Modality
modal control in SuperCollider

Jeff Carey, Alberto de Campo, Wouter Snoei
Hannes Hölzl, Robert van Heumen, Bjørnar Habbestad
Marije Baalman, Till Bovermann, Miguel Negrão

The beginning

What is Modality?

- a loose collaboration between a couple of SuperCollider developers and users
- goal is to create a toolkit to make it easy to hook up controllers to sound in SuperCollider
- do complicated mapping
- change mappings on the fly, while playing the instrument (modal control)

What is Modality?

- a loose collaboration between a couple of SuperCollider developers and users
- goal is to create a toolkit to make it easy to hook up controllers to sound in SuperCollider
- do complicated mapping
- change mappings on the fly, while playing the instrument (modal control)

First meeting between Jeff Carey, Bjørnar Habbestad, Alberto de Campo, Wouter Snoei and Marije Baalman in September/October 2010 in Bergen (BEK).

Second meeting at STEIM in May 2011 with Jeff Carey, Alberto de Campo, Marije Baalman, Till Bovermann, Miguel Negrão, Hannes Hölzl and Robert van Heumen.

Semi-regular email contact and chance meetings since then...

Process and work approach

STEIM residency:

- Presentation and discussion at SC-user meeting
- first three days presentations of each others work
- and a lot of discussion
- and brainstorming
- Then coding, coding, coding
- and brainstorming
- and playing a concert
- some more coding and discussion
- and creating some instruments for presentations.

Bergen residency followed a similar scheme (but also involving walks on mountains)

The concept

Concept

- Support different devices that work with different protocols (MIDI, HID, OSC, Serial, etc)
- Provide a common interface to use these devices
- Provide a system to process the data from these devices
- Make real and virtual interfaces interchangeable (GUI for device, processed data for device, etc)

Realisation

- Support different devices that work with different protocols (MIDIMKtl, HIDMKtl)
- Provide a common interface to use these devices (MKtl)
- Provide a system to process the data from these devices (MDispatch or FRP approach)
- Templates for various devices
- Templates for various common ways of processing
- Same interface for MKtl and MDispatch

We are considering moving MDispatch out again in favour of FRP...

Workshop

Workshop overview

- Intro (to which you just listened)
- Installing Modality and dependencies
- Hooking up your devices
- Creating templates for your devices
- Actions for controllers
- What is FRP?
- Writing FRP’s with your controllers

Installing Modality

- Get the code from: `https://github.com/ModalityTeam/Modality-toolkit`
- Open the file “Installation.scd” in SuperCollider, and follow the instructions there
- Install the “FP” quark from the main quarks.
- Recompile sclang!

The end

We’re not there yet...

- GUI replacements and/or visualisation for controllers
- Backends for OSC and Serial based devices (and others?)

We’re not there yet...

- GUI replacements and/or visualisation for controllers
- Backends for OSC and Serial based devices (and others?)

Where to have our next workshop week?

We’re not there yet...

- GUI replacements and/or visualisation for controllers
- Backends for OSC and Serial based devices (and others?)

Where to have our next workshop week?

Who else could join in?

Acknowledgements

BEK - Bergen, Norway
`www.bek.no`

STEIM - Amsterdam, The Netherlands
`www.steim.org`

Jeff Carey - getting us all together

The rest of the presentation

What is the problem?

- Different protocols have different ways of transporting the data
- Different operating systems (well, Linux and OSX) provide data in different ways
- Semantics of different controllers

A general solution is not so trivial.

Issues in accessing devices

- HID: different subsystems in OSX and Linux:
  - causing elements to be numbered differently,
  - as well as scaling,
  - as well as names reported with slight variations.
- MIDI: device names are reported differently between OSX and Linux,
- real MIDI devices (not USB-MIDI) will not report their names, but simply be MIDI-ports.
- Similar issues are to be expected for OSC or SerialPort devices.

Device description index


```
IdentityDictionary[
\nanoKONTROL ->
        (osx: ( device: "nanoKONTROL"),
        linux: ( device: "nanoKONTROL-nanoKONTROL MIDI 1"),
        protocol: \midi, file: "nanoKONTROL.desc.scd" ),
\GamePad -> ( type: \template, protocol: \hid,
        file: "GamePad.desc.scd" ), // gamepad template
\Run_N_Drive ->
        (osx: ( device: "Run'N' Drive"),
        linux: (device: "Thrustmaster Run\'N\' Drive"),
        protocol: \hid, file: "Run_N_Drive.desc.scd" ),
\manta ->
        (inport: 1234, outport: 5678, protocol: \osc,
        file: "Manta.desc.scd" )
]
```

Device description


```
// right hand side four labeled buttons
\bt_R_1, (type: \button, osx: (cookie: 2), linux: (slot: [1,304]),
    spec: \hidBut, mode: \push),
\bt_R_2, (type: \button, osx: (cookie: 3), linux: (slot: [1,305]),
  spec: \hidBut, mode: \push),

// joystick axes switches
\joy_L_X, (type: \joyAxis, osx: (cookie: 15, spec: \cent255inv),
linux: (slot: [3,0], spec: \cent1 ),  mode: \center),
\joy_L_Y, (type: \joyAxis, osx: (cookie: 16, spec: \cent255 ),
linux: (slot: [3,1], spec: \cent1 ),  mode: \center),
```

Basic elements of device descriptions

- Array of elements: name, event/dictionary with specifications.
- Whenever something is different between platforms, you put the platform specific stuff in a sub-dictionary containing the specifics. When the file is parsed, MKtl will check on which platform it is running, and handle things accordingly.
- Naming of elements are somewhat hierarchical, so they are easy to sort out using pattern-matching
- Since it's SC code you can also programmatically fill in the description...

Basic elements of device descriptions

- Array of elements: name, event/dictionary with specifications.
- Element specification contains:
  - the element in the controller (cookies, slots, midichannels and control numbers, notes, etc.)
  - a ControlSpec (by name). Some of these are defined in MKtl, but you can also define custom ones inside the device description file (it's just SC-code!).
  - types: we've tried to classify a number of typical controls (joyAxis, button, slider, encoder, etc).
  - modes: we've tried to classify a number of typical behaviours of controls (push vs. switch, center, ..)

Chain of events

- Each control can cause a chain of actions
- They can be handwritten
- Or created from one or more templates

Chain of events

- Each control can cause a chain of actions
- They can be handwritten
- Or created from one or more templates

  - trigger
  - paged
  - threshold
  - thresholdUp, thresholdDown
  - up, down
  - thresholdZone
  - merge
  - multiclick

A simple instrument


```
MKtl.find;
a = MKtl.new( 'ngms0' );

Ndef( \sine, { Pan2.ar( Mix.new( SinOsc.ar(
  [\freq1.kr(400).lag(0.3,0.5), \freq2.kr(400).lag(0.3,0.5),
  \freq3.kr(400).lag(0.3,0.5)]  * [3/4,1,4/3] ) ) / 10,
  \pos.kr(0) ) } );
Ndef( \sine ).fadeTime = 0.1;

d = Dispatch.new( \trigger, a,
      a.elements.select{ |it| it.type == \button }.collect( _.name ) );

d.addToOutput( \btgreen, \playSynth, { Ndef(\sine).play( fadeTime: 1 ); } );
d.addToOutput( \btred, \stopSynth, { Ndef(\sine).stop(1); } );

a.addToOutput( \accX, \changeFreq, { |c| Ndef( \sine ).set( \freq1,
      [400,2000,\exponential].asSpec.map( c.value ) ) } );
a.addToOutput( \accY, \changeFreq, { |c| Ndef( \sine ).set( \freq2,
      [400,2000,\exponential].asSpec.map( c.value ) ) } );
a.addToOutput( \accZ, \changeFreq, { |c| Ndef( \sine ).set( \freq3,
      [400,2000,\exponential].asSpec.map( c.value ) ) } );
```