FP7-ICT-2013-C FET-Future Emerging Technologies-618067

SkAT-VG: Sketching Audio Technologies using Vocalizations and Gestures

D6.6.1
Automatic system for the generation of sound sketches
(Companion Document)

First Author	Stefano Baldan
Responsible Partner	IUAV
Status-Version:	Final-1.0
Date:	December 28, 2016
EC Distribution:	Consortium
Project Number:	618067
Project Title:	Sketching Audio Technologies using Vocalizations and Gestures

Title of Deliverable: (Companion Document)	Automatic system for the generation of sound sketches
Date of delivery to the EC:	31/12/2015

Workpackage responsible for the Deliverable	WP6
Editor(s):	Stefano Baldan, Stefano Delle Monache, Davide Mauro, Davide Rocchesso
Contributor(s):	Stefano Baldan, Stefano Delle Monache, Davide Mauro, Davide Rocchesso, Hélène Lachambre
Reviewer(s):	Davide A. Mauro
Approved by:	All Partners

Abstract	The current deliverable presents the results of tasks T6.1 and T6.2.
Keyword List:	SDT, sound synthesis, physical modeling

Document Revision History

Version	Date	Description	Author
Skeleton	03/18/2015	Import from .tex template	ROC
Draft	10/15/2015	For reviewers	ROC
Update	12/28/2016	Major update for distribution	STEB

Table of Contents

Index of Figures

List of Acronyms and Abbreviations

DoW Description of Work

EC European Commission

PM Person Months

WP Work Package

1 Overview

Work package 6 (WP6) represents the generative block of the SkAT-VG project. In the ideal SkAT-VG workflow, the sound designer prompts the system to recognize her vocal and gestural imitation within a dataset of referent sounds, sorted in perceptually discriminable categories. Vocalizations and gestures are used in a first stage to select a sound category, and in a second stage to control and refine its synthetic counterpart. Within this general workflow, WP6 is responsible for the development of

1. sound synthesis tools able to simulate the sound sources represented by imitations;
2. high-level control strategies and layers to combine the sound models and to manipulate their parameters using vocalization and gestures;
3. software architectures and user interface modules (UI) to facilitate the activity of the sound designer.

WP6 activities are therefore firstly devoted to the development of *timbral families*, namely collections of sound models and relative parameter spaces designed to accurately represent the corresponding sound categories which have been experimentally assessed in WP4 as unambiguously discriminable in terms of interaction, temporal and timbral properties. Physically-informed modeling is being adopted as the main sound synthesis paradigm, according to the general ecological approach of the SkAT-VG project. Secondly, WP6 is also concerned with the extraction of features and descriptors from the user's input, and their mapping to the parameter spaces of the defined timbral families to dynamically control the sound synthesis using vocalizations and gestures. Finally, WP6 takes care of the integration of the selection classifier being developed in WP5 with the temporal control layer and the synthesis tools described above, to produce a general user interface framework for the intuitive exploration of voice-driven sound design spaces. The three tasks, spanned over the whole duration of the work package, represent the development steps towards the final achievement of an automatic system for the generation of sound sketches.

1.1 Achievements

WP6 officially started in March 2015, although some activities were anticipated to the first year to support activities in WP4 and WP7. The deliverable presents two pieces of software, being developed in the project:

1. The *Sound Design Toolkit* (SDT) – a collection of interactive, physics-based sound models suitable to synthesize a variety of acoustic phenomena, mechanical interactions and machines;
2. *SkAT Studio* – a general UI framework, developed in Max, for the integration of the several SkAT-VG blocks as loadable modules within custom audio processing workflows.

The current status of WP6, which is reflected in the activity schedule shown in Figure 1, is the following:

- The SDT legacy code has been ported into a new software architecture;
- The SDT sound models have been reorganized into a new taxonomy, reflecting the state of the art on the classification of environmental sounds;
- The SDT palette of physics-based sound synthesis algorithms has been expanded, to provide an exhaustive framework and a modular lexicon of sound models. This achievement is functional to the definition of timbral families, namely peculiar parametrizations of a collection of one or more sound synthesis models, unambiguously discriminated in terms of interaction, temporal and timbral properties;
- Timbral families have been defined and developed as Max patches. Emerged as synthetic counterparts of the perceptually relevant sound categories defined by WP4, timbral families are also sorted in three main classes: *Machines*, *Mechanical interactions* and *Abstract sounds*;
- Sound analysis tools specifically tailored for vocal control of the sound synthesis models, as well as post-processing modules functional to the design of the timbral families, have been developed as Max externals;
- The SkAT Studio framework and modules are being refined to improve the stability of the system and the effectiveness of the workflow.

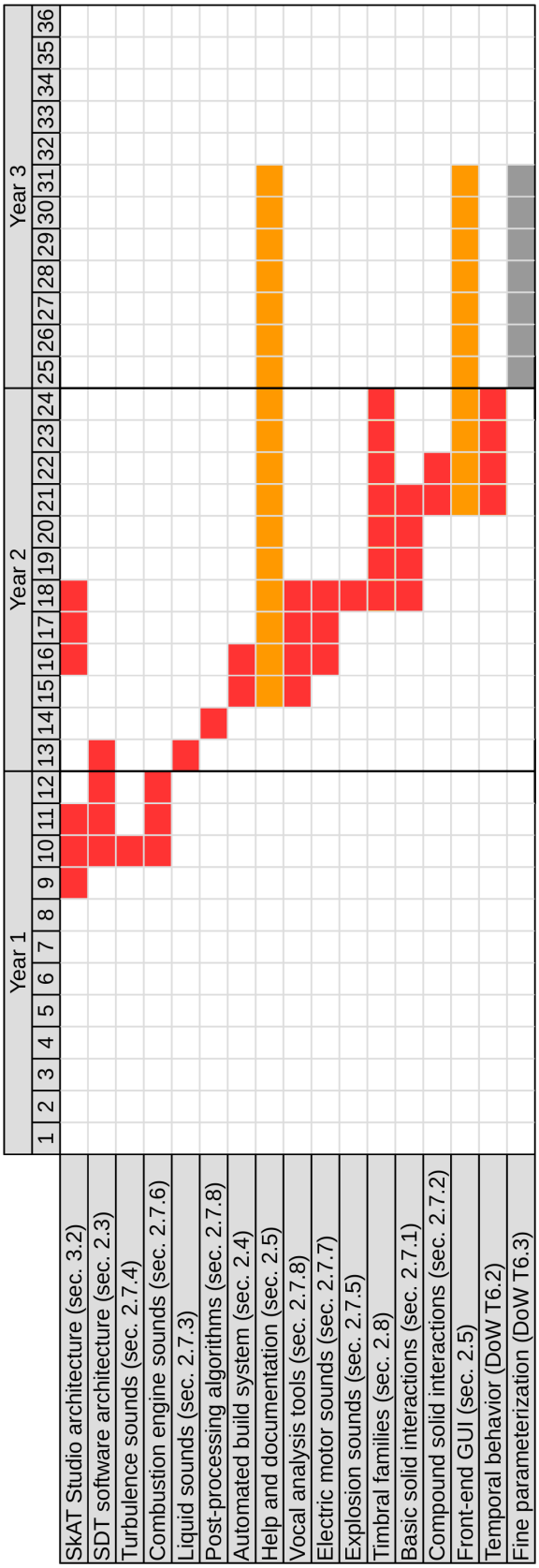

Figure 1: Timing of WP6 activities. Items in red are completed, in orange are in progress, and in gray are planned.

2 The Sound Design Toolkit

The Sound Design Toolkit (SDT) is a software package that provides a palette of sound models that can be exploited in sonic interaction design research. It can be described as a virtual Foley pit which can be used to sketch sonic interactive experiences in a variety of contexts, from gaming, to product design, to audio-visual shows.

2.1 Objectives

Goal of the SDT is to provide an advanced, perception-oriented and physically-consistent sound synthesis toolbox, tailored to sound design thinking and practice [DPR10]. The SDT development is carried out in order to accomplish a three-folded objective:

1. Providing a collection of sound models that covers a mixture of acoustic phenomena, basic mechanical interactions (i.e., everyday sounds) and machines, which are relevant from the perceptual and design practice perspectives;

2. Providing a procedural approach to sound generation, responding to the requirements of design thinking in terms of expressiveness, and immediacy of use in sonic sketching and prototyping;

3. Providing a computationally affordable software environment for real-time applications on ordinary hardware.

The first objective deals with i) the selection of those categories of sounds that cover the major applications of sound design and that are relevant for the listeners, ii) the understanding of the underlying sound-generating physical processes, iii) providing physically-informed synthetic sonic spaces which are wide, malleable and intuitive enough to foster creativity in sound design.

The second objective is accomplished by providing a physically-informed approach that emphasizes the role of sound as a behavior, a process rather than a product. By growing on the metaphor of drawing, each sound model can be seen as a colored pencil, readily available to the sound designer for sketching and prototyping. The generated sound is the resultant of computed descriptions of mechanical interactions occurring between virtual sounding objects with specific configurations, materials and geometries. These simulations of real world phenomena yield acoustic results which are easily predictable resorting to basic everyday experience. Efforts are focused on providing economical control layers and parameter spaces that interpret and map the physical descriptions in an intuitive way.

The above requirements favor the accomplishment of the third objective: Auditory perceptual relevance, *cartoonification* (i.e., simplification of the underlying physics and exaggeration of its most relevant aspects), and parametric control are exploited as specific design constraints of economy of means and reduction of parameters, in order to increase both computational efficiency and perceptual clarity.

2.2 History

The SDT was originally developed at the University of Verona in the scope of the two EU projects Sounding Object (SOb, 2001 - 2003), and CLOSED (Closing the loop of Sound Evaluation and Design, 2006 - 2009), and maintained later at IUAV [Roc14].

In the SOb project, the first version of a PureData[1] library of physics-based sound models was developed and demonstrated in tasks of human-object continuous interaction [RF03]. Aim of the project was to investigate the objective nature of audible objects, in terms of invariants of the structure and behavior of the sound sources. The elementary sound models of impact, friction and derived textures and events such as crumpling, rolling and bouncing were developed to stress the possibilities of sound-mediated interaction and non-visual displays. This highlighted the importance of dynamic sound models in interfaces, whose acoustic manifestations do not need to be realistic to be perceived as physically consistent. The SOb project set the framework of the SDT, in terms of the basic requirements for its development (i.e., perceptual relevance and *cartoonification*, dynamic control).

In the CLOSED project, further development of the SDT (sound models and GUIs) was instrumental to investigate a structured and iterative process of product sound design, from sound ideas generation, to sound creation and evaluation [DR14]. The library was reorganized according to a taxonomy of everyday sounds, based on Gaver's framework [Gav93], and expanded to include sound models of liquid interactions. The pre-existing library, inherited from the SOb project, was made available to be compiled as both PureData and Max[2] externals on Mac, Windows and Linux operating systems. A front-end application in Max 4 was developed and extensively used in various interactive installations, workshops and research activities [RPD09, DPR10]. Since 2010, the SDT has been maintained by IUAV, where the core libraries and front-end applications have been updated for the latest releases of Max and PureData.

In the SkAT-VG project, the sound palette has been further enriched and the whole library radically re-wrapped. The taxonomy of sound models has been completed with the families of aerodynamic interactions (e.g., gasses, explosions, types of airflows) and machines (e.g., combustion engine, DC motor), and organized to reflect the state of the art on the perception of everyday sounds. In general, the new software architecture emphasizes the malleability and modularity of the sound models as basic bricks to design the sound of more complex machines and mechanical interactions, yet preserving a stable auditory representation in listeners. A complete re-design of the patches and front-end application in the Max environment is aimed at strengthening and supporting the seamless ubiquity of sound design thinking. In addition, the toolkit has been enhanced by a set of sound processors (pitch shifter and reverb), and audio features extractors (fundamental frequency estimator, envelope follower, spectral analyzer), developed within the SkAT-VG project to respectively support the definition of the timbral families and the vocal control of the sound synthesis models.

[1] `http://puredata.info`
[2] `https://cycling74.com/products/max/`

2.3 Software architecture

The legacy Sound Design Toolkit was coded in C/C++ using flext[3], a layer for cross-platform development of Max and PureData externals. This choice allowed to build the SDT software for both environments on Mac, Windows and Linux with minimum effort. Unfortunately, the development tracks of Max and PureData kept diverging over the years, up to a point where the flext middleware became outdated and cross-compilation became a problematic task. Moreover, the use of flext limited the availability of the Sound Design Toolkit to Max and PureData, impeding the use of its algorithms and synthesis models in projects built with different platforms.

In the SkAT-VG project, the Sound Design Toolkit was extended with new features, its software architecture was also completely redesigned and its legacy source code was entirely ported in the new framework. The system is now composed of:

1. A core library coded in ANSI C, with few and widely supported dependencies, exposing a clean and streamlined API to all the implemented synthesizers, signal processors and timbre descriptors;

2. A set of wrappers for Max (version 6 or above) and PureData, providing access to most of the SDT framework features by means of *externals*;

3. A collection of Max patches and help files, providing a user-friendly GUI and an extensive user documentation for the whole framework.

This modular and hierarchical structure makes the code extremely portable and consistent in its behavior across different platforms and operating systems. Although the integration with Max is the most actively supported, the C API exposed by the core library can be used on its own in a wide variety of developing environments, greatly expanding the number of possible application domains for the system.

[3] `http://grrrr.org/research/software/flext/`

2.4 Downloading and installing

2.4.1 System requirements

The Sound Design Toolkit runs on Windows and Mac OS X with Max version 6 or above. It also runs on Windows, Mac OS X and Linux with PureData version 0.41.4 or above. Max can be downloaded and installed from the official Cycling '74 website (`http://www.cycing74.com`), while PureData can be downloaded from `http://puredata.info/downloads/pure-data`.

2.4.2 Precompiled binaries

Precompiled binaries are available for Windows and Mac OS X as Max and PureData externals. Simply download the universal SDT package for from the official website:

`http://www.soundobject.org/SDT`

then unpack it and copy the branch for your operating system and platform into the `Packages` folder of your Max installation.

2.4.3 Compiling from source

In alternative, the Sound Design Toolkit source code can be obtained cloning the SDT repository using Git:

```
git clone https://github.com/SkAT-VG/SDT.git
```

The compilation process is extremely simple and fully automated thanks to GNU Make[4] and custom Makefiles for each specific platform. To build the software package, please refer to the instructions for your operating system:

Mac OS X

1. In a terminal, type the following commands to compile the software:

   ```
   cd build/macosx
   make
   ```

2. Install one or more products (Max package, Pd library or Apple framework). The script will install the desired product in the given `DSTDIR` path, creating a `SDT` subfolder:

   ```
   make install_max DSTDIR=<path>
   make install_pd DSTDIR=<path>
   make install_core DSTDIR=<path>
   ```

3. As an optional step, clean the source directories after compilation:

   ```
   make clean
   ```

[4] `https://www.gnu.org/software/make/`

Windows To compile the Sound Design Toolkit under Windows, you need a distribution of the GNU C Compiler and a UNIX style shell, as provided in MinGW + MSYS (http://www.mingw.org, recommended) or Cygwin (http://www.cygwin.com).

1. Once the compilation environment is installed, open its shell and issue the following commands to compile the software:

   ```
   cd build/windows
   make
   ```

2. Install one or more products (Max package, Pd library or DLL library). The script will install the desired product in the given `DSTDIR` path, creating a `SDT` subfolder:

   ```
   make install_max DSTDIR=<path>
   make install_pd DSTDIR=<path>
   make install_core DSTDIR=<path>
   ```

3. As an optional step, clean the source directories after compilation:

   ```
   make clean
   ```

Linux

1. In a terminal, type the following commands to compile the software:

   ```
   cd build/linux
   make
   ```

2. Install the SDT. By default, the script will install a shared library in `/usr/lib` and a collection of PureData externals and patches in `/usr/lib/pd/extras/SDT`:

   ```
   make install
   ```

   Root privileges may be required to access the default install path. If you want to change it, provide a `PREFIX` argument:

   ```
   make install PREFIX=<path>
   ```

3. As an optional step, clean the source directories after compilation:

   ```
   make clean
   ```

2.5 Front-end application and GUI

The current version of the SDT is being developed as front-end application and patches in the Max environment, version 6 or above. The organization of the patches follows the standard convention for the release of Max packages[5].

The SDT overview patch, available as an item in the `extras` menu and shown in Figure 2, lists all the sound models, currently available as Max externals, arranged according to the type of interaction primitives (i.e., vibrating solids, liquids, gasses) and the hierarchy of dependencies occurring between the low-level sound models and the derived basic textures and processes. In addition, the SDT overview shows the available sound processors and spectrum analyzers, included in the package release.

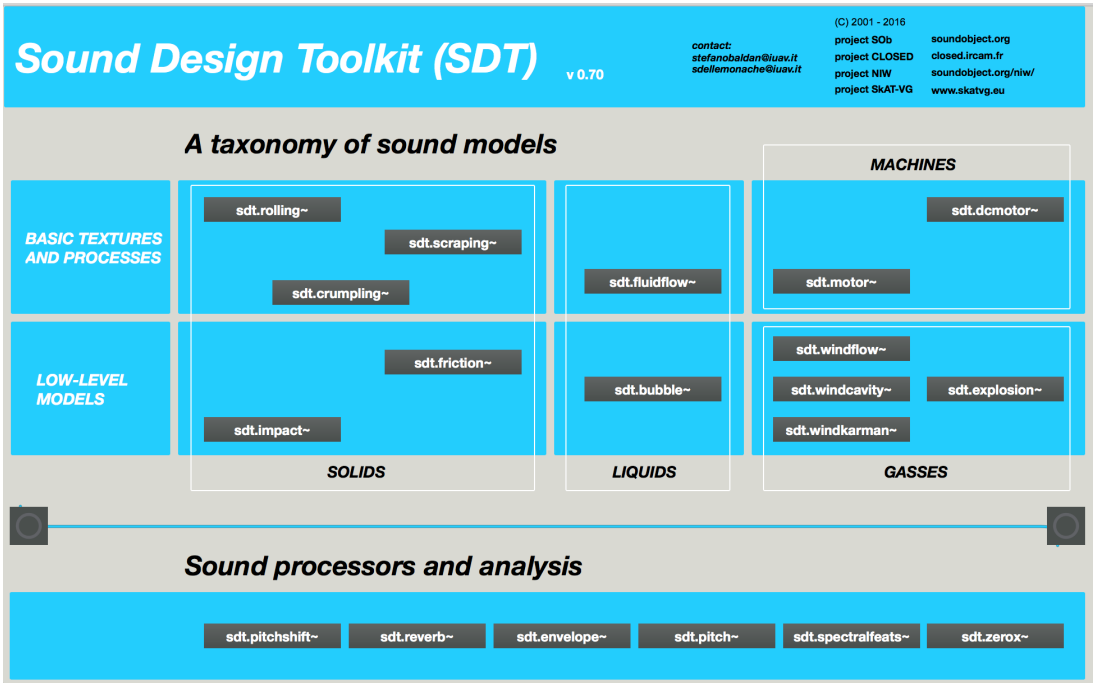

Figure 2: The display of the available sound models, in the SDT overview `extras` patch, reflects the types of underlying interaction primitives, and the dependencies between the low-level models and the textures / processes that can be directly derived.

`Help` patches for each sound model can be recalled by clicking on the corresponding `sdt.name~` gray button. As an example, Figure 3 shows the help patch for the impact sound model, which describes a non-linear impact between one inertial object (`sdt.inertial~`) and one modal object (`sdt.modal~`), according to the characteristics of the collision set in the interactor (`sdt.impact~`). For a detailed description of the resonator–interactor–resonator mechanism adopted to simulate basic interactions between solid objects, please refer to section 2.7.1.

A convention shared by all the sound models is the exploitation of Max attributes to drive all the parameters updated at control rate. In addition, each patch is provided with a detailed description of the involved externals (i.e., input, output, arguments and attributes), as described

[5] `https://docs.cycling74.com/max7/vignettes/packages`.

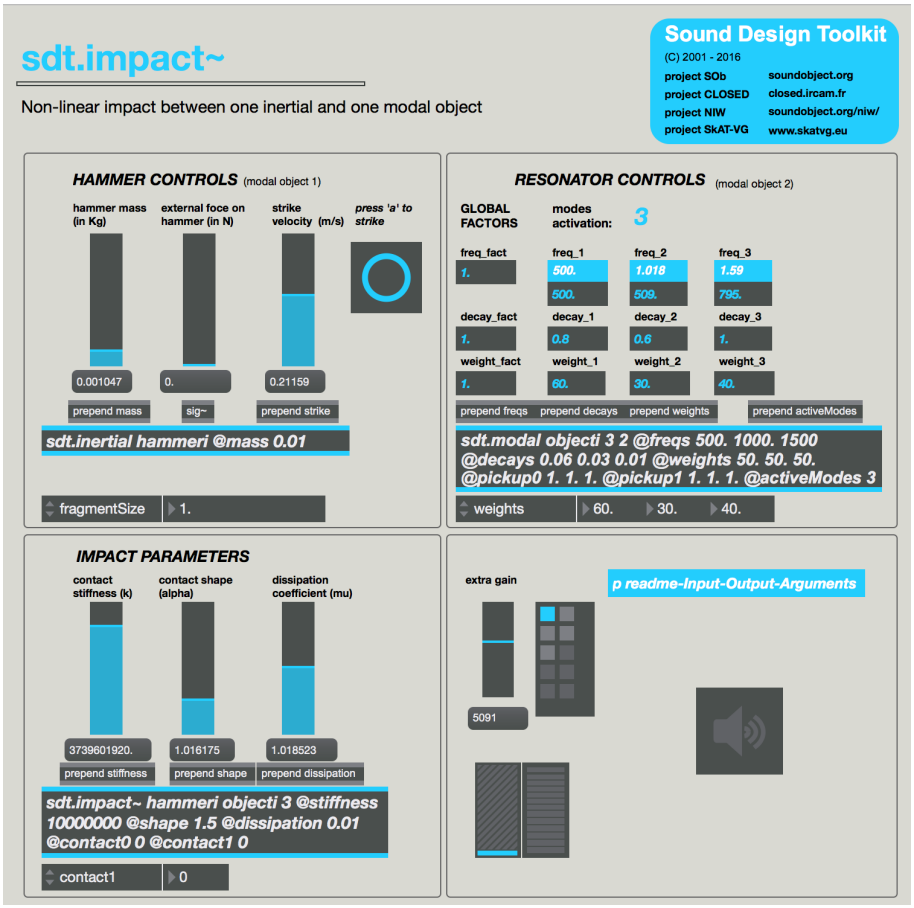

Figure 3: The `help` patch of the impact sound model.

in section 2.6, accessible by double-clicking on the `[p readme-Input-Output-Arguments]` blue framed box.

Finally, the SDT front-end application[6] is undergoing a deep redesign, in order to make readily available all the features provided by the new SDT software architecture, and to ensure forward compatibility with the recent release of Max 7 by Cycling '74.

[6] see `https://vimeo.com/album/2105400/video/51050158` for an introduction to the release based on the legacy software architecture.

2.6 Externals

2.6.1 Basic interactions between solids

[sdt.inertial] Physical model of an inertial mass.

Initialization arguments (mandatory) :

1. (symbol) Unique user-defined resonator ID.

Inlets:

1. strike (float) (float): Reset displacement and velocity to the given values.

Attributes :

@mass: (float) Mass of the inertial object, in Kg,

@fragmentSize: (float) Fraction of the whole object, mostly used by the crumpling algorithm to simulate fragmentation [0.0, 1.0].

[sdt.modal] Physical model of a modal resonator.

Initialization arguments (mandatory):

1. (symbol) Unique user-defined resonator ID,
2. (int) Number of available modes,
3. (int) Number of available pickup points.

Attributes:

@freqs: (list of float) Frequency of each mode, in Hz,

@decays: (list of float) Decay of each mode, in s,

@fragmentSize: (float) Fraction of the whole object, mostly used by the crumpling algorithm to simulate fragmentation [0.0, 1.0],

@activeModes: (int) Number of currently active modes out of all the available ones,

@pickup0, @pickup1, ... : (list of float) Modal weights for each pickup, in 1/Kg.

[sdt.impact~] Nonlinear impact model between two resonators.

Initialization arguments (mandatory):

1. (symbol) Resonator ID of the first object,
2. (symbol) Resonator ID of the second object,
3. (int) Number of outlets, which should correspond to the total number of pickups.

Inlets:

1. (signal) External force applied to the first object, in N,
2. (signal) If not 0 instantaneously puts objects in contact and sets the first object velocity, in m/s,
3. (signal) Fragment size of the first object [0.0, 1.0],
4. (signal) External force applied to the second object, in N,
5. (signal) If not 0 instantaneously puts objects in contact and sets the second object velocity, in m/s,
6. (signal) Fragment size of the second object [0.0, 1.0].

Outlets:

Pickup displacements for first and second object, ordered left to right. If there are fewer outlets than pickups, some outputs will not be available. Outlets in excess will constantly output 0.

Attributes:

@stiffness: (float) Impact stiffness,

@dissipation: (float) Dissipation coefficient, [0.0, 1.0],

@shape: (float) Contact shape factor (1.5 for spherical objects),

@contact0: (int) Pickup index of the contact point for the first object,

@contact1: (int) Pickup index of the contact point for the second object.

[sdt.friction~] Elastoplastic dynamic friction model between two resonators.

Initialization arguments (mandatory):

1. (symbol) Resonator ID of the first object,
2. (symbol) Resonator ID of the second object,
3. (int) Number of outlets, which should correspond to the total number of pickups.

Inlets:

1. (signal) External force applied to the first object, in N,
2. (signal) If not 0 instantaneously puts objects in contact and sets the first object velocity, in m/s,
3. (signal) Fragment size of the first object [0.0, 1.0],
4. (signal) External force applied to the second object, in N,
5. (signal) If not 0 instantaneously puts objects in contact and sets the second object velocity, in m/s,
6. (signal) Fragment size of the second object [0.0, 1.0].

Outlets:

Pickup displacements for first and second object, ordered left to right. If there are fewer outlets than pickups, some outputs will not be available. Outlets in excess will constantly output 0.

Attributes:

@stiffness: (float) Bristle stiffness,

@dissipation: (float) Bristle dissipation,

@viscosity: (float) Bristle viscosity,

@noisiness: (float) Amount of sliding noise,

@kStatic: (float) Static friction coefficient [0.0, 1.0],

@kDynamic: (float) Dynamic friction coefficient [0.0, 1.0],

@breakAway: (float) Break-away coefficient,

@stribeck: (float) Stribeck velocity, in m/s,

@force: (float) Normal force between the two objects, in N,

@contact0: (int) Pickup index of the contact point for the first object,

@contact1: (int) Pickup index of the contact point for the second object.

2.6.2 Compound interactions between solids: textures and processes

[sdt.bouncing~] Impact model controller for bouncing sounds.

Inlets:

1. (bang) Triggers a bouncing process.

Outlets:

1. (signal) Impact velocity, in m/s.

Attributes:

@restitution: (float) Restitution coefficient [0.0, 1.0],
@height: (float) Initial height of the bouncing object, in m,
@irregularity: (float) Shape irregularity (divergence from a sphere) [0.0, 1.0].

[sdt.breaking~] Impact model controller for discrete breaking sounds.

Inlets:

1. (bang) Triggers a breaking process.

Outlets:

1. (signal) Impact velocity, in m/s,
2. (signal) Fragment size, compared to the whole object size [0.0, 1.0].

Attributes:

@crushingEnergy: (float) Energy of the micro-impacts [0.0, 1.0],
@granularity: (float) Crumpling granularity [0.0, 1.0],
@fragmentation: (float) Tendency of the object to break into smaller pieces [0.0, 1.0].

[sdt.crumpling~] Impact model controller for continuous crumpling sounds.

Outlets:

1. (signal) Velocity of micro impacts, in m/s,
2. (signal) Fragment size, compared to the whole object size [0.0, 1.0].

Outlet values are meant to be appropriately assigned and rescaled to control the parameters of two impacting objects.

Attributes:

@crushingEnergy: (float) Energy of the micro-impacts [0.0, 1.0],
@granularity: (float) Crumpling granularity [0.0, 1.0],
@fragmentation: (float) Tendency of the object to break into smaller pieces [0.0, 1.0].

[sdt.rolling~] Impact model controller for rolling sounds.

Inlets:

1. (signal) Rolling surface profile.

Outlets:

1. (signal) Normal force acting on the rolling object, in N.

Attributes:

@depth: (float) Depth of the surface irregularities, affects the amplitude of the micro-impacts,

@grain: (float) Surface granularity, affects the density of the micro-impacts,

@mass: (float) Mass of the rolling object,

@velocity: (float) Rolling velocity of the object.

[sdt.scraping~] Friction model controller for scraping sounds.

Inlets:

1. (signal) Scraped surface profile.

Outlets:

1. (signal) Normal force acting on the surface, meant to be directly applied to a resonator. Friction with another object can also simulate rubbing phenomena.

Attributes:

@grain: (float) Horizontal smoothness of the surface grain,

@force: (float) Normal force applied to the scraping probe,

@velocity: (float) Velocity of the probe.

2.6.3 Bubbles and liquid sounds

[sdt.bubble~] Single bubble model.

Inlets:

1. (bang) Triggers a bubble.

Outlets:

1. (signal) Bubble sound.

Attributes:

@radius: (float) Bubble radius, in mm [0.15, 150.0],

@riseFactor: (float) Audible rise in frequency, ~0.1 for bubbles in water [0.0, 3.0].

[sdt.fluidflow~] Liquid sounds model, as a stochastic population of bubbles.

Initialization arguments (optional):

1. (int) Object polyphony. Default number of voices is 128.

Outlets:

1. (signal) Liquid sound.

Attributes:

@avgRate: (float) Average number of bubbles per second [0.0, 100000.0],

@minRadius: (float) Minimum bubble radius, in mm [0.15, 150.0],

@maxRadius: (float) Maximum bubble radius, in mm [0.15, 150.0],

@expRadius: (float) Bubble radius gamma factor, determines the distribution of bubble radii across the range [0.0, 10.0],

@minDepth: (float) Minimum bubble elevation (deep) [0.0, 1.0],

@maxDepth: (float) Maximum bubble elevation (shallow) [0.0, 1.0],

@expDepth: (float) Bubble depth gamma factor, determines the distribution of bubble elevations across the range [0.0, 10.0],

@riseFactor: (float) Audible rise in frequency, ~0.1 for bubbles in water [0.0, 3.0].

@riseCutoff: (float) Bubbles below this elevation do not change frequency [0.0, 1.0].

2.6.4 Wind and gas turbulence

[sdt.windflow~] Turbulence model of gases impacting against a surface.

Inlets:

1. (signal) Gas velocity [0.0, 1.0].

Outlets:

1. (signal) Turbulence sound.

[sdt.windcavity~] Turbulence model of gases passing through cylindrical cavities.

initialization arguments (optional):

1. (int) Buffer size of the internal comb filter, in samples. Default is 44100, which allows for a maximum cavity length of ~343 m at a sampling rate of 44.1 kHz.

Inlets:

1. (signal) Gas velocity [0.0, 1.0].

Outlets:

1. (signal) Turbulence sound.

Attributes:

@length: (float) Cavity length, in m,

@diameter: (float) Cavity diameter, in m.

[sdt.windkarman~] Turbulence model of gases flowing across thin objects, such as branches or wires.

Inlets:

1. (signal) Gas velocity [0.0, 1.0].

Outlets:

1. (signal) Turbulence sound.

Attributes:

@diameter: (float) Object diameter, in mm.

2.6.5 Supersonic explosions

[sdt.explosion~] Supersonic explosion model.

initialization arguments (optional):

1. (int) Average length of the reverberation delay lines, in samples. Default is 44100, which allows for a maximum scatter time of ~100 s at a sampling rate of 44.1 kHz,
2. (int) Length of the propagation delay lines, in samples. Default is 4410000, which allows for a maximum propagation time of ~100 s at a sampling rate of 44.1 kHz.

Inlets:

1. (bang) Triggers an explosion.

Outlets:

1. (signal) Shockwave sound,
2. (signal) Blast wind sound,

Attributes:

@blastTime: (float) Initial pressure peak duration, in s,

@scatterTime: (float) Turbulent tail duration, in s,

@dispersion: (float) Amount of turbulence [0.0, 1.0],

@distance: Distance of the explosion from the listener, in m,

@waveSpeed: Shockwave propagation velocity, in m/s,

@windSpeed: Blast wind propagation velocity, in m/s.

2.6.6 Combustion engines

[sdt.motor∼] Combustion engine model.

initialization arguments (optional):

1. (int) Buffer size of the internal digital waveguides, in samples. Default is 44100, which allows for a maximum length of ∼343 m for each tube section at a sampling rate of 44.1 kHz.

Inlets:

1. (signal) Engine Revolutions Per Minute (RPM),
2. (signal) Throttle load [0.0, 1.0].

Outlets:

1. (signal) Intake sound, from the front of the vehicle,
2. (signal) Engine vibrations, from the inside of the vehicle,
3. (signal) Exhaust sound, from the back of the vehicle.

Attributes:

@nCylinders: (int) Number of cylinders in the engine block [1, 12],

@cycle: (int) 0 to select four-stroke, 1 to select two-stroke,

@sparkTime: (float) Fuel ignition time, compared to a full cycle [0.000001, 1.0],

@compressionRatio: (float) Engine compression ratio [5.0, 20.0],

@asymmetry: (float) Engine eccentricity [0.0, 1.0].

@cylinderSize: (float) Volume of each cylinder, in cc.

@intakeSize: (float) Average length of the intake collectors, in m,

@extractorSize: (float) Average length of the exhaust collectors, in m,

@exhaustSize: (float) Length of the main exhaust pipe, in m,

@expansion: (float) Impedance mismatch between extractors and exhaust pipe, present in two-stroke engines to limit the expulsion of unburnt fuel [0.0, 1.0].

@mufflerSize: (float) Average size of the muffler resonators, in m.

@mufflerFeedback: (float) Muffler efficiency [0.0, 1.0].

@outletSize: (float) Length of the exhaust outlet, in m.

@backfire: (float) Amount of backfiring when the engine revs down [0.0, 1.0]

2.6.7 Electric motors

[sdt.dcmotor~] Electric motor model.

initialization arguments (optional):

1. (int) Buffer size of the internal comb filter, in samples. Default is 44100, which allows for a maximum chassis length of ~343 m at a sampling rate of 44.1 kHz.

Inlets:

1. (signal) Rotor Revolutions Per Minute (RPM),
2. (signal) Mechanical load on the rotor [0.0, 1.0].

Outlets:

1. (signal) Electric motor sound,

Attributes:

@coils: (int) Number of coils on the rotor,

@harshness: (float) Spectral density [0.0, 1.0],

@size: (float) Chassis length, in m,

@reson: (float) Chassis resonance [0.0, 1.0],

@gearRatio: (float) Gear ratio,

@rotorGain: (float) Amount of noise coming from the rotor [0.0, 1.0],

@brushGain: (float) Amount of noise coming from the brushes [0.0, 1.0],

@gearGain: (float) Amount of noise coming from the gears [0.0, 1.0],

@airGain: (float) Amount of air turbulence caused by the spinning rotor [0.0, 1.0].

2.6.8 Sound post-processing and analysis

[sdt.reverb~] Maximally diffusive yet efficient Feedback Delay Network (FDN) reverb, as presented in [Roc97].

initialization arguments (optional):

1. (int) Average length of the reverberation delay lines, in samples. Default is 44100, which allows to simulate room dimensions up to ~343 m at a sampling rate of 44.1 kHz.

Inlets:

1. (signal) Dry signal.

Outlets:

1. (signal) Reverb signal.

Attributes:

@xSize: (float) Room width, in m,

@ySize: (float) Room height, in m,

@zSize: (float) Room depth, in m,

@randomness: (float) Shape deviation from a rectangular room [0.0, 1.0],

@time: (float) Global reverberation time, in s,

@time1k: (float) Reverberation time at 1kHz, in s.

[sdt.pitchshift~] Time domain pitch shifter, as presented in [Zoe02].

initialization arguments (optional):

1. (int) Size of the internal buffer, in samples. Default is 4096.

Inlets:

1. (signal) Original signal.

Outlets:

1. (signal) Pitch shifted signal.

Attributes:

@ratio: (float) Pitch shifting ratio.

[sdt.envelope~] Simple envelope follower, based on a one-pole lowpass filter with different attack and release times.

Inlets:

1. (signal) Input signal.

Outlets:

1. (signal) Signal envelope.

Attributes:

@attack: (float) Attack time, in ms,
@release: (float) Release time, in ms.

[sdt.myo~] Vocal myoelastic activity detector.

Inlets:

1. (signal) Input signal.

Outlets:

1. (float) Myoelastic activity amount [0.0, 1.0],
2. (float) Myoelastic average frequency, in Hz.

Attributes:

@lowFrequency: (float) Long term envelope cutoff, in Hz,
@highFrequency: (float) Short term envelope cutoff, in Hz,
@threshold: (float) Signal gate [0.0, 1.0].

[sdt.pitch~] Fundamental frequency estimator based on NSDF [MW05].

initialization arguments (optional):

1. (int) Analysis window length, in samples. Default is 4096.

Inlets:

1. (signal) Input signal.

Outlets:

1. (float) Detected pitch, in Hz,
2. (float) Pitch clarity [0.0, 1.0].

Attributes:

@overlap: (float) Window overlap ratio [0.0, 1.0],
@tolerance: (float) Peak tolerance [0.0, 1.0].

[sdt.spectralfeats~] Spectral analyzer, extracting several audio descriptors.

initialization arguments (optional):

1. (int) Analysis window length, in samples. Default is 4096.

Inlets:

1. (signal) Input signal.

Outlets:

1. (list) Spectral magnitude, centroid, spread, skewness, kurtosis, flatness, flux, whitened/rectified flux (useful for detecting onsets).

Attributes:

@overlap: (float) Window overlap ratio [0.0, 1.0]

@minFreq: (float) Lowest frequency included in the analysis, in Hz (0 for DC),

@maxFreq: (float) Highest frequency included in the analysis, in Hz (0 for Nyquist).

[sdt.zerox~] Extract the zero crossing rate of an input signal.

initialization arguments (optional):

1. (int) Analysis window length, in samples. Default is 4096.

Inlets:

1. (signal) Input signal.

Outlets:

1. (float) Normalized zero crossing rate [0.0, 1.0].

Attributes:

@overlap: (float) Window overlap ratio [0.0, 1.0]

2.7 The SDT taxonomy of sound models

Figure 4 shows the organization of the synthesis models available in the Sound Design Toolkit. Originally based on a revised version of Gaver's perceptual organization of everyday sounds [Gav93, DPR10], the framework has been updated to stress the sophistication of the physics-based algorithms as privileged viewpoint, and reflects the state of the art on the categorization of environmental sounds [LHMS10, HLM+12].

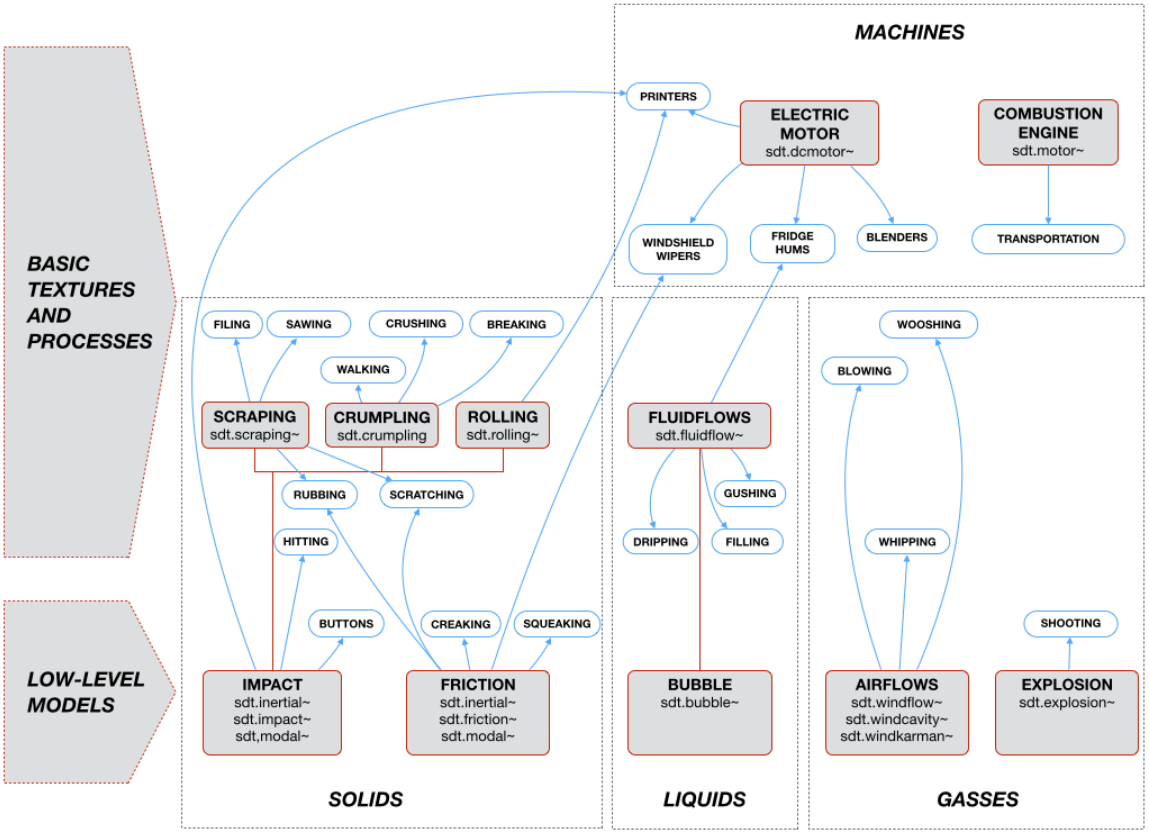

Figure 4: The SDT taxonomy of sound models. The bottom-up hierarchy represents the dependencies between low-level models and temporally-patterned textures and processes, for the four classes of sounds, solids, liquids, gasses, and machines.

In practice, a bottom-up hierarchy of sound models is established, as shown in the graph of figure 4. The first level presents the basic algorithms with the corresponding Max externals, suitable for the generation of a large family of simple sound events. The second level highlights the temporally-patterned events (with the corresponding Max externals), basic textures and processes, that can be straightly derived from the exploitation of the low-level models.

The available sound models are grouped according to a criterion of causal similarity, in four main classes of sounds, that is 1) vibrating solids, 2) liquids, 3) gasses, and 4) machines. As a note, the fourth class (machines) is arranged in the second level, since the available corresponding sound models describe complex mechanisms that would have been too onerous and cumbersome to develop as a Max chain of separate basic events (i.e., low-level models).

In addition, the blue arrows set a direct connection between the sonic space of each model and a higher-level classification of referent sounds that have been identified as cognitively stable in listeners' representations. These perceptually-relevant sound categories, available in the current release of the SDT framework, are further grouped into the two main families of Machines and Mechanical Interactions (see SkAT-VG Deliverable D.4.4.1 for more details). As seen from the SDT taxonomy viewpoint, a peculiar parametrization of single sound models or a specific configuration of more models, which describe a perceptually-relevant category of sound, unambiguously discriminated in terms of interaction, temporal and timbral properties, define a timbral family[7] (see further subsection 2.8). To conclude, the new SDT framework stresses the modularity of the sound models (low-level and temporally-patterned), with practical implications in terms of a reliable selection and manipulation of the synthetic referent sounds.

2.7.1 Basic interactions between solids

The solids sound models represent the main legacy part of the SDT library, as described in section 2.2, yet they have been completely re-designed and re-written, as described in section 2.3. The algorithms share a common, modular structure "resonator–interactor–resonator", representing the interaction between two resonating objects, as shown in figure 5.The currently available object models are:

Inertial mass Simulates a simple inertial point mass. This kind of object is exploited as exciter for the resonators, and its only settable attribute is its mass.

Modal resonator Physical model of a set of parallel mass-spring-damper mechanical oscillators. Each oscillator represents a normal mode of resonance of the object, with the oscillation period, the mass and the damping coefficient of each oscillator corresponding respectively to the resonance frequency, the gain and the decay time of each mode.

Although object models can behave in different ways and be implemented through different algorithms, they all must expose one or more *pickup points*. Each pickup point gives information about its *displacement* and its *velocity*, and can be used to apply an external *force* to the resonator.

Interactor algorithms use the pickup points as interfaces to the resonators, to compute and apply forces to the contact points, based on their relative displacements and/or velocities. The available interactors are:

Impact Simulates a non-linear impact, by computing the impact force from the total compression, namely the relative displacement between the two contact points. The resulting force is the sum of an elastic component and a dissipative one. The elastic component is parameterized by the force stiffness (or elasticity) and a non-linear exponent which

[7] As a final note, the current release of the SDT does not include the class of the Abstract Sounds. This is because the corresponding timbral families (i.e., the synthesis modules) can be achieved with the repertory of classic signal-based techniques and modulations. However, this family will be included in the forthcoming release of the SDT which will account a reorganization according to the timbral families (i.e., the perceptually-relevant categories of sounds sorted in the three main classes of machines, mechanical interactions, and abstract sounds.)

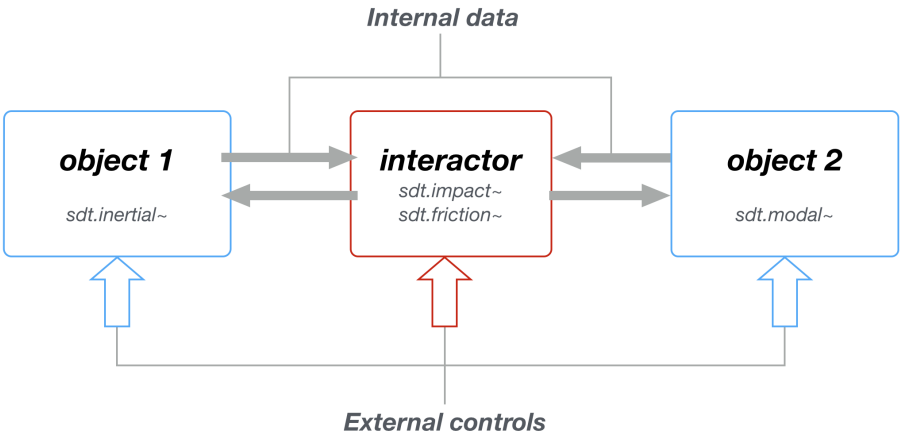

Figure 5: The modular structure implements a feedback communication between interaction and resonator models.

depends on the local geometry around the contact area. The dissipative component is parameterized by the force dissipation (or damping weight).

Friction Simulates a non-linear friction, by computing the friction force from the relative velocity between the two contact points. The resulting force is the sum of four components: an elastic term, an internal dissipation term, a viscosity term, and finally the gain of a pseudo-random function, representing noise related to the surface roughness. More subtle phenomena, such as pre-sliding behavior (gradual increase of the friction force for very small displacements), are parameterized by several other quantities: dynamic and static friction coefficients, break-away coefficient and Stribeck velocity. These phenomena are mostly related to the transients and they are particularly important for a realistic simulation of friction sounds.

Pickup points also allow to "listen" to the resonators: The sound output of a simulated interaction of this kind is usually made of the displacement or the velocity values read from one or more object pickups.

This particular description of solid interactions is not inherently conservative in terms of total energy of the system, and can therefore exhibit wildly unstable behaviors under certain configurations. To avoid this problem, the forces computed by the interactors are limited by an energy conservation scheme, which guarantees the stability of the algorithms under all circumstances.

2.7.2 Compound interactions between solids: textures and processes

The basic interactions between solids are used as low-level building blocks and controlled by higher-level models to create more complex sound textures and events, such as rolling, scraping, breaking and crumpling sounds.

Rolling A perfectly round object rolling along a perfectly smooth surface should be completely silent. In the real world, however, irregularities in both the object and the surface induce

small but sudden changes in the rolling trajectory, which cause a series of micro impacts, as shown in Figure 6. The amplitude and temporal distribution of these impacts yields a characteristic acoustic signature, often recognizable as a rolling sound, in the everyday listening experience.

The rolling model accepts an audio sample or a real time noise generator, to represent the surface profile. The input signal is then filtered by an envelope follower to approximate the object path along the profile.

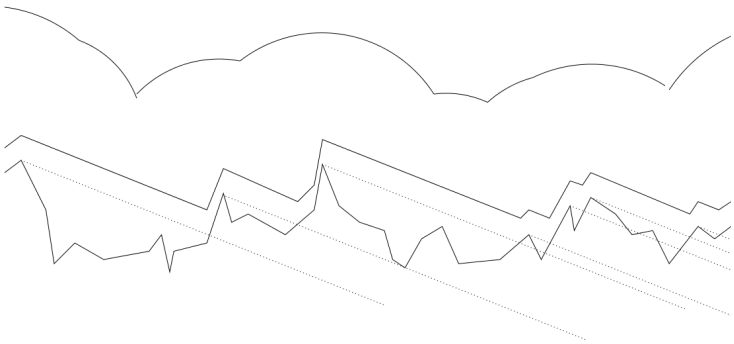

Figure 6: Approximation of the rolling trajectory using an envelope follower with instant attack and linear release. Source: *The Sounding Object* [RF03].

The output of the rolling algorithm is the normal force exerted on the rolling object. In addition to the gravity force, whenever a discontinuity occurs in the rolling trajectory the object experiences an upward lift, with an energy proportional to the depth of the surface irregularities and to the kinetic energy of the rolling object. If the upward lift is strong enough, the ball loses contact with the surface and bouncing occurs. The rolling simulation takes this phenomenon into account, and new impacts are generated only after the gravity force has counterbalanced the last upward lift applied to the ball.

Scraping The sound of a nail, a stylus, a saw tooth or another point-like object scraping against an irregular surface can be also represented as a series of micro impacts, although characterized by a different acoustic signature compared to the rolling sounds. In the SDT development of the scraping model, the surface profile and the object trajectory are computed similarly to the rolling, being the output a normal force. Unlike the rolling scenario, the computed force acts directly on the surface with a downward pressure, instead of an upward lift.

Bouncing The sound of an object dropped from a given height and repeatedly bouncing on a surface can also be represented by a series of impacts. The energy, timing and number of these impact is mostly affected by the initial falling height, the restitution coefficient of the bounce and the irregularity of the object. The higher the falling height, the more velocity will be acquired by the object under the constant drag of gravity and therefore the more powerful and sparsely distributed will be the impacts. High restitution coefficient will yield long and slowly decaying distributions, whereas low coefficients will cause the object to stop sooner. A

spherical object will typically show a regular bouncing pattern, with impacts of exponentially decaying power and delay, whereas an irregular object will yield a more randomly distributed impact series.

Breaking and crumpling Crumpling and breaking sounds are yet other cases involving a superposition of impacts. The main difference with respect to rolling and scraping is that the object is fragmented into facets with different sizes and shapes. Since this type of deformation largely affects the acoustic properties of an object, it is necessary to continuously change and update over time the parameters of the resonators involved in the process.

Highly chaotic processes such as breaking a glass or crushing a can are very difficult to describe in all their physical details. For this reason, a stochastic approach is adopted. The model is defined as a Bernoulli process with finite energy, where each atomic sound event consumes a certain fraction of the global energy of the process. Following results in [HS96], this energy fraction is picked from an exponentially distributed pseudorandom number generator. The remaining energy influences the rate of the Bernoulli process, causing a progressive density reduction which is one of the main acoustic signatures of crumpling and breaking sounds.

Each atomic event represents a fracture of the original object, which splits the whole solid into smaller pieces. The size of each fragment is also determined by a stochastic algorithm and influenced by the remaining energy of the process. Smaller fragments are generated towards the end of the process, as one would expect in a real world situation. Energy value and excited fragment size represent the outputs of the crumpling model. In addition, to provide an effective breaking impression, a short noise impulse can be added at the beginning of the process, as proposed in [WV84].

2.7.3 Bubbles and liquid sounds

The occurrence of acoustic emissions in water or other liquids is due to the gasses, contained in the liquid mass, which emerge as a population of bubbles. From the physical viewpoint, a spherical bubble acts as an exponentially decaying sinusoidal oscillator: The gas volume is excited by the energy involved in the bubble formation process, and then gradually dissipated through the volume pulsation [Doe05]. The cause of the pulsation is the presence of a compressible region (the bubble) injected in an incompressible fluid, which allows the liquid mass to oscillate as it would happen in a spring-mass system. A subtle frequency rise is sometimes added to simulate the characteristic "blooping" sound, generated by bubbles forming close enough to the surface and caused by a reduced effective liquid mass around the trapped gas.

The bubble model can be used as a building block to generate rich and complex liquid sound events. A population of bubbles can be easily obtained by means of a sinusoidal oscillator bank, with each oscillator set to a fixed base frequency. Each voice is modulated in amplitude and frequency according to the exponential decay and the frequency rise of each bubble. Amplitude and frequency envelopes are updated according to a stochastic algorithm to control the behavior of the bubbles population: The bubble generation rate follows a Bernoulli process, whereas the radius (i.e., the voice number in the oscillator bank) and the depth (i.e., the amplitude) for each new bubble are randomly chosen.

2.7.4 Wind and gas turbulence

Wind and gas turbulence models are suitable to simulate wooshes, wind gusts and howls, helicopter rotors and so forth [Far10]. A gas flowing in a constant direction usually does not produce any sound by itself, since the pressure variations are too low to fall into the audible range. Nevertheless, objects obstructing the air flow are likely to cause turbulence at much higher frequencies, and produce sounds.

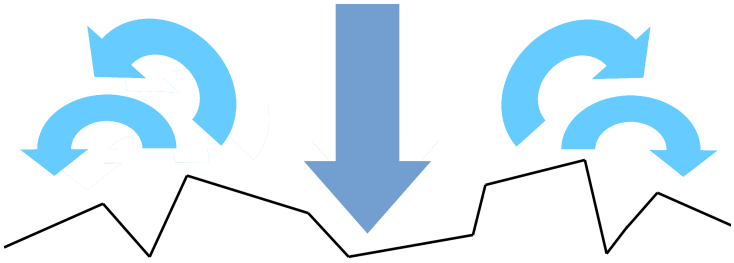

Figure 7: Turbulence noise produced by a gas flow impacting against a rough surface.

One of the possible sources of turbulence is the impact on a large solid surface, shown in Figure 7. In this case, the turbulence is generated due to the impact of the air molecules on the surface and to their random change of direction caused by the irregularities of the surface itself. The resulting sound is modeled through a simple bandpass-filtered white noise generator, modulated in amplitude according to the velocity of the air flow.

The gas trapped in a cylindrical cavity is simulated by means of a comb filter applied to the noise signal, namely with a delay line with feedback. Delay time determines the length of the cavity, while feedback gain determines the amount of energy redirected into the delay line and hence the amount of resonance of the tube. Tube resonance generates harmonics, whose presence depends on the velocity of the gas flowing in the cavity, as shown in Figure 8. This phenomenon is modeled by means of a highly resonant bandpass filter with variable center frequency.

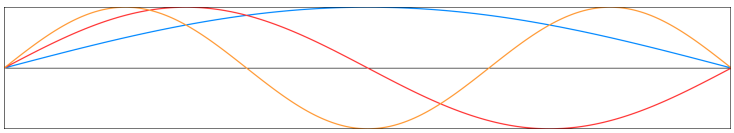

Figure 8: The first three resonance modes in an open cylindrical tube.

Finally, an air flow hitting a thin object, such as a tree branch or a suspended wire, produces a repeating pattern of swirling vortices caused by the unsteady separation of the gas flow around the object, known as *Kármán vortex street*, and shown in Figure 9. A broadband noise going through a highly resonant bandpass filter with variable frequency is an acceptable approximation of its characteristic singing and howling sound.

[8] `https://upload.wikimedia.org/wikipedia/commons/b/b4/Vortex-street-animation.gif`

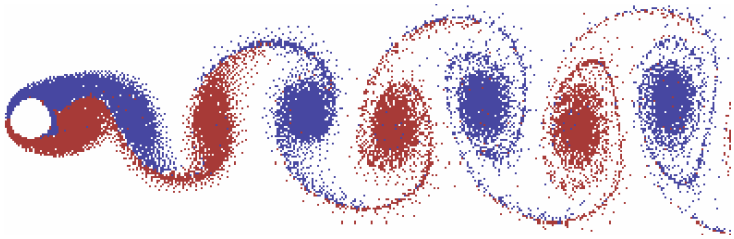

Figure 9: Kármán vortex street created by a cylindrical object. Source: Wikipedia[8].

2.7.5 Supersonic explosions

The explosion model is a novel algorithm, designed and developed from scratch as part of the SkAT-VG project. Powerful explosions, as well as objects travelling at supersonic speed such as rifle bullets or cracking whip tails, create shock waves, namely a sudden peak in pressure followed by a negative expansion tail. In the SDT, this event is modeled with a Friedlander waveform. As the gas runs back to fill the vacuum left by the explosion, a blast wind is generated. Turbulence caused by the blast wind is rendered by means of a bandpass filtered white noise, modulated in amplitude by the Friedlander waveform as the wind intensity loosely follows the profile of the initial shock wave. Real world explosions are almost never perfectly impulsive, on the contrary they are subject to various scattering phenomena. A feedback delay network reverberation unit is used to simulate scattering, adding complexity to the initial blast wave and improving the realism of the acoustic result [RBDM15].

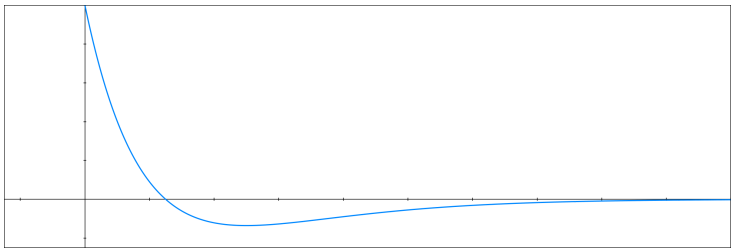

Figure 10: The Friedlander waveform.

2.7.6 Combustion engines

The SDT combustion engine model represents a novel approach in the physical modeling of motor sounds. From a mechanical viewpoint, an internal combustion engine converts chemical energy into kinetic energy by means of a series of controlled explosions. From an acoustic point of view, this process can be described by means of a set of resonating pipes, excited by the explosions occurring in the combustion chambers. The tubes resonances are modeled through digital waveguides, while the engine operation cycle is modeled by means of four distinct functions representing pistons movement, intake valves operation, exhaust valves operation and fuel ignition.

As shown in Figure 11, The entire model is composed by an engine block simulation, including intakes, cylinders and extractors, and by an exhaust system formed by a main exhaust pipe, a set of resonators acting as a muffler and a terminal exhaust outlet. Intake outputs,

engine cycle functions and exhaust output are processed as three independent sound emissions, that can be blended together to simulate different listening points inside and outside the vehicle. A more detailed description of the synthesis model can be found in [BLDMB15].

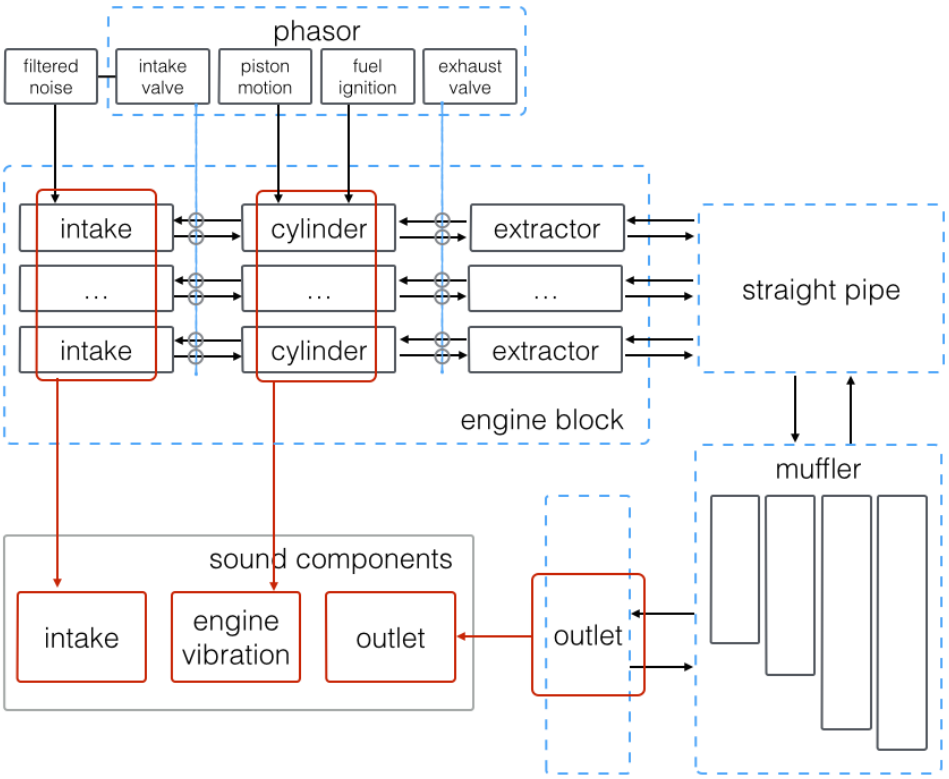

Figure 11: Block diagram of the whole combustion engine model.

2.7.7 Electric motors

Electric motors exploit magnetic induction to convert electric energy into mechanical energy. The simplest arrangement is the DC brushed motor, composed by a stator with magnets of opposite polarity and a rotor with induction coils. Electric contacts (brushes) power the coils, inducing a magnetic field. The coils are pushed away from one magnet and pulled toward the other, thus causing the rotation. A commutator ring is used to invert the polarity of the coils, allowing the process to repeat.

An ideal electric motor should be perfectly silent. In the practice, rotors are never perfectly balanced and generate pitched tones depending on their revolutions per minute (RPM). Moreover, contacts between the brushes and the commutator ring cause friction noise, as it does the gearing system which transmits the mechanical energy to the engine load. Finally, the rotation induces air movement and turbulence noise, sometimes increased by the presence of a cooling fan attached to the rotor.

The pitched tone of the rotor is obtained through additive synthesis, summing a fixed number of harmonic partials. Frequency modulation simulates the irregularity of the rotation, caused by the attached loads. The gear noise is modeled like the rotor noise, with each

partial frequency multiplied by the gear ratio. Rotor, gears and brushes are usually framed inside a closed chassis, whose resonance modes are modeled through a comb filter. Finally, aerodynamic turbulence caused by the spinning parts (rotor, cooling fan) is synthesized with a bandpass-filtered white noise.

2.7.8 Sound post-processing

The current release of the SDT includes some DSP algorithms which are not sound synthesis models per se, but can be used as post-processing tools to render specific timbral properties found in several sound events:

Reverb A "maximally diffusive yet efficient" [Roc97] reverberation algorithm based on a Feedback Delay Network (FDN). Beyond the conventional simulation of room resonances, the algorithm is especially used to enhance the texturization of acoustic elements, such as turbulence and scattering in the explosion synthesis model. For more detailed information about the textural use of the reverberation unit, please refer to [RBDM15].

Pitch shifter A time domain pitch shifting algorithm, based on [Zoe02], initially used to simulate doppler effect in the explosion model and successively discarded in later versions of the synthesizer. It is still available in the Sound Design Toolkit as a post-processing tool to simulate moving sound sources.

2.7.9 Sound analysis

The Sound Design Toolkit offers some sound analysis tools addressing the issue of temporal control and behaviour (task 6.2 in the DoW) of sound synthesis. These algorithms extract a set of audio descriptors, which allow to act in real time on the synthesis parameters through the use of vocalizations. Although already existing third party libraries and externals could be used for the purpose, we chose to write our modules from scratch and include them in our open source release, in line with the founding principles of the Sound Design Toolkit. The sound analysis tools include:

Envelope follower A simple envelope follower based on a one-pole lowpass filter with different cutoff coefficients for attack and release. Useful to extract control data from signal amplitude.

Myoelastic activity detector The ratio between a short-term and a long-term amplitude envelope highlights the low frequency amplitude oscillations typical of most vocal myoelastic activity.

Pitch estimator A fundamental frequency estimator, based on the Normalized Squared Difference Function (NSDF) [MW05]. Useful to extract control data from vocal pitch.

Spectral features Several descriptors related to spectral information are available and include spectral magnitude, four spectral moments (centroid, spread, skewness, kurtosis), spectral flatness, spectral flux and whitened/rectified spectral flux for onset detection [SP07].

Zero crossing rate detector A zero crossing rate detector, reporting how often an audio signal changes from positive to negative and vice versa. It is useful as a rough estimation of noisiness.

2.8 Timbral families

The concept of timbral family emerged as a specific configuration of one or more sound models to represent categories of imitated sounds that are unambiguously discriminable in terms of interaction, temporal and timbral properties (see SkAT-VG Deliverable D.4.4.1).

In the context of the SkAT-VG project, the SDT sound models represent the basic building blocks to compose the emerging timbral families. Table 1 shows the 26 categories of sounds, sorted in the three main families of Abstract Sounds, Machines, and Mechanical Interactions, and the corresponding timbral families with the SDT sound models used. In this respect, whether compound or not, a timbral family can be described in terms of specific, appropriate spaces and trajectories of sound synthesis parameters. Figure 12 shows the Max GUI to access the Machines and Mechanical Interactions synthesizers, and two example patches of timbral families (whipping and blowing). All the synthesizers are made available as example patches, with the exception of the Alarm sound category and the class of Abstract sounds which will be included in the next release of the Sound Design Toolkit, foreseen for December 2016.

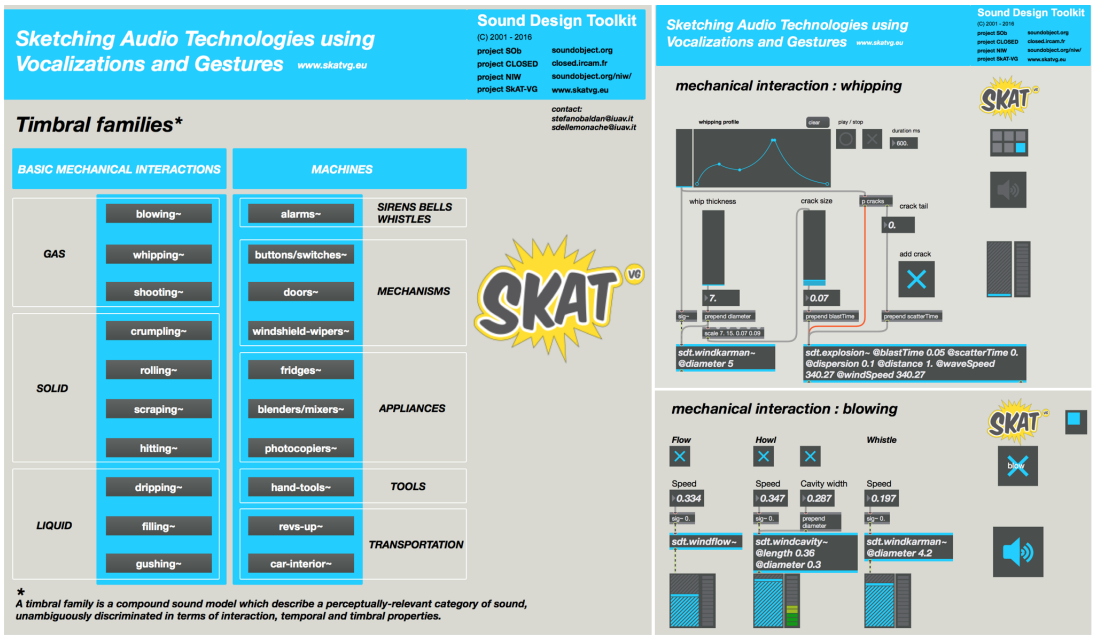

Figure 12: (left) The Max GUI of the timbral families of Machines and Mechanical Interactions; (right) the whipping and the blowing timbral families.

Abstract sounds	
Up Down Up/Down Impulse Repetition Stable	Conventional synthesis techniques (additive, subtractive, AM, FM...)
Machines	
Alarms	Conventional synthesis
Buttons and switches	Impact
Doors closing	Impact
Filing and sawing	Scraping
Fridge hums	Electric motor
Mixers and blenders	Electric motor
Printers fax and xerox	Electric motor + rolling + impact
Windshield wipers	Electric motor + friction
Vehicles exterior revs up	Combustion engine
Vehicles interior accelerating	Combustion engine
Mechanical interactions	
Blowing	Aerodynamic noise
Whipping	Aerodynamic noise + explosion
Shooting	Explosion
Crumpling	Crumpling
Rolling	Rolling
Rubbing and scraping	Friction + scraping
Hitting and tapping	Impact
Dripping and trickling	Fluid flow
Filling	Fluid flow
Gushing	Fluid flow

Table 1: The 26 sound categories that emerged from WP4 experiments are listed on the left column and coupled with the SDT sound models used to represent them. The sounds are grouped in the three classes of Abstract Sounds, Machines, and Mechanical Interactions.

2.9 Temporal control and behavior

To achieve the main goal of WP6, namely imitation-driven sound synthesis, an important task is the extraction of information from vocal imitations and its use in the temporal control and behavior of the available synthesis models. During the last years, IRCAM defined and developed a wide range of descriptor for the analysis of audio signals, fruit of an extensive work towards a deeper understanding of the concept of timbre [PGS+11]. The sound analysis tools described in section 2.7.9 partly reimplement and partly extend the family of IRCAM descriptors as free and open source software.

Perceptual experiments conducted in the last years [LDSA11] and as part of WP4 [LJH+15] point out that:

- Vocal imitations appear to be characterized by a few, simple acoustic features,
- Not all of those features can be consistently and reliably controlled at the same time.

For these reasons, only a limited amount of audio descriptors is actually useful, and an even smaller subset is used to control a particular timbral family at any given time. The most reasonable approach seems to use the voice for a very coarse control of the synthesis models, and to refine the result by hand through a graphical or physical interface.

According to the perceptual experiments mentioned above, the most salient features in recognizing and producing vocal imitations include:

- Amplitude variations and temporal patterns,
- Fundamental frequency, closely related to the sensation of pitch,
- Signal zero crossing rate, a rough estimate of the noisiness of a sound,
- Spectral centroid, directly related to the sensation of brightness of a sound,
- Spectral energy distribution, changing for different vowels.

Amplitude variations can be tracked at signal rate by the `[sdt.envelope~]` envelope follower, or at control rate by the spectral magnitude descriptor implemented in `[sdt.spectralfeats~]`. A fixed threshold on the first derivative of the signal envelope is already a simple but effective onset detector, useful for detecting transients and recognizing discrete temporal patterns in sound events. The spectral features external offers other two descriptors (regular and whitened/rectified spectral flux) which can also be used as onset detection functions.

Fundamental frequency is extracted using the `[sdt.pitch~]` pitch detector, which doubles as an estimator of signal noisiness thanks to its clarity measure. Noisiness can be also estimated by `[sdt.zerox~]`, implementing the zero crossing rate descriptor, or using the spectral flatness descriptor implemented in the spectral analyzer.

`[sdt.spectralfeats~]` is finally used to extract spectral centroid, namely the first statistical moment of the spectrum. Second, third and fourth statistical moments are also computed, as they are other potentially useful features: Spectral spread (variance around the centroid) is an estimate of the signal bandwidth, while spectral skewness and kurtosis describe

the shape of the energy distribution and can be used to roughly discriminate vowel sounds without performing formant analysis.

Analysis externals take an audio signal as input and return the audio descriptors as output. For each timbral family, a small subset of these descriptors is scaled, combined and assigned by an *ad-hoc* mapping to vocally control the sound synthesis process. Although very different in their definition and nature, some of the implemented descriptors represent similar features of the input signal and their domains overlap significantly. Despite their apparent equivalence, they all have been included in the Sound Design Toolkit, because some implementations are better fit than others depending on the kind of vocal imitation produced and on the timbral family that needs to be controlled.

3 SkAT-Studio

SkAT Studio is a prototype demonstration framework developed by Genesis, designed to facilitate the integration with other partners' technologies.

3.1 Application workflow

SkAT Studio strives to provide a practical implementation of the main SkAT-VG workflow, as described in the DoW:

Selection: The user produces a vocal imitation of the desired sound. SkAT Studio classifies the imitation into a sound category, returning the corresponding timbral family (a collection of properly configured synthesizers) and control layer through the use of voice and/or gesture.

Play: The user controls the synthesizers in real time with vocalization and gesture, transforming her vocal sketches in prototype sounds.

The *selection* step accepts a sound signal as input, and outputs a *configuration* which defines the behavior of the *play* step. A SkAT Studio configuration is composed of the five following elements:

Input: Acquisition of voice and gesture.

Analysis: Extraction of meaningful features and descriptors from the input.

Mapping: Transformation of the analysis features into synthesis parameters by further elaboration, rescaling and/or combination.

Synthesis: Production of sound. This can be either purely procedural sound synthesis or post-processing of an existing sound (e.g. pitch shifting or time stretching).

Output: Playback or recording of the final sound.

At the time of writing, only the *play* step is implemented. The *selection* step will be developed during the next year of the project, integrating future results and contributions coming from the work of IRCAM in WP5.

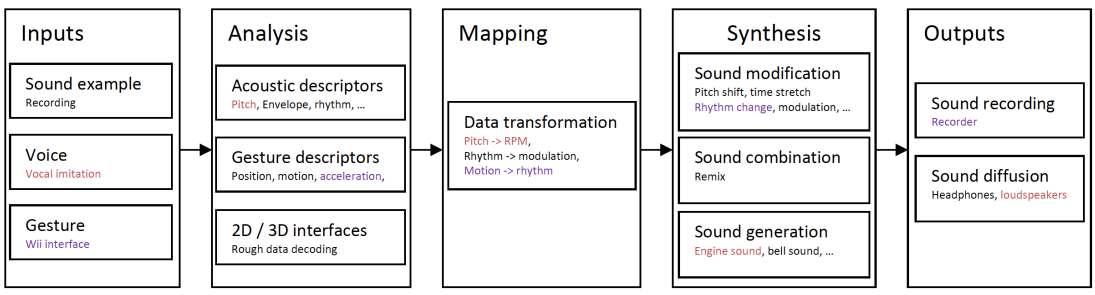

Figure 13: The SkAT-Studio workflow.

3.2 Downloading and installing

3.2.1 System requirements

SkAT Studio runs on Windows PCs with Max 6.1 or above. You can download and install Max from the official Cycling '74 website: `http://www.cycling74.com`

3.2.2 Installation procedure

The software can be downloaded as a compressed archive on GitHub, at the following address:

`https://github.com/SkAT-VG/SkATStudio/releases/tag/V1.0`

After unpacking the archive in the `Packages` folder of your Max installation, double click on the included file `libraries/FTM.2.5.0.BETA.23.exe` and install the FTM&Co library.

To run SkAT Studio open the main patch, named `SkatStudio_v1.0.maxpat`, which is reachable from the `Extras` menu.

3.3 Software overview

The framework is entirely developed in Max, and it is composed by a main GUI which can host and link together a collection of loadable *modules*, each one taking care of a specific operation in the global process. Several modules can be loaded simultaneously, and signal and/or control data can be routed at will among different modules.

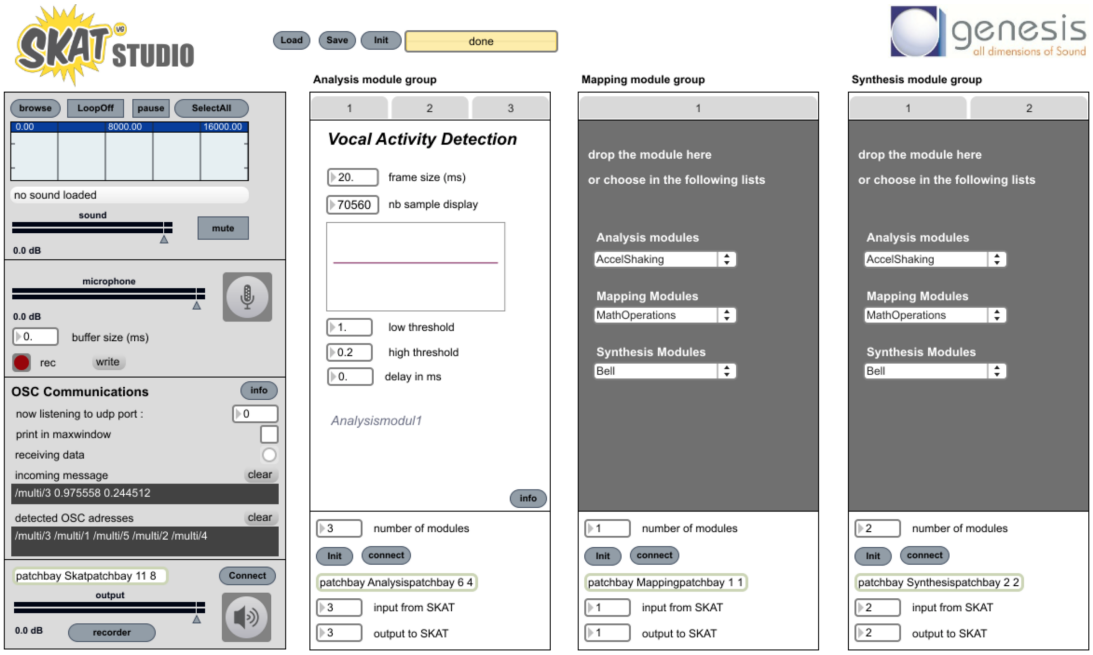

Figure 14: The SkAT-Studio main window.

3.3.1 Modules

Each module is realized as a Max patch and must adhere to a specific template. The SkAT Studio module template provides a common interface for back-end communication with the other parts of the framework and front-end integration into the main GUI. To comply with the template, modules must graphically fit a given area, and provide the following information: (highlighted in orange in figure 15):

- Name of the module,
- Number of inputs and outputs,
- Input and output labels,
- Documentation (input/output data types, author, description of the underlying algorithms and so on).

A wide variety of modules is already available in the SkAT Studio framework, offering the basic building blocks for the composition of complex configurations. Future integration of new features is going to be an easy task, thanks to the capabilities of the Max patching environment and to the simple yet versatile module template.

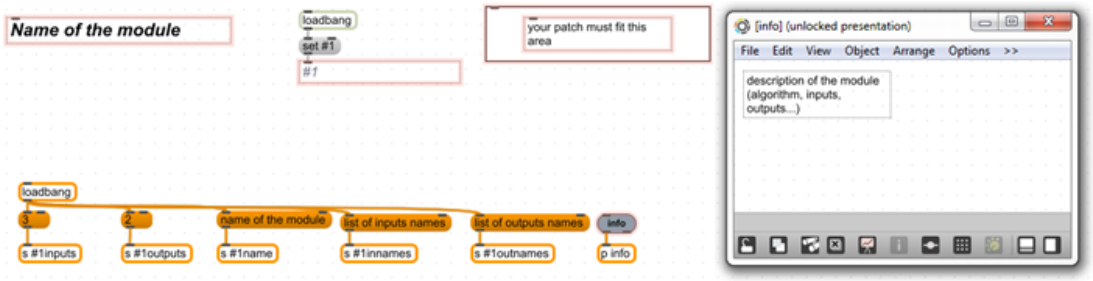

Figure 15: The SkAT-Studio module template.

3.3.2 Data routing: patchbays and groups

Data can be routed from any inputs to any outputs of each module. The communication is done using routing matrices called *patchbays*. A patchbay presents itself as a double entry table, as displayed in figure 16, with all the module outputs listed on the top row and all the module inputs listed on the left column. A toggle matrix allows to associate each output to one or more inputs, simply activating the appropriate toggles in the double entry table. To simplify the data routing process, modules are divided in *groups* reflecting the five stages of the *Play* process. Data is first routed among single modules inside the group, and then among different groups inside the main framework.

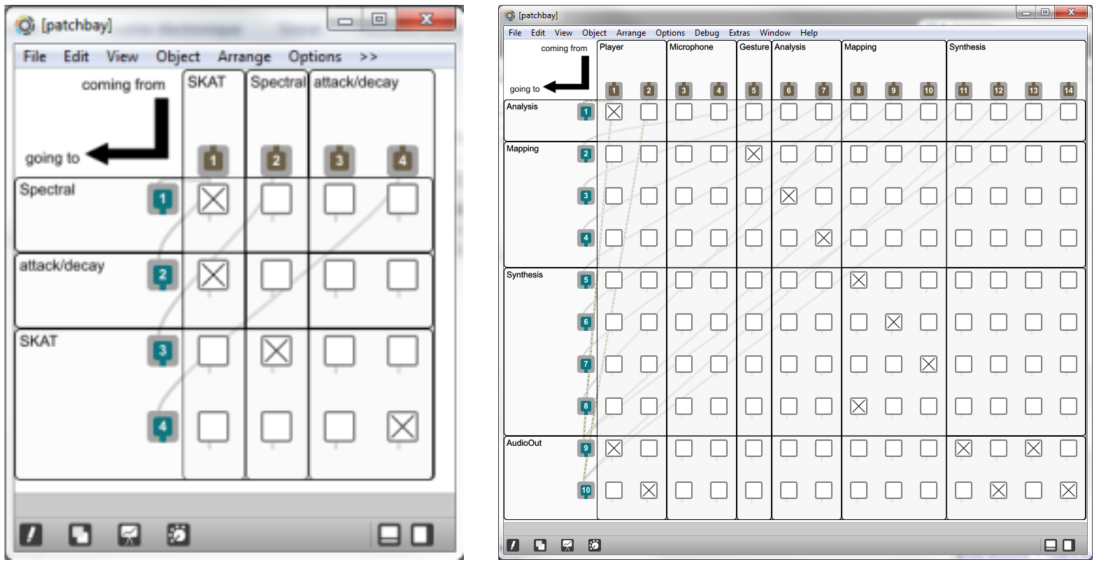

Figure 16: On the left, example of a patchbay for the analysis group. On the right, the global SkAT Studio patchbay.

3.3.3 Building a configuration

The procedure we describe thereafter is systematically illustrated with references to labels positioned on Figure 17.

To build a configuration, the successive steps are:

1. In each group, choose the number of modules to instantiate.

2. This operation creates as many tabs as required by the modules to be loaded.

3. In each tab, load the desired module. This may be done by drag and drop (from the Windows explorer) or by choosing a module in the list.

4. For each group, define the number of inputs and outputs that should be exposed to the other groups. By default, the number of inputs/outputs of a group is the total number of inputs/outputs of all the modules in the group.

5. Limiting this number allows to reduce the amount of data to route between groups, and therefore the size of the corresponding patchbays.

6. Click on connect to open the group patchbay and route data in each group.

7. Click on connect (top of the screen) to open the global patchbay and route data between groups.

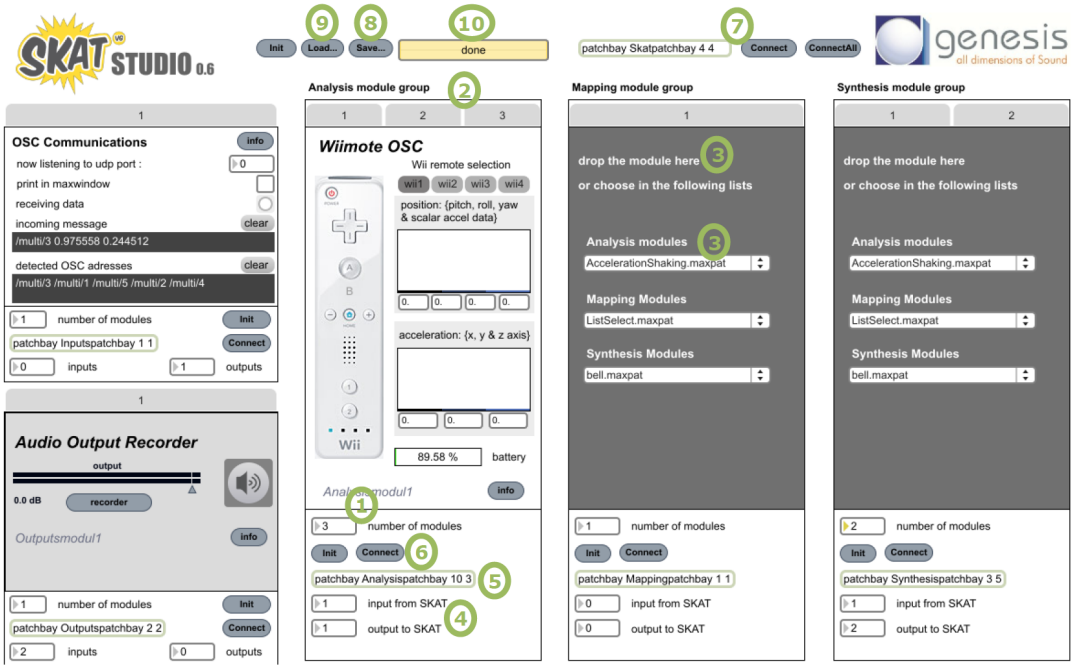

Figure 17: Building a SkAT Studio configuration.

3.3.4 Saving a configuration

Click on the *Save* button (8) to save a configuration. A dialog window allows choosing the saving folder and the name of the saving file. Configuration are saved as .json files.

3.3.5 Loading a configuration

Click on the *Load* button (9), then choose the .json file containing the desired configuration. The progress bar (10) shows the loading progress and prints *Done* when loading is finished.

3.4 Available modules

3.4.1 Input

Microphone Acquires vocal signals as real time control data.

Player Plays back sound files, useful for offline vocal control.

OSC Communications Acquires OSC data, such as sensor outputs in objects manipulated with hands, useful for live gestural control.

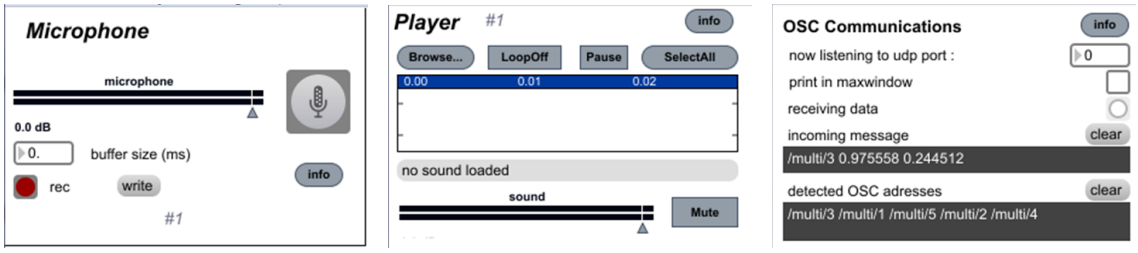

Figure 18: From left to right: The *Microphone*, *Player* and *OSC communications* input modules.

3.4.2 Analysis

Vocal Activity Detection Detects vocal activity in an audio signal, by means of calculations on signal energy in a specified temporal window.

Pitch detection Detects the fundamental frequency of a signal. Based on the YIN algorithm [dK02].

Attack/Decay analysis Detects attack, sustain, release and decay times of musical notes in the input signal. This module is particularly useful for music and singing voice.

Formant detection Detects the center frequencies of voice formants.

Partials frequency analysis Detects the partials of an input signal, namely its fundamental frequency and multiples.

Spectral peak extraction Extracts spectral peaks from the input signal.

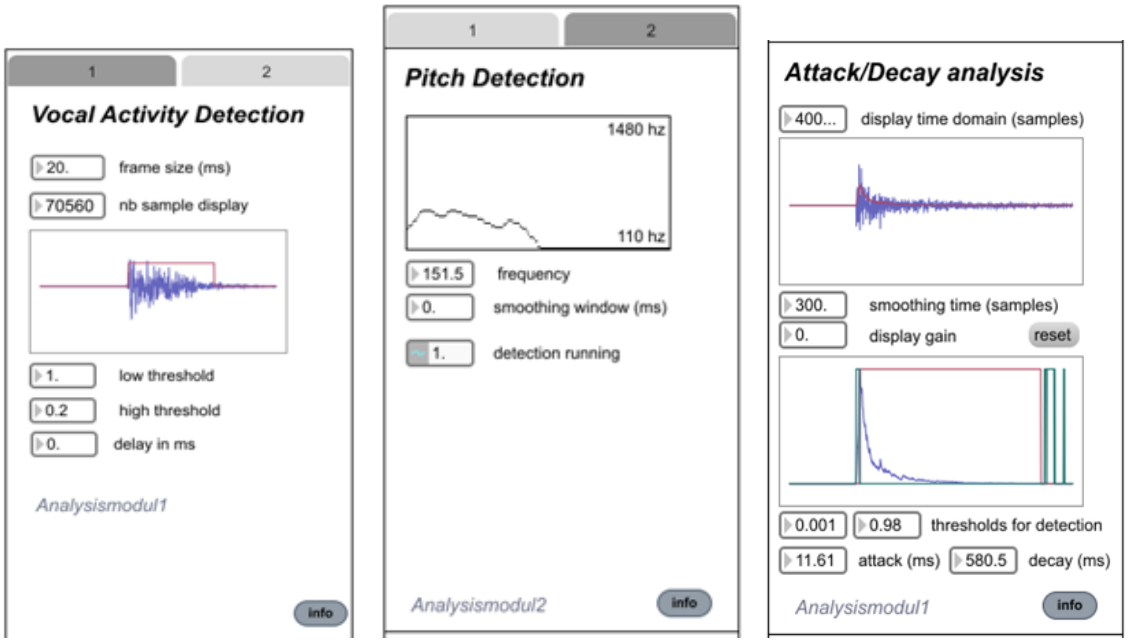

Figure 19: From left to right: The *Vocal Activity Detection*, *Pitch detection* and *Attack/Decay analysis* analysis modules.

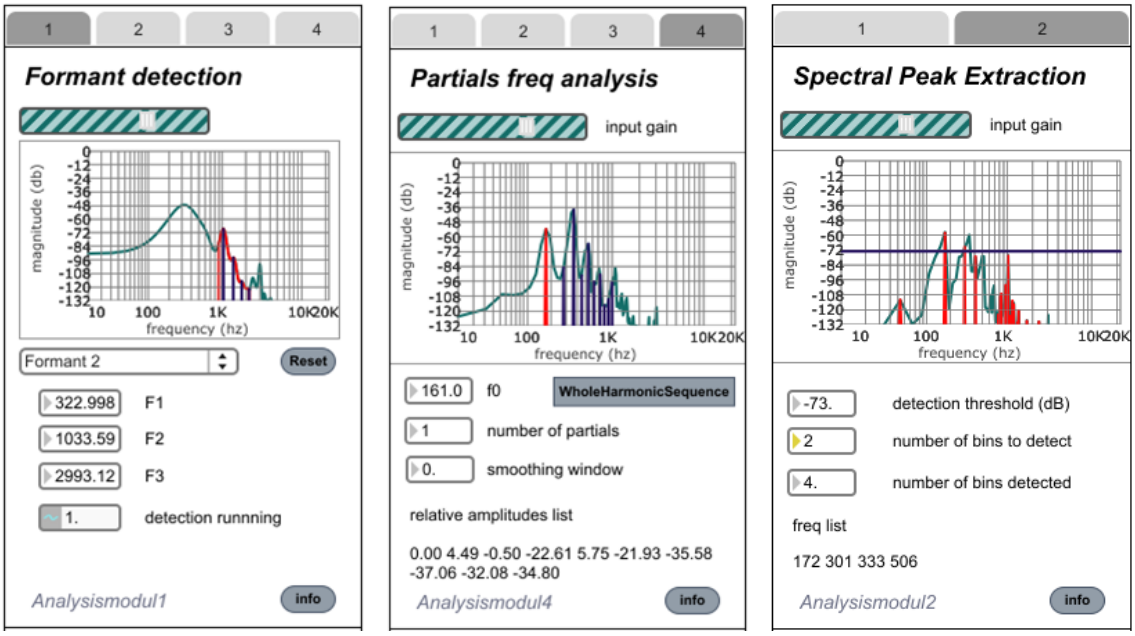

Figure 20: From left to right: The *Formant detection*, *Partials freq analysis* and *Spectral peak extraction* analysis modules.

Control OSC Used with the Control app[9] on an android or IOS device, it detects accelerometer data and gyroscope data or XY coordinates on the multitouch screen. In this last case, the coordinates can be offered in Cartesian or Polar form.

[9] `http://charlie-roberts.com/Control/`

Wiimote OSC This module has to be linked to the OSC input module of SkatStudio. Used with a Wiimote, it detects accelerometer data.

Shaking acceleration Detects acceleration peaks. A threshold can be defined to avoid detection of small fluctuations.

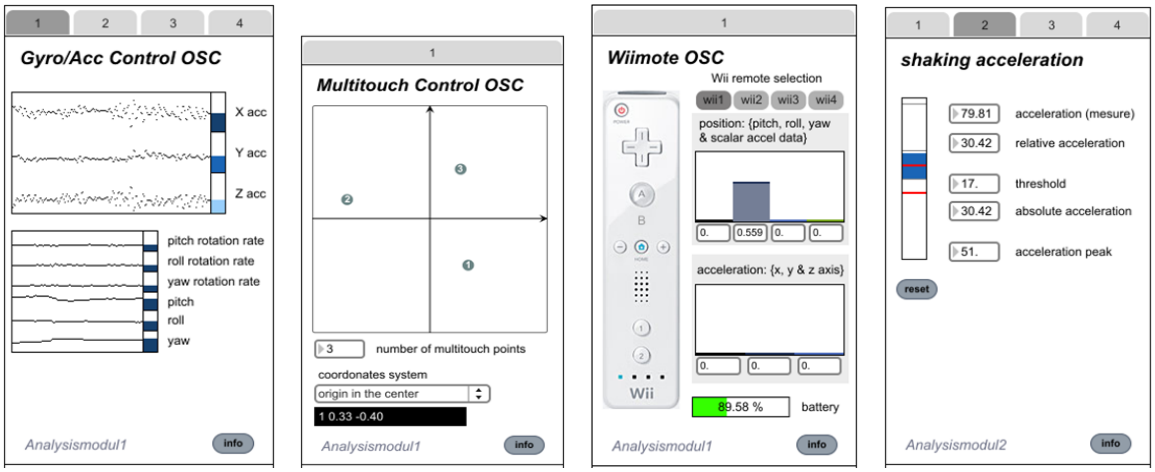

Figure 21: From left to right: The *Control OSC* (sensors and touchscreen), *Wiimote OSC* and *Shaking acceleration* analysis modules.

3.4.3 Mapping

Mapping Transforms data in three possible ways: graphical definition of a piecewise linear function (top), definition of a linear function by its coefficients (middle), definition of a custom function (bottom).

Scaling Allows to intuitively define of a linear mapping function, providing two examples of input data and their corresponding desired output values.

3.4.4 Synthesis

GeneCARS Synthesizes engine sounds, using the GeneCARS technology.

IUAV Engine Engine sound synthesizer, based on the sdt.motor~ sound model available in the SDT.

Bell Synthesis of bell-like sounds, based on the sdt.impact~ sound model, available in the SDT.

Impact Synthesis of impact sounds, using the Sound Design Toolkit models for basic mechanical interactions between solids.

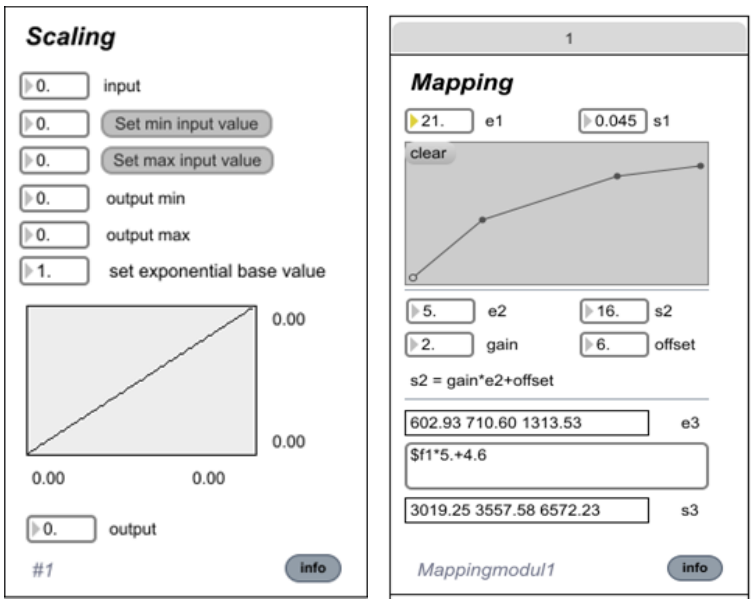

Figure 22: From left to right: The *Mapping* and *Scaling* mapping modules.

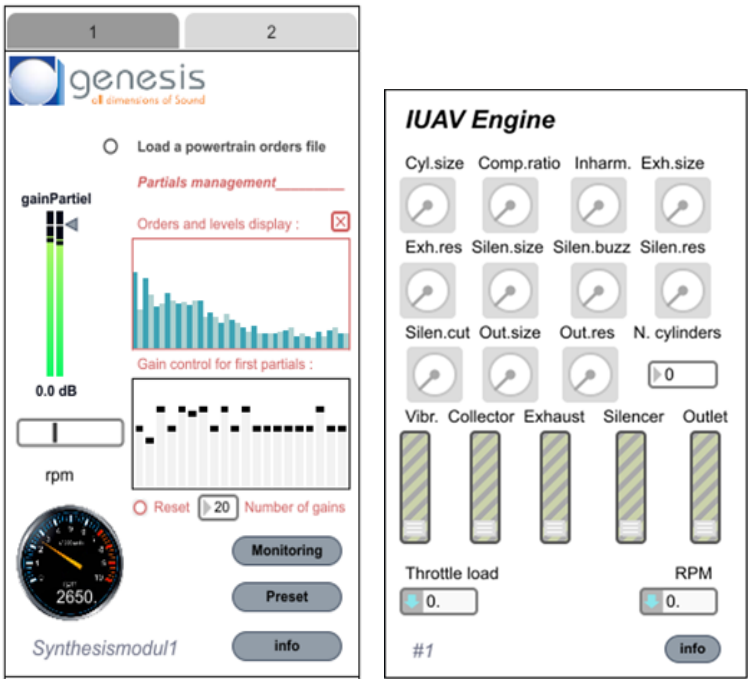

Figure 23: From left to right: The *GeneCARS* and *IUAV Engine* synthesis modules.

Water Synthesis of water sounds, using the Sound Design Toolkit fluid flow model.

Remix Four channel audio mixer.

VST Shaker This module is the integration of an external VST plugin, to demonstrate the feasibility of such integration.

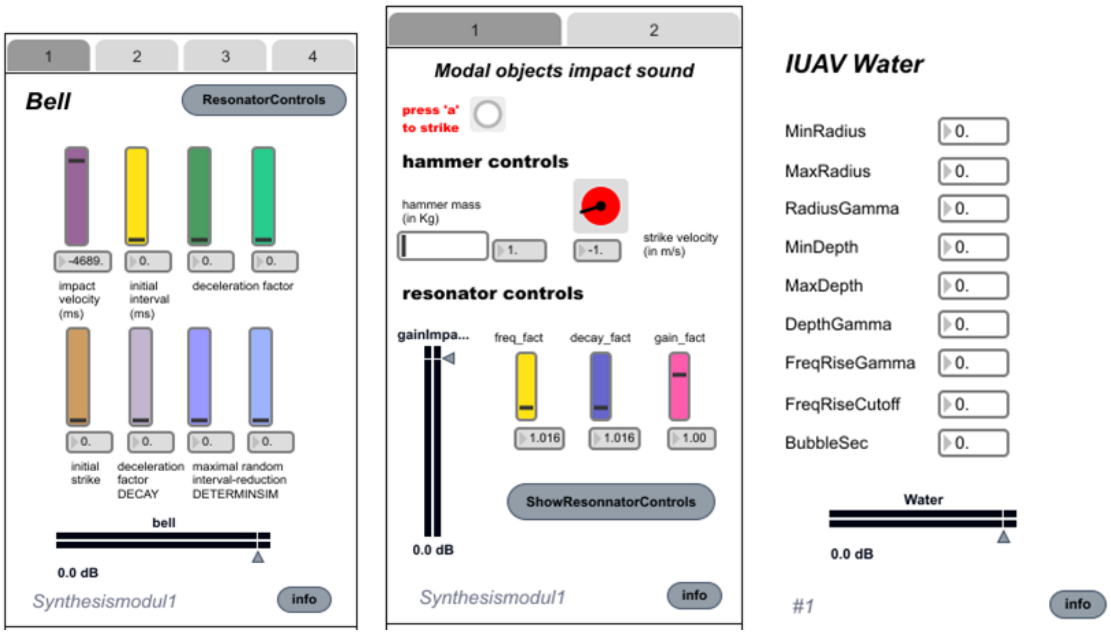

Figure 24: From left to right: The *Bell*, *Impact* and *Water* synthesis modules.

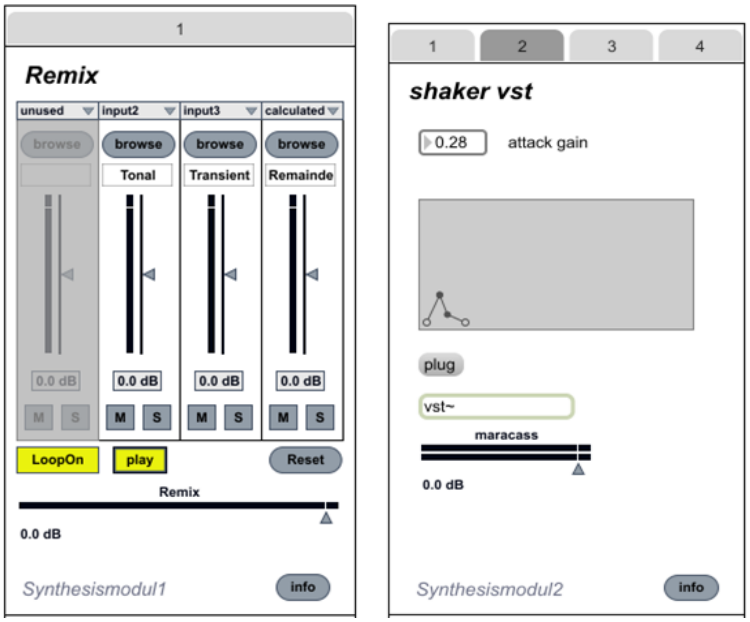

Figure 25: From left to right: The *Remix* and *Shaker VST* synthesis modules.

Parametric EQ Parametric equalizer.

Pitch shifter/Time stretcher Applies a pitch shift and/or a time stretch to a given sound.

Doppler effect Applies a doppler effect and a left/right spatialisation to a sound, to simulate moving sound sources.

Figure 26: From left to right: The *Parametric EQ*, *Pitch shifter/Time stretcher* and *Doppler effect* synthesis modules.

3.4.5 Output

Audio output Plays back the produced sounds on the default audio output device (loudspeakers, headphones, . . .).

Recorder This module allows to record the produced sounds to an audio file.

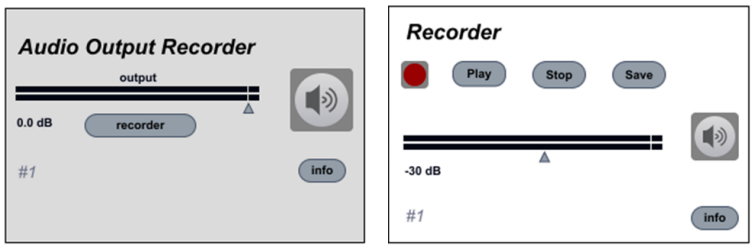

Figure 27: From left to right: The *Audio output* and *Recorder* output modules.

3.5 Configuration examples

The configuration examples described in the following paragraphs are available in the `presets` folder of the SkAT Studio package.

3.5.1 Genecars

In this configuration, the user controls the RPM of a car engine with her voice. The implementation in SkAT Studio is displayed in figure 28 and is composed of the following modules:

1. Voice is acquired through the microphone and used as a system input.
2. A Vocal Activity Detector coupled with a pitch estimator are then used to analyze the input signal and extract the fundamental frequency during vocal activity,
3. The pitch of the voice, expressed in Hz, is linearly mapped to the Revolutions Per Minute (RPM) of the engine,
4. The RPM value is used to control Genesis' GeneCARS module, which synthesizes the engine sound,
5. Synthesized sound is routed to the speaker output.

Please note how the GeneCARS configuration workflow reflects the general five-point structure of the *play* step of the SkAT Studio framework.

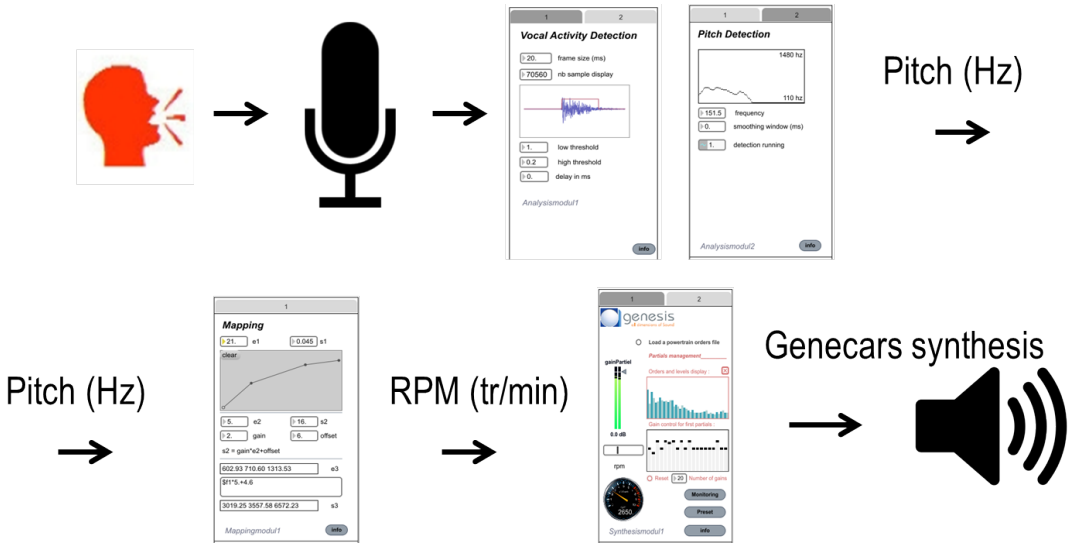

Figure 28: Implementation of the control of a car engine with voice.

3.5.2 Bell

By shaking her smartphone or a Nintendo Wiimote, the user controls a bell sound as if it had a real bell in her hand. The implementation in SkAT Studio is displayed in figure 29: The communication between the smartphone (or the Nintendo Wiimote) and SkAT Studio is done using OSC format, via bluetooth or wifi. OSC data are decoded to get the acceleration, which is analyzed to look for peaks. When a peak is detected, its value is used as input to control the strength of the impact on the bell.

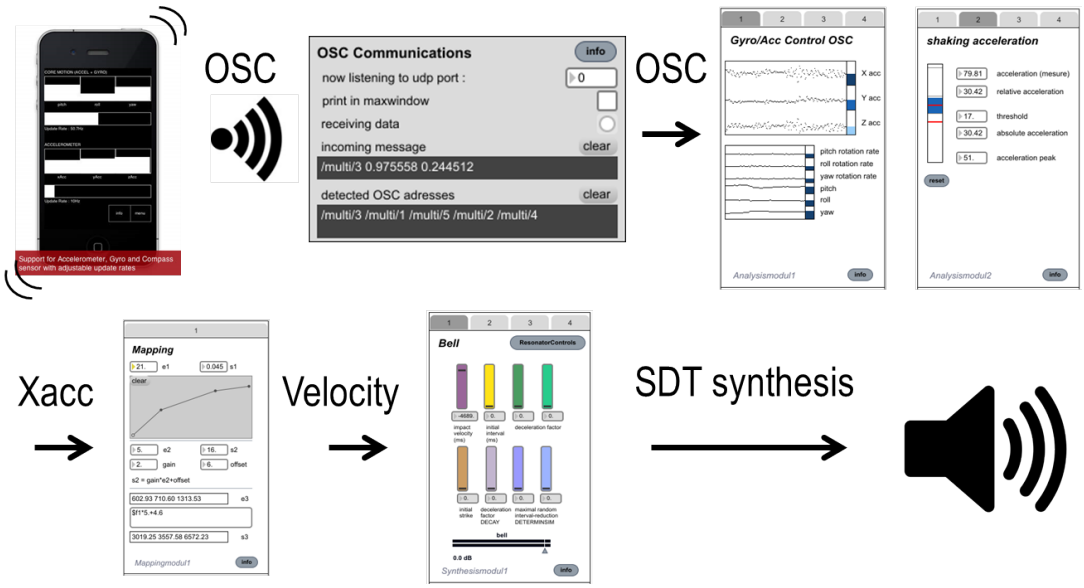

Figure 29: Implementation of the control of a bell sound with gesture.

3.5.3 Remix

By moving her finger on a 2D device (for example a smartphone or a tablet), the user dynamically controls the mixing of three sounds: the 2D position directly controls the level of two sounds, while the third one is implicitly computed to keep constant the overall level. The implementation in SkAT Studio is displayed in figure 30: the communication between the smartphone (or the tablet) and SkAT Studio is done using OSC format, via bluetooth or wifi. The 2D position (between -1 and 1) is directly used as a control for the mixing levels. The mixed sounds are the separation of the tonal, transient and noisy parts of a diesel engine with a turbo whistling.

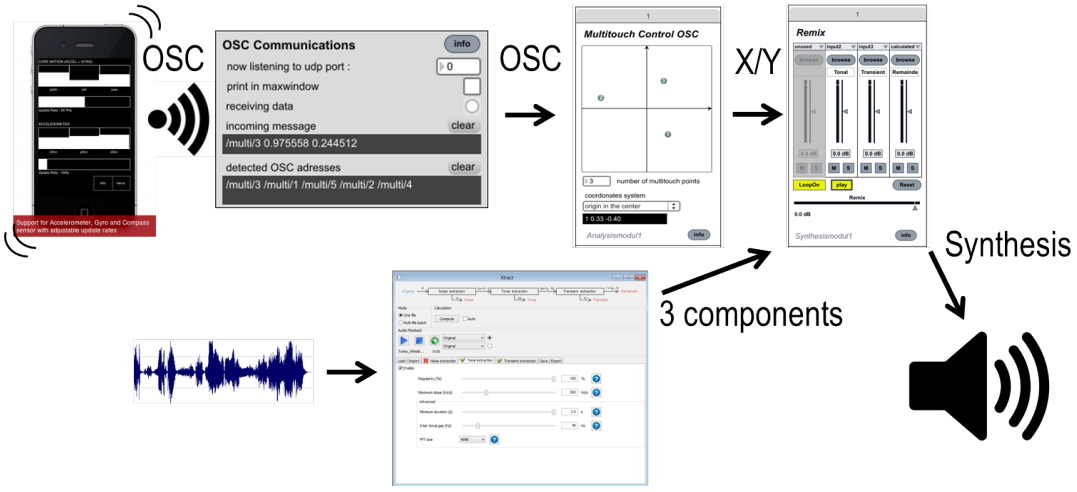

Figure 30: Implementation of the 2D remixing configuration.

References

[BLDMB15] Stefano Baldan, Hélène Lachambre, Stefano Delle Monache, and Patrick Boussard. Physically informed car engine sound synthesis for virtual and augmented environments. In *Proceedings of the IEEE 2nd VR Workshop on Sonic Interactions for Virtual Environments*, Arles, France, 2015.

[dK02] A. de Cheveigné and H. Kawahara. YIN, a fundamental frequency estimator for speech and music. *JASA*, 111(4):1917–1930, 2002.

[Doe05] Kees van den Doel. Physically based models for liquid sounds. *ACM Trans. Appl. Percept.*, 2(4):534–546, October 2005.

[DPR10] Stefano Delle Monache, Pietro Polotti, and Davide Rocchesso. A toolkit for explorations in sonic interaction design. In *Proceedings of the 5th Audio Mostly Conference: A Conference on Interaction with Sound*, AM '10, pages 1:1–1:7, New York, NY, USA, 2010. ACM.

[DR14] Stefano Delle Monache and Davide Rocchesso. Bauhaus legacy in research through design: The case of basic sonic interaction design. *International Journal of Design*, 8(3):139–154, 2014.

[Far10] Andy Farnell. *Designing sound*. Mit Press Cambridge, 2010.

[Gav93] William W Gaver. What in the world do we hear?: An ecological approach to auditory event perception. *Ecological psychology*, 5(1):1–29, 1993.

[HLM+12] Olivier Houix, Guillaume Lemaitre, Nicolas Misdariis, Patrick Susini, and Isabel Urdapilleta. A lexical analysis of environmental sound categories. *Journal of Experimental Psychology: Applied*, 18(1):52, 2012.

[HS96] Paul A Houle and James P Sethna. Acoustic emission from crumpling paper. *Physical Review E*, 54(1):278, 1996.

[LDSA11] Guillaume Lemaitre, A. Dessein, Patrick Susini, and K. Aura. Vocal imitations and the identification of sound events. *Ecological psychology*, 23(4):267–307, 2011.

[LHMS10] Guillaume Lemaitre, Olivier Houix, Nicolas Misdariis, and Patrick Susini. Listener expertise and sound identification influence the categorization of environmental sounds. *Journal of Experimental Psychology: Applied*, 16(1):16, 2010.

[LJH+15] Guillaume Lemaitre, Ali Jabbari, Olivier Houix, Nicolas Misdariis, and Patrick Susini. Vocal imitations of basic auditory features. In *169th Meeting of the Acoustical Society of America*, Pittsburgh (USA), 18-22 May 2015.

[MW05] Phillip McLeod and Geoff Wyvill. A smarter way to find pitch. In *Proceedings of International Computer Music Conference, ICMC*, 2005.

[PGS+11] Geoffroy Peeters, Bruno L. Giordano, Patrick Susini, Nicolas Misdariis, and Stephen McAdams. The timbre toolbox: Extracting audio descriptors from musical signals. *Journal of the Acoustical Society of America*, 130(5):2902, 2011.

[RBDM15] Davide Rocchesso, Stefano Baldan, and Stefano Delle Monache. Reverberation still in business: thickening and propagating micro-textures in physics-based sound modeling. In *Proceedings of the 18th International Conference on Digital Audio Effects (DAFx-15)*, Trondheim, Norway, 2015.

[RF03] Davide Rocchesso and Federico Fontana. *The sounding object*. Mondo estremo, 2003.

[Roc97] Davide Rocchesso. Maximally diffusive yet efficient feedback delay networks for artificial reverberation. *Signal Processing Letters, IEEE*, 4(9):252–255, 1997.

[Roc14] Davide Rocchesso. Sounding objects in europe. *The New Soundtrack*, 4(2):157–164, 2014.

[RPD09] Davide Rocchesso, Pietro Polotti, and Stefano Delle Monache. Designing continuous sonic interaction. *International Journal of Design*, 3(3):13–25, 2009.

[SP07] Dan Stowell and Mark Plumbley. Adaptive whitening for improved real-time audio onset detection. In *Proceedings of the International Computer Music Conference (ICMC 07)*, Copenhagen, Denmark, 2007.

[WV84] William H Warren and Robert R Verbrugge. Auditory perception of breaking and bouncing events: a case study in ecological acoustics. *Journal of Experimental Psychology: Human perception and performance*, 10(5):704–712, 1984.

[Zoe02] Udo Zoelzer, editor. *Dafx: Digital Audio Effects*. John Wiley & Sons, Inc., New York, NY, USA, 2002.