THE IXIQUARKS: MERGING CODE AND GUI IN ONE CREATIVE SPACE

Thor Magnusson
ixi software
Creative Systems Lab
Department of Informatics
University of Sussex

ABSTRACT

This paper reports on *ixiQuarks*; an environment of instruments and effects that is built on top of the audio programming language SuperCollider. The rationale of these instruments is to explore alternative ways of designing musical interaction in screen-based software, and investigate how semiotics in interface design affects the musical output. The ixiQuarks are part of external libraries available to SuperCollider through the Quarks system. They are software instruments based on a non-realist design ideology that rejects the simulation of acoustic instruments or music hardware and focuses on experimentation at the level of musical interaction. In this environment we try to merge the graphical with the textual in the same instruments, allowing the user to reprogram and change parts of them in runtime. After a short introduction to SuperCollider and the Quark system, we will describe the ixiQuarks and the philosophical basis of their design. We conclude by looking at how they can be seen as epistemic tools that influence the musician in a complex hermeneutic circle of interpretation and signification.

1. INTRODUCTION

The question of affordances and constraints in musical software [8] is highly interesting as it is inevitably concerned with aesthetics and musicology. As all musicians who have worked with digital software know, the software itself suggests certain work methods, outlining a methodology to be followed. Often this means that the musicians have to change their natural ways of composing or playing and subscribe to work patterns that were defined by the designers of the software [10]. Musical software comes in various forms and includes both production and performance tools. The difference of such software compared to a word editor, a browser or a mail program is that the interaction design of the software itself has much stronger aesthetic implications for the user. Software is never neutral in its expressive scope and the more refined it is, the more it constrains.

In order to escape from the expressive constraints of commercial or closed source software, musicians are increasingly making use of free and open source programming environments such as SuperCollider, ChucK, Pure Data and other similar patching environments that allow for either textual or graphical programming. The word "free" above connotes not only free as in "free speech", or "free beer", but also free as in "free jazz". The freedom of musical expression when utilising these tools is characteristic of their nature and important criteria their designers had in mind when designing the environments [11, 16]. Of course, one could argue that these audio programming languages are constraining as well, but this is more on the level of software design and rarely on the level of musical interaction. As such, the musician or instrument designer always has ways of getting around software engineering limitations to reach the goal of the composition or instrument design.

In this paper, I will introduce the ixiQuarks: a graphical user interface (GUI) environment of audio tools and instruments for live improvisation that allow for user interaction on both the GUI and the code level. The ixiQuarks are written in the SuperCollider programming language and are part of the Quarks repository (external libraries for the SuperCollider language). I will commence by outlining the general SuperCollider/Quarks environment, then introduce the ixiQuarks instruments and finally talk about the philosophical and aesthetic implications of this unified creative space where code and interface can merge in a continuous performance activity.

2. SUPERCOLLIDER, QUARKS AND CODING

2.1. The SuperCollider Environment

SuperCollider 3 (or SC Server) [11] is the state-of-the-art audio programming environment of today, written by James McCartney and released as open source software in 2002. Since then it has become ever more sophisticated and powerful, used by musicians, artists and scientists alike that form a strong developer and user community. SuperCollider is split up in two independent parts, the SC language and the SC server. The former is an interpreted, object orientated language written in C/C++ that takes inspiration from the design of SmallTalk; the latter is an audio server that supports a powerful C plugin architecture which makes audio digital signal processing effective, fast and easy to program. The language and the server communicate through the Open Sound Control (OSC) protocol [17], which makes it possible for all OSC supporting programming languages to talk to the SC server. This split between a language and a server, and the usage of OSC makes SuperCollider an ideal audio programming environment for networked performances, live-coding, interface creation (hardware or software) and collaborative playing. It also means that the language can perform complex real-time calculations without

resulting in glitches in the audio as they are two separate processes.

The SuperCollider language and server are open source and anyone can write class extensions for the language or plugins (unit generators) for the server. The environment is compiled into a binary file (the distributed application) with a C compiler, but then runs as an interpreted language that has source class files that interface with the C primitives. The SuperCollider class files are written in the SC-language and are an important way to modularize code and compositional concepts when working with the language. SuperCollider itself is an extremely broad and flexible language and easily extendable by writing one's own classes.

2.2. Quarks

Authors of 3rd party SuperCollider classes tend to share them with the rest of the community if they have a general scope and are useful in other than a private capacity. In order to extend SuperCollider with new classes, users can either install them manually into the appropriate ClassLib folder or use the dynamic Quarks system. The Quarks system was introduced for the creators of 3rd party class files to simplify the process of creating, updating and distributing their code but also simplifying the updating channel for the users of the classes. This is achieved by using a SVN[1] (Subversion Control) repository system where the author commits the latest changes in his/her classes and then the users can update their classes with a simple two line command:

```
Quarks.checkout( "ixiQuarks" );    // downloads from svn
Quarks.install( "ixiQuarks" );     // installs in sc-classpath
```

The repository is online on the SourceForge website and the Quarks class in SuperCollider takes care of downloading and installing the chosen classes "under the hood" so to speak. The user only needs the two lines above to download new class libraries. The SVN system makes it easy for the author to track changes in the code, but also for the user to follow development of the class.

2.3. Programming and Live Coding with SC

One of the frustrations for the computer musician who performs in a live situation with acoustic instrumentalists, for example in an improvisation band, is the difficulty of carrying out spontaneous and intuitive change in playing [10]. Musical software is often more focused on the score or the textural level of a musical performance rather than on the note level. This fact tends to make software instruments less expressive than their acoustic counterparts. But there are various ways to get around the rigidity of musical software. One solution is the field represented by the new musical interface research[2] but another approach is live-coding where the instrument/music is created and modified as a performance act.

SuperCollider has extensive support for creating interfaces on the graphical user interface level, using MIDI, HID (Human Interface Devices), serial or OSC (Open Sound Control) communication. As such it lends itself well to all common interface work, instrument making and installations. But SuperCollider is also one of the most powerful environments for live-coding musical performances [12, 13] as it is an interpreted language and new code can be evaluated in run-time without saving or recompiling anything. Programs can be created that evolve or change according to user input or additional programming. As an example, here is a small JITLib [13] program that generates a simple snare sound every second:

```
~trig = { Impulse.ar(1) }; // the trigger
~snare = { WhiteNoise.ar(1) * EnvGen.ar( Env.perc, ~trig.ar )};
```

Say we wanted to add a low pass filter to the sound without interrupting its continuity, we simply run the latter line again where the white noise goes through the filter:

```
~snare={LPF.ar(WhiteNoise.ar * EnvGen.ar(Env.perc,~trig.ar), 2000)};
```

There are countless ways of doing these things in SuperCollider as it is a wide and powerful programming language whose users have different agenda, emphasis and programming styles. Some people prefer writing classes and/or graphical user interfaces to be controlled by sensors or controllers. Others work purely in code composing algorithmic music and yet others enjoy the tense experience of coding in front of the audience in a live situation.[3] In contrast to much closed source software, there are as many ways of using SuperCollider as there are people working with it. There is no rigid methodology as all programmers/musicians have their own way of thinking; their own style of writing code/music.

3. THE IXI QUARKS

The *ixi software*[4] project started in 2000 as an exploration of how structures of interaction in musical software could be redefined. The aim was to resist the imitation of physical hardware or acoustic instruments in the way the interaction and interface design was implemented. The ixi instruments are designed from the affordances and premises of the computer itself and not those of physical reality. As music is in essence the execution of sonic patterns through time, we concentrated on creating pattern-generating interfaces with strong graphical elements implemented in the interaction design. These interfaces were outputting OSC information to sound engines that were written in SuperCollider, Pure Data or Max/MSP, but some also included closed sound engines [9].

Recently the interface support of SuperCollider has matured to the level that interfaces in the style of ixi software can easily be built with the SuperCollider

[1] http://sourceforge.net/docs/E09
[2] http://www.nime.org
[3] See the TOPLAP manifesto - http://www.toplap.org/
[4] http://www.ixi-audio.net

language itself.[5] As SuperCollider is the programming language of choice for the current author, it became more natural to write the interfaces in SuperCollider itself, rather than in Python or Java as we had been doing before. The interfaces are still OSC controllers that can be used with other sound engines as well but they are streamlined for use with the SC audio server.

3.1. The ixiQuarks Environment

SuperCollider is an open and dynamic environment that allows for running many programs simultaneously, using any number of groups, nodes and audio busses. Any process can be started, paused, stopped or freed without interfering with other processes that are also running in the environment. The ixiQuarks toolbox is a collection of tools that perform various tasks that could be time-consuming to code up in a live/improv situation, but easily accessed from a GUI window that contains a list of all the ixiQuarks. The ixiQuarks don't need to be used exclusively as an independent environment, but can be used with any other program written in SuperCollider. As an example one could imagine a performer that is running some process using the Pattern classes, suddenly deciding to add reverb to the output. In this case it is trivial to open a reverb ixiQuark and route the audio from the original process through the reverb program that has a simple GUI to control the basic parameters.

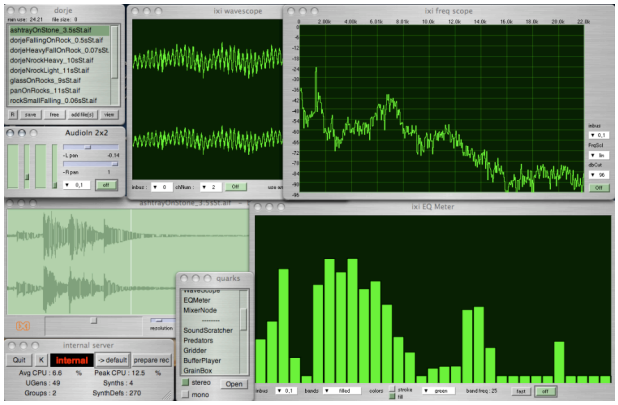

Figure 1. Screenshot of some ixiQuarks utilities

The ixiQuarks is a modular environment that consists of three different types of tools: *basic utilities*, *audio effects* and *instruments*. The environment is built around audio busses that can be used to patch audio streams into one another. An audio bus can contain the output of many sound-generating processes. Here below, I will explain briefly the first two types and then focus on the main research-topic of this paper, the instruments.

3.2. Basic Utilities

The basic utilities are tools such as AudioIn, Recorders (of any audio channel), BufferPools (that stores sound buffers in RAM), Players (streaming soundfiles from the hard disk), NodeMixers, and various scopes for viewing the audio (such as an EQMeter, FreqScope, an adaption of Lance Putnam's FreqScope, WaveScope, etc.) These are general utilities that are needed to set up an environment very quickly. The design idea is to let the instruments make use of these utilities rather than integrating them into the instruments themselves. That would create unnecessary complexity and be against the modular design philosophy of the environment.

[5] What was needed was a class that detected mouse movements, drawing functionality and hardware interfacing.

3.3. Audio Effects

The audio effects are the typical effects known from most sound editors: delay, reverb, distortion, compression, chorus, flanger, tremolo, equalizer, vocoder, randompanner, and some strange effects such as MrRoque (which is an effect that records incoming sound with reverb and succinctly plays it backwards, also through a reverb). The effects run on the audio channels and can be turned on and off as one wishes. The user can plug many effects onto the same channel or route the sound through one effect into the next on another channel.

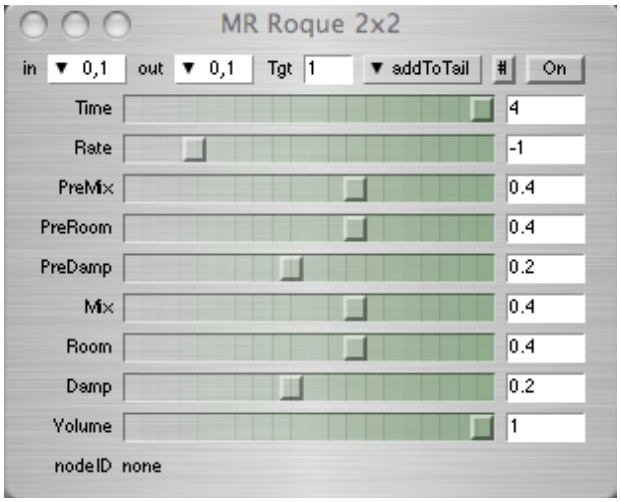

Figure 2. Screenshot of a typical ixiQuarks effect

3.4. Instruments

The ixiQuarks environment is designed for the building of instruments that make use of other ixiQuarks such as the utility tools or the audio effects. At the point of writing there are 9 different instruments available with more in the pipeline, and users can easily write their own instruments that work seamlessly in the environment. In general these instruments are pattern-generators that allow for sample manipulation, synthesis and live coding. Below I explain five of them.

3.4.1. The SoundScratcher

The SoundScratcher is an instrument that represents the waveform of a sample buffer in a graphical display. Any buffer stored in the RAM memory can be manipulated by the instrument. The instrument receives input from hardware such as the mouse or a Wacom tablet and the basic idea is that the user can draw on top of the

waveform representation and control the playback of the sound that way.

There are various interactive modes in which the user can play with the sound: *warp* – a granular synthesis where the location in the sound file is played constantly using overlapping soundgrains. Here the vertical location of the cursor represents the pitch; *scratch* – by moving the pen (or some other sensor input) the user can scratch the buffer forwards and backwards like a needle on a turntable. In the spirit of turntables, the speed of the gestural movement maps naturally to the pitch; *random grains* – here grains are represented as dots on the canvas. The sound engine reads randomly though an array of the locations of those dots, creating a granular cloud. The density of the cloud, the envelope length and envelope type of each grain can be controlled from the interface; *linear grains* – same functionality as in random grains, but here the sound engine reads linearly through the list of grains. However, there is a randomising button on the left of the interface where the users can randomise the list if they so wish; *worm* – the worm is a creature that moves over the space of the sound with variable numbers of grains in its spine. The speed of the worm and the grain duration can be controlled; *graincircles* – these are circles of variable size and amplitude (represented as alpha in colour) inside which the sound engine spawns grains according to the set speed. Again the envelope and duration of the grain can be controlled; *grainsquares* – as opposed to the graincircles the squares always play the grains from the left point of its location. This makes it easier to create interesting rhythms and periodic sound textures than in the random space of the circles. The speed of the grain repetition can be affected in both the graincircles and the grainsquares by pressing the 1, 2, 3, or 4 number keys, representing the respective time relationships (3 against 2 or 4 against 3, etc.)

Figure 3. Screenshot of the SoundScratcher instrument

The gestural movements of the user can be recorded, stored and played back by the instrument itself. That way the user can draw patterns on the instrument and leave it to perform on its own. The user could then for example open up another instance of Soundscratcher to perform with.

3.4.2. The Gridder

The Gridder is an instrument that focuses on micro-tonality in various ways. It consists of a scaleable grid of nodes (from 5 to 48 squared) which is mapped in equal temperament (implementing the formula *fundamental * 2.pow(i/steps_per_octave))* in octaves (one octave per horizontal line) but wrapping at a set ceiling frequency.

Like SoundScratcher, the Gridder is a broad conceptual environment on its own that affords various interaction modalities. We could divide the analysis of the instrument into two parts: the *environmental part* – where we look at the acoustic properties of the instrument; and the *interaction part* – where we focus on how the instrument can be controlled.

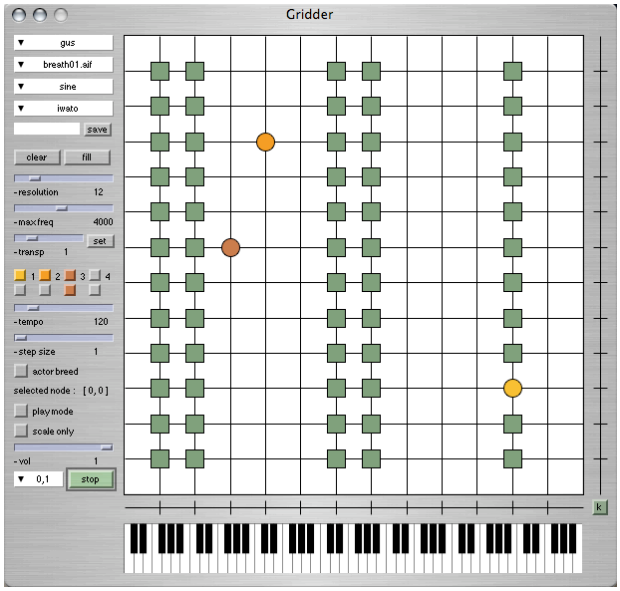

Figure 4. Screenshot of the Gridder in 12-tET tuning

Looking at the *environmental* properties of the instrument, the most important part is the definition of the pitch resolution in an octave. The grid can be scaled from a 5-tET scale to 48-tET scale. (5-tET = 5 tone equal tempered tuning). Each node on the grid is a note in the scale. The horizontal axis represents the N notes in the octave and the vertical axis is the next pitch range, by default an octave above, but it could be the fundamental scale transposed by a 3rd, a 5th or any other relation. There is a setting that controls maximum frequency on the vertical axis forcing the scale to wrap back to the fundamental note when the ceiling is reached. The user can define scales in any of the pitch resolutions by drawing vertical lines as represented by the columns of boxes in Figure 4 and these can be stored in a dictionary. There are 8 different types of synthesized sound that can be selected, and the user can also write his/her own synthesis using code in a special coding window. The nodes can also contain sound samples that are triggered when the node is activated. The piano keyboard view is optional and not graphically connected to the grid itself in any way. It shows the notes played, indicated with grey if the pitch maps to the western 12-tone scale but red if it is a microtone.

The *interactive* attributes (related to playing) of the instrument consist of a space where one can play scales or notes with the mouse, the Wacom pen or any other interface. There are two types of playing: free playing, where the player can play any note on the grid;

and restricted playing, where only the selected notes can be played. That way the instrument can be played as a customly tuned string instrument. Related projects are Spiegel and Hunt [5, 14]. The interface also contains 4 agents that can move through the grid-space, each with its own tempo and step size properties. If the agent lands on a selected node, it triggers the assigned action (playing a synthesised note or triggering a sample). The agents also have a mode where they breed notes (by selecting it) if they mate.

3.4.3. *The Predators*

The Predators is an artificial life (ALife) instrument where predators and preys are left to interact in a neutral environment. The idea behind the instrument was to create a non-linear note or sample player; one that was controlled by the properties of an ALife system rather than random number generators.

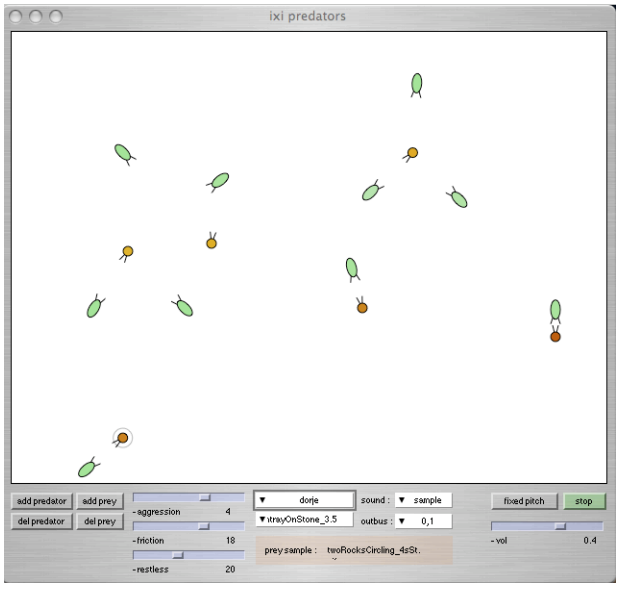

Figure 5. Screenshot of the Predators instrument

The predators have properties like energy, speed, restlessness, friction and focus on a prey. They loose energy by not eating from the preys or by fighting about them. When a predator takes a bite of the prey, it emits a sound. The sound is either synthesized sound or samples from the buffer pool. The pitch of the sound is defined either by the vertical location of the prey or by assigning a special pitch to the prey by choosing a note from a keyboard that pops up. Of course, preys and predators can be added to or removed from the environment. The ALife properties of this instrument are rather simple. We could have added things like environmental obstacles, age, death, reproduction, etc. but we did not find that it would serve any musical purpose in this simple instrument.

3.4.4. *The PolyMachine*

The PolyMachine is a polyrhythmic pattern sequencer that implements four different TempoClocks controlling each channel. The instrument originates from a study of Indian *talas* but turned into a parody of the typical drum sequencer.

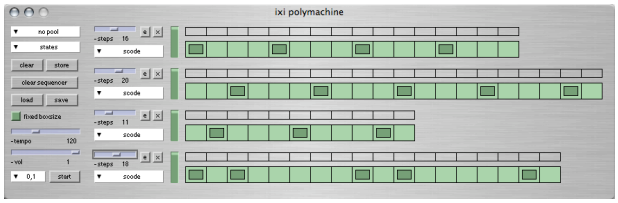

Figure 6. Screenshot of the PolyMachine instrument

The PolyMachine consists of 4 tracks where the number of steps in each track can be defined individually. One can view the instrument either with a fixed box-size where the GUI window grows in size (Figure 6) or a relative box-size where the tracks adjust to the size of the window (Figure 7). Both modes have different qualities to them and it can be interesting to study the conceptual understanding of time and rhythm in the different representational modes.

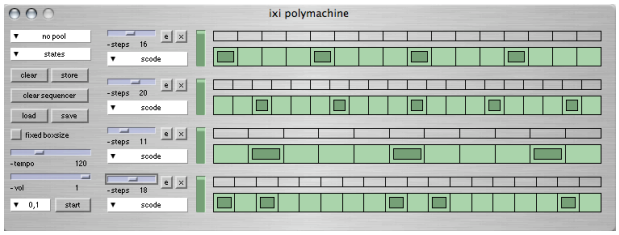

Figure 7. Same as Figure 6 but different view mode

Each track has a time indicator that travels above the sequence line and triggers an event if the box is selected. The event can be triggering of a sound sample or any function that SuperCollider can evaluate, such as sound synthesis or sending OSC or MIDI messages to other environments or applications. The tracks have volume control and an envelope generator.

3.4.5. *The GrainBox*

The GrainBox is a two dimensional parameter space for granular synthesis. The problem with granular synthesis is often how to represent it graphically at the interface level as there are so many parameters involved. Here we represent the parameters with coloured boxes in a two dimensional space where boxes with related parameters are connected with lines. This makes it easy for the musician to intuitively understand the state of the sound engine by quickly glancing at the interface, as opposed to the complex analysis of slider positions where one has to read the label of each slider.

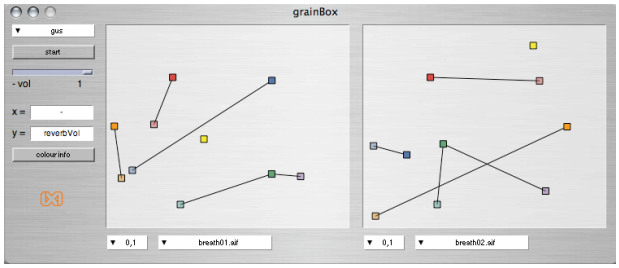

Figure 8. Screenshot of the GrainBox instrument

The audio stream of the GrainBox can be output on any audio bus and used as the source sound fed into and controlled by other instruments such as PolyMachine, Gridder or Predators. This way, the sound can be directly adjusted from the GrainBox application but utilised by the other pattern generating instruments that wrap the sound in an envelope. Of course, the GrainBox can also be used independently as sound texture generator.

4. EPISTEMIC TOOLS

Something in the world forces us to think. This something is an object not of recognition but of a fundamental encounter. [3]

4.1. The Fundamental Encounter

A fundamental encounter with an object is not something that reinforces our identity or habits. Quite the opposite, it ruptures the stabilised habits of the self. From this perspective, it can be useful to ask: what is the process of encountering an instrument or a tool for the first time? Does the instrument change the musical ideas of the user or reaffirm them? The first encounter with a music software and the process getting an understanding of it is essentially a hermeneutic process. It involves at least two mental models of musical theory – that of the designer and that of the user – where it is the user's task to mould his/her model to the functionality of the software. As the software itself is an artefact on a much more complex and conceptual scale than an acoustic instrument, the process concerns a circular (as in Gadamer's hermeneutic circle [4]) interpretation of the meaning of the system. This meaning can be modified (or "hacked") and appropriated to one's own work methods and musical aesthetics, but in general we could say that the software defines the musician through its interface.

Now, how do the ixiQuarks instruments relate to the mental models the user has about how musical software should look and work? They might not relate in any way to such models. These instruments are eccentric, limited and focused on certain tasks, unlike much music software that tries unsuccessfully to attain generality in design. We are interested in constrained tools that focus on certain tasks, are easy to learn, but still provide scope for in-depth study and mastery. The ixiQuarks do hardly relate to any acoustic musical instrument or hardware, but their design does not arise from a vacuum. The design metaphors and interaction models are inspired by computer games and multimedia design on the one hand, but physical actions (such as scratching or drawing) on the other. The user does not necessarily find the software alien to his/her thought, as it connects into prior experiences with digital technologies or physical actions.

In musical software, there exists a continuum from a narrow scope (where the software is a clear personal expression by its author - perhaps allowing for some user interaction) to a wide scope where the tool has few restrictions and allows its users to express themselves more freely (see Figure 9). A rule of thumb is that the wider the environment is, the more the user has to invest time studying it and the longer it takes to design a composition or an instrument for their own needs.

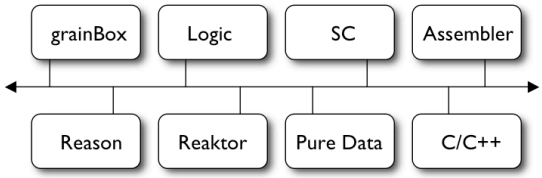

Figure 9. A non-scientific representation of the expressive constraints of musical software. To the right we have more freedom, requiring more investment.

The ixiQuarks are limited and constraining instruments with focused functionality. They are intended for live, improvisational performance and to be played like acoustic instruments as part of an ensemble. The main criteria for their design is to enable the musician to respond quickly to structural changes in the music, to change directions and to be expressive by bodily gestures by extension via the supported joystick, tablet, mouse control, etc. However, the focus is not on the gestural part of the tools but on the cognitive or musical implications that a computational system for music always incorporates through its design: i.e. the epistemic nature of the tool.

4.2. Epistemic Actions

Andy Clark [2] has put forth a theory of external cognition, where he shows how our mind uses tools external to the body as part of the cognitive process. He uses the phrase "external scaffolding" to explain how we for example move Scrabble tiles around in their tray to form new words, rather than memorising the letters and representing the words internally in the mind. Another example is how we rotate the bricks in Tetris to *see* how they fit rather than thinking it through. In a 1994 paper, Kisch and Maglio [7] show how actions can be divided into epistemic actions and pragmatic actions. Epistemic actions are physical actions that make mental cognition faster and easier. In order to facilitate optimal cognition, intelligent agents adapt their environment to get the most out of their limited cognitive resources. The human being has developed its cognitive skills in a symbiotic relationship with its technology and continues to do so [1]. Epistemic tools constrain, direct and enable certain cognitive tasks on the individual level and on a more historical level we see how cultures produce, adapt and are effected by the technologies of epistemic action.

Analogously to Tetris, musical software serves as external scaffolding for the composer. The interface both affords musical potential and stores musical parameters. It becomes the "locus" of the composition or the musical performance; an extension of the musician's mind. The music theory is carved into the functionality of the software's interface. The interface of a digital instrument becomes a cultural territory; a space where

musical ideas can originate and take shape. It is a result of our musical culture and technology, but at the same time it actively influences the contemporary musical soundscape around us. This is not to deny that the acoustic instrument has specific affordances and lends itself to certain musical styles and playing, but more in order to point out how musical software normally resembles the score more than the instrument used to play the score and thus affects the musician on a more formal level.

4.3. Software Tools as an Extension to the Mind

The digital interface/instrument and the musician can be seen as one coupled system; one cognitive feedback loop. The musician offloads some of his/her cognitive activities onto the interface that in turn affects and influences their thinking with its interactive affordances.

In her work on semiotic engineering, De Souza [15] defines "intellectual artefacts" as tools that encode a particular understanding or interpretation of a problem situation, but also a set of solutions to that problem situation.[6] This encoding is fundamentally semiotic. Both the user and the designer have to share understanding in the same semiotic system. This shared understanding can only be achieved through a hermeneutic process where the conceptual or mental model of the user adapts to the model of the designer. In this process the user will inevitably change his/her musical ideas according to the affordances of the instrument itself. We have labelled this dynamic elsewhere [9] as a "dual semiotic stance" with reference to Jakobson's model of communication [6]. Figure 10 is a representation of Jakobson's model, where on the left we have a representation of the instrument as a semiotic system, but on the right we view the music itself as the meaning conveying medium.

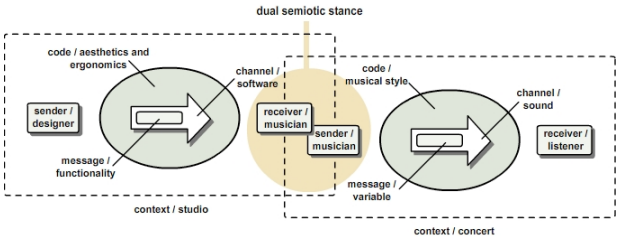

Figure 10. A representation of the dual semiotic stance

Semiotic theory explains all systems essentially as sign systems that convey specific meanings. The instrument maker should be aware of his or her position in affecting the musical ideas of the users (and to see how western synthesizers, hardware and software is musically limited for some other cultures. Just consider Indian and Arabic musicians' use of the pitch-wheel when playing synthesizers) and be honest about the limitations of the instruments. As De Souza points out, there is an obligation for the designers to communicate properly the mental model that serves as the foundation for the design decisions that took place in creating the particular software.

4.4. The ixiQuarks as Epistemic Tools

The instruments of the ixiQuarks software suite contain active, affecting, adaptive and automatic elements of various degrees. The musician "offloads" [1] some of his/her cognitive functions to the instrument itself that continues playing or influencing the music in its own way. The tool either represents a mechanism too convoluted to keep in mind at any one time or activities too complex to perform with a bodily performance. The ixiQuark instruments are limited to a specific design idea but the user is still provided with the power to extend the functionality of the instruments by coding extensions to it or creating new synthesis algorithms for it to use.

A recent survey we conducted [10] has shown that people find the limitations of acoustic instruments fascinating and enjoyable, leading to mastery of the tool and affecting the relationship to it. On the other hand, the survey also showed that people often found that the problems of acoustic instruments are that they were hard to change and playing them was a cliché-prone activity. In a special part of the survey we stepped out of its general focus and asked questions about ixi software in particular. From those who answered that section, we learned that the concerns were the same, i.e. that people found joy in the limitations of the instruments and in exploring their scope and expressive depth. However, many reported that after a while they missed the option of being able to extend the instrument and get it to work with other software in a more complex setup. We believe that with the ixiQuarks, and the merging of code and GUI in one epistemic tool, we have addressed these questions in one possible way; a way that has served ourselves well in our live improvisations with acoustic instrumentalists where quick response, fast changes and liveliness are common traits of the performance.

5. CONCLUSION

An interface is essentially an abstraction. It is a higher level representation of a structure of further complexity. As such, all efforts to build tools with interfaces are a process of limiting the scope of expression. We can distinguish two types of interfaces in musical performance with digital instruments: physical and virtual interfaces. Here we have focused on the virtual interface as the conceptual engine of the instrument, the location where parameters of the epistemic tool are set and controlled. This applies in all situations, whether the instrument is controlled by complex gestural sensors or the typical mouse and keyboard interface.

The concept of the interface as limitation does apply to both graphical user interfaces and programming languages. The classes of a programming language are essentially interfaces for potential functionality. It is therefore already a limitation, but from a musical perspective (as opposed to software engineering), it is probably the least limited and constrained environment

[6] Ironically, the "problem situation" we are talking about is that of creating music.

available today. We have seen here how an interface is a conceptualization of an action of epistemic nature; a semiotic design that invariably defines the possibilities in thinking and performing whilst using the tool.

The ixiQuarks presented here are constrained ideological instruments, designed for specific expression. The aim is not that of musical generality, but rather a focus on particular interactive patterns. However, this happens in a context where anything can be added or built as satellites to the instruments either as part of the ixiQuarks or simply by live-coding it in real-time in a performance.

6. ACKNOWLEDGMENTS

The ixiQuarks have been written in the beautiful programming environment SuperCollider. I want to acknowledge the hard work of its developers and users who make up a very helpful user community. I would also like to thank my colleagues in ixi software for constant inspiration, especially Enrike Hurtado Mendieta. Tom Hall, Nick Collins and Chris Thornton also provided invaluable feedback and discussions on the ideas in this paper.

7. REFERENCES

[1] Clark, Andy. *Natural-Born Cyborgs: Minds, Technologies, and the Future of Human Intelligence*. Oxford: Oxford University Press, 2003.

[2] Clark, Andy & Chalmers, David. "The Extended Mind" *Analysis* 58: 1: 1998.

[3] Deleuze, Gilles. *Difference and Repetition*. New York: Columbia University Press, 1994. p. 139.

[4] Gadamer, Hans-Georg. *Truth and Method*. New York: Crossroad, 1989.

[5] Hunt, Andy & Kirk, Ross. "MidiGrid: Past, Present and Future". *Proceedings of NIME 2006*, Montreal: McGill University.

[6] Jacobson, Roman. "Closing statement: linguistics and poetics" in *Style in Language* (ed.) Sebeok. Cambridge: MIT Press, 1960.

[7] Kirsh, D. & Maglio, P. 1994. "On distinguishing epistemic from pragmatic action". *Cognitive Science*. 18:513-49.

[8] Magnusson, Thor. "Affordances and Constraints in Screen-Based Musical Instruments" in *Proceedings of the NordiCHI Conference*, Oslo: Oslo University, 2006.

[9] Magnusson, Thor. "Screen-Based Musical Instruments as Semiotic-Machines" in *Proceedings of NIME 2006*. Paris: IRCAM, 2006.

[10] Magnusson, Thor & Hurtado Mendieta, Enrike. "The Acoustic, the Digital and the Body: A Survey on Musical Instruments", *Proceedings of the NIME Conference*, New York, USA, 2007.

[11] McCartney, James. "Rethinking the Computer Music Language: SuperCollider" in *Computer Music Journal,* 26:4, pp. 61-68, Winter 2002. MIT Press 2002.

[12] Nilson, Click. "Live Coding Practice" in *Proceedings of NIME 2007*. New York: New York University, 2007.

[13] Rohrhuber, Julian; Campo, Alberto de & Wieser, R. "Algorithms today - notes on language design for just in time programming". in *Proceedings of International Computer Music Conference. ICMC*. Barcelona: Escola Superior de Música de Catalunya, 2005.

[14] Spiegel, Lauri. "Music Mouse™ - An Intelligent Instrument" Program and Manual: http://tamw.atari-users.net/Atari_Music_Mouse_Manual.pdf, 1993.

[15] Souza, Clarisse de. *The Semiotic Engineering of Human-Computer Interaction*. Cambridge: MIT Press, 2005. p. 10.

[16] Puckette, Miller. "Using PD as Score Language" in *Proceedings of the ICMC Conference 2002*. Pp. 184-187.

[17] Wright, Matt. "OpenSound Control: State of the Art 2003." in *Proceedings of the NIME Conference*. Montreal: McGill. 2003.