 1

[2] Max Mathews. *The Technology of Computer Music*. The MIT Press, Cambridge, Massachusetts, 1969. 1

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentiss-Hall, 2 edition, 1988. 1, 47, 56

[4] Richard Boulanger, editor. *The Csound Book*. The MIT Press, Cambridge, Massachusetts, 2000. 1, 35

[5] Riccardo Bianchini and Alessandro Cipiriani. *Virtual Sound: Sound Synthesis and Signal Processing — Theory and Practice with Csound*. ConTempo, Rome, 1998. English edition (2000), translated by Agostino Di Scipio. 1, 35

[6] Andrew Horner and Lydia Ayers. *Cooking with Csound Part 1: Woodwind and Brass Recipes*. A-R Editions, Middletown, Wisconsin, 2002. 1, 35

[7] James McCartney et al. SuperCollider, 2004. http://supercollider.sourceforge.net. 1

[8] Cycling 74. Max/MSP. http://www.cycling74.com/products/maxmsp.html. 1

[9] Native Instruments. Reaktor 4. http://www.native-instruments.com. 1

[10] Barry Vercoe, John ffitch, et al. The Canonical Csound Reference Manual, 2006. http://www.csounds.com/manual/html/indexframes.html. 1, 10, 11

[11] Michael Gogins. Double Blind Listening Tests of Csound 5 Compiled with Single-Precision and Double-Precision Samples, 2006. http://ruccas.org/pub/Gogins/csoundabx.pdf. 3

[12] SciTE: A Free Source Code Editor for Win32 and X, 2006. http://www.scintilla.org/SciTE.html. 9, 51, 55

[13] Steinberg Media Technologies GmbH. http://www.steinberg.net. 37

[14] Lejaren Hiller and L.M. Isaacson, editors. *Experimental Music: Composition with an Electronic Computer*. McGraw–Hill, New York, New York, 1959. 47

[15] Tom Johnson. *Self-Similar Melodies*. Editions 75, Paris, 1996. 47

[16] David Cope. *The Algorithmic Composer*. Number 16 in Computer Music and Digital Audio. A-R Editions, Middleton, Wisconsin, 2000. 47

[17] Heinrich K. Taube. Notes from the Metalevel. http://pinhead.music.uiuc.edu/~hkt/nm/. 47

[18] Bjarne Stroustrup. The C++ Programming Language, 2006. http://www.research.att.com/ bs/C++.html. 47, 56

[19] Sun Developer Network. The Source for Java Developers, 2006. http://java.sun.com. 47, 53, 57

[20] Association of Lisp Users, 2006. http://www.lisp.org/alu/home. 47, 53, 57

[21] Robert Ierusalemichy, Waldemar Celes, and Luiz Henrique de Figueirido. The Programming Language Lua, 2006. http://www.lua.org. 47, 57

[22] Guido van Rossum. Python, 2006. http://www.python.org. 47, 53, 57

[23] Guido van Rossum and Jr. (Ed.) Fred L. Drake. Python Tutorial, 2006. http://docs.python.org/tut/tut.html. 48

[24] Michael Gogins. Music Graphs for Algorithmic Composition and Synthesis with an Extensible Implementation in Java. In Mary Simoni, editor, *Proceedings of the 1998 International Computer Music Conference*, pages 369–376, San Francisco, California, 1998. International Computer Music Association. 48, 56

[25] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. Chaos and fractals: New frontiers of science. In *Chaos and Fractals: New Frontiers of Science*, chapter 5, pages 229–296. Springer-Verlag, 1992. 48

[26] Juan Pampin, Oscar Pablo Di Liscia, Pete Moss, and Alex Norman. Analysis – transformation – synthesis (ats), 2006. http://www.dxarts.washington.edu/ats/. 53

[27] Miller Puckette. Pure Data. http://puredata.info. 53, 56

[28] Tcl Developer Xchange, 2006. http://www.tcl.tk. 53

[29] Audacity. http://audacity.sourceforge.net. 55

[30] Richard W. Stallman et al. GNU Emacs, 2006. http://www.gnu.org/software/emacs/. 55

[31] Christopher Ariza. athenaCL. http://www.flexatone.net/athena.html. 56

[32] Steven Yi. blue. http://csounds.com/stevenyi/blue. 56

[33] Rick Taube. Common Music. http://commonmusic.sourceforge.net/doc. 56