DADA: NON-STANDARD USER INTERFACES FOR COMPUTER-AIDED COMPOSITION IN MAX

Daniele Ghisi
STMS Lab (IRCAM, CNRS, UPMC)
Conservatory of Genoa
danieleghisi@bachproject.net

Carlos Agon
STMS Lab (IRCAM, CNRS, UPMC)
carlos.agon@ircam.fr

ABSTRACT

This article introduces the *dada* library, providing Max with the ability to organize, select and generate musical content via a set of graphical interfaces manifesting an interactive, explorative approach. Its modules address a range of scenarios, including, but not limited to, database visualization, score segmentation and analysis, concatenative synthesis, music generation via physical or geometrical modelling, wave terrain synthesis, graph exploration, cellular automata, swarm intelligence, and videogames. The library is open-source and extendable; similarly to *bach*, it fosters a performative approach to computer-aided composition (as opposed to traditional off-line techniques): the outcome of all its interfaces can be recorded in scores, or used in real time to drive, among other things, digital signal processes, score transformations, video treatments, or physical actuators.

1. INTRODUCTION

Real-time computer-aided composition is a relatively recent and promising field of study. In particular, the development of the *bach* library [1] for Max [2] has made possible to operate on symbolic scores scores as interactively as on sound buffers. Although *bach* features a certain number of interactive, graphical objects, all of them essentially implement established representations of music, be they traditional scores or alternative but widespread representations such as the clock diagram or the Tonnetz [3]. This is both a strength and a limitation: it is a strength, inasmuch as it allows *bach* to be a general-purpose, highly adaptable tool; it is a limitation, inasmuch as it limits the scope of *bach* as a toolbox for experimental, non-standard musical practices and research.

This article introduces a new library, *dada*, based on the *bach* public API, meant to fill this gap, focusing on real-time, non-standard graphical user interfaces for computer-aided composition. Hence, most of the modules in *dada* are interactive user interfaces; nonetheless the library also features a small number of non-UI modules designed to complement the operation of some of the interfaces in the library. The *dada* library is the third library in the "*bach* family" [4] (the *cage* library being the second [5]). It is part of the PhD thesis of one of the authors, and although it has been widely used in his recent musical production, for lack of space, this article will not describe such examples of usages (while the full PhD thesis does [6]).

2. MOTIVATION AND RATIONALE

The philosophy behind *dada* is profoundly different from the one which informed *bach*: *dada* is to *bach* what a laboratory is to a library. Under the umbrella of non-standard, strictly two-dimensional graphic user interfaces, all of its components participate of a ludic, explorative approach to music; most of its components also refer to the fields of plane geometry, physical modelling or recreational mathematics.

A preliminary alpha version of *dada* (0.1) is available on its official website [1] . The modules included in the *dada* library can be roughly divided into three categories: tools for corpus-based composition (including database interfaces and score analysis mechanisms), tools for physical or geometrical modelling of music (including gravity-based models, pinballs, kaleidoscopes and wave terrain synthesis), and tools to handle rule-based systems and games (including cellular automata, swarm intelligence models and platform videogames). Before providing, in the next few sections, a detailed overview of the modules, we would like to motivate our development choices.

Differently from *bach* and *cage*, *dada* is a personal library, tailored on the compositional needs of one of the authors. Essentially all implementation choices have been taken with this consideration in mind, a fact that is most notable in some specific modules (such as *dada.bodies* or *dada.music*~). In other words, the choice of what to develop has not been influenced by the needs of the computer music community, but rather by a very personal effort to experiment with geometry-based musical ideas.

That being said, *dada* is by design an open box: it is open-source [2] , and we hope that other interested musicians and developers will contribute with new modules. Such additions will be facilitated by the *dada* API, implementing a set of common operations (to provide, among other things, support for graphic display, selection handling and undo mechanisms).

[1] http://www.bachproject.net/dada
[2] https://github.com/bachfamily/dada

It should be remarked that most processes in *dada* are not new—some of them have also been implemented and distributed as third-party Max externals. In particular: the portion of the library dealing with database visualization has been inspired by the CataRT library for concatenative synthesis [7] (which has two different Max implementations [8]); the swarm intelligence module relates to the Boids library [3]; the wave terrain synthesis module relates to WAVE [4] and to Stuart James's work [9]; and there is a large number of implementations of cellular automata in Max, including Bill Vorn's Life Tools [5]. There are, however, two good reasons for our choice to re-implement these tools inside *dada*.

Firstly, all *dada* modules follow the *bach* paradigm of real-time computer-aided composition [4]. The contribution of *dada* is hence novel, inasmuch as it builds on top of the rich hierarchical representation and algorithmic manipulation afforded by *bach* and *cage*, integrating its processes within a single, unified, coherent system. As an example, all *dada* modules are designed to be easily used in combination with *bach.ezmidiplay*, to obtain a quick MIDI rendering of the musical outcome, and with *bach.transcribe*, to record the outcome in proportional notation in a *bach.roll*.

Secondly, the *dada* implementation is more general, more customizable or has a different scope. As an example, the *dada.catart* module, a two-dimensional interfaces of datasets, differently from CataRT is not limited to audio datasets; on the contrary, it is able to organize on the cartesian plane entries of a generic SQLite database—and its focus is, most notably (but not uniquely), on score datasets, providing mechanisms to segment and analyze symbolic scores. As another example, the *dada.boids* object, differently from the Boids library, allows for customized rules to be set via snippets of C code, compiled on-the-fly. The same is true for *dada.life*, dealing with cellular automata.

3. TOOLS FOR CORPUS-BASED COMPOSITION

The tools in this category are primarily designed to handle scores databases, but can be more generally applied to the creation and visualization of general datasets. Some of the modules in this category were already introduced in [10], and have been, since then, improved and extended.

The overall system relies on four different modules: *dada.segment*, performing score segmentation and feature extraction; *dada.base*, implementing the actual database engine; *dada.catart* and *dada.distances*, two-dimensional graphic user interfaces capable of organizing and interacting with the extracted grains.

3.1 Segmentation

The *dada.segment* module performs the segmentation of a score, contained in a *bach.roll* (as proportionally notated musical data) or a *bach.score* (as classically notated musical data, see Figure 1), in one of the following manners: using the markers in the original score as slice points; defining an equation for the size of each grain; using labels assigned to notes and chords (outputting one grain for each label).

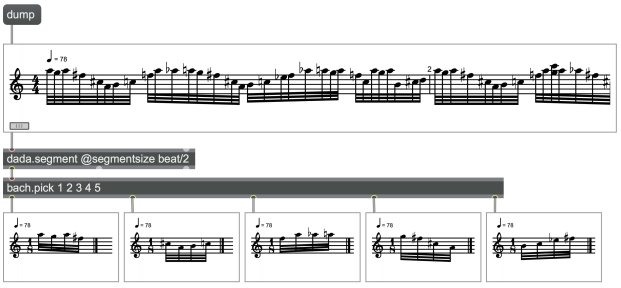

Figure 1. Segmentation of a *bach.score* into grains having length equal to half of the beat (i.e. an eighth note).

The segmentation can be carried on with overlapping windows, both on proportional and classically notated scores, and standard windowing techniques can be applied to MIDI velocities, if desired.

3.2 Analysis

Grain analysis is performed during the segmentation process. On one side, *dada.segment* is capable of adding some straightforward metadata to the segmented grains, such as their duration, onset, index, label (if segmentation is carried out via label families) and notation object type (either 'roll' for *bach.roll* or 'score' for *bach.score*); in case the grain comes from a *bach.score*, tempo, beat phase, symbolic duration and bar number can also be added.

On the other hand, *dada.segment* allows the definition of custom features via a loopback patching configuration named "lambda loop" [11]: grains to be analyzed are output one by one from the rightmost outlet, preceded by the custom feature name; the user should provide a subpatch to extract the requested feature, and then plug the result back into *dada.segment*'s rightmost inlet. Feature names, defined in an attribute, are hence empty skeletons which will be "filled" by the analysis implementation, via patching. This programming pattern is widely used throughout the *bach* library (one can compare the described mechanism, for instance, with *bach.constraints*'s way of implementing custom constraints [1]), and allows users to implement virtually any type of analysis on the incoming data.

Some ready-to-use abstractions are provided for quick prototyping, whose terminologies are mostly borrowed from the audio domain, even if they are applied to symbolic data; hence *dada.analysis.centroid* will output an average pitch, *dada.analysis.spread* will output the standard deviation of the pitches, and so on. The reason behind this choice is to underline the duality between this symbolic framework and the digital signal processing approach. Moreover, since analysis modules are standard Max patchers, it is easy for users to inspect and adapt them to different behaviors.

Analyzed features are collected for each grain, and output as metadata from the middle outlet of *dada.segment*.

[3] http://s373.net/code/
[4] http://www.noisemaker.academy/blog/
[5] http://billvorn.concordia.ca/research/software/lifetools.html

3.3 Database

Once the score grains have been produced and analyzed, they are stored in a SQLite database, whose engine is implemented by the *dada.base* object. Data coming from *dada.segment* are properly formatted and fed to *dada.base*, on which standard SQLite queries can be performed.

Some higher-level messages are provided to perform basic operation and to handle distance tables (i.e. tables containing distances between elements in another table, useful, for instance, in conjunction with the *dada.distances* module, as explained below).

Databases can be saved to disk and loaded from disk.

3.4 Interfaces

Two objects provide graphic interfaces for the database: *dada.catart* and *dada.distances*.

Figure 2. The *dada.catart* object displaying a database of score fragments. Each element of the database (grain) is represented by a circle. On the horizontal axis grains are sorted according to the spread, while on the vertical axis grains are organized according to their centroid. The colors scale is mapped on the grain onsets in the original file, while the circle size represents the grain loudness.

The *dada.catart* module provides a two-dimensional Cartesian graphic interface for the database content. Its name is an explicit acknowledgment to the piece of software which inspired it [7]. Grains are by default represented by small circles in a two dimensional plane. Two features can be assigned to the horizontal and vertical axis respectively; two more features can be mapped on the color and size of the circles. Finally, one additional integer valued feature can be mapped on the grain shape (circle, triangle, square, pentagon, and so forth), adding up to a total number of five features being displayable at once (see Figure 2).

The *dada.distances* module provides a distance-based representation of the database content. Points are the entries of a table, characterized via their mutual distances, contained in a different table. They are represented in a two-dimensional plane via the multidimensional scaling algorithm provided by [12]. Edges are drawn only if the corresponding distance is below a certain threshold (see Figure 3). The resulting graph is navigable in a Markov-chain fashion, where distances are interpreted as inverse probabilities. As for *dada.catart*, features can be mapped to colors, sizes and shapes.

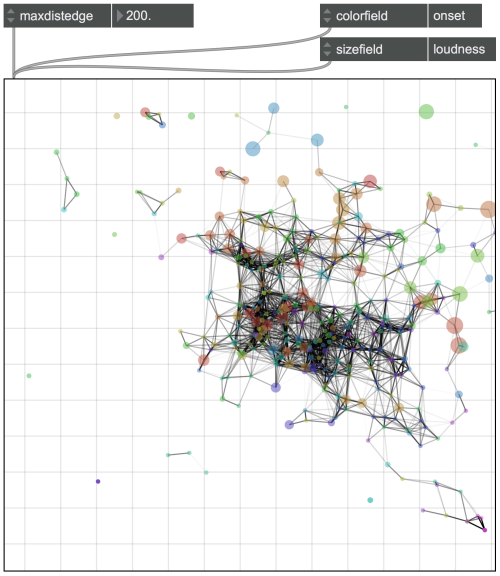

Figure 3. The *dada.distances* object displaying a database of score fragments. As for the *dada.catart* case (Figure 2), each element is represented by a circle. Grains are only positioned only according to a certain defined distance function (in this case, the distance of their centroids, spreads and loudnesses, as tridimensional vectors), the positioning in the plane is carried out via multidimensional scaling.

Both in *dada.catart* and in *dada.distances* each grain is associated with a "content" field, which is output either on mouse hovering or on mouse clicking. The content is usually assigned to the *bach* list representing the score. The sequencing can also be beat-synchronous, provided that a tempo and a beat phase fields are assigned: in this case the sequencing of each grain is postponed in order for it to align with the following beat, according to the current tempo (obtained from the previously played grains).

A *knn* message allows to retrieve the k nearest samples for any given (x, y) position. A system of messages inspired by turtle-graphics is also implemented, in order to be able to move programmatically across the grains: the *setturtle* message sets the turtle (displayed with an hexagon) on the nearest grain with respect to a given (x, y) position; then the *turtle* message moves the turtle of some $(\Delta x, \Delta y)$, choosing the nearest grain with respect to the new position (disregarding the original grain).

The database elements can be sieved by setting a *where* attribute, implementing a standard SQLite 'WHERE' clause. The vast majority of the display features can be customized, such as colors, text fonts, zoom and so on. In combination with standard patching techniques, these features also allow the real-time display, sequencing and recording of grains.

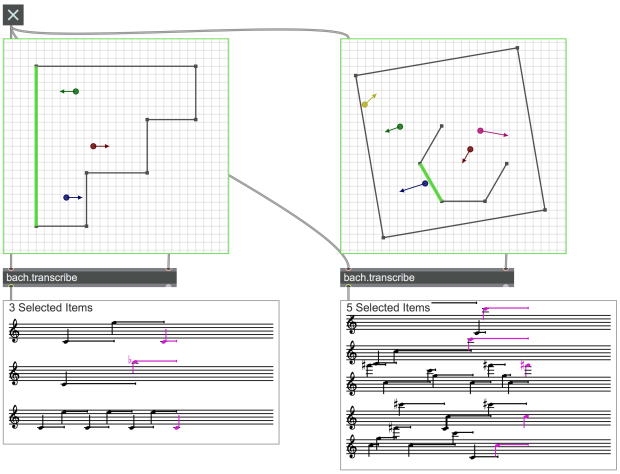

Figure 4. Two *dada.bounce* objects producing respectively a polyrhythm (left) and a more complex pattern (right). Each edge is mapped on a note which can be played or recorded as soon as the the edge is hit.

4. TOOLS FOR PHYSICAL OR GEOMETRICAL MODELLING OF MUSIC

The interfaces in this group share the idea that objects in space can lead to music generation by means of geometry and motion.

4.1 Pinball-like bouncing

The *dada.bounce* module suggests a pinball-like scenario, where a certain number of balls move inside a space delimited by a user defined graph, called "room". The ball movement is uniform (constant speed [6], no gravity), except when a ball bounces off an edge. Each edge contains metadata either as a couple of MIDI pitch and velocity, or as a complex score; such metadata will be output whenever a ball hits the corresponding edge. Information about the collision (identifying the point, the edge and the ball) can be retrieved. Ball and room properties and metadata can be changed dynamically.

Simple room configurations may lead to loops or polyrhythmic patterns; more complex results are achievable by modifying the geometry of the room and the number of balls (see Figure 4), or by using feedback loops as programming patterns—e.g., by adding edges at each hit.

4.2 Gravitation

A different paradigm is enforced by the *dada.bodies* module, modelling a two-dimensional universe with gravity, containing two types of objects: "stars", fixed circles, from which a certain number of radii stand out, each representing a note (see Figure 5); and "planets", which orbit around the stars according to a customizable gravitational law, triggering the playback of radial notes whenever they orbit "close enough" to a star. The MIDI velocities are scaled according to the distances between planets and stars. As

[6] In order to avoid confusion with MIDI velocities, the term "speed" is used in this context also to refer to the velocity vector, and not just to its scalar intensity.

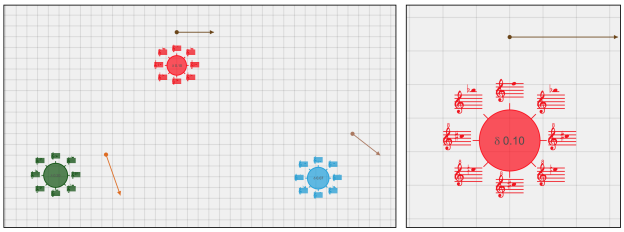

Figure 5. Configuration of *dada.bodies* gradually distorting the loops of Gerard Grisey's *Vortex temporum*. At right: a zoomed version of one of the stars (corresponding to the flute's notes).

a metaphor, one could imagine "stars" as being "radial aeolian harps", played by the planets whenever they circle around them.

This model is a convenient representation to handle continuous modification of loops. In a situation with a single star and a single planet, one could set the distances and speeds so that the planet motion around the star is circularly uniform (convenience methods are provided), resulting in a perfectly looping pattern. Modifying the planet position or speed, ever so slightly, results in a time warping operation on the loop. Adding more stars will trigger complex scenarios. Chaotic loops and attractor-like situations can be achieved via this system.

4.3 Kaleidoscopes

The *dada.kaleido* module traces the disposition and movement of a certain number of polygons in a kaleidoscope-like container. A certain number of shapes (polygon or ellipses) are positioned inside a 2- or 3-mirror chamber. The 2-mirror chamber has a couple of mirrors of equal length hinged at the origin, producing circular "snowflake"-like patterns. The angle between the mirrors is set by the user via the *count* attribute, an integer number $n \geq 2$ relating to the mirror angle α in the following way: $\alpha = \pi/n$: for $n = 2$ mirrors are at right angles, for $n = 3$ they are two sides of an equilateral triangle, and so on (see Figure 6). For $n = 2$ and $n = 3$, a third mirror can be introduced [13, p. 210], closing the triangle formed by the other two, hence extending the tiling to the whole plane.

The shapes inside the chamber can be modified either via the interface or via a set of messages, such as 'move', 'rotate', 'scale' and 'shake'. A combination of rotation with a certain amount of shaking will result in an elementary yet effective modelling of a hand rotating the body of a kaleidoscope.

Users can assign test points on the plane, so that the object may report whenever any of the polygons, during a movement, hits a point (i.e. when the point enters a polygon or any of its kaleidoscopic reflections) or releases a point (i.e. when the point is no longer on the polygon, or on any of its kaleidoscopic reflections). Information about the distances between test points and polygons can also be retrieved, and can be used as control for symbolic or DSP processes. As an example of application, one might associate each shape with a portion of audio file, which, like a

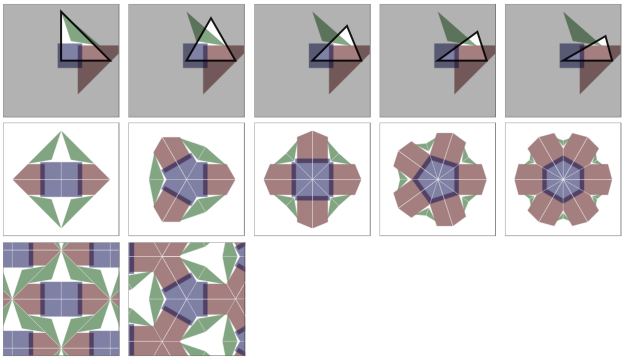

Figure 6. Same shapes reflected into different chambers of a *dada.kaleido* object, for increasing values of the *count* attributes. Last row shows the 3-mirror version of the patterns, only available for $n = 2$ and $n = 3$.

vinyl, is only read, with variable speed, when a certain test point (the "stylus") is positioned over the shape.

4.4 Wave terrain synthesis

The *dada.terrain*∼ module implements wave terrain synthesis [14, pp. 163–167]: a function $z = f(x, y)$ yields the "height" of the terrain for each point of a plane. Evaluating the function f on a specific path $p : x = x(t), y = y(t)$ produces a one-dimensional function $z = g(t) = f \circ p(t)$, which represents the wave terrain synthesis along the path p. Wave terrain synthesis essentially constitutes an extension of the ordinary wavetable synthesis to bidimensional lookup tables, and it is traditionally implemented in this way, in order to lower computational costs. A typical scenario is when the surface f is a direct product of sinusoids, such as $f(x, y) = sin(n\pi x)cos(m\pi y)$: in this case, by sampling the terrain on circular or elliptic orbits p, one obtains FM-like timbres.

In the *dada.terrain*∼ module, the function $f(x, y)$ is however not defined via a wave table, and is set via an explicit portion of C code compiled on-the-fly (see Figure 7). The wave terrain is displayed so that black corresponds to $z = -1$, white corresponds to $z = 1$, and 50% grey corresponds to $z = 0$.

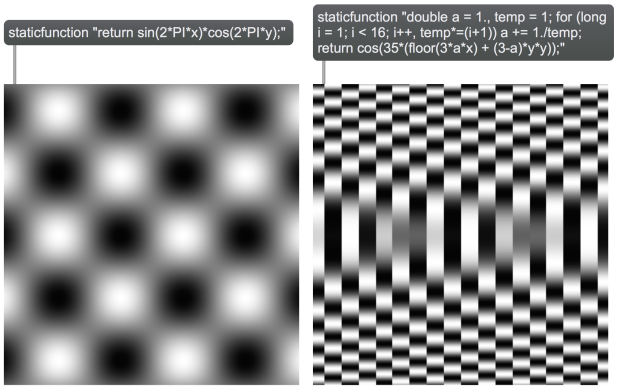

Figure 7. Two wave terrains displayed in *dada.terrain*∼.

Four auxiliary modules help producing specific paths, namely: segments, rectangles, ellipses and spirals; such modules produce coordinates at sample rate, to be used as input for the wave terrain module.

The *dada.terrain*∼ module also supports the a "buffer wheel" mode, where the terrain is the result of a morphing between radially arranged buffers. Such morphing could be additive (result being a simple crossfade) or multiplicative; the equation for the contribution of each buffer can be set as a portion of C code compiled on-the-fly. As an example, consider Figure 8, where four instruments playing the same notes are arranged radially, and a spiral path samples the wave terrain, yielding a morphing between the four sounds.

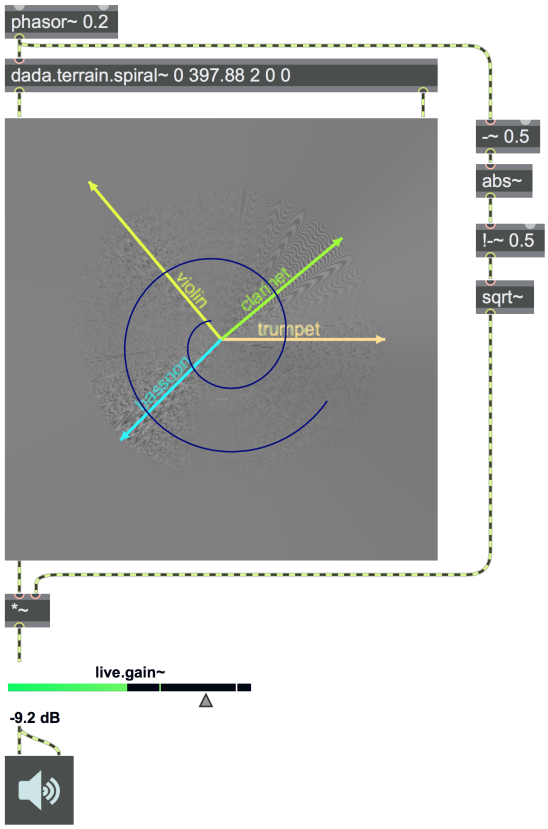

Figure 8. Four buffers, each containing an instrument playing an A3 in pianissimo, are arranged radially on a *dada.terrain*∼. The terrain is then sampled via a spiral path, yielding a morphing between the four sounds.

5. RULE-BASED SYSTEMS, GRAPHS, AND MUSIC AS A GAME

A certain number of tools explore the relationship between music, mathematics and games, and how this relationship ramifies towards combinatorics, algebra, topology and computer science (the link between canonical processes and topology being of course well known [15], further interesting examples can be found in tools such as origami [16] or juggling patterns [17] [7]).

[7] See for instance Tom Johnson: *Three notes for 3 jugglers* (2012).

The modules in this family share two important ideas. The first one is that interesting emergent behaviors may arise from dynamical systems even when their agents adhere to sets of extremely simple rules; this is well known, for instance, in the study of cellular automata, swarm intelligence and in Chaos Theory. The second idea is that digital scores may somehow be harbringers of a form of "gamification", i.e. the usage of game design elements in non-game scenarios.

After all, there are fundamental similarities between musical scores (in any form) and digital games [18]. Playing videogames often resolves in following a (graphically notated) rhythmical score, not dissimilar to a percussionist playing his or her own part in an orchestra: in both cases, the ability to stay within an acceptable level of precision affects the outcome. If the score is hard-coded, gamers can progressively learn the precise timing for their actions; if the score is open, gamers are obliged to play *a prima vista*.

5.1 Cellular automata

The first module in this family is *dada.life*: a graphical interface for two-dimensional cellular automata, on square or triangular grids. Cellular automata are rule-based systems, consisting of a regular grid of cells, each in one of a finite number of states (such as "alive" and "dead"). A set of cells called "neighborhood" is defined relative to each specific cell. Given a configuration of states, a new generation can be created according to a given rule, usually a mathematical function, determining the new state of a cell depending on the current states of the cells in its neighborhood. The most famous cellular automaton is arguably Conway's *Game of Life*. Extremely complex patterns can arise in cellular automata, even from simple rules.

A Max module handling two-dimensional cellular automata was already included in *cage* [5]; nevertheless the *dada.life* object improves the approach, by making it interactive, more customizable and faster. The customization possibilities are not limited to colors and sizes: rules themselves can be defined either via attribute combinations (for simple scenarios similar to Conway's *Game of Life*) or via a portion of C code, compiled on-the-fly—a more agile approach than *cage.life*'s Max patchers. [8]

Automata in *dada.life* can live on square or triangular lattices, such as the Tonnetz [3]. One can use the Tonnetz grid as basis for a two-states cellular automaton (see Figure 9): cells can be 'on' (playing) or 'off' (silent). Pattern hence result in musical sequences; for instance, oscillators (patterns that repeat after a finite number of steps) yield harmonic or melodic loops.

5.2 Swarm intelligence

The *dada.boids* module investigates swarm intelligence models. The object contains a certain number of "swarms" or "flocks", each containing a certain number of "birds" or "particles", singularly represented on the screen as points or arrows. The movement of each particle is dictated by a

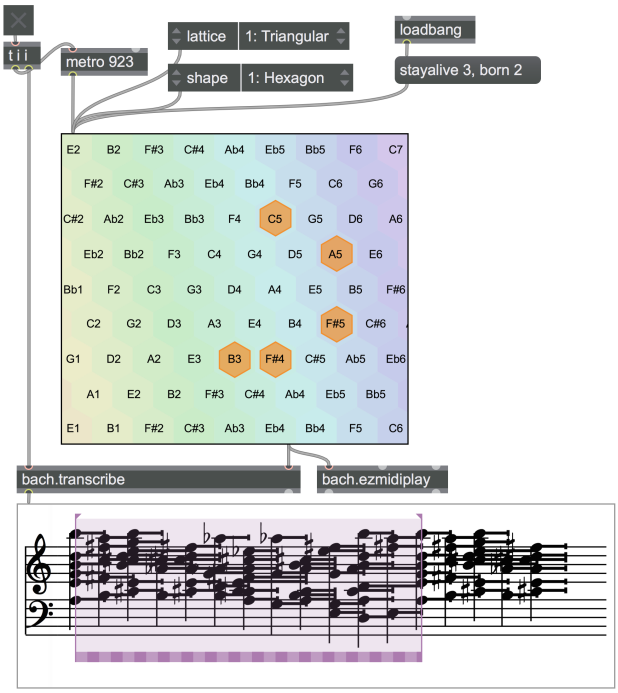

Figure 9. The harmonic cycle for the third movement in *Come un lasciapassare*, by one of the authors, as an oscillator of a two-dimensional cellular automata played on the Tonnetz.

sequence of higher-level rules, usually in the form of differential equations, accounting for the global behavior of the flock. Particles are traditionally called "boids" [19], a shortened version of "bird-oid objects".

In the traditional boids scenario, three rules apply: separation (particles steer to avoid crowding local flockmates), alignment (particles steer towards the average heading of local flockmates) and cohesion (particles steer to move toward the average position of local flockmates). The *dada.boids* module is able to account for such rules, as well as a for the presence of external barriers (obstacle avoidance) and winds. Moreover, each user can define his or her own set of rules, by compiling on-the-fly a portion of C code. Rules can have parameters, defining their position (such as the location of an obstacle), their orientation (such as the wind direction), their intensity (such as the wind speed, or the strength of a barrier), or, more generally, their behavior (such as a threshold for particle separation). Some of these parameters can also be associated to editable graphical user interface elements, such as points, vectors or lines—for instance, users can modify the direction of the wind by dragging the tip of the corresponding arrow, or the position of a barrier by dragging the corresponding horizontal or vertical line (see Figure 10).

In addition to their position and speed, particles can have a scalar intensity value, and custom rules can be set to modify intensities along with speeds. In practice, both built-in and user-defined rules are compiled functions that, for each particle, take as input its state, together with the state of the entire flock (coordinates, speeds and intensities of each particle), and yield as output, according to the current

[8] On the other hand, the fact that *cage.life* is an abstraction is consistent with the design of the whole *cage* project.

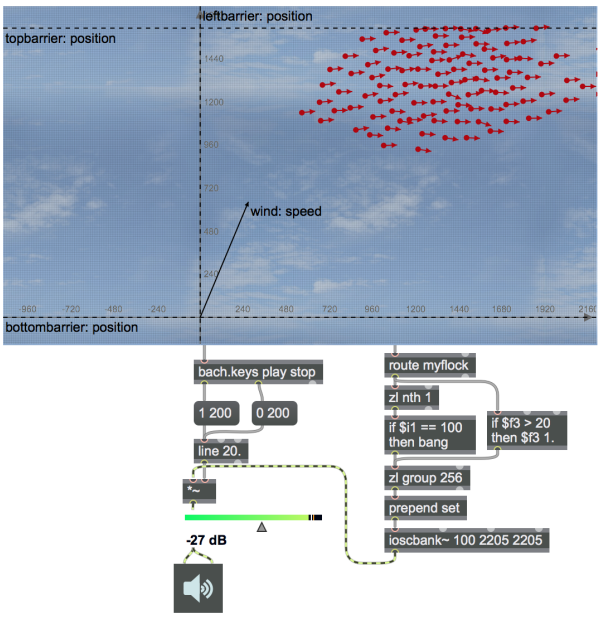

Figure 10. A *dada.boids* object where the vertical position of each boid is mapped on the frequency of a sinusoidal oscillator.

value of their parameters, a speed vector, to be added to the current particle speed (a "steering" vector), and possibly a value to be added to its intensity. By summing the contributions of all rules, one gets the discrete derivative of the particle speed (and intensity).

5.3 Graphs

The *dada.graph* module (see Figure 11) is a simple graph interface and editor, also featuring two automatic node placement algorithms provided by the Boost library [20]: the Fruchterman-Reingold force-directed layout [21] and the Kamada-Kawai spring layout [22]. Similarly to *dada.distances*, the graph can be also navigated in a Markov-chain fashion, starting at a given point, and then choosing each following steps according to the edge probability distribution (weights) and to a desired memory length.

A variation on *dada.graph* is the *dada.machines* module (see Figure 12), essentially a graph where each node represents some "machine", i.e. a simple, prototypal operation to be performed on one or more inputs. By default these operations are elementary symbolic score transformations, such as transposition, retrogradation, circular shift, splitting, merging, and so on; user-defined operations are also supported. In a way, *dada.machines* represents a patch inside a patch, taking a score as input, processing it via the transformation graph, and outputting the result; however its spirit is more peculiar, and it was designed to be used with randomly generated graphs (the 'random' message produces graphs where the number of machines of each type matches a desired distribution). Via *dada.machines* one can apply a performative, exploratory paradigm to music, somehow reversing the functional and ergonomic relationship between algorithm and data.

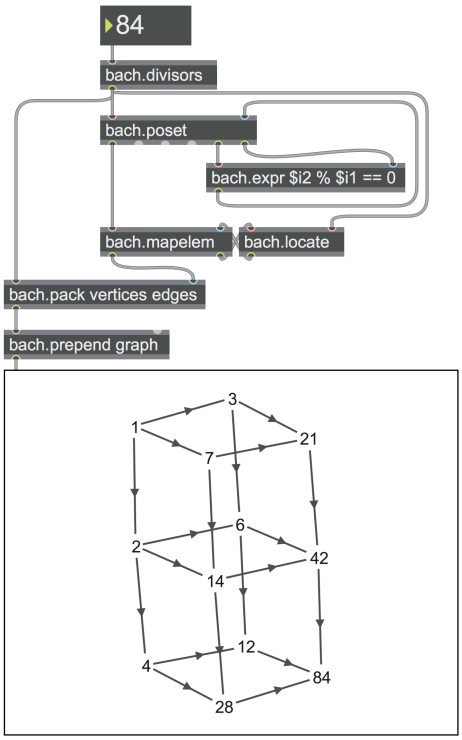

Figure 11. A simple patch displaying, via *dada.graph*, the lattice of divisors for an incoming natural number.

We are used to operate on data via carefully designed functions, and to modify them if the output result on a certain input is different from what we desire. As an example, to create a symbolic distorted granulation of a given Mozart sonata, one would spend quite some time designing the way the symbolic granulation should be achieved and the type of distorsion modelling needed. Nonetheless, one might reverse the principle, taking a random algorithm for granted, and carefully exploring input data in order to see if the results are interesting. If the algorithm is "complex enough", one might attempt to detect simple patterns (such as scales or counting-like patterns) along with more complex ones. (Of course, operatively, it makes little sense to search for a counting machine by tweaking inputs of a complex, random algorithm—which would categorize *dada.machines* module more as a mental experiment than a practical tool.)

5.4 Videogames

Developing a game engine in Max might seem awkward; and indeed there is a large number of environments specifically dedicated to the task (Unity probably being one of the most popular [9]). Max is neither designed nor optimized for such scenarios.

It can however be interesting to have a (crude, primitive) game engine natively coded in a Max external, since Max is a general purpose environment, and its visual paradigm can be applied to a large number of scenarios (digital audio, video, lighting, actuators...), making it easier to communicate between different media and techniques.

[9] `https://unity3d.com`

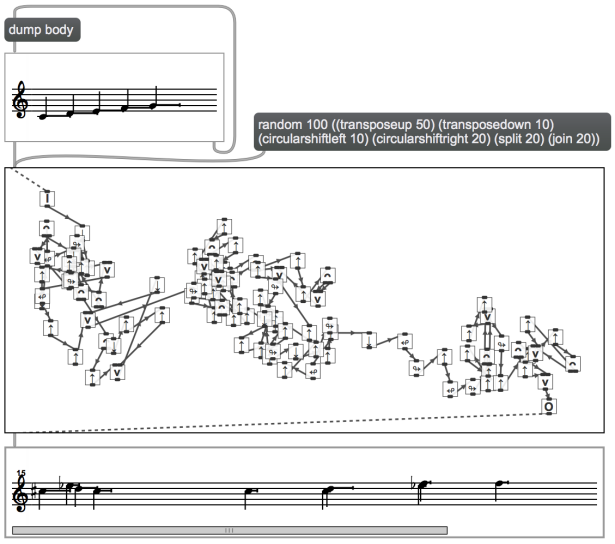

Figure 12. A patch featuring a *dada.machines* interface, generating network graphs containing 100 machines according to a distribution of some "atomic" score operations (transposition, circular shift, splitting and joining). The incoming score is processed via the randomly generated graph, and the result is output.

The *dada.platform* module, allowing the design of graphical interactions inspired by platform videogames, has been imagined and developed with these considerations in mind. Due to the complexity of designing a usable game engine, the module is currently in a prototypal phase, slightly more than a "proof of concept". Nevertheless *dada.platform* already supports four categories of objects:

Blocks: fixed objects which can possibly be broken;

Coins: fixed objects which can possibly be taken;

Game characters: moving elements which can interact with any other element in a more complex way. Game characters' motion is governed by a crude physical modelling: characters may possess the ability to jump, run, fly, swim, fire, glide, break elements, kill other characters, be killed by other characters. Game characters, in turns, belong to one of the following categories: 'usercontrol' (currently at most one character can be controlled by the user, also called 'hero'); 'idle' (do-nothing characters); 'food' (characters feeding the hero); 'enemy' (characters with the ability to harm or kill the hero); 'bullet' (projectiles potentially killing the hero);

Portals: objects which can dislocate the 'hero' to a new position in the same level, or to a brand new level.

All the properties of each object (such as its position, dimension, speed, abilities, image or sequence of images used to display it, and so on) can be set or fine-tuned via a dedicated inspector (see, for instance, Figure 13).

Linking game actions to musical events can be done in two ways. On one side, some of the objects' properties are musical scores (in *bach.roll* or *bach.score* syntax), output from a dedicated outlet whenever coins are taken, blocks are broke, and so on. More powerfully, any user action and any game interaction is notified via a dedicated outlet, so that any musical process can be triggered from them, such as sound synthesis, score production, video generation, and so on.

As it is not infrequent for objects in each level to share the same properties (just like identical blocks, coins or enemies), prototypes can be created, in order to easily handle multiple instances of indistinguishable objects.

Some of the properties of an object can be sequences of instructions, wrapped in levels of parentheses, written in a dedicated scripting language, designed to modify the configuration of the object itself, or of other objects. Instruction sequences are provided whenever a character dies, a block is hit, or a portal is entered, and so on. Script commands allows a wide range of actions, including: breaking blocks, assigning points or victory points, generating new objects, adding or removing abilities to characters, changing the state of objects, notifying some action, changing level or position in the level, pausing the game, preventing the hero from dying, winning, losing ("game over").

As a simple example, the script

```
(add hero ability fly)
(goto level mynewlevel.txt at PipeRev
with (keephero 1)),
```

assigned to a given portal, provides the current hero with the ability to fly, and then loads the level contained in the file `mynewlevel.txt`, at the position of the portal named `PipeRev`, keeping the current hero state (including its properties, points and victory points).

Each game character has a script sequence for its death (the "death sequence"); as another example, among many others, if one needs to turn a character named 'Juan', whenever he eats a certain fruit, into a character named 'SuperJuan', who, in turns, when killed returns to be a simple 'Juan' (like for the Mario/SuperMario classic Nintendo duality), one might want to assign to the fruit a death sequence along these lines:

```
(add hero ability break)
(change hero (name SuperJuan)
(idlesprite superjuanidle)
(walksprite superjuanwalk)
(jumpsprite superjuanjump)
(flysprite superjuanswim)
(height 1.625)
(ext 0.35 0.35 0.825 0.825)
(deathseq (dontdie) (remove hero ability
die during 2000) (change hero (name
Juan) (idlesprite juanidle) (walksprite
juanwalk) (jumpsprite juanjump) (height
1) (ext 0.4 0.4 0.5 0.5) (deathseq))
(remove hero ability break))).
```

Specific information about keywords and syntax can be found in the *dada.platform*'s help file and reference sheet. I shall just underline, in particular, how the last example is based on the fact that the fruit's death sequence changes the hero's death sequence, which in turns contain an instruction to clear its own death sequence, when triggered.

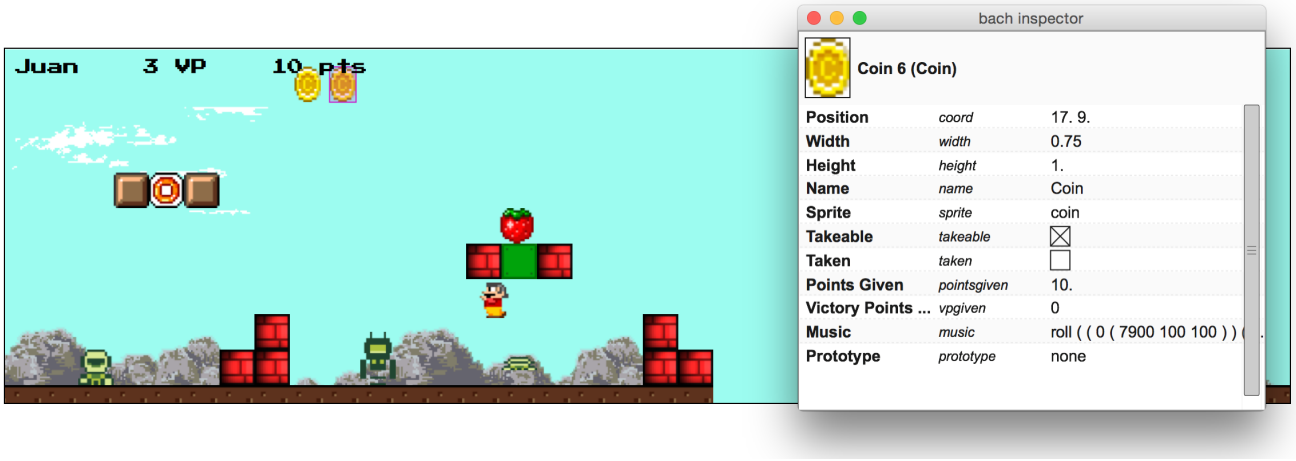

Figure 13. A screenshot of a *dada.platform* editor; the properties of the selected coin are displayed in the inspector.

6. COMPARISON WITH OTHER SOFTWARE

We have already emphasized the relationship of *dada* with project such CataRT, WAVE, Stuart James's objects, the Boids library, and Bill Vorn's Life Tools.

There is some correspondence between *dada*'s geometric approach and graphical sequencers such as Iannix [23] (as a matter of fact, a partial, two-dimensional porting of Iannix into *dada* might be a good addition to the library). On the other hand, the sequencing capabilities of Iannix largely outperform *dada*'s, whose purpose is not sequencing *per se*, but rather a seamless integration with the *bach* and Max environment, allowing, among many other things, live recording of scores.

The *dada* library shares with InScore [24] the interest in designing interactive non-standard symbolic representation. The idea of using games to interactively structure musical content resonates with Paul Turowski's researches and works, such as *Frontier* [18]. The *dada.life* module shares with Louis Bigo's HexaChord [25] the possibility of visualizing trajectories on musical lattices such as the Tonnetz—although the former focuses on the generation of cellular automata, while the latter is tailored for analysis purposes.

One should also remark the relationship of *dada* with music applications such as Björk's Biophilia, or Brian Eno's generative apps, or with interactive web tools such as some of the *Chrome Experiments* [10] or of the *A.I. Experiments* [11] (e.g., *The Infinite Drum Machine*); all these cases share with *dada* an interest for a tight, creative connection between visuals, gestures and music, and for exploring the grey area between interfaces and musical objects—however, if at least in Björk's case the musical apps are themselves art objects, *dada* modules are designed as simple instruments for composition [12].

[10] https://www.chromeexperiments.com/

[11] https://aiexperiments.withgoogle.com/drum-machine

[12] This has possibly one notable exception: the *dada.music*~ module, included in the library, organizing and representing on a segment *all* music tracks, might be considered both as a conceptual work and as a piece of evidence for an exploratory approach to music.

7. FUTURE WORK

The *dada* library is still in its infancy, and a certain number of additions and improvements are needed to complete it and to make it more usable.

First of all, thorough testing and optimization are necessary to make the library more stable and the user experience more comfortable. Besides, a Windows porting is also needed (currently the library only works on MacOS).

One of the most important lines of development would be porting the interfaces on mobile operative systems (tablets, smartphones), where they might take advantage of multitouch support. The most convenient way would be to exploit the Miraweb package [13], developed by Cycling '74, which allows mirroring on web browsers specific interface elements contained in a patch; the possibility to add Miraweb support to third party externals should be explored.

As far as the documentation is concerned, comprehensive help files and complete reference sheets are already provided for each module. However, some video tutorials would be a valuable addition for users who need to get used to the *dada* environment.

The set of tools for corpus-based composition can be improved in a number of ways.

- The number of analysis modules should be increased, by attempting to bring into the symbolic domain important audio descriptors such as roughness, inharmonicity, temporal centroid, and so on.
- *dada.catart* and *dada.distances* should be provided with the capability to modify column values by dragging points on the display.

The tools for physical or geometrical modelling are probably the modules in *dada* whose development is most advanced; nonetheless the *dada.terrain*~ module should be provided with anti-aliasing capabilities.

Finally, a certain number of improvements can affect the subset of tools dealing with rule-based systems and graphs:

- *dada.graph* is already capable of displaying graphs where the vertices are notes; it might also be pro-

[13] https://cycling74.com/articles/content-you-need-miraweb

vided with the possibility of displaying vertices as complex scores, which would open the way for potentially interesting applications.
- The *dada.graph* object should compute minimum spanning trees and shortest paths. It also should be provided with dedicated algorithms for special classes of graphs (such as trees or partially ordered sets). Automatic graph type detection, triggering the corresponding placement algorithm, might be a nice feature to have.
- The *dada.platform* object is currently little more than a "proof of concept". It would be interesting to issue something akin to a "call for scores" for pieces of interactive music based on it; this would probably also help detecting the bugs and the flaws of the system.

8. REFERENCES

[1] A. Agostini and D. Ghisi, "A Max Library for Musical Notation and Computer-Aided Composition," *Computer Music Journal*, vol. 39, no. 2, pp. 11–27, 2015.

[2] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[3] R. Cohn, "Introduction to neo-riemannian theory: a survey and a historical perspective," *Journal of Music Theory*, vol. 42, no. 2, pp. 167–180, 1998.

[4] D. Ghisi and A. Agostini, "Extending bach: A Family of Libraries for Real-time Computer-assisted Composition in Max," *Journal of New Music Research*, vol. 46, no. 1, pp. 34–53, 2017.

[5] A. Agostini, E. Daubresse, and D. Ghisi, "*cage*: a High-Level Library for Real-Time Computer-Aided Composition," in *Proceedings of the International Computer Music Conference*, Athens, Greece, 2014.

[6] D. Ghisi, "Music Across Music: Towards a Corpus-Based, Interactive Computer-Aided Composition," Ph.D. dissertation, Université Pierre et Marie Curie / Sorbonne Université, IRCAM, France, 2017.

[7] D. Schwarz, G. Beller, B. Verbrugghe, and S. Britton, "Real-Time Corpus-Based Concatenative Synthesis with CataRT," in *Proceedings of the International Conference on Digital Audio Effects*, Montreal, Canada, 2006.

[8] D. Schwartz, "Interacting with a Corpus of Sounds," in *Symposium on Sound and Interactivity*, Singapore, 2013.

[9] S. G. James, "Developing a flexible and expressive realtime polyphonic wave terrain synthesis instrument based on a visual and multidimensional methodology," Ph.D. dissertation, Edith Cowan University, Australia, 2005.

[10] D. Ghisi and C. Agon, "Real-time corpus-based concatenative synthesis for symbolic notation," in *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR'16)*, Cambridge, United Kingdom, 2016.

[11] A. Einbond, C. Trapani, A. Agostini, D. Ghisi, and D. Schwarz, "Fine-tuned Control of Concatenative Synthesis with CataRT Using the bach Library for Max," in *Proceedings of the International Computer Music Conference*, Athens, Greece, 2014.

[12] Q. Wang and K. L. Boyer, "Feature learning by multidimensional scaling and its applications in object recognition," in *SIBGRAPI Conference on Graphics, Patterns and Images*, Arequipa, Peru, 2013.

[13] D. Gay, *Geometry by Discovery*. Wiley, 1997.

[14] C. Roads, *The Computer Music Tutorial*. MIT press, 1996.

[15] D. R. Hofstadter, *Gödel, Escher, Bach*. Basic Books, 1999.

[16] E. Andersen, "Origami and math," http://www.paperfolding.com/math/, 2012, accessed: 2015-13-12.

[17] M. Macauley, "Braids and juggling patterns," Harvey Mudd College thesis, USA, 2003.

[18] P. Turowski, "Digital game as musical notation," Ph.D. dissertation, University of Virginia, USA, 2016.

[19] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 25–34, 1987.

[20] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *Art and Agency: An Anthropological Theory*. Addison-Wesley Professional, 2001.

[21] T. M. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[22] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989.

[23] T. Coduys and G. Ferry, "Iannix-aesthetical/symbolic visualisations for hypermedia composition," in *Proceedings of the Sound and Music Computing Conference*, Paris, France, 2004.

[24] Y. Fober, Y. Orlarey, and S. Letz, "An Environment for the Design of Live Music Scores," in *Proceedings of the Linux Audio Conference*, CCRMA/Stanford University, USA, 2012.

[25] L. Bigo, D. Ghisi, S. Antoine, and A. Moreno, "Representation of musical structures and processes in simplicial chord spaces," *Computer Music Journal*, vol. 39, no. 3, pp. 11–27, 2015.