

MidiKinesis — MIDI controllers for (almost) any purpose

Peter Brinkmann

April 21, 2005

MidiKinesis — MIDI controllers for (almost) any purpose

Peter Brinkmann

April 21, 2005

*The basic idea is a horrible hack suitable to frighten
small children ;)* anonymous referee

Motivation

When programming a softsynth, one frequently wants to tweak parameters using the dials, sliders, and buttons on a MIDI keyboard controller. There are various approaches to mapping MIDI control change messages, such as:

Motivation

When programming a softsynth, one frequently wants to tweak parameters using the dials, sliders, and buttons on a MIDI keyboard controller. There are various approaches to mapping MIDI control change messages, such as:

- Bind MIDI controllers to synth parameters according to MIDI specification

Motivation

When programming a softsynth, one frequently wants to tweak parameters using the dials, sliders, and buttons on a MIDI keyboard controller. There are various approaches to mapping MIDI control change messages, such as:

- Bind MIDI controllers to synth parameters according to MIDI specification
- Watch incoming MIDI events and allow the user to bind new MIDI events to synth parameters on the fly (example: `AlsaModularSynth`)

Motivation

When programming a softsynth, one frequently wants to tweak parameters using the dials, sliders, and buttons on a MIDI keyboard controller. There are various approaches to mapping MIDI control change messages, such as:

- Bind MIDI controllers to synth parameters according to MIDI specification
- Watch incoming MIDI events and allow the user to bind new MIDI events to synth parameters on the fly (example: `AlsaModularSynth`)

Goal: Build a tool, `MidiKinesis`, that adds `AlsaModularSynth`'s follow-and-bind approach to (almost) any piece of software

Basic decisions

MidiKinesis will

- be implemented in Python, with a few extensions written in C,

Basic decisions

MidiKinesis will

- be implemented in Python, with a few extensions written in C,
- receive and send MIDI events via the ALSA sequencer,

Basic decisions

MidiKinesis will

- be implemented in Python, with a few extensions written in C,
- receive and send MIDI events via the ALSA sequencer, and
- interact with other software by creating X events

Basic decisions

MidiKinesis will

- be implemented in Python, with a few extensions written in C,
- receive and send MIDI events via the ALSA sequencer, and
- interact with other software by creating X events

Some issues:

- Using ALSA and X effectively locks MidiKinesis into the Linux platform

Basic decisions

MidiKinesis will

- be implemented in Python, with a few extensions written in C,
- receive and send MIDI events via the ALSA sequencer, and
- interact with other software by creating X events

Some issues:

- Using ALSA and X effectively locks MidiKinesis into the Linux platform
- One might expect the approach via X events to be rather fragile (more on this later)

ALSA and Python

The module `pyseq.py` provides Python bindings for the ALSA sequencer using the `ctypes` package. It provides a thin abstraction layer consisting of the following two classes:

ALSA and Python

The module `pyseq.py` provides Python bindings for the ALSA sequencer using the `ctypes` package. It provides a thin abstraction layer consisting of the following two 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` and `callback`.

ALSA and Python

The module `pyseq.py` provides Python bindings for the ALSA sequencer using the `ctypes` package. It provides a thin abstraction layer consisting of the following two 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` and `callback`.

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.

Sample code

```
from pyseq import *

class MidiTee(PySeq):
    def init(self, *args):
        self.createInPort()
        self.out=self.createOutPort()
    def callback(self, event):
        self.sendEvent(event, self.out)
        print event
        return 1      # free event; we're done with it

seq=MidiTee("miditee")
t=MidiThread(seq)
t.start()
raw_input("press enter to finish")
```

XTest and Python

The module `pyrobot.py` uses `ctypes` to create a Python shadow class for the XEvent union of X, and it introduces a simple abstraction layer consisting of two classes:

XTest and Python

The module `pyrobot.py` uses `ctypes` to create a Python shadow class for the XEvent union of X, and it introduces a simple abstraction layer consisting of two classes:

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

XTest and Python

The module `pyrobot.py` uses `ctypes` to create a Python shadow class for the XEvent union of X, and it introduces a simple abstraction layer consisting of two classes:

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

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

XTest and Python

The module `pyrobot.py` uses `ctypes` to create a Python shadow class for the XEvent union of X, and it introduces a simple abstraction layer consisting of two classes:

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

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

```
from pyrobot import *
R=PyRobot()
S=Script()
print "record script, press Escape"
S.record(R)
raw_input("press enter to replay")
S.play(R)
```

The main application

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

The main application

The module `midikinesis.py` is the main application of the package. If it receives a known MIDI 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.

The main application

The module `midikinesis.py` is the main application of the package. If it receives a known MIDI 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.
- Slider** maps MIDI events from a slider or dial on the MIDI keyboard to mouse dragging events along a scrollbar.

The main application

The module `midikinesis.py` is the main application of the package. If it receives a known MIDI 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.

Slider maps MIDI events from a slider or dial on the MIDI keyboard to mouse dragging events along a scrollbar.

Selection is for use with radio buttons.

The main application

The module `midikinesis.py` is the main application of the package. If it receives a known MIDI 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.

Slider maps MIDI events from a slider or dial on the MIDI keyboard to mouse dragging events along a scrollbar.

Selection is for use with radio buttons.

Counter handles counter widgets whose values are changed by repeatedly clicking on up/down arrows.

The main application

The module `midikinesis.py` is the main application of the package. If it receives a known MIDI 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.

Slider maps MIDI events from a slider or dial on the MIDI keyboard to mouse dragging events along a scrollbar.

Selection is for use with radio buttons.

Counter handles counter widgets whose values are changed by repeatedly clicking on up/down arrows.

Circular Dial behaves much like Slider, except it drags the mouse in a circular motion.

The main application

The module `midikinesis.py` is the main application of the package. If it receives a known MIDI 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.

Slider maps MIDI events from a slider or dial on the MIDI keyboard to mouse dragging events along a scrollbar.

Selection is for use with radio buttons.

Counter handles counter widgets whose values are changed by repeatedly clicking on up/down arrows.

Circular Dial behaves much like Slider, except it drags the mouse in a circular motion.

Linear Dial sounds like an oxymoron, but the name merely reflects the dual nature of the widgets that it targets.

Subtleties

The approach of controlling software via X events seems fragile at first sight, but it works rather well in practice.

Subtleties

The approach of controlling software via X events seems fragile at first sight, but it works rather well in practice.

- MidiKinesis computes X events relative to *reference points*, so that windows may be moved without breaking mappings.

Subtleties

The approach of controlling software via X events seems fragile at first sight, but it works rather well in practice.

- MidiKinesis computes X events relative to *reference points*, so that windows may be moved without breaking mappings.
- One can have one instance of MidiKinesis per target window.

Subtleties

The approach of controlling software via X events seems fragile at first sight, but it works rather well in practice.

- MidiKinesis computes X events relative to *reference points*, so that windows may be moved without breaking mappings.
- One can have one instance of MidiKinesis per target window.
- MidiKinesis only accepts events whose channel and parameter match certain patterns. Different patterns for different instances keep them from interfering with each other.

Conclusion

- MidiKinesis is meant to be fun.

Conclusion

- MidiKinesis is meant to be fun.
- With luck, it may help popularize the follow-and-bind approach to MIDI mappings.

Conclusion

- MidiKinesis is meant to be fun.
- With luck, it may help popularize the follow-and-bind approach to MIDI mappings.
- PyRobot fills a gap in the Python libraries and should be useful for regression testing.

Conclusion

- MidiKinesis is meant to be fun.
- With luck, it may help popularize the follow-and-bind approach to MIDI mappings.
- PyRobot fills a gap in the Python libraries and should be useful for regression testing.
- PySeq allows users to create MIDI filters on the fly, with very little code.

Conclusion

- MidiKinesis is meant to be fun.
- With luck, it may help popularize the follow-and-bind approach to MIDI mappings.
- PyRobot fills a gap in the Python libraries and should be useful for regression testing.
- PySeq allows users to create MIDI filters on the fly, with very little code.
- ...creative misuse of MIDI events...