In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 17 Pages 106 - 119

# Attuning: A Social and Technical Study of Artist–Programmer Collaborations

Greg Turner, Alastair Weakley, Yun Zhang, Ernest Edmonds

Creativity and Cognition Studios, Faculty of IT,
University of Technology, Sydney,
PO Box 123, Broadway NSW, Australia
greg@gregturner.org, alastair@weakley.org.uk,
yunzhang@it.uts.edu.au, ernest@ernestedmonds.com
http://www.creativityandcognition.com

**Abstract.** This paper presents findings from a grounded theory study of the social and technical roles of programmers in art-technology collaborations. Combined with a review of the roles of technology with respect to helping artists engage with the computing medium, we show that programmers can play several roles in such collaborations, both supportive of and obstructive to the requirements of artists, beyond merely 'doing the programming'. Of central importance is the process of 'attuning' between the actors and artefacts involved, and this can show us ways of making programming systems more comprehensible to artists.

## 1. Background

### 1.1 How We Think About Interactive Art

From a technological perspective, computer-mediated interactive art involves a computer system with some form of input from the audience-participant, some form of output to the audience-participant, and, most interestingly from a programming perspective, some form of processing in between. However, interactive art is in practice quite a complex field, involving various creators, producers and audiences, not just a set of computational artefacts with an 'optimal' configuration. When we think about any of the entities situated within these art systems [5], we should not lose sight of that very situatedness.

### 1.2 How We Think About Programming

The authors adopt the view that any reasonable definition of programming (for example, that programming is a specification of a computation [3], or that programming is taking actions with the objective of creating an application that serves some function for the user [22]) can describe all uses of a computer. There is no particular ontologi-

cal distinction between programming a computer and using it. This is not to say that all languages are the same – we do not ignore the obvious technological and cognitive differences between, for example, the language of Direct Manipulation [26] and Assembly Language. These differences to suggest a continuum between two poles, which we call 'popular programming' and 'deep programming'. For the sake of simplicity, the term 'programming' will from now on be used to refer to the *entire* continuum of styles between popular and deep. So, by 'ability to program', we mean 'ability to deep program'.

### 1.3 Why is Programming Important to Art?

Computers allow us to think thoughts and to experience things that would be impossible without them, particularly in aid of entertainment and scientific discoveries, but what is it about the tool that allows new thought? An examination of the important developments in computing (and particularly programming) history indicates following four technological strengths, which we call the four Ss: *speed*, *slavery*, *synaesthesia* and *structure*. Other significant developments (such as the Internet and digital artworks), exploit one or more of these, and programming tools which provide access to these strengths are presumably to be desired.

Treated briefly here, with particular reference to applications in interactive art, 'Speed' refers to the computer's ability to do certain things quicker than we can. We use computers to think thoughts that would otherwise require too much time. In interactive art, the goal is often to generate the response 'in real time' (<50ms for immersion, <10ms for musical performance).

'Slavery' is descriptive of both the incredible cheapness and unquestioning obedience of computation, and it is what allows artists and programmers to create massive and wide-ranging programmatic edifices. However, this pedantic obedience carries with it the danger of genie-in-the-bottle or sorcerer's apprentice-style mishaps.

'Synaesthesia' is another way of describing Negroponte's 'formless bits' [23] – that all a computer does is perform operations on collections of 1s and 0s. Inside a computer, video is the same stuff as audio is the same stuff as text is the same stuff as time, which means it is possible, for example, to combine them and convert between them. Synaesthesia is itself strongly exploited in interactive art, probably as a consequence of this quality.

'Structure', that mysterious descriptive power of computing, is the hardest quality to nail down, perhaps because it has no direct analogue in the physical world. Abstracting situations into structures is the current distinctive speciality of programmers and systems analysts, and may be an important reason why artists employ them. Understanding the potential of abstract structure is the key to fully engaging with the computing medium. As a result there are people who advocate that everyone should learn to program (e.g. Kay and Goldberg [18], Flanagan and Perlin [9]), and people who advocate that artists who make computer art must learn to program (e.g. Maeda [20], though he has since revised this opinion [21]). However, since many artists make good computer art without deep programming (by getting a programmer to do it for them), we have to assume that programming is not always necessary. Although many artists do indeed find the requisite level of engagement by learning the neces-

sary technical skills from scratch, many others find it vicariously, by delegating the work to a programmer.

## 2. Technology Support for Creativity

### 2.1 Expressivity Support

An important issue at stake is language expressivity (expressivity presumably being valued in tools for artists). Metaphorically speaking, popular programming languages allow us to make "big brush strokes" with benefits in terms of effort and limitations in terms of fine control and extensibility. Deeper programming languages, on the other hand, are equivalent to not just making "smaller brush strokes", but even to making new paint and brushes, and this means that it is difficult to construct large systems from small articulations. Popular programming languages each impose an aesthetic which is so easily linked with a corresponding artistic subgenre that it becomes increasingly hard to avoid cliché, particularly on an interactive level: Flash and Director for net art and CD-ROMs; Max/MSP for performance-based electronic sound art. Deeper programming languages suffer less from this problem, but at the expense of ease of learning and obviousness of use. Neither popular nor deep languages are very expressive – a truly expressive language would allow both large and small computing granularity, both obvious gestures and subtle nuances.

One way of achieving this expressivity is to have the entire computer system built around a *homogeneous, universal,* concept that can be used at both microscopic and macroscopic levels of systems, and everywhere in between, a bit like glass lenses in microscopes and telescopes. A homogeneous universal computing concept means that a) the environment will be built in itself and b) there would be no technological distinction between programming and using the system, to match the ontological nature described in section 1.2. Any single Turing-complete system would do for such a concept, but of particular power is Squeak Smalltalk [16], which is attractive for our purposes because it is written in itself (beyond fundamental logic and some hardware calls), which means that the language and environment itself can be modified. Single universal systems in general present security and intellectual property issues, and currently the Squeak environment has usability, visualisation, multimedia and aesthetic problems which makes it less than ideal for artistic expressivity, but there is potential for these to be developed by interested parties.

### 2.2 Creativity Support

There is a need for an interactive art programming environment to support the creativity, as well as the expressivity, of its users. Earlier work [30] reviewed the work of creativity support researchers, in particular the generalised operations which they had identified as being useful for creative workers. Ben Shneiderman lists eight specific operations that should "help more people be more creative more of the time": Search-

ing (for knowledge and inspiration), Visualising, Consulting, Thinking, Exploring, Composing, Reviewing, Disseminating [25]. In the digital art domain some of these tasks would be highly interrelated (for instance, visualisation, exploration and composition). Michael Terry and Elizabeth D. Mynatt highlight the need for support of Schön's theory of reflection-in-action: near-term experimentation (previews of the results of actions), longer-term variations, and evaluation of actions [29]. In our own study, empirical evidence was used to identify some examples of aspects of creative exploration: Breaking with convention, immersion in the activity, holistic view and parallel channels of exploration [7].

### 2.3 Methodological Support

It is unclear whether and how methodologies for making interactive art differ from conventional software development methodologies, not least because not much is known about the practical realities of either of these, hence it is difficult to evaluate technological support for any practical software development methodologies. As Kautz et al. relate, "The literature on [Information Systems development methodologies] is extensive and wide-ranging. It consists however largely of prescriptive and normative textbooks and work that is based on anecdotes, but there is limited scientifically collected and analyzed empirical documentation…" [19]. Kautz et al. provide an illuminating study of actual use of development methodologies in one large company, but the implications for the less economically motivated, more experimental and usually smaller-scale development in interactive art are unclear.

There is one exception: art developers' and software methodologists' views are converging on Human-Centered design. Höök, Sengers and Andersson [15], and Robertson, Mansfield and Loke [24] are artists, HCI researchers and software development methodologists looking at ways to use HCI techniques to evaluate interactive art, and contend that audience design evaluation should be a formal part of interactive art's development process.

### 2.4 Social Support – Communication and Collaboration

Collaborations take place within two styles of communities; one is 'Communities of Practice' (CoPs) [22] and the other is 'Communities of Interests' (CoIs). In a CoP, people work together who come from similar backgrounds, e.g. programmers who are producing software. In CoIs, (which are also 'communities of communities' [2]), people who come from different backgrounds concentrate the same project, e.g. art-technology collaborations. Communication between collaborators in CoIs is difficult because they come from different backgrounds and therefore use different languages and different knowledge systems. The fundamental challenge which faces CoIs is to build a shared understanding of the task at hand. This shared language rarely exists at the beginning, but is evolved incrementally and collaboratively and emerges in people's minds and in external artifacts [8]. Boundary objects [27] are important artefacts in CoIs, because they have meaning across the boundaries of individual knowledge systems. Boundary objects allow different knowledge systems to interact by providing

a shared reference that is meaningful within both systems and they serve to communicate and coordinate the perspectives of various constituencies [32].

Our earlier studies have reported on the use of the visual programming language Max/MSP in close collaborations, partly by supporting such shared references. As well as being a popular programming environment for interactive art, it is suggested that Max/MSP can also contribute to the social interaction between collaborators. Weakley et al. [31] suggest that the developing Max/MSP program can be regarded almost as a working sketch of the finished artefact. In the same way that one may work collaboratively using sketches to explore, explain and develop ideas, the expressive and accessible nature of the visual program encourages collaboration. Edmonds et al [6] report seeing "... a radical shifting and sharing of responsibility" with both parties able to contribute to the other's domain of interest. Of course, this is not to say that such visual programming languages (VPLs) are a panacea. For instance, Green and Petre [14] identify a number of areas where, in terms of usability, VPLs appear to be worse than text-based languages. In the VPLs that they studied, they identified notable shortcomings in the two areas of 'viscosity' (the diagram editor associated with the VPL makes it difficult to change the developing program) and 'secondary notation' (the readability and clarity of text-based programs is typically enhanced by such secondary notations as whitespace or commenting, and the VPLs that Green and Petre studied tended to constrain the ways that objects could be arranged or highlighted). Max/MSP shares such constraints to a lesser extent than the VPLs examined in Green's and Petre's study. It is often desirable and necessary, to spend some time tidying up the program, particularly if it is to be easily understood by another person. This is, however, also true of text-based languages (and, it might be said, of any human endeavour!) and, in comparison with text-based languages, Weakley et. al. report on MAX/MSPs particular ability to allow the programmer to deliberately make the program more self explanatory (by colour-coding, highlighting or quickly adding live visual elements to the code to indicate the status of processes), and how that ability assisted the artist-technologist collaboration that they studied. Max/MSP also supports real-time interaction between collaborators by allowing the program to be modified while it is actually running and therefore it supports synchronous collaboration. This is in contrast to a more traditional approach where an artist might seek help from a programmer who would work largely alone, successive iterations of the program being tested from time to time; a much more disjoint process.

Max/MSP is an instance of technology which supports close collaboration, but the technical system was not specifically designed for this purpose; instead the support for collaboration arose from use of the tool itself. Although there are many tools available to support *communication*, we do not yet have dedicated tools which explicitly support *collaboration* in the development of interactive art.

## 3. The Supportive Role of the Programmer

As outlined in sections 2.3 and 2.4, not much is known about what precisely it is that programmers do in the context of producing interactive art, besides writing programs. Section 1.3 shows the artistic and technological rationale for working with program-

mers, but the question remains: how good are programmers at living up to the need for them? Specifically, which facets of the programmer's role are supportive to the artist, and which are obstructive? How can technology enhance the support and ameliorate the obstruction? To answer these questions, we carried out a social study on the role of the programmer in art collaborations.

### 3.1 Methodology

An approach based upon grounded theory was adopted for this study. Grounded Theory [12, 13, 28] is an approach where the theory emerges from the data itself, and is thus grounded in it, rather than being an approach which tests existing theories. The theory's emergence from the data is good for discovery of the important issues in a field, because any biases or preconceptions held by the researcher should have minimal impact.

There are two main schools of grounded theory analysis, those of Anshelm Strauss [28] and Barney G. Glaser [12], former colleagues, and the two originators of the approach [13]. The two disagree on whose version is most correct. We have chosen to follow Glaser's approach, since it appears to be more elegantly simple, and his critique of the partiality of Strauss and Corbin's question-asking process is convincing.

The first iteration of grounded theory analysis is the open coding stage, which begins with no preconceived codes, and produces codes from the asking of neutral questions of the data. Glaser then advocates several iterations of gathering more data, comparison and selective coding in order to establish categories and super-categories of codes, and to situate the most important categories within a framework that shows the overall picture. Categories that get 'saturated' with data points are more important to a theory than categories that do not.

The preliminary source for open coding was the collected Case Reports for the COSTART (Computer SupporT for ARTists) project, which centred round seven intensive artist-technologist collaborations. Artist, technologist and observer statements, from recordings, interviews and diaries, for seven projects, were coded by hand, and arranged to produce a list of categories.

More codes were taken from primary data – transcripts of interviews with artists and technologists from the larger and more meticulous COSTART 2 project [4], which involved 9 further residencies, bringing the total to 16 artists, 6 technologists and 4 observers. These were coded using NVivo software [11]. At this point there was a list of categories indicating the many and various issues that arose and were recorded during the collaborations.

It was decided that an appropriate way to gather some of the data needed to saturate some of these categories was to conduct a series of qualitative interviews with artists and programmers who had been involved in collaborations. In order to distinguish the feelings and actions of people in the artist role from the feelings and actions of people in the programmer role, and to explore the issues facing non-programmers, initial subjects were selected who were interactive artists who do not program, and programmers who program for such artists.

The interviews were conducted in December and January 2005, face-to-face (with one exception where iChat was used), were recorded and, for the first iteration, tran-

scribed for coding in NVivo. Six subjects were chosen—four programmers and two artists who had worked with these programmers.

The interviews were semi-structured and qualitative, with questions designed according to best practice (as described in [10]), based upon the concepts that were popular, but as yet unsaturated. The lengths of the interviews ranged from 30 minutes to 1 hour 45 minutes.

After the several iterations of selective coding and data gathering, a hierarchy of about 200 codes and categories of codes emerged, with associated memos describing the relationships between codes according to grounded theory analysis.

To present the findings for this paper, we selected out several irrelevant codes, such as ownership and quality issues.

To avoid repeating ourselves, we also selected out incidents that further validated what has already been discussed in section 2. This left us with categories and memos from which we can form a grounded theory about the core social values, rather than those dictated by technology, in the relationship between artist and programmer.

### 3.2 Findings and Analysis

The memos from the study were collected to form the theory that follows. Broadly speaking, our theory contends that of fundamental importance is the process of *attuning* between humans, and between artist and programmer-attuned computer. The programmer's role is to attune the computer to the artist, through either 'intimate iteration' or 'toy-making'. Ideally the attuning happens to the extent that the artist no longer needs the programmer, yet this does not always happen in practice because of the extra work involved. Technological development of art systems starts with an artistic description which is transformed into a technological description by the programmer, starting with fundamental technological choices, which are tested with the artist for appropriate artistic meaning, then abstracted and built into higher-level structures.

We will now explore the theory in detail.

**Collaboration Context.** Artists work with programmers for several stated reasons, given that a need was established for a computer system beyond the capabilities of the artist his or herself to create. With non-programming artists, there is a definite hierarchy, albeit, a complex and contested one. Briefly, the perception is that programmers are *craftsmen* who are engaged as assistants, consultants and teachers, depending on the artist's expertise, willingness to learn and skills of others in the project. Yet artists must at the same time *trust* programmers to estimate and lay out the work plan, and make thousands of other decisions about the project.

Mamykina et al. used the COSTART data to discern types of artist-technologist collaboration (technologist as assistant, full partnership with artist control, and full partnership) [21]. Although our findings align with that theory, we didn't find an explicit example of a "full partnership" that involved non-programming artists.

Ability to trust the programmer comes from valuing the programmer's *personality, expertise* and *sensitivity*. Expertise (which is particularly valued with *agility*, or willingness to reach beyond expertise) is interesting, because artists will often want to

challenge expertise, by testing their own hunches, or requiring extensions to or re-evaluations of the expertise. An example in the data is of an artist requiring non-realistic lighting from a VR expert. Sensitivity refers to the programmer's ability to respond to the artist's needs, both explicit and, importantly, implicit, making it a pre-requisite for *attuning*.

We found that programmers work with artists for different sets of stated reasons, primarily *satisfaction* (fulfilling goals, such as earning money, technical challenge, ownership) and *enjoyment* (with no well-defined goals, for example creative urges, 'soulfulness', to teach, friendliness). When programmers are motivated by learning (technical and aesthetic skills), they seem to derive both satisfaction and enjoyment.

The level of *dependency* of the artist on the programmer, combined with the motivations of each, has a strong influence on the passivity of the programmer – their level of comfort with, for example, suggesting changes for either technical or aesthetic reasons. However, there appears to be less which influences the ability of the programmer to suggest changes for pragmatic reasons.

There was some evidence of *subversive* or *passive-aggressive* behaviour amongst programmers who made changes without consulting the artist, or made disingenuous suggestions, because it was technically easier, or because they preferred one aesthetic course over another. It was difficult to get information in retrospect about the effects of this behaviour without breaching confidentiality, though the artists in the study were generally not shy of suggesting even 'pedantic' changes, and any dissatisfaction with programmers was due to 'inflexibility' or 'grumpiness'.

**Approach to problem-solving.** In all cases where it was mentioned, artists presented, or were characterised as initially presenting, what to a technologist sounds like very imprecise descriptions of the systems they envisaged. The form of these descriptions ranged from using 'vague language' through describing it 'in terms of effect' to consciously communicating a 'metaphorical understanding'. There was some evidence of dissatisfaction at this approach amongst programmers who wanted more specificity, logic and sequencing. However, this artistic meaning needs to be understood by the programmer – it is the first stage of the *attuning* process.

What then appears to immediately take place is a process of transforming this conceptual description into technological terms. This was mainly achieved by *question asking* on the part of the programmer – about critical detail, what-if scenarios, and so on. Rather than the traditional requirements gathering process of 'breaking down' a 'top-level' description into 'low-level' terms, this process seems to be more a way to informally map the artistic expression of the piece into a technological expression, at all levels of expression. We speculate, with two data points in support, that this is also a comfort-bringing process of changing the problem domain from an open world into a 'hermetic' system.

In order for the programmer to check that his mapping of the artistic expression to the technological expression is appropriate, it becomes necessary for him to test fundamental technological components with the artist, beginning with the most fundamental, to see if the artistic meaning these components create is appropriate. In our studies, the first result of the mapping is often a perception of what technology will be used, or what potential technology needs to be researched, first in terms of hardware,

then programming environment, then fundamental algorithms (a reversal of this process was found in one collaboration where the piece was based upon an algorithmic process described by the artist – the initial development was characterised as 'an engine' around which input and output could be added). As these fundamental things are chosen, the artist in turn asks the technologist several questions about this low-level technology. The artist and technologist begin to get *attuned* over the low-level technology.

Out of all cases, there was only one example of a formal top-down approach to design, when the artist was able to be quite specific about required features (in this case the system was explicitly a toolkit).

Both artist and technologist seem to agree that 'knowing the rules' for the system allows the system to be developed. But here arises a dilemma: the programmers in the study indicated a greater level of comfort with and satisfaction from achieving set rules and goals within a design, whereas it was difficult to get these rules from the artists, and the artists indicated a need to 'play' in order to discover what the system should do. This is essentially the dichotomy between *analysis* and *synthesis*.

**Developing subsystems.** Bottom-up development was used in all of our studies. Small programs are developed by the programmer to test each of the fundamental technologies. One respondent likens the process to sketching, as a way of finding out more about the sub-problem: "I make things up as I go along, usually, I think. But I think that's more my designery [sic] training because with that you have a vague idea, then you, like, draw it and then look at that and discover something new in the drawing."

Many of the programmers exhibited a particular preoccupation at this point with the input and output technologies system. This appeared to be for a variety of reasons: chiefly, that these are the lowest-level technologies; also that they are the hardest thing to get right (input is characterised as being technologically harder than output, partly because of the human-controlled element, partly because computer technologies have greater bandwidth for output than input so output is easier to model, and partly because interactive art often involves unfamiliar sensor hardware). As one respondent puts it, "usually I want everything to talk to everything else before I start working on how decisions are made … invariably you'll make the brain wrongly if you don't have the right shaped skull for it".

Simultaneously and separately, in cases where it was applicable, the artist works on his or her material ("the content") then presents it to the programmer. This can be seen as a process of attuning between the artist and his or her material, and also between the artist and programmer. There was one case where this approach helped the artist feel as if she had something to do. There is no evidence of a particular methodical approach here, which is attributable to the uniqueness of the artists' practice.

Once sufficient understanding of the sub-problem has been acquired, the programmer begins to generalise and to build up the sketch: "small parts of the system [are] being experimented with and you see really how they operate and you, as in the artist, or me, as in the programmer, are having to think about how these things fit into the system, then that's where the ideas of generality and structure and abstractions start to come in. It's an interesting thing."

**Intimate Iteration vs. Toy-making.** At all stages of development, there will be facets of the technology that the artist will want to make decisions about, and facets that are not of interest. Reports about the level of engagement that artist had with the system varied from being interested in everything (itself associated with desire to learn programming), to being interested only in 'front-end' facets, i.e. those aspects which would have an effect on the audience. Decision points for artists can be triggered by any of the artist, programmer or computer (via the programmer). The artist's decision-making process was described by one programmer as a way of adding the 'character' to the system.

In many cases, the artistic decisions were made by working intensively with the programmer, who makes small changes to test different outcomes. This 'intimate iteration' seems to work well enough in some cases, and may save time, but there was at least one case of an artist saying she did not feel she was 'hands-on' enough.

A more encouraging approach, in terms of the goal of creating a supportive environment, was for the programmer to build a technological 'toy' for the artist. We found this to be useful for several reasons. Firstly, concept of a toy directly aligned with the oft-reported behaviour of artists 'playing' with systems, data, mappings and algorithms, in order to discover both the necessary rules for the system (remember the earlier concept that finding the rules would allow the problem to be solved), and exploring what one artist called 'probabilities and tendencies' within the data. Secondly, producing the toy is a way for the programmer to take himself 'out of the loop'; this means that, as one programmer put it, 'by taking myself out of the loop it makes it really clear what the dynamics of the system are as opposed to what my interpretation is'.

Thirdly, and crucially, as one artist stated, "instead of [the programmer] just doing it and you saying 'can it be more squiggly?' and him going back and changing parameters, I found I *understood the language of the algorithm* just by playing with the parameters and understanding what the software developer, how they had broken down this organic thing" (our emphasis, edited for repetition and anonymity). The artist was not referring to the language in which the algorithm was implemented (Python, incidentally), but rather the language necessary to communicate meaning to and from the algorithm itself. This attuned manipulation of an algorithm's language produces meaning *both technically and aesthetically*.

## 4. Conclusions

Returning to Kautz et al,'s 2004 study [17], mentioned first in section 2.3, we can draw some comparisons between the commercial software development in that study and the non-commercial software development in this. The Kautz study identified five categories which affected the practical use of systems development methodologies: *universality* (there is no universally applicable methodology), *confidence* (in the work's progress as shown by the methodology), *experience* (means a programmer is more likely to pick and choose specific techniques from methodologies rather than follow them all), *co-determination* (choosing the methodology created a feeling of

responsibility to use it) and *introduction* (providing appropriate training in and transition to a methodology).

Except for in one of the cases in our study, we found no cases of plan-centric (or 'engineering-style') software development methodologies being used. (The exception was in the case where the artist was able to specify precisely what he wanted, and a requirements analysis and UML diagrams and so forth were produced.)

We found Kautz et al.'s categories to be in alignment with this study, but the small scale of the communities, and the high expertise and autonomy of the programmers means that *introduction* category becomes virtually irrelevant, and conversely that the *expertise* category becomes highly relevant. The *confidence* category is also diminished in importance, because the artists (the equivalent of customers or managers) are closely involved in the development process, and trust of programmers is valued. *Co-determination* becomes sometimes inaptly named, as usually the programmer is the only individual with a stake in the methodology choice. In the large collaboration we studied, *co-determination* seems an important factor in methodology adoption by individuals concerned, but the details of this would require further investigation.

*Universality* is interesting. The approaches taken by programmers working with artists could all (with the single exception mentioned above) be characterised as being forms of agile programming, and indeed a form of agile programming was explicitly adopted as the methodology for the large project in the study. One of the artists from that project said: "I continue to be attracted to the philosophies and some of the techniques of agile programming, principally because I see it as a means of operating whereby you do have a holistic understanding of the eventual delivery but you address immediate and evolving needs rather than fantasising that you can describe a project in the brief right at the start". Agile methodologies are characterised by their ability to accept changing situations, rather than trying to predict what they might be. They also tend to place themselves in subordination to the people using them, rather than being processes or frameworks in which people must operate. These things make it eminently suitable for developing artwork, since the technological requirements are often imprecise throughout the project, and also that part of creative practice is breaking with the convention of processes and frameworks. However, agile development methodologies used in isolation would result more in the 'intimate iteration' style of development, identified in this study to be limiting in terms of artist engagement with the computing medium.

Which brings us on to the next important finding – that 'playing' with technological 'toys' is crucial to the development of interactive art systems, for six reasons, reiterated briefly here:

1. finding rules,
2. developing the 'character' of the system,
3. intuitively learning the 'language' of the algorithmic system (which is distinct from the less-intuitive language in which the program was written),
4. making the toy's place within the system apparent,
5. producing technical and aesthetic meaning simultaneously (which also assists transdisciplinary collaboration), and
6. making the artist feel more comfortable and empowered.

These toys, then, are boundary objects that embody a language for creating meaning between artist and technologist, and artist and computer, and perhaps less necessarily, technologist and computer. (There is also scope for these toys to become boundary objects for other communities, such as audience members, other artists and technologists, researchers and curators).

Software engineering methodologists should look into whether toy-making is beneficial to other kinds of interdisciplinary software development communities, and into ways that the toy-generation process can become a formal part of methodologies for these communities.

It is worth considering briefly the potential of the intuitive language-conveying abilities of toys to encourage more abstract thinking about the computer medium itself than would necessarily be involved in engaging with a particular system. We can take one pointer from linguistics: Boroditsky and Ramscar claim that abstract knowledge can be built up from experience-based knowledge [1]. They write: "To properly characterize abstract thought, it will be important to look not only at what comes from innate wiring and physical experience, but also at the ways in which languages and cultures have allowed us to go beyond these to make us smart and sophisticated as we are." – we have found evidence to show that learning the language of an algorithmic system takes place as a result of toy-using, but the ways in which we can use such new languages to go technologically beyond the experience of playing with the present algorithmic system are not yet clear.

Designers of programming languages and programming environments for interactive art communities should take into account the support needs identified in section 2 of this paper, and also this new study's finding of the usefulness of toy-making in these and potentially other creative collaborative endeavours.

## 5. Acknowledgements

The authors would like to thank the interview respondents for their insights, Dr. Tim Mansfield of DSTC and the reviewers for their thoughtful and helpful comments. Part of this paper is based upon Turner et. al. Seeing Eye-to-Eye:Supportive Transdisciplinary Environments for Interactive Art in IV05 Conference, London, 2005 (forthcoming), ©2005 IEEE.

## References

1. Boroditsky, L. and Ramscar, M. The Roles of Body and Mind in Abstract Thought. Psychological Science, 13 (2). 185-189, 2002.
2. Brown, J.S. and Duguid, P. Organizational Learning and Communities-of-Practice: Toward a Unified View of Working, Learning, and Innovation. Organization Science, 2 (1). 40-57, 1991.
3. Campbell, J. Lessons in Object-Oriented Programming, 2003.
4. Candy, L. and Edmonds, E.A. Explorations in Art and Technology. Springer, 2002.

5. Cornock, S. and Edmonds, E. The Creative Process Where The Artist Is Amplified Or Superseded By The Computer. Leonardo (6). 11-16, 1973.
6. Edmonds, E., Candy, L., Fell, M., Knott, R., Pauletto, S. and Weakley, A., Developing Interactive Art Using Visual Programming. in HCI International 2003 Proceedings, (Crete, Greece, 2003), Lawrence Erlbaum Associates, 2003.
7. Edmonds, E.A. and Candy, L. Creativity, Art Practice and Knowledge. Communications of the ACM, 45 (10). 91-95, 2002.
8. Fischer, G., Communities of Interest: Learning through the Interaction of Multiple Knowledge Systems. in IRIS'24, (Norway, 2001), 2001.
9. Flanagan, M. and Perlin, K., Endpapers: collaboration, process and code. in ISEA2004, (Helsinki, Tallin, 2004), 2004.
10. Foddy, W.H. Constructing questions for interviews and questionnaires: theory and practice in social research. Cambridge University Press, Cambridge, England; Melbourne, 1992.
11. Fraser, D. QSR NUD*IST Vivo: Reference Guide. Qualitative Solutions and Research Pty., Melbourne, 1999.
12. Glaser, B.G. Basics of Grounded Theory Analysis. Sociology Press, 1992.
13. Glaser, B.G. and Strauss, A.L. The Discovery of Grounded Theory: strategies for qualitative research. Aldine Publishing Company, Hawthorne, N.Y, 1967.
14. Green, T.R.G. and Petre, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. Journal of Visual Languages and Computing, 7. 131-174, 1996.
15. Höök, K., Sengers, P. and Andersson, G. Sense and Sensibility: Evaluation and Interactive Art. CHI Letters, 5 (1). 241-248, 2003.
16. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S. and Kay, A., Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. in OOPSLA'97 Conference, (Atlanta, Georgia, 1997), 1997.
17. Kautz, K., Hansen, B. and Jacobsen, D. The Utilization of Information Systems Development Methodologies in Practice. Journal of Information Technology Cases and Applications, 6 (4), 2004.
18. Kay, A. and Goldberg, A. Personal Dynamic Media. Computer, 10 (3). 31-41, 1977.
19. Kurtz, K., Hansen, B. and Jacobsen, D. The Utilization of Information Systems Development Methodologies in Practice. Journal of Information Technology Cases and Applications, 6 (4), 2004.
20. Maeda, J. Design By Numbers. MIT Press, 1999.
21. Maeda, J. and Burns, R. Creative code. Thames & Hudson, London, 2004.
22. Nardi, B. A Small Matter of Programming. MIT Press, Cambridge, MA, 1993.
23. Negroponte, N.P. Being Digital. Alfred A. Knopf, New York, 1995.
24. Robertson, T., Mansfield, T. and Loke, L., Human-Centred Design Issues for Immersive Media Spaces. in Futureground 2004, (Melbourne, Australia, 2004), 2004.
25. Shneiderman, B. Creativity Support Tools: Establishing a framework of activities for creative work. Communications of the ACM, 45 (10). 116-120, 2002.
26. Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages. IEEE Computer, 16 (8). 57-67, 1983.
27. Star, S.L. The Structure of Ill-Structured Solutions: Boundary Objects and Heterogenous Distributed Problem Solving. in Gasser, L. and Huhns, M.N. eds. Distributed Artificial Intelligence, Morgan Kaufmann, San Mateo, CA, 1989, 37-54, 1989.
28. Strauss, A.L. and Corbin, J. Basics of qualitative research: Techniques and procedures for developing grounded theory. SAGE Publications Ltd., Thousand Oaks, CA, 1998.

29. Terry, M. and Mynatt, E.D., Recognizing Creative Needs in User Interface Design. in Creativity and Cognition Conference 2002, (Loughborough, UK, 2002), ACM Press, 38-44, 2002.
30. Turner, G. and Edmonds, E.A., Towards a Supportive Technological Environment for Digital Art. in OzCHI 2003: New directions in interaction, information environments, media and technology, (Brisbane, Australia, 2003), 2003.
31. Weakley, A., Johnston, A. and Turner, G., Creative Collaboration: Communication Translation and Generation in the Development of a Computer-Based Artwork. in HCI International, (Las Vegas, 2005 (to appear)), 2005 (to appear).
32. Wenger, E. Communities of Practice - Learning, Meaning and Identity. Cambridge University Press, Cambridge, England, 1998.