MidiKinesis — MIDI controllers for (almost) any purpose

Peter Brinkmann
Technische Universität Berlin
Fakultät II – Mathematik und Naturwissenschaften
Institut für Mathematik
Sekretariat MA 3-2
Straße des 17. Juni 136
D-10623 Berlin
`brinkman@math.tu-berlin.de`

Abstract

MidiKinesis is a Python package that maps MIDI control change events to user-defined X events, with the purpose of controlling almost any graphical user interface using the buttons, dials, and sliders on a MIDI keyboard controller such as the Edirol PCR-30. Key ingredients are Python modules providing access to the ALSA sequencer as well as the XTest standard extension.

Keywords

ALSA sequencer, MIDI routing, X programming, Python

1 Introduction

When experimenting with Matthias Nagorni's AlsaModularSynth, I was impressed with its ability to bind synth parameters to MIDI events on the fly. This feature is more than just a convenience; the ability to fine-tune parameters without having to go back and forth between the MIDI keyboard and the console dramatically increases productivity because one can adjust several parameters almost simultaneously, without losing momentum by having to navigate a graphical user interface. Such a feature can make the difference between settling for a "good enough" choice of parameters and actually finding the "sweet spot" where everything sounds just right.

Alas, a lot of audio software for Linux does not expect any MIDI input, and even in programs that can be controlled via MIDI, the act of setting up a MIDI controller tends to be less immediate than the elegant follow-and-bind approach of AlsaModularSynth. I set out to build a tool that would map MIDI control change events to GUI events, and learn new mappings on the fly. The result was MidiKinesis, the subject of this note. MidiKinesis makes it possible to control almost any graphical user interface from a MIDI controller keyboard.

2 Basics

Before implementing MidiKinesis, I settled on the following basic decisions:

- MidiKinesis will perform all its I/O through the ALSA sequencer (in particular, no reading from/writing to `/dev/midi*`), and it will act on graphical user interfaces by creating plain X events.
- MidiKinesis will be implemented in Python (Section 8), with extensions written in C as necessary. Dependencies on nonstandard Python packages should be minimized.

Limiting the scope to ALSA and X effectively locks MidiKinesis into the Linux platform (it might work on a Mac running ALSA as well as X, but this seems like a far-fetched scenario), but I decided not to aim for portability because MidiKinesis solves a problem that's rather specific to Linux audio.[1] The vast majority of Mac or Windows users will do their audio work with commercial tools like Cubase or Logic, and their MIDI support already is as smooth as one could possibly hope. The benefit of limiting MidiKinesis in this fashion is a drastic simplification of the design; at the time of this writing, MidiKinesis only consists of about 2000 lines of code.

When I started thinking about this project, my first idea was to query various target programs in order to find out what widgets their user interfaces consist of, but this approach turned out to be too complicated. Ultimately, it would have required individual handling of toolkits like GTK, Qt, Swing, etc., and in many cases the required information would not have been forthcoming. So, I decided to settle for

[1] I did put a thin abstraction layer between low-level implementation details and high-level application code (Section 3), so that it is theoretically possible to rewrite the low-level code for Windows or Macs without breaking the application code.

the lowest common denominator — MidiKinesis directly operates on the X Window System, using the XTest standard extension to generate events. I expected this approach to be somewhat fragile as well as tedious to calibrate, but in practice it works rather well (Section 5).

For the purposes of MidiKinesis, Python seemed like a particularly good choice because it is easy to extend with C (crucial for hooking into the ALSA sequencer and X) and well suited for rapidly building MIDI filters, GUIs, etc. Python is easily fast enough to deal with MIDI events in real time, so that performance is not a concern in this context. On top of Python, MidiKinesis uses Tkinter (the de facto standard toolkit for Python GUIs, included in many Linux distributions), and ctypes (Section 8) (not a standard module, but easy to obtain and install).

3 The bottom level

At the lowest level, the modules `pyseq.py` and `pyrobot.py` provide access to the ALSA sequencer and the XTest library, respectively. There are many ways of extending Python with C, such as the Python/C API, Boost.Python, automatic wrapper generators like SIP or SWIG, and hybrid languages like pyrex. In the end, I chose ctypes because of its ability to create and manipulate C structs and unions. This is crucial for working with ALSA and X since both rely on elaborate structs and unions (`snd_seq_event_t` and `XEvent`) for passing events.

3.1 Accessing the ALSA sequencer

The module `pyseq.py` defines a Python shadow class that provides access to the full sequencer event struct of ALSA (`snd_seq_event_t`). Moreover, the file `pyseq.c` provides a few convenience functions, most importantly `midiLoop(...)`, which starts a loop that waits for MIDI events, and calls a Python callback function when a MIDI event comes in.

The module `pyseq.py` also provides an abstraction layer that protects application programmers from such implementation details. To this end, `pyseq.py` introduces the following classes:

PySeq manages ALSA sequencer handles, and it provides methods for creating MIDI ports, sending MIDI events, etc. Application programmers will subclass PySeq and override the methods `init` (called by the constructor) and `callback` (called when a MIDI event arrives at an input port of the corresponding sequencer).

MidiThread is a subclass of `threading.Thread` that provides support for handling incoming MIDI events in a separate thread. An instance of MidiThread keeps a pointer to an instance of PySeq whose callback method is called when a MIDI event comes in.

Using this class structure, a simple MIDI filter might look like this:

```
from pyseq import *

class MidiTee(PySeq):
  def init(self, *args):
    self.createInPort()
    self.out=self.createOutPort()
  def callback(self, ev):
    print ev
    self.sendEvent(ev, self.out)
    return 1

seq=MidiTee('miditee')
t=MidiThread(seq)
t.start()
raw_input('press enter to finish')
```

This filter acts much like the venerable `tee` command. It reads MIDI events from its input port and writes them verbatim to its output port, and it prints string representations of MIDI events to the console. The last line is necessary because instances of MidiThread are, by default, daemon threads.

Once started, an instance of MidiThread will spend most of its time in the C function `midiLoop`, which in turn spends most of its time waiting for events in a `poll(...)` system call. In other words, instances of MidiThread hardly put any strain on the CPU at all.

3.2 Capturing and sending X events

Structurally, the module `pyrobot.py` is quite similar to `pyseq.py`. It uses ctypes to create a Python shadow class for the XEvent union of X, and it introduces a simple abstraction layer that protects application programmers from such details. The main classes are as follows:

PyRobot is named after Java's java.awt.Robot. It provides basic functionality for capturing and sending X events.

Script uses PyRobot to record and play sequences (scripts) of mouse and keyboard events.

The following code records a sequence of mouse and keyboard events and replays it.

```
from pyrobot import *

R=PyRobot()
S=Script()
print 'record script, press Escape'
S.record(R)
raw_input('press enter to replay')
S.play(R)
```

4 The top level

The module `midikinesis.py` is the main application of the package. It waits for incoming MIDI control change events. If it receives a known event, it triggers the appropriate action. Otherwise, it asks the user to assign one of six possible mappings to the current event:

Button maps a button on the MIDI keyboard to a sequence of X events (Section 3.2) that the user records when setting up this sort of mapping. This sequence will typically consist of mouse motions, clicks, and keystrokes, but mouse dragging events are also admissible.

It is also possible to assign Button mappings to dials and sliders on the keyboard. To this end, one defines a threshold value between 0 and 127 (64 is the default) and chooses whether a rising edge or a falling edge across the threshold is to trigger the recorded sequence of X events.

Slider maps MIDI events from a slider or dial on the MIDI keyboard to mouse dragging events along a linear widget (such as a linear volume control or scrollbar). For calibration purposes, `midikinesis.py` asks the user to click on the bottom, top, and current location of the widget. Hydrogen and jamin are examples of audio software with widgets of this kind. The button used to click on the bottom location determines the button used for dragging.

Selection divides the range of controller values (0 . . . 127) into brackets, with each bracket corresponding to a mouse click on a user-defined location on the screen. It primarily targets radio buttons.

Counter handles counter widgets whose values are changed by repeatedly clicking on up/down arrows. Specimen uses widgets of this kind. Counter mappings are calibrated by clicking on the location of the up/down arrows and by specifying a step size, i.e., the number of clicks that a unit change of the controller value corresponds to.

Circular Dial behaves much like Slider, except it drags the mouse in a circular motion. amSynth has dials that work in this fashion.

Linear Dial sounds like an oxymoron, but the name merely reflects the dual nature of the widgets that it targets. To wit, there are widgets that look like dials on the screen, but they get adjusted by pressing the mouse button on the widget and dragging the mouse up or down in a linear motion. Rosegarden4, ZynAddSubFX, and Hydrogen all have widgets of this kind.

These six mappings cover most widgets that commonly occur in Linux audio software. Button mappings are probably the most general as well as the least obvious feature. Here is a simple application of some of Button mappings, using Rosegarden4:

- Map three buttons on the MIDI keyboard to mouse clicks on Start, Stop, and Record in Rosegarden4.
- Map a few more buttons to mouse clicks on different tracks in Rosegarden4, followed by the Delete key.

Pressing one of the latter buttons will activate and clear a track, so that it is possible to record a piece consisting of several tracks (and to record many takes of each track) without touching the console after the initial calibration.

5 Subtleties

One might expect the approach of mapping MIDI events to plain X events to be rather fragile because simple actions like moving, resizing, or covering windows may break existing mappings. In practice, careful placement of application windows on the desktop will eliminate most problems of this kind. Moreover, MidiKinesis provides a number of mechanisms that increase robustness:

- MidiKinesis computes coordinates of X events relative to a reference point. If all mappings target just one window, then one makes the upper left-hand corner of this window the reference point, and if the window moves, one only needs to update the reference point.

- The notion of reference points also makes it possible to save mappings to a file and restore them later.
- It helps to start one instance of MidiKinesis for each target window, each with its individual reference point. This will resolve most window placement issues. In order to make this work, one launches and calibrates them individually. When an instance of MidiKinesis has been set up, one tells it to ignore unknown events and moves on to the next instance. Like this, instances of MidiKinesis won't compete for the same MIDI events.
- Instances of MidiKinesis only accept events whose channel and parameter match certain patterns.[2] By choosing different patterns for different instances of MidiKinesis, one keeps them from interfering with each other, while each still accepts new mappings.

6 Fringe benefits

Using the module `pyseq.py`, one can rapidly build MIDI filters. I even find the simple MidiTee example from Section 3.1 useful for eavesdropping on conversations between MIDI devices (tools like amidi serve a similar purpose, but I like being able to adjust the output format on the fly, depending on what I'm interested in).

One of the first applications I built on top of `pyseq.py` was a simple bulk dump handler for my MIDI keyboard. It shouldn't be much harder to build sophisticated configuration editors for various MIDI devices in a similar fashion.

I was surprised to find out that there seemed to be no approximation of java.awt.Robot for Python and Linux before `pyrobot.py`, so that `pyrobot.py` might be useful in its own right, independently of audio applications.

7 Where to go from here

While the focus of MidiKinesis has squarely been on audio applications, it also opens the possibility of using MIDI events to control all kinds of software. For instance, when I was implementing Slider mappings, I used the vertical scrollbar of Firefox as a test case.

In order to illustrate the basic idea of creative misuse of MIDI events, I implemented a simple game of Pong, controlled by a slider or dial on a MIDI keyboard. Generally speaking, a mouse is a notoriously sloppy input device, while MIDI controllers tend to be rather precise. So, a tool like MidiKinesis might be useful in a nonaudio context requiring speed and accuracy.

Finally, I feel that the applications of MidiKinesis that I have found so far barely scratch the surface. For instance, the ability to record and replay (almost) arbitrary sequences of X events has considerable potential beyond the examples that I have tried so far.

8 Resources

MidiKinesis is available at `http://www.math.tu-berlin.de/~brinkman/software/midikinesis/midikinesis.tgz`. MidiKinesis requires Python, Tkinter, and ctypes, as well as an X server that supports the XTest standard extension.

Python is a powerful scripting language, available at `http://www.python.org/`. Chances are that you are using a Linux distribution that already has Python installed.

Tkinter is the de facto standard toolkit for Python GUIs, available at `http://www.python.org/moin/TkInter`. It is included in many popular Linux distributions.

ctypes is a package to create and manipulate C data types in Python, and to call functions in shared libraries. It is available at `http://starship.python.net/crew/theller/ctypes/`.

9 Acknowledgements

Special thanks go to Holger Pietsch for getting me started on the X programming part of the project, and to the members of the LAU and LAD mailing lists for their help and support.

[2]By default, MidiKinesis accepts events from General Purpose Controllers on any channel.