

Project Acronym: **MusicBricks**
Project Full Title: **Musical Building Blocks for Digital Makers and Content Creators**
Grant Agreement: **N°644871**
Project Duration: **18 months (Jan. 2015 - June 2016)**

D4.3 Advanced Processing libraries

Deliverable Status: **Final**
File Name: **MusicBricks_D4.3.pdf**
Due Date: **31st December 2015 (M12)**
Submission Date: **4th January 2016 (M13)**
Dissemination Level: **Public**
Task Leader: **IRCAM**

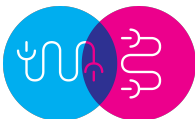
Authors: **Frédéric Bevilacqua (IRCAM), Gaël Dubus (IRCAM), Emmanuel Fléty (IRCAM), Riccardo Borghesi (IRCAM), Norbert Schnell (IRCAM), Joseph Larralde (IRCAM), Cyril Laurier (STROMATOLITE), Jordi Janer (UPF-MTG), Thomas Lidy (TUW)**



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n°644871

D4.3 - Final release of API ■ December 2015 ■ IRCAM

The MusicBricks project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n°644871



The MusicBricks project consortium is composed of:

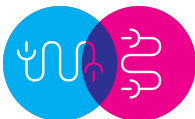
SO	Sigma Orionis	France
STROMATOLITE	Stromatolite Ltd	United Kingdom
IRCAM	Institut de Recherche et de Coordination Acoustique Musique	France
UPF	Universitat Pompeu Fabra	Spain
Fraunhofer	Fraunhofer-Gesellschaft zur Foerderung der Angewandten Forschung E.V	Germany
TU WIEN	Technische Universitaet Wien	Austria

Disclaimer

All intellectual property rights are owned by the MusicBricks consortium members and are protected by the applicable laws. Except where otherwise specified, all document contents are: "©MusicBricks Project - All rights reserved". Reproduction is not authorised without prior written agreement.

All MusicBricks consortium members have agreed to full publication of this document. The commercial use of any information contained in this document may require a license from the owner of that information.

All MusicBricks consortium members are also committed to publish accurate and up to date information and take the greatest care to do so. However, the MusicBricks consortium members cannot accept liability for any inaccuracies or omissions nor do they accept liability for any direct, indirect, special, consequential or other losses or damages of any kind arising out of the use of this information.



Revision Control

Version	Author	Date	Status
0.1	Frederic Bevilacqua (IRCAM), Gaël Dubus (IRCAM), Emmanuel Fléty (IRCAM)	November 17, 2015	Initial Draft
0.2	Frederic Bevilacqua (IRCAM), Gaël Dubus (IRCAM), Emmanuel Fléty (IRCAM), Jordi Janer (UPF), Thomas Lidy (TUW)	December 14, 2015	Revised Table of Content
0.3	Cyril Laurier (STROMATOLITE)	December 23, 2015	Internal Quality Check
0.4	Cyril Laurier (STROMATOLITE)	December 28, 2015	Review
0.5	Frédéric Bevilacqua (IRCAM)	January 3, 2016	Final corrections
1.0	Marta Arniani (SO)	January 4, 2016	Final draft reviewed and submission to the EC

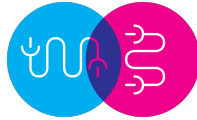
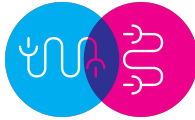


Table of Contents

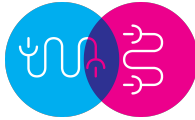
1	Introduction	6
1.1	Context and objectives.....	6
2	Updating Tangible User Interfaces and software	6
3	R-IoT description	8
3.1	Improvement and readiness level:.....	8
3.2	Final Specifications.....	8
4	Architectures for motion data processing	9
5	R-IoT firmware	10
5.1	Basic Firmware and data fusion.....	11
5.1.1	Description	11
5.1.2	Calibration and Data fusion.....	11
5.1.3	Basic OSC data streaming and extension.....	11
5.2	Motion analysis	11
5.2.1	Acc_Intensity.....	11
5.2.2	Gyr_Intensity.....	12
5.2.3	Still	13
5.2.4	Spin	13
5.2.5	Freefall.....	14
5.2.6	Shake.....	14
5.2.7	Kick.....	15
5.2.8	General methods used by several modules (feature.h)	17
6	Advanced processing and visualization.....	17
6.1	Metaphors/playing techniques	17
6.2	Processing and Visualization	18
6.2.1	vecdisplay	18
6.2.2	fftspectrum	19
6.2.3	waveletspectrum	21
6.3	Gesture Recognition and Mapping	22
6.3.1	Movement qualities recognition based on the wavelet analysis	22
6.3.2	Gestures or postures recognition using the R-IoT sensor	22
6.4	Demo: Example of R-IoT used with other API.....	23
6.4.1	HandsFreesoundMachine	23
6.5	Other applications with Graphical User Interfaces (GUIs).....	24
7	Conclusions.....	24
8	Appendix A - Principles for embedded motion processing	
9	Appendix B - Adapting the motion processing to different framerate	



Executive summary

The present document is a deliverable of the MusicBricks project, funded by the European Commission's Directorate-General for Communications Networks, Content & Technology (DG CONNECT), under its Horizon 2020 research and innovation programme.

We present here the description of the latest part of the processing and visualization tools designed for musical Tangible User Interfaces (TUIs) of Work Package 4. The general aim is to provide user with tools to facilitate the use of Tangible User Interfaces, and in particular the wireless inertial measurement units developed, such as the R-IoT developed by Ircam. This deliverable describes the general possible architectures of the Musical TUIs processing library. In particular, this description completes the previous deliverable (D4.2) by describing how the motion processing library presented in DE4.2 can also be embedded in the R-IoT device (or other microcontroller). We also present new advanced processing and visualization tools. Different examples illustrate how these tools can be used for smart mapping using machine learning for gesture recognition. Overall, all the technologies of WP4 have been upgraded from TRL4 to TRL5/6 and can be combined with the APIs of WP3 to create new applications.



1 Introduction

We present here processing and visualization tools designed for musical Tangible User Interfaces (TUIs). These components correspond to elaborated versions of the basic concepts and early developments described in the documents D4.1 and D4.2.

1.1 Context and objectives

Concerning Tangible User Interfaces (TUIs), we particularly focus on movement sensing based on wireless inertial measurement units (IMU), which requirements were described in D4.1. An example of such device is the Ircam wireless motion sensor called R-IoT that has been largely used at MusicBricks events and incubations. This device has been improved and duplicated for the MusicBricks events, which allowed us to validate its current version at TRL6 level (see description in section 3).

Based on user feedbacks at MusicBricks events and incubation, we have also extended the software tools to be used with such Tangible User Interfaces (TUIs) (see the overview in section 2). In particular, we developed two types of software tools that can be used with the R-IoT or similar devices.


First, we present an ensemble of modular processing components that can be embedded in R-IoT (or compatible Arduino microcontrollers). The implemented functionalities follow the ones that were first developed in the Max environment (Cycling' 74) and described in D4.2. This choice has been motivated by feedback during the MusicBricks hackathons and workshops, showing these movement descriptors would gain to be also implemented in the wireless device. This allows for an easier integration in different platforms and software. This development corresponds thus to upgrading our initial motion analysis library from TRL4 to TRL6 as planned in the description of work (DoA), and corresponds to the necessary preparation towards the Industry Testbeds. This library is described in section 5.

Second, as planned in the DoA, we developed advanced movement processing and visualization tools (referred as TUI Smart Mapping and GUI interfaces). We released new max objects (external and abstraction) and applications examples that are described in section 6.

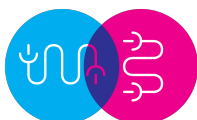
Note that all partners implied in WP4 (IRCAM, UPF-MTG, TUW, STROMATOLITE) have also built projects and examples with the R-IoT interfaces and software. We also summarize these developments in this document (which were also reported in D3.3).

2 Updating Tangible User Interfaces and software

The development of WP4 on Musical Tangible User Interfaces has been presented at the MusicBricks events as a set of tools called **Gesture Sensors for Music Performance**. As shown below with its specific logo, the WP4 technologies complement the different APIs described in WP3:

	Gesture Sensors for Music Performance <ul style="list-style-type: none">• R-IoT wireless motion sensors for building musical tangible interfaces and instruments• Real-time motion analysis and visualization to easily map sensor data to musical processes
---	--

We summarize in Table 1 (similarly to the table of section 4 in D3.1) in more details the different components that have been developed and/or improved for MusicBricks, reported in this deliverable (D4.3) and the previous one



(D4.2). These “bricks” have been used to build example applications (including also the APIs of the other partners, as described in section 6). The TRL improvements are detailed in the sections 4-6.

Tools	Platform	Programming Environment	Real-Time Usage	Links for Installation and Usage	Provider	License	Deliverable
R-IoT wireless sensors (hardware)	Any platform connected to a Wi-Fi router	Any software with OSC	yes	http://ismm.ircam.fr/devices/	IRCAM Provided at MusicBricks events and for incubation	TBD	D4.2 and D4.3
Motion Analysis Low-level descriptors	WinDoAs/OSX	Max	yes	https://github.com/Ircam-RnD/RIoT/tree/master/max	IRCAM	free Max patches	D4.2/D4.3
	WinDoAs/OSX	IDE: Energia Arduino (C code) to embed processing in microcontrollers	yes	https://github.com/Ircam-RnD/RIoT/tree/master/energia	IRCAM	GPL Part of the code can be also released as close source for commercial applications	D4.3
Advanced Motion Analysis and Visualization	WinDoAs/OSX	Max	yes	http://forumnet.ircam.fr/product/gesture-sound-en/ http://forumnet.ircam.fr/product/mubu-en/ Some externals are not released yet, but beta versions were provided at MusicBricks events	IRCAM	Forumnet IRCAM	D4.2/ D4.3

D4.3 - Final release of API ■ December 2015 ■ IRCAM

The MusicBricks project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement n°644871



3 R-IoT description

Following closely the requirement that were set in D4.1, the R-IoT module allows for sensing movement, processing and wireless transmission. The R-IoT embeds a ST Microelectronics¹ 9-axis sensor with 3 accelerometers, 3 gyroscopes and 3 magnetometers, all 16 bit. The data are sent wirelessly (Wi-Fi) using the OSC protocol². The framerate is adjustable, and was found reliable at 200Hz, which guarantees a low enough latency for musical applications (as specified in D4.1).

The core of the board is a Texas Instrument Wi-Fi module with a 32 bit Cortex ARM processor that execute the program and deals with the Ethernet / WAN stack. It is compatible with TI's Code Composer and with Energia³, a port of the Arduino⁴ environment for TI processors.

3.1 Improvement and readiness level:

In the MusicBricks project, this board has been updated and improved:

- Easy Configuration of the R-IoT by accessing a web page hosted by the module where you can configure its network behaviour & parameters
- Update of the USB micro serial port based on FTDI R232. The R-IoT can be programmed using the Energia IDE using OSX or WinDoAs (see section 5 for the description of the firmware).
- On line documentation, accessible at <http://ismm.ircam.fr/devices/>
- Testing and evaluation have been carried on in the MusicBricks hackathons and during incubation projects.
- External duplication has been successively carried on by an electronics manufacturer

Overall, the TRL level have been improved from TRL4 to TRL6.

3.2 Final Specifications

- 34 x 23.5 mm – 7mm thick
- CC3200 processor – 80 MHz – 32 bits – 256 kB of RAM – 80 mA while transmitting Wi-Fi.
- 2.4 GHz Wi-Fi – Open Sound Control Oriented
- 9 DoF motion sensor LSM9DSO – 3D accel + gyro + magneto
- On-board web server for configuration
- On-board general purpose tactile switch
- 2 GPIO + 2 analog inputs (or GPIO) exported
- I2C bus SCL & SDA pins exported
- On-board li-ion / li-po charger via μ USB
- At least 6 hours of runtime with a 16340 li-ion cell
- Battery voltage sample and report via Wi-Fi
- USB UART for serial port debugging and configuration

¹ http://www.st.com/web/en/catalog/sense_power/FM89

² <http://opensoundcontrol.org/>

³ <http://energia.nu/>

⁴ <http://www.arduino.cc/>

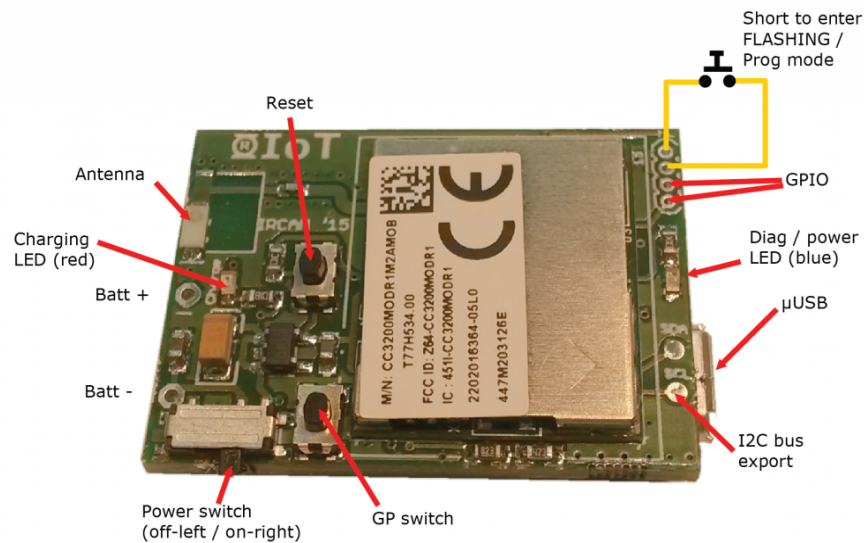
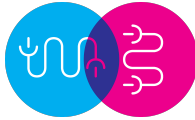


Figure 1a. Picture of the R-IoT wireless motion sensing. The 3D accelerometers, gyroscopes, magnetometers are located in the back.

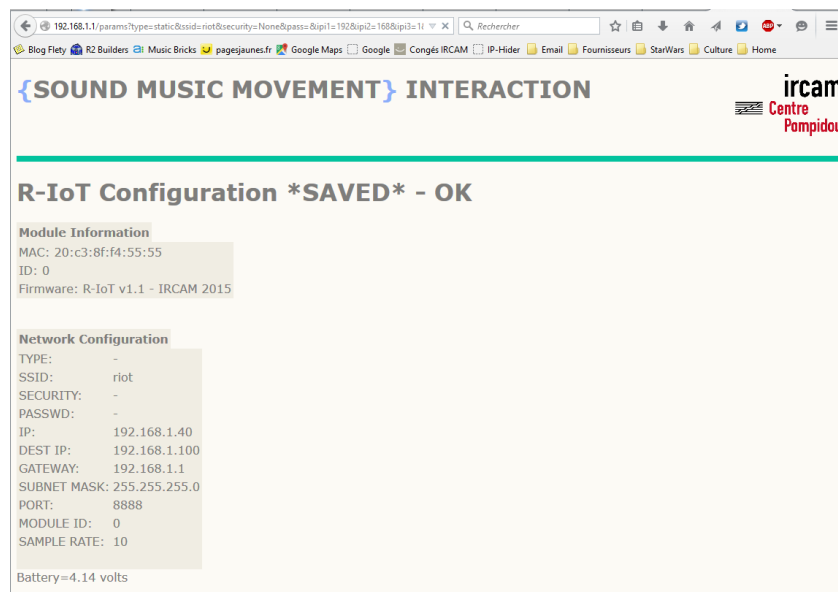


Figure 1b. Configuration webpage of the R-IoT

4 Architectures for motion data processing

The figures below the two possible options for processing the data streamed from wireless motion sensors such as the R-IoT (described in section 3). In figures 2a and 2b, we show the two complementary options for calibration and processing: implemented on the computer side and/or on the sensor side (embedded processing at the microcontroller level).

D4.3 - Final release of API ■ December 2015 ■ IRCAM

The MusicBricks project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n°644871

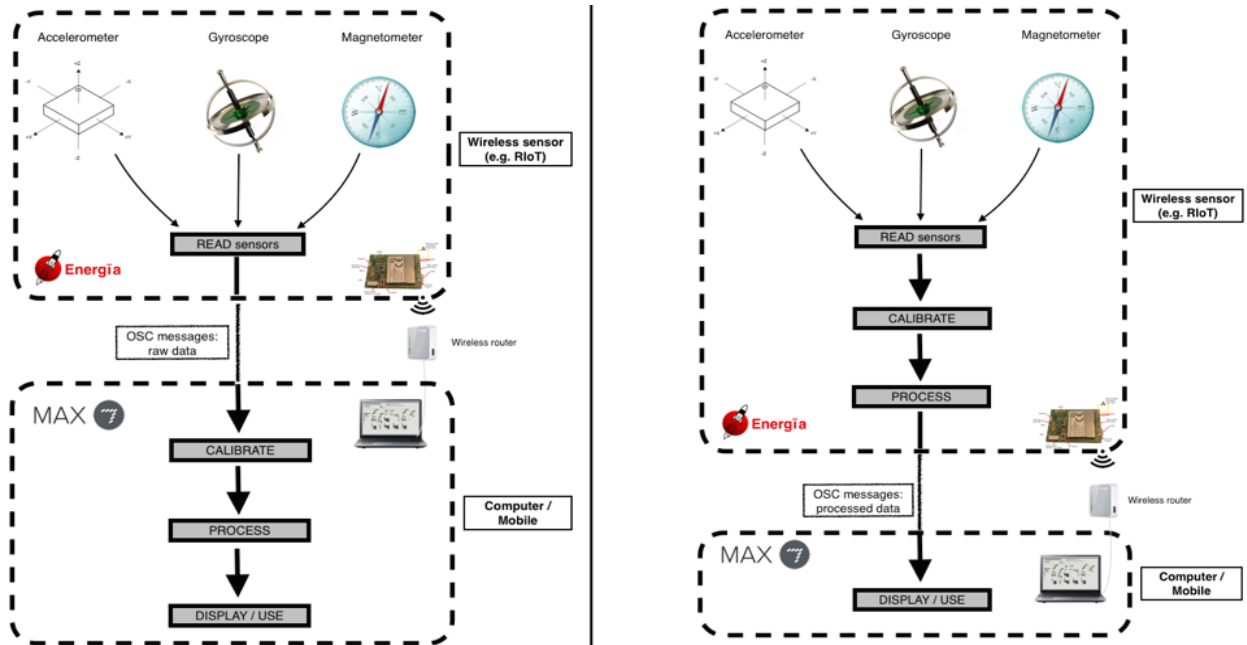
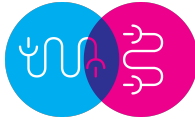


Figure 2a (left) and Figure 12b (right). Data processing architecture using wireless sensor. The wireless sensor and computer are represented by the dotted rectangles. a) The calibration and processing is performed on the computer side. b) In this case, the calibration and processing is performed on the sensor side (embedded processing at the microcontroller level).

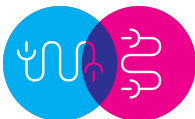
In the previous deliverable (D4.2), we reported on the motion processing that was only implemented on the computer side (in the Max environment), corresponding thus to the case of Figure 2a (left). The Max version of the motion processing library has been improved to include the data sampling rate as parameters, as described in Appendix B.

In this deliverable D4.3, we report first on an equivalent motion processing library that has been ported to embedded processing on the sensor side, corresponding thus to the case of Figure 2b (right). Globally, the porting of the original Max library to C code represents a clear improvement of the readiness level for future applications. This is described in section 5.

Second, we also report on advanced processing and visualization that could not be implemented at the sensor level due to the limited processing power of the embedded microcontroller and the absence of visualization capabilities. In this case, only the case of Figure 2a is viable. This is described in section 6.

5 R-IoT firmware

This section presents the different firmwares that can be built by assembling the basic firmware with different motion analysis functions. The code is modular in order to let users to optimize memory and CPU depending on the applications. These motion analysis functions are based on the Musical TUIs processing library that was described in D4.2 (mainly from the section *metaphors/playing techniques*). Appendix A describes in more details the general principles we used to design embedded motion analysis.



5.1 Basic Firmware and data fusion

5.1.1 Description

The basic firmware contains all the elements necessary for acquiring the sensor data (9 DoF motion sensor LSM9DSO) by the CC3200 processor and for enabling the OSC data stream.

5.1.2 Calibration and Data fusion

Calibration of the sensors and the Data fusion allowing to compute the quaternions and Euler angles are also provided. The data fusion is implemented using the open-source code provided by Sebastian Madgwick⁵

5.1.3 Basic OSC data streaming and extension

In the basic firmware implementation, the following output parameters are streamed as OSC packets

- `/raw <11 int data list>` that contains the battery voltage, the GP switch state and the raw 9 channels from the motion sensor (all 16 bit)
- `/quat <4 float data list>` that contains the quaternion computation results based on Madgwick algorithm
- `/euler <3 float data list>` that contains the euler angles computed by the module from the quaternions mentioned above
- `/analog <2 int data list>` that contains the analog inputs digital conversion

As described in section 5, other parameters and motion features can be streamed as well, if the firmware is completed by the different modules described below.

5.2 Motion analysis

5.2.1 Acc_Intensity

Description

Computes a quantity related to the “motion intensity” with respect to acceleration. The value is equal to zero when there is no movement.

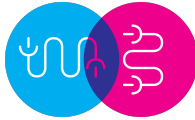
Computation

Acc_Intensity is defined recursively (see figure). For an input signal $\mathbf{s}(\mathbf{t})$ discretised as the series (\mathbf{s}_n) , the corresponding intensity (\mathbf{I}_n) is defined as :

$$\begin{cases} I_1 &= 0 \\ I_{n+1} &= aI_n + bs'_n{}^2 \end{cases}$$

where $\mathbf{s}'(\mathbf{t})$ is the time derivative of $\mathbf{s}(\mathbf{t})$ approximated via a centred finite difference of order 2, and \mathbf{a} and \mathbf{b} are parameters of the model. It can be easily shown that the general term of the series (\mathbf{I}_n) can be expressed as a function of terms of the series $\mathbf{s}'(\mathbf{t})$ as follows:

⁵ See <http://www.x-io.co.uk/open-source-imu-and-ahrs-algorithms/> for more details



$$I_n = b \sum_{i=1}^n a^{n-i} s_i'^2$$

Here $s(t)$ is the measured acceleration and $s'(t)$ is the jerk, estimated by discrete derivation (centred finite difference of order 2).

Input/variable used

a_x, a_y, a_z

Output/variable streamed

acc_intensity_norm, acc_intensity_x, acc_intensity_y, acc_intensity_z

OSC stream (Energia)

/id/acc_intensity [acc_intensity_norm, acc_intensity_x, acc_intensity_y, acc_intensity_z]

Defined variables and methods

Variables: ACC_INTENSITY_PARAM1, ACC_INTENSITY_PARAM2

Methods: lcm, delta, intensity1D

5.2.2 Gyr_Intensity

Description

Computes a quantity related to the “motion intensity” with respect to gyration. The value is equal to zero when there is no movement.

Computation

Gyr_Intensity implements the same algorithm as Acc_Intensity but with angular acceleration as input data instead of acceleration. Here $s(t)$ is the measured angular velocity and (s'_n) is the angular acceleration, estimated by discrete derivation (centred finite difference of order 2).

Input/variable used

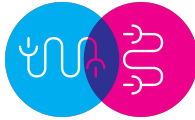
g_x, g_y, g_z

Output/variable streamed

gyr_intensity_norm, gyr_intensity_x, gyr_intensity_y, gyr_intensity_z

OSC stream (Energia)

/id/gyr_intensity [gyr_intensity_norm, gyr_intensity_x, gyr_intensity_y, gyr_intensity_z]



Defined variables and methods

Variables: GYR_INTENSITY_PARAM1, GYR_INTENSITY_PARAM2

Methods: lcm, delta, intensity1D

5.2.3 Still

Description

Detection of a state of stillness

Computation

The quantity $\|\vec{w} \wedge \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\|^2$ is computed. This measure is invariant to an equal shift in all 3 dimensions. As such shifts generally occurs in gyroscopes (IMU), this measure remains robust over time.

Input/variable used

g_x, g_y, g_z

Output/variable streamed

isStill, still_slide, gyr_norm

OSC stream (Energia)

/id/still [isStill, still_slide]

Defined variables and methods

Variables: STILL_THRESHOLD, STILL_SLIDE_FACTOR

Methods: slide, still_cross_product

5.2.4 Spin

Description

Detection of a state of spinning

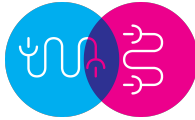
Computation

The amount of spin is estimated by the norm of the angular velocity vector, for which a threshold is set for detection of a spinning motion pattern.

Input/variable used

D4.3 - Final release of API ■ December 2015 ■ IRCAM

The MusicBricks project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n°644871



g_x, g_y, g_z

Output/variable streamed

isSpinning, spinDuration

OSC stream (Energia)

/id/spin [isSpinning, spinDuration, gyr_norm]

Defined variables and methods

Variables: SPIN_THRESHOLD

Methods: magnitude3D

5.2.5 Freefall

Description

Detection of a state of free fall

Computation

The magnitude (Euclidean norm) of the acceleration vector is computed, along with the magnitude of the angular velocity vector and the magnitude of the angular acceleration vector (obtained via discrete derivation of the angular velocity). A detected free fall can be linear (acceleration magnitude close to zero) or rotational (angular velocity magnitude greater than a threshold AND angular acceleration magnitude close to zero).

Input/variable used

$a_x, a_y, a_z, g_x, g_y, g_z$

Output/variable streamed

isFalling, fallDuration, acc_norm

OSC stream (Energia)

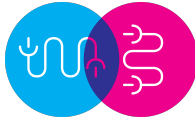
/id/freefall [acc_norm, isFalling, fallDuration]

Defined variables and methods

Variables: FREEFALL_ACC_THRESHOLD, FREEFALL_GYR_THRESHOLD, FREEFALL_GYR_DELTA_THRESHOLD

Methods: lcm, delta, magnitude3D

5.2.6 Shake



Description

Analysis of accelerometer data for quantification of shaking

Computation

The time derivative of each accelerometer stream is estimated with a centred finite difference of order 2. The percentage of time this derivative (jerk) exceeds the parameter τ within a time winDoA of size β is computed and averaged over the 3 dimensions (root mean square). The result is then smoothed to give the shaking magnitude.

Input/variable used

a_x, a_y, a_z

Output/variable streamed

shaking

OSC stream (Energia)

/id/shake [shaking]

Defined variables and methods

Variables: SHAKE_THRESHOLD, SHAKE_WINDOASIZE, SHAKE_SLIDE_FACTOR

Methods: lcm, delta, magnitude3D, slide

5.2.7 Kick

Description

Detection of a “kick”, i.e. sudden movement (referred to as “strike detection” in DE4.1)

Computation

A median filter is applied to the intensity given in input. The difference between the input and the filtered value triggers a kick detection when exceeding a threshold. A speedgate implements the minimum delay between two kick detections.

Input/variable used

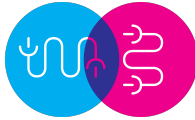
acc_intensity_norm

Output/variable streamed

isKicking, kick_intensity

OSC stream (Energia)

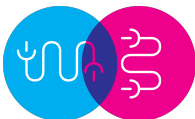
/id/kick [kick_intensity, isKicking]



Defined variables and methods

(In addition to the ones defined in Acc_Intensity)

Variables: KICK_THRESHOLD, KICK_SPEEDGATE, KICK_MEDIAN_FILTERSIZE



5.2.8 General methods used by several modules (feature.h)

- **Least common multiple**
`int lcm(int a, int b)`
- **Discrete derivation with centered finite difference of order 2**
`float delta(float previous, float next, float dt);`
- **Euclidean norm in 3D**
`float magnitude3D(float x, float y, float z);`
- **One-pole filter**
`float slide(float previous_slide, float current_value, float slide_factor);`
- **Computation of the intensity along/around one axis**
`float intensity1D(float xnext, float xprev, float intensityprev, float param1, float param2, float dt);`
- **Cross product between angular velocity vector and (1 1 1) for the module 'still'**
`float still_cross_product(float x, float y, float z);`
-

6 Advanced processing and visualization

The Max objects we report here complete the Musical TUIs processing library presented in D4.2.

6.1 Metaphors/playing techniques

This section was fully presented in D4.2. Nevertheless, we present here improvement and new features.

First, the functions presented in D4.2 for motion processing have been updated to take into account the framerate as parameters, as shown in Figure 3. This allows to obtain output results that are independent of the data sampling rate (considering framerate below the Nyquist rate), which greatly facilitate the porting and adaptation of these functions to different context. In appendix B, we describe all the changes that must be performed to the initial version.

Second, new functions or implementations have been added for data stream processing in the *pipo* plugin format (<http://ismm.ircam.fr/pipo-sdk-release-v0-1/>, integrated in the MuBu package⁶)

- *pipo.biquad*, which implements a biquad filter (as in the max object *filtering*)
- *pipo.finitedef*, which implements finite difference computation on a data stream, allowing for example for obtaining in a single step 1st and 2nd order derivatives.
- *pipo.lpc*, which allows to compute linear predictive coding coefficient on an audio stream or an sensor data stream.

⁶ <http://forumnet.ircam.fr/product/mubu-en/>

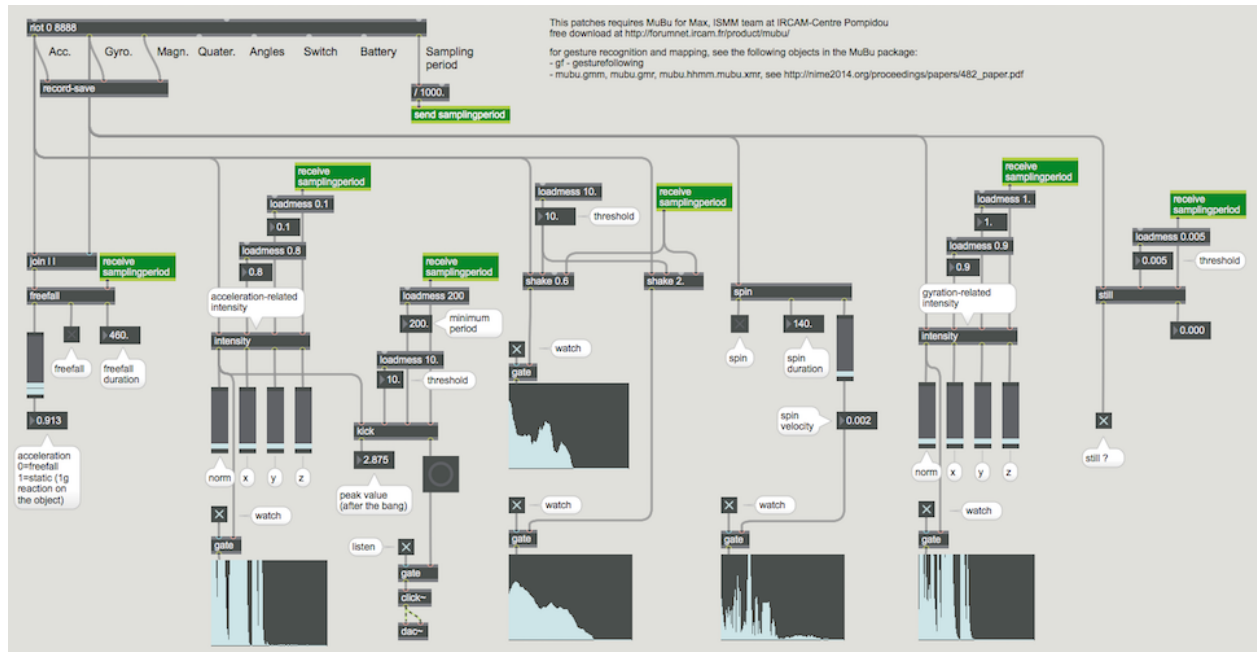
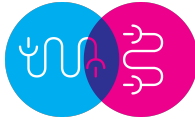


Figure 3. This figure illustrates all the objects of the Max example (corresponding to metaphor/playing techniques) that have been updated to take into account the framerate (in green).

6.2 Processing and Visualization

6.2.1 vecdisplay

description

This max external allows for displaying vector and spectrum, with time history displayed on a specific winDoA

input

[1] input data <list of float>

main parameters

display parameters: background, vector and history colors, shape, refresh rate

history size <int> : number of past vector shown in the history winDoA (in frames)

output

none

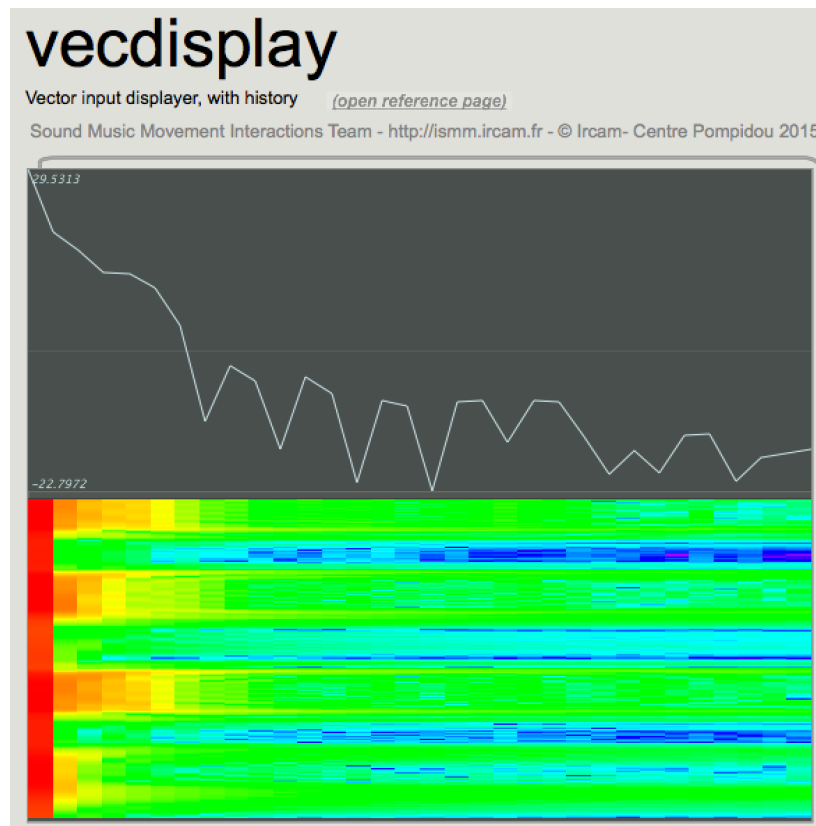
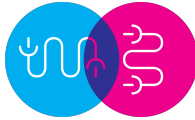


Figure 4. Screenshot of the vecdisplay

6.2.2 fftspectrum

description

This max external allows for:

- computing the short fourier transform of a sensor data stream
- displaying the spectrum on a specific frequency range
- applying a filter in the fourier domain by drawing a function on the display

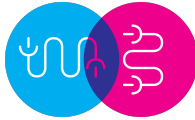
This is useful to design frequency bands that match the frequency content of specific movement patterns: the obtained movement features corresponding to specific movement frequencies.

input

[1] input data <float or list of float> (regularly sampled)

main parameters

short-term fourier transform: fftsize, hopesize, framerate, fft mode (power, complex, logpower, magnitude), inputwinDoA (hamm, hamming, blackman, blackmanharris, since, none)



display parameters: background, vector and history colors, shape, refresh rate

history size <int> : number of past vector shown in the history winDoA (in frames)

output

spectrum, filtered signal

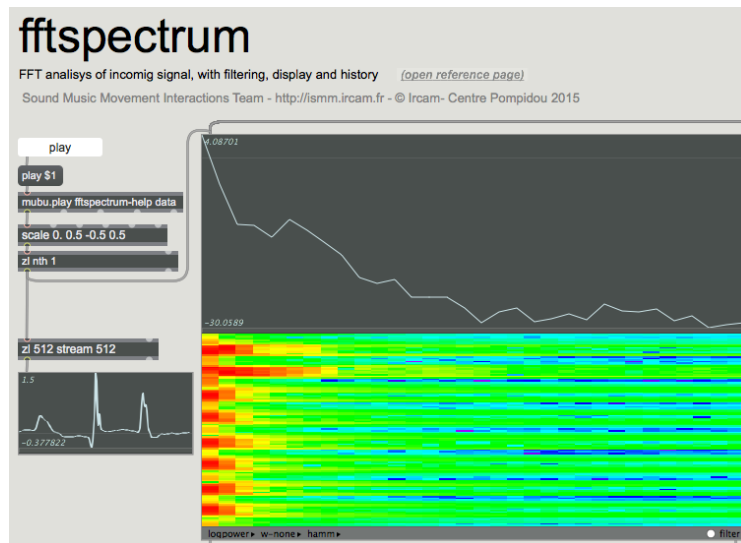
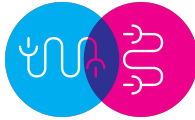


Figure 5. Screenshot of the *fftspectrum* Max object used for real-time sensor analysis



6.2.3 waveletspectrum

description

This max external allows for:

- computing the continuous wavelet transform of a sensor data stream (Morlet wavelet). The computation is based on the code developed at IRCAM by Jules François, which has been implemented here in a single object with the visualization
- displaying the spectrum (referred to the “scalogram”)
- applying a filter in the frequency domain by drawing a function on the display

This is useful to design frequency bands that match the frequency content of specific movement patterns. The object is complementary to the *fftspectrum* object.

input

[1] input data <float or list of float> (regularly sampled)

main parameters

wavelet transform: bands per octave min and max frequencies, ω_0 , delay.

display parameters: background, vector and history colors, shape, refresh rate

history size <int> : number of past vector shown in the history winDoA (in frames)

output

spectrum, filtered signal

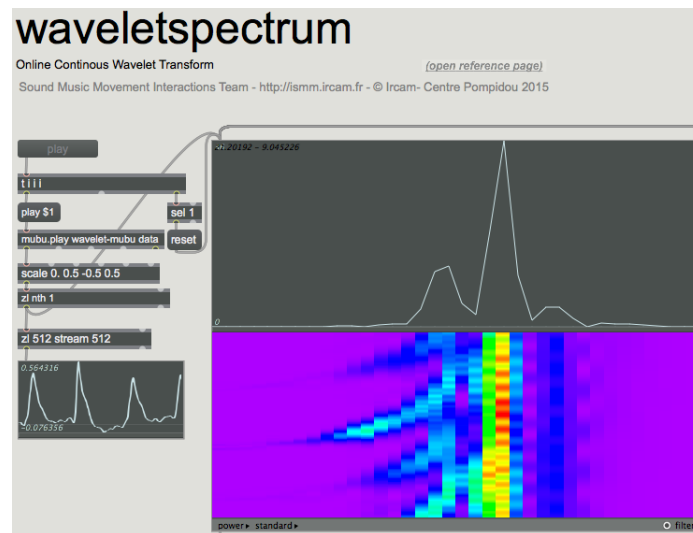
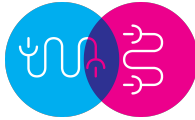


Figure 6. Screenshot of the *waveletspectrum* Max object used for real-time sensor analysis



6.3 Gesture Recognition and Mapping

All descriptors of the Musical TUIs processing library, including the wavelet and fft analysis we presented above, can be fed into the different gesture recognition and mapping procedures. In particular the free MuBu for Max library was made available at the MusicBricks hackathon and used by several participants. We present below two examples.

6.3.1 Movement qualities recognition based on the wavelet analysis

The following patch (*mapping_with_wavelet.maxpat*) illustrates how the object *waveletspectrum* can be used in conjunction with R-IoT wireless sensors to enable gesture recognition. The wavelet analysis is performed on the three-accelerometer axis and input to the *gesture follower* (gf), which allows for recognition prerecorded examples. In this case, the wavelet analysis is particularly well suited to recognise *movement qualities*, in contrast to precise spatial trajectories. Importantly, the likelihood values related to each movement can be used as continuous parameters that can be mapped to any continuous sound processing.

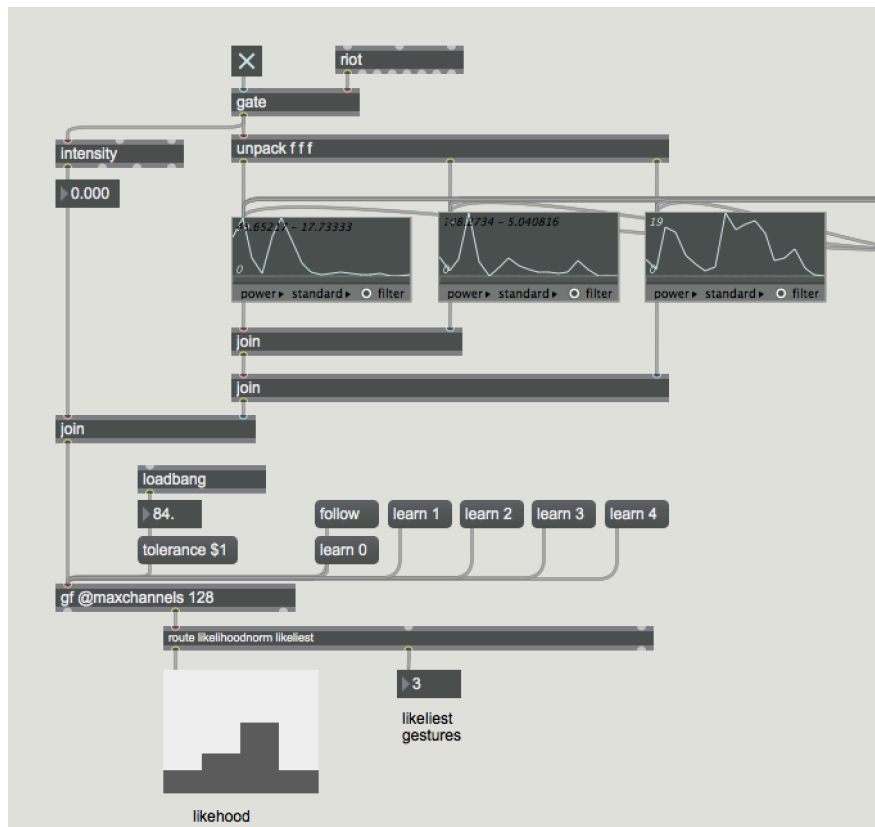


Figure 7. Screenshot of the *waveletspectrum* Max object used for movement qualities recognition

6.3.2 Gestures or postures recognition using the R-IoT sensor

The following patch (*riot-gmm.maxpat*) illustrates how the R-IoT sensor can be used to perform gesture recognition. Two quaternion parameters are chosen to characterize the orientation of the R-IoT and are input in the *mubu.gmm* object (part of the IRCAM MuBu library that was made available at the hackathon). This is an example of prototypes that was initiated at a MusicBricks hackathon to provide advanced movement processing.

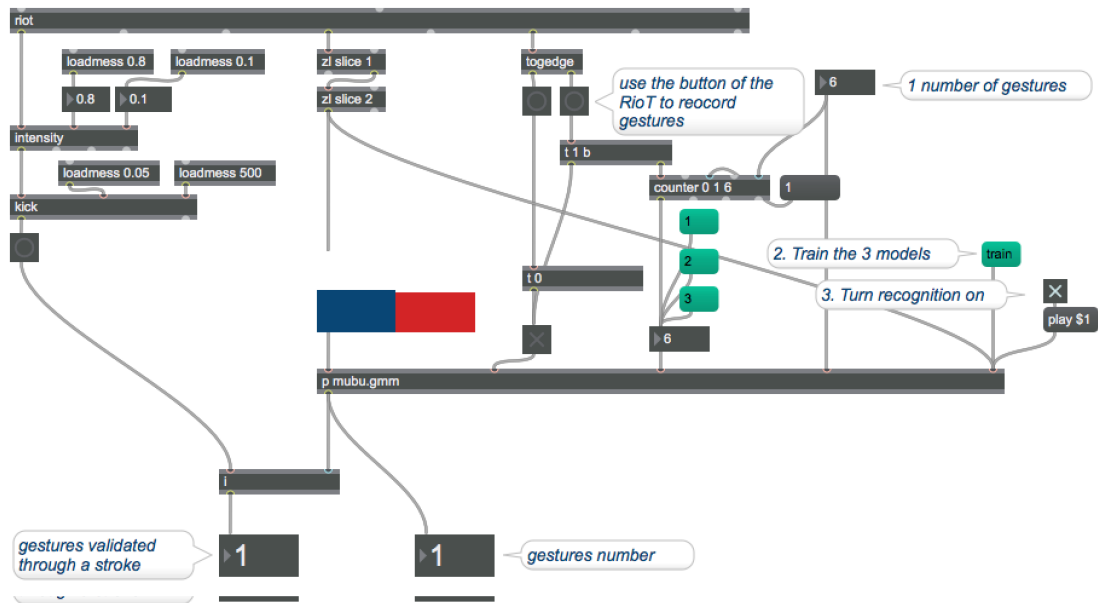
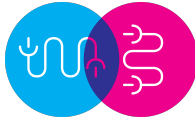


Figure 8. Screenshot of the *riot-recognition* Max object used for gesture recognition

6.4 Demo: Example of R-IoT used with other API

6.4.1 HandsFreesoundMachine

This demo created by UPF-MTG is fully described in D3.3. We just recall here some points that illustrate well how the technologies developed in WP4 can be used with the APIs of WP3.

In the *HandsFreesoundMachine* application, the R-IoT is used to detect head movements. This was directly implemented using “kick” detection of the R-IoT firmware (see section 5). This allows for setting a tempo and selecting steps of a drum machine. The sound samples are retrieved directly from Freesound using the FS API v2, and a voice interface is used for specifying the search query (using Google API).

This demonstration also illustrates how the R-IoT OSC messages can be used in a web application. In this case, a web server built using Python’s Flask package establishes a web sockets connection with the client browser, which includes an Open Sound Control (OSC) module. This allows for processing OSC messages received from the R-IoT sensor and sends them to the web browser using the web sockets connection.

All source code for the web application is available online in this Github repository:

<https://github.com/MTG/hands-free-sound-machine>

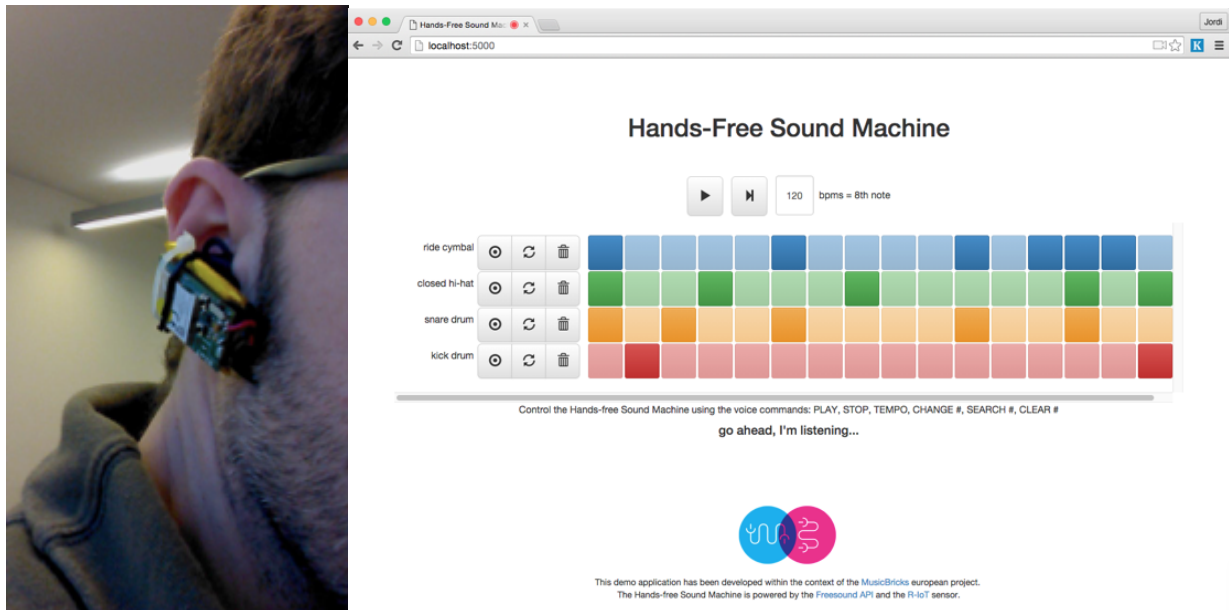
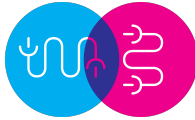


Figure 9: Left: R-IoT used with a Bluetooth headset Right: screenshot of the web application (by UPF-MTG)

6.5 Other applications with Graphical User Interfaces (GUIs)

Other applications with advanced Graphical User Interfaces have been carried on in MusicBricks. For example, the application Sonarflow developed by TU Wien / Spectralmind is a visual music discovery app for iPad, iPhone and Android. It features a slick UI for browsing music by zooming into a colourful world of bubbles which represent genres, artists or moods and allows discovering new music online from various sources. Through MusicBricks, it is now available open-source for hacking and extending it. Sonarflow is fully described in D3.3 in section 3.8.

7 Conclusions

This deliverable D4.3, along with the deliverables D4.1 and D4.2, conclude the work of WP4 on Tangible User Interfaces. This work, and their deployment in the MusicBricks events and incubation projects, allows us to improve significantly our available technologies, such as the R-IoT interface and software for musical interactions and performance. As planned in the Description of Work, these technologies have been ported from TRL4 to TRL5/TRL6 for the creative test beds, and are ready to be implemented in Industry test beds (WP6).

Appendix A

In this section, we introduce a few principles used in the context of real-time embedded programming and how they have been implemented in the data processing modules. The firmware running on the RIoT board is compiled and installed with the Energia software (similar to Arduino framework), including an integrated development environment based on the Processing language.

A.1 Elementary blocks

A few concepts are of common use in the specific context of embedded real-time programming. We describe below elementary blocks used to tackle some of these issues.

A.1.1 Three dimensions

Since all three spatial dimensions are treated in the same way in many of our data processing modules, it appears more convenient to use a 3-dimensional array whenever variables or data structures correspond to spatial dimensions. In this way, all computations involving them are implemented using very similar code, and are only differing by the array index. This should not be less efficient than using variables with different names, neither in terms of computational time nor for memory use.

A.1.2 Measuring a duration

Another common concern is to measure the duration corresponding to a given state (e. g. falling, spinning...) Since the state can be current, the duration may be increasing at each step, depending on the considered state. The idea is to increment the duration by the sample period value whenever the corresponding state is active. States are implemented as Boolean quantities (e. g. `isFalling`.) The following code is generic:

```

if (Condition){
    //Condition for state activity is satisfied
    if (isActive == 0){
        //State was previously inactive, becomes active
        isActive = 1;
        //Current time is start time
        FirstTimeActive = millis();
        ...
    }
    else {
        //State was already active
        ...
    }
    //Update duration
    StateDuration = millis() - FirstTimeActive;
    ...
}
else {
    // State is inactive
    isActive = 0;
    ...
}

```

The duration is updated whenever the state is active, which means that it increases until the state becomes inactive. An alternative option (not implemented) is to update the duration during the transition from an active to an inactive state, leading to a single update of the duration, its value being then the total duration of the latest terminated active state. Some additional conditions can be added, e.g. to include a speedgate as in the module `kick` to avoid unwanted multiple detections of a discrete event (cf. Section A.2.6).

A.1.3 Storage and use of previous values

One very common issue in real-time programming is the management of previous values which have to be stored for some time to be used in the near future. An efficient way to manage previous values is to implement a ring buffer, which minimizes both memory use and computation time (values are not moved but inserted and read in place).

In the present implementation, this is realized by storing the concerned values in an array, the size of which is equal to the number of values to be stored. For example, if the computation requires values measured at times t_n (current time), t_{n-1} (previous sample) and t_{n-2} , the size of the array should be 3. Moving forward in time by one step, the required values would be the ones measured at t_{n+1} , t_n and t_{n-1} . Overwriting the value measured at t_{n-2} with the one measured at t_{n+1} minimizes memory use, while not moving the other values minimizes computation time.

In practice, a *ring buffer* (or *circular buffer*) can be implemented by manipulating the read/write access index of a regular array. To simulate the ring, any index calculation has to be performed modulo the size of the array. Instead of shifting all values in the array at each new time step, say to the left, and then writing the newly read value at the right end of the array, one only needs to increment an offset giving the position

of the oldest sample and calculate the other indices relatively to this offset. In order to avoid overflow errors when having the system running for a long time, the offset can itself be computed modulo the size of the array. Since several ring buffers of different size are usually implemented, we chose to use a single offset for all of them, called `LoopIndex`, which is computed modulo the least common multiple of all ring buffer sizes (called `LoopIndexPeriod`).

A.1.4 Discrete derivation

Discrete derivation was initially performed by linear interpolation over n samples ($n = 3$ by default). This not only introduces a delay by $n/2$ samples, it is also diffusive and inaccurate, especially for a large filter size or when derivating several times successively. The *finite difference method* can be used instead, performing discrete derivation with the possibility of removing completely the delay due to the method (using a *backward* method). It can also be used to estimate derivatives of higher order with fair accuracy. Various aspects of the finite difference method are developed in [1], but only the 1D formulas lie within the scope of this work. We summarize the principle of the finite difference method below.

Assuming that a mathematical function f is differentiable at a sufficiently high order ($\geq m$), the Taylor series expansion of f at the neighbourhood of any point x can be written as:

$$f(x+h) = f(x) + \sum_{k=1}^m \frac{h^k}{k!} f^{(k)}(x) + o(h^m) \quad (\text{A.1})$$

In the case of a time derivative, h (assumed to be small) is usually the sampling period. This formula provides an estimate of f at the point $x+h$ using m orders of derivation at x that are assumed to be known. It can also be written at other neighbouring points such as $x-h$, $x+2h$, $x-3h$, etc. The finite difference method reverses the problem: assuming that the values of f are known at a certain number of points, it uses the Taylor expansion formula at each of these points to define a set of linear equations that can be combined with each other to estimate the desired $f^{(k)}(x)$ (which are not known).

As an illustration, we estimate the first derivative using a backward method over 3 points (x ; $x-h$; $x-2h$). The two following equations, resulting from the Taylor expansion formula, are used:

$$f(x-h) - f(x) = -hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f^{(3)}(x) + o(h^3) \quad (\text{A.2})$$

$$f(x-2h) - f(x) = -2hf'(x) + \frac{4h^2}{2}f''(x) - \frac{8h^3}{6}f^{(3)}(x) + o(h^3) \quad (\text{A.3})$$

Combining them as $-2 \times (\text{Equation A.2}) + \frac{1}{2} \times (\text{Equation A.3})$ gives the following result:

$$\frac{1}{2}f(x-2h) - 2f(x-h) + \frac{3}{2}f(x) = hf'(x) - \frac{h^3}{3}f^{(3)}(x) + o(h^3)$$

Whence:

$$\begin{aligned} \frac{1}{h} \left(\frac{1}{2}f(x-2h) - 2f(x-h) + \frac{3}{2}f(x) \right) &= f'(x) - \frac{h^2}{3}f^{(3)}(x) + o(h^2) \\ &= f'(x) + O(h^2) \end{aligned}$$

This gives an estimate of the derivative of f at the point x , at an accuracy order of 2 since the exact value is asymptotically approached within an error of $O(h^2)$ when h approaches 0. In the context of the estimation of the time derivative of a discretized function, this can be written as:

$$f'_n \approx \frac{1}{\Delta t} \left(\frac{1}{2}f_{n-2} - 2f_{n-1} + \frac{1}{2}f_n \right)$$

It is clear that the accuracy order can be increased by using more points, giving several more equations allowing us to eliminate higher order derivative terms. Naturally, the same method can be used to estimate $f''(x)$ or higher order derivatives, and that an estimate of the k -th order derivative will include a factor $1/(\Delta t)^k$. As explained in [1], the finite difference method corresponds to the resolution of a linear system to find the coefficients to apply to a certain number of known samples of the function, in order to find an estimate of the derivative of desired order, by a required accuracy order.

A.1.5 Median filter

To implement a real-time median filter of size n (odd) in an efficient way, we use an array `median_values` of size n to store the n latest read samples, with the following loop invariant: the array is sorted in ascending order at the beginning of the loop (i.e. before inserting a new value). At this point, the current median is located at `median_values[(n-1)/2]`. A new value, when first inserted, should be written over the oldest sample present in the array. Finally, the new sample has to be moved to the right place, which is accomplished by swapping it with its neighbouring values as far as is required.

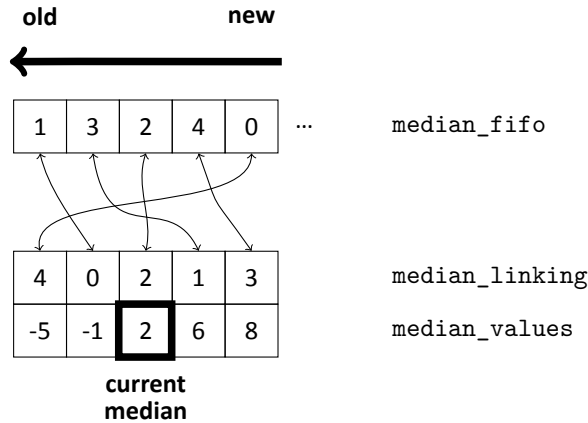


Figure A.1: Illustration of the operating principle of the real-time median filter.

In order to keep track of the order in which the values have arrived, a first-in first-out (FIFO) buffer `median_fifo` is used. As shown in Fig. A.1, its first element contains the index at which the oldest sample is located in the array `median_values`. That is, the insertion process should begin with the new value overwriting the oldest value, located in the array `median_values` at the index stored at `median_fifo[0]`. The new

value should then be moved to its correct position to ensure the loop invariant, and this position should be entered at the end of the FIFO buffer. Considering `median_fifo` as a circular buffer as described previously, this only requires to write the new location (i.e. the index of the inserted value in `median_values`) in place of the location of the oldest value before insertion (i.e., at `median_fifo[0]`) and increment the loop index.

In this process, the position of each item swapped with the new value in the array `median_values` is modified and should therefore be updated accordingly in the array `median_fifo`. To be able to do this, we chose to use a linking layer `median_linking`, allowing bidirectional communication between `median_values` and `median_fifo`. This layer consists in an array `median_linking` “tied up” to `median_values`, i.e., for all array index i , the value at `median_linking[i]` corresponds to a piece of information regarding the value at `median_values[i]`. This piece of information is the location of the index corresponding to the value `median_values[i]` in the FIFO buffer `median_fifo`. In other words, as illustrated in Fig. A.1, for all array index i we have the following property:

$$\text{median_linking}[\text{median_fifo}[i]] = \text{median_fifo}[\text{median_linking}[i]] = i.$$

Other solutions using less memory (here we use 3 arrays of size n) could be used, but would probably be more expensive in terms of computational time. For example, it is possible to avoid using `median_linking` by keeping track of the initial and final positions of the inserted value, and then perform a loop on `median_fifo` updating the values comprised between them.

A.2 Implementation of the data processing modules using elementary blocks

A.2.1 Spin

Spinning **duration** is computed as described in Section A.1.2, the active state (spinning) corresponding to either an angular velocity norm exceeding a predefined threshold value.

A.2.2 Still

The Max object `slide a a` used to smooth the output stream implements the following formula, equivalent to a onepole filter:

$$y_n = y_{n-1} + \frac{x_n - y_{n-1}}{a}$$

Since only one **previous value** is necessary, this function is not implemented with a circular buffer as described in Section A.1.3, but only using a single local variable.

A.2.3 Freefall

Fall **duration** is computed as described in Section A.1.2, the active state (falling) corresponding to an acceleration magnitude lying below a predefined threshold, or to a simultaneous condition of angular velocity magnitude below a threshold and angular acceleration magnitude above another threshold. **Discrete derivation** is performed

as explained in Section A.1.4 on gyroscope data to compute angular acceleration. Depending on the method used, a certain number of **previous values** (3 values by default) have to be stored, which is done using a circular buffer (cf. Section A.1.3). Since all steps of computation are the same in the **3 spatial dimensions**, the method mentioned in Section A.1.1 is implemented for the derivation of angular velocity.

A.2.4 Shake

The amount of shaking is basically defined as the amount of time within a given time window that the jerk (time derivative of acceleration) exceeds a predefined threshold value. As for the module `freefall`, a certain number of **previous values** (n_1) are stored to be used in the **discrete derivation** — this time of acceleration data, in a similar way for the **3 spatial dimensions**.

The considered time window is implemented as a **circular buffer** of size n_2 containing a Boolean value equal to 1 for each of the considered sample where the jerk exceeds the threshold, 0 otherwise. This is also accomplished in a similar way in each of the **3 spatial dimensions**.

Finally, the Max object `slide` used to smooth the output stream is used as in the module `still`, i.e. only using a single local variable. In this case, the congruence divisor `LoopIndexPeriod` mentioned in Section A.1.3 is the least common multiple of n_1 and n_2 .

A.2.5 Intensity

Computation of the **discrete derivative** of either accelerometer or gyroscope data is performed similarly for the **3 dimensions**. Storing **previous values** is required for this computation (as explained in Section A.1.4), but also for the integration part (a single previous value in the **3 dimensions**, treated as explained in Section A.1.3 this time).

A.2.6 Kick

The detection of a kick is built on the top of the output stream of the `intensity` module with accelerometer data given as input. Therefore, the all aspects developed in Section A.2.5 are also implemented in the module `kick`. Downstream the intensity computation, a real-time **median filter** is implemented as explained in Section A.1.5.

The **duration** of an active “kicking” state is computed as described in Section A.1.2, in order to implement a speedgate preventing the detection of multiple kicks within a given time frame. The state is considered as “active” if the difference between the current intensity and its median computed over a given time window exceeds a predefined threshold value. The generic code used to measure duration is adapted to include the speedgate, setting the state back to “inactive” only after a certain duration. The “intensity” of a kick given in output is the maximum value of the input intensity while the state is active.

A.3 Numerical parameters

The following thresholds and numerical parameters are used in the current implementation of the data processing modules.

For the module `spin`

- detection threshold on angular velocity, set to 200 deg/s

For the module `still`

- detection threshold on the computed cross-product quantity, set to 5000 (deg/s)^2
- slide parameter, set to 5

For the module `freefall`

- detection threshold on acceleration, set to 0.15 g
- detection threshold on angular velocity, set to 750 deg/s
- detection threshold on angular acceleration, set to 40 deg/s^2

For the module `shake`

- threshold on jerk, set to 0.1 g/s
- window size, set to 200 samples
- slide parameter, set to 10

For the module `intensity`

- model parameter a , set to 0.8 for acceleration-related intensity or 0.9 for gyration-related intensity
- model parameter b , set to 0.1 for acceleration-related intensity or 1.0 for gyration-related intensity

For the module `kick`

- detection threshold on the difference between intensity and its median, set to 0.01
- median filter size, set to 9
- speedgate duration, set to 200 ms

Bibliography

- [1] Bengt Fornberg. 2011. Finite difference method. *Scholarpedia*, 6(10):9685.

Appendix B

In this section we show how the different data processing modules depend on the sample rate of the input data, and how these dependencies can be overcome.

B.1 Spin

B.1.1 Analysis

Since only instantaneous values are used (at t_n), there is no dependency.

B.1.2 Proposed modifications

None. No output stream value is modified.

B.2 Still

B.2.1 Analysis

The only dependency found is due to the **smoothing process** right before outputting the stream. This is performed by a `slide` object in Max, equivalent to a onepole filter. The formula for the Max object `slide α α` is:

$$y_n = y_{n-1} + \frac{x_n - y_{n-1}}{\alpha}$$

The formula for a onepole filter is commonly written as:

$$y_n = ax_n + (1 - a)y_{n-1}$$

The two formulas are equivalent for $\alpha = 1/a$.

B.2.2 Proposed modifications

The cutoff frequency f_c of the corresponding onepole filter should be invariant, which determines the filter entirely. The following relationship is verified for a onepole filter:

$$a = \sin\left(2\pi\frac{f_c}{f_s}\right)$$

where f_s is the sampling frequency.

In the original version of the module, the default parameters are $f_s = 100$ Hz and $\alpha = 5$ (i.e. $a = 0.2$), hence the original cutoff frequency:

$$f_c = 100 \frac{\arcsin(0.2)}{2\pi} \approx 3.2 \text{ Hz}$$

To keep f_c constant independently of the sampling rate, keep `slide α α` but modify α such that:

$$\alpha = \frac{1}{\sin\left(2\pi \frac{f_c}{f_s}\right)} = \frac{1}{\sin\left(\frac{100 \arcsin(0.2)}{f_s}\right)}$$

No output stream value is modified.

B.3 Freefall

B.3.1 Analysis

For the detection of linear free fall, only instantaneous values are used, therefore there is no dependency.

For the detection of rotational free fall (free spin), a dependency is introduced by the **time derivation** of angular velocity (angular acceleration): the value of the derivative is computed with `pipo_delta`, which corresponds to an approximation by linear interpolation using previous values over a stencil of size k ($k = 3$ in the original version). This means that for a sample arriving at time t_n , samples at t_{n-k+1} , ..., t_{n-1} , t_n are used to compute the derivative. However, only the combination of samples over this stencil is performed (multiplying them by relevant coefficients), while the real derivative value — which should be independent of the sample rate — would require to divide this quantity by the sample period.

B.3.2 Proposed modifications

Use the real value of the derivative (i.e. divide by the sample period Δt), if possible using a better method than linear interpolation, e.g. the finite difference method implemented in the module `pipo_finitedif`, which is more accurate and does not introduce any latency). Update the concerned threshold values accordingly.

The following threshold should be updated: in the free spin detection subpatch, the norm of the derivative of the angular velocity should be lower than 4.0 rad.s^{-2} (in the original version with an assumed sample rate of 100 Hz, the threshold value is 0.04 rad.s^{-1} per sample).

No output stream value is modified.

B.4 Shake

B.4.1 Analysis

As for the module `still`, the **smoothing process** at the end of the computation introduces a dependency to the sample rate.

As for the module `freefall`, the **time derivation** of the acceleration (jerk) introduces a dependency to the sample rate.

For each dimension, the contribution to the shaking amount corresponds to the proportion of jerk exceeding a threshold within a window of given size, which introduces another dependency to the sample rate via the module `pipo slice`.

B.4.2 Proposed modifications

To eliminate the sample rate dependency from the smoothing process, use the same method as for the module `still`. Here, the original parameters are $f_s = 100$ Hz and $\alpha = 10$ (i.e. $a = 0.1$), which leads to the following modification to implement:

$$\alpha = \frac{1}{\sin\left(2\pi\frac{f_c}{f_s}\right)} = \frac{1}{\sin\left(\frac{100 \arcsin(0.1)}{f_s}\right)}$$

To eliminate the sample rate dependency from the derivation of acceleration, use the same method as for the module `freefall`, i.e. use the real value of the derivative by dividing by the sample period Δt . The threshold given as input parameter should be updated to 10.0 m.s^{-3} instead of 0.1 m.s^{-2} per sample in the original version of the module.

Finally, to eliminate the sample rate dependency due to the sliding window considered for the proportion of jerk exceeding the updated threshold, the time equivalent should be given as input instead of the window size, and the window size given as parameter of the module `pipo slice` should be computed according to the sample rate. In the original analysis patch, two examples of window size are given: $n = 60$ and $n = 200$, corresponding respectively to 0.6 s and 2.0 s. Providing these durations (or any other duration) as input, the relevant window size is given multiplying by the sampling frequency (100 Hz in the original version of the module). No output stream value is modified.

B.5 Intensity

B.5.1 Analysis

As for the modules `freefall` and `shake`, the **derivation** of the acceleration (jerk) or of the angular velocity (angular acceleration) given in input introduces a dependency to the sample rate.

As described in Deliverable 4.2, we consider the input signal $s(t)$ discretized as s_n at time t_n (and such that $t_{n+1} - t_n = 1/f_s = \Delta t$) and its time derivative $s'(t)$ discretized as s'_n . The output I_n of the integration-like part of the module is then defined by the following recurrence:

$$\begin{cases} I_1 &= 0 \\ I_{n+1} &= aI_n + bs_n'^2 \end{cases}$$

It can be easily shown that the general term of the series (I_n) can be expressed as a function of terms of the series (s'_n) as follows:

$$I_n = b \sum_{i=1}^n a^{n-i} s_i'^2$$

where a and b are the parameters of the module `intensity`. This expression corresponds to the approximation of the following integral quantity via the rectangle method:

$$I(T) = \frac{b}{\Delta t} \int_0^T a^{*T-t} s'(t)^2 dt$$

where T is the current time and $a^* = a^{1/\Delta t} = a^{f_s}$.

B.5.2 Proposed modifications

To eliminate the sample rate dependency from the derivation of acceleration or angular velocity, use the same method as for the module `freefall`, i.e. use the real value of the derivative by dividing by the sample period Δt .

For the integration-like part, use the following parameters: $A = (a^*)^{\Delta t} = a^{\frac{\Delta t}{0.01}} = a^{\frac{100}{f_s}}$ instead of a , b unchanged, and multiply the output by the sample period.

Since the output of the derivation module is squared and used as input to the integration part, one could just divide the module's output stream by Δt once (i.e. multiply by f_s) but the aim is to work with real derivation and integration, so the modules `pipo delta` and `pipo finitedif` (which may be used instead) will embed information about the sample rate in the future (i.e. perform the division by the sample period). Therefore it is better to keep the two aforementioned alterations clearly separated.

The modified version of the module will output a stream which value is equivalent to the original module's output stream value divided by Δt . This affects any downstream threshold, such as in the module `kick`.

B.6 Kick

B.6.1 Analysis

The input of the module `kick` is the output of the module `intensity`, which is tackled above. Assuming that this input is independent from the sample rate, the only dependency is introduced by the median filter module `pipo median`. However, since the value of the intensity output stream is modified, the threshold for kick detection should also be taken care of.

B.6.2 Proposed modifications

As for the module `shake`, the sample rate dependency can be eliminated by taking the time-domain equivalent to the window size. In the original version of the module, the filter size is 9, which corresponds to 0.09 s for an assumed sample rate of 100 Hz. The modified median filter size should be:

$$n_{\text{median}} = \left\lceil \frac{9f_s}{100} \right\rceil + \delta_{\text{even}}$$

where $\delta_{\text{even}} = 1$ if $\left\lceil \frac{9f_s}{100} \right\rceil$ is even and $\delta_{\text{even}} = 0$ otherwise so that the median filter size is odd. Additionally, a minimum value of 3 may be set for n_{median} .

Kick detection occurs when the difference between intensity given as input and its

median over the given window exceeds a threshold originally set to 0.1. To take into account the modification of the value of the intensity output stream (which is multiplied by f_s , i. e. 100 Hz in the original version of the module), this threshold should be set to $0.1 \times 100 = 10$.