Blue

a music composition environment for Csound

Steven Yi

Blue: a music composition environment for Csound

by Steven Yi

2.7.2_beta

Published 2017.10.12

Abstract

Blue is a music composition environment for use with Csound. It is written in Java. The homepage for Blue is available at http://blue.kunstmusik.com.

Table of Contents

List of Tables

Preface

Blue is my tool for composing with Csound. I originally created it for its ability to organize SoundObjects in time as well as to organize and automate the more mundane tasks that I've found in working with Csound. It's here to make working with Csound a more enjoyable and focused experience.

As the program develops over time I am finding more ways to make the user experience intuitive and the program faster to work with, as well as finding interesting new things to create to further the capacity for musical expression. The program's architecture is designed to be as open as possible in both things the user can do within Blue itself as well as the ability to expand Blue through the creation of plugins, and by taking advantage of both one can find many ways to create and shape musical ideas.

Beyond my own ideas on where to go with Blue, user feedback has also greatly shaped the direction and richness of the program. Please feel free to email me anytime with feedback, or join the Blue mailing list and discuss your ideas and thoughts there.

If you'd like to find out more about me you can do so on my on my website:

http://www.kunstmusik.com

Thanks and I hope you enjoy composing with Blue!

Steven Yi

Credits

I would like to thank the following people following people for their contributions to blue:

Michael Bechard	Java development
Andrés Cabrera	Documentation, Tutorials, and Ideas
Dave Seidel	Java development
Kevin Welsh	PHP work on blueShare server, Java code

Part I. User's Manual

This section is for users

Chapter 1. Getting Started

Introduction

About Blue

Blue is a music composition environment for use with Csound. Blue interacts with Csound by generating .CSD files, which it then feeds to Csound for compilation. Any version of Csound that can be called by commandline is able to be used with Blue.

One of Blue's main features is the graphical timeline of SoundObjects, of which a special SoundObject-the polyObject-is able to group other SoundObjects. polyObjects feature their own timeline, allowing for timelines within timelines. SoundObjects can be anything that generates notes, be it a block of Csound SCO, a script in python, or a graphical object.

SoundObjects may further have noteProcessors added to them, which may do things like "add .3 to all p4 values" or "of these notes, only generate the first three". NoteProcessors are especially useful when used with Instance SoundObjects from the SoundObject Library.

The SoundObject library allows for working with SoundObjects by making instances of a SoundObject. Instances point to a SoundObject, and when they generate notes, they use the SoundObject they are pointing to to generate the notes for them. Instances however can have their own properties which they will apply to the generated notes, so that they can be of different duration and have different start times, as well as have different noteProcessors used with them. the advante of instances versus manually copying a SoundObject over and over again is that if you want to change the notes for all of these SoundObjects, you'd have to manually go and change all of them, while with instances, you'd only have to change the one SoundObject all of the instances are pointing to. (which also means you could make a song template file, all pointing to empty SoundObjects, build the form of the piece, then change the SoundObject's note material to "fill in the form".)

Other features include the orchestra manager, which allows for importing of instruments from .CSD files, a list to manage instruments, and the ability to selectively generate instruments when compiling .CSD files. instruments can be either a single Csound instrument or may be a GUI instrument.

Installation

Requirements

Blue requires a Java 8 (also known as 1.8) or greater JVM (Java Virtual Machine) to be installed on your system, as well as Csound. Java installations usually come in two flavors: a JRE (Java Runtime Environment) which contains just a JVM for running applications, and a JDK (Java Development Kit) which contains the JVM as well as development tools. Because Blue comes with scripting tools that use JDK features, it is recommended to install the full JDK, and not just the JRE.

To test to see if you have a JVM installed and what version, at a command prompt type "java -version". If you see something along the lines of "command not found" then you need to install a Java Virtual Machine.

If you do not have Java installed, you can install the Oracle JDK available at:

http://java.oracle.com

On this page, look for the download link for Java SE and choose the JDK download for your platform. Be sure to choose the version of JDK that matches your CPU type and Csound version (choose 64-bit if in doubt).

Blue also works with OpenJDK versions of Java. These are pure open-source versions of Java that are often available through package managers on Linux. If you do use an OpenJDK version of Java, be sure to install JavaFX, a UI toolkit for Java that is often packaged separately from the main OpenJDK package. (Installing the Oracle JDK will include JavaFX automatically.)

Debian/Ubuntu Notes

If you are using Debian, Ubuntu, or a Linux distribution based upon one of those, be sure that you have JavaFX installed with Java. (If you hava Java installed but not JavaFX, you will find that Blue will open but an error happens during startup and a blank screen shows.) If you have OpenJDK Java installed, you can try running the command "sudo apt install openjfx" to install JavaFX and then try running Blue.

If that does not work, you should try installing Oracle Java. This can be done by using apt together with a PPA. Complete instructions are given here [http://www.webupd8.org/2012/09/install-oracle-java-8-in-ubuntu-via-ppa.html].

Installing Blue

To install Blue, you should download the latest ZIP file or DMG from the Blue releases page here [https://github.com/kunstmusik/blue/releases]. For OSX users, download the DMG file which contains a Blue.app. You can double-click the Application to run, as well as copy it to your Applications folder to install it.

For Linux and Windows users, download the ZIP file and unzip it. Inside of the bin folder you will see a "blue" script for Linux or a "blue64.exe" file for use on Windows. (A 32-bit version, "blue.exe", is also included for those using an older version of Csound that is compiled for 32-bit Windows.)

Note: After starting Blue, you may want to explore the example projects and pieces found in the Blue/example folder (or right-click and explore contents of Blue.app/example if on OSX).

Platform Specific Notes

The section below has notes for individual platforms.

Mac OSX

Blue uses the right mouse click often to show popup menus. If you do not have a right mouse button, you can use ctrl-click for all "rt-clicks" that are mentioned in this documentation.

To make use of the function key shortcuts (F1-F12), you will need to go into System Preferences, choose Keyboard, then enable "Use all F1, F2, etc. keys as standard function keys".

Linux

For 64-bit systems, you may run into issues when running Blue with the API enabled where modifying widget values is not reflected in performance. This is likely due to the Csound Java API being compiled with SWIG < 2.0. For example, some older versions of Csound in Debian for amd64 is compiled with SWIG 1.3.0 and does not work with Blue. To work around this, you can install a newer version of Csound if available (i.e. from a testing repo), or compile Csound yourself and ensure you are using SWIG version 2.0 or greater.

Installing Csound

Blue is able to interact with Csound either by calling Csound like a command-line program (classic Blue), or by directly interacting with Csound via the Csound API. Instructions on setting up Blue for each method is described below as well as discussion on benefits and limitations.

Using Blue with command-line Csound

This may be considered "classical" Blue usage and interaction with Csound as this was the method by which Blue ran with Csound for the first eight years in existance. The way Blue operates in this mode is by rendering the .Blue project into a temporary CSD file on disk, then calling Csound with that temporary CSD file in the same way as if you were on a command-line shell and executing Csound yourself with that file.

The benefit to this mode of Csound usage is that it is easier to switch out your version of Csound or use multiple versions of Csound on the same computer. It is also a little more stable than using the API in that if Csound crashes for some reason, it won't take down Blue with it. Also, it may be more performant to use the command-line mode. These benefits however need to be weighed against the benefits of using the API, which is described below.

To use the command-line version, one needs to set up the Csound executable option for Realtime and Disk Render settings in Program Options.

Using Blue with the Csound API

Enabling Blue to use the Csound API when rendering with Csound opens up the ability to manipulate and edit widget values and automations in realtime during a render, as well as the ability to use BlueLive on Windows. Because of its enhancement to the user-experience while rendering and composing, it is now the preferred method of using Blue with Csound. Blue should work out-of-the-box with the API if Csound is installed using the installers provided on Github, or installed via a package manager if on Linux.

Note

Blue currently only works with the API if the version of Csound used is compiled using 64-bit doubles. (The float version is not currently supported when using the API.) There are technical difficulties in supporting two different versions of Csound API in the same build and it is not known if or when the float build will be supported. For users interested in using the float build of Csound with Blue, you will need to run Blue using the command-line Csound mode.

Additionally, the architecture that Csound is compiled for must match the architecture of the Java runtime you are using. For example, on Windows, Csound is currently only built for x86_64/amd64 CPU (i.e. 64-bit Windows) and not x86 (i.e. 32-bit Windows). In this case, you will need to run Blue using a 64-bit Java Runtime. For OSX, this is not an issue as Csound is compiled as a universal binary for both i386 and x86_64. On Linux, it is likely that the version of Csound you install/compile and the Java Runtime that you install will likely be the same, but if the API does not show as available it may be something to check.

The Java API for Csound is split into two parts: the csnd6.jar file as well as the lib_jcsound6.so native library (this file is called _jcsound6.dll on Windows, and lib_jcsound6.jnilib on Mac OSX). Blue comes with its own copy of csnd6.jar; to use the API from Blue it will need to have access to the native library to work. If the API is not enabled for use out-of-the-box, the following explains how to setup the API on different platforms.

Windows

Users using the Windows Installer for Csound should use the double precision version from Github. (This is the default.) After installing, the installer should setup everything such that Blue should work with the API. If for some reason it is unable to do so, or you have compiled Csound yourself and the location of jcsound6.dll is different from where it is installed with the installer, you can modify the Blue/etc/Blue.conf file to tell Blue where to find the _jcsound6.dll. For example, if the directory where jcsound6.dll is located is in c:\myCsound, open up Blue/etc/Blue.conf and modify the default so that it contains:

```
default_options="--branding Blue -J-Xms256m -J-Xmx768m -J-Djava.library.path=c:/my
```

Linux

Linux users should install a doubles version of Csound. The version of Csound found in package repositories should be one compiled for doubles. After installing Csound and the Java interface for Csound, locate where lib_jcsound6.so is and modify the Blue/bin/Blue file. Search for the lines that contain "-J-Djava.library.path=/usr/lib/jni" and modify /usr/lib/jni (the default for Debian/Ubuntu-based systems) to the directory where lib_jcsound.so is located.

Mac OSX

Mac OSX users should use the installer for Csound from Github. The installer should install both the float and doubles version of Csound. The lib_jcsound.jnilib will be installed into /Library/Java/Extensions. If you are compiling your own version of Csound, you can remove the symlink in /Library/Java/Extensions and either symlink your your version there or copy it into that folder.

Checking the API is Enabled

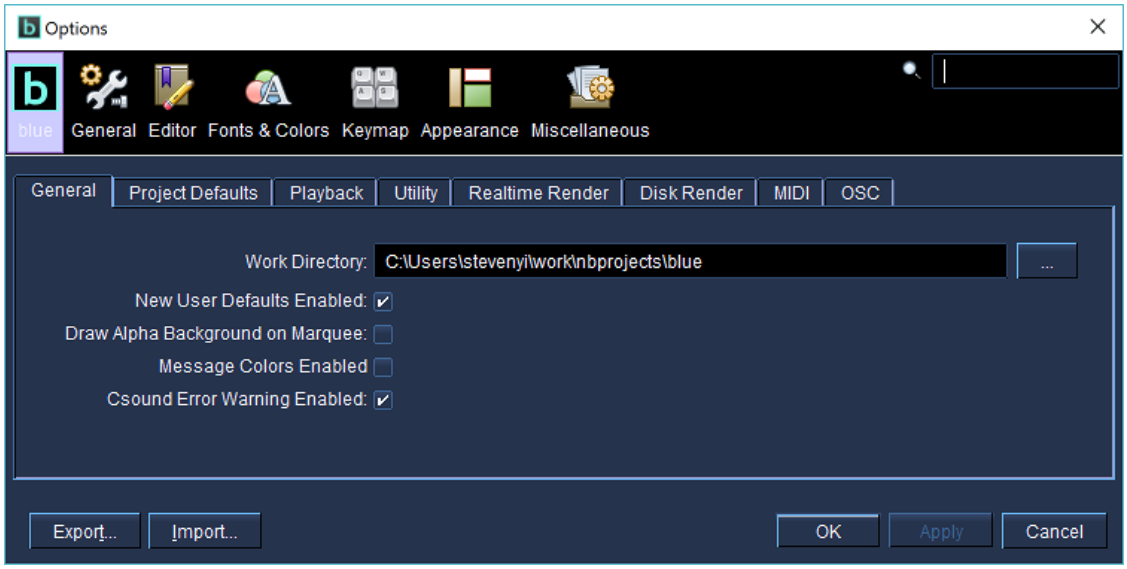
To check if the API is enabled, open Blue and open up the Program Options. This is available from the Blue->Preferences menu option on OSX, and from the Tools->Options menu option Windows and Linux. In the Blue tab, under both Disk and Realtime Render settings, there is an option called "Render Method". If Csound 6 is availble on your system and Blue was able to find it, it will show a "Csound 6 API" option. If the Csound 6 Java API could be loaded, you will only have the "Commandline Runner" option which is always available.

Setting Up Program Options

The first thing you'll want to do is set up your Program Options. Some settings are required for features of blue to work (i.e. Render Settings, the Csound Documentation Root for opening opcode help), others are there to help setup your project defaults and save you time when creating new projects.

To open up the Program Options dialog, go to the File menu and click on "Program Options".

General



General Options

Csound Documentation Root

This is the url of the root of your Csound Documentation. blue uses this for opening up Csound Documentation as well as when opening up help for an opcode. The Csound Documentation Root can be local (i.e. "file:///home/user/blue/manual/" or "file://c:/Programs and Files/blue/manual/") or online (i.e. "http://www.csounds.com/manual/html/").

You can get the Csound html documentation from http://www.csounds.com/manual .

Work Directory

The default directory that blue should open when loading/saving blue project files.

Maintain Last Window State

Enables saving of window location and size. If enabled, upon starting of program, window states will be restored.

New User Defaults Enabled

Enables default text for different objects in blue. Curently this affects:

- CodeRepository - when adding new Code Snippet

Draw Flat SoundObject Borders

Enables older flat drawing style of SoundObject borders on the score timeline.

Draw Alpha Background on Marquee

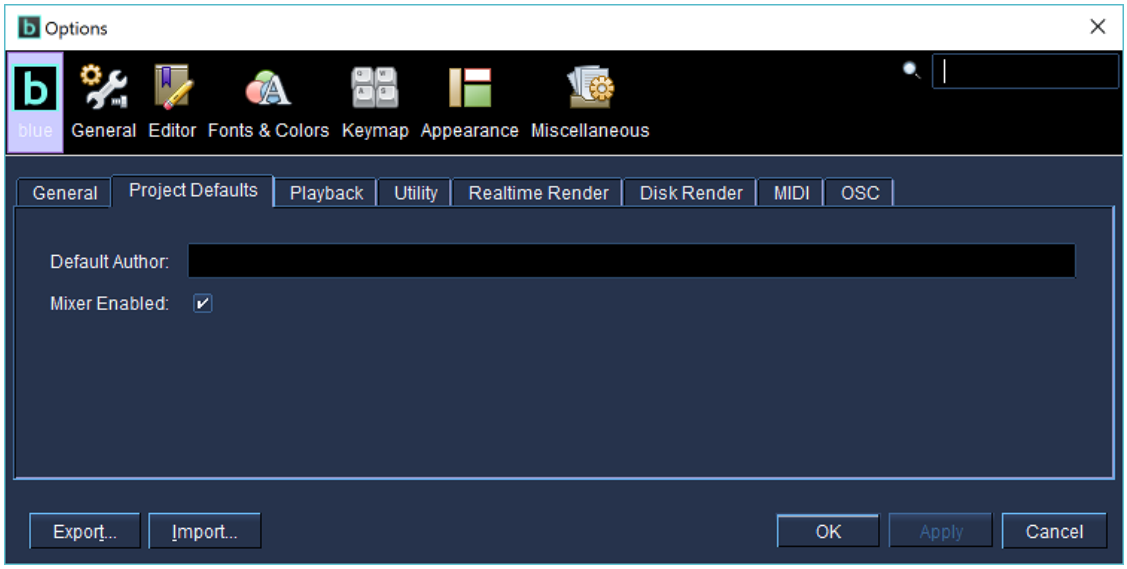
If this is enabled, when selecting a group of SoundObjects on the Score Timeline the marquee that is drawn to show the selection area will paint with an alpha-transparent white background. (May slow down performance when enabled.)

Show Csound Output

If this is enabled, output from rendering with Csound will be shown in the Csound Output Dialog (accessible from the Window menu or available with the F7 shortcut). If disabled, output will be shown

	in the std.out of the console that opens when blue starts. (If the blue starter script has been modified not to show the console by using javaw instead of java, then the output will not be able to be seen). It is recommended to enable this option.
Message Colors Enabled	Since Csound5, the default for printed messages is to show them with colors by using ANSI text modifiers. However, this only works within ANSI-compliant consoles, and does not work within blue's Csound Output Dialog. It is recommended to keep this disabled if the "Show Csound Output" option is enabled.
Language	Which language to use for blue. This affects the labels of the User Interface, messages given to the user in dialog boxes, menu titles, etc. (Translation files may not always be up to of date so some UI items may not be translated when not in English.)

Project Defaults

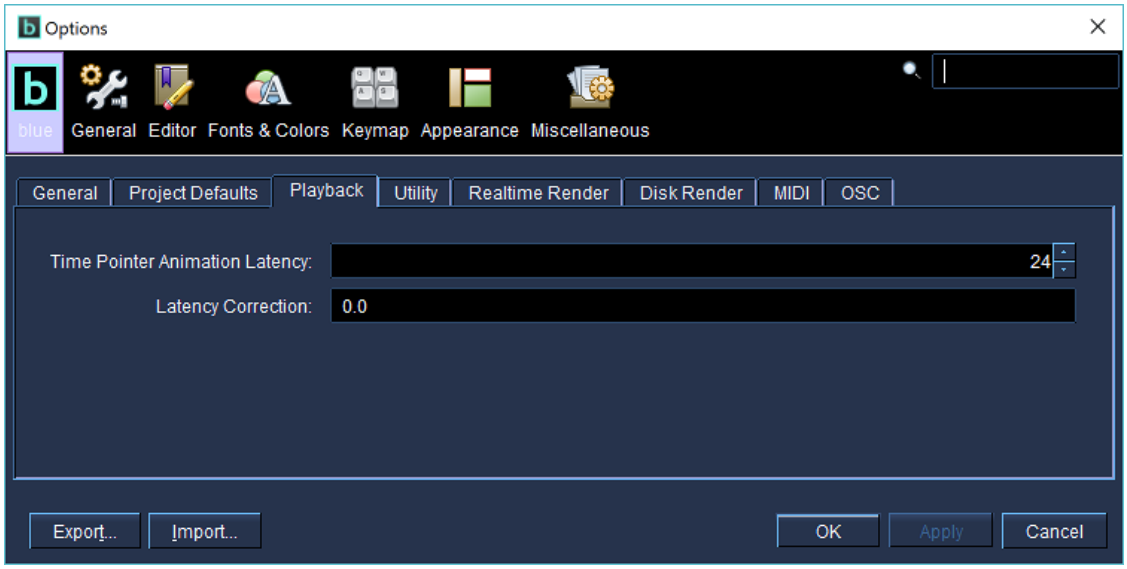


These settings are used whenever new projects are created as defaults for the project, unless a default.blue file is found in the user's blue home directory, in which case the settings from the default.blue file are used.

Project Defaults

Author	Default author to use in new projects.
Mixer Enabled	Enable using the blue Mixer in new projects by default.

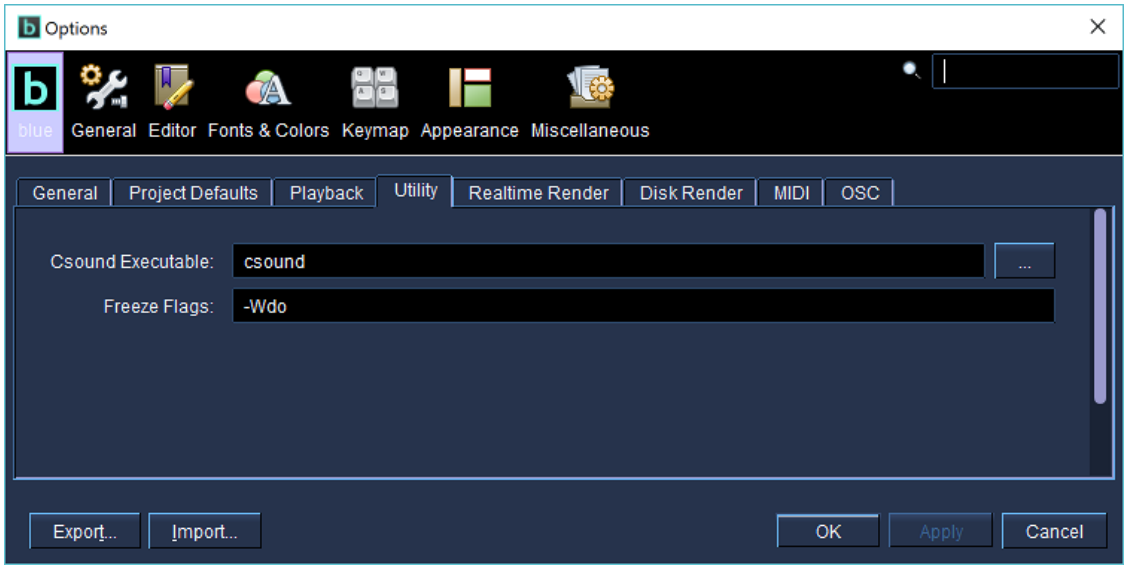
Playback



Playback

Time Pointer Animation Rate	Rate of animation for time pointer to update when playing back in realtime, expressed in frames per second. For example, the default value of 20 means the time pointer is updated 20 times per second, giving a smooth animation rate. Slower values may be desirable on slower computers if the playback pointer is affecting performance.
Latency Correction	Float value in seconds to use as a correction for audio latency in the user's sound playback mechanism. For example, if latency is quite bad on your soundcard and there is a delay of .5 seconds between when audio is passed to your soundcard and when the audio is actually realized from the DAC, the visual time pointer for blue may appear ahead in time of what is being heard. Using a .5 value for latency correction would correct for this.

Utility



Program Options - Utility

Utility

Csound Executable

This is the command for what version of Csound to execute when blue uses utilities that depend on Csound (for example, freezing SoundObjects, the SoundFont Viewer utility). The default value of "csound" works if you have a version of Csound in your path named "csound". You may use any command here that would call Csound that would work as if you were running csound from a terminal.

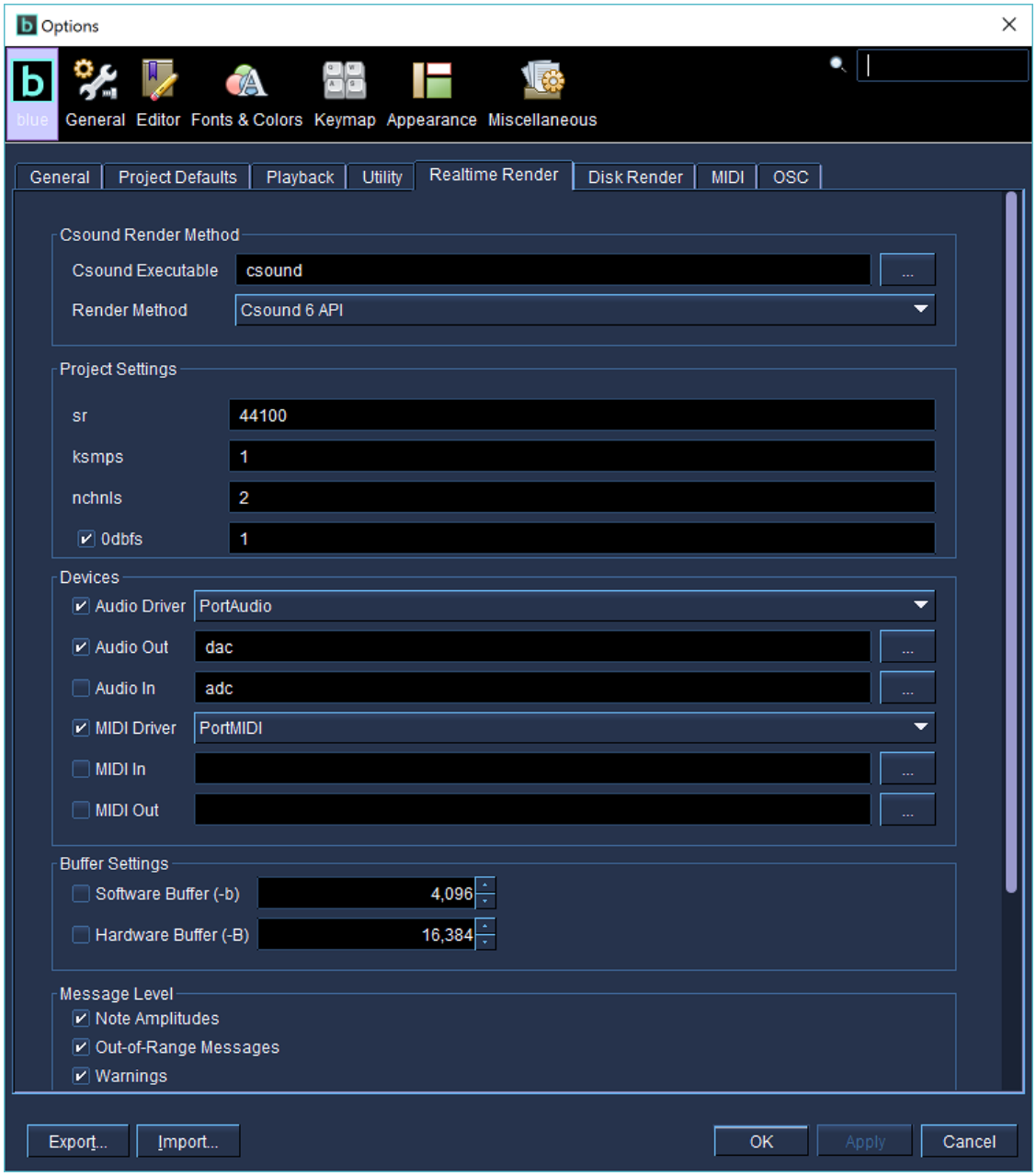
Note

The value defined for Csound Executable should only be the command to call Csound and no other flags should be added here.

Freeze Flags

These are the flags for Csound that blue uses when performing SoundObject freezing. Defaults are "-Ado" for Mac and "-Wdo" for all other operating systems. Users may wish to modify this value if they would their frozen soundObjects to be in a different format, i.e. 32-bit float soundfiles.

Realtime Render



Project Settings

Csound Executable

This is a command for what version of Csound to execute when blue renders the project in realtime. The default value of "csound" works if you have a version of Csound in your path named "csound". You may use any command here that would call csound that would work as if you were running csound from a terminal.

Using the button marked "..." on the right will open up a file dialog so that you can find and select the Csound executable to use to run Csound.

If you are using the API, you still need to have something set here. You can set it to "csound" in that case.

> **Note**
>
> The value defined for Csound Executable should only be the command to call Csound and no other flags should be added here.

Render Method — Choose which render service to use for rendering the project. When the program loads, Blue will first try to see if the Csound 6 API is available and if so, show that option here. If it can not find the Csound 6 API, it will search for the Csound 5 API and show that option here instead if it is found. If neither is found, the Commandline Runner option will always be available to use.

Note: If you have both Csound 5 and Csound 6 installed, you can force Blue to ignore Csound 6 on load by adding "-J-DDISABLE_CSOUND=true" to the blue/etc/blue.conf file in the default_options section. On OSX, this will be in blue.app/Contents/Resources/blue/etc/blue.conf.

sr — Default sr to use for realtime settings in new projects.

ksmps — Default ksmps to use for realtime settings in new projects.

0dbfs — Default value to use for 0dbfs. Also, the checkbox denotes whether projects should use 0dbfs by default or not.

Devices

Audio Driver — Driver to use for audio input and output. This is equivalent to using the --rtaudio= flag setting on the commandline.

Enabling the checkbox determines if this value will be used at all when rendering, and if so, will use the value from the dropdown as the driver setting.

Audio Out — Audio output device to use. Equivalent to using -o flag setting on the commandline. This setting is dependent on the setting used on the audio driver setting. Using a value of "dac" will use the default device for the driver.

The value of this setting will be used for all projects that set "Audio Out" enabled in the project-level realtime render settings.

By selecting the [...] button to the right of this field, blue will try to detect what devices are available for the chosen Audio Driver. If blue is able to find the devices, a popup dialog will appear with a list of available devices. Selecting from the popup will then populate the textfield with the correct device string that Csound will use to choose the device requested.

> **Note**
>
> If the driver is chosen but not enabled for use via its checkbox, when auto-detecting, blue will check for devices against the default driver and not necessarily what is in the dropdown. Please be sure that if you are planning to use the auto-detect feature with a particular driver that you also select the driver and enable it with the checkbox.

Enabling the checkbox determines if this device will be enabled by default for new projects.

Audio In

Audio input device to use. Equivalent to using -i flag setting on the commandline. This setting is dependent on the setting used on the audio driver setting. Using a value of "adc" will use the default device for the driver.

The value of this setting will be used for all projects that set "Audio In" enabled in the project-level realtime render settings.

By selecting the [...] button to the right of this field, blue will try to detect what devices are available for the chosen Audio Driver. If blue is able to find the devices, a popup dialog will appear with a list of available devices. Selecting from the popup will then populate the textfield with the correct device string that Csound will use to choose the device requested.

Note

If the driver is chosen but not enabled for use via its checkbox, when auto-detecting, blue will check for devices against the default driver and not necessarily what is in the dropdown. Please be sure that if you are planning to use the auto-detect feature with a particular driver that you also select the driver and enable it with the checkbox.

Enabling the checkbox determines if this device will be enabled by default for new projects.

MIDI Driver

Driver to use for MIDI input and output. This is equivalent to using the --rtmidi= flag setting on the commandline.

Enabling the checkbox determines if this value will be used at all when rendering, and if so, will use the value from the dropdown as the driver setting.

MIDI Out

MIDI output device to use. Equivalent to using -Q flag setting on the commandline. This setting is dependent on the setting used on the MIDI driver setting.

The value of this setting will be used for all projects that set "MIDI Out" enabled in the project-level realtime render settings.

By selecting the [...] button to the right of this field, blue will try to detect what devices are available for the chosen MIDI Driver. If blue is able to find the devices, a popup dialog will appear with a list of available devices. Selecting from the popup will then populate the textfield with the correct device string that Csound will use to choose the device requested.

Note

If the driver is chosen but not enabled for use via its checkbox, when auto-detecting, blue will check for devices against the default driver and not necessarily what is in the dropdown. Please be sure that if you are planning to use the auto-detect feature with a particular driver that you also select the driver and enable it with the checkbox.

Enabling the checkbox determines if this device will be enabled by default for new projects.

MIDI In

MIDI input device to use. Equivalent to using -M flag setting on the commandline. This setting is dependent on the setting used on the audio driver setting.

The value of this setting will be used for all projects that set "MIDI In" enabled in the project-level realtime render settings.

By selecting the [...] button to the right of this field, blue will try to detect what devices are available for the chosen MIDI Driver. If blue is able to find the devices, a popup dialog will appear with a list of available devices. Selecting from the popup will then populate the textfield with the correct device string that Csound will use to choose the device requested.

> **Note**
>
> If the driver is chosen but not enabled for use via its checkbox, when auto-detecting, blue will check for devices against the default driver and not necessarily what is in the dropdown. Please be sure that if you are planning to use the auto-detect feature with a particular driver that you also select the driver and enable it with the checkbox.

Enabling the checkbox determines if this device will be enabled by default for new projects.

Buffer Settings

Software Buffer

Size of software sample buffer to use (-b). For more information, see CommandFlags section of Csound manual for settings.

Enabling the checkbox determines if this value will be used at all when rendering.

Hardware Buffer

Size of hardware sample buffer to use (-B). For more information, see CommandFlags section of Csound manual for settings.

Enabling the checkbox determines if this value will be used at all when rendering.

Message Level

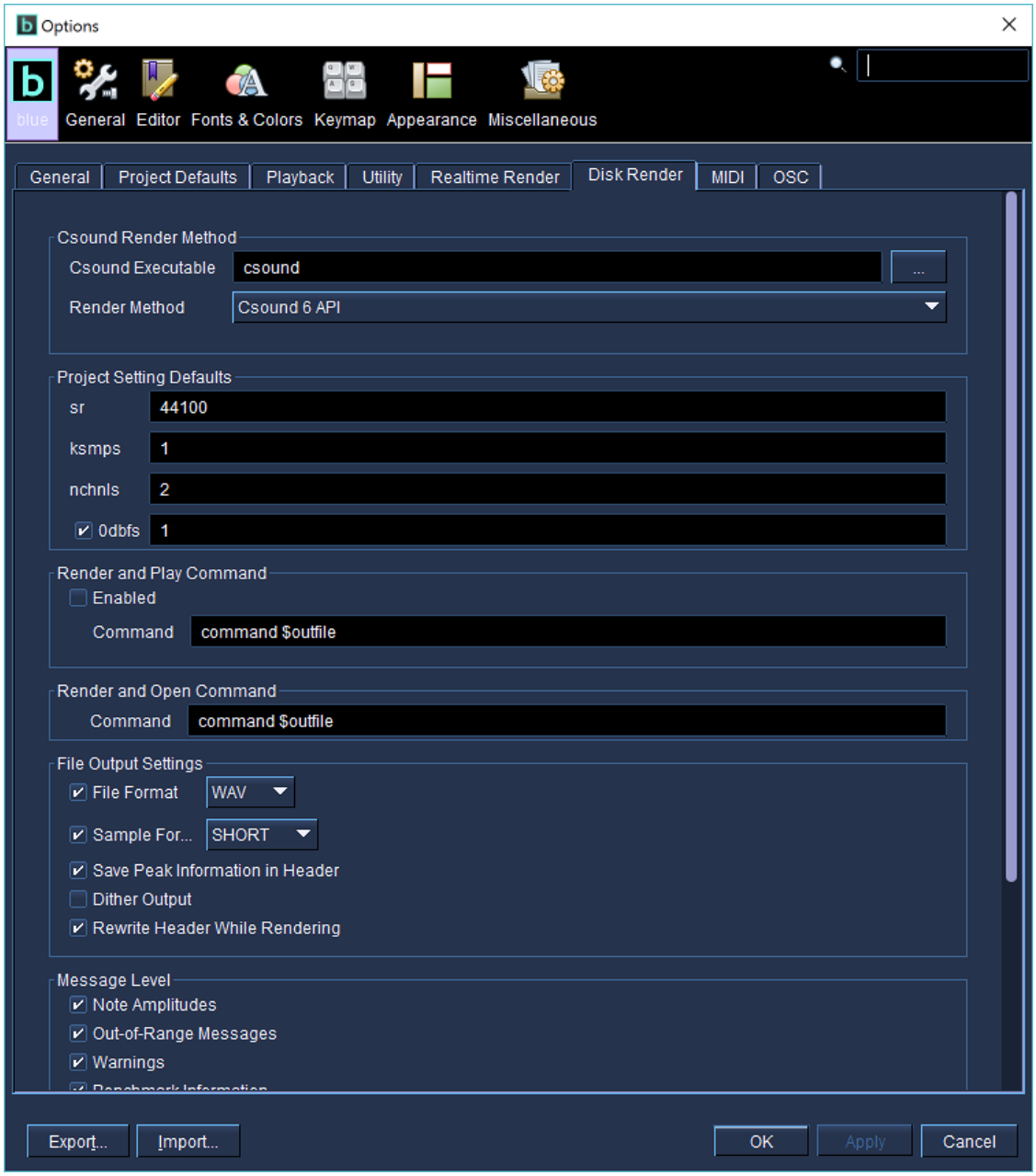
Note Amplitudes	Enables note amplitude messages from Csound (-m1)
Out-of-Range Messages	Enables samples out of range messages from Csound (-m2)
Warnings	Enables warning messages from Csound (-m4)
Benchmark Information	Enables benchmark information from Csound (-m128)

Advanced Settings

Advanced Settings

Extra flags to append to the commandline that might not be covered by options in the UI. Pressing the [?] button will open the documentation for the Csound command flags (Csound Documentation Root but be set for this to work).

Disk Render



Project Settings

Csound Executable

This is a command for what version of csound to execute when blue renders the project to disk. The default value of "csound" works if you have a version of Csound in your path named "csound". You may use any command here that would call csound that would work as if you were running csound from a terminal.

Using the button marked "..." on the right will open up a file dialog so that you can find and select the csound executable to use to run Csound.

Note

The value defined for Csound Executable should only be the command to call Csound and no other flags should be added here.

Render Method — Choose which render service to use for rendering the project. When the program loads, Blue will first try to see if the Csound 6 API is available and if so, show that option here. If it can not find the Csound 6 API, it will search for the Csound 5 API and show that option here instead if it is found. If neither is found, the Commandline Runner option will always be available to use.

Note: If you have both Csound 5 and Csound 6 installed, you can force Blue to ignore Csound 6 on load by adding "-J-DDISABLE_CSOUND=true" to the blue/etc/blue.conf file in the default_options section. On OSX, this will be in blue.app/Contents/Resources/blue/etc/blue.conf.

sr — Default sr to use for disk render settings in new projects.

ksmps — Default ksmps to use for disk render settings in new projects.

nchnls — Default nchnls to use for disk render settings in new projects.

0dbfs — Default value to use for 0dbfs. Also, the checkbox denotes whether projects should use 0dbfs by default or not.

Render and Play Command

Enabled — Enable using custom play command when using "Render and Play". If not enabled, blue's built-in audio player will be used once the project is finished rendering to disk.

Command — Command to call after finished rendering a project. The command line given should use the $outfile property where the name of the generated audio file should be passed to the program. For example, to open the rendered audio with VLC on Mac OSX, you can use this command line:

```
open -a VLC $outfile
```

File Output Settings

File Format — File format to use (i.e. WAV, AIFF, AU, etc.)

Enabling the checkbox determines if this value will be used at all when rendering.

Sample Format — Sample format to use. The default of SHORT is the same as 16-bit integer audio (the same as used for CD's). Other formats are available in the dropdown to use.

Enabling the checkbox determines if this value will be used at all when rendering.

Save Peak Information in Header — Save the peak information in the file header.

Enabling the checkbox determines if this value will be used at all when rendering.

Dither Output	Use dither on output. Enabling the checkbox determines if this value will be used at all when rendering.
Rewrite Header while Rendering	Rewrite the file header while rendering. This makes it possible to play the audio file in the middle of a render and is useful when the rendering of the project will take a very long time. However, it does slow down overall render time. Enabling the checkbox determines if this value will be used at all when rendering.

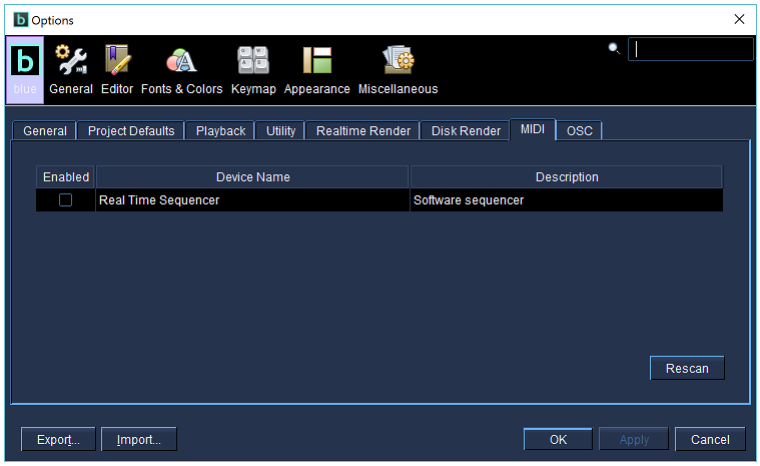
Message Level

Note Amplitudes	Enables note amplitude messages from Csound (-m1)
Out-of-Range Messages	Enables samples out of range messages from Csound (-m2)
Warnings	Enables warning messages from Csound (-m4)
Benchmark Information	Enables benchmark information from Csound (-m128)

Advanced Settings

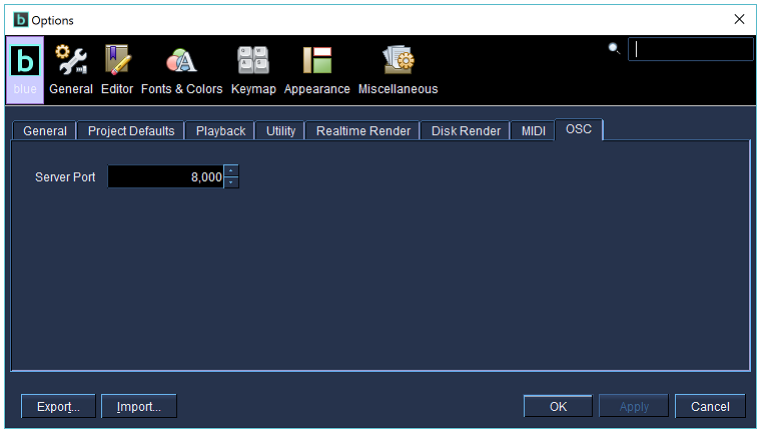
Advanced Settings	Extra flags to append to the commandline that might not be covered by options in the UI. Pressing the [?] button will open the documentation for the Csound command flags (Csound Documentation Root but be set for this to work).

MIDI



These settings are for blue's MIDI input, used with blueLive. The table shows what devices are currently available, and whether they are enabled for use with blueLive or not. You can use the "rescan" button to search for devices if you have just plugged in a device.

OSC

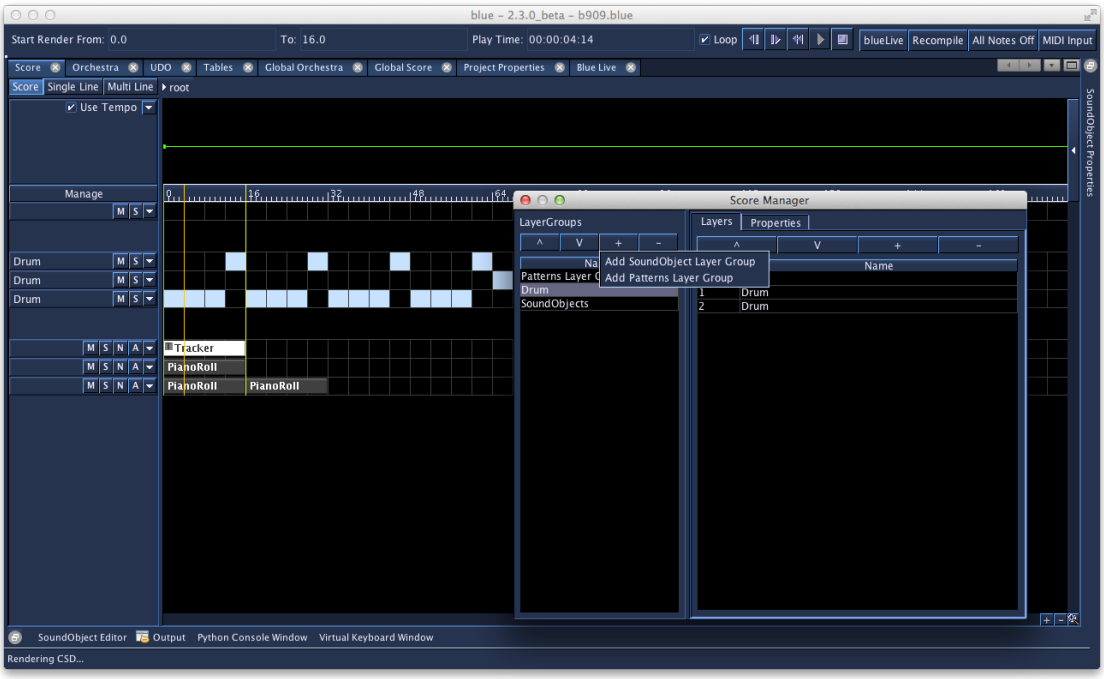


This allows you to set what port blue will listen to for incoming OSC messages. Defaults to 8000. The following messages are understood by blue:

- /score/play
- /score/stop
- /score/rewind
- /score/markerNext
- /score/markerPrevious
- /blueLive/onOff
- /blueLive/recompile
- /blueLive/allNotesOff
- /blueLive/toggleMidiInput

Primary Editors

Score Timeline



Score Timeline

Overview

The Score is the tool in Blue for working with musical material in time. It is a canvas upon which to create and organize musical ideas. The Score features a modular timeline, where each Layer Group within the Score is a module. These modules are a plug-in point that can be extended with other plug-ins. Each Layer Group in turn may be broken down into layers.

The image above shows a Score with three Layer Groups. The first two are Patterns LayerGroups, while the last is a SoundObject Layer Group. Also shown is the ScoreManager dialog (accessible from the Manage button on the left part of the Score).

Layer Groups

Introduction

A Score is divided into Layer Groups. Each Layer Group has a user-interface for the main timeline area, as well as a header interface that shows on the left-hand side. The header area usually shows meta-information and controls for each of the Layer Group's layers, such as layer name, muting, soloing, and other features.

By using the Score Manager Dialog, you can add as many Layer Groups as you like, as well as many layers to each group as you like. You can also push up and down Layer Groups to reorganize them. The same add/remove/push up/push down actions are also available for Layers within a Layer Group.

Note

Add/Remove/Push Up/Push Down for layers is also available while working in the main score area by right-clicking the Layer panels on the right and selecting the options from the popup menu.

Also, all LayerGroups support having NoteProcessors used with them. Using a NoteProcessor on a LayerGroup will affect all notes generated within that LayerGroup. Editing the LayerGroup's NoteProcessors can be done by right clicking on the root Score node in the Score Bar, described below.

Regarding the design, a Layer Group is primarily responsible for generating Csound Score notes. However, they are also able to generate tables, instruments, and anything else that is usable in Csound. It is up to the developer to choose what features the Layer Group will use.

SoundObject

SoundObject Layer Groups are one of the primary Layer Group types. They support the SoundObject system in Blue for scoring them in time on the timeline. They also are the canvas on which automations are drawn.

SoundObject Layer Groups are divided into SoundLayers. Each SoundLayer can contain SoundObjects of any type as well as have automations assigned to the layer. Unlike MIDI-based studio software, where layers are bound to a single instrument or channel in a mixer, Blue's SoundLayers are free to hold SoundObjects that generate any data for any instrument. This design choice allows freedoms for you to design your project as you wish, though at the expense of being able to implement some things which may be commonly found in MIDI-based studio software.

To use the SoundObject system, first create a SoundObject Layer Group using the Score Manager Dialog. (By default, new projects in Blue start with a single SoundObject Layer Group.) Next, add as many layers as you would like to the LayerGroup. You can then edit the names of the layers using the ScoreManager Dialog, or back in the main Score area by double-clicking the area on the left of the Layer's panel on the left.

On the timeline, rt-clicking on any soundLayer will show menu options for adding different types of SoundObjects, pasting a SoundObject from the buffer if any SoundObjects in the buffer are available, as well as other commands. (Copying and pasting of SoundObjects can also be done using ctrl-c and ctrl-click on a layer.)

Once you have SoundObjects on the timeline, you can click on them to select it, or click in an empty area and drag to show a selection rectangle. You can also add and remove SoundObjects to/from the selectionn by holding shift when clicking.

Once selected, you can move selected SoundObjects by pressing the mouse down on one of the selected SoundObjects, then drag to move them in time as well as up and down layers. If you press shift down when you click, once you drag you will create clones of the original material and move them in time. If you move the mouse to the right-side of a SoundObject, the cursor will change to a resize cursor, and clicking and dragging will allow you to change the duration of the SoundObject. (Note you can only resize one SoundObject at a time.)

If you have a single SoundObject selected, you can edit the properties of the SoundObject by using the SoundObject Property Window. This window can be opened from the menu "Windows -> Sound Object Properties" or by using the shortcut "F3". From this window you can change the name of the SoundObject, its start time and subjective duration, as well as add and remove NoteProcessors (if the SoundObject supports it).

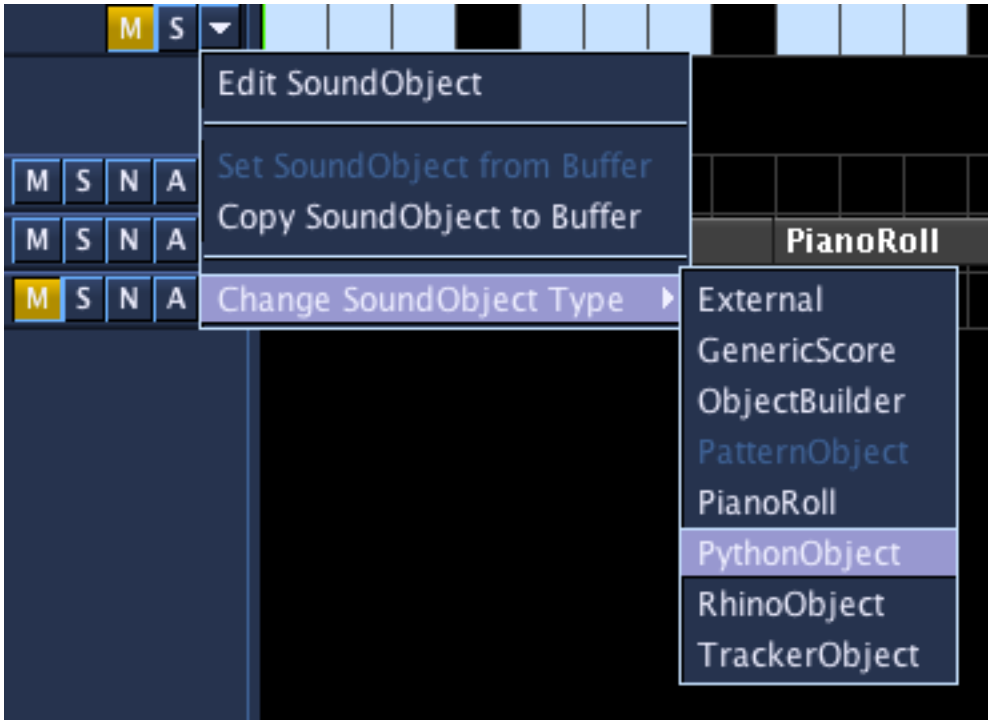
To edit a SoundObject, select a single SoundObject and open the SoundObject Editor window. You can also double-click a SoundObject, which will set the editor for the SoundObject in the Editor window

as well as focus the SoundObject Editor window. (If the Editor window is docked, double-clicking the SoundObject will cause the editor to show the docked window.)

Patterns

Patterns Layer Groups are based upon the same SoundObject system as the SoundObject Layer Groups, but presents a different way to working with these objects. At a high-level, each Pattern layer has one SoundObject assigned to it. The SoundObject is the source score material for a Pattern Layer. Selecting boxes in the grid that shows for the Patter Layer determines where that score material will be used to generate notes. For music projects that rely on patterns and regular metered time, using the Pattern Layer Groups can be more efficient for creating a piece than using SoundObject Layer Groups.

The grid of time for Pattern Layers is currently hardcoded to repeat every four beats, or one measure at 4/4 time. However, a SoundObject is not constrained to generate score for only one measure. For example, you can create a PianoRoll that has a time behavior of "None", and have it last sixteen beats (four measures). Everywhere that the pattern grid is selected for the Pattern Layer that uses that SoundObject will then create that sixteen beat score and start it at the time of the grid box.



Patterns Layer Menu Options

By default, new Pattern Layers will use a GenericScore SoundObject that is set to a duration of four beats, and use a time behavior of "None". This means that the score will generate "as-is". In the layer's popup menu options, you can replace the SoundObject that is being used for the layer with one from the SoundObject buffer (i.e. copied from BlueLive or from a SoundObject Layer). Also, you can switch to a different type of SoundObject using the menu.

SoundObjects will generate accordingly to the values set in the SoundObject Properties dialog, with the exception of the start time. Blue will reassign the start time to 0 when it generates the notes for the layer. On the whole this behavior will not affect most users, but is worth noting.

To edit the SoundObject for a Pattern Layer, you can either use the popup menu option for "Edit SoundObject", or you can also click the layer panel on the left. A single click will bring up the SoundObject editor.

To edit the layer's patterns, click on a grid box to toggle between on and off for that box. You can also press down and drag along to set multiple boxes. The value that is set for the multiple boxes will depend on the action of the first box selected. For example, if the first box when the initial mouse down is on, the mouse down will turn it off, and dragging will turn off all boxes that the mouse extends through.

Audio

Audio Layer Groups provide standard Digital Audio Workstation (DAW) functionality. Users can use Audio Layers to drag-and-drop Audio files onto the timeline. Each Audio Layer maps to a channel in Blue's mixer, where effects can be added for the audio layer. Automation for effect parameters is available on the audio layer.

To use the Audio layers, first create an Audio Layer Group using the Score Manager Dialog. Next, add as many layers as you would like to the LayerGroup. You can then edit the names of the layers using the ScoreManager Dialog, or back in the main Score area by double-clicking the area on the left of the Layer's panel on the left. The name of the Audio layer will also show under its corresponding mixer channel.

To add Audio files to a layer, you can either drag and drop a file from the operating system's file mananger, or you can use the Blue File Manager to locate a file, then drag and drop it to an Audio layer.

Once an audio clip is on the timeline, you can modify its properties in a few ways. First, you can click and drag it to move it in time. If you click and drag from the left hand side, you will alter both the audio clip's start time, as well as the audio clip's file start time. For example, if a clip was added that started at time 0.0 and had an audio file start of 0.0, if you drag from the left hand side to time 3.0, both the start of the clip and audio file start will be set to 3.0. This means that 3.0 beats into the project, the audio clip will start rendering from 3.0 seconds into the file. If you drag from the right hand side, it will alter the duration of the audio clip. If you hold down alt+shift, then press within an AudioClip, you will split the AudioClip into parts. If you hold down the alt key, then press and drag the mouse, you can alter just the file start time.

If you have a single Audio clip selected, you can edit the properties of the clip by using the ScoreObject Property Window. This window can be opened from the menu "Windows -> Sound Object Properties" or by using the shortcut "F3".

Additionally, if a single Audio clip is selected, additional properties are available to edit using the ScoreObject Editor window. You can also double-click an Audio clip, which will set the editor for the clip in the Editor window as well as focus the ScoreObject Editor window. (If the Editor window is docked, double-clicking the clip will cause the editor to show the docked window.)

Audio clips support fades. Like Ardour, upon which much of the audio layer system is based, all fades are crossfades. Clips will crossfade with signals that are already in the bus, with the first clip crossfading with silence. Fade types and durations may be set within the clip's editor panel. The clip duration may be visually modified by mousing over the clip, pressing with the mouse on either the fade-in or fade-out handle that appears after mousing over the clip, then dragging and releasing to update the fade's duration. The fade type may also be modified by right clicking within a fade area and choosing the fade type using the popup menu.

For more information about Fades, please see Ardour's manual entry on Region Fades and Crossfades [http://manual.ardour.org/editing-and-arranging/create-region-fades-and-crossfades/].

User-Interface Walkthrough

Play Bar

The play bar at the top has:

- time to start playing from
- what time to play to (if the render end time is set)
- the current play time
- Loop checkbox to have render looping from render start time to render end time
- Forward/Back buttons for jumping between markers
- Back button to start from beginning of Score
- Play/Stop buttons to start/stop rendering
- BlueLive buttons:
  - BlueLive toggle button to turn on/off BlueLive
  - Recompile button that will stop the current BlueLive run, recompile the project, and start BlueLive again
  - All Notes Off Button to turn off any hanging notes in a BlueLive run
  - MIDI Input toggle button enabled/disables Blue MIDI input into BlueLive

Score Bar

the poly object bar (shown above with only one polyObject, "root") shows what polyObject you are currently editting. if you were to add a polyObject named "phrase 1" to the main timeline shown above, then double click that polyObject to edit it, the polyObject bar would have two buttons on it, one for "root", and one for "phrase 1". you would then be editting "phrase 1"'s timeline. by clicking on the "root" button of the timeline, you would then return out of the polyObject's timeline and back in the root's timeline.

SoundLayer Editor

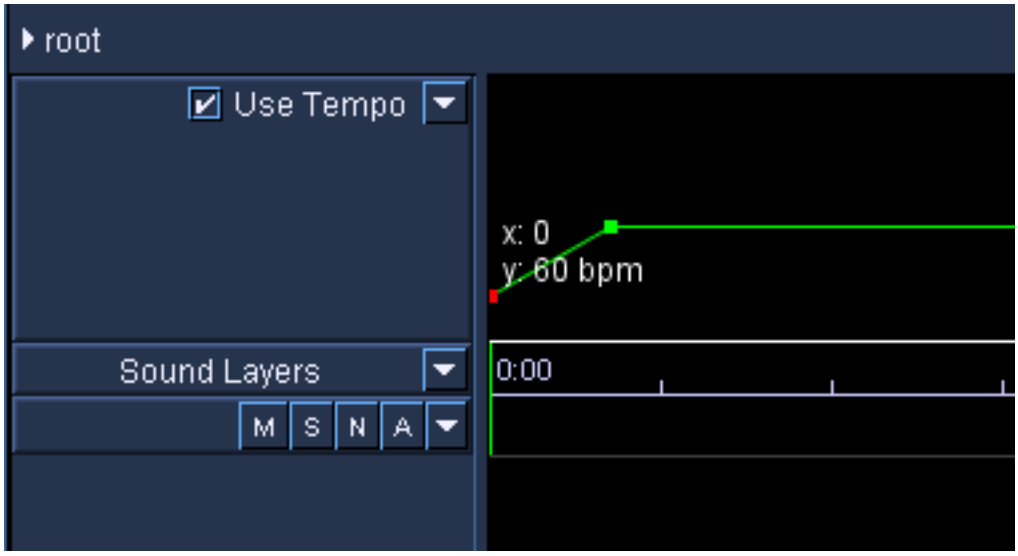
Below the polyObject bar on the left, you will see the soundLayer edittor. here you can change the name of the soundLayer, as well as mute the layer (all soundObject's on muted layers will not be used to generate notes when creating .CSD files).

on the bottom of the soundLayer edittor are four buttons, "^", "V", "+", and "-". "^" and "V" will push up or push down soundLayers. (HINT: You can move multiple soundLayers by clicking on one soundLayer, then holding down shift and clicking on the last of the soundLayers you want to move, then using the "^" and "V" buttons.) the "+" will add a soundLayer after the currently selected soundLayer. if no soundLayers are selected, then it will add one to the end of the list. the "-" button will remove any selected soundLayers. it should ask for a confirmation before removing any layers.

The Timeline

Below the polyObject bar on the right is the main time line. it shows the time line for the currently editted polyObject. The +/- buttons next to the scrollbars are used to zoom in on the time line, and those settings will be maintained between work sessions.

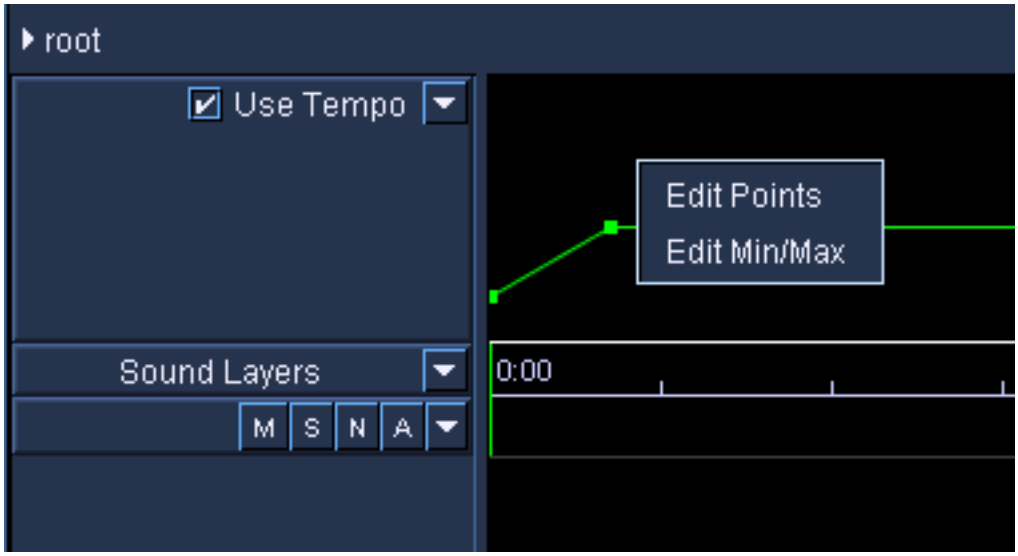
Tempo Editor



The tempo editor is an optional use feature that allows editing of overall tempo for a project using a line editor. This feature starts off as disabled and closed. When in this state, whatever tempo values are saved will show as dark gray line that is uneditable. To enable the use of the tempo editor, select the checkbox marked "Use Tempo". Selecting this will redraw the tempo line in green. To open up the tempo editor for a larger view and for editing, select the down arrow button next to the "Use Tempo" checkbox.

Like other line editor objects in Blue, left-clicking on an area where there is no point will insert a new point, while hovering over an existing point and pressing down, then dragging will allow moving of that point. Right-clicking a point will delete a point.

If right-clicking the tempo editor when not on a point, the following popup menu will appear:

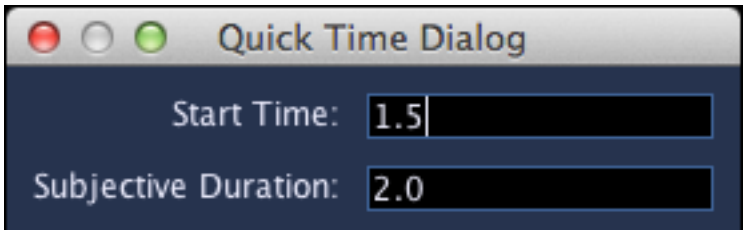


The first option will allow editing of the values of the points entered into the line editor by use of a table with two columns: the first column being the beat on which the tempo change occurs and the right column being the tempo value that it should have. One may find using the table editor easier to use to fine-tune values.

The second option will allow changing the boundary min and max tempo values for the line editor, as well as the option for what to do for points that fall outside of the new range. The options here are "Truncate" which will set any points' values that lie outside the new range to the closest boundary value, or "Scale" which will take all point values from the old range and scale them to the new range.

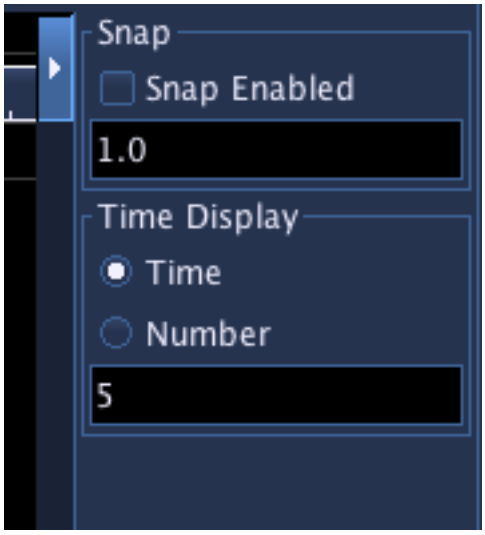
Use of the tempo editor is completely optional and users familiar with Csound's t-statement for controlling global tempo may opt to disable using Blue's tempo editor and to use a t-statement in the global orc section of the globals tab. Also, older Blue projects that existed from before Blue's tempo editor was developed can count on their projects loading and running correctly even if opened with versions of Blue that do have a tempo editor, due to Blue's default to disable the Blue tempo editor. Regardless of which tempo system is chosen by the user, one should be careful not to use both at the same time as this will cause two t-statements to exist in the generated CSD (the hand-entered one and the one generated by Blue), causing unexpected performance results.

Quick Time Dialog



While in the main timeline area, pressing ctrl-t when a soundObject is selected will bring up the quick time dialog. Here you can edit the start time and duration of the currently selected soundObject. When the dialog is first popped up the start time field is automatically focused for editing. You can press tab to switch between the two fields. Pressing enter will update the properties. Pressing escape, closing the dialog, or clicking anywhere outside of the dialog will cancel any changes and close the dialog.

Score Time Properties



In the upper right of the Score window is an arrow button; pressing this reveals the time properties for the Score. The two options here are Snap and Time Display. Snap provides a grid by which to line up your SoundObjects. Snap is measured in beats. Time Display allows setting how the time bar shows numbers, i.e. every 4 beats, rendered as a number or as a time value.

Shortcuts for the SoundObject LayerGroups

Table 1.1. Shortcuts for the Timeline

Shortcuts	Description
ctrl-c	copy selected soundObject(s)
ctrl-x	cut selected soundObject(s)
ctrl-click	paste soundObject(s) from buffer where clicked
shift-click	paste soundObject(s) from buffer as a PolyObject where clicked
shift-click	when selecting soundObjects, adds soundObject to selected if not currently selected and vice-versa
double-click	if selecting on timeline, select all soundObjects on layer where mouse clicked
ctrl-d	duplicate selected SoundObjects and place immediately after the originals
ctrl-r	repeat selected SoundObjects by copying and placing one after the other n number of times where n is a number value entered by the user (user is prompted with a dialog to enter number of times to repeat)
ctrl-drag	if ctrl is held down when drag is initiated of selected SoundObjects, a copy of the originals is made and left at their original times
ctrl-t	show quick time dialog
alt-1	switch to Score mode
alt-2	switch to Single Line mode
alt-3	switch to Multi Line mode

Orchestra Manager

Orchestra Manager

The Orchestra Manager is where the user organizes and edits the orchestra for the project as well as their user Instrument Library.

User Instrument Library

In the User Instrument Library (the tree pictured on the left), you can create instruments as well as categorize them into groups. Most actions in this editor are accessible by right clicking with the mouse to access popup menus. (keyboard shortcuts of ctrl-x, ctrl-c, and ctrl-v work for cutting, copying, and pasting nodes of the tree). To edit the name of Instruments or Groups, double click the name in the tree or hilight the node and press F2. This library of instruments is available across projects and is useful for building up and organizing a personal instrument library.

To copy instruments into the library from the project's Orchestra, you can drag and drop the instrument on the library tree to make a copy of the instrument. To copy an instrument from the Library to the project's orchestra, you can similarly drag and drop an instrument from the Library to the Orchestra. You may also use cut/copy/paste to move instrument from one to the other.

Orchestra

The Orchestra panel (located in the top left of the picture) is where you will place your instruments to be used for your project. The "+" button allows you to create new instruments from one of the available Blue instrument types. You can also right-click on the Orchestra to show options for copying/pasting. Once instruments are in the Orchestra, you can edit what their instrument ID's: these may be either numbers or strings (i.e. 3 for "instr 3", violin for "instr violin"). Selecting an instrument will show its editor in the Instrument editor section on the right.

Instrument Editor

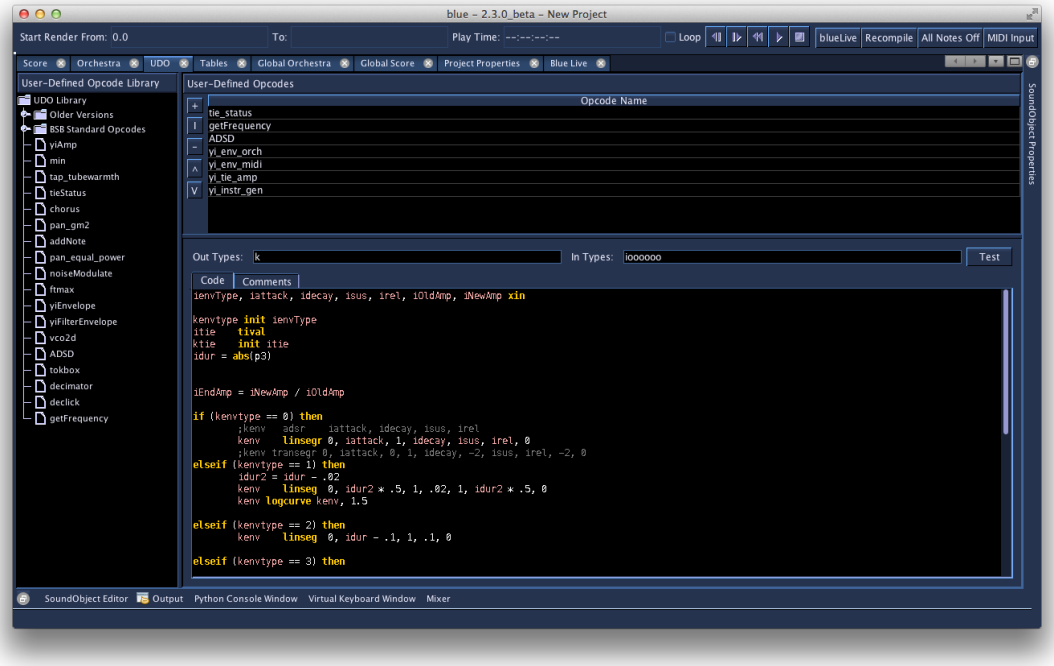
On the right hand side of the Orchestra Manager is the area for editing instruments. To make more space available for editing an instrument, you can drag the split pane splitter to the left, or double-click the splitter to have it collapse to the left-hand side. Each Blue instrument-type has its own editor. Note: if you are editing and instrument from the Library, Blue will show a green border with the title "Editing Library Instrument" to help notify you what instrument you are editing.

Legacy Projects

For projects before 0.94.0, imported projects will have their instruments largely setup in the same way as they were before.

For projects that were in the 0.94.0 beta series, per-project Instrument Libraries will be removed. An option to import the instrument library will be presented to the user and if the user agrees, a folder entitled "Imported from Project" will be added to the root of the user's InstrumentLibrary. On the whole, it is recommended to import the library, review the instruments, and simply remove what is not necessary. Deleting the imported library is as simple as deleting the folder for the imported group. Instruments in these project's Arrangements will be modified to hold copies of the instruments from the library as opposed to being references to instrument in the library. (Previously, if two instruments in the Arrangement pointed to the same instrument in the per-project InstrumentLibrary, changes to the instrument in the library would affect all instruments in the Arrangment that pointed to it. Now, only copies are made and no references are used.)

User-Defined Opcodes Manager



User-Defined Opcodes Manager

The User-Defined Opcodes Manager contains three main parts:

- Program-wide User-Defined Opcode Library

- UDO list for Project-wide UDO's
- UDO Editor

The program-wide UDO library is where you can manage your entire library of UDO's. Right-clicking on the tree allows adding and removing groups, as well as adding and removing UDO's. You can also drag UDO's into the library from the project-wide UDO list, or paste them using the popup menu. Clicking on a UDO in the library will populate the UDO Editor, and a green border will be shown to highlight that you are editing the library copy of the UDO.

The project-wide UDO list contains what UDO's are available for your project. Since UDO's can be embedded into Blue instruments, it is usually better to do so so that an instrument is encapsulated and copying the instrument will include all of its UDO's it depends on. However, using the project-wide UDO list can be useful when creating a new project and multiple new instrument designs might be using a developing UDO.

UDO's can be created in the project-wide list by using the "+" button, and removed by using the "-" button to the left the table on top. UDO's can also be dragged into this list from the library. You can also drag in a folder of UDO's from the library into this list, or copy/paste them using the popup menu. Dragging a folder is useful if you have a set of UDO's you commonly use in all of your instrument designs.

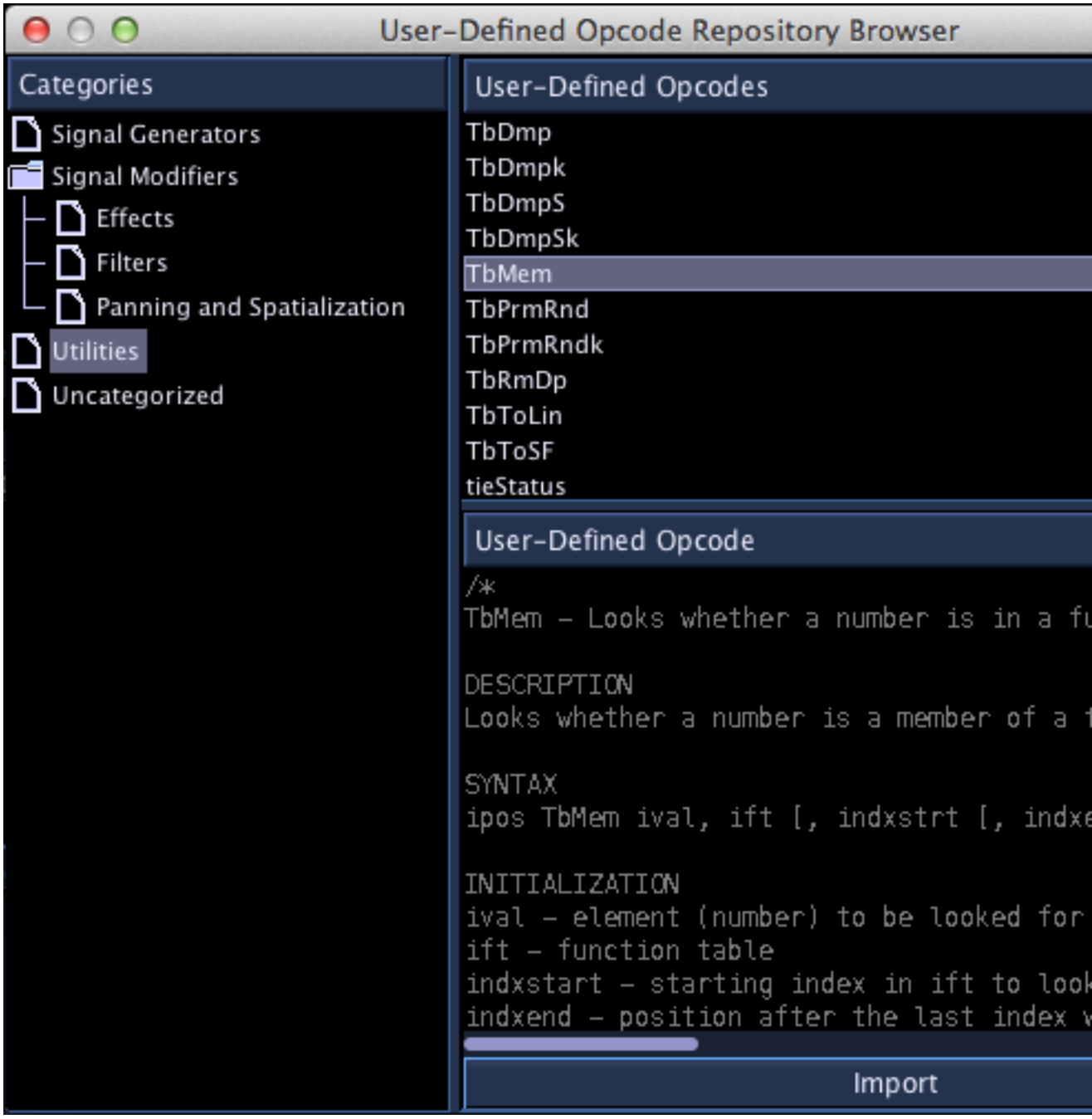
As UDO's may depend on other UDO's, the order in which they are generated can be significant. The UDO's in the table are generated from top-down. To shift the order of the opcodes up and down, select an opcode in the table and use the "^" and "V" buttons to push up and push down.

To edit the UDO, select one from the table. After selecting a UDO, the UDO Editor will be populated with that UDO. This time, no green border will show, as that is only done when a Library UDO is being edited.

Notes on Editing UDO's

For UDO's, you will need the name of the UDO, the intypes and outtypes, and the body of the code itself. For the body of the code, you will not need anything from the "opcode" line that normally starts a UDO definition (i.e. "opcode myOpcode, a, ak"), as those should be in the text fields for Opcode Name, In Types, and Out Types, and you will not need an "endop", as Blue will add that itself.

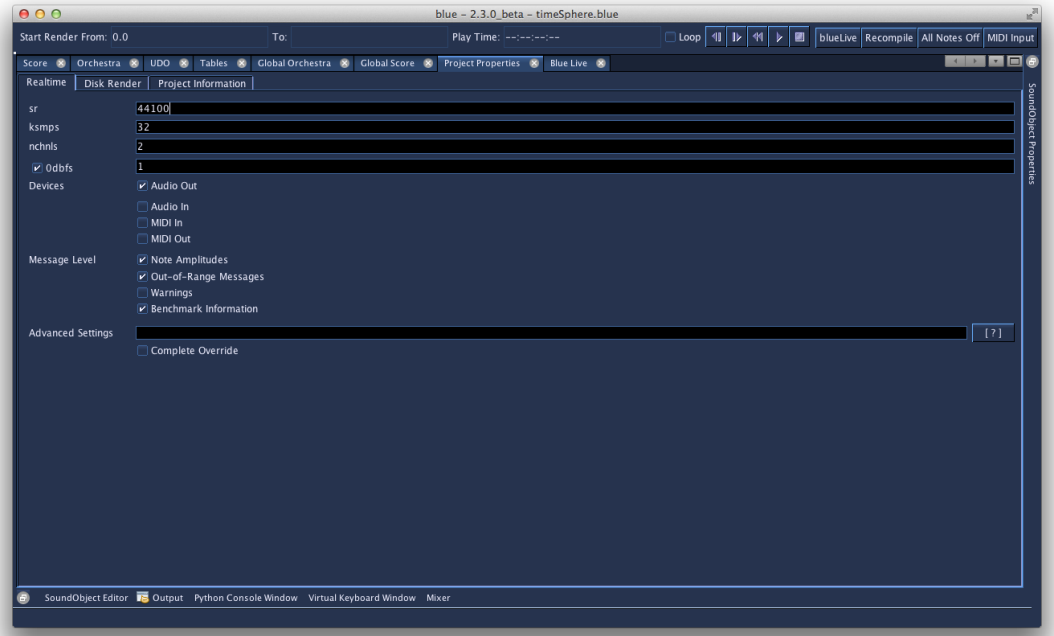
User-Defined Opcode Repository Browser



User-Defined Opcode Repository Browser

Using the "I" Button will open up the UDO Repository browser. The browser shows the available UDO's in the repository on Csounds.com and allows for importing from the repository straight into your project.

Project Properties



Real-Time Render Settings

sr	sr to use when rendering in real-time. Value defaults to value set in Program Options.
ksmps	ksmps to use when rendering in real-time. Value defaults to value set in Program Options.
nchnls	nchnls to use when rendering in real-time. Value defaults to value set in Program Options.
0dbfs	The checkbox sets whether 0dbfs is used at all in the project. If enabled, the value will be assigned to the value in the textfield. The default for the value is set ing Program Options, as well as if 0dbfs is enabled by default or not.
Devices	Devices to use when rendering in real-time (Audio In/Out, MIDI In/Out). The value of the device is dependent on the values set on the Program Options. By delegating the value to use to what is set on the Program Options, the project does not have to store settings which are hardware dependent, so projects can be easily moved from one computer to the next. For example, if your project is set to use "Audio Out", one system may use a value of "-+rtaudio=alsa -o dac:hw:0,0" while another system may use a value of "-+rtaudio=winmme -o dac2". The project only needs to be set to use "Audio Out" and when the project goes to render, the settings set for that system's audio out will be used.
Message Level	Enables what kind of messages Csound should report. The values default to what is set in Program Options.

Advanced Settings

Extra flags to append to the commandline that might not be covered by options in the UI. Pressing the [?] button will open the documentation for the Csound command flags (Csound Documentation Root but be set for this to work).

If "Complete Override" is enabled, the value given in the "Advanced Settings" textbox will be used as given and no other values set from the UI will be used. Projects prior to 0.106.0 will have their commandline settings copied to here and the "Complete Override" section will be enabled. When this setting is enabled, the commandline should set the call to the Csound executable to use and the flags to use but with the name of the CSD left out as it will automatically be appended to by Blue. An example of a commandline to use here with "Complete Override" is:

```
csound -Wdo dac
```

Disk Render Settings

sr

sr to use when rendering to disk. Value defaults to value set in Program Options.

ksmps

ksmps to use when rendering to disk. Value defaults to value set in Program Options.

nchnls

nchnls to use when rendering to disk. Value defaults to value set in Program Options.

0dbfs

The checkbox sets whether 0dbfs is used at all in the project when rendering to disk. If enabled, the value will be assigned to the value in the textfield. The default for the value is set ing Program Options, as well as if 0dbfs is enabled by default or not.

Filename

Name to use for the rendered sound file. If a value is not given, Blue will ask on each render what to name the file and where to render it to.

If the "Ask on Render" is enabled, Blue will always ask on each render what to name the file and where to render it to. This is useful to enable if temporarily rendering parts of a project or if the project is only meant to be used to render small sound samples.

Message Level

Enables what kind of messages Csound should report. The values default to what is set in Program Options.

Advanced Settings

Extra flags to append to the commandline that might not be covered by options in the UI. Pressing the [?] button will open the documentation for the Csound command flags (Csound Documentation Root but be set for this to work).

If "Complete Override" is enabled, the value given in the "Advanced Settings" textbox will be used as given and no other values set from the UI will be used. Projects prior to 0.106.0 will have their commandline settings copied to here and the "Complete Override" section will be enabled. When this setting is enabled, the commandline should set the call to the Csound executable to use and the flags to use but with the name of the CSD left out as it will automatically be appended to by Blue. An example of a commandline to use here with "Complete Override" is:

```
csound -Wdo mySoundFile.wav
```

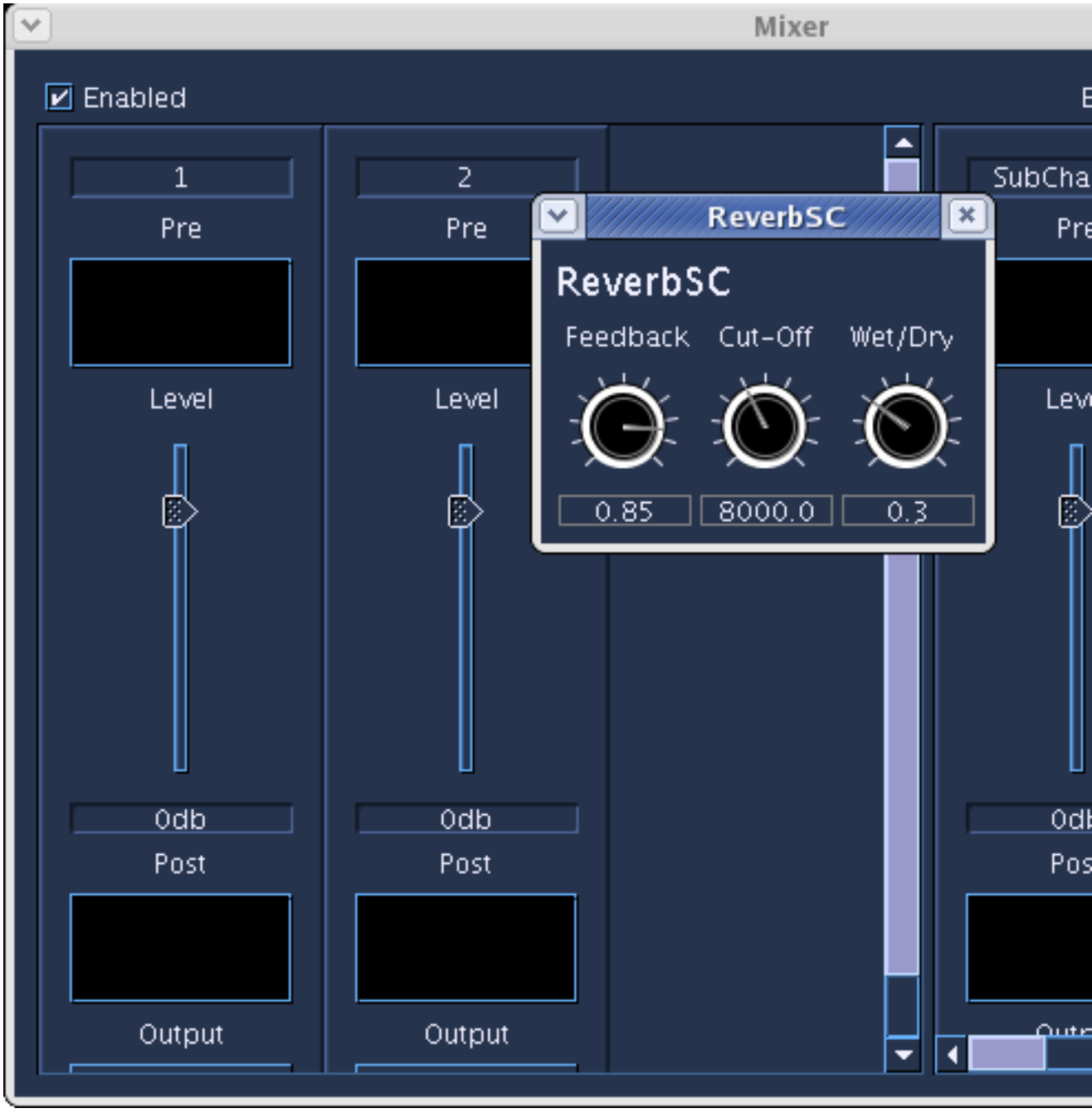
Project Information

Title — Title for this project. For general information purposes; is also used when generating header comments in CSD.

Author — Author for this project. Defaults to value set in Program Options. For general information purposes; is also used when generating header comments in CSD.

Notes — Notes for this project. For general information purposes; is also used when generating header comments in CSD.

Mixer



The Mixer system in Blue allows for graphically editing levels for instruments, applying pre- and post-fader effects, and routing and mixing of signals through subchannels.

Architecture

The Mixer system has three panel sections:

Mixer Sections

Channels	Channels are auto-created and bound to Instrument ID's in the Orchestra for a Project. In the Mixer Dialog they are located in the first section on the left within a splitpane that separates them from the SubChannels. Channels can be set to route to either SubChannels or directly to the Master Channel.
SubChannels	SubChannels are user-created and are located in the center section of the Mixer Dialog, on the right side of the splitpane. Channels can be set to route to SubChannels, and SubChannels can route into other SubChannels or out to the Master Channel.
Master Channel	The Master Channel is located on the right side of the Mixer Dialog. There is only one master Channel per project which all channel and subchannel signals ultimately route through.

Each channel allows for applying effects to the incoming signal either pre- or post-fader.

Using the Mixer

For most MIDI-based music composition environments, the typical user interaction with a Mixer system is that users first create tracks on a timeline, and for each track an audio channel is automatically bound in the mixer. The user then selects an instrument for that track(if it is a MIDI track); MIDI information from the track is routed to the instrument and that instrument generates audio signals. If the instrument happens to be a software synthesizer, the audio signals are then usually taken from the instrument and routed out to the Mixer channel that has been bound to that track.

However, since Blue does not bind music information on a SoundLayer to an instrument (you can have heterogenous note data generated in any soundObject for any number of instruments, a flexibility which allows for soundObjects to be more representative of musical ideas), and nor does it bind SoundLayers to channels, the abstraction of the musical system and the interaction with the Mixer system requires different handling.

In Blue's Mixer system, Mixer channels are automatically bound to instruments by their instrument ID. Binding to ID and not per-instrument in the Orchestra allows for the case where users have set multiple instruments to the same instrument ID but only having one which is enabled. If you then disable one and then enable another instrument to test out different instruments with the same musical note data, the mixer channel's settings will be maintained for the new instrument as it is bound by ID.

Channels in themselves can not be created or removed directly by the user, but are automatically added or removed depending on how how instruments are added and removed from the project's orchestra. For cases of when an instrument has an ID and a user wishes to change the ID, if the new ID already exists and a channel is already created, the old channel is removed as long as no other instrument has the old ID. If the new channel ID has no existing mixer channel bound to it and if the channel for the old ID only has one instrument with that ID (the instrument that is changing ID's), the bound mixer channel is simply reassigned to the new ID and maintains its settings.

Subchannels are added by right-clicking with in the SubChannels area and choosing "Add SubChannel" from the popup menu. To remove a subchannel right-click the channel strip to be removed and select "Remove" from the popup menu.

At the bottom of every Channel and SubChannel strip is an output dropdown box to choose where to route that Chanel's audio to. For SubChannels, they can only route to other SubChannels which lead to the Master Channel, meaning that there is no feedback allowed (i.e. routing SubChannelA to SubChannelB and then to SubChannelC and that back to SubChannelA is not allowed as it would create a loop).

For the instruments in the orchestra to be able to route out to the Mixer, a special pseudo-opcode must be used for the output of the instrument, entitled "blueMixerOut". You use blueMixerOut in the same way as the outc opcode. If the Mixer is not enabled in the Dialog, when Blue goes to process the instrument, blueMixerOut will be replaced with outc, thus making the audio of that instrument directly route out to dac or disk, depending on how Csound is set to render. If the Mixer is enabled, blueMixerOut is translated into global variables and the Mixer's instrument code is generated automatically without the user having to worry about the code details of setting up a Mixer system themselves in Csound code.

There is also a subChannel form of blueMixerOut available that is able to target a subchannel by name. This form is used in the following way:

```
blueMixerOut "subChannelName", asig1, asig2 [, asig3...]
```

Using this form, the asig signals will be mixed into the subChannel given by name. This form is available to use within Instruments but is also very useful to use when working Sound SoundObjects and AudioFile SoundObjects, which do not have channels created for them. This way, you can route the output of an AudioFile to a named subchannel and apply Effects, etc.

Effects

Effects in Blue are implemented as User-Defined Opcodes, and understanding of how User-Defined Opcodes work in Csound is recommended before creating Effects. Understanding how UDO's work however is not necessary if one simply wants to use Effects.

The workflow for using Effects with your Mixer channels is:

1. Populate your Effects Library by either creating effects or importing them from BlueShare.

2. In the Mixer, choose either the pre-fader or post-fader effects bin to add effects. Right click on the bins to open up a popup menu that shows effects that are currently in your library from which to choose to insert.

3. In the Mixer, choose either the pre-fader or post-fader effects bin to add effects. Right click on the bins to open up a popup menu that shows effects that are currently in your library from which to choose to insert.

4. Configure your effect by double-clicking it in the effects bin. Double-clicking the effect will open up a dialog that shows the effect's graphical user interface.

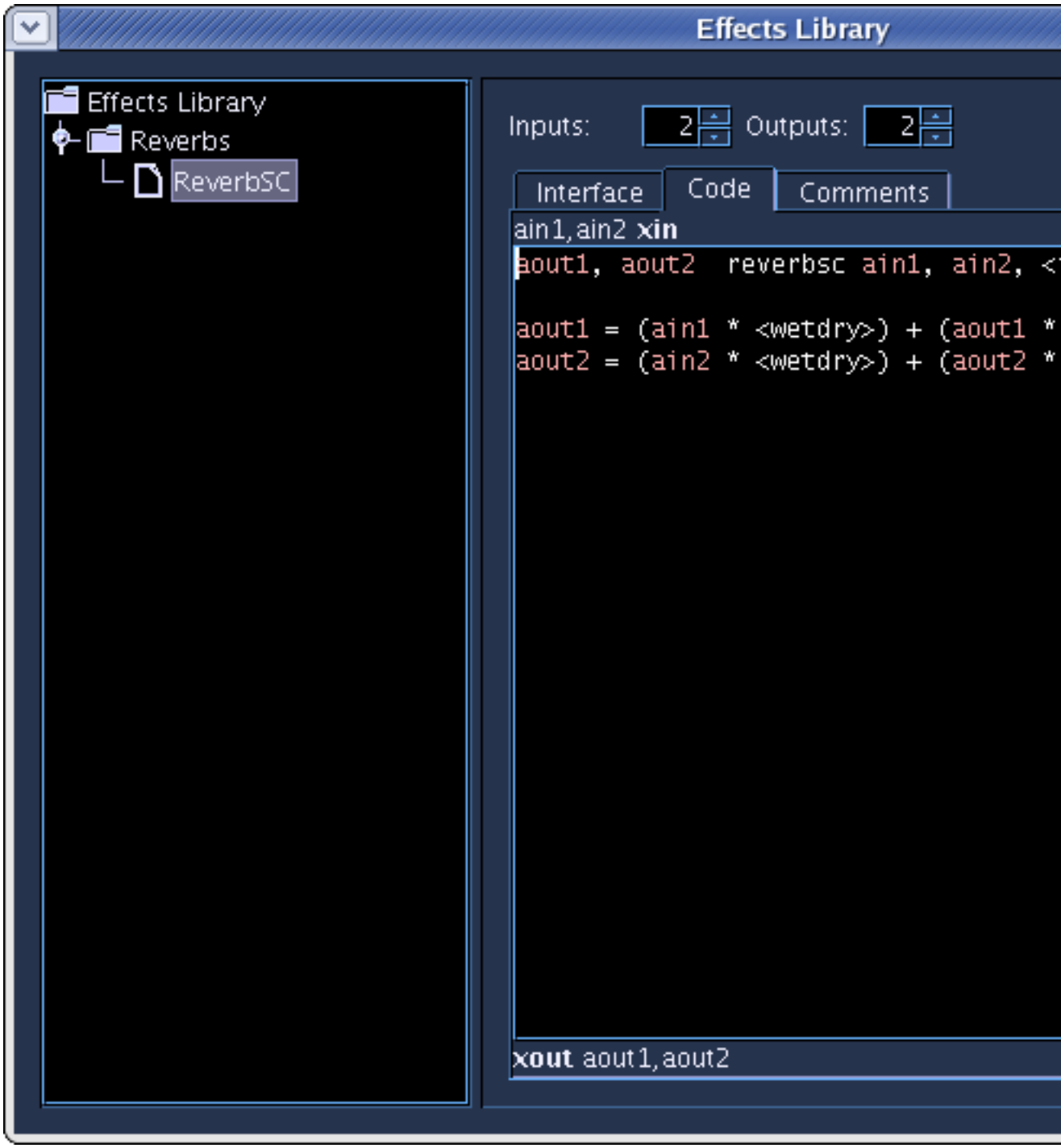
Effects Library

Effects are created and managed in your Effects Library, which is accessible from the Tools menu. The Effects Library is program wide, so any Effect in the library will be accessible to use from any project you are working on.

In the picture above, you will see the library where one can create Effects as well as organize them into groups. The organization into groups within the library will be reflected in the Effects popup that appears when the user is looking to add Effects to their Mixer channels.

To add a group or new Effect, right click on any node and choose the option from the popup. The popup will also allow you to cut/copy/paste groups or Effects which is useful when reorganizing or when creating new Effects based on older ones.

Once an Effect or group is added, you can edit the the name by double-clicking the node in the tree. Selecting an effect by clicking it once will populate the editor on the right. The picture above shows the User Interface editor for the Effect. Like other BlueSynthBuilder-based editors in Blue, clicking Edit Enabled will allow you to move between to edit modes, one in which you can add, remove, and move around widgets, and another where you can interact with the widgets (useful for setting an initial value for the Effect.)



In the code editor for the Effect, one sees that the xin and xout lines of the User-Defined opcode display according the number of in and out audio signals the Effect will support. The names of the signals are hard-coded, which makes it easier for others to know exactly what the incoming and outgoing signals will be should they look at your code.

Code for the Effect should follow the same principles as User-Defined Opcodes(i.e. one can add a setksmps line at the top of the code area). Values from the widgets follow the same principles as BlueSynthBuilder, and code completion for opcodes (ctrl-space) and BSB widgets (ctrl-shift-space) work within the code editor.

Note

Blue currently expects Effects to have nchnls number of channels in and out where nchnls is the number set by the project.

Other Notes

- The Extra Render Time option in the Mixer dialog allows the user to add extra time to the end of the score. This is useful to allow time for Effects which may have time delay (i.e. 3-second long reverb) to have time enough for processing, as well as simply to add time at the end of a piece.

Sends

Besides effects, users are also able to put in Sends into the pre- and post-fader Effects bins. Sends will output the signal from that point in the channel's signal chain to a selected SubChannel. As with the output channels, Sends can only feed-forward down the chain to the Master Out and can not be made to create a feedback loop. Users are able to set the channel to send to as well as the amount to send. This send amount is also able to be Automated, just as the Effects are.

Randomization

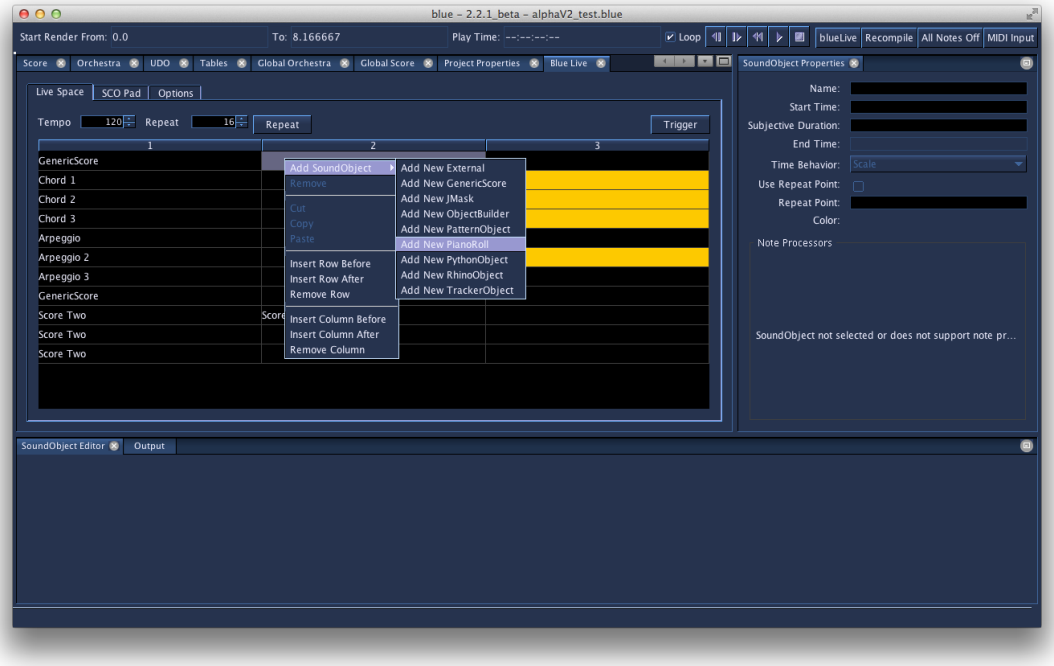
Widget values are able to be randomized in the same way as in the BlueSynthBuilder, and is available in the usage mode when working with Effects in the Effects Library or when using Effects from the mixer. Please further information, please see the documentation for BlueSynthBuilder Widget Randomization .

Code Generation Optimization

Blue's mixer system optimizes when generated code for Csound to use. When compiling down to ORC code, Blue checks all signal paths to see if any path will result in unused audio and if so, will optimize out any code that is along that path. The rules for optimization are as follows:

- If channel does not generate signal to channel output (if no automation for level fader and fader == -96.0f), only generate code up to last send in preFader effects chain
- When processing Sends, checks down the graph to see if send signal will make it to the Master output or not, checking both down graph of output channels as well as sends for each channel. If not, does not generate output for Send.
- If channel does not have channel output and no valid prefader sends, do not generate anything for channel.
- If SubChannel has no signal inputs (whether it's from another Channel's out channel, a Send, or dependency from code that uses the subChannel form of blueMixerOut), do not generate anything for channel.

Blue Live



Blue Live

Blue Live allows you to work with Csound in realtime. It allows for generating score with SoundObjects and working with MIDI keyboard input to create notes and run with Csound instruments defined in your project. Note: Blue Live works when using the Csound API, or on non-Windows platforms when not using the Csound API. Windows does not allow piping text to executable in a non-blocking way and therefore limits what can be done when not using the API.

Motivation

The motivation of Blue Live is primarily to aid the composer in working out ideas and to help configure instrument, effects, and mixer settings. You may find it helpful early on within the life a composition, when you want to try out a number of different ideas in realtime. You may tweak instrument parameters, mixer settings, try out different notes and chords with a MIDI keyboard, and even work with different SoundObjects. Later, when you have some ideas worked out, you can take your SoundObjects from BlueLive into your score timeline and continue working from there.

Beyond this primary capacity as a mode to aid composition, Blue Live has some capacity to be used for realtime performance. The focus of Blue Live's development is as a compositional aid first, and performance second, though work continues to expand its usefulness in both regards.

Working with Blue Live

Blue Live is designed to work with the rest of your Blue project file. When Blue Live is turned on, the Blue project generates everything from the project except for the Score generated from the Score timeline. The Global Score text will be used, but instead of <TOTAL_DUR> being calculated from the score timeline, a default value of 3600 is used. This allows your notes that would be used for effects instruments to run for 3600 seconds (this size can be modified; please make a request if desired).

The main toolbar has four buttons for Blue Live:

BlueLive	Toggle button that stops and starts BlueLive
Recompile	If BlueLive is running, this button will cause Blue to recompile the CSD from your project and restart BlueLive. This is useful if you modify your orchestra code and want to quickly recompile and continue working with BlueLive.
All Notes Off	Turns off any score notes that are actively playing
MIDI Input	Toggle button that turns on and off the configured MIDI devices setup in Program Options (discussed further below).

The primary Blue Live window is available from the Window Menu, or by using the ctrl-8 shortcut. The Blue Live window has three main tabs: the Live Space, the SCO Pad, and Options. These will be discussed int he following sections.

Live Space

The Live Space is an area to work with SoundObjects. It is a table divided into bins and rows of spaces to place SoundObjects. SoundObjects can be copied to/from the Score Timeline as well as the Live Space. SoundObjects can also be created within the Live Space by right clicking an empty bin within the bins and choosing "Add SoundObject" from the popup menu. Clicking on an occupied bin will select that SoundObject. The properties for the SoundObject can be modified using the SoundObject Properties Window, and the contents of the SoundObject can be modified from the SoundObject Editor Window. Besides the selected and unselected states, you can double-click a soundObject to put it into an enabled state. Enabled objects are highlighted in orange and are the SoundObjects that are triggered when multi-trigger or repeat is used.

Once a SoundObject is added to the Live Space, it can be triggered in one of two ways:

Single Trigger	Trigger the contents of the currently selected SoundObject. This is done by using the keyboard shortcut ctrl-T (cmd-T on OSX).
Multi Trigger	Triggers the contents of the currently enabled SoundObjects. This is done either by using the Trigger button, or using the keyboard shortcut ctrl-shift-T (cmd-shift-T on OSX).

Triggering a soundObject will take the score generated from it and pass it immediately to the active Csound instance for BlueLive (BlueLive must be running for this to work). Scores are always generated with Time Behavior of None, processed through their NoteProcessors, then scaled according to the given tempo.

One can turn on Repeat to cause repeated trigger of enabled SoundObjects according to the tempo given and the number of beats in which to retrigger. For example, using a repeat of 4 means it will re-trigger every four beats. The score is processed as mentioned above, but there is no truncation done. Therefore, if a SoundObject generates a score of 8 beats, and there is a repeat of 4 beats, when the repeat occurs, there will be 4 beats of overlap between the first trigger and the second trigger.

When Repeat is on, it will always finish out the current number of beats in the repeat at the current tempo, before applying any changes to tempo and repeat. Modifyications to tempo and repeat will apply then on the next repeat trigger.

SCO Pad

Note

This feature will likely be removed in a future release.

This is an experimental feature to record MIDI input in a manner similar to to notation programs (press keys, then press 4 for a quarter note, 8 for and 8th note, etc.). This feature requires MIDI Input to be turned on. This feature is experimental at this time.

Options

The options panel allows setting up parameters for Blue Live. Currently it contains options for modifying the commandline string used when running BlueLive. For most users, these modification will not be necessary and the default commandline used will be sufficient.

Working with MIDI

Blue's MIDI system, when enabled, will listen to configured MIDI devices for notes, map the key and velocity, and generate Csound notes to achieve MIDI-like note-on and note-off type behavior. This allows working with a MIDI keyboard in realtime with your project instruments without modifying your instruments specifically for MIDI. This also means that when Blue's MIDI system is enabled, Csound MIDI processing should be disabled for your project.

To configure what MIDI devices to use, go to the program Options settings (on OSX it is the application's Preferences, on other platofrms it is the Options menu item in the Tools menu) and under "Blue", go to MIDI. There you will see a list of MIDI devices connected to your computer. If you connected a device after starting Blue, you can rescan to find your MIDI device. In this window you will configure what MIDI devices you want to use with Blue, but these devices will not be opened for listening until you enable MIDI Input on the BlueLive toolbar in the main application.

Once you have configured what devices to use with Blue, return to the main program and enable MIDI input using the "MIDI Input" button, then start BlueLive. At this point, when MIDI notes are played, Blue will take the incoming note data, map it according to the values configured in the MIDI Input Panel window (available from the Windows Menu), and then generate notes and pass them to Csound.

For a MIDI note on, Blue will the channel number of the note and map it to the instrument in the orchestra manager by index. For example, if you have three instruments numbered 1, 3, and 5, notes for MIDI channel 1 will generate with instr 1, notes for MIDI channel 2 will generate with instr 3, and notes for MIDI channel 3 will generate with instr 5.

The MIDI key number as well as the velocity will be mapped according to the settings set in the MIDI Input Panel window. This allows for generating frequency, Csound PCH, and using Scala Tuning files to generate either frequencies or BluePCH format text, amongst other values.

Instruments that are intended to be used with Blue's MIDI system will have to work with a 5 p-field note format. This does not mean your instrument can only work with 5 p-fields, but rather that your instrument must support at least that. An example Blue project can be found in examples/features/blueLiveMidi.blue (if you are using the OSX application, you may need to explore the contents of the Blue.app program to find the examples folder).

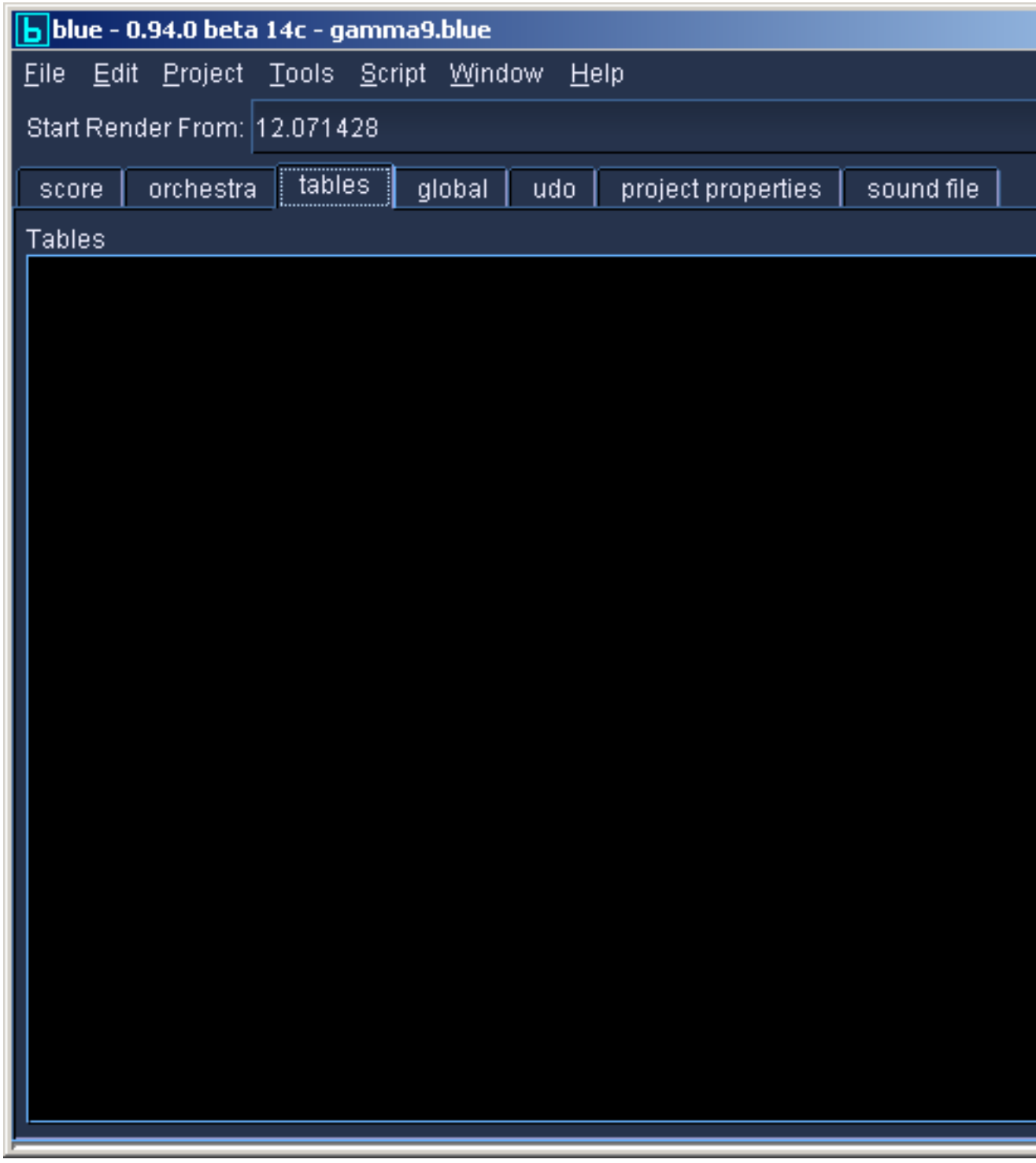
Exploring Ideas

Blue Live uses a subset of the same soundObjects that are available on the score timeline within the main editing window of Blue. Some possible ways you could explore ideas are:

- Trying out SCO ideas with GenericScore
- Testing your python functions with the PythonObject
- Testing your javaScript functions with the JavaScriptOObject

- Using Cmask, nGen, Common Music, and other programs with the external soundObject to try out things live

Tables Manager



Tables Manager

The tables manager contains a text area for global tables. Table statements put here will be immediately inserted above i-statements of the <CsScore> area of the generated CSD file.

Globals Manager

Csound Orchestra and Score text can be passed directly into a generated CSD by using the Global Orchestra and Global Score windows.

Global Orchestra

Anything here will be inserted into the <CsOrchestra> section of the .CSD file before anything generated by the Orchestra. Things like global variables, macros, or GUI instrument definitions may go here.

Global Score

Anything here will be inserted into <CsScore> section of the .CSD file before anything generated by the score timeline. the global score's processing is done outside of the score timelines, so any notes put here are not factored into values calculated from the timeline, like total duration, nor are any notes in the global score section translated like the timeline is when a different start time is used other than 0.0. for instance, if you use 2.0 as a start time, a note in the score timeline at 2.0 will be translated to start at 0.0, while a note with a start time of 2.0 in the global score section is not affected and will start at time 2.0.

There are also certain variables made available from Blue (Blue variables) that are useful for specific purposes. One of them is <TOTAL_DUR>. An example of its use is:

```
i20 0 [<TOTAL_DUR> + 10] .95 1 1000
```

The note given above is for a global effect(a global reverb unit). This note will always start at time zero regardless of when the score starts, and will always last as long as the duration of the generated score from the timeline plus 10 seconds. Because Blue variables are a text swap, the above use of the bracket notation that Csound uses was necessary. For a score with a 20 second duration, the above note would have been generated as:

```
i20 0 [20 + 10] .95 1 1000
```

Because of the Blue variable used, the note will correclty play regardless of when the score starts. If the the Blue variable was note used and was put as a note in the Score timeline, the reverb may not run.

Note

When Blue variables were created, Blue did not have a Mixer system implemented. The above system continues to function, the practice of using the Mixer is highly recommended. This feature is left in for legacy projects.

Tools

Code Editor

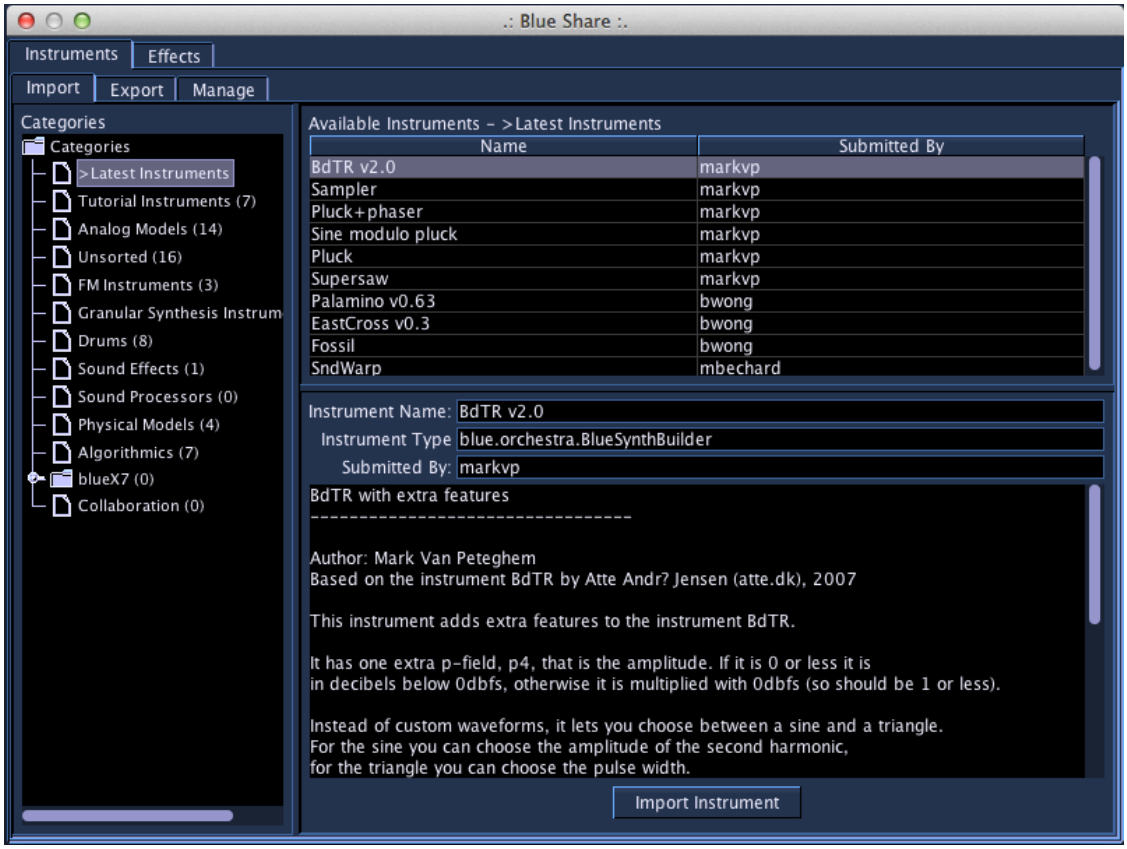
Blue uses the Netbeans Code Editor library for editing code. The editor is customized depending on where you are in the application, but there are common features shared. Some things which are available are:

- Syntax Highlighting
- Code Completion (ctrl-space)

- Line Numbering (enabled globally in program settings)
- Popup Menu (use right-mouse click) with additional options

You can adjust colors, fonts, and other settings within Blue's Program Options.

BlueShare



BlueShare

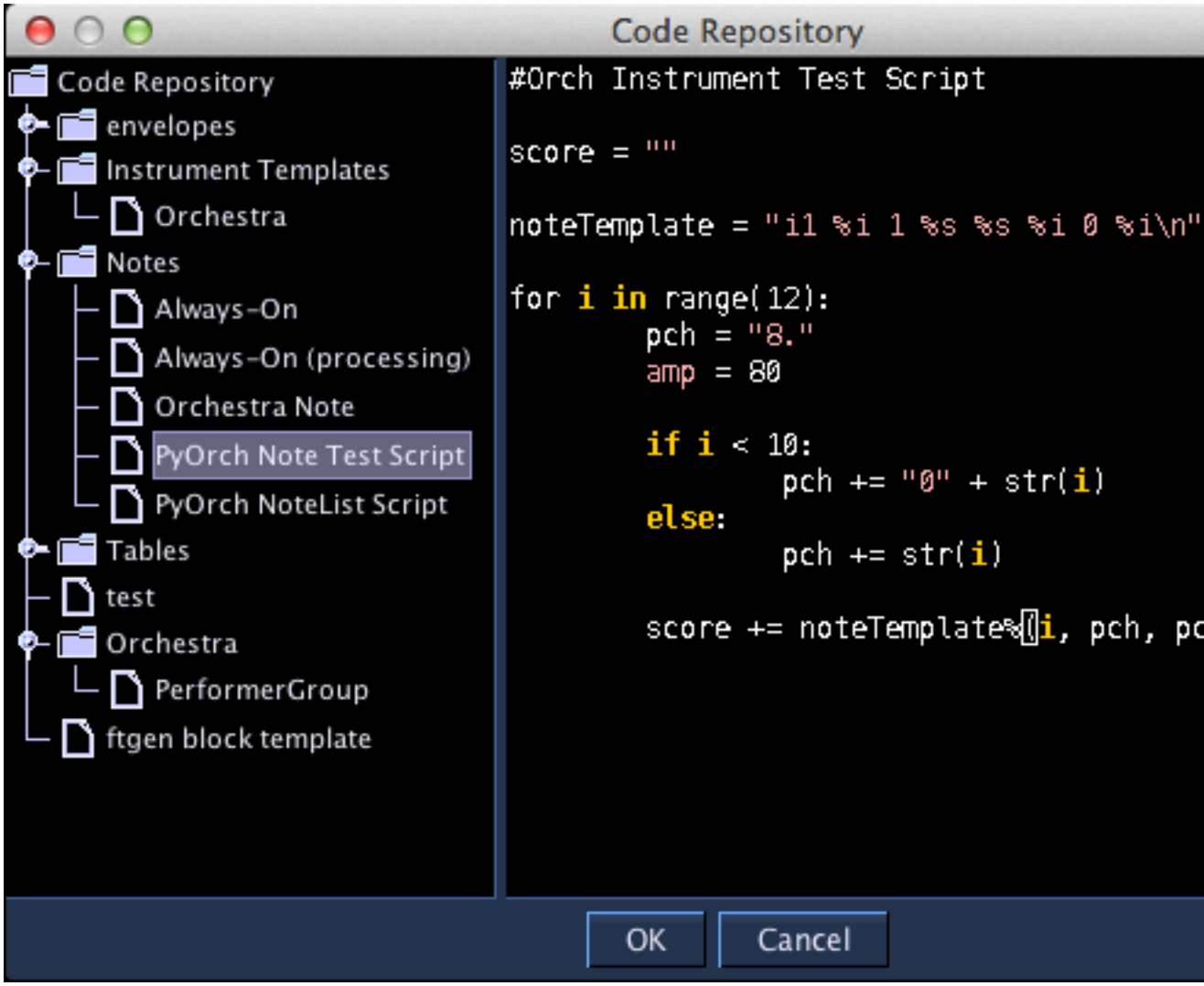
BlueShare is an online, in-program way to share instruments, effects, and SoundObjects with other Blue users. BlueShare does not require a user account to download items, but does require one for uploading. If you would like to share an instrument or effect, please sign up for an account at http://blue.kunstmusik.com

To use BlueShare, go to the Tools menu to open it up. Blue will contact the server to get a list of Instrument, Effects, and SoundObjects available. From there, you can browse categories, then select an item in the upper-right table to get more information. Once you find something you are interested to try, select "Import Instrument", "Import Effect", or "Import SoundObject". Blue will download the instrument, effect, or SoundObject into the User Instrument Library, Effects Library, or User SoundObject Library in a folder called "Imported Instruments", "Imported Effects", or "Import SoundObjects".

To upload and instrument or effect, switch to the Export tab. From there you will see a place to enter your username and password, a listing of instruments, effects, or SoundObjects from your libraries, a tree of categories to use for uploading, and a description box (pre-populated with the Comments field of your Instrument or Effect). You can then press the "Submit" button to send it to the server.

The manage tab allows you to pull down a list of your contributed instruments, effects, and SoundObjects. You can then remove the item from BlueShare using "Remove" button.

Code Repository



Code Repository

The Code Repository is where you edit the options that will come up from the code popup (the popup menu that is accessible by rt-clicking on most of the text areas in blue).

The Code Repository features a tree to organize your code. Rt-clicking on a folder will give you options to add a new code group, remove the current group, or add a code snippet to the group.

If you add either a code group or a code snippet it will be created with a default name. To edit the name, double click on the group or snippet and the area on the tree will change into a text field to edit the name.

To edit a snippet, simply click on the snippet in the tree and on the right a text area will appear. In that text area, edit the text that is associated with the code snippet.

When you are done editing your code repository, the next time you rt-click on a text area, the code popup will reflect the changes you have made, including any new code groups and snippets you've added.

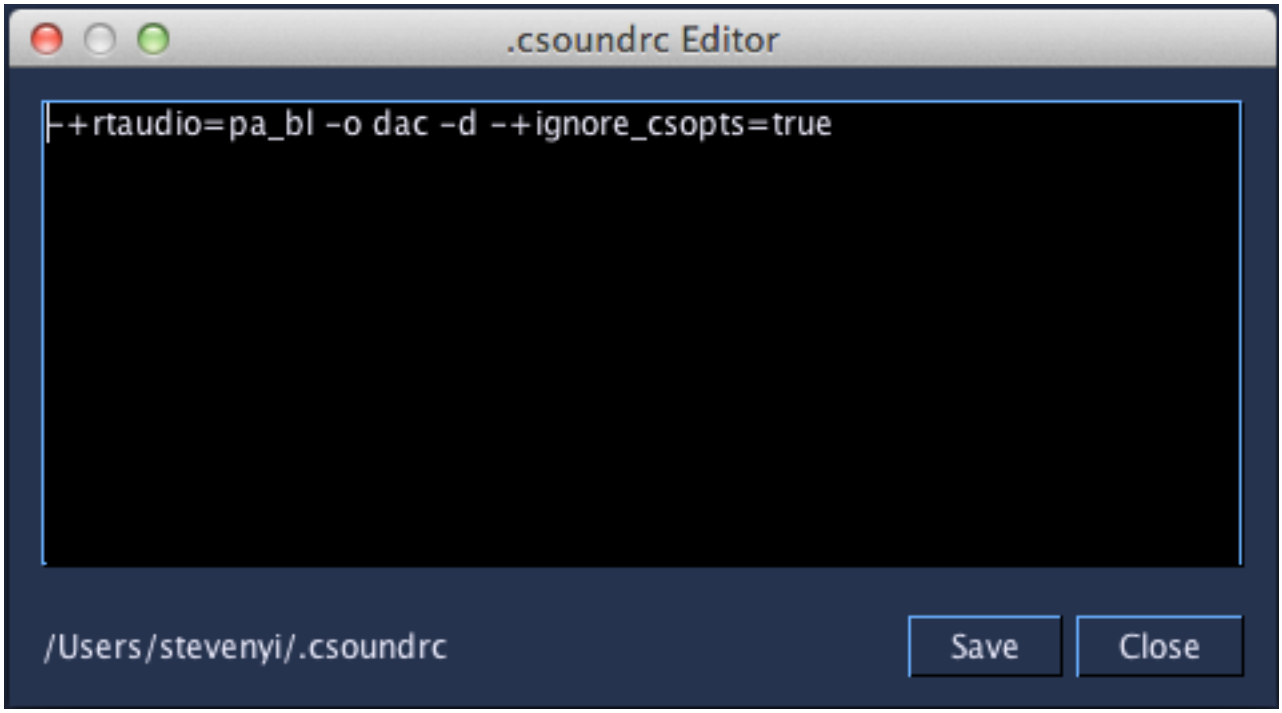
Effects Library

The Effects Library organizes Effects used with Blue's mixer. You can create and edit Effects, as well as organize Effects into Groups. The layout of the tree that contains the effects will be used when building the popup menu in the Mixer for selecting effects.

Note

This information will be further expanded and moved into a new section with the manual reorganization, planned for after Blue 2.3.0.

.csoundrc Editor



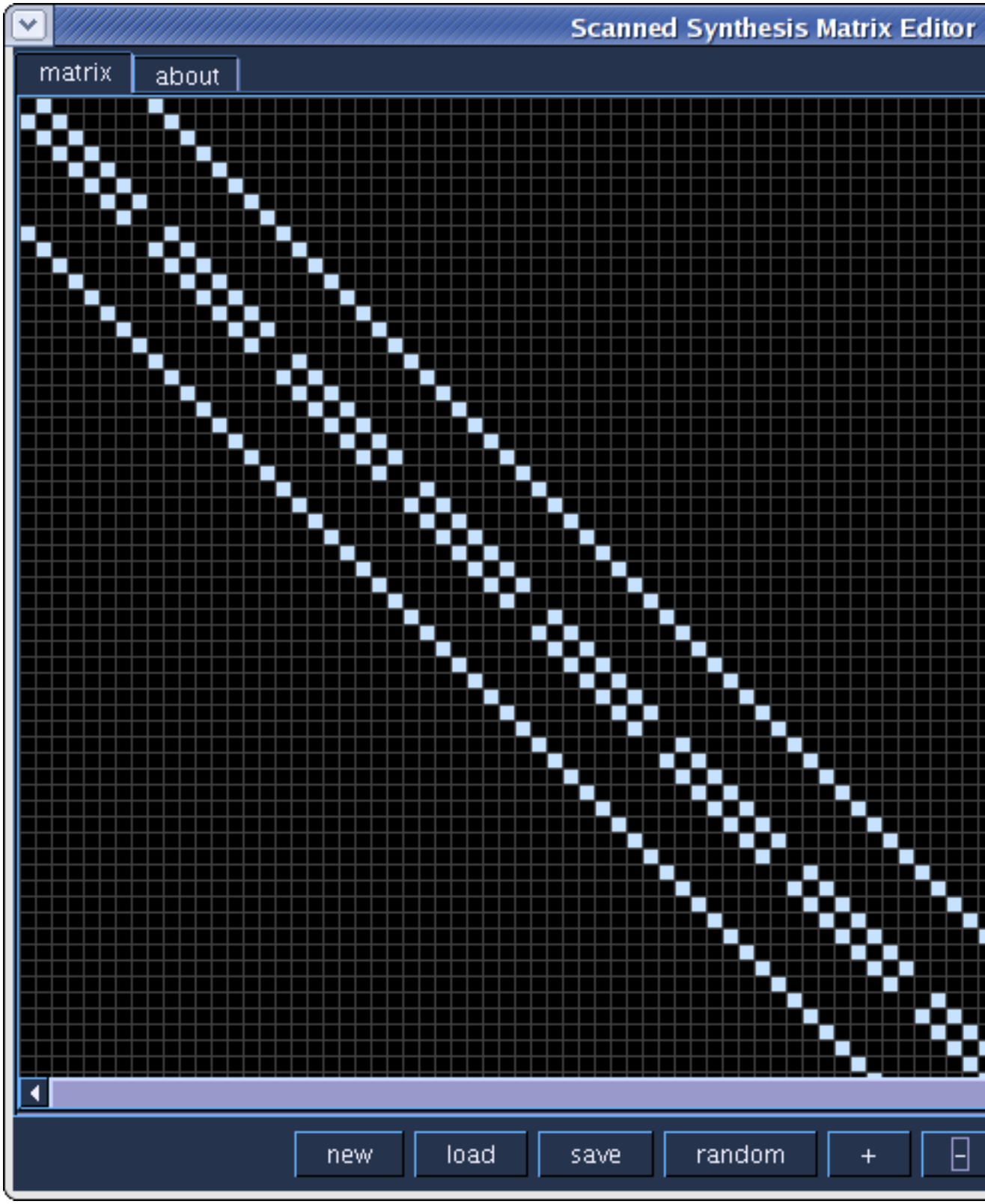
.csoundrc Editor

The .csoundrc Editor tool allows for editing the system-wide .csoundrc file. The editor is accessible from the Tools menu and launching it will open up the file pointed to by the environment variable CSOUNDRC or search for the file in $HOME/.csoundrc. If neither is found, the editor will open up with a file pointing to $HOME/.csoundrc.

When the dialog opens, the contents (if a file is found) is shown to edit in a simple text area. The absolute path of the file found is shown in the bottom left hand side. Pressing the Save button will save the contents and close the dialog, and pressing the Close button will close the dialog without changes.

For more information about .csoundrc, please view the Csound Manual entry for Command Line Parameter File (.csoundrc) [https://csound.github.io/docs/manual/CommandUnifileParFile.html].

Scanned Synthesis Matrix Editor



Scanned Synthesis Matrix Editor

The Scanned Synthesis Matrix Editor tools allows for editing and creating matrix files used by the Scanned Synthesis opcodes in Csound.

- Using "New" will ask what size matrix to create.
- For editing, click anywhere on the matrix. This will either turn on or turn off that square, setting the connection between the masses to either 1 or 0.
- After editing, press "Save" to save out the matrix to a file.
- You can also click the "Random" button to create a randomized matrix.

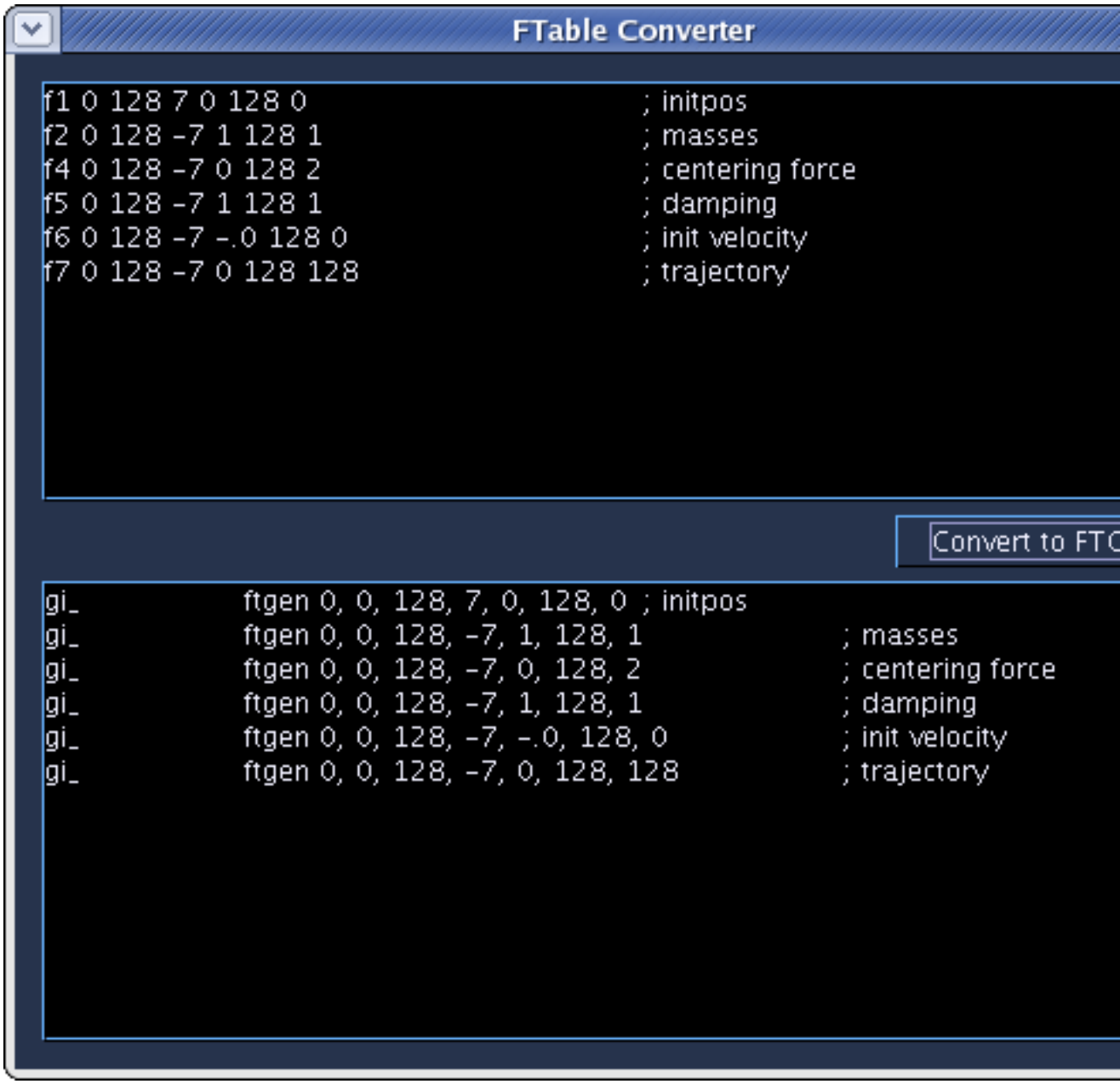
Sound Font Viewer

Sound Font Viewer

The Sound Font Viewer allows you to select SoundFonts from the file selection panel (on the left) and get a listing of the instruments and presets within that SoundFont.

To use, navigate using the file selection panel and double-click on a SoundFont. After processing is complete, the tables on the right side will be populated with a listing of instruments and presets held within that SoundFont.

FTable Converter



FTable Converter

The FTable Converter tool converts ftable statements into ftgen statements. When converted, the prior ftable statement numbers are ignored and ftgen statements are created using requested ftable number of

0. The generated ftgen statements generate with the values of the ftables set to "gi_" with the expectation that the user will fill out the rest of the name to be meaningful to them.

Conversion of ftable statements to ftgen statements is useful in creating ftables that will be referenced by global variables instead of hard-coded numbers.

To use, simply put your ftable statement text in the top text area and press the "Convert to FTGEN" button to convert. For example, if you use the following ftable statement text:

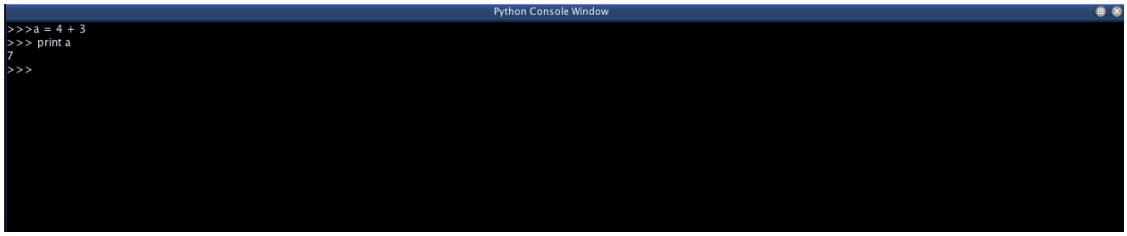
```
f1 0 128 7 0 128 0  ; initpos
f2 0 128 -7 1 128 1  ; masses
f4 0 128 -7 0 128 2  ; centering force
f5 0 128 -7 1 128 1  ; damping
f6 0 128 -7 -.0 128 0 ; init velocity
f7 0 128 -7 0 128 128 ; trajectory
```

you will get the following output:

```
gi_ ftgen 0, 0, 128, 7, 0, 128, 0   ; initpos
gi_ ftgen 0, 0, 128, -7, 1, 128, 1    ; masses
gi_ ftgen 0, 0, 128, -7, 0, 128, 2    ; centering force
gi_ ftgen 0, 0, 128, -7, 1, 128, 1    ; damping
gi_ ftgen 0, 0, 128, -7, -.0, 128, 0  ; init velocity
gi_ ftgen 0, 0, 128, -7, 0, 128, 128  ; trajectory
```

This work is based on the Steven Yi's FTable Converter web page utility located here [http://www.csounds.com/stevenyi/ftable.html].

Python Console



Python Console

The Python Console allows you to interactively write and execute python code using the Jython interpreter built in to Blue. Because Blue does not clear the environment of the interpreter between runs, the console can be useful to interactively inspect objects. Some ways you might use the console:

- Use `dir()` to inspect objects to see what methods and members they have
- Use `help()` to view the documentation on an object or class.
- After generating a score or using the "Test" button on a PythonObject, you can interactively test functions you have written by calling it using code in the console.
- Practice your python coding live.

The console works much like running python interactively in a console or terminal. At the ">" prompt, you can type in some code, then press enter to execute the code. All output from python functions (including

things like print statements in PythonObjects in the Score or Python NoteProcessors) will output to the Python Console.

Some useful shortcuts available while in the console:

Table 1.2. Shortcuts for the Python Console

Shortcuts	Description
ctrl-up/ctrl-down	cycle through previous commands used
ctrl-l	clear the console (also available from the rt-click popup menu

Other Features

AutoBackup and Recovery

Temporary backups of open projects are generated every minute for projects which have previously been saved to disk (new projects not yet saved are not backed up). If Blue quits unexpectedly, those files will remain, otherwise on normal closing of the file they are deleted. If on opening a file a backup if found with a date newer than the original file, the option to open from the backup or the original is given.

If the backup is chosen, it is required to use "Save As" to save over the original file or to a new file. At that point the backup is deleted.

If the original file is chosen, then the backup file will be overwritten the next time the autobackup thread runs.

Sound Object Freezing

Introduction

Sound Object Freezing allows you to free up CPU-cycles by pre-rendering soundObjects. Frozen soundObjects can work with global processing instruments, and files are relative to the directory the project file is in, so can be moved from computer to computer without problem. Frozen soundObjects can be unfrozen at anytime, returning the original soundObject and removing the frozen wave file.

To freeze a soundObject, select one or many soundObjects on the timeline, rt-click on a selected soundObject, and then select "Freeze/Unfreeze SoundObjects". To unfreeze, select one or many frozen soundObjects and select the same menu option.

On the timeline, if your soundObject rendered wave is longer in duration than the original soundObject's duration (as is the case if you have reverb processing), the frozen soundObject's bar will graphically show the difference in times with two different colors.

Note: As currently implemented, when Blue goes to freeze soundObjects it may appear to be frozen, but messages will continue to appear in the console showing that csound is rendering the frozen soundObjects. Future versions will be more polished.

How SoundObject Freezing Works

1. An soundObject is selected
2. Using the same project settings (all of the instruments, tables,global orc/sco, etc.) but not scoreTimeline generated sco, Blue generates the sco for the selected soundObject and produce a temporary .csd file

3. Blue runs csound with "csound -Wdo freezex.wav tempfile.csd" where the x in freezex.wav is an integer, counting up. This wav file is generated in the same directory that the projectFile is located.

4. Blue replaces the soundObject in the timeline with a FrozenSoundObject. The FrozenSoundObject keeps a copy of the original soundObject (for unfreezing), as well as shows the name of the frozen wav file, the original soundObject's duration, and the frozen wav file's duration (not necessarily the same, as is the case if using global reverb, for example).

5. When you do a render of the entire piece now, the frozen sound object generates a very simple wav playing csound instrument that will play the rendered wav file as-is. The instrument looks something like:

```
aout1, aout2    diskin    p4
                outs      aout1, aout2
```

   and the FrozenSoundObject only generates a single note that has the start-time, the duration of the frozen wav file, and the name of the file. This will end up playing the soundFile exactly as if the SCO for the original soundObject was generated. This also bypasses any routing to global sound processing, as if you had any of these effects originally, the would be generated as part of the frozen file.

Notes

- You can select multiple soundObjects and batch freeze and unfreeze -the generated wav file may be longer than the original soundObject, due to global processing instruments (like reverb, echo, etc.) This is taken into account.

- The freezing system does *not* work for all graph toplogies. If you're using soundObjects with instruments used as control signals, this won't work unless the notes for the instruments they are controlling are alsoin the same soundObject. I.e. I have one soundObject that has only notes that affect global variables, while I have one instrument thatuses those global variables. This could work though if you repackage the set of soundObjects into a polyObject. Probably best to generalize as:

  - Your soundObject must be self-contained

  - All sound output from instruments go directly out or piped through always-on instruments, that most likely should take advantage of the <total_dur> variable, as well as the new <processing_start> variable (more about this when I release, but together with freezing, this lets you set the start time of always-on instruments to the first time where non-frozen soundObjects occur, so if the first half of your piece is frozen and you're unfrozen stuff is in the second half, you don't need always on instruments to be turned on until the second half as the first half is routed to outs

- This system is tested with 2-channel pieces. I'm not sure if this will work with higher number of channels, but I don't see why it wouldn't.

- Changing the number of channels on the project after a freeze may cause Csound errors when rendering the frozen soundObject (can be remedied by unfreezing and refreezing)

- Frozen files are referenced relatively to the project file, so you are free to move your project directory around or rename it and the frozen files will work fine.

Auditioning SoundObjects

You are able to quickly audition individual SoundObjects or groups of SoundObjects on the timeline. This can be very useful to test out single SoundObject ideas or see how a few different SoundObjects

sound together without having to mute/solo different SoundLayers. To use this feature, select desired SoundObjects for testing, right-click on a selected SoundObject, and from the SoundObject popup menu choose "Audition SoundObjects". You can also initiate the audition by using the ctrl-shift-a shortcut or by going to the Project menu and choosing "Audition Selected SoundObjects". To stop auditioning, click anywhere on the Score timeline.

Importing ORC/SCO and CSD Files

Blue is able to import ORC/SCO and CSD files and set up a Blue project file, with instruments parsed out and put into the orchestra manager, project settings set, etc. Currently, there are three options to choose from when importing a CSD file, all relating to how you would like to import the notes from the CsScore section:

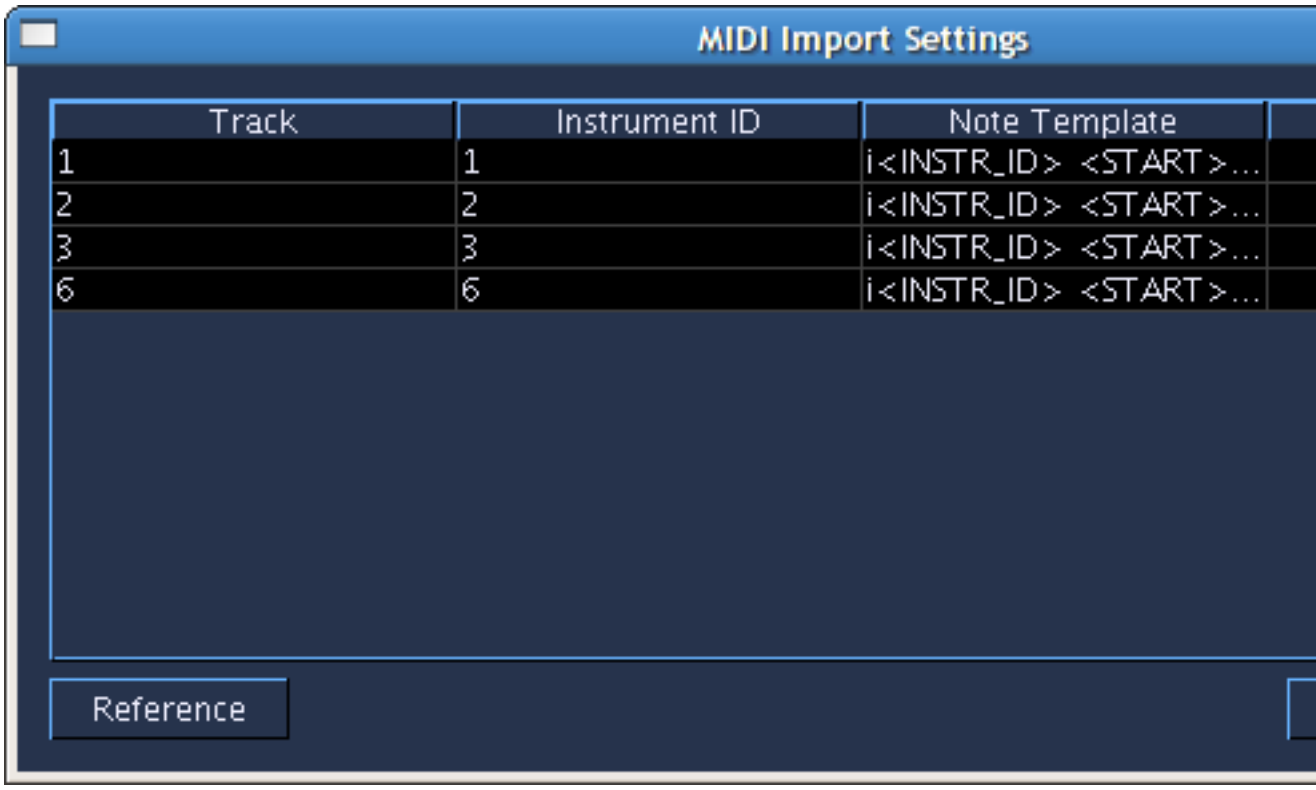
- Import Score to Global Score - all score goes to the Global Score section
- Import All Score to Single SoundObject - all score goes into a single GenericScore SoundObject. The duration of the soundObject will be set to the duration of the imported score. This is broken up into different soundObjects and layers if sections (s-statement) are found in the score.
- Import Score and Split into SoundObjects by Instrument Number - score is split by instrument number and a different GenericScore soundObject will be created for each block of score text per instrument. This is broken up into different soundObjects and layers if sections (s-statement) are found in the score.

Notes

- From the CsScore, Blue will only import f-, i-, and s- statements. Other score statements are not currently supported at this time.

Importing MIDI Files

Blue is able to import MIDI files and set up a Blue project file from the note information in the MIDI file, using the settings given by the user. To import a MIDI file, choose the "Import MIDI File" option from the File menu. Next, using the file dialog to locate the MIDI file to import. After selecting the desired file, Blue will show the following MIDI Import Settings dialog for you to configure how you would like to import the MIDI note information. (Note: Blue will only show information for tracks where note data was found.)



MIDI Import Settings

The table column information is as follows:

MIDI Import Settings

Track — The original MIDI track number to which this setting is to be applied to. This column is not editable is for reference purpose only.

Instrument ID — The Csound instrument ID to use for this track. This will replace the <INSTR_ID> key within the note template. This value is treated as a string to allow users to assign the track information to Csound named instruments. If one is doing so, one must quote the name, i.e. use "trumpet" instead of trumpet (without quotes), otherwise the output will not be legal Csound SCO. Default value is the number of the MIDI track.

Note Template — Template note text to use for generating Csound SCO from the MIDI data. The default note template is "i<INSTR_ID> <START> <DUR> <KEY> <VELOCITY>". By having note templates, the user can massage the note information to work with any number of pfields that their instruments require.

The following values are allowed in the note template:

Table 1.3. Key Values

Shortcuts	Description
<INSTR_ID>	The instrument ID assigned in the track settings.
<START>	Start Time of Note

Shortcuts	Description
<DUR>	Duration of Note
<KEY>	MIDI key number
<KEY_PCH>	MIDI key number as Csound PCH
<KEY_OCT>	MIDI key number as Csound OCT
<KEY_CPS>	MIDI key number as CPS
<VELOCITY>	MIDI velocity number
<VELOCITY_AMP>	MIDI velocity number as amplitude

The button labelled "Reference" on the dialog will pop open the above information for quick reference of the allowable replacement keys for note templates.

Trim Time — This option will shift the generated SoundObject to the time of the first note and then take the generated notes for the track and shift them all so that the first note starts at time 0 so that there is no empty time at the beginning of the track's note information.

After finishing configuring settings for the imported MIDI data, Blue will generate the notes with one SoundLayer per MIDI track, and on each SoundLayer it will contain one GenericScore SoundObject containing the converted MIDI score.

Note

The current implementation does not handle cases where there are overlapping notes of the same MIDI note number within the same track and results are unpredictable. Also, only MIDI files where time is PPQ is supported at the moment (non-SMPTE). Users wanting support for either of these cases or have other ideas they would like implemented are requested to make feature requests on the Blue mailing list or to use the help menu "Request a Feature" option.

Blue Variables

Blue treats special text as special Blue variables. Below is a list of those variables.

Table 1.4.

Variable	Value
<TOTAL_DUR>	Duration of the generated score from the timeline (Available in Global SCO text area.)
<INSTR_ID>	Replaces with instrumentId; if the instrument is a named instrument, value is quoted. Generally used when creating notes.(Available within instruments and some SoundObjects.)
<INSTR_NAME>	Replaces with instrumentId; if the instrument is a named instrument, the value is not quoted. Generally used when working with ORC code to give a variable a unique ID, i.e. "gk_var<INSTR_NAME>".(Available within instruments and some SoundObjects.)
<RENDER_START>	The start time of rendering. Does not take into account time warping of score. (Available in Global SCO text area.)

Variable	Value
<RENDER_START_ABSOLUTE>	The start time of rendering. Takes into account time warping of score. (Available in Global SCO text area.)
<PROCESSING_START>	The time value when always-on effects instruments need to start. Calculated as the time value of the first soundObject that is not a FrozenSoundObject or Comment. (Available in Global SCO text area.)

Command Line Options for Blue

To view the options that Blue has from the commandline, type "blue --help". After that, you should see information printed to the console. Some of these flags are used by the Netbeans Platform that Blue is built upon. The following are ones Blue uses itself:

```
-c, --compile <arg>
-o, --output <arg>
```

The flags above allow for commandline compilation of a .blue project into a CSD. Both flags must be set to work. An example of usage is:

```
blue -csomeFile.blue -ooutput.csd
```

This will use somefile.blue and produce output.csd. The generated CSD will use the Disk Render settings for the project. Using the commandline option for compiling .blue projects is useful for automating builds of projects using a build system like Make, Rake, Ant, or other tool.

Chapter 2. Concepts

Rendering

Introduction

Blue offers a number of ways to render a project, each method useful for different purposes you may have. The following section explains the ways which Blue is able to render a .blue project file as well as possible use-case scenarios to guide you on how that feature may be useful.

Render Methods

Render in Real-Time

Rendering in uses the project's Real-Time render settings to render the project in realtime to the users soundcard. This is the most common rendering one will likely use. To use this, you can press the play button at the top of the window, use the "Project->Render/Stop Project" menu option, or use the F9 shortcut key.

Generate CSD to Screen

Blue's general method of communicating with Csound is by generating a CSD file and calling Csound to run it. For debugging purposes, it is often useful to see if what you think you are doing inside Blue is matching the generated CSD file. You can generate the CSD to screen by using the "Project->Generate CSD to Screen" menu option, or by using the ctrl-shift-g shortcut.

Generate CSD to Disk

If you'd like to generate a CSD from the current project to a file, you can do so by using the "Project->Generate CSD to File" menu option, or by using the ctrl-g shortcut.

Render to Disk

The Render to Disk option will use the project's Disk Render Options from its project properties to generate a soundfile on disk (format of file depends on how user configures commandline options). I've found this useful for rendering a wav to disk and converting to MP3 to listen to on the go so I can review my current work. To render to disk, you can do so by using the "File->Render to Disk" menu option or use the ctrl-shift-F9 shortcut.

Render to Disk and Play

The Render to Disk and play option will use the project's Disk Render Options from its project properties to generate a soundfile on disk and after finishing, play the generated file using Blue's soundfile player, located in the SoundFile manager tab. This feature is useful if you're working on a piece that has processing demands beyond what your computer is capable of in realtime, so you can have it render to disk first and then play, which will give very smooth playback. It is also useful if you render in realtime with lower quality settings but use higher quality settings for final project output, as you can occasionally test what the output would be like using this feature. To use this option, you can use the "File->Render to Disk and Play" menu option or use the ctrl-F9 shortcut.

Render to Disk and Open

The Render to Disk and Open option will use the project's Disk Render Options from its project properties to generate a soundfile on disk and after finishing, open the generated file using the configured command with the program options. This command may differ from the external command to use with "Render to Disk and Play". To use this option, you can use the "File->Render to Disk and Open" menu option.

SoundObjects

Introduction

The concept of SoundObjects is the foundation of Blue's design in organizing musical ideas. This section will discuss what is a SoundObject, how this idea is implemented in Blue, and strategies on how to use the concept of SoundObjects in organizing your own musical work.

What is a SoundObject?

SoundObjects in Blue represent a *perceived sound idea, whether it be a single atomic sound event or aggregate of other sound objects* . A SoundObject on the timeline can represent many things, whether it is a single sound, a melody, a rhythm, a phrase, a section involving phrases and multiple lines, a gesture, or anything else that is a perceived sound idea.

Just as there are many ways to think about music, each with their own model for describing sound and vocabulary for explaining music, there are a number of different SoundObjects in Blue. Each SoundObject in Blue is useful for different purposes, with some being more appropriate for expressing certain musical ideas than others. For example, using a scripting object like the PythonObject or JavaScriptObject would service a user who is trying to express a musical idea that may require an algorithmic basis, while the PianoRoll would be useful for those interested in notating melodic and harmonic ideas. The variety of different SoundObjects allows for users to choose what tool will be the most appropriate to express their musical ideas.

Since there are many ways to express musical ideas, to fully allow the range of expression that Csound offers, Blue's SoundObjects are capable of generating different things that Csound will use. Although most often they are used mostly for generating Csound SCO text, SoundObjects may also generate ftables, instruments, user-defined opcodes, and everything else that would be needed to express a musical idea in Csound.

Beyond each SoundObject's unique capabilities, SoundObjects do share common qualities: name, start time, duration, end time. Most will also support a Time Behavior (Scale, Repeat, or None) which affects how the notes generated by the SoundObject will be adjusted--if at all--to the duration of the SoundObject. Most will also support NoteProcessors, another key tool in Csound for manipulating notes generated from a SoundObject. All SoundObjects also support a background color property, used strictly for visual purposes on the timeline.

PolyObjects

Michael Bechard
2005
Revision History
Revision 1.0 2005.05.05
First version of article.

Introduction

PolyObjects are, in my opinion, one of the most powerful tools in blue. They provide encapsulation of any grouping of SoundObjects, any way you like, into one logical SoundObject. Once encapsulated, the rest of blue's powerful features (namely NoteProcessors) can be leveraged against the PolyObject, and that is when one really begins to realize the benefits of using PolyObjects.

Basic

Explanation

To repeat what is stated in the reference documentation, a PolyObject can be seen as a container of other SoundObjects. It contains its own series of SoundLayers, just like the main score, and, like the main score, one can add as many SoundLayers and SoundObjects as one likes to a PolyObject.

Think of the main score as a big box that holds sounds. Now, we can put whatever sounds we want in the box, but we can also put smaller boxes inside the main box; these smaller boxes are our PolyObjects. We can put sounds inside the smaller boxes, just like we can in the big box, and we can arrange them in the same manner too. When we put sound A in the upper-right corner of a smaller box, it will stay in that corner no matter where we move our box inside of the bigger box. However, it is the position and arrangement of the sounds themselves relative to the main score that is important and is, ultimately, the reason to use PolyObjects. With sound A in the upper-right corner of our small box, its relative position in the main box will be dependent on the smaller box's position in the big box. Keep in mind, too, that we can put boxes inside of boxes inside of boxes, as many layers as we like. And when you realize that you can change the sounds inside of any box by applying a NoteProcessor to it (like making the box metal instead of cardboard), you begin to see the power of using PolyObjects.

Usage

The way it works is, you create a PolyObject in a SoundLayer like you would any other SoundObject; by right-clicking on the SoundLayer and selecting "Add New PolyObject;" it should be at the very top. After doing this, the PolyObject is empty and will not generate any notes in its current state. You have to double-click on it in order to edit the contents of the PolyObject. Once you're editing the empty PolyObject, the score editor looks a lot like it does when you're editing an empty blue score. That's because the main score is just one big container for notes, like PolyObjects (in fact, in the code, the main score IS a PolyObject).

You can see which container you are currently editing by looking at the buttons just underneath the tabs for "score," "orchestra," etc. When you're editing the main score, there will only be one button there called "root;" you're editing the root container. When you're editing a PolyObject called "Group A," you'll see two buttons; "root" and "Group A." Pressing any of these buttons will change the view to edit that container.

Once a PolyObject has been created, you can add and arrange SoundObject on it just like you do in the main score. Each PolyObject is going to have its own Snap and Time Display settings too, so you may want to set those accordingly. Add SoundLayers just as you would in the main score too.

After you've added all of the notes you wish to the PolyObject, click on the "root" button and feel free to move the PolyObject around in the main score, just as you would any other SoundObject. Add NoteProcessors too, or alter the Time Behavior; these changes, in turn, will be applied to the contents of the PolyObject.

Advanced

Before tackling some of the more advanced concepts of PolyObjects, a couple of clarifications must be made. When this document refers to editing the actual PolyObject, that means editing it as a SoundObject

(moving it around on the timeline, stretching it, editing its properties, etc.). This is very different from editing the contents of said PolyObject. The distinction will be made in this section whenever applicable.

Time

PolyObjects can have their Time Behavior changed like any other SoundObject when one is editing the actual PolyObject. This is very important to keep in mind, because your sounds within the PolyObject may span 10 beats, but if you leave the Time Behavior settings for the PolyObject to the default (Scale), they may in actuality span only two beats, depending on the size of your PolyObject. One can test this with the Test button: shrink a PolyObject that has the Scale Time Behavior and the notes within it will shrink as well.

No Special Time Behavior

There are a couple of easy ways to make sure that the sounds you place in a PolyObject remain to scale in the parent container, or the root score. One method is to right-click on the PolyObject itself and click "Set Subjective Time to Objective Time." This will change the length of the PolyObject to be equal to the beat at the end of the last SoundObject within it. In other words, it will expand or contract to the actual total length of all of the notes within it. The other method is to change the Time Behavior of the PolyObject to None. Blue will not do any processing of the length of the notes within the PolyObject; they'll just be played exactly the way they are in the PolyObject.

There are advantages and disadvantages to each method. The first is good because it allows one to see the actual length of the PolyObject within the main score, but you have to remember to click "Set Subjective Time to Objective Time" each and every time you make a change to the PolyObject's contents. If you set the Time Behavior to None, the length of the PolyObject may appear to be two beats, but it may actually be ten or 20 or whatever; you don't know until you edit the contents of that PolyObject. However, with a Time Behavior of None, you can edit the contents of the PolyObject frequently and drastically without having to worry if your changes will be accurately represented in the CSD rendering. A good way to work may be to set a new PolyObject's Time Behavior to None and edit the contents of it as much as you like. Then, when it sounds satisfactory, you can click "Set Subjective Time to Objective Time" on it, so that you may edit the rest of the score while having an accurate picture of the actual length of the PolyObject. Keep in mind that you may use the "Set Subjective Time to Objective Time" feature on any SoundObject with any Time Behavior setting.

Repeat/Looping Time Behavior

One can lay down a series of sounds in a PolyObject and setup the PolyObject to loop them. The Time Behavior of the PolyObject itself needs to be set to Repeat, and the notes will loop for as long as the PolyObject's length. When the PolyObject's contents actually start looping in relationship to each other is determined by the Repeat Point; one can edit the Repeat settings in the property box of any SoundObject. The Repeat Point specifies the beat at which the group of notes will begin after the previous note group begins. If the Use Repeat Point is not checked, the next note group will begin immediately after the previous note group's last note ends.

An example would be a PolyObject whose contents are three one-beat notes played back-to-back:

```
i1 0.0 1.0 3 4 5
i2 1.0 1.0 3 4 5
i3 2.0 1.0 3 4 5
```

If one set the Time Behavior to Repeat without using a Repeat Point and stretched the length of the actual PolyObject to six beats, six one-beat notes would play back-to-back, like this:

```
i1 0.0 1.0 3 4 5
i2 1.0 1.0 3 4 5
i3 2.0 1.0 3 4 5
i1 3.0 1.0 3 4 5
i2 4.0 1.0 3 4 5
i3 5.0 1.0 3 4 5
```

However, if one were to change the PolyObject to use Repeat Points and set the repeat point to 2.5, this would be the result:

```
i1 0.0 1.0 3 4 5
i2 1.0 1.0 3 4 5
i3 2.0 1.0 3 4 5
i1 2.5 1.0 3 4 5
i2 3.5 1.0 3 4 5
i3 4.5 1.0 3 4 5
i1 5.0 1.0 3 4 5
```

Note that there are more notes being played this time. Why? Because with a Time Behavior of Repeat, blue will try to fit as many notes that it can into the PolyObject using the repeat settings given, without violating the order and length of the PolyObject's note arrangement. With the previous example, blue played the second group of three notes at beat 2.5, and realized it could squeeze in a partial third loop. This third loop begins at beat five, since that is equivalent to beat 2.5 in the second loop, which is our Repeat Point. It inserted the first note there and realized it couldn't fit any more notes, since the next note would begin at beat six and play until beat seven; that's longer than the length of our PolyObject, which is six.

NoteProcessors and PolyObjects

NoteProcessors are great when used in conjunction with PolyObjects. They can be very powerful, but you have to be careful of a few gotcha's.

P-fields and PolyObjects

One thing to be aware of with PolyObjects and NoteProcessors are p-fields. The user must make sure that the p-field(s) that is being operated on by the Note Processor(s) is the same for all sounds within the PolyObject. If one has a lot of typical notes in a PolyObject where p4 equals the pitch, and one errant note where p4 equals phase, for instance, things could get disastrous if one tries to apply a PchAddProcessor to the whole PolyObject. Please refer to the Best Practices section for more information.

Time Behavior and NoteProcessors

Everything that is to follow in this section can be summed up in one important fact: NoteProcessors get applied to SoundObjects before the Time Behavior does. What this means is, if you have a PolyObject with Repeat Time Behavior and set it up to loop twice, and have a LineAddProcessor on that same PolyObject, you will have a group of notes looped twice with the LineAddProcessor behavior applied twice; that is, applied individually to each looped group of notes.

So, to borrow from the example in Repeat/Looping Time Behavior, our group of three notes with no NoteProcessor and no Repeat Point would render like this:

```
i1 0.0 1.0 3 4 5
i2 1.0 1.0 3 4 5
```

```
i3 2.0 1.0 3 4 5
i1 3.0 1.0 3 4 5
i2 4.0 1.0 3 4 5
i3 5.0 1.0 3 4 5
```

If I add a LineAddProcessor to p-field four, starting from zero and going to two, here's what the result would look like:

```
i1 0.0 1.0 3.0 4 5
i2 1.0 1.0 4.0 4 5
i3 2.0 1.0 5.0 4 5
i1 3.0 1.0 3.0 4 5
i2 4.0 1.0 4.0 4 5
i3 5.0 1.0 5.0 4 5
```

This may be the desired effect, and it may not. In order to apply a NoteProcessor after the Time Behavior has been applied, take the following steps:

1. Apply the Time Behavior to you PolyObject, with no NoteProcessor on it.
2. Right-click on the PolyObject and select "Convert to PolyObject." What this will do is embed your PolyObject in yet another PolyObject.
3. Edit the Properties of the new PolyObject and apply the NoteProcessor to it.

If one applies the previous steps to the example cited above, the rendered notes look like this:

```
i1 0.0 1.0 3.0 4 5
i2 1.0 1.0 3.4 4 5
i3 2.0 1.0 3.8 4 5
i1 3.0 1.0 4.2 4 5
i2 4.0 1.0 4.6 4 5
i3 5.0 1.0 5.0 4 5
```

One thing the user will have to keep in mind, though; when referring to beats, those beats will be applicable to the notes as they are after the Time Behavior change. So, in the previous example, in order to get those results, one has to change the ending beat in the LineAddProcessor to five instead of two, because five is the beat at which the last note plays in the looped sequence.

Multi-layered Polyobjects

As mentioned previously, PolyObjects can contain other PolyObjects. The thing to remember when doing this is that each PolyObject's settings, NoteProcessor's, etc. are local to itself. For blue, the notes generated from a PolyObject SoundObject are the same as the ones generated from a GenericScore or any other type of SoundObject, and can be manipulated in the same way.

Best Practices

Separate notes with dissimilar behavior into their own PolyObjects

Let's say you're making a PolyObject with lots of regular notes whose p4 field represents the pitch. At some point you also want to insert a note that affects something a bit more irregular, like the phase of the

other notes, and its p4 field represents, say, the amount of variance in the phase. It would be prudent to separate that irregular note away from the notes in this PolyObject in order to keep your expectations about the p-field of the PolyObject as a whole consistent. Without the irregular note, you can safely say that p4 of PolyObject X (or whatever its called) represents the pitch of all of the notes it contains. However, with that extra note in it, p4 has no consistent meaning in PolyObject X, and you can't safely apply NoteProcessors to it that affect that p-field.

Group co-reliant notes together into PolyObjects

Co-reliant notes, in this case, would be notes that rely on each other to make their sound. For instance, if you had a group of notes that fed their output to a zak channel, then another note to process and play that channel's sound, those notes would be co-reliant. The reason to do this is to try to make each SoundObject in the main score independent so that one can move, remove, and edit each of them without worrying about affecting other notes. The PolyObjects, effectively, begin to represent one atomic sound in your score, making it easier to experiment with.

Doing this also allows one to reliably freeze PolyObjects. If one tried to freeze a PolyObject that did nothing but write its output to a zak channel, the resulting sound file would be dead silence, and the score would never work (since a frozen sound does nothing but read and play a sound file).

Wait, what about separating dissimilar notes?

That's still possible with co-reliant but dissimilar notes. What one would do, to use the example in the previous section, is make a PolyObject that contained the instrument processing the zak channel output and another embedded PolyObject. The embedded PolyObject would then contain all of the actual notes writing to the zak channel. Now what you can do is apply all kinds of NoteProcessors to the embedded PolyObject, then freeze or do whatever to the parent PolyObject in a reliable fashion.

Try to group logical phrases in your music into PolyObjects

It's much easier on the user as a composer to arrange the notes this way. This enables one to see opportunites for reuse of certain musical phrases, and implement that repetition easily by adding said PolyObjects to the SoundObject Library and copying instances wherever they're needed.

Try to keep the length of the PolyObject representational of its actual length

This only really applies if the Time Behavior being used is None, otherwise a PolyObject's length is always going to represent accurately its length in the composition. Remember to use "Set Subjective Time to Objective Time" as needed.

Debugging

When things don't work out the way you expected them too (and they won't), a great tool to use to help track down the problem is the "Test" button. This button renders the i-statements that any given SoundObject will generate, and it comes in very useful with PolyObjects. So, if you've got a PolyObject that won't render when you Test it, drill down into it and test its constituent notes, Testing and drilling down into its sub-PolyObjects as well. You'll eventually find the cause. Examine NoteProcessors and the Time Behaviors of each SoundObject; these can be common culprits with sound problems too.

If one follows the best practice "Group co-reliant notes together into PolyObjects," this also makes it easier to debug problems. You can isolate PolyObjects in the main score easily by soloing that SoundLayer, etc.

(since they won't rely on other SoundObjects for their sound) and investigate the problem that way; many times, you'll find that the problem stems from the PolyObject not being as independent as you'd thought.

NoteProcessors

Introduction

NoteProcessors are a powerful tool for manipulating notes. Each type of NoteProcessor serves a different purpose to adjust values for pfields. They can be used to apply musical effects to a group of notes; some possible uses include humanizing rhythm, added crescendo/decrescendo, reassigning notes to different instruments, applying time curves, and transposition of material.

NoteProcessors can be applied on a per-SoundObject basis. They can also be applied to an entire Layer to affect all SoundObjects in that layer, as well as applied to a LayerGroup or to the entire Score.

Usage

You can add NoteProcessors to SoundObjects by using the SoundObject Properties window. You can add them to Layers the support them by using the "N" button the layer's header panel. To apply them to a LayerGroup or Score, right click the Root Score in the Score Bar and use the popup menus to choose which item you'd like to apply NoteProcessors to.

Role in Score Generation

NoteProcessors are applied after the notes of a SoundObject, Layer, LayerGroup, or Score are generated and before time behavior is applied. Processing starts with the first NoteProcessor in the chain and the results of that are passed down the chain.

SoundObject Library

Introduction

The SoundObject Library is a place to store SoundObjects either project-wide or program-wide. The project-wide library can be used to store a SoundObject that you may want to user later but it also enables Instance SoundObjects to be made. Instances of SoundObjects point to the SoundObject in the library and when the instance generates its score, it will call the SoundObject in the library to first generate score and then apply its own properties and NoteProcessors to the generated score. Updating a SoundObject in the library will then update all instances of that SoundObject. This feature is useful to represent the idea of a motive, with instances of the motive allowing to have transformations by use of NoteProcessors.

The program-wide library (called "User SoundObject Library" in the interface) allows one to build up a catalog of reusable SoundObjects. SoundObjects such as Sound and ObjectBuilder allow users to build highly configurable sound and score generators that have a high degree of reusability. The program-wide library is also where SoundObjects are exported from and imported to with BlueShare.

Usage

The general pattern of usage for the project-wide SoundObject Library entails:

1. Add SoundObject to the Library. This is done by selecting a SoundObject, right-clicking the SoundObject to open up the popup menu and selecting "Add to SoundObject Library".

2. After doing this, your SoundObject will have been added to the library and the SoundObject on the timeline will have been replaced with an Instance SoundObject which will be pointing to the SoundObject now in the library.

3. At this point, the user can now take advantage of the library by making copies of the instance object on the timeline and pasting in more instances. These instances can be placed anywhere, have different durations and time behaviors, as well as have their own individual NoteProcessors. This allows expressing ideas such as "This is an instance of the primary motive (SoundObject in the library) but transposed up a major 3rd, retrograded, and inverted", or an idea like "I've got drum pattern A in the library and I have instances of it here and here and ...".

   ### Note

   When copying and pasting Instance SoundObjects, they are all pointing to the SoundObject in the library.

4. You can also then make instances of SoundObjects in the library by opening up the SoundObject Library dialog (available from the Window menu or by using the F4 shortcut key). There you have the following options:

Copy	This makes a copy of the selected SoundObject and puts it in the buffer. This is a copy of the original SoundObject and not an Instance. After copying to the buffer, you can paste as normal on the timeline.
Copy Instance	This makes a Instance of the selected SoundObject and puts it in the buffer. This Instance will point to the original SoundObject. After copying to the buffer, you can paste as normal on the timeline.
Remove	This will remove the selected SoundObject from the library.

5. You can also then edit the SoundObject in the library from within the SoundObject Library dialog by selecting the SoundObject in the list. The editor for the SoundObject will appear below.

   ### Note

   Editing the SoundObject in the library will affect all instances of that SoundObject.

The general pattern of usage for the program-wide SoundObject Library entails:

1. Add SoundObject to the Library. This is done by selecting a SoundObject on the timeline, right-clicking the SoundObject to open up the popup menu and selecting "Copy".

2. After copying an object, open up the SoundObject library window and go to the User SoundObject Library. Right-click a folder node in the library tree and choose "Paste" to add a copy of the object to the library.

3. You can use the context popup menu to create new folders, or double-click a node to rename the node.

Parameter Automation

Introduction

Blue allows for a few ways to enter in values to automate parameters over time. The parameters that can be automated are those in BlueSynthBuilder Instruments, Mixer Effects, Mixer Sends, and Mixer Volumes.

The following sections will describe how to use Parameter Automation and technical details in how this all works within the context of Blue and Csound.

Available Parameters

Besides the Volume sliders in the mixer and level amount on mixer sends, the following GUI Parameters are automatable when found in BlueSynthBuilder instruments and Mixer Effects:

- Knob
- HSlider
- HSliderBank
- VSlider
- VSliderBank
- XYController
- Dropdown
- Checkbox

Assigning Parameters

Automations for Parameters are assignable to SoundObject LayerGroup Layers that exist in the root Score by selecting from the popup menu that appears when selecting the "A" button on a SoundLayer's panel (on the left of the timeline). The menu shows what Parameters are available to use from Instruments and the Mixer. Those with normal color text are Parameters which are not currently automated, those in green are ones currently automated in that SoundLayer, and those in orange are ones which are currently automated on a different SoundLayer. Selecting normal color text will enable automation for that parameter on the SoundLayer, while selecting a green colored text one will disable automation for that parameter, and selecting an orange one will move it from the layer it is currently on to the one being worked with.

Score Timeline Modes

The Score timeline has three different modes:

Score — This is the primary mode for adding and editing SoundObjects. Parameter Automations will be drawn but are not editable in this mode.

Single Line — This is the primary mode for editing line points for Parameter Automations. The user is able to edit a single line at a time per SoundLayer, and can enter and remove points, modify points by dragging them, as well as selecting a region and move the the points by dragging the region. While dragging the region, the region will show that data will overwrite the area dragged to. This overwriting will not be committed however until the user releases the mouse button.

Multi Line — This is the mode for selecting and moving line points for all Parameter Automations on a single or multiple SoundLayers. The use can click and drag to create a selection region, then click within the selection region to move all points from all lines within

the region. While dragging the region, the region will show that data will overwrite the area dragged to. This overwriting will not be committed however until the user releases the mouse button.

Editing Automations

To edit line values in a Parameter Automation, first switch to Single Line mode. Then, for the desired SoundLayer, select which of the assigned parameters to edit. This can be done either by using the SoundLayer Edit Panel's Parameter selector (found on the second row below the row with mute, solo, etc.; only shown when SoundLayer size is greater than 1), or by right-clicking the SoundLayer in the main area and selecting from the popup menu (the actively editable Parameter Automation will be disabled from the popup menu).

Once a parameter is selected, the user can change the color of the line by using the color box next to the parameter selector panel on the SoundLayer Edit Panel. To add new points, use the mouse and mouse over to where you would like to add a point and press the left mouse button. After pressing, the point will be added, and the user can drag to modify the just entered point. Releasing the button will complete the addition of the line point. To edit an existing point, mouse over a point until it is highlighted in red, then press the left mouse button, drag, and release when finished. To remove an existing point, mouse over a point until it is highlighted and press the right mouse button.

For finer control over the line points, right-click on the parameter line panel when not over an existing line point. A popup menu will appear and select "Edit Line Points". A dialog showing a table of line points with time values in one column and parameter values in a second column. The user can then enter in values by text entry.

Technical Details

Blue's parameter automation system is implemented in Csound code in a few different areas. For instruments and effects, when an Automatable Parameter is not set for automation, it will replace its replacement key in its ORC code with a constant value from the UI item. If the parameter is set to allow automation, then the replacement key is replaced with a Blue generated global k-rate signal name. Therefore, when coding instruments and effects, the user should be careful to make sure that if they want the parameter to be automatable that the replacement key is placed in the code where a k-rate signal is legal to use. If the user is not expecting to automate that value, then it would be safe to place that replacement key wherever a constant is allowed. This is also useful to know when migrating older BSB Instruments and Effects to be used with the purpose of Automation.

After setting up the Instrument and Effects, Blue will then handle compiling the automations into a way that works with Csound. Currently, Blue will first calculate init statements from where the render start time is and add that to the CSD to make sure that the global krate signals will be initiated by the time any instrument could use them. Then, Blue creates instruments for each parameter that will be automated. The instruments are driven by score and either an instrument for continuous data or resolution dependent data will be created. Score is then generated for the parameter automation data that together with the instruments generated will create the signals.

Care was taken to optimize the generated score and instruments. Unnecessary score will not be generated if multiple points with the same value are found as well as when there is a discontinous break in the values. Values from the parameter automation will also correctly be generated from render start time until render end time (if used). For parameters with resolution, every value is a discrete jump, so the instrument for resolution based parameters will on it's i-time run will simply set the new value and then turn the instrument instance off to save CPU cycles.

Command Blocks

Introduction

Command Blocks are Csound ORC code with special directions to Blue on how to processes them. The are available to use only within the Global Orchestra areas of Instruments (not to be confused with the global orchestra area in the globals manager tab) and were created to help users build fully-encapsulated instruments, such that an instrument and all of its necessary parts could be grouped together.

Note

This feature has largely been made unnecessary due to new Csound programming practices that have developed since this was initially introduced. This feature is left in Blue though to support legacy projects.

Basic Usage

To use a command block, you will need to wrap the section of orc code that you want Blue to process with the command in the following manner:

```
;[command to use]{
...Csound Orchestra Code...
;}
```

Available Commands

once

Once blocks are useful to limit how many times a block of code will be generated in the final CSD and is more for efficiency than anything else (so Csound doesn't define the same ftable more than it has to, for example). An example of this:

```
;[once]{
gaLeft init 0 gaRight init 0
;}
```

In this case, if the above gaLeft and gaRight init statement, wrapped in a once block, is used in multiple instruments, then regardless it will be generated only once in the final CSD. So a use case for the above would be if you have all of your instruments output to the above global variables for extra processing by an always-on reverb, you would want to add the above to all of the instruments and also the reverb. Then, if you start a new project and may want to reuse a few of your instruments as well as the reverb, you can just copy them into your new project's orchestra and start working away, knowing your global variables will be initialized and ready to go.

pre

Pre blocks are useful if you need the enclosed ORC code to be generated before any other global orc code. When Blue goes to process the project to generate a CSD file, it normally starts of the existing global orchestra code from the globals manager, then appends all global orc from instruments and soundObjects

to it, one after the other. By using a pre block, the code enclosed will be pushed up to the top of the global orchestra of the CSD (instrument 0).

The scenario which required a solution such as this was when working with the fluidSynth opcodes. For those set of opcodes, a fluidEngine needs to be initialized before loading of soundfonts into that engine. The design I had wanted was to have the instrument that did the fluidOutput also have the engine initialization. This was a problem because the fluidOutput needed to be placed at an instrument number higher than the fluidNote playing instruments for the output chain to work correctly. So here was a case of an instrument that needed to have its instrument generate later than the instruments that it depended on, but also have its own fluidEngine initialization happen before all the other soundFonts load in in their own global orc statements. The solution then was to create the once block feature and allow for targetting global orc code to be prepended to the top of the generated global orc code.

Chapter 3. Reference

Sound Objects

Sound Objects are objects on the score timeline that are primarily responsible for generating score data.

Common Properties

The following are properties that all SoundObjects share.

Common Properties

Name	Name of the Soundobject
Subjective Duration	The duration of the soundObject on the timeline (versus the duration of the generated score within the soundObject, which may be different). How the duration relates to the generated score contents is controlled by the "Time Behavior" property.
End Time	Read-Only property that shows the end-time of the soundObject
Time Behavior	Selects how subjective time should be used on a soundObject. Options are: 1. Scale - The default option, stretches generated score to last the duration of the soundObject 2. Repeat - repeats generated score up to the duration of the soundObject 3. None - Passes the score data as-is (When using Time-Behavior of None, width of soundObject no longer visually corresponds to duration of the soundObject's score.)

About Note Generation

Sound Objects generate notes in the following manner:

- SoundObject generates initial notes
- NoteProcessors are applied to the generated notes
- Time Behavior is applied to the notes

Partial Object Rendering

When using a render start time other than 0.0, how soundObjects contribute notes depends on if they support partial object rendering. Normally, notes from all soundObjects which can possibly contribute notes to the render (taking into account render start and render end) are gathered and then if any notes start before the render start time they are discarded as there is no way to start in the middle of a note and to know exactly that it sounds as it should as blue's timeline only knows about notes and not how those instruments render.

However, there are certain cases where blue soundObjects *can* know about the instrument that the notes are generated for and can therefore do partial object rendering to start in the middle of a note. This the is the

case for those soundObjects which generate their own instruments, such as the AudioFile SoundObject, FrozenObject, the LineObject, and the ZakLineObject. For those soundObjects, if render is started within the middle of one of those, you will hear audio and have control signals generated from the correct place and time.

AudioFile

Accepts NoteProcessors: no

Allows for quickly putting in a soundfile on the timeline.

CeciliaModule

Accepts NoteProcessors: no, Supports TimeBehavior: no

Note

This SoundObject is unfinished and not enabled for normal use. This information is left here for the future if the module is ever finished and reenabled.

Description

The CeciliaModule is a soundObject that is meant to be compatible with modules for the program Cecilia (http://www.sourceforge.net/projects/cecilia), created by Alexandre Burton and Jean Piche. The CeciliaModule is not as fully featured as Cecilia itself as some features found in Cecilia do not make sense within the context of Blue.

One major advantage of the CeciliaModule is the ability to have as manyinstances of a module on the timeline as one likes, all without one worrying about variable name clashes or other issues. Blue automatically parses and renumbers instruments, ftables, and score statements so that they do not interfere with other instances of the same module.

Differences between Cecilia and CeciliaModule

Modifications required for Cecilia Modules to run in CeciliaModule

Cecilia modules require at least one modification before they are able to run in Blue, and also must adhere to a few more constraints of design.

Appending “ftable” in Instrument Definitions . Instruments are required to append the word “ftable” before any number that is meant to be an ftable. For example:

```
instr 1

aout oscil 10000, 440, 1

out aout, aout endin
```

would have to become:

```
instr 1
```

```
aout oscil 10000, 440, ftable1

out aout, aout endin
```

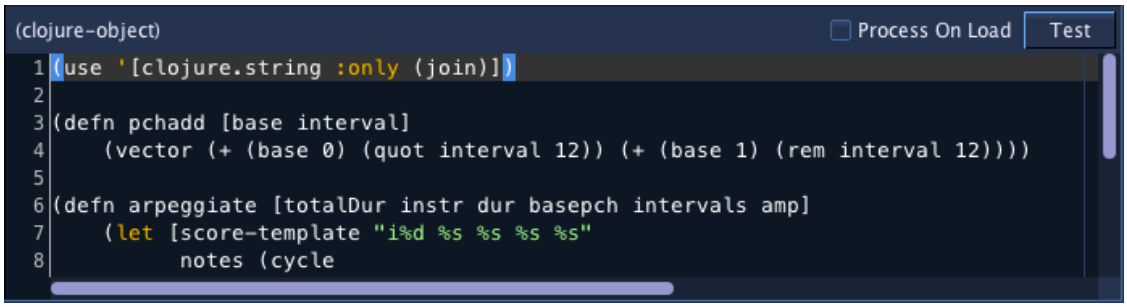
The purpose of this is so that Blue will know what ftable references will need to be updated when Blue reassigns ftable numbers.

Comment

Accepts NoteProcessors: no

Used to add comment text on the timeline and does not generate any notes.

ClojureObject



Accepts NoteProcessors: yes

Allows using the Clojure [http://www.clojure.org] programming language to generate score data. The Clojure interpreter is included with Blue, so ClojureObjects are portable between systems without any external dependencies. Users do not have to install anything further to use this object, and the code will continue to function for the duration of Blue's existence.

When writing your script to generate notes, assign the string value of the notes to the symbol 'score'. Blue will then read in the value from that variable and continue processing.

Example 1

```
(def score "i1 0 2 3 4 5")
```

The above example shows the simplest script and will generate a single note. If the ClojureObject is set with a start time of 0 and a duration of 2, then it will generate the following score:

```
i1 0.0 2 3 4 5
```

Example 2

```
(use '[clojure.string :only (join)])

(defn pchadd [base interval]
```

```
    (vector (+ (base 0) (quot interval 12)) (+ (base 1) (rem interval 12))))

(defn arpeggiate [totalDur instr dur basepch intervals amp]
    (let [score-template "i%d %s %s %s %s"
          notes (cycle
                    (map #(apply format "%d.%02d" (pchadd basepch %1))
                        (concat intervals (subvec intervals 1 (- (count intervals)
    (join \newline
        (map #(format score-template instr %1 dur %2 amp)
            (range 0 totalDur dur) ; list that will limit the map
            notes))))

(def score (arpeggiate blueDuration 1 0.25 [8 0] [0 4 7] -12))
```

The above example is taken from blue/examples/soundObjects/clojureSoundObject.blue. This script defines an apreggiate function, then calls that assigns the value to score. Notice the use of blueDuration, a symbol that is automatically assigned to the duration of the ClojureObject.

If the ClojureObject is set with a start time of 0 and a duration of 4, then it will generate the following score:

```
i1 0.0 0.25 8.00 -12
i1 0.25 0.25 8.04 -12
i1 0.5 0.25 8.07 -12
i1 0.75 0.25 8.04 -12
i1 1.0 0.25 8.00 -12
i1 1.25 0.25 8.04 -12
i1 1.5 0.25 8.07 -12
i1 1.75 0.25 8.04 -12
i1 2.0 0.25 8.00 -12
i1 2.25 0.25 8.04 -12
i1 2.5 0.25 8.07 -12
i1 2.75 0.25 8.04 -12
i1 3.0 0.25 8.00 -12
i1 3.25 0.25 8.04 -12
i1 3.5 0.25 8.07 -12
i1 3.75 0.25 8.04 -12
```

Regarding Processing

Blue processes soundObjects by going through each SoundLayer and generating score for each object within each layer. This is useful to know so that if you are using a ClojureObject that has utility functions that you later use in other ClojureObjects, you should put that utility ClojureObject on the first SoundLayer closest to the top, or at least on a layer above all others that contain ClojureObjects.

Also to note, as a feature, Blue uses a single interpreter instance for processing Clojure code. Therefore, if one ClojureObject has code evaluated, the values from that code can be read by other objects. This allows creating utility ClojureObjects. However, one can use stale values (or values from another project even) if one is not careful to always assign values in the project that require being set for this particular project.

Variables from Blue

The following variables are available from Blue:

Variables from Blue

blueDuration	Duration of the Clojure SoundObject
blueProjectDir	The location of the current project's directory. Includes path separator at end.

Process at Start

There is a checkbox entitled "Process at Start". Selecting this option will have the script of the ClojureObject run when a .blue project is loaded. This is useful for scripts that act as library functions, but themselves do not generate any notes. For example, you might define a number of score generation utility functions in one ClojureObject that has "Process at Start" enabled. Your other ClojureObject may then use the functions from that ClojureObject Next time you load your project, if that ClojureObject hasn't been run, your other ClojureObject will not be able to be run either. If you are rendering from the beginning of a project, this won't be an issue, but if you're starting work in the middle of a project, you will need to evaluate that utility ClojureObject at least once. You can either do a run from the start at least once, use the "Test" button to have that evaluated, or use "Process at Start" and have Blue ensure it is loaded into the Clojure interpreter when you load your projects.

Using External CLJ Scripts

Blue is able to load external .clj scripts, resolved from the .blue/script/clojure or PROJECT_DIR/script/clojure directory. For example, if you use:

```
(use 'my.script)
```

This will try to load the script from "/Users/me/.blue/script/clojure/my/script.clj" or "/path/to/blueProject/script/clojure/script.clj".

External SoundObject

Accepts NoteProcessors: no

Allows you to write script within Blue and execute an external program on that script, bringing in generated notes back into Blue. There is a field for 'command line' as well as a text area for your script. When this soundObject is called for generating it's notes, it will write the text within the text area into a temp file and then use the user-supplied 'command line' to execute on that temp file.

When Blue runs the commandline, it defaults to appending the temp file's name to the end of the commandline. For example, if you wrote a perl script in the text area and used a commandline of "perl", then when Blue runs the command, it would be something like "perl /the/path/to/the/tempFile". If you need to explicitly put the name of the temp file somewhere else in the command line than the end, you can use "$infile" in your commandline. For example, if you needed something like "myCommand somefileName -w -d --some-other-flags" and had to have it in that order, you could type "myCommand $infile -w -d --some-other-flags" and Blue will replace that $infile with the temp file's name instead of appending it to the end.

When designing your scripts that you will be using with external programs, you should either set commandline flags to the program or script your script in a way that it will write out to stdout(to the screen), as that is what Blue will capture and bring in as notes.

A second method for the external object was created for bringing in score data after running the commandline as some programs (i.e. Cmask) do not have any possible way to write directly to stdout. The

process is to use "$outfile" in the commandline to pass in a filename to the program. That $outfile will be the name of a temp file that the external program will write score data into. After the program is finished, Blue will open up that temp file, read in the score data, and then remove the temp file. So, if your program needed a commandline of something like "myProgram -i inputfilename -o outputfilename" and no options for writing to screen, then you could use a commandline of "myProgram -i $infile -o $outfile".

There is a test button that you may use to check to see that Blue properly brings in the generated notes.

Command Lines and Notes Other Programs

This is a list of commandlines to use when using other programs with Blue and any notes that may concern other programs when being used with Blue. These commandlines assume that the programs being called are on the system path. If not on the path, please append the full path to the program before the program name.

Command Lines for Programs

CMask — Homepage: http://www.bartetzki.de/en/software.html for the Linux, Windows and old Mac builds, http://www.anthonykozar.net/ports/cmask/ for OSX Author: Andre Bartetzki

Comandline to use:

```
cmask $infile $outfile
```

nGen — Homepage: http://mikelkuehn.com/index.php/ng Author: Mikel Kuehn

Comandline to use:

```
ngen $infile $outfile
```

AthenaCL — Homepage: http://www.flexatone.org/athena.html Author: Christopher Ariza

Comandline to use:

```
python athenaPipe.py $infile
```

Examples of the External SoundObject, as well as athenaPipe.py, can be found in the blue/examples/soundObjects folder.

GenericScore

Accepts NoteProcessors: yes

Contains a block of Csound score text. The objective time within the GenericScore starts at time 0, and notes within the genericScore are written relative to 0. The start time of the GenericScore object within the timeline will translate the score text's time, and the SoundObject's time behavior will determine how the notes are processed relative to the GenericScore's duration. For example, for the following score:

```
i1 0 1 2 3 4 5
i1 1 1 2 3 4 5
i1 2 1 2 3 4 5
```

If this SoundObject is moved to start at time 2.0 with duration 6.0, the generated score will be:

```
i1 2 2 2 3 4 5
```

```
i1 4 2 2 3 4 5
i1 6 2 2 3 4 5
```

Note

Score blocks support only a subset of the Csound Score Syntax. Using "+" in p2 and "." in pfields, as well as "<" and ">" is supported.

Instance

Accepts NoteProcessors: yes

A SoundObject that points to a SoundObject in the SoundObject library. The content of the SoundObject is not editable except by editing the SoundObject in the library to which the instance is pointing to. If editing the SoundObject in the library, all instances that point to that SoundObject will be changed.

When clicking on an Instance object to edit it, the SoundObject from the library will show its editor instead in the SoundObject edit area, but it will be clearly marked that the user is currently editing a library SoundObject and not an individual SoundObject.

The instance object is very useful if your work entails using many variations of single source of material. For example, you can take a SoundObject that represents a drum pattern, then make many Instances of that pattern and add them to your project. If you want to later change the base pattern, it will update everywhere that there are Instance objects pointing to it.

Another example, if you have a SoundObject that represents a motive or melodic fragment, you can put it into the SoundObject library, then make 10 instances of it, adding noteProcessors to do things like transpose or get the retrograde, etc. Later, if you decide you want to change the base material, you'd only have to edit it once within the library. All Instance objects pointing to the changed object will be updated and will maintain all of their transformations and relationships that were added via NoteProcessors.

If you later decide that you want to work with the original material where you have an Instance and want to break the link, replacing the instance with a copy of the original to modify, you can convert the Instance object to a GenericScore. To do this, you can use one of two methods:

- Right-click the Instance object and choose "Convert to Generic Score"
- Go the SoundObject Library, select the original and use "Copy", then select the Instance object and use "Replace with SoundObject in Buffer"

JMask

Accepts NoteProcessors: yes

JMask is GUI score generating soundObject based on Andre Bartetzki's CMask [http://www.bartetzki.de/en/software.html]. JMask currently supports all features of CMask except field precision, though support for this feature is planned. Users of JMask are encouraged to read the CMask manual available at Andre Bartetzki's site to get familiar with CMask concepts as they are represented 1:1 in JMask.

JMask builds on top of CMask's concepts and feature set by providing the user a graphical way to edit parameter values, tables, masks, etc. Also, by using JMask, the user's projects are guaranteed to run on all Blue installations, while using CMask from Blue requires anyone opening and using the project to have CMask installed on their system.

The following documentation for JMask will assume the user is familiar with CMask and that the user has read through the CMask manual available from the link above. The following will discuss how to use JMask, covering how the CMask features are implemented in JMask and how to work with the user interface. Over time, this manual entry for JMask will attempt to grow to be self sufficient and not require knowledge of CMask, but for now familiarity with CMask is suggested.

Overview

A single JMask object is equivalent to one CMask field. A single JMask object holds a collection of Parameters, one for each pfield to be generated in the score. Each of the Parameter types in CMask is available in JMask:

Table 3.1. Parameter Types

Type	Description
Constant	Generate the given value when generating values for notes
Item List	Generates values from a list of values given by the user. Options for generation include cycling through the list (cycle), going backwards and forwards through the list (swing), going through the list in random order using all values once before repeating (heap), and choosing a random list item (random).
Segment	Generates values along a user-defined line.
Random	Generates a random value.
Probability	Generates a randomized value that is weighted by the given probability type and the configuration of the parameters of that probability type.
Oscillator	Generates values that oscillate in a given period using different user-chosen shapes.

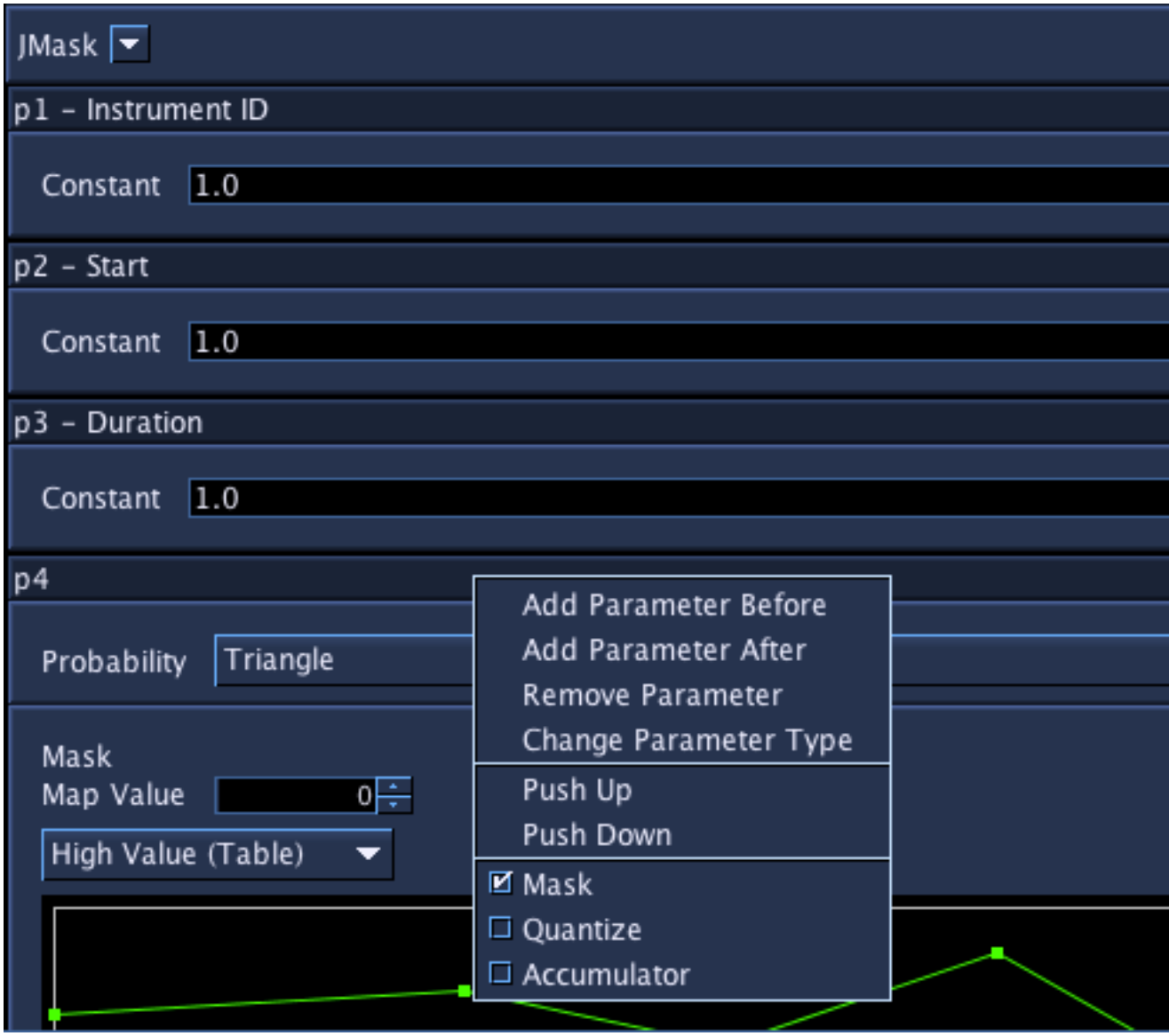
Beyond the Parameters, JMask supports the three modifier types found in CMask: Masks, Quantizers, and Accumulators. Certain parameters support different modifiers, and their support by paramater type is listed below:

Table 3.2. Parameter Modifier Support

Type	Supports Mask	Supports Quantizer	Supports Accumulator
Constant	No	No	Yes
List	No	No	Yes
Segment	No	Yes	Yes
Random	No	Yes	Yes
Probability	Yes	Yes	Yes
Oscillator	Yes	Yes	Yes

JMask also supports the use of a seed value. When enabling the use of seeding, the seed value will be used to initialize the pseudo-random number generator used in randomized operations. This allows the user to set a seed and get consistently reproducible results. The default is to use the system time to seed the random number generator, thus giving different results each render.

Using the JMask Interface



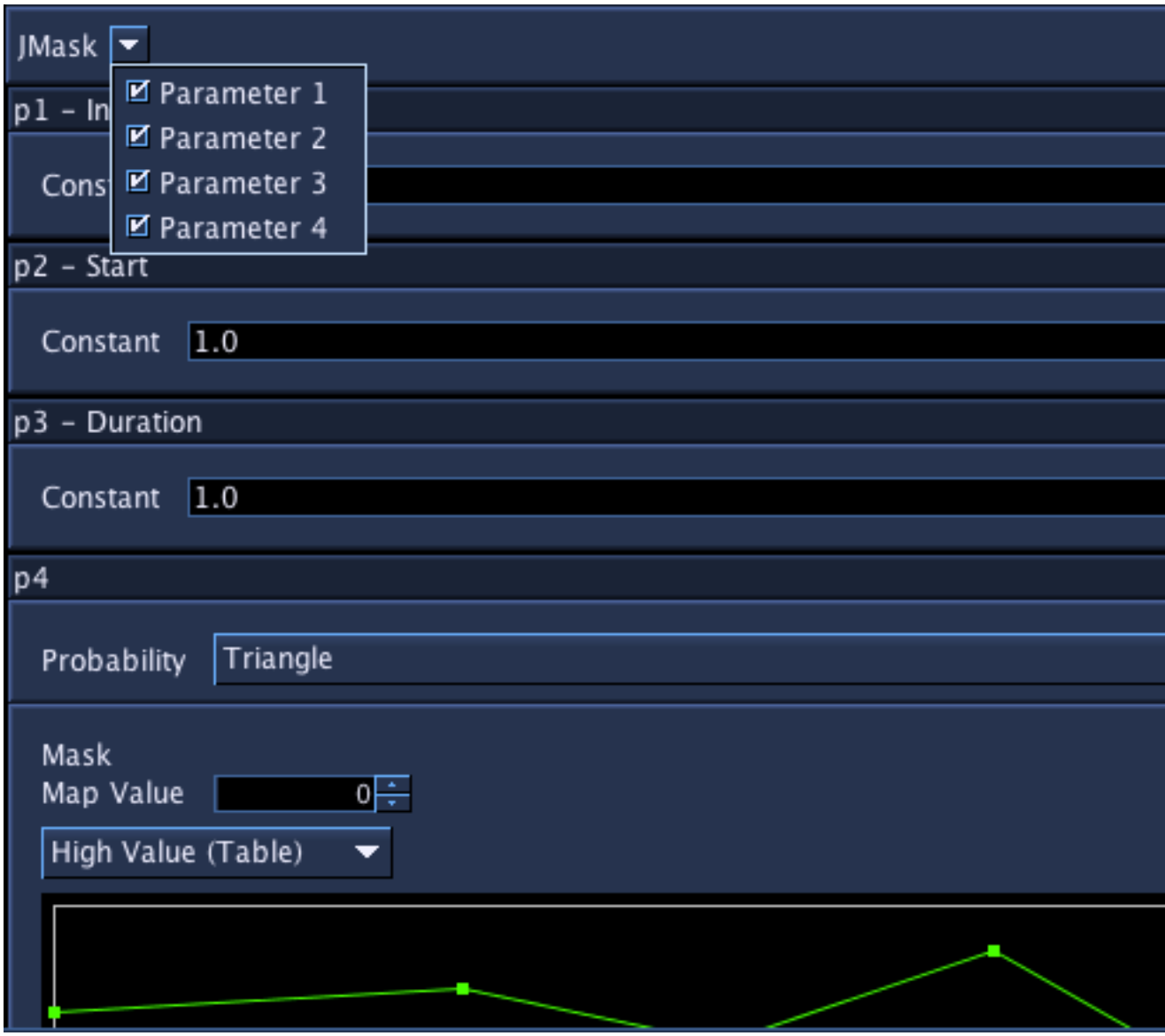
The JMask SoundObject Editor allows for viewing the editors for all of the assigned parameters. Parameters are each set to generate values for one pfield. On the top side of each row is the Parameter Edit Panel. This panel shows a number at the top that corresponds to what pfield this parameter will generate values for, as well as the field name. To change things about the Parameter, right-click the panel to show a popup menu as shown in the image above. The options are described below:

Parameter Edit Options

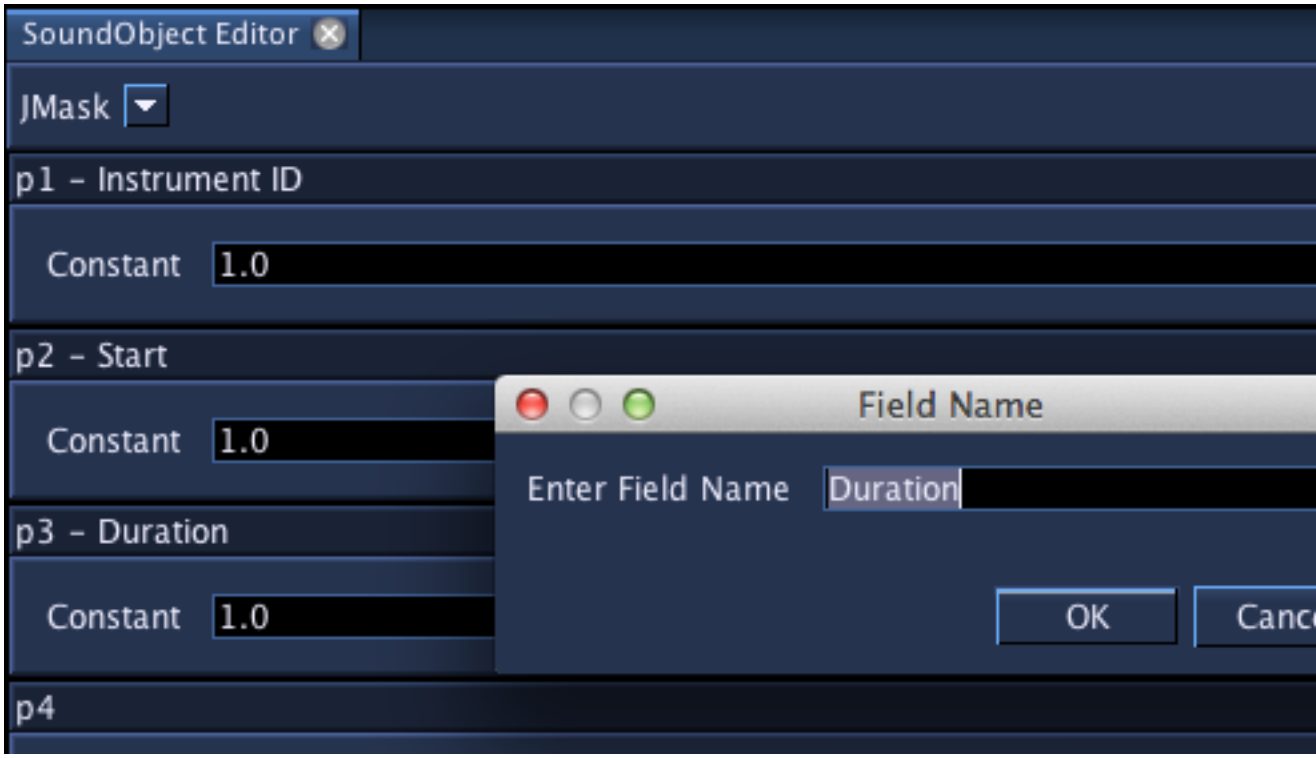
Add Parameter Before	Create a new Parameter and insert it before the Parameter clicked on by mouse. When this option is selected, a dialog will appear with a dropdown of options of what type of Parameter to add.
Add Parameter After	Create a new Parameter and insert it after the Parameter clicked on by mouse. When this option is selected, a dialog will appear with a dropdown of options of what type of Parameter to add.

Remove Parameter	Remove this Parameter. Will not be allowed if trying to edit parameters 1-3 as JMask requires a minimum of 3 pfields.
Change Parameter Type	Choose a different type of Parameter and replace the current one with the selected one. Any edits from the old Parameter will be lost once a new parameter type is chosen.
Push Up	Move the selected Parameter to before the previous Parameter. Example: Push up parameter at pfield 5 to pfield 4, moving what was previously at 4 to 5.
Push Down	Move the selected Parameter to after the next Parameter. Example: Push down parameter at pfield 4 to pfield 5, moving what was previously at 5 to 4.
Mask	Enable/disable using a Mask with this parameter. If enabled, the Mask editor will appear, and if disabled, the Mask editor will disappear. This menu option will not show for those Parameters that do not support Masks.
Quantizer	Enable/disable using a Quantizer with this parameter. If enabled, the Quantizer editor will appear, and if disabled, the Quantizer editor will disappear. This menu option will not show for those Parameters that do not support Quantizers.
Accumulator	Enable/disable using an Accumulator with this parameter. If enabled, the Accumulator editor will appear, and if disabled, the Accumulator editor will disappear. This menu option will not show for those Parameters which do not support Accumulators.

Beyond the basics of moving around Parameters, adding new ones, and choosing whether to enable Masks, Quantizeres, and Accumulators, one can also choose to show/hide Parameters by using the popup menu that appears when choosing the down arrow button in the JMask title bar, as shown in the screenshot below.



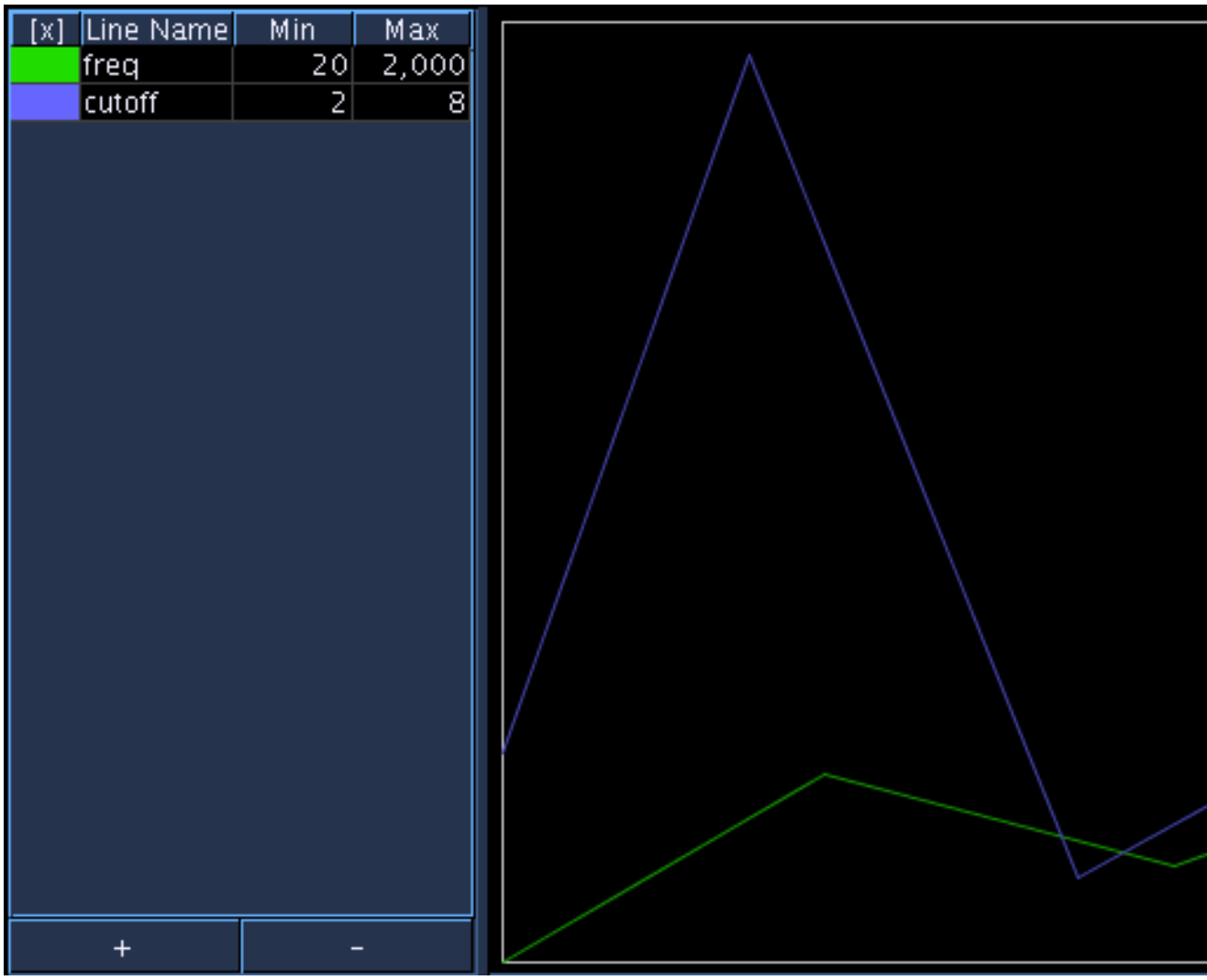
To edit the field name, double click the Parameter Edit Panel. A dialog will appear where you can modify the field name, as shown below:



Beyond these basics for working with the Parameters in general, each Parameter type has its own editor, each customized for the values and options allowable for each Parameter. Currently, documentation is omitted for each Parameter's GUI as they correspond in feature and parameters as CMask, and the user is encouraged to consult the CMask manual for more information on editing each Parameter.

LineObject

Accepts NoteProcessors: no



Line Object

Add and graphically edit global k-rate signals.

Use the bottom + and - buttons to add/remove lines.

Use the table to select a line. After selecting a line, you can edit the points by dragging them, add points by clicking where you want the new point, and remove points by rt-clicking them.

Use the table to edit the name of the signal, as well as the max and min values (need to be floating point values). For editing the color of the line, double-click the color box to the left of the lineName. A color selection dialog will appear for the user to choose their desired color.

The name of the signal will be prepended with "gk" when outputing a signal, i.e. a line name of "cutoff" will become "gkcutoff".

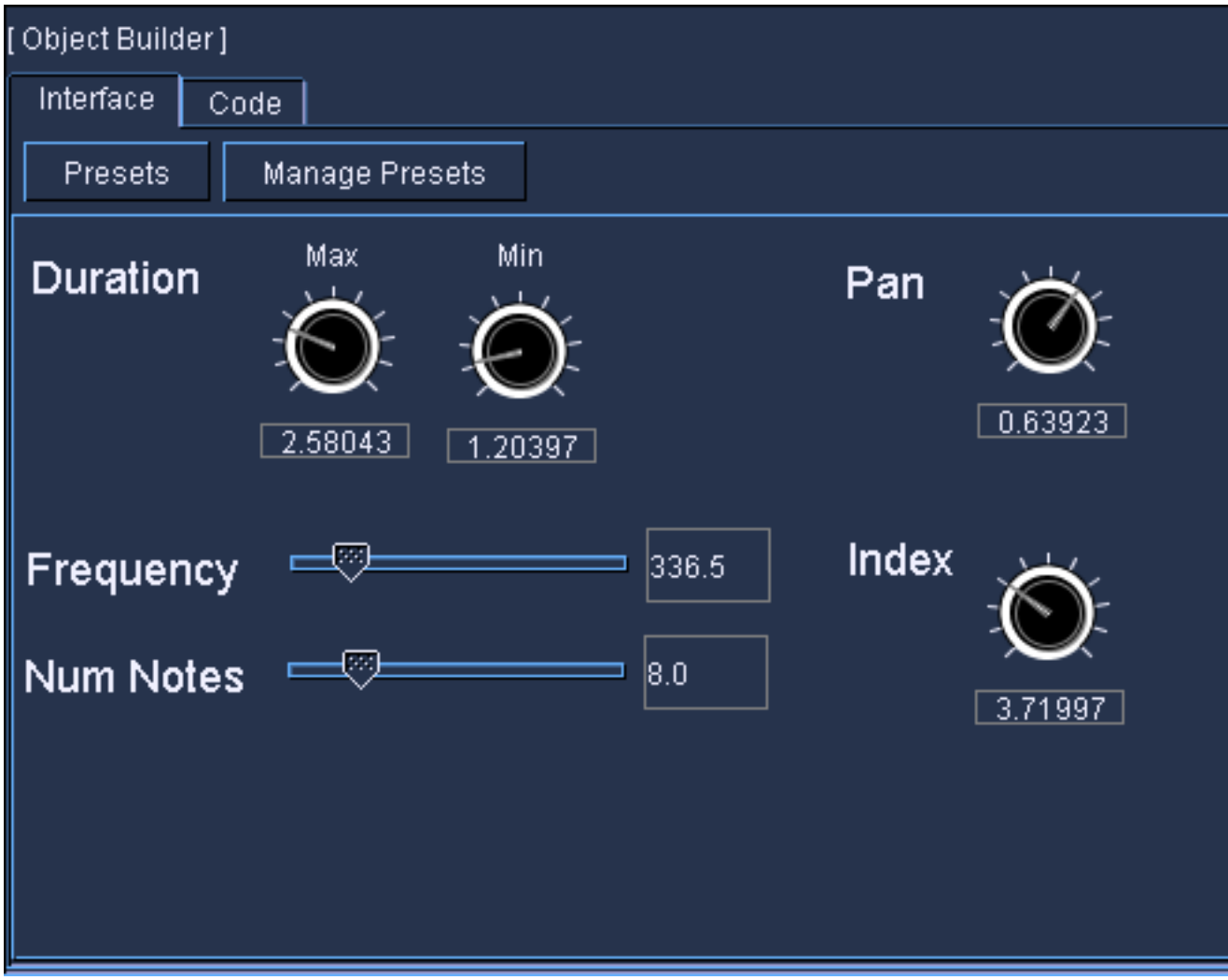
ObjectBuilder

Accepts NoteProcessors: Yes

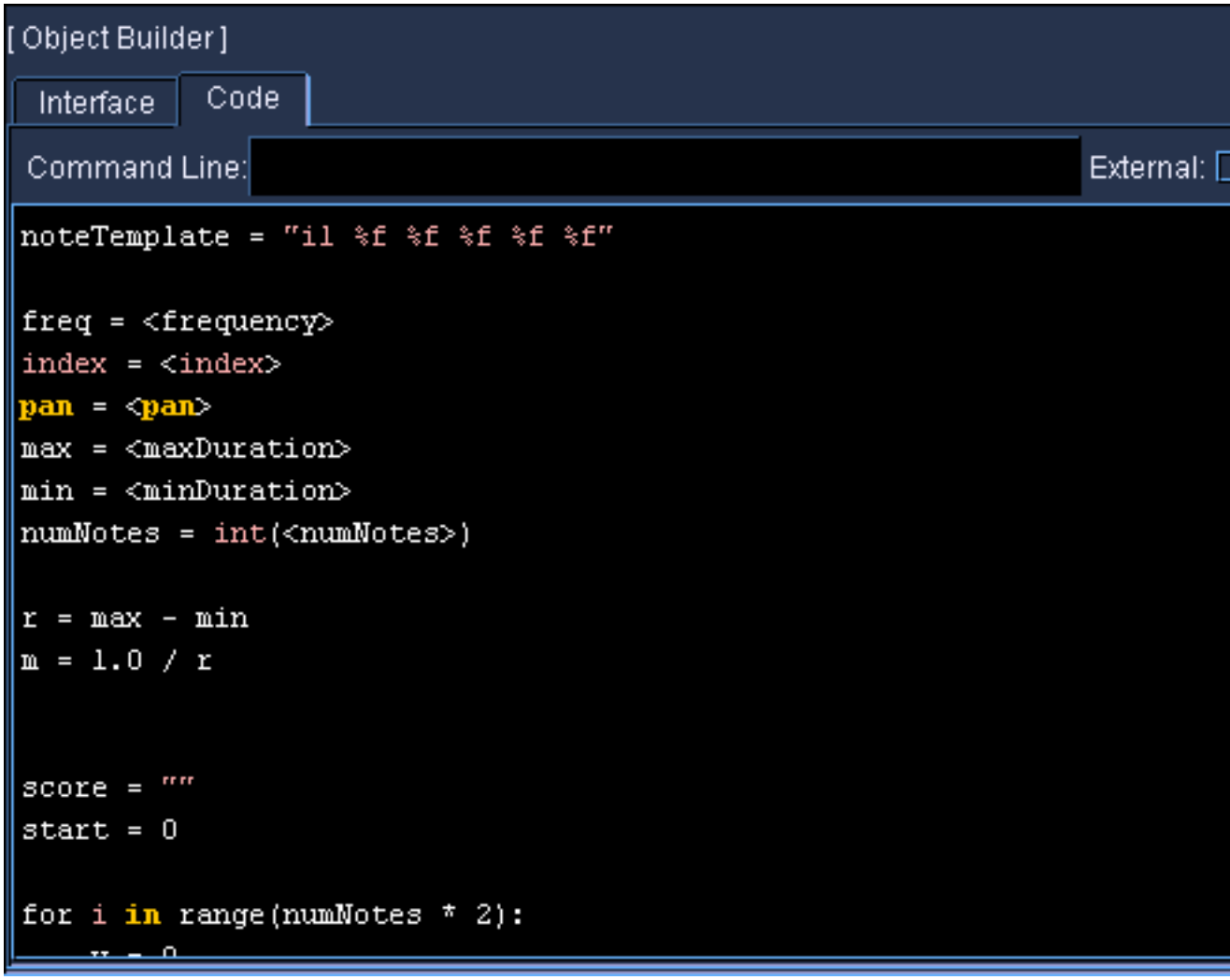
This SoundObject allows users to build their own SoundObjects by using the same widgets and features as the BlueSynthBuilder for creating user interfaces, as well as using either Python script or External Script for the score generation. When generating score, the script has token values replaced by values from the user interface before generating. The scripts generated in the same manner as the PythonObject and as the ExternalObject .

Note: The code completion feature using ctrl-shift-space for values from the UI that is available in BlueSynthBuilder is also available in this SoundObject.

Interface Editor



Code Editor



PatternObject

Accepts NoteProcessors: yes


Pattern Name
[x]
Snare
Hihat
Hihat2
Bass Drum
1
2
3
4
^
V
+
-
Pattern Score
i4 0 .25 80

About the PatternObject

The PatternObject is pattern-based score editor, based on the author's previous project "Patterns". It is a flexible pattern-oriented score editor, useful for musical ideas which are pattern based, such as drum parts or minimalist-style musical ideas.

Usage

For the general workflow of using the PatternObject, users will liked likely want to:

1. Setup the PatternObject Properties (number of beats and subdivisions)
2. Add Patterns, giving each pattern a name to help identify what the pattern is for
3. Edit each Pattern's score
4. Visually edit the PatternObject score

Setting PatternObject Properties

The PatternObject's Time Properties can be modified by clicking on the button in the upper-right corner of the editor. Clicking the button will hide or show the properties panel on the right, as shown below:

Note

Editing the time values will clear out any pattern triggers that the user has entered. It is recommended that one first decide the pattern time properties before working with the PatternObject.

Adding Patterns

To add Patterns to the PatternObject, use the "+" button on the bottom of the left hand section. Double-clicking the name of the pattern name will allow editing of the name, and clicking on the [x] box will allow for muting the Pattern. Clicking on the Pattern in the table will also bring up it's score in the area below.

Editing the Pattern's Score

The Pattern's score is standard Csound SCO text, with the same features supported as by the GenericScore SoundObject. Each score should be started from time zero. The Pattern's score should be set such that its score's total duration fits within the time of the subDivision. For example, if the properties set for the PatternObject's time values are 4 beats and 4 subdivisions, each beat is 1.0 in duration (corresponds to p3 value of a note) and thus a single subdivision in this case would be equivalent to .25. Scores shorter or longer than the subdivision length are allowed, but one should be aware that the resultant score may or may not longer than what is visually represented on the PatternObject score.

Editing the PatternObject Score

The PatternObject score is visually edited by click on squares which correspond to subdivisions of the beat. For example, if a Pattern's score is .25 in total duration, and the time value of the PatternObject is set to 4 and 4, then clicking a square would be to insert a 16th note in a score.

To add to the PatternObject score, simply click on the square where one wishes to have a Pattern triggered. To remove, simply click on a selected square. The user can also click and drag to add or remove mutliple triggers.

Other Notes

- Users may be surprised at the generated score if time behavior is not set to None(while others may prefer to use Scale, which is the default). Please be mindful of this, especially when using PatternObjects and the SoundObject library, where the default for creating instances of SoundObjects is to have Time Behavior of Scale on the Instance SoundObject.

- When the PatternObject is used with a Scale time behavior, you may not get the results you think you will get if the pattern is not filled in in the last subdivision of the beat. When Blue goes to generate a score for a soundObject, with a scale time behavior, the score is first generated, then scaled to the duration of the soundObject. So, if you're pattern has empty subdivisions and you're expecting there to be a space at the end of your pattern, when you go to generate the CSD, the space won't be there.

  To get around this, you may want to use a time behavior of None, or you may want put in a "ghost note" in a Pattern layer, using a note like:

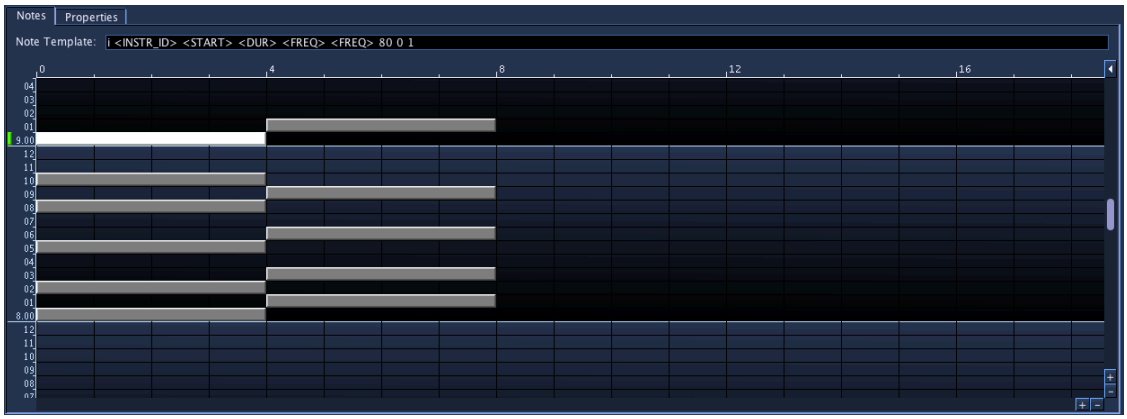
  ```
  i 10000 0 .25
  ```

  Where 10000 is not an instrument in use in your project. Then you can put in a trigger for this pattern in the last subdivision. What will happen is that Blue will process the score with this note's time and thus will generate with this out to the CSD, but Csound will ignore the note.

  When the PatternObject is set to use a time behavior of Repeat, the same situation can occur as when using Scale if a Repeat Point is not set, as the default for Repeat when no repeat point is set is to take the generated score and repeat it after the duration of the last note. To prevent this, either use the ghost note technique above or set a repeat point.

PianoRoll

Accepts NoteProcessors: yes



About the PianoRoll

The PianoRoll SoundObject is a graphical tool to enter in notes, commonly available in many MIDI sequencer environments. This PianoRoll is unique in that it is Microtonal: it supports loading any Scala [http://www.huygens-fokker.org/scala/] scale file and editing of notes adapts to that scale. For example, in the picture above, the scale loaded is a Bohlen-Pierce scale with 13 tones to the tritave. The PianoRoll above has adapted to that scale to show 13 scale degrees per octave of its editor. The generated notes can output values as either frequency or PCH notation (octave.scaleDegree). But don't worry, if you're not interested in alternate tunings, the PianoRoll is set by default to use 12-TET tuning, the "standard" tuning system in use today.

About Note Template Strings

The PianoRoll uses Note Template strings as a way to maintain flexibility and be able to handle the open-ended nature of Csound's instruments. Since the user who builds the instrument designs what each pfield will mean(besides p1, p2, and p3), the Note Template string should be made to match the instrument the user wants to use the PianoRoll with. When the note is generated, certain special text values (those enclosed in < and >) will be replaced by values unique to the note.

For example, the following Note Template string:

```
i<INSTR_ID> <START> <DUR> <FREQ> 0 1 1
```

Will have the <INSTR_ID> replaced with the value set in the Piano Roll properties, <START> replaced the start time for the note, <DUR> replaced with the duration of the note, and <FREQ> replaced with either a frequency or PCH value, depending on how the Piano Roll is configured in its properties. The other values will pass through as part of the note.

Warning

Caution should be used when creating a Note Template string to make sure that there are enough spaces allowed between replacement strings. For example, the following:

```
i<INSTR_ID> <START> <DUR> <FREQ> <FREQ> 0 1 1
```

would result in:

```
i1 0 2 440 440 0 1 1
```

while the following:

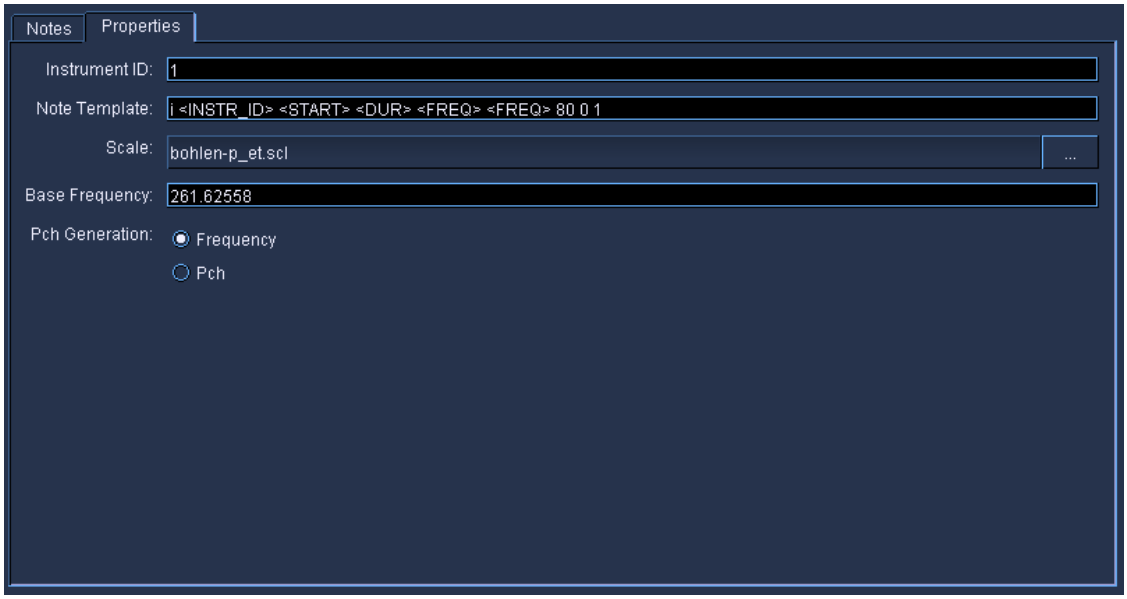
```
i<INSTR_ID> <START> <DUR> <FREQ><FREQ> 0 1 1
```

which does not have proper space between the two <FREQ> tags, results in:

```
i1 0 2 440440 0 1 1
```

Piano Roll Properties

The PianoRoll requires a bit of configuration before using. The properties page below shows the properties that should be configure before using the actual note drawing canvas.



PianoRoll Properties

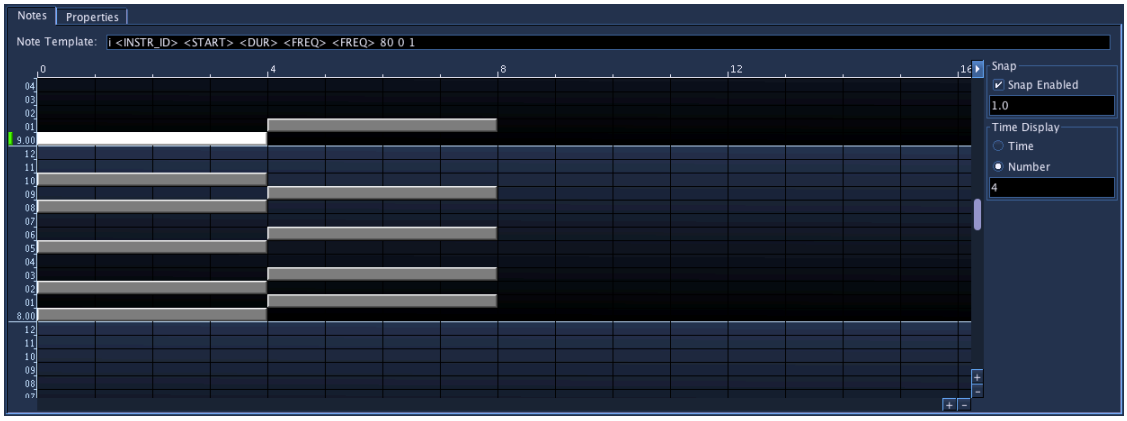
Instrument ID	Instrument name or number to be used when replacing <INSTR_ID> in Note template strings.
Note Template	The default note template when inserting notes. Notes make a copy of this template string when created and edits to the note's string stay with the note. Generally, you'll want to create a template string that will match the instrument this PianoRoll will be used with.
Scale	The scale used with this PianoRoll. The PianoRoll defaults to a 12-TET scale, the "standard" scale in use in Western classical and popular music. Pressing the button labeled "..." will open a file browser for selecting Scala scales to use in place of the default. After selecting a scale, the PianoRoll will adjust the note canvas for the number of scale degrees the newly selected scale contains.

Base Frequency — The base frequency of the scale for octave 8 and scale degree 0 (8.00). Defaults to C below A440.

Pch Generation — Selects how the notes will generate their value to be used when replacing the <FREQ> tag value in a note template. The options are:

- Frequency — The value of the note's pitch expressed in terms of frequency in hertz. This value is calculated using the chosen Scale for the PianoRoll.
- Blue PCH — Value of note expressed in Blue PCH, a format similar to Csound PCH but differs in that it does not allow fractional values. Values are generated as "octave.scaleDegree" i.e. "8.22" would be octave 8 and scale degree 22 in a scale that has 23 or more notes, or would wrap around as it does in Csound PCH. If the scale had 12 scale degrees, the "8.22" would be interpreted as "9.10". Blue PCH is allowed as an option to be used with Blue PCH note processors and then to be used with the Tuning NoteProcessor.
- MIDI — When this Pch Generation method is chosen, a MIDI note value (0-127, 60 = Middle-C) is used for <FREQ> and the chosen Scale will not be used. The display for the editor will automatically switch to show octaves and notes for standard MIDI scale values. Using MIDI note values is useful for instruments that exepct MIDI note values such as the fluidsynth opcodes as well as midiout.

Time Options



The Time Options in the PianoRoll are accessed and behave very much in the same manner as those that are in the main timeline. The button labelled "..." in the upper right corner of the PianoRoll canvas will open and close the panel on the right that contains the properties.

Time Options

Snap Enabled — Enables snapping behavior on the timeline. If enabled, vertical lines will be drawn at snap points, set by the value below it. In the screenshot above, the snap is enabled and set to every 1.0 beats.

Time Display — Controls how the time in the time bar above the PianoRoll canvas will display. The time value will show as time, while Number display will display as integers. The number below show how often to put a label. In the screenshot above, the Time Display is set to show a label in units of time and at every 5.0 seconds.

Using the Note Canvas

To enter notes, hold down the shift key and press the left mouse button down on the canvas. A note will be entered where you pressed and will be set to resize as you move the mouse around. When you finally release the mouse, the note will be finished entering.

After that, you can select notes by clicking on them or drag and selecting notes by marquee. You can also press the shift key and click on notes to add to the currently selected notes. You can then drag the notes around by click a selected note and dragging. To resize a note, select a single note, and after hilighted, move the mouse to the right edge of the selected now, and then click and drag.

To remove a note or notes, select the notes, then press the del key.

To cut or copy a note, select a single note (only one note in the buffer is currently supported), then press ctrl-x or ctrl-c to cut or copy, respectively.

To paste, ctrl-click on the PianoNote canvas. (This is the same behavior as pasting soundObjects on the main score timeline.)

To edit a note's template, select a single note. After selecting a note, the text field for editing the note's template text will be enabled. Here you can then edit the note's values.

For more Information

See the example .blue file in the blue/examples/soundObjects folder.

PolyObject

Accepts NoteProcessors: yes

A timeline object that acts as a container for other SoundObjects. PolyObjects can also be embedded within each other.

Try This

On the root timeline, rt-click on a soundLayer and select "Add new PolyObject". You should have added a new PolyObject to the timeline. Now either double-click or rt-click on the PolyObject and select "Edit SoundObject". You should now be within the PolyObjects timeline, and the button which says [root] should now have another button next to it that says [PolyObject]. Now try adding a few soundLayers and a few GenericScore Objects. Now click on [root]. You have now returned to the root timeline. Here you should see the PolyObject you've edited. Now you can scale the object you've created and all of the objects held within will scale together as a group.

PythonObject

Accepts NoteProcessors: yes

Allows for using of the Python programming language to generate score data, using the Jython interpreter to interpret Python scripts. You may add your own python classes to the library for use with "import" statements by adding them to your BLUE_HOME/pythonLib folder. Included with Blue is Maurizio Umberto Puxeddu's pmask, as well as Steven Yi's Orchestral Composition library, found in Blue's application directory under blue/pythonLib.

After writing your script to generate notes, assign the string value of the notes to the variable 'score'. Blue will then read in the value from that variable and continue processing.

Example

```
temp = ""

for i in range(4):
  temp += "i1 %d 1 %s %s\n"%(i, "8.0" + str(i), 80)

score = temp
```

The above example script will generate four notes at ascending half steps. If the PythonObject is set with a start time of 0 and a duration of 2, then it will generate the following score:

```
i1 0.0 0.5 8.00 80
i1 0.5 0.5 8.01 80
i1 1.0 0.5 8.02 80
i1 1.5 0.5 8.03 80
```

Regarding Processing

Blue processes soundObjects by going through each SoundLayer and generating score for each object within each layer. This is useful to know so that if you are using a PythonObject that has utility functions that you later use in other PythonObjects, you should put that utility PythonObject on the first SoundLayer closest to the top, or at least on a layer above all others that contain PythonObjects.

Also to note, as a feature, Blue uses a single interpreter instance for processing python code. Therefore, if one PythonObject has code evaluated, the values from that code can be read by other objects. This allows creating utility PythonObjects. However, one can use stale values (or values from another project even) if one is not careful to always assign values in the project that require being set for this particular project.

Variables from Blue

The following variables are avaialable from Blue:

Variables from Blue

blueDuration	Duration of the Python SoundObject
blueProjectDir	The location of the current project's directory. Includes path separator at end.

Process at Start

There is a checkbox entitled "Process at Start". Selecting this option will have the script of the PythonObject run when a .blue project is loaded. This is useful for scripts that act as library functions, but themselves do not generate any notes. For example, you might define a number of score generation utility functions in one PythonObject that has "Process at Start" enabled. Your other PythonObjects may then use

the functions from that PythonObject. Next time you load your project, if that PythonObject hasn't been run, your other PythonObjects will not be able to be run either. If you are rendering from the beginning of a project, this won't be an issue, but if you're starting work in the middle of a project, you will need to evaluate that utility PythonObject at least once. You can either do a run from the start at least once, use the "Test" button to have that evaluated, or use "Process at Start" and have Blue ensure it is loaded into the python interpreter when you load your projects.

JavaScriptObject

Accepts NoteProcessors: yes

Allows for the use of the JavaScript programming language to create score data. Uses the built-in JavaScript interpreter (Nashorn) to interpret JavaScript scripts.

After writing your script to generate notes, you'll have to bring back into Blue by assigning the variable 'score' the text string of the generated JavaScript score.

Example

```
[code for generating score]
...

score = myScoreGenerator();
```

(where myScoreGenerator() is a function that will generate Csound SCO)

Sound SoundObject

Accepts NoteProcessors: no

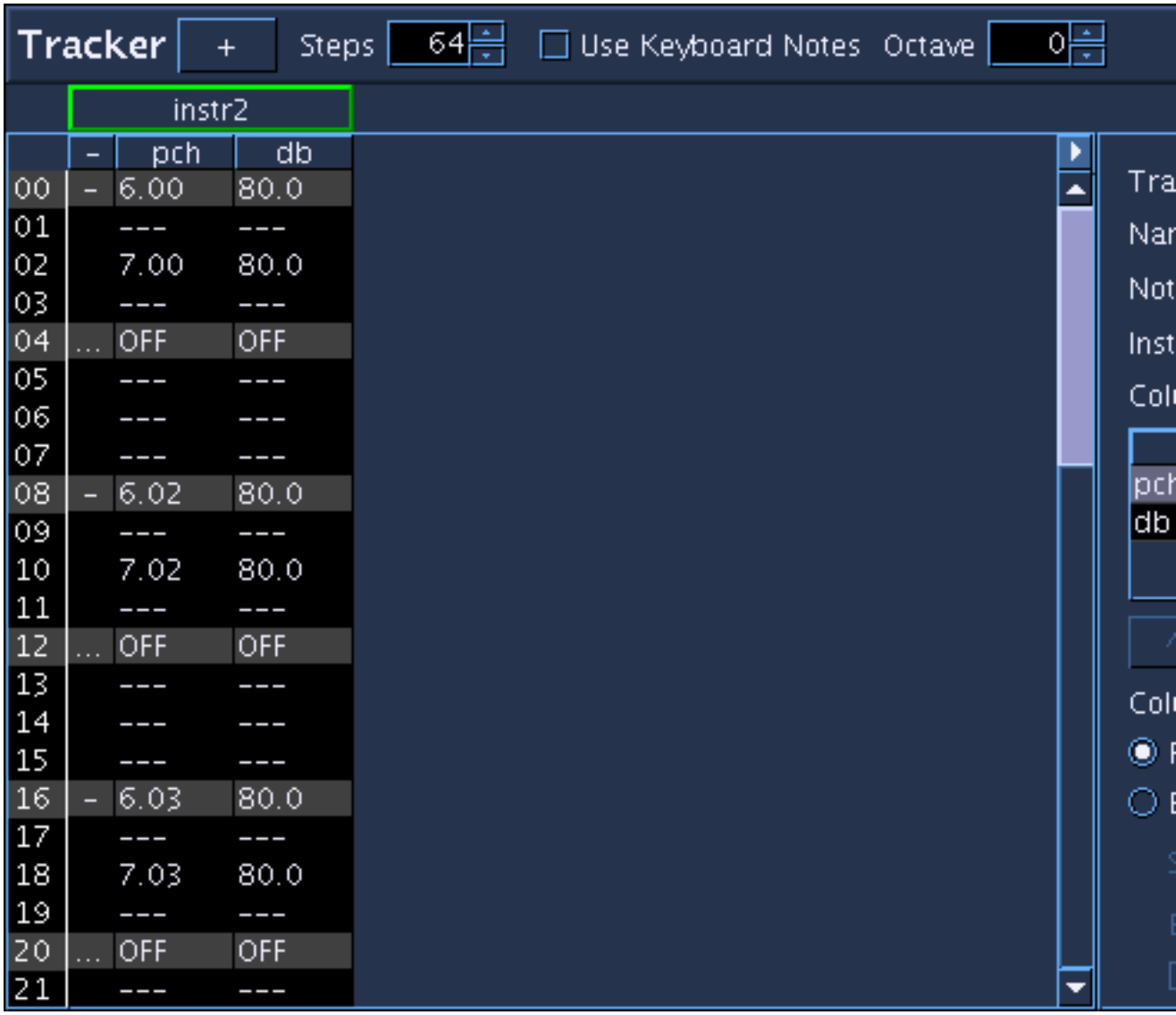
The Sound SoundObject allows one to develop solitary sounds by using Csound Orchestra code and graphical user interfaces. It uses the same GUI and coding system as found in BlueSynthBuilder.

Sound SoundObjects will generate an instrument as well as i-statement with the duration of the SoundObject. Sound SoundObjects can route signals to mixer sub-channels and master out by using the blueMixerOut pseudo-opcode, same as in BlueSynthBuilder. Multiple copies of the same Sound can be placed in the Score timeline.

The Sound SoundObject also has a special Automation panel that allows automating the controls. The automation is always scaled to the duration of the SoundObject and effectively encapsulates the automation changes with the object. Users can automate any widget that is allowed by BlueSynthBuilder.

Tracker

Accepts NoteProcessors: yes



About the Tracker

The Tracker SoundObject is a table-based tool to enter in patterns of notes using the Tracker paradigm but in a way specific to Csound SCO work. Each Tracker is organized in vertical Tracks of n number of steps where n is configurable by the user (defaults to 64 steps). Notes are entered into each Track with Tracks being configurable as to what parameters (columns) to use. Unique to the Blue Tracker SoundObject is the support for microtonal scales using Scala scale files as well as support for Csound's Tied-Notes feature.

Introduction to the Interface

The interface consists of three main areas: the top tool bar, the main tracking area, and the track properties editor.

The top tool bar has a "+" button which adds new tracks to the Tracker, a Steps entry widget to enter in the number of steps the Tracker should have, a toggle for using Keyboard Note mode (see below for more information), an Octave entry widget to determine the relative octave for the Keyboard Note mode, a Test button to see what the generated score will be for the Tracker, as well as a help button that opens up a quick reference sheet for keyboard shortcuts.

The main tracking area is where all of the score work is done to enter and modify notes. More information about note entry and modification is available below in the section "Entering in Notes".

The last interface area is the track properties editor. The track properties editor is held in a collapsible pane and when the tracker editor initially loads it will be collapsed. To open up the track properties, at least one track needs to be added to the tracker. Once one track exists, click the small button above the right scroll bar to open and close the track editor properties. More information on using the properties editor follows below in the section "Settings Things Up".

Setting Things Up

When the Tracker Object is edited it is completely blank. To start, click the "+" button to add as many tracks as you would like to use. After adding the number of tracks you'd like to use you will need to configure the track to work with your Csound instruments. Open up the track properties editor using the "<" toggle button above the right scroll bar. Afterwards, select a track by clicking the name panel above a track. Selecting a track will populate the track properties editor as well as hilight the name panel with a green border. A track's properties consist of a Name, Note Template, Instrument ID, and Columns; descriptions of the values are listed below.

Track Properties

Name — The name property is used only for reference; editing the name changes the title shown on the name panel and is for the user's reference.

Note Template — The note template is used when generating the notes for the track. Items in the template that are within < and > tags will be replaced by values either from the Tracker (START and DUR), the Instrument ID (INSTR_ID) or values from the columns, using the column's name as a key (i.e. if a column is called "space", when generating a note for the track, any value in the space column will replace the <space> text in the note template). Note templates will generally follow the form "i <INSTR_ID> <START> <DUR>" and then have tag keys for each column for the track.

Instrument — Instrument name or number to be used when replacing <INSTR_ID> in Note template strings.

Columns — Each track has a minimum of one configurable column (the tied-note is a feature of all tracks and is not a part of this editor) and is user-configurable to add as many columns as the user needs for the values to use in their notes for their instruments. Columns are added and removed using the Columns table and can be organized by pushing up and down in the table which will move their order left and right in the main tracking area. To edit the name of the Column, use the Columns table to edit the name of the column.

Each Column also has a type. The type information is used by the tracker when entering data to verify that the data being input is of that column's type, as well as used when using shortcuts to manipulate data in that column.

Column Properties

PCH — Csound PCH format. Entering data will verify that data is in the octave.pitch format. Using the increment and decrement value shortcuts will propertly add or subtract one to pitch value, i.e. incrementing the value of 8.11 will result in 9.00.

Blue PCH
: The Blue PCH format is like the Csound PCH format except that the pitch part is always a whole number integer and is the scale degree of the selected Scale. A valid value in Csound PCH such as 8.01 is not valid in Blue PCH as 01 is not an integer (the equivalent in Blue PCH would be 8.1).

  Using Blue PCH allows for using Scala scale files to do microtonal tracking. To choose a Scala scale, use the "..." button to open up a file selector to choose a Scala scale. Afterwards, enter in the base frequency for the scale (the default is 261.62558 or middle-c). The "output frequencies" checkbox will determine how the values entered into this column will be interpreted. By default, "output frequencies" is enabled, meaning when the tracker goes to generate notes, it will take the Blue PCH values that are entered and convert them to a frequency value in hertz. If you deselect this option, the tracker will pass the Blue PCH value out. This option should generally be left enabled unless the user is planning to do further operation on the pch values via NoteProcessors that work with Blue PCH.

  Using Blue PCH, data will be verified on entry and increment/decrement value options will work in the same way as for PCH.

MIDI
: MIDI will limit the values entered to whole number integers from 0-127. Using the increment and decrement value shortcuts will add or subtract 1 to the value.

String
: The String type allows the user to input any value they want. No verification is done on entry and the increment/decrement value shortcuts will have no effect.

Number
: The Number format will limit the values entered to only numbers. Values can be further restricted to a given range as well as to only use whole number integers. Using the increment and decrement value shortcuts will add or subtract 1 to the value.

Entering in Notes

Entering in data into the tracker is much like entering data into any other table, though learning the keyboard shortcuts will vastly speed up entering and modifying data. To begin click anywhere on a track where you would like to add a note. Now, begin typing to enter in a value for that note, then press enter when you are finished. Depending on the type of column you have configured, Blue will verify that the data entered is allowable and if so it will save that data to the note. If the value is not allowable, the cell will become hilighted in red and will require you to either fix your input to be valid or press esc to cancel entering in data.

When entering in data for a new note, the first time you enter in information for a column in the note's row, it will not only enter in the data for the column, but also copy values for all other columns from the first note that exists previous to the note being edited. If there is no notes entered, some default settings will be used based on the column type.

Like other tracker systems, the duration of a note entered will last as long as until either the next entered note, the end of the pattern, or until an OFF note is encountered. So, for example, if a note is entered in step 0 and step 2, the duration of the first note will last 2 steps while the second note will last until the end of the pattern (62 steps in a default 64 step track). To enter an OFF statement, go to the row where

you want the note to end and press ctrl-shift-space. This will make the row an OFF note. So, if a note is entered in step 0 and step 2 and an OFF is entered into step 1 and step 4, the first note will last 1 step while the second note will last 2 steps.

To increment and decrement values in a cell, use the arrow keys to go over the cell you want to increment or decrement and then use ctrl-up or ctrl-down respectively to change the value. (NOTE: This operation operates differently for each column type and does nothing for the String type. Please see the column type information above for more information.)

Keyboard Note Mode

Like most trackers, the Tracker object has keyboard shortcuts that will allow for very quickly adding notes. To enable Keybaord Note mode, either click the checkbox on the top tool bar or use the keyboard shortcut ctrl-k. By enabling Keyboard Note mode, the keys on the keyboard will be mapped to note values much like a piano keyboard. When the selected cell is of type PCH, Blue PCH, or MIDI, pressing those keys will enter in a value related the keyboard mapping (see Shortcuts section).

The user is also able to change the base octave of the Keyboard Note mode. To change the octave, use either the spinner control on the top tool bar or use the keyboard shortcuts ctrl-shift-up or ctrl-shift-down. By default, the base octave starts at middle-c.

Shortcuts

Table 3.3. Keyboard Shortcuts

Shortcuts	Description
ctrl-space	clear or duplicate previous note
ctrl-shift-space	set or clear OFF note
ctrl-up	increment value
ctrl-down	decrement value
ctrl-t	toggle note tie
ctrl-x	cut selected notes
ctrl-c	copy selected notes
ctrl-v	paste notes from copy buffer
insert	insert blank note into currently selected row, notes in current row and after are shifted down; if notes are at end are shifted off they are lost
del	delete selected note(s), move selection to next row after current selection
shift-backspace	delete selected notes, notes after selected notes are shifted up to fill in place where deleted notes were, empty notes appended to end
ctrl-k	toggle keyboard notes mode
ctrl-shift-up	raise keyboard octave by one
ctrl-shift-down	lower keyboard octave by one

Table 3.4. Keyboard Note Mode

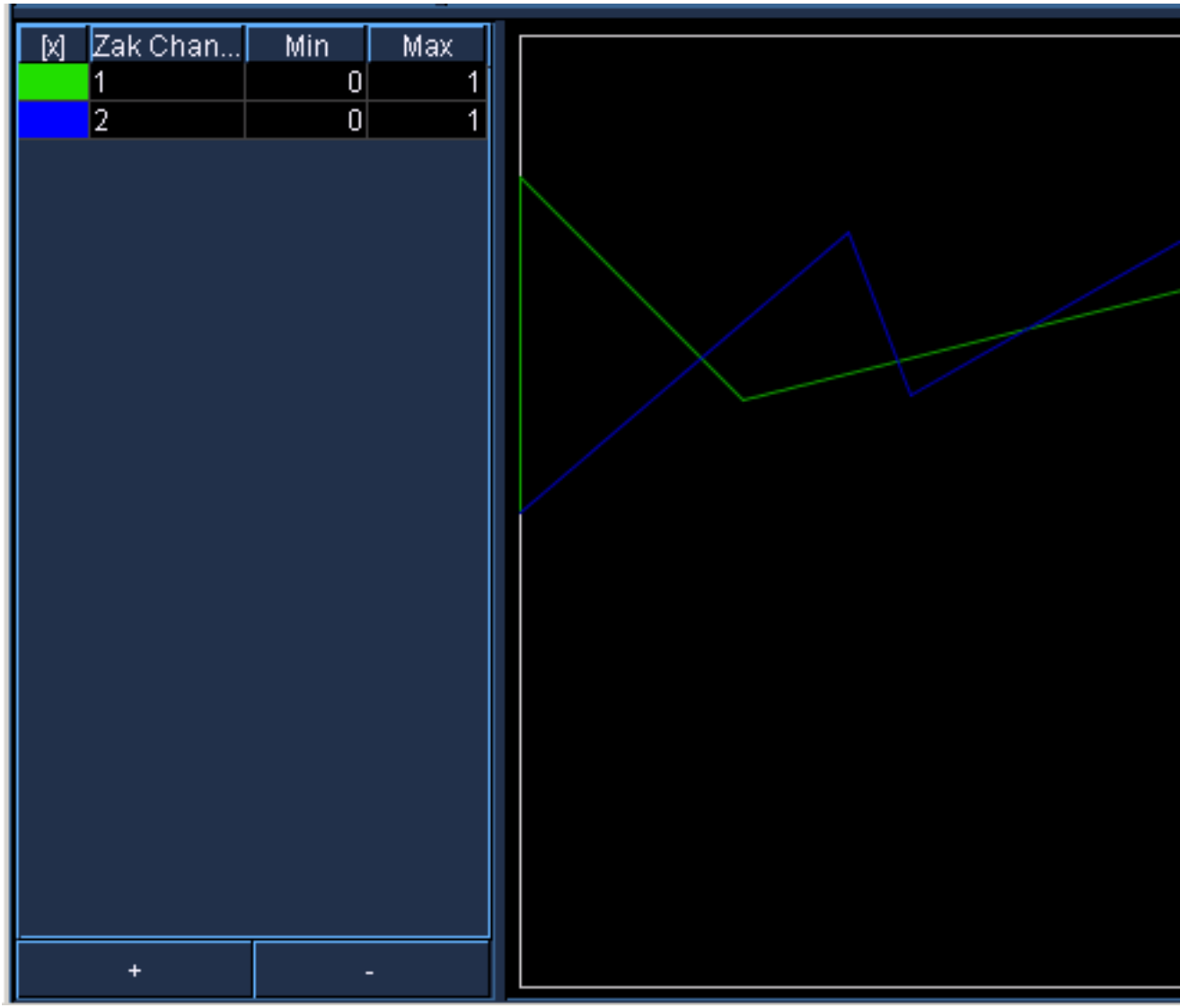
Shortcut	PCh Value	Blue PCH Value	MIDI Value
z	8.00	8.0	60
s	8.01	8.1	61
x	8.02	8.2	62
d	8.03	8.3	63
c	8.04	8.4	64
v	8.05	8.5	65
g	8.06	8.6	66
b	8.07	8.7	67
h	8.08	8.8	68
n	8.09	8.9	69
j	8.10	8.10	70
m	8.11	8.11	71
q	9.00	9.0	72
2	9.01	9.1	73
w	9.02	9.2	74
3	9.03	9.3	75
e	9.04	9.4	76
r	9.05	9.5	77
5	9.06	9.6	78
t	9.07	9.7	79
6	9.08	9.8	80
y	9.09	9.9	81
7	9.10	9.10	82
u	9.11	9.11	83
i	10.00	10.0	84
9	10.01	10.1	85
o	10.02	10.2	86
0	10.03	10.3	87
p	10.04	10.4	88

For More Information

See the tracker.blue example file in the blue/examples/soundObjects folder.

ZakLineObject

Accepts NoteProcessors: no



Zak Line Object

Add and graphically edit zak k-rate signals.

Use the bottom + and - buttons to add/remove lines.

Use the left panel to select a line by clicking on the row describing the line you wish to edit; you'll know which one is select by the edit points showing up in the right panel showing the line. To edit the color of the line, double click in the left panel on the color box to the left of the "Zak Channel" column. A color selection dialog will appear for the user to choose their desired color.

Enter the zak channel that you whish the signal to be written to under the "Zak Channel" column. The "min" and "max" columns define the min and max values of the graph that the line is drawn in to the right. However, the min and max values can be different on a per-line basis.

When editing the line in the right panel, left clicking adds a point at the current cursor position. Right-clicking will delete a point when a point is under the cursor (you can easilly tell this because the a point will change to red when the cursor is hovering over it). You can move a point by left-clicking and dragging.

NoteProcessors

NoteProcessors are used in conjunction with soundObjects, and are used post-generation of the soundObject's noteList. They are used to modify values within the noteList.

NoteProcessors can be added via the soundObject property dialog. When a soundObject is selected on the timeline, and if the soundObject supports noteProcessors, you can add, remove, push up, or push down noteProcessors on the property dialog.

Role in Score Generation

NoteProcessors are applied after the notes of the soundObject are generated and before time behavior is applied. Processing starts with the first NoteProcessor in the chain and the results of that are passed down the chain.

Add Processor

Parameters: pfield, value

The AddProcessor takes two parameters, one for pfield (positive integer > 0) and one for value (any number). When applied, it will add the user-defined value to the set pfield for all notes.

For example, if you have a SoundObject with notes for an instrument that uses p4 as its amplitude. If you have values for p4 within the range of 78 and 82, such as the following score:

```
i1 0 2 78
i1 + . 80
i1 + . 81
i1 + . 82
```

If an AddProcessor is used that was set to 4.4 for value 4 for pfield, your notes afterwards would be moved to the range of 82.4 to 86.4:

```
i1 0.0 0.5 82.4
i1 0.5 0.5 84.4
i1 1.0 0.5 85.4
i1 1.5 0.5 86.4
```

(The p2 and p3 times above are post-processing for a 2 second duration SoundObject with time behavior set to scale.)

Equals Processor

Parameters: pfield, value

Sets user-given pField of all notes in soundObject to user-given value, i.e. set all p4's to value "440", or set all p6's to value "/work/audio/wav/mySample2.wav". The following score:

```
i1 0 2 8.00
i1 + . 8.04
i1 + . 8.07
i1 + . 9.00
```

If used with an EqualsProcessor with value 7.00 and pfield 4, would result in the following score:

```
i1 0.0 0.5 7.00
i1 0.5 0.5 7.00
i1 1.0 0.5 7.00
i1 1.5 0.5 7.00
```

(The p2 and p3 times above are post-processing for a 2 second duration SoundObject with time behavior set to scale.)

Tip: One can use this NoteProcessor to quickly try testing a score with another insturment. To do this, use this NoteProcessor to reassign p1.

Inversion Processor

Parameters: pfield, value

This NoteProcessor flips all values in designated pfield about an axis (value). The following score:

```
i1 0 2 80
i1 + . 85
i1 + . 95
i1 + . 100
```

If processed with an InversionProcessor with value 90 and pfield 4 would result in:

```
i1 0.0 0.5 100.0
i1 0.5 0.5 95.0
i1 1.0 0.5 85.0
i1 1.5 0.5 80.0
```

(The p2 and p3 times above are post-processing for a 2 second duration SoundObject with time behavior set to scale.)

LineAdd Processor

Parameters: pfield, LineAdd String

The LineAdd Processor adds values to a user-defined pfield. This noteProcessor differs from the AddProcessor in that the value added is variable over time. The LineAdd String is a set of beat/value pairs that are like the breakpoints on curve. The following score:

```
i1 0 2 80
i1 + . 80
i1 + . 80
```

```
i1 + . 80
```

If processed with a LineAdd Processor with LineAddString set to "0 0 6 3" and pfield set to 4, would result in:

```
i1 0.0 0.5 80.0
i1 0.5 0.5 81.0
i1 1.0 0.5 82.0
i1 1.5 0.5 83.0
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

The LineAddString can be interpreted to mean: start at beat 0 with value 0, and by beat 6, arrive at value 3. For each note, the LineAddString will find the note's beat and compare against the LineAddString to see how much should be added. The user should be careful to remember that SoundObjects do not apply scaling of score until after processing with noteProcessors. Therefore, the beat values given in the LineAddString were to be applied against the beat values of the original score.

LineMultiply Processor

Parameters: pfield, LineMultiply String

The LineMultiply Processor multiplies values in a user-defined pfield. This noteProcessor differs from the Multiply Processor in that the value muliplied by is variable over time. The LineMultiply String is a set of beat/value pairs that are like the breakpoints on curve. The following score:

```
i1 0 2 80
i1 + . 80
i1 + . 80
i1 + . 80
```

If processed with a LineMultiple Processor with LineMultiplyString set to "0 0 6 2" and pfield set to 4, would result in:

```
i1 0.0 0.5 0.0
i1 0.5 0.5 53.333336
i1 1.0 0.5 106.66667
i1 1.5 0.5 160.0
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

The LineMultiplyString can be interpreted to mean: start at beat 0 with value 0, and by beat 6, arrive at value 2. For each note, the LineMultiplyString will find the note's beat and compare against the LineMultiplyString to see how much should be multiplied. The user should be careful to remember that SoundObjects do not apply scaling of score until after processing with noteProcessors. Therefore, the beat values given in the LineMultiplyString were to be applied against the beat values of the original score.

Multiply Processor

Parameters: value, pfield

The MultiplyProcessor works like the addProcessor, but multiplies the given pfield by the value. The following score:

```
i1 0 2 80
i1 + . 80
i1 + . 80
i1 + . 80
```

If processed with a MultiplyProcessor with value 2 and pfield 4 would result in:

```
i1 0.0 0.5 160.0
i1 0.5 0.5 160.0
i1 1.0 0.5 160.0
i1 1.5 0.5 160.0
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

Pch Add Processor

Parameters: pfield, value

The PchAddProcessor works like the AddProcessor, but is used with pfields written in Csound pch notation(i.e. 8.00, 7.11, 9.03). The value field, unlike the AddProcessor, is a whole-number, representing the number of scale degrees in which to add to the pch value. If a soundObject has notes where p5 is a pch value, and you the notes have values of 8.00, 8.04, 8.07, and 9.00:

```
i1 0 2 8.00
i1 + . 8.04
i1 + . 8.07
i1 + . 9.00
```

If a PchAddProcessor with value 7 and pfield 5 is used, after processing, the score would result in:

```
i1 0.0 0.5 8.07
i1 0.5 0.5 8.11
i1 1.0 0.5 9.02
i1 1.5 0.5 9.07
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

If you had used a regular AddProcessor and tried to transpose 7 scale degrees down using -.07, you would have gotten 7.93, 7.97, 8.00, and 8.97, and the first few notes would have really be equal to 14.11, 15.01, and 8.00 in pch notation. This is due to the way that Csound pch notation works to hold the scale degree, from 0-11, in the two fields to the right of the period.

Pch Inversion Processor

Parameters: value, pfield

This noteProcessor flips all values in designated pfield about an axis (value). The values that are read in the given pfield will be interpreted as Csound pch format. The following score:

```
i1 0 2 8.00
i1 + . 8.04
i1 + . 8.07
i1 + . 9.00
```

If processed with a PchInversionProcessor with value 8.06 and pfield 4 would result in:

```
i1 0.0 0.5 9.0
i1 0.5 0.5 8.08
i1 1.0 0.5 8.05
i1 1.5 0.5 8.0
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

Python Processor

Parameters: code

Allows user to write python code to process NoteList. The code is run using Jython, the Java implementation of Python that is included with Blue. Users using the PythonProcessor should be aware of Jython's implementation details: most classes and modules from CPython are included, users can not import modules with native libraries, and users can use any Java class that exists in the classpath.

When Blue generates notes from a SoundObject, the SoundObject first generates its Notes as a NoteList, then NoteProcessors are called one at a time to process that NoteList in place. For the PythonProcessor, before any of the user-written code is run, the generated NoteList is first added and defined in the memory space as "noteList". From there, the script for the PythonProcessor should treat noteList as the list of Note objects that have been generated. The NoteList and Note objects are the very same and have the same methods and properties as those in the Java code, so one must work with them the same way as one would in Java.

The following code shows an example of the Python Processor. The code first imports the random module, then for every note in the noteList it creates a random space variable between the values of -1 and 1, then assigns that to pfield 7 of the note. (This example code is used in the blue/examples/noteProcessors/pythonProcessor.blue and can be run within Blue to hear the example).

```
import random

for i in noteList:
  newVal = str((2 * random.random()) - 1)
  i.setPField(newVal, 7)
```

Random Add Processor

Parameters: pfield, min, max, seedUsed, seed

The RandomAddProcessor generates a random value between min and max and adds it to the designated pfield for all notes. A random value is generated for each note.

The RandomAddProcessor expects either a positive or negative float value for the min and max, and a postive integer for the pfield. The following score:

```
i1 0 2 80
i1 + . 80
i1 + . 80
i1 + . 80
```

when processed with an RandomAddProcessor with min set to 0.0, max set to 1.0, and pfield set to 4, resulted in the following on one pass:

```
i1 0.0 0.5 80.92294
i1 0.5 0.5 80.50539
i1 1.0 0.5 80.112495
i1 1.5 0.5 80.93934
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

The above is an example of random values, and because it is random, the results will be different on another processing pass. .

If seedUsed is set to "true", the random number generator will be initialized using the given seed value. The seed value must be a valid long integer value (-2^63, 2^63-1).

Hint: One useful way to use this noteProcessor is as a way to "humanize" velocity or pitch values values.

Random Multiply Processor

Parameters: pfield, min, max

The RandomMultiplyProcessor takes a pfield to apply the random multiply to, as well as a min and max for the boundaries of the random values. When applied, it will multiply a random value for each note in the assigned pfield, using a new random value per note.

The RandomMultiplyProcessor expects either a positive or negative float value for the min and max, and a postive integer for the pfield. The following score:

```
i1 0 2 80
i1 + . 80
i1 + . 80
i1 + . 80
```

when processed with an RandomMultiplyProcessor with min set to 1.0, max set to 2.0, and pfield set to 4, results in the following:

```
i1 0.0 0.5 85.553246
i1 0.5 0.5 148.94167
i1 1.0 0.5 125.57565
i1 1.5 0.5 97.00755
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

If seedUsed is set to "true", the random number generator will be initialized using the given seed value. The seed value must be a valid long integer value (-2^63, 2^63-1).

The above is an example of random values, and because it is random, the results will be different on another processing pass.

Retrograde Processor

Parameters: none

Reverses all the generated notes in time. The following score:

```
i1 0 2 1
i1 + . 2
i1 + . 3
i1 + . 4
```

If processed with a RetrogradeProcessor would result in:

```
i1 1.5 0.5 1
i1 1.0 0.5 2
i1 0.5 0.5 3
i1 0.0 0.5 4
```

which if re-sorted by start time would result in:

```
i1 0.0 0.5 4
i1 0.5 0.5 3
i1 1.0 0.5 2
i1 1.5 0.5 1
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

Rotate Processor

parameters: noteIndex

The RotateProcessor rotates the order of the notes, making the noteIndex the first note. For example a noteIndex of 2 means "make the second note of the notes generated the first, shifting the first note to the end". With a noteIndex of 3, the following score:

```
i1 0 2 1
i1 2 2 2
i1 4 2 3
i1 5 2 4
i1 6 2 5
```

will become:

```
i1 0 2 3
i1 2 2 4
i1 4 2 5
i1 5 2 1
i1 6 2 2
```

As the third note now becomes the first. If a negative number is given, the number will count backwards as to what note should now become the first note. With a noteIndex of -2 (meaning, "Second note from the end"), the following score:

```
i1 0 2 1
i1 2 2 2
i1 4 2 3
i1 5 2 4
i1 6 2 5
```

Will become:

```
i1 0 2 4
i1 2 2 5
i1 4 2 1
i1 5 2 2
i1 6 2 3
```

Start times of notes are all modified such that the notes that are pushed to the end of list will start at the end time of the last note before it. For the above examples, the duration of the note list ast a whole did not change at all. However, when there are overlapping notes as in the following example, where note index of -3 was used:

```
i1 0 5.0 0 4
i1 2 5.0 1 4
i1 4 5.0 2 4
i1 6 5.0 3 4
i1 8 5.0 4 4
i1 10 5.0 5 4
i1 12 5.0 6 4
i1 14 5.0 7 4
i1 16 5.0 8 4
i1 18 5.0 9 4
```

The resultant scores duration has changed as shown below:

```
i1 0.0 5.0 7 4
i1 2.0 5.0 8 4
i1 4.0 5.0 9 4
i1 9.0 5.0 0 4
i1 11.0 5.0 1 4
i1 13.0 5.0 2 4
i1 15.0 5.0 3 4
i1 17.0 5.0 4 4
i1 19.0 5.0 5 4
```

```
i1 21.0 5.0 6 4
```

Please be aware of this behavior when using this NoteProcessor with scores that have overlapping notes.

SubList Processor

Parameters: start, end

The SubListProcessor will cut out notes from the soundObject's generated noteList. An example of it's use may be that you have a 12-tone row as soundObject in the soundObject library, and you're using instances of it as the basis of your work. You may only want to use notes 1-3 of the row, so you would use the SublistProcessor with a start of 1 and an end of 3.

The SubListProcessor will cut out notes, then translate them to start at the start of the soundObject, and then scale them so that they take up the duration of the soundObject. If you had a five note soundObject with all notes have a duration of 1 second, all starting one after the other, with the soundObject starting at 0 seconds on the timeline, and if you used a SubListProcessor with start of 1 and end of 4, you'd end up with four notes being generated(the first four from the original soundObject), starting a 0 seconds on the timeline, with each notes duration lasting 1.25 seconds, each starting one right after the other.

The following score:

```
i1 0 2 1
i1 + . 2
i1 + . 3
i1 + . 4
```

when processed with a SubList processor with start 2 and end 3, would result in:

```
i1 0.0 1 2
i1 1.0 1 3
```

(The p2 and p3 times above are post-processing for a 2 second duration soundObject with time behavior set to scale.)

Switch Processor

parameters: pfield1, pfield2

A Switch Processor switches pfield1 with pfield2 for all notes in SoundObject. It is useful in conjunction with RetrogradeProcessor when reversing notes that have start and end values in pfields, i.e.

```
;inum start dur   start   end
i1    0     1     8.04    8.00
i1    1     2     8.00    8.07
```

with just retrograde processor becomes:

```
;inum start dur   start   end
i1    0     2     8.00    8.07
```

```
i1    2     1     8.04    8.00
```

with retrograde and switch on p4 and p5 becomes:

```
;inum start dur   start   end
i1    0     2     8.07    8.00
i1    2     1     8.00    8.04
```

Time Warp Processor

parameters: TimeWarpString

Warps time in the same way as Csound t-statement, but does not require "t" to be used. Statements are in alternating pairs of beat number and tempo.

From the Csound Manual:

Time and Tempo-for-that-time are given as ordered couples that define points on a "tempo vs. time" graph. (The time-axis here is in beats so is not necessarily linear.) The beat-rate of a Section can be thought of as a movement from point to point on that graph: motion between two points of equal height signifies constant tempo, while motion between two points of unequal height will cause an accelarando or ritardando accordingly. The graph can contain discontinuities: two points given equal times but different tempi will cause an immediate tempo change.

Motion between different tempos over non-zero time is inverse linear. That is, an accelerando between two tempos M1 and M2 proceeds by linear interpolation of the single-beat durations from 60/M1 to 60/M2.

The first tempo given must be for beat 0.

Notes on Usage

- Beat values for beat/tempo pairs should related to the score *before* any time behavior is applied. For example, for the following score:

  ```
  i1 0 1 2 3 4
  i1 1 1 3 4 5
  i1 2 1 3 4 5
  i1 3 1 3 4 5
  ```

  if it is in a GenericScore SoundObject of duration 20, if you want the tempo to decrease in half by the last note, you would enter a value for the processor as "0 60 3 30" and not "0 60 20 30"

- If you're using a time behavior of "Repeat", remember that time behavior is applied *after* noteProcessors, and the resulting score will be a time warped score repeated x times and *NOT* a score repeated x time and then timewarped

- Time Warping, when used with a time behavior of "Scale", be aware that estimating the final tempo of the object may be tricky, as the scaling will alter the duration of notes.

Tuning Processor

Parameters: pfield, baseFrequency, scalaFile

Converts Blue PCH notation to frequency according to scale values in a Scala scale file. The scale will default to 12TET wheen TuningProcessor is initially created. The file selector for choosing a Scala .scl file will default to user's .blue directory, under the scl subdirectory. It is advised that users download the 3000+ scale archive from the Scala website at: http://www.huygens-fokker.org/scala/ and place them in the .blue/scl directory or whever they find convenient.

Base frequency determines what 8.00 should be set to, defaulting to middle-C below A440 (same as in Scala). Input for the noteProcessor should be:

```
oct.scaleDegree
```

where oct is equal to octave, and scale degree equal to what degree of the scale to use. The output will be frequency values, so instruments should be set to accept values as frequency.

NOTES

- Fractional scaleDegree's are not supported.
- For the scaleDegree, the Blue PCH does not work exactly like Csound pch notation. In Csound pch, "8.01" would be different than "8.1", while for the Tuning processor, there is no difference. The tuning processor takes everything from the right side of the decimal and converts that to an integer to figure out the scale degree.
- If you enter in a scale degree higher than the number degrees in the scale, it will be converted as if the octave is raised. For example, in a 19 tone scale, an input of "8.21" would get converted to "9.1".

Instruments

Blue instruments are a higher level construct than Csound instruments. They are designed to allow encapsulating all dependencies together with the instrument, thus allowing easy sharing of instruments between projects. Blue instruments may use Csound ORC code, User-Defined-Opcodes, and f-tables, and they may have a graphical user interface.

Generic Instrument

A generic editor for Csound instruments. Insert your instrument text in the editor, without "instr" or "endin", as they're not necessary and will be generated by blue.

Python Instrument

Use Python code to generate Csound instrument text.

JavaScript Instrument

Use JavaScript code to generate a Csound instrument.

BlueX7

Note

This instrument is currently undergoing re-implementation.

A 6 Operator Phase Modulation instrument using Russell Pinkston's DX7 Emulation Patches.

The BlueX7 editor contains two tabs, the 'patch' tab, where the sound creation parameters are tweaked and a second tab called 'csound' which can contain further post-processing algorithms or special routing of the instrument's output.

The patch tab contains three panels: Common, LFO and operators. The common panel deals with global aspects of the instrument like transposition, FM algorithm used, feedback, operator enable/disable and LFO characteristics. The transposition made by Key Transpose is handled in half-tones, with C3 being the normal (central) pitch.

There are 32 algorithms to choose from which represent the original DX7 possibilities. You can see a visual representation of the algorithm on the left of this panel.

Feedback represents the amount of feedback into a modulator. The position and result of this feedback depends entirely on the algorithm used. The range of feedback is 0 to 7(this range is the same as on the original DX7). The operator on/off checks turn on and off operators (current, this is here but the emulation patches do not use them).

The LFO panel sets the global LFO parameters. The parameters are: Speed, Delay, PMD (Pitch Modulation Depth) and AMD (Amplitude Modulation Depth). They all have ranges from 0 to 99(the range is the same as on the original DX7). Additionally you can select the shape of the LFO from: triangle, saw up, saw down, square, sine and sample-and-hold. You can also sync the operators: if sync is on, changing Modulation Sensitivity for pitch (marked with an asterisk *) on any of the operators will affect the rest.

Finally you have the 6 operators plus an envelope generator (PEG) each on its own tab. All six operators have the same parameters divided in the following sections:

- Oscillator
- Keyboard Level Scaling
- Operator
- Modulation sensitivity
- Envelope Generator

An important feature of the BlueX7 is that it can import DX7 banks. A large collection can be found in the bigdx7.zip file within the dx72csound.zip file from: http://www.parnasse.com/dx72csnd.shtml. (to steven: reading this page it seems part of the model is unfinished, which may answer some of my questions above)

On the 'csound' tab you find by default:

```
blueMixerOut aout, aout
```

This routes the output of the instrument directly to the stereo output of csound. You can include further code to process the 'aout' signal produced by the BlueX7, or to route it as needed. For example, if you are outputing in mono, you could code such as:

```
out aout
```

To call the BlueX7 instrument in the orchestra, create a GenericScore object in the timeline, and call the the BlueX7 instrument. BlueX7 requires 2 additional p-fields. P-field 4 is the pitch class of the note in

format octave.semitone (e.g. C4 is 4.00, C#1 is 1.01 and F6 is 6.05). P-field 5 contains the midi velocity for the note with values from 0 to 127. For example:

```
i1 0 1 8.00 100
i1 + 1 8.02 100
i1 + 1 8.04 100
i1 + 1 8.05 100
i1 + 1 8.07 100
```

BlueSynthBuilder

BlueSynthBuilder (BSB) allows the user to graphically build instrument interfaces for their Csound instruments. The graphical interface is designed for the use of exploring configuration of an instrument as well as adjustment to values in realtime(requires enabling using the Csound API, see here.

About BlueSynthBuilder

Principle of Design

It has been my experience that most Csound instruments tend to be limited in user configurability of parameters than commercial synthesizer counterparts. It is my belief that a large part of that is due to the text-based nature of instruments. I've found that:

- Modular instruments are easier to express connections of modules via text or code rather than visual paradigms (patch cables, line connections), and thus easier to create the instrument by text
- Graphical elements, however, excel in relaying information about the configuration of the instrument to the user and also invite experimentation, while text-based configuration of instruments is often more difficult to quickly understand the parameters settings and limits

Going completely graphical for the building of instruments, in the case of systems like Max/PD/jMax or Reaktor, I've found that the instrument's design no longer become apparent when viewing complicated patches. On the other hand, using completely textual systems such as Csound or C++ coding, the design of the instrument has a degree of transparency, while the configuration of the parameters of the instrument becomes difficult to understand and invites less exploration.

Hybrid Design

Using systems like MacCsound's widgets or Blue's BlueSynthBuilder, one is able to use graphical elements where they excel, in showing configuration of an instrument and for manipulation of values, while using textual elements where they excel, in the design of instruments and expressing the connections between modules.

I have found that this sort of hybrid design offers the best of both worlds and when I am spending time building and using new instruments, I can quickly design an instrument and also explore the parameters of an instrument's design by using BlueSynthBuilder.

Using BlueSynthBuilder

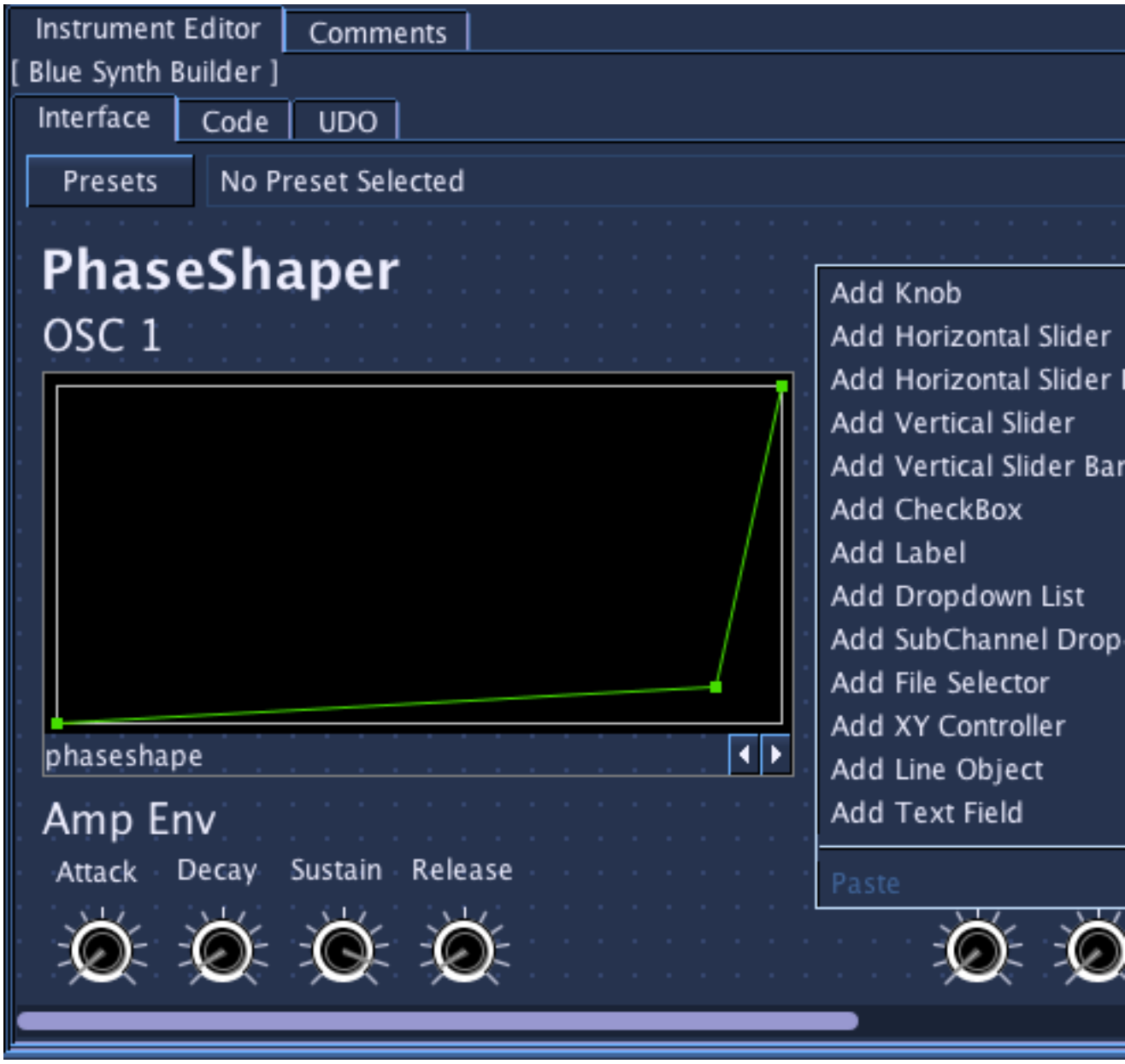
BlueSynthBuilder is divided up into three tabs, each of which handles the different concerns of the instrument builder. Also, the process of creating instruments and using instruments is also split between

edit and *usage* modes. Instrument builders will tend to use both modes, while users of BSB instruments may not ever have to touch a line of Csound instrument code or have to modify the instrument UI at all.

Interface Editor

The Interface editor has two modes:

Edit Mode

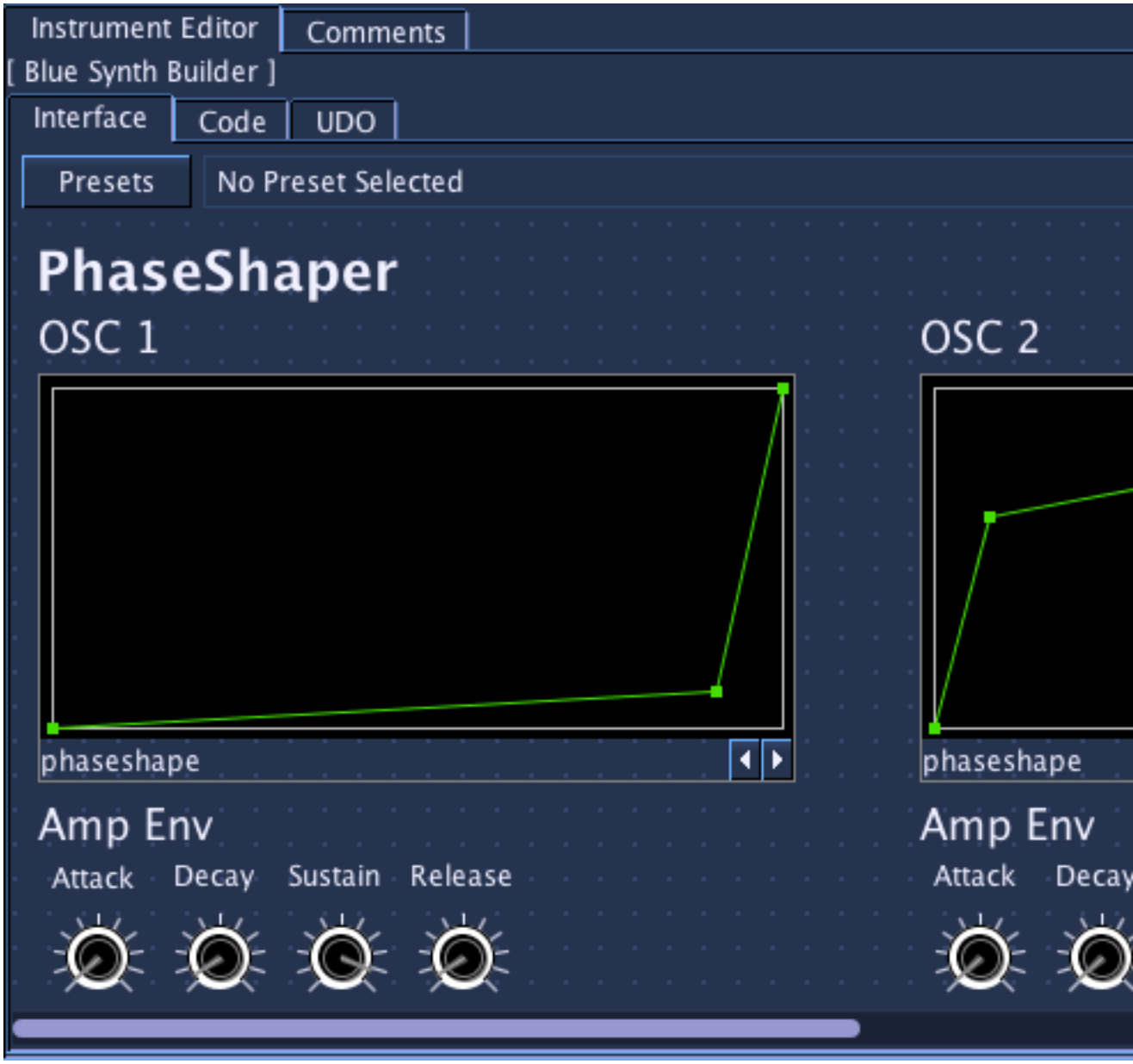


The user can add interface elements and modify their properties using the property sheet on the right. To enable edit mode, click on the "Edit Enabled" checkbox in the upper right of the BSB instrument editor.

Once the edit mode is enabled, right clicking in the main panel will show a popup menu of available UI widgets for your instrument. After selecting and inserting a widget, clicking on the widget will hilight it and show it's properties in the property editor. You can also then drag the widget around to place as you desire.

You may select multiple widgets by shift-clicking them, or by clicking on an empty part of the edit panel and dragging a marquee to select a group of widgets. After selecting multiple widgets, you can drag them around as a group, as well as use the alignment and distribution options found on the right side bottom to tidy up the UI.

Grid Settings



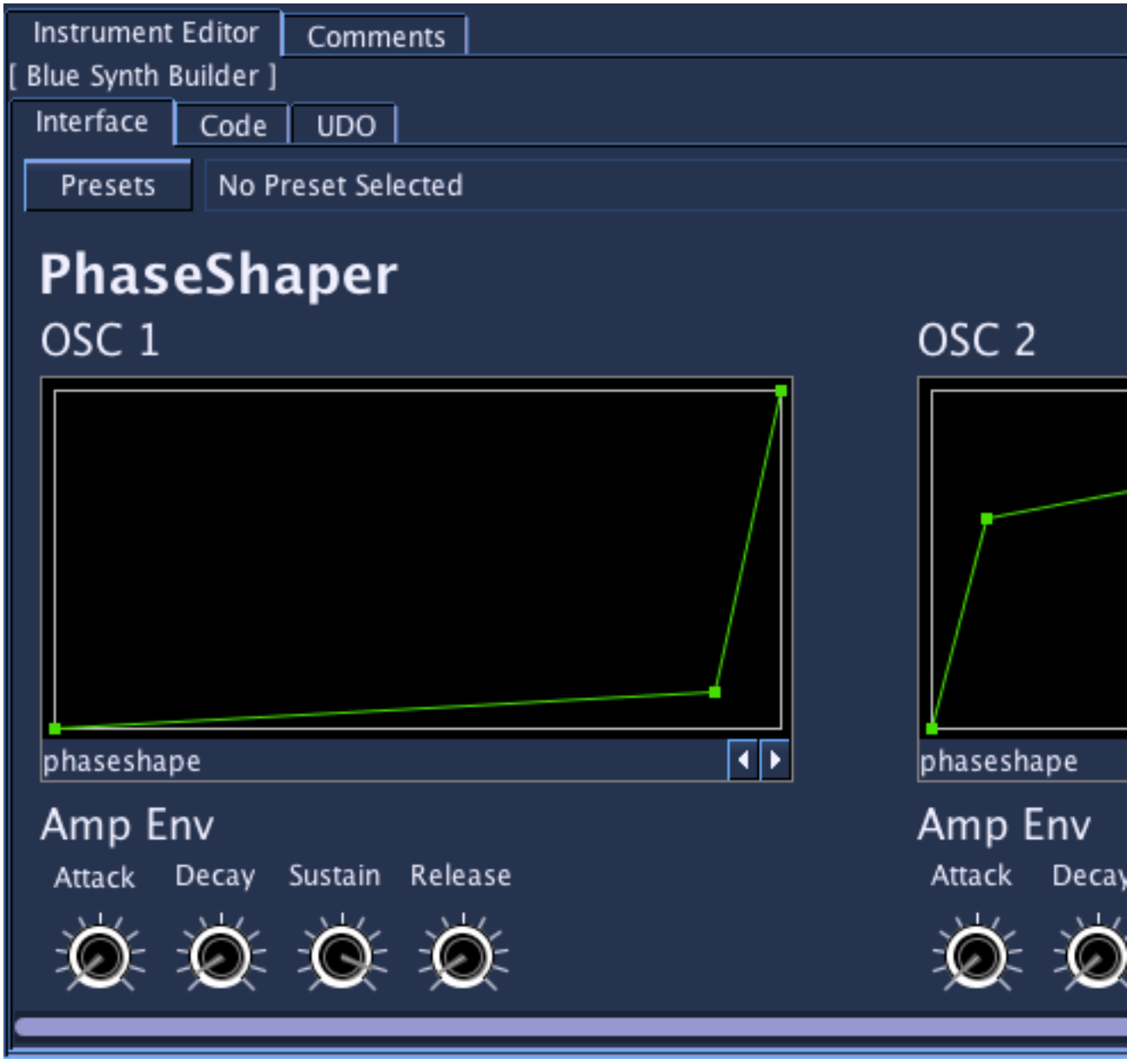
When in edit mode, you can modify the Grid Settings by using the properties editor on the right-hand side. This allows you set the following values:

Table 3.5. Grid Settings

Property	Description
Grid Style	Affects the visual style of the Grid. Options are NONE, DOT, or LINE.

Property	Description
Snap Enabled	Affects whether adding, pasting, and dragging of BSBObjects in the editor will snap to the grid.
Width	Sets the width of each grid box. Defaults to 15 pixels.
Height	Sets the height of each grid box. Defaults to 15 pixels.

Usage Mode



Once in usage mode, users can configure the instrument by working with the UI widgets: rotating knobs, moving sliders, etc. The values of the different UI widgets will be reflected in the generated Csound instrument and will affect the values in realtime if using the Csound API.

Code Editor

The code editor is where the Csound instrument code is to be written. To use the values from the widgets, use the replacement key of the widget within < >'s, and that value will be replaced when the instrument is generated (the replacement key is usually the objectName of the object; see table below).

For example, if there's a knob with an objectName of *amplitude* and the value is set to 0.5, the following instrument code:

```
iamp    = <amplitude> * 0dbfs
aout    vco2 iamp, 440

        outs aout, aout
```

will generate the following instrument:

```
iamp    = 0.5 * 0dbfs

aout    vco2 iamp, 440

     outs aout, aout
```

For convenience, the standard code completion popup will auto-complete text for replacement keys. When editing code, type "<" then press ctrl-space. The standard code completion popup will show a list of all of the replacement keys that have been assigned to interface objects. Selecting a replacement key will insert that key into the code text area, already formatted within < and >.

Always-On Instrument Code

MIDI and acoustic instruments often break down into sound generation qualities that are per-note as well as per-instrument. Having always-on code that can be encapsulated with the instrument allows modeling things like body filters as well as adding effects like chorus/echo/reverb to the instrument itself. While one can always add always-on effects to a mixer channel, having the capability to add these features to an instrument can be useful as part of the instrument's design.

To use this, one writes code in teh Always-On code tab. The code from this tab will be used to generate an instrument that will be run after the primary instrument code, and will have a single instace of the generated instrument running for the duration of the instrument. The signal from the main instrument code should be written out using the blueMixerOut pseudo-opcode. If the always-on code is disabled, the audio will go from the instrument to the mixer. If the always-on code is enabled, the signal generated from the main instrument code will be intercepted and read by the always-on instrument. To facilitate this feature, there is Blue pseudo-opcode called blueMixerIn:

```
asig1 [, asig2...] blueMixerIn
```

The always-on code should then do processing code, then write out to blueMixerOut.

To note, BSB widget values work perfectly fine when used within always-on code.

Widget Values

The following lists what values the widgets will emit when generating instruments:

Table 3.6. Widget Values

Widget	Replacement Key	Value
Knob	objectName	float value from knob
Horizontal Slider	objectName	float value from slider
Horizontal Slider Bank	objectName_sliderNum (for each slider)	float value from slider
Vertical Slider	objectName	float value from slider
Vertical Slider Bank	objectName_sliderNum (for each slider)	float value from slider
Label	none	none
Checkbox	objectName	1 or 0, depending on if checked or not
Dropdown List	objectName	Generates index of selected item when item is made automatable, or uses value from user-assigned Dropdown List
SubChannel Dropdown List	objectName	Value from Dropdown List that is a named subchannel from the Blue Mixer
XY Controller	objectNameX, objectNameY	X and Y value
LineObject	objectName_lineName (for each line)	list of values from line (can be comma separated, with or without leading 0.0 X value, and x values can be generated either in absolute or relative terms)
Text Field	objectName	Value from text field (compile-time only).
File Selector	objectName	If stringChannelEnabled is selected, outputs a Csound string variable (S-var) that can be updated at runtime if API is enabled, if stringChannelEnabled is set to off, outputs as a string (without quotes) at compilation time
Value	objectName	Generates using its default value or values from automation. Widget is only visible during edit mode.

NOTES

- For the Label object, you are able to style the label by using HTML. To use this feature, when setting the text of the label, enter the HTML label within <html> tags, such as "<html>My Label</html>".

Groups

Widgets may be organized into Groups that provide titled border around the set of widgets. Double-clicking on a group will allow editing of the group's set of widgets. The Breadcrumb bar that appears when edit mode is enabled allows for navigating back up the hierarchy of groups to the root group for the interface.

Users may either start by creating an empty group, double-clicking, then editing the interface for the group. Optionally, they may select a number of existing widgets, right-click on one of the selected widgets, then choose "Make Group" to embed the selected widgets within a group.

Presets

Since 0.95.0, BlueSynthBuilder now has the capability to save and load presets. These presets are for usage-time and not design-time, and they save a snapshot of all of the values for the widgets. They do not save x/y coordinates or other configuration for the widget, only the value.

You can add presets and folders of presets using the presets menu in the upper left of the BSB editor. Each menu has an option for adding a folder or adding a preset to it. You can also manage presets by using the "Manage Presets" button. This will open up a dialog with a tree view of your presets, allowing you to rename the presets and folders, as well as reorganize by dragging and dropping. You can remove presets and folders here by right-clicking and selecting the remove option. Changes in the dialog are not committed until you press the save button, so if you close the window or cancel, you're old settings will be still in tact.

Automation

Blue supports automation of BSB Widget values for those which support automation. For these widgets, automation must be enabled before they allowed to be automated. When making them automatable, the place in Csound instrument code where the widget value will be added to the generated Csound code must be a place where a k-rate signal is allowed, otherwise the code will not compile and Csound will not run the project. This is required because when using the Csound API, the signal will need to be k-rate to allow for live modification of the value when rendering.

More information on paramenter automation can be found here.

Note

In 0.124.3, a change was made that breaks backwards compatibility. Previously, if a BSBObject was set to have "Automation Allowed" but was not itself automated, it would compile as a constant in the generated CSD. As of 0.124.3, if a widget is made to allow automation, the Csound code that uses the widget value must accept a k-rate signal, whether the API is used or not.

If you have a project that rendered fine before 0.124.3 but afterwards can not render due to problems with Csound complaining that "k-rate signals not allowed", then you will need to either set the widget to not allow automation or change the Csound code so that it will work with the generated k-rate signal.

Randomization

Since 0.117.0, users are able to randomize values for widgets in a BlueSynthBuilder instrument. To use, first choose which widgets are set to be randomized in edit mode, then in usage mode, right click on the panel in an area not covered by a widget, then select "Randomize" from the popup menu. The following widgets are cable of being randomized:

- Knob

- Horizontal Slider
- Vertical Slider
- Horizontal Slider Bank
- Vertical Slider Bank
- XY Controller
- Checkbox
- Dropdown List

Shortcuts

Table 3.7. General Shortcuts

Shortcuts	Description
ctrl-1	brings the score tab into focus
ctrl-2	brings the orchestra tab into focus
ctrl-3	brings the tables tab into focus
ctrl-4	brings the globals tab into focus
ctrl-5	brings the project properties tab into focus
ctrl-6	brings the soundfile tab into focus
alt-right	brings the next manager tab into focus
alt-left	brings the previous manager tab into focus
F9	start/stop a render (equivalent to pressing the render/stop button)
ctrl-g	generate a CSD file
ctrl-shift-g	generate a CSD to screen (for previewing)
ctrl-o	open a work file
ctrl-s	save work file (must use "Save as" from file menu if a new work file)
ctrl-w	close the current work file
alt-F4	close Blue

Table 3.8. Rendering

Shortcuts	Description
F9	Render project using project's real-time render options
shift-F9	Render to Disk and Play using project's disk render options and playing with Blue's builtin sound file player
ctrl-shift-F9	Render to Disk using project's disk render options

Table 3.9. Score Timeline

Shortcuts	Description
ctrl-c	copy selected soundObject(s)
ctrl-x	cut selected soundObject(s)
ctrl-d	duplicate selected soundObject(s) and place directly after originals
ctrl-r	the repeat selected SoundObjects by copying and placing one after the other n number of times where n is a number value entered by the user (user is prompted with a dialog to enter number of times to repeat)
ctrl-click	paste soundObject(s) from buffer where clicked
shift-click	on timeline, paste soundObjects from buffer as a PolyObject where clicked, if only one sound object is in the buffer it will be pasted as the type it is and not a PolyObject
shift-click	when selecting soundObjects, adds soundObject to selected if not currently selected and vice-versa
double-click	if selecting on timeline, select all soundObjects on layer where mouse clicked
ctrl-t	show quick time dialog
right click	pops up a popup menu for adding soundObjects and other tasks
left click + drag	creates a selection marquee for selecting multiple soundObjects (when Score mode)
alt-1	switch to Score mode
alt-2	switch to Single Line mode
alt-3	switch to Mult Line mode
left	nudge selected soundObjects one pixel to the left
right	nudge selected soundObject one pixel to the right
shift-left	nudge selected soundObjects ten pixels to the left
shift-right	nudge selected soundObjects ten pixels to the right
up	move selected soundObjects up one layer
down	move selected soundObjects down one layer
ctrl-left	decrease horizontal zoom
ctrl-right	incrase horizontal zoom

Table 3.10. Orchestra Manager

Shortcuts	Description
ctrl-left click	if on the column header of the instruments table, will enable/disable all instruments

Table 3.11. In a text box

Shortcuts	Description
right click	pops up a popup menu that contains entries for opcodes as well as user-defined entries from the code repository
ctrl-space	brings up a dialog that shows all possible opcode matches for the current word being typed in (code completion)
shift-F1	if cursor is within a word that is an opcode, attempts to look up manual entry for that opcode
shift-F2	if cursor is within a word that is an opcode, attempts to find a manual example CSD for that opcode and if found opens a dialog showing the example
ctrl-;	comment out line or selected lines (prepends ";" to every line)
ctrl-shift-;	uncomment out line or selected lines (removes ";" at beginning of every line if found)

Table 3.12. Editing GenericScore, PythonObject, JavaScriptObject

Shortcuts	Description
ctrl-T	when the text area is focused, ctrl-T runs test on the soundObject (same as pressing the test button)

Appendix A. Introduction to Csound

Currently, this section is not yet written.

For more information on Csound, please read Dr. Richard Boulanger's wonderful TOOTS for Csound, available at here [http://csounds.com/toots/].

Appendix B. Breaks in Backwards Compatibility

This section logs any breaks in backwards compatibility.

Table B.1. Compatibility Breaks

Version	Description
0.124.3	Previously, if a BSBObject was set to have "Automation Allowed" but was not itself automated, it would compile as a constant in the generated CSD. As of 0.124.3, if a widget is made to allow automation, the Csound code that uses the widget value must accept a k-rate signal, whether the API is used or not. If you have a project that rendered fine before 0.124.3 but afterwards can not render due to problems with Csound complaining that "k-rate signals not allowed", then you will need to either set the problematic widgets to not allow automation or change the Csound code so that it will work with the generated k-rate signal.

Appendix C. Glossary

Glossary

B

.blue directory

blue Home

blue PCH

Part II. Tutorials

Tutorials

Your First Project

Andrés Cabrera

2004

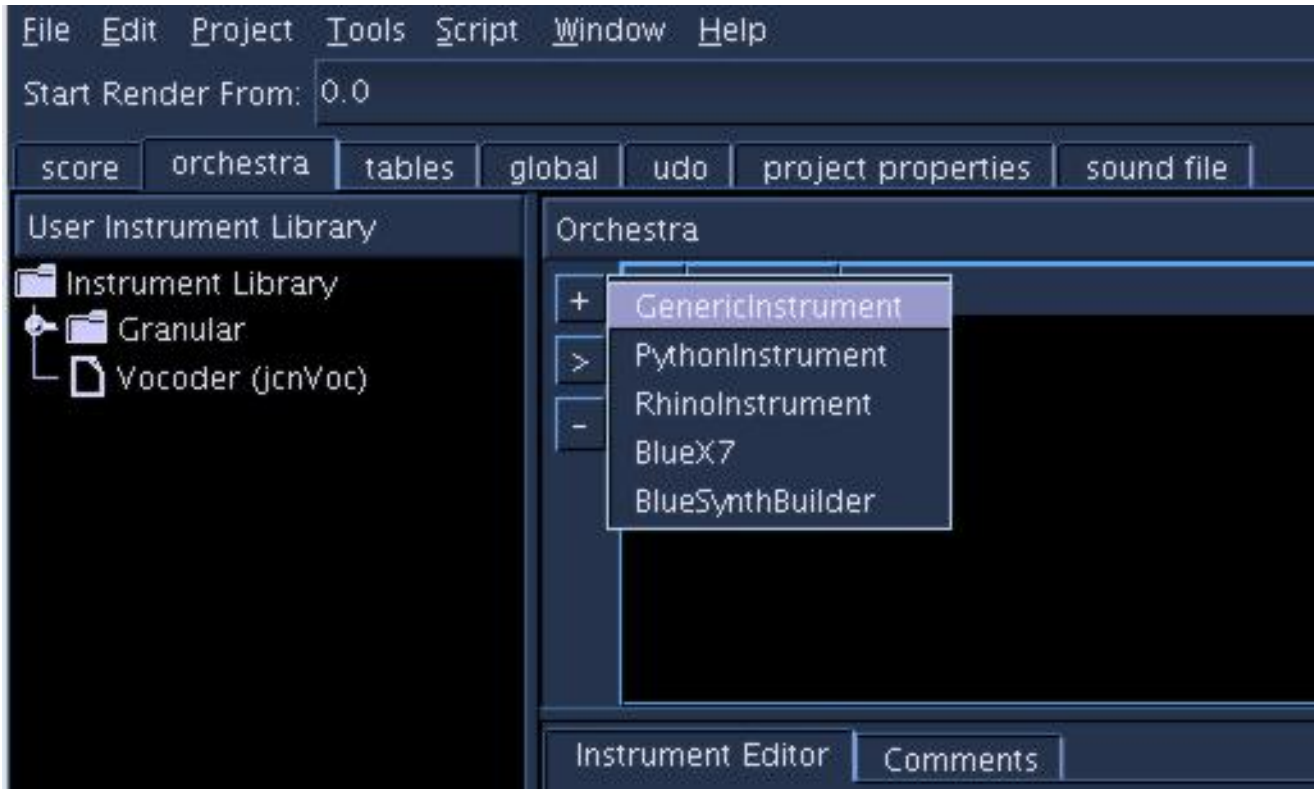
Revision History

Revision 1.1	2005.07.15
	Updated for new Instrument Library
Revision 1.0	2004.11.30
	First version of article.

This tutorial will explain how to start using blue. It assumes a little knowledge of csound, but not too much (If you don't know csound, understanding Dr. Boulanger's toots at http://www.csounds.com/toots/index.html should be enough to get you through this tutorial).

Run blue using the appropriate script provided in the blue/bin directory. Use the .bat file for windows, the .sh for linux and the .command file for Mac OS X. If blue is not running check the section 'Installation' on the blue documentation.

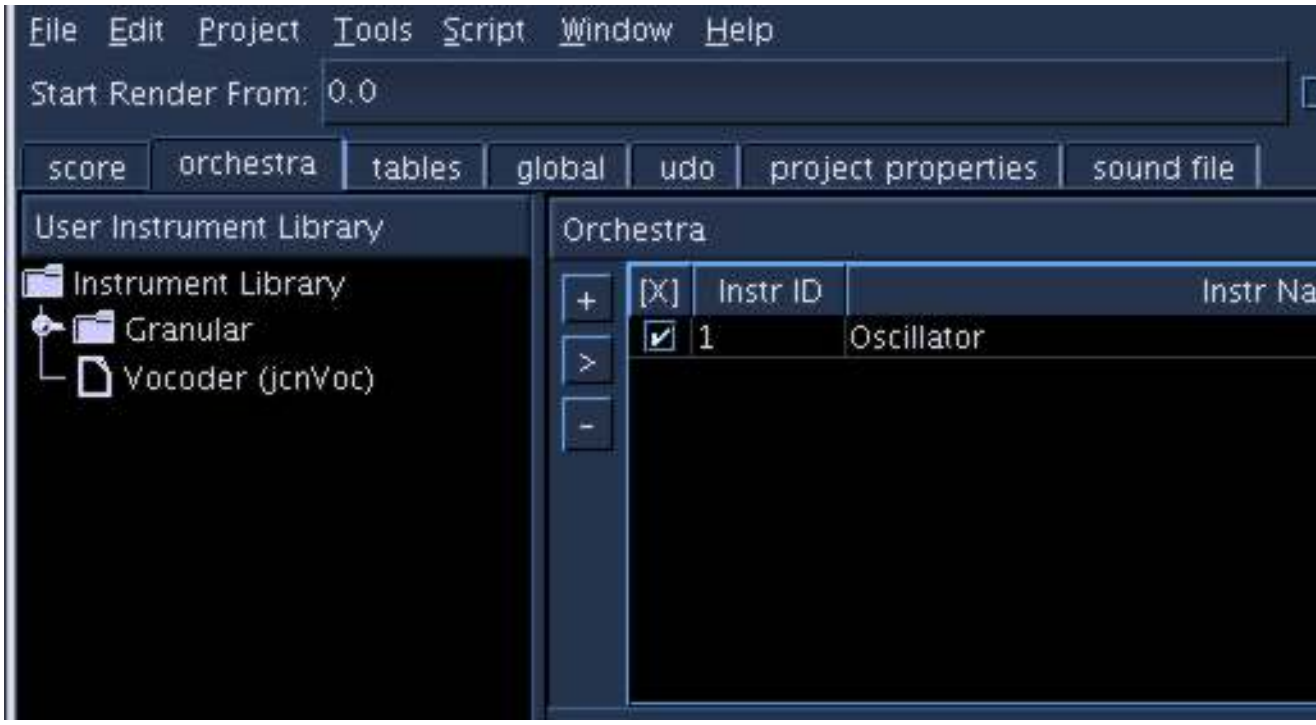
We will first create an instrument. Go to the orchestra tab, and click on the [+] button under the 'Orchestra' title. From the pop-menu select Add Instrument>GenericInstrument.



Instrument Library

You will want to choose GenericInstrument when you want to create an ordinary csound instrument. An empty 'untitled' instrument iFs created. To rename an instrument, select it, press F2 and type the new name (e.g. 'Oscillator').

The instrument library on the left panel is where instruments are saved for use in any blue project. To place an instrument in the library so it is always available, drag it to the Instrument Library folder.



Assigning Instrument to Arrangement

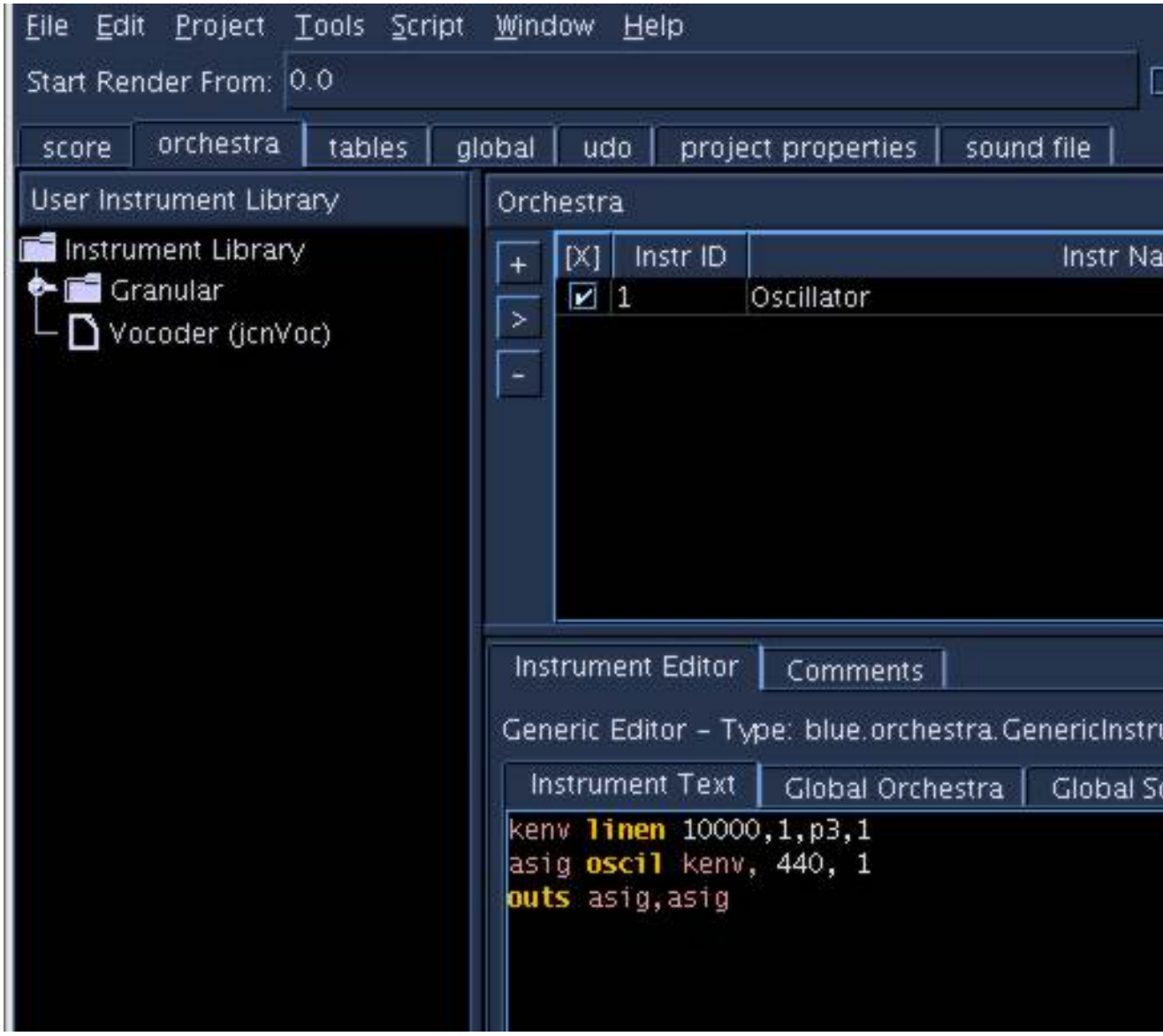
The '>' button brings an instrument from the library to the project. To remove an instrument from the project, select it and press '-'.

It is important to understand that instruments which are in the Instrument Library but not the Arrangement are not available for use in the csd because they haven't been assigned an instrument id (number or string, if used as a Csound named instrument).

Now that the instrument is there, we need to define how it will sound. We use ordinary csound code to do this (since it is a GenericInstrument). The code for the instrument is written on the lower right panel, inside the Instrument Text tab within Instrument Editor tab. Make sure you type the code in this tab, and not the 'Global Orchestra' or 'Global Score' tabs.

We will create a simple oscillator with an amplitude envelope like this:

```
kenv linen 10000,1,p3,1
asig oscil kenv, 440, gitab
outs asig,asig
```



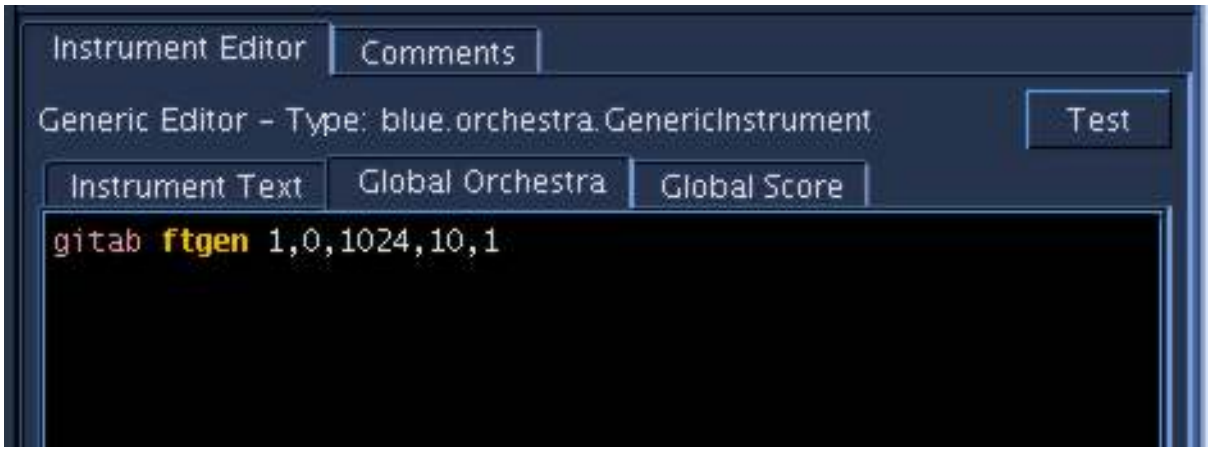
Editing Instrument

Try using blue's autocompletion feature like this: type 'line' and press ctrl-Space Bar. A pop-up window will display all known opcodes that begin with line. Once you select 'linen' you will have a template to fill in the apropriate values.

If you need to remember the syntax for an opcode, place the cursor over it and press Shift-F1. If you have set correctly the Csound Documentation Directory, the html help for the opcode will be displayed if available. See the section 'Installing blue' on the main blue documentation (F1) for details on setting up the C.D.R.

To define the oscillator table we will use ftgen, and place the following code in the Global Orc tab for the instrument:

```
gitab ftgen 1,0,1024,10,1
```



Adding an FTable for the Instrument

Placing things like tables in the global tabs inside the instruments makes it very simple to share instruments between projects, because the required initialization and tables travel with the instrument.

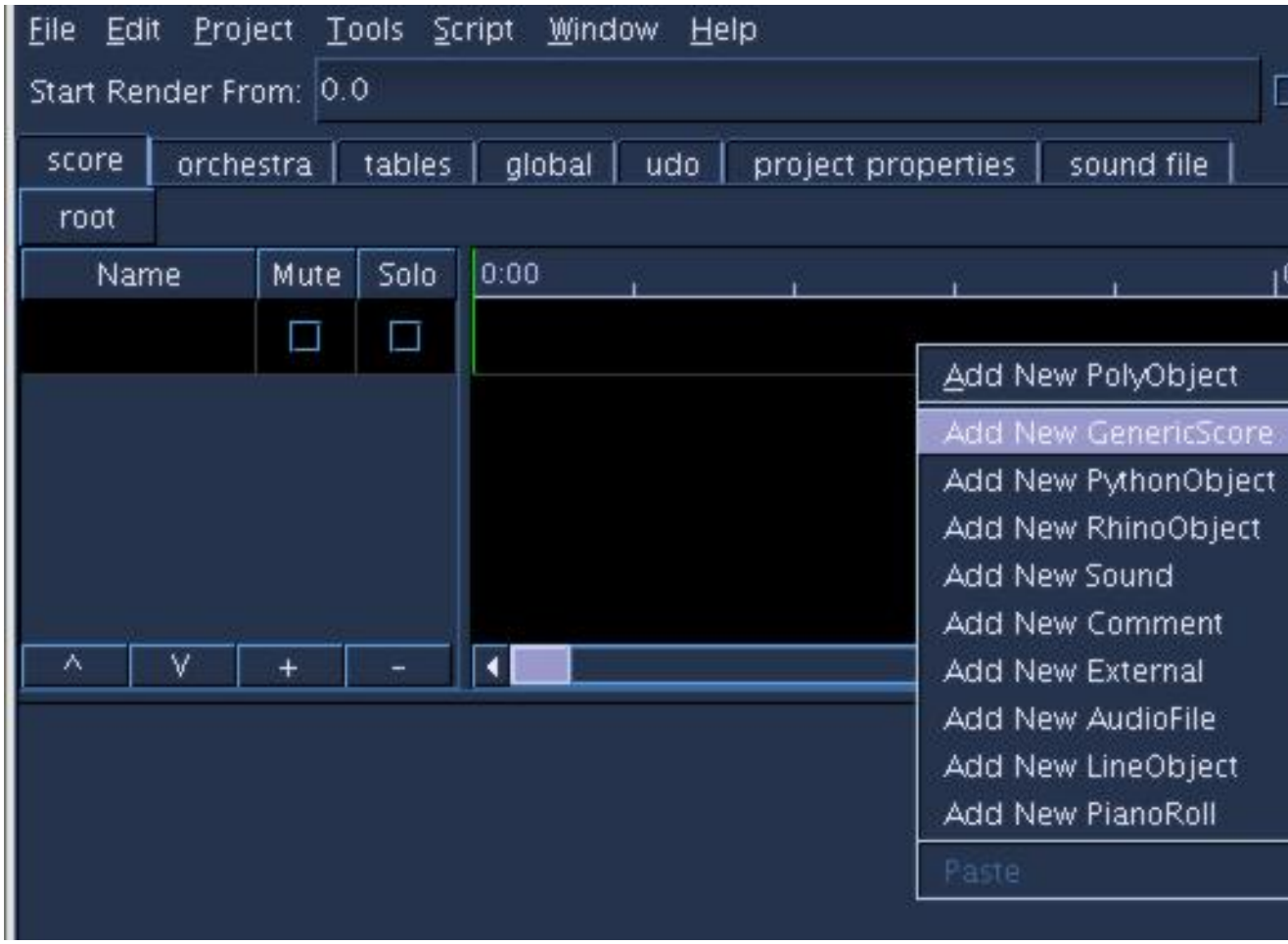
You can preview the csd file to be generated by using Ctrl-Shift-G or Project>Generate CSD to screen. See how the ftgen opcode that was placed in the Global Orc tab is placed above the instrument definition.

Now that our instrument is ready we will create notes for it. This is done by going the Score tab (You can use ctrl-1).

The score tab contains horizontal 'tracks' called Sound Layers. On sound layers you place objects which generate notes that instance the instruments.

First we will give a name to our sound layer. Simply click on the black area below 'Name' and type the name (e.g. 'Notes').

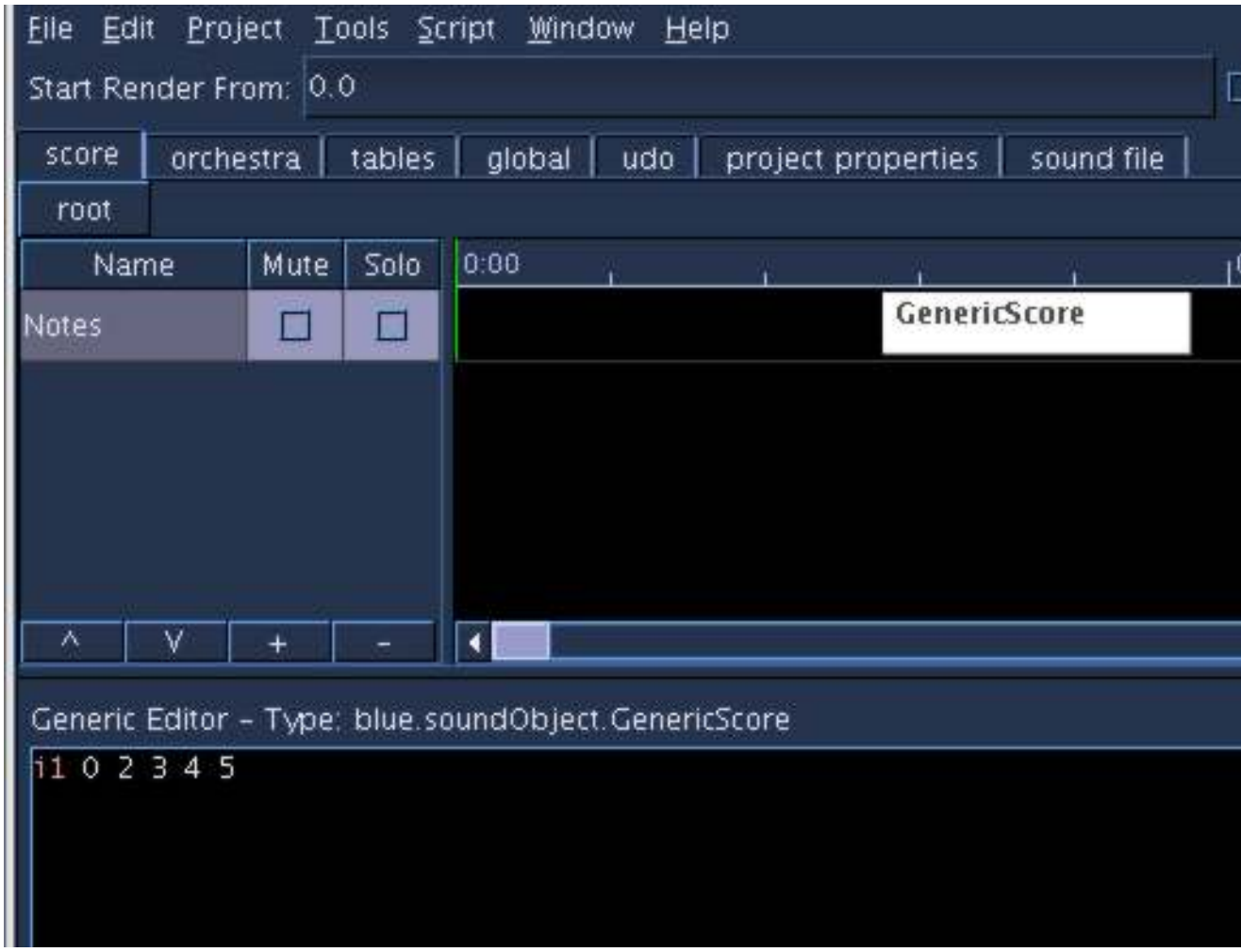
To create a note, go to the timeline on the right, and in line with the sound layer, right-click, and select 'Add New GenericScore'.



Adding a Generic Score

GenericScore allows ordinary csound code to be used, in this case things like 'i' statements. When you click on the genericScore that has been created, the bottom panel will show the contents of the object. You can add as many 'i' statements as you need inside a genericScore Object. We want our score to generate a note for instrument 1, our 'oscillator' instrument. We'll use the simple:

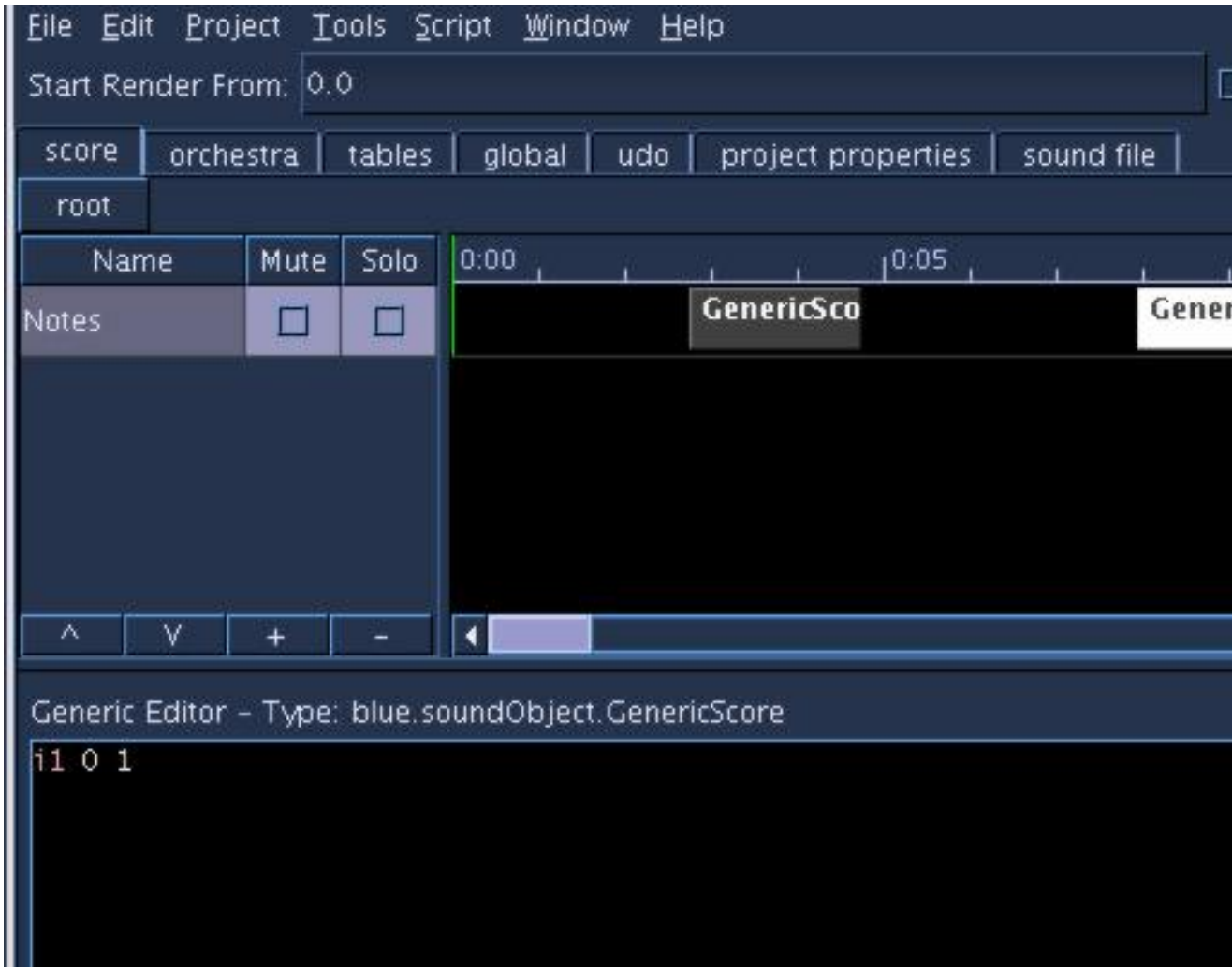
```
i 1 0 1
```



Editing a Generic Score

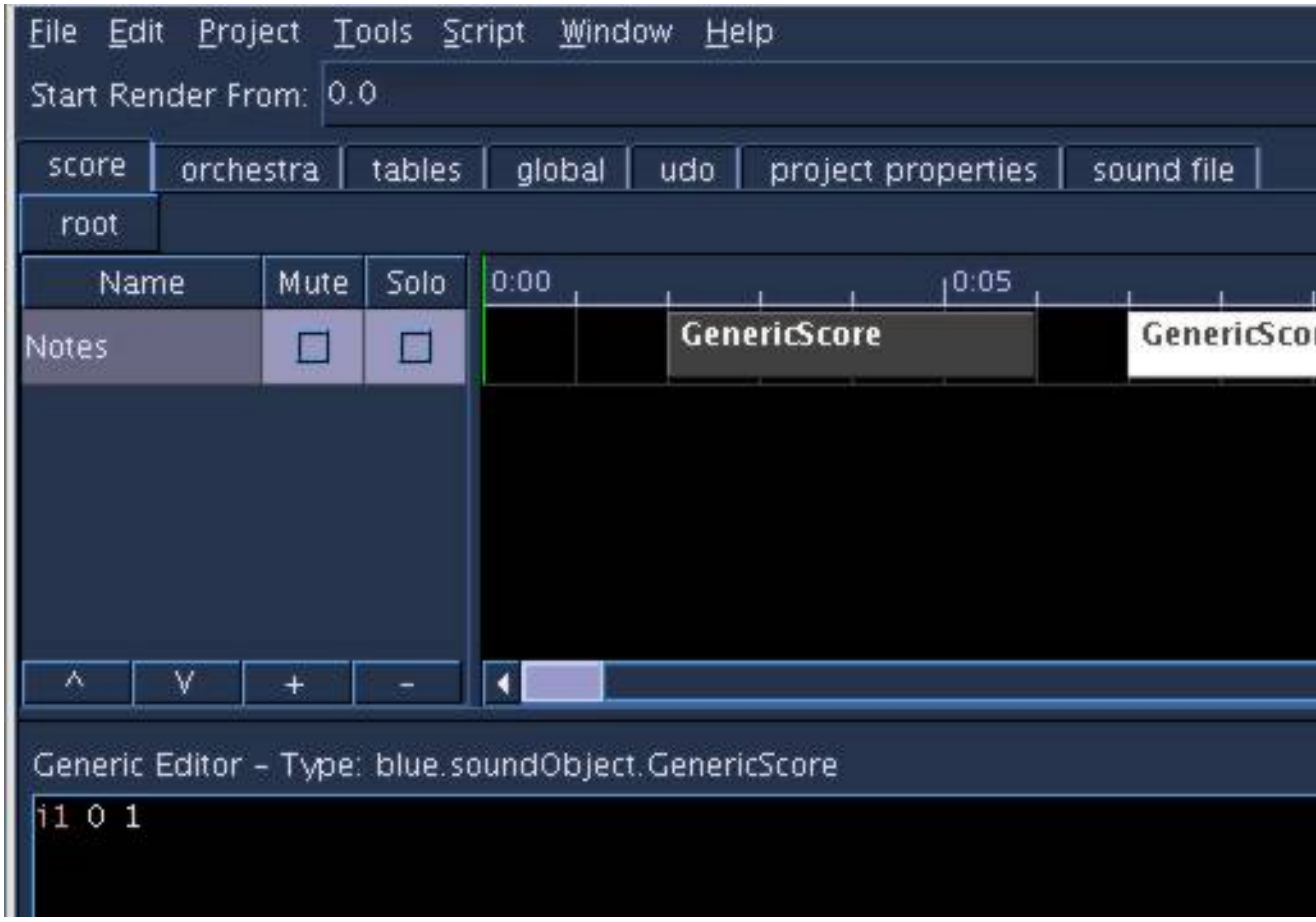
You can preview how the object will generate the csound score by pressing the [t e s t] button on the right side of the panel. Notice that the start time is the place where the object was placed on the timeline, but the duration of the note is 2 seconds instead of the 1 second we typed. By default blue creates genericScore objects set to 'scale' the notes inside it. Try dragging the right corner of the object to make it longer, and try the [test] button again. The duration of the note now scales to the size of the object. Don't forget that when multiple notes are inside an object, it is the note with the longest duration that scales to the size of the object and the rest of the notes are scaled proportionally.

Now let's create a second note a few seconds later. First zoom out by pressing the '-' button on the right endof the horizontal scroll bar. Now select the genericScore we have created and press ctrl-C (or command-C on Mac), and right click where you want to paste the copy and select 'Paste'. You should now have two objects like in figure 7.



Copying SoundObjects

You can use the snap feature to move the objects by fixed values. press the [...] button on the upper right corner of the timeline to open the snap properties. Enable snap to 1 second (Fig. 8). You can also configure the timeline time display to show numbers instead of time. This is useful if you set the tempo using a 't' statement, and want blue to display numbers instead of time values.



Setting Timeline Options

Now move and shape the objects so the first starts at 0, with a duration of 5 seconds, and the second starts at 10, again with a duration of 5. It's always good to check using [test] to make sure things are how you want them to be.

Now to produce sound, we will first check the global settings. Go to the 'project properties' tab by pressing ctrl-6. Here you can set the project's sr, kr and nchnls, and determine the csound executable and flags.

The main thing to set here is the command line to call csound. The command line depends entirely on your setup and operating system. In Linux, csound is usually located in an executable directory, so just using 'csound' works for most setups. In Windows, you may need to specify the full path for the location of the csound executable. Note that after the executable name you can add flags to determine how csound will process the csd generated by blue. For example, you may want to use a real-time flag for this project, like -o dac0 or -+P depending on your system.

An important thing to note here is that when using the play button, the flags used are the ones inside te command line. The <CsOptions> below only affect the generation of external csd's using the 'Generate CSD to screen/file' commands from the menu.

When you are ready to compile your csd, press the play button on the upper right corner or press F9. If you set the output for realtime, you should hear the tones produced by the oscillator.

Scripting

Steven Yi

2002

Revision History

Revision 1.1 2004.11.7

Converted to DocBook. Some information revised as no longer applicable

Revision 1.0 2002.10.28

First version of article.

Introduction

The blue music composition environment entails the use of timelines on which SoundObjects are placed. these SoundObjects are basically lists of notes, grouped as a single perceived object. These can be as simple as a single note, or a group of notes one might handle as a "motive", the "bassline for the A phrase", "the results of an predetermined matrix of sounds and silences through 28 flippings of coins", or "the stochastic group motion of migrating birds mapped to 17 instruments that produce variable timbres", etc. but underneath it all, regardless of what SoundObject you use, ultimately the SoundObject will produce notes in which blue will add to the running score and output to a .CSD file.

Seen this way, blue represents a sort of visual note-generator scheduling system. when blue was earlier on in it's development, I realized that in creating blue, I created a simple generic temporal framwork which would allow people to create score generators and only worry about generating those scores, while easily adding them to the framework of blue which would allow the manipulation and scheduling of those score generators without the original authors having to worry about developing that aspect of it. as my imagination began to wander, I saw that things like cmask, mother, cybil, and other scoring languages that have been built over time for csound could be used in the way they've always been used, but that they could easily be integrated into blue.

for myself, the SoundObjects I've built have offered me the ability to separate my concerns: as the author of blue, I concern myself with finding new ways to organize and manipulate SoundObjects, and as a SoundObject author, I concern myself just with making objects that generate notes, leaving the scheduling of that SoundObject to the framework.

But, not everyone programs in Java, and there are also scripts and script libraries to be used in other languages already made.

Having seen that the SoundObject system and temporal framework of blue was a pretty open architecture, I decided at the time to explore the possibilities of using script within blue by creating the Python SoundObject so that i could use maurizio umberto puxeddu's Python pmask library within blue. what that allowed was for the ability to write Python scripts within blue, blue executing and bringing back in the generated notes from the Python script and integrating that with the other notes of the native Java SoundObjects.

After successfully developing and proving the possibility to myself, I continued on to build other SoundObjects as user's requested or as I desired for my own compositional explorations.

One day, some time back, a user suggested that I build a generic external SoundObject that would let you write any script within blue, provide a commandline to be used, and have blue write the script to a temporary file, run the commandline on it, and bring the generated notes back into blue. busy at the time, I later did build that SoundObject, and it opened up a world of possibilities.

With the new external SoundObject, it became possible to write script in any language within blue. this means one could use not only scripting languages like perl, Python, shell scripts, batch files, JavaScript, wscript, etc., but also run specially created text input for score generators like CMask, nGen, etc. (note: cmask does not currently work within blue, as it is unable to print to stdout; more on this and possible ways around it later). It also meant that other people's work on score generating libraries and scripts, regardless of language, now had the ability to be reused within blue.

As blue has developed the past couple of years, from its initial goals of embedding timelines within timelines, to its current state as a generic temporal framework and all-around composition environment, I've always developed with the mindset of keeping this as open and generic as possible, to build a tool that could grow as I knew I'd always be exploring new things in composition. These days, the inclusion of these two SoundObjects, the external SoundObject and the Python SoundObject, are now becoming part of my everyday use as I work with blue, allowing me to explore ideas rapidly, creating objects and functions which follow the models of my musical work, and which also serve as possible prototypes to new SoundObjects which I may later decide to create in Java and add to blue.

From my own experience, I'd highly recommend to anyone that they learn a little bit about scripting and how to use it to achieve their musical goals, as I've found the experience highly liberating and musically rewarding .

Why Use Scripting?

For some, the use of scripting or programming may never be a factor in how they go about creating their music, as the tools they have already are enough. But for some, it may be easier to go about expressing their needs musically by use of script. and for others, there just might not be anything available to do what one wants to do.

Within blue, there are many ways to go about expressing and designing your musical work. Included with blue are the built in SoundObjects which can take in standard csound scores. Together with the use of the noteProcessors in blue, already there are the tools to write standard csound scores, move them in time, and manipulate those scores by adding or multiplying pfields by user given values(to achieve things like scaling amplitudes or transposing notes) or retrograde notes.

However, lets say you might find scripting in your prefered language to be easier to achieve your goals, or you want to use a scoring language other than csound's standard score format, or someone has advertised a new set of scripts or libraries to use. you could do all of this within blue.

In my case, I was recently modeling an object system that represented a musical model that hasn't been used much in current electronic music programs. I enjoy using the blue environment, but this was a case where I needed to do some work by programming, and the existing tools didn't exist in blue. The process was pretty experimental and the use of compiled languages like Java would have been slower than using a scripting language, as the prototypes for the object model I was building were changing rapidly. What ended up happening is that I worked in Python within blue using the Python SoundObject, developing my core objects, and was then able to rapidly experiment with composing with these objects by using other Python SoundObjects which used my core class definitions, moving them around visually and listening to how the musical output. I could have done all of this outside of blue, strictly using script, but the visual element and the ease of quickly copying blocks of Python script and moving them around, made experimentation much quicker and more apparent to me. A great benefit out of all of this is now I have a prototype which I can then perhaps use to program SoundObjects for blue in Java, maybe adding a GUI to the object to control parameters, making the process of using the SoundObjects that much easier.

The total process of prototyping and experimenting was a great experience, as I felt that each part of the process functioned to help me best express what I was going after. I used script to quickly create the objects I wanted to use to model the musical model I had in mind, and then used blue to experiment with those scripts to hear the sounds they could produce.

Finally, it lead to strong candidates for objects which I could go on to build in a more generic fashion for everyone to else to use in blue, adding to the compositional possibilities.

In the past, I've also built simple functions to do a very specific task, like "add a value to all p4's of all notes". Something like this could be the case where you have a simple task that might be monotonous and you could probably achieve your goal much more quickly by a little script. (For example, I want to generate a test file for my orchestra and all it has are notes lasting two seconds duration, whose start times are every three seconds; something like this could be done in maybe 6-10 lines of script, regardless of the size of the orchestra.)

Those are some of my reasons for the use of script for musical work, and I'm sure others have their own reasons. In the end, it's up to you to evaluate your situation and see if it will aid in your work or not.

Scripting and blue

Every once in a while someone releases a new program for score generation, a utility they've decided to make for their own benefit that they've graciously shared with the community. Often, these musical tools are exclusive in their use, meaning they're good for their use, but are difficult to integrate with other work.

For example: say you're working on a piece and want to use an external score generator to generate a score to add to your piece. now, in experimenting, you might have to:

1. Create your csound score
2. Create your input file for the external score generator in another file
3. Run the score generator
4. Copy the generated score into your csound score file
5. Repeat process as you experiment

Now imagine that process to the nth degree if you decide to use multiple external programs!

However, with blue, you can do all of that in one environment. You could write your standard csound scores as well as your input to your chosen score generator within blue, press the play button and then have blue do the work of going out to compile the score generator material and bring it back in, integrate it with your csound score, and output a single unified score file. You get the benefit of using the visual timeline to move your generated score blocks around in time, as well as only having to concentrate on the score input to the score generator, leaving all of the tedious work to blue to handle.

Or, say you're working on a score generating library in a scripting language. Now that you've finished it, you want to start working with it. but say you want to also do standard csound scores along with your scripts; having to do the two in the single scripting environment might not be optimal, as you would constantly have to use "print()" functions or use note objects, when it might just be easier to write "i1 0 1 2 3 4 5" and work with that for some of your work, and use your libraries for others. In this case, blue can handle that in that you can call your libraries you've developed and script the parts you want to script within blue, as well handle all of the standard csound score type work.

Within blue there are currently two main objects for scripting, the Python SoundObject and the external SoundObject. The external SoundObject runs scripts outside of blue and brings the results back into blue, and the execution of those scripts is as fast as it would be running the script outside of blue. With the external SoundObject, one may also choose from a variety of scripting and score generating languages to use. The Python SoundObject is limited to Python, and runs Python scripts much slower than they would be for running a Python script outside of blue using the external SoundObject. However, the PythonObject has

two main features which may influence your decision in which to use it or not: it does not require Python to be installed on the system as it uses Jython to interpret the script (Jython is an all-Java implmentation of the Python interpreter; for more information about Jython, visit http://www.jython.org), and it does not reinitialize the interpreter between uses.

The second point regarding the Python SoundObject may need some explanation: by not reinitializing the interpreter between uses, all Python SoundObjects in blue all share the same interpreter. By doing this, you can define variables, functions, and objects in one block, while using those variables in another block.

As an example, in my recent work with the Python SoundObject, I've set up the first PythonObject in my blue work file to have nothing but class definitions as well as function definitions. The second PythonObject block only has script that sets up instances of those classes written in the first PythonObject, and in this second block I did all of my initialization work for these classes. Finally, throughout the rest of the blue work file, I had PythonObjects that used the class objects I set up in the second PythonObject. This would not be possible using the external SoundObject, as once a script is run, all existing information in memory for that script is lost (unless you decided to make some kind of serialization scheme to write data to a file to be accessed by other scripts, or perhaps were using a web server to maintain state data).

The next two sections below will explain how to use the two different SoundObjects, and for the Python programmer, help you determine when you might want to use one object verse the other.

External SoundObject

The external SoundObject is a generic object that allows you to write script within blue to be executed outside of blue and have the generated score be brought back in. The way this is implemented, blue will read in anything that the external program writes to stdout. for most scripting languages, this equates to a "print()" command of some sort in your script.

The external SoundObject's editor takes in two inputs, a script and a commandline to run on that script. technically, when blue is in the score generating pass and comes to an external SoundObject, the external SoundObject writes your script to a temp file, then runs the commandline given, either swapping “$infile” with the name of the temporary file generated, or appending the name of the temporary file to the end of the commandline if no “$infile” is found.

A Simple Example

Let's say you're writing a simple perl script:

```
print "i1 0 2 3 4 5\n"
```

and you for your commandline you use:

```
perl
```

or perhaps:

```
/usr/bin/perl $infile
```

When blue goes to generate a .CSD file and comes across your external SoundObject, what will happen is that:

1. Your script will get written to a temp file (for this example, let's say it's "/tmp/temp4253.txt")

2. blue executes the commandline given with that temp file.

a. For the first commandline, it'll then run:

```
perl /tmp/temp4253.txt
```

b. For the second commandline, it'll then run:

```
/usr/bin/perl /tmp/temp4253.txt
```

3. perl runs, and will print "i1 0 2 3 4 5\n" to stdout

4. blue will get that output from perl, then bring that back in, convert it to blue Note class.

5. blue then will shift the note over to the start time of your external SoundObject, then scale the note to the duration of the SoundObject.

And that's it!

Note. In general, all time values for start are from time-zero within the SoundObject. This is like taking a three notes: you know one starts at the beginning, one starts 1 second later, and one starts 2 seconds later. No matter where you put those three notes, the relationship of that block of notes is the same, one at the beginning, one 2 seconds later, etc. so it is with SoundObjects in blue. When you make a script to generate notes, have it start at time zero and let blue do the scaling and translation of time for the SoundObject. blue was made this way so that no matter what you generate within the block, the generated notes will start at the start of the SoundObject and will last the duration of the SoundObject.

A More Complex Example

Now, any script that has a commandline that can execute on a file will work with blue. so, as another example, let's say you want to use Python and you're using the pmask library. You have both of these installed on your system and you know they work fine because you've tested them outside of blue. Now, let's say you wrote the following Python script (which really, was written by Hans Mikelson as a an example for using pmask, taken from the CSound Magazine):

```
from pmask import *


    density = Mask(UniformRandom(), PowerSegment([(0.0,
 0.03),  (20.0, 0.5)], 3.0), PowerSegment([(0.0, 0.08), (20.0, 1.0)], 3.0))
    duration = Mask(UniformRandom(), PowerSegment([(0.0,
 0.4), (20.0, 3.0)], 1.0), PowerSegment([(0.0, 0.8), (20.0, 5.0)], 1.0))
    frequency_mask = Mask(UniformRandom(), PowerSegment([(0.0,
  3000.0), (20.0, 90.0)], 1.0), PowerSegment([(0.0, 5000.0), (20.0, 150.0)],
  1.0))
    frequency = Quantizer(frequency_mask, LinearSegment([(0.0,
  400.0), (20.0, 50.0)]), 0.95)

    index = Mask(UniformRandom(), PowerSegment([(0.0,
2.0),   (20.0, 3.0)], 1.0), PowerSegment([(0.0, 3.0), (20.0, 5.0)], 1.0))
    panorama = Range(0, 1)

    amplitude = Lorenz('x')

    ss = ScoreSection(0.0, 20.0, 1, density, duration,
frequency,  index, panorama, amplitude)
```

```
print str(ss)
```

and you tested it outside of blue and it worked fine. Well, now you could create an external SoundObject in blue, paste in this script, set the commandline to "python", and now have the notes imported into blue.

Using a Score Generating Language

Let's try a non-scripting example now and use Mark Williamson's drum pattern scoring utility to generate a drum pattern(available at http://www.junklight.com). Mark's utility is itself a perl script, but it takes in a text file using a scoring language he developed.

copying in the example drum.pat file into the scripting area:

```
instrument1 "i1 $now .2 2000 2.2 0"
instrument2 "i1 $now .2 2000 2.4 1"
instrument3 "i1 $now .1 2000 4 1"
instrument4 "i1 $now .2 2000 1.6 0.5"

instrument5 "i1 $now .1 2000 4 0"
pattern1 1 1 1
pattern2 1 1

pattern3 1 1 1 1
pattern4 1 1 1 1 1 1 1 1
pattern5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
plist1 1 2
ilist1 3 5

bar length 1 second
time error .01
# poly rhythm example
loop 5

 play 1 with 1
 play 2 with 2

endloop
# straight 4/4

loop 5
 play 2 with 1
 play 3 with 3


endloop

loop 5
 play plist 1 with 4

 play 4 with ilist 1
```

```
 play 3 with 1
endloop


loop 5
 play plist 1 with 4
 play 5 with ilist 1

 play 4 with 1

endloop
```

Having unzipped the drumscript.zip into my home direcotory, I used the following commandline:

```
perl /home/steven/drumText/drums.pl -input
```

I ran the test button and got... nothing! I got alot of exceptions printed to the console. Running the drum script outside of blue, I saw that by default, some logging information was being printed (the name of the infile and input). Modifying mark's perl script a bit to comment out those two print statements, I reran the script within blue and... success! The generated output from mark's script successfully was rebrought back into blue.

Notes on Score Generating Programs and the External SoundObject

Score generation programs can either print out to stdout or write out to a file. If writing out to a file, care must be taken to use the $outfile parameter of the External SoundObject so that after the external program finishes, blue will know to grab the file's contents back into blue.

Python SoundObject

The Python SoundObject is a Python-only scripting SoundObject which uses the Jython Python interpreter (more information about this all-Java Python interpreter is available at http://www.jython.org). Jython is included with blue, and being so, it means that if you are using the Python SoundObject in blue, you *do not need Python installed on your computer*. The Python SoundObject is made in such a way that it in no way uses any Python installations you may or may not have on your computer, but instead only uses what is a part of Jython. If you want to add extra packages and libraries to use with the Python SoundObject, you may copy them into the lib/PythonLib directory or in the userhome/.blue/pythonLib folder. As an example of adding your own library, you can see that already included with blue in the lib/PythonLib directory is the pmask library by Maurizio Umberto Puxeddu (currently, I do not know where this library exists anymore on the internet; if you know where it can be downloaded from, please let me know and I will update this part of the documentation to better give credit to Maurizio).

So, to clarify, having a Python interpreter installed on your computer does not in any way affect the Python SoundObject within blue; they are two completely different entities. also, if you have a library installed for use with your Python interpreter installed on your computer, it will *NOT* be accessible to the Python SoundObject. to use it, you will have to copy it into the lib/PythonLib directory of blue.

Note: what you could do, also, is put the individual libraries you install in the lib/PythonLib directory as part of your PythonPATH environment variable, such that all of the libraries will be both accessible to your Python interpreter on your computer as well as the one in blue. It is not recommended to put the entire PythonLib directory as part of your PythonPath, however, as the PythonLib has many of the same files

from the standard Python installation and may cause issues with locating standard library files. However, I find that just copying the library twice into both the PythonLib and my standard Python install's directory to be sufficient and easier to manage.

Using Python with the External SoundObject versus the Python SoundObject

This has been the most often asked question since the addition of the external SoundObject blue, and when I initially put in the external SoundObject, I thought that the Python SoundObject would become somewhat unnecessary.

But a week or so after the addition the of externalObject, I started using Python alot more in conjunction with blue and found that the Python SoundObject has a few key benefits which make me use it now everyday.

The Case for Using the External SoundObject

First of all, performance of the PythonObject is not as fast as using the standard Python interpreter with the external SoundObject. For simple scripts, the difference in speed isn't that noticeable, but when using something like pmask which has alot of calculations when generating score output, the speed difference does become an issue. If speed is an issue, then you might choose to use the external SoundObject, if the benefits listed below aren't an issue.

Also, if you have an existing Python interpreter installed on your computer and have libraries installed and are just simply used to your setup, then it might be just plain easier to use your existing Python setup rather than dealing with where things are in blue and managing having libraries installed for both the resident Python interpreter and blue's PythonLib.

And to add to all of that, you may be using features which are part of the most recent Python interpeter version, and those features may not be available as part of Jython (more information as to what Jython supports is available at the Jython homepage, http://www.jython.org). Also, if you're using libraries which call native code, they will not work with Jython.

So, if the Python SoundObject is slower, if you have to manage another set of libraries, and you might not have all of the features of standard Python available, why use the Python SoundObject at all?

The Case for Using the Python SoundObject

Now, just to let you know, I use exclusively the Python SoundObject when I'm doing Python scripts in blue. I haven't come across any of the standard libraries I've needed that were't included with Jython, and if it's a sign of anything, Maurizio Umberto Puxeddu's pmask library works fine with the PythonObject.

One of the things I used to do is keep an installation of blue on a zip disk, including my work. The nice thing about this was that I was able to take this disk and work off it while at work on a windows machine, as well as take it home everyday to my linux machine and start up blue without having to do anything different. Because I used the PythonObject at that time, I didn't have to worry at all about anything installed within either system, as all of it was contained within blue. It's also a reassuring thought that if I took my work to a friend's computer, be it a macintosh, windows, or linux pc, I wouldn't have to worry about if Python was installed on their computer.

Besides these very practical issues, the way you can go about using the PythonObject differs very much from the usage of Python with the externalObject.

With the externalObject, each script run is its own entity, run with a different interpreter. This means that memory is not shared between instances of the externalObject: variables and results of execution

in one externalObject are not accessible to another externalObject. However, with the PythonObject, the interpreter is not reinitialized between runs, meaning you *can* share results between one PythonObject and the next. This can of course lead to very convoluted methods of developing scripts within blue, which is of course not recommended, but it does have some very pragmatic possibilities which are very useful.

In my latest work with blue, I've been using almost exclusively the PythonObject for all of my SoundObjects in blue. I did use some sound SoundObjects initially to draft some sounds out, converting them to instruments in the orchestra manager later. I also used genericScore objects for writing some test notes for my instruments which I keep around also for experimenting, but for the piece's final output, they are not being used.

For setting up my piece, I developed some classes to use in Python, instantiated multiple instances of the classes set with different parameters, and then use those instances in different PythonObjects in blue for creating the score of my piece. Because the interpreter is not being reinitialized between uses by the PythonObject, I'm able to use one PythonObject to hold only definitions of functions and classes, another PythonObject which only has my initialization scripts that makes instances of the classes defined in the first PythonObject, and from all other PythonObjects, I use the instances from the second PythonObject as my generators.

to better clarify, my first PythonObject has scripts like:

```
class Yo():
    def do(self, a,b):
        print "doing something with " + str(a) + " : " + str(b)

    def yoyo(self, a,b):
        print "do something with a and b which are instances of Yo()"
```

while my second PythonObject has script like:

```
generator1 = Yo()
generator2 = Yo()
generator3 = Yo()
```

while all other PythonObjects in my piece have a minimal amount of script, something like:

```
generator1.generateNotes(input1, input2, input3)
```

I find that it's a nice way to separate the concerns of each object. as I'm currently in the middle of working on the piece, I'm finding that first PythonObject getting a bit big, so I'm thinking i'll split that into different PythonObjects, which I can visually see labeled on the timeline by changing their names to something like "generator classes", "generator utility functions", "constants", etc. and probably in the future, as I get to finishing this piece, I'll probably take all of these class definitions and make a library of Python code out of it and just have it installed into my PythonLib directory. But having the way it is now I've found to be very convenient in development as it's easy to find all of my code.

Note: the execution order of blue is top-down per soundLayer, meaning it will process all SoundObjects in the first soundLayer, and if it finds a polyObject, it will go down into that and process the first soundLayer, etc. before going on to the next SoundObject. because of the execution order, I can put all of my PythonObjects that have nothing to do with note generation (my class definitions, utility functions, and intitialization scripts) in the first few soundLayers of my piece and know that they'll always get interpreted by the jython interpreter and left in the memory space first before trying to process the other SoundObjects.

Note: as a warning, also because of the execution order, if you reopen your piece and try the [test] button on a PythonObject that uses code from the initial PythonObjects and find that it's not generating notes, it's because that code has yet to be entered into the memory space of the interpreter. either individually running all of the PythonObjects that deal with class definitions and initialization or just pressing the [play/stop] button to generate your score will get those objects interpreted and entered into the jython interpreter's memory.

Because of this alone, I find I use the Python SoundObject more than I do using the external SoundObject. if I was to use the external SoundObject and wanted work an even remotely similar manner, I'd have to come up with some strange hack to maybe write script in one external object have the execution of that object write that script into a file which would have to be reimported into other scripts. not a horrible hack, but enough for me to want to avoid, especially when it's not necessary. (though, if you're programming in a scripting language besides Python, you would hack to work in a manner like this...)

Usage

For the PythonObject, instead of using a print command to stdout to bring things back into blue, you simply assign the variable

```
score
```

to the text string of your generated notes. when the PythonObject is done processing, blue gets whatever value is assigned to that variable and parses it as a text score, then proceeds to do the standard operations of scaling and translating the notes in time, then applying any noteProcessors which may have been added to it.

Also, should you need, the variable

```
blueDuration
```

is assigned the value of the subjective duration of the PythonObject on the timeline.

So, within your scripts, you have access to the duration of the SoundObject should you want to make your script use that for any purpose you might have. one example might be that you have a score generating function that depends on duration, maybe if it has less duration it produces notes with louder amplitude and than if it has a longer duration.

A Simple Example

Let's say you're writing a simple Python script:

```
score = "i1 0 2 3 4 5\n"
```

1. the first thing blue does is clear the

   ```
   score
   ```

   variable in the interpreter and assign the variable

   ```
   blueDuration
   ```

   the value of the duration of the SoundObject on the timeline. For this example, it does not affect the outcome of the score generated

2. Next, blue runs the script. in this case, the only thing that happens is that the

`score`

variable is being assigned the text string of a single note, along with a newline.

3. At the end of the script's execution, the

   `score`

   variable is read. the score variable may or may not have anything assigned to it, but the script within the PythonObject is still run. this allows for the possibility of code that needs to be run but doesn't necessary need to generate score text at that time. (as mentioned above in the section called “The Case for Using the Python SoundObject”)

4. blue parses the score text, making Note objects for blue to use, applies scaling and translation of time to make the genreated notes start at the time of the PythonObject and last the subjective duration, then applies any noteProcessors.

And that's it!

A More Complex Example

For the following example, I will make a very simple score generator that produces as many notes as I give as an argument. the entire code for the script is:

```
def generateNotes(numOfNotes):
    scoreText = ""

    for i in range(numOfNotes):
        scoreText += "i1 " + str(i) + " 2 3 4 5\n"

    return scoreText

score = generateNotes(10)
```

The function I made,

`generateNotes(numOfNotes)`

, takes in the number of notes I want to generate. for the above, I wanted it to generate 10 notes, and if I printed out the above

`generateNotes(10)`

, I would have gotten the result:

```
i1 0 2 3 4 5
i1 1 2 3 4 5
i1 2 2 3 4 5
i1 3 2 3 4 5
i1 4 2 3 4 5
i1 5 2 3 4 5
i1 6 2 3 4 5
```

```
i1 7 2 3 4 5
i1 8 2 3 4 5
i1 9 2 3 4 5
```

This text string above is returned by the

```
generateNotes()
```

function and assigned to the

```
score
```

variable. blue then grabs that text from the

```
score
```

variable and parses it and the proceeds with compiling out the .CSD file.

Note: this is a very basic example of a note generator. As you work with Python more for score generation, you'll probably either create a slew of text-handling classes that use lots of regular expressions or a more object-based system that interacts amongst itself before generating score text output. I hope to address these types of design decisions in the near future in another tutorial which will hopefully help those new to scripting and programming learn how to go about analyzing music goals in terms of programming and then designing and implementing their solutions.

Usage Ideas

Using the external SoundObject and the Python SoundObject, besides their general usefulness of allowing scripting in blue, may be considered alternative methods of extending blue through script rather than compiling Java programs. The extensibility of the system is very open-ended, as whatever is available to Python and other languages now becomes accesible to blue, as well as using score-generating programs that are able to print to stdout.

So if you wanted to write a perl script that hits weather websites on the internet, gathers data from them, and then maps that data to different parameters of a note generating algorithm, you could build that outside of blue, but then have the added benefit of being able to use that within blue.

If you build a C++ program to generate scores, you're able to call it from within blue.

Maybe even something simple like doing a text include of a text file could be done rather easily with a few lines of Python or perl script.

Also, you can approach building scripts for tools as also being prototypes for building fully qualified blue SoundObjects and noteProcessors in Java, which has the added benefit of being able to have a GUI added to it. so if you have a utility script you use often, suggesting it to a Java programmer could lead to it's inclusion into blue, where it could be remade in a way to have it be usable by non-programmers as well.

Future Enhancements

As blue has developed over time, and as the ways to use it are being augmented with each new build, I'm always looking for new ways to make the process of working with blue easier. My recent work with the Python SoundObject and the External SoundObject have shown me that there's a lot of neat things that can be done with blue, things I really didn't account for when I first built blue, and would now love to handle. For the future, in addition to developing more and more tools for blue itself, it would be nice to be able to have more conventional coding tools to aid scripters in the script work for blue. I'm aware of

the limitations for debugging and console output right now in blue, but, then again, all of the scripting additions to blue are rather young, and blue's development continues every day.

For immediate goals, I've written down the following as things I should look into now:

Python test console for the PythonObject

Like the test button, but would show all output to stdout as it looks before being processed by blue, as well as showing all output to stderr.

Language neutral script library

With the PythonLib for the Python SoundObject, it's nice to have a portable way of keeping all of the scripts that can be used with blue along with blue, if you happen to be carrying your work around with you. I think that this ability would be fantastic for the external SoundObject. already i have a method that returns the absolute path to the lib directory of blue; with this, I think it should be possible that before blue runs any script, it could look for a "<BLUE_SCRIPT_LIB>" constant of some sort and do a text replace to put in the absolute path to the lib directory. that way, you could do something like:

```
from <BLUE_SCRIPT_LIB>.myLibrary import *
```

as well as use a commandline like:

perl <BLUE_SCRIPT_LIB>/myLibrary/myProgram -input $infile

Which would make the library accessible from any system. the nice thing about something like this is that I could then include other people's script libraries and programs as part of the blue release, or other people may wish to advertise that their scripts are usable within blue and have a place to install it in that is easily accessible on any blue system. (For example, using the above convention, if I have a blue work file that repeatedly calls a library that someone else developed, I can put that library in my lib dir, use the convention above, and then say if someone else wanted to run that file or if I was bringing that file to another computer, all I'd have to do is install the library in the lib directory and everything should work, as opposed to having to do a find and replace on ever call to the library to change the path to where it is on the new computer.)

Finding a text editor component thats made for programming

This would be fantastic, to find a Java text component that I can swap in that would handle things like syntax-hilighting of different languages, maybe has some kind of help system for syntax and references, etc. I haven't found one I could easily use or modify yet, but hopefully something will come along.

Final Thoughts

As time goes on, I'm sure there will be alot of additions to blue to help aid scripting and the use of other score generating programs within blue. I hope this tutorial has helped to show how to use the external SoundObject and Python SoundObject in blue, as well as helped show what's very possible by using these SoundObjects.

If you have any comments, suggestions for improving this tutorial, or questions, please feel free to email me at `<stevenyi@gmail.com.>`

Thanks and good luck!

Steven

Sound SoundObject

Steven Yi

2002

Revision History

Revision 1.1	2004.11.7
Converted to DocBook. Some information revised as no longer applicable	
Revision 1.0	2002.11.6
First version of article.	

Introduction

One of the more curious soundObjects in blue is the *Sound SoundObject*, which allows for the direct writing of instruments in the score. It's origin came a long time ago now when I was first beginning to create more soundObjects for blue. I was thinking of what exactly should be possible for a soundObject, thinking that if someone were to make a soundObject, they should be able to also embed instruments and f-tables that they'll know will always be generated and available. The case which I was thinking about at the time were non-generic soundObjects like a drum machine, where the soundObject would not only have a GUI to develop pattern tracks for different drum sounds, but also would be able to furnish the instruments and f-tables needed to generate those sounds, the idea being that one could release a soundObject that a user could use "straight out of the box", no instrument writing or tables required. That design decision eventually lead me to the conception of the *Sound SoundObject*, which uses those mechanisms that were put in place for all soundObjects. Theoretically I was very fascinated with the possibility of writing instruments within the main scoring area, mixing note blocks with pure sound blocks. It became possible to really compose with the sound in a very direct manner; saying things like "I want a sine wave here with this envelope, then a triangle wave here, and on top of that i want a processed sample sound to come in here" now had a direct translation. You didn't need to write the instrument, then write a note for it, and then have to work with both of them as separate entities. You could just add a *Sound SoundObject* block on the timeline, write your sound using standard orc code, and move it around in time on the timeline. After having thought through that possibility to express my musical goals by using the *Sound SoundObject*, I found that it opened up a lot of how I saw blue's timeline as well as how I thought about ways to work with blue.

As a Csound user or general electronic musician, it might seem strange to think of directly writing instruments in a score, especially in context to the existing music tools and musical models that you've probably worked with. However, by having the *Sound SoundObject*, I've found that interesting possibilities have opened up, and since it's creation and inclusion into blue it has been used and has played a part of just about every piece that I've worked on.

Note: It's not that it's impossible to implement this any other way. Really, it's just a single instrument with a single note, but, that's a technical issue, an implementation issue. The interesting thing is that when working with it, it's really a different thing altogether conceptually. It's the *sound soundObject* in the context of the timeline that makes its usage interesting and useful(well, for me at least!).

The following sections will go over how to use *Sound SoundObject*, what happens when it gets processed by blue, as well as some usage scenarios and patterns that have arose while using it in my own work.

How to Use the Sound SoundObject

First, insert the *Sound SoundObject* as you would any soundObject by rt-clicking (for Mac users, hold down the apple key and click) on the main timeline of the score area and chose "Add New Sound" from the popup menu.

A *Sound SoundObject* will be inserted wherever you clicked on the timeline. You'll notice that this soundObject acts and behaves like any other soundObject: you can move it around in time, drag to change the duration, change it's name in the soundObject property dialog, and click on it to edit it. You won't be able to add any noteProcessors to it, however, but that will be explained more in detail in the "What happens when it gets processed" section.

If you click on the soundObject to pull up its editor, you'll see a text box with a default message "insert instrument text here". In the editor is where you write your instrument definition, but without writing any "instr 11" or "endin": the number of the instrument and the correct formatting of the instrument is handled by the soundObject. For a simple example, you might try:

```
aout    oscili 30000, 440, 1
        outs aout, aout
```

where the 1 at the end of the oscili line is an ftable numbered 1, defined in the tables editor under the tables tab. (For our example, let's go ahead and definte the ftable as "f1 0 65536 10 1", which is a sin wave). At this point you might want to try listening your work file with the play button (assuming you've set up the command line under project properties to a command line that will output to speaker; for more information on this, please consult the blue user's manual). What you should hear is that where you've put your *Sound SoundObject* on the timeline (i.e. starts at .5 seconds) should play the sound you've written, in this example case, a sine wave at 440hz at 30000 amp, playing at equal volume out both left and right channels.

And that's pretty much it! From here you might want to try copying the block and pasting it a few times, changing some parameters, embellishing your instrument definitions, etc. Also, you might want to try mixing this with other soundObject, i.e. write some instruments in the orchestra manager, then write some genericScore soundObjects to play those instruments, as well as use some *Sound SoundObjects* directly on the timeline.

What Happens When It Gets Processed

If your confused as to what's going on, it'll probably help to know exactly what happens when blue processes *Sound SoundObjects*. blue, whenever it goes to create a .CSD file to use with a commandline or to generatoe out to a file, has different stages of its compilation. The relevant parts to know here are that all instruments from the orchestra manager are first generated, but not yet put into the .CSD. Next, all soundObjects are called to generate any instruments they might have. This is where the *Sound SoundObject* would generate an instrument from your text input. The generated instruments from *Sound SoundObjects* at this point are assigned an instrument number. After all instruments are generated from soundObjects, all score text is then generated. At this point, the instrument number assigned in the earlier pass is now used by the *Sound SoundObject* to generate a note for your instrument. The note generated by the *Sound SoundObject* consists of only three p-fields: the instrument number, the start of the soundObject, and the duration. No other p-fields are generated (so your instrument should not use any other p-fields).

For example, let's say you have a *Sound SoundObject* with a start time at 0.5 seconds and a duration of 2 seconds. When blue goes to get it's instrument, let's say it is assigned instrument number 2. The generated note will be:

```
i2 0.5 2
```

Perhaps the best way to see it is to do the simple example from the "How to use the Sound SoundObject" section, generate a CSD file (from the Project menu, select "Generate CSD to file"), and inspect what got generated. Comparing the soundObject's representation on the timeline as a sound and seeing how it got generated out might explain things better.

(NOTE: because the *Sound SoundObject* only write out the three p-fields, using a noteProcessor really doesn't have any purpose, which is why the *sound soundObject* does not support noteProcessors)

Ultimately in the lowest level implementation, there is a separation of a note as well as an instrument, but within blue, that separation is hidden from the user. From the user's point of view, all they have to do is write their sounds on the timeline, and they don't have to worry about numbering the instruments or creating notes for that instrument.

Usage Scenarios and Patterns

Prototyping/Sketching

Most of the time I've found myself using the *Sound SoundObject* at the start of a project when I'm creating new sounds for a piece. I find it's good a prototyping tool, initially working with sounds on the scoreTimeCanvas (this is the name of java object that is the main score timeline area). Usually, when I write instruments in the *Sound SoundObject*, I define all of the i-time variables in a way that will facilitate easy conversion to full-fledged instruments should I later want to do so. This is a general instrument writing pattern of mine that I would do anyways even before I had blue to use. For example, I might be designing an sound on the timeline with the something like the following at the top of the text:

```
ipch   = cpspch(8.02)
iamp  = ampdb(80)
ispace  = .2
```

Later, when I get to a point after sketching out some sounds and finding I like how things are beginning to flow in the piece, I find that i usually want to start working with the sounds then as instruments. At this point, I normally convert the *Sound SoundObject* to a genericScore object (done by rt-clicking on the soundObject and picking "Convert to Generic Score" from the popup menu), which automatically takes the the instrument from the soundObject and adds it to the orchestra under the orchestra manager, and also leaves me with a single three p-field note, which also shows me what instrument number the instrument was assigned. After that I'll go to the orchestra manager and edit the instrument to now take in more p-fields, changing the top text to something like:

```
ipch   = cpspch(p4)
iamp  = ampdb(p5)
ispace  = p6
```

I usually find myself making two or three different sounds, then copying a bunch of them and changing a few parameters to try out things in time, then converting them into instruments. It's a nice separation to have the ability to work just with sounds at the start of a piece for me, as really, that's what I'm concerned with at the beginning of a piece, finding the sounds and initially sculpting the sound space that the piece will take on. After I find what I'm looking for, it's easy for me to convert all the sounds into instruments and then proceed from there.

Sound Design

Sometimes when you're wanting to just to build a single sound or texture, maybe to use as a sound effect in project, you might not really be thinking in terms of scores, notes, and instruments but rather in terms of sounds in time. In situations like this, the *Sound SoundObject* would be the first thing I would use, and might really be the only soundObject I would use. Notes, as a concept, somtimes really don't play a part of the musical model for a piece. It's not that you have all these instruments being played everywhere, but

rather you have sounds going on here and there. It might seem like I'm being a little to theoretical here, but I really think it does play a part in the work process.

In the situations I've been in when I've been asked to make a sound for a friend's website or game, I've found it nice to fire up blue, set the timeline to a really close-up zoom on time, and just work from there to craft a sound. I would add a *Sound SoundObject* here and there, maybe use some global variables so I can make *Sound SoundObjects* that might just function as an lfo or other control instrument, moving things around just slightly around in time to sculpt the sound. An oscillator here, maybe blending it into an fm sound, throw in a noise generator with some formants and a notch filter sweep...

Learning and Practice

Sometimes I find myself just making sounds with blue and Csound. It might be because I'm just curious to try something out, I might be working on really getting to know a synthesis technique, trying to learn how to express a sound in my mind or maybe to better train my imagination to know what a sound will really sound like when I write it down, etc. Sometimes its just that I want to try out some new opcodes I haven't really ever used.

It's times like this when I find myself just using the *Sound SoundObject*, as I'm not interested in the note-instrument paradigm, it's the furthest thing from my mind. I'm focused on achieving a sound, or on experimenting to see what is the sound of instrument code I've just written. And I want the flexibility to add more sounds on the timeline: I don't want to break my concentration to go and think about numbering instruments, writing notes, moving the note around in time. I want to see and work with it all of it in one place., as that's what's going on in my mind.

It's also great practice too, just writing alot of instruments. I'd imagine that the *Sound SoundObject* would be a useful tool for a person new Csound, as it allows just working with instrument code. (Note: you would still need to know the basics of how Csound works, what is a CSD, and understand how things in blue map the different parts if itself to the different parts of a CSD file).

Final Thoughts

Thanks for reading the tutorial! I hope this tutorial has helped to show how to use *Sound SoundObject* in blue, as well as helped show some ways in which you might want to use it.

If you have any comments, suggestions for improving this tutorial, or questions, please feel free to email me at `<stevenyi@gmail.com.>`

Thanks and good luck!

Steven

Part III. Developers

Information here for Developers.

Chapter 4. Extending Blue

Introduction

Developers have a number of different methods for extending Blue. As Blue is a modular application, built using the Netbeans Rich Client Platform (RCP), any standard RCP module and suite can be added to Blue. In conjuction with Blue's module's public APIs, this allows great flexibility for a developer to add new modules to Blue.

Additionally, rather than adding completely new windows and pieces to Blue, a developer may decide to create plugins to existing Blue architectural pieces. The following sections discuss the various plugin types and the steps to take to create your own plugin.

Basics of Plugins

Blue Plugins come in various forms. In general, a plugin is a Java Interface that has a number of implementations. Besides the implementation of the plugin, plugins must be registered for them to be discovered. The following discusses these various aspects in further detail

Note Processors

Instruments

Sound Objects

BlueSynthBuilder Widgets

Chapter 5. Core

Building blue

Requirements

To build blue, the following programs are used:

Java Development Kit 1.7+	Contains javac java compiler and other development tools.
Netbeans 8.0	The Netbeans IDE is used to develop Blue. It is recommended to use Netbeans for compiling and developing the program. It is available at http://www.netbeans.org.
Apache Ant	Ant is build tool for Java projects. It is available at http://ant.apache.org. This is required if building from the commandline.
JUnit	JUnit is required to run the unit tests and is integrated as part of the build.
xsltproc, Apache FOP, Sun JAI library	xsltproc is used to build all versions of the manual. To build the PDF manual, Apache FOP and Sun JAI library are required.

Getting the Sources

Blue's source code is maintained at Github at https://github.com/kunstmusik/blue. You can download archives of the source code or clone the Git repository from there.