Seeing Eye-to-Eye:
Supportive Transdisciplinary Environments for Interactive Art

Greg Turner, Ernest Edmonds & Alastair Weakley
Creativity and Cognition Studios, University of Technology, Sydney
greg@gregturner.org, ernest@ernestedmonds.com, alastair@weakley.org.uk

Abstract
This paper presents findings from a study of the social and technical roles of programmers in art-technology collaborations. Combined with a review of the supportive and obstructive roles of technology with respect to helping artists to learn programming, 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. All this is used to suggest some high-level ways in which visualisation technology can be employed to aid both the programming process (for artist and technologist) and the art-technology collaboration process.

1. Motivation

1.1. What is Interactive Art?

In 1973 Stroud Cornock and Ernest Edmonds proposed that rather than talk about ‘artworks,’ it was helpful to think in terms of ‘art systems’ that embraced all of the participating entities, including the human viewer [1]. For the purposes of this paper, we define interactive art as an art system that changes from the presence of or actions by the audience-participant. Viewing interactive art as an art system shows us that interactive art is quite a complex field, involving various creators and audiences, not just a set of computational artefacts with an ‘optimal’ configuration for the task at hand.

1.2. How to Think About Programming

We can say that programming is the articulation of statements in a programming language. Programming is made technologically possible by the intertwined existence of both a programming language and an environment in which to make articulations in that language.

But what is a programming language? It is short-sighted of us to include only those styles of programming that involve writing and compiling ASCII code, or to add only visual programming languages (VPLs) and end-user programming (EUP) environments (many such languages (e.g. Max/MSP [2]) are indistinguishable from other forms of GUI manipulations).

Limiting ourselves to a formal requirement, such as Turing completeness (i.e. the property of a logical system that it can calculate anything that is capable of being calculated [3]) does little to cure our myopia. Not only are useful programming-like activities carried out in non-Turing-complete systems (e.g. spreadsheet formulae with no cycles), but also the primacy of Turing completeness recedes in the context of interactive systems. As Edsger Dijkstra commented “Modern machinery is basically more difficult to handle than the old machinery. ... we have got the I/O interrupts, occurring at unpredictable and irreproducible moments” [4]. This nondeterministic programming contributed to what Dijkstra called “the software crisis”.

In fact, any reasonable definition of programming today (for example, the common one that programming is a specification of a computation) can describe all uses of a computer. This means that there is no particular ontological distinction between programming a computer and using it, so we might as well call all uses of a computer different forms of ‘programming’. We do not ignore the obvious differences between, say, the language of Direct Manipulation [5] and Assembly
Language. In fact, we can use these differences to place different sorts of programming languages on an axis between two poles (see Figure 1), which we call ‘popular programming’ and ‘deep programming’. The axis is broadly labelled ‘Complexity’, but there are relationships with the level of abstraction from the hardware, the granularity and flexibility of the language. This concept and its implications are explored further in section 2.1.

‘Popular programming’ ranges from simple direct manipulation, through developing spreadsheet formulae, to, at best, hacking another’s JavaScript, Max or Director code. ‘Deep programming’ is characterised by expert usage of a general-purpose language, such as C or assembly language, in combination with an expert knowledge of computer architecture.

For the sake of simplicity, the term ‘programming’ will from now on be used to refer to the entire continuum between popular and deep.

1.3. Why is Programming Important to Art?

We have known for some time that computers allow us to think thoughts that are impossible to think with any other tool, 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) exploit one or more of these.

Treated briefly here, ‘Speed’ refers to the computer’s ability to do certain things quicker than we can. In interactive art, the goal is often to generate the response ‘in real time’. ‘Slavery’ is descriptive of both the incredible cheapness and unquestioning obedience of computation, and is what allows artists and programmers to create massive and wide-ranging programmatic edifices. ‘Synaesthesia’ is a way of describing that all a computer does is perform operations on collections of 1s and 0s. Inside a computer, video 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 of the four to nail down, perhaps because it is a mental operation with 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 say that everyone should learn to program (e.g. Kay and Goldberg [6], Flanagan and Perlin [7]), and people who say that artists who make computer art must learn to program (e.g. Maeda [8], although he has since revised this view).

However, since many artists make good computer art without deep programming, we have to assume that programming is not always necessary. Although many artists do indeed find the requisite level of engagement by learning the necessary technical skills from scratch, many others find it vicariously, by delegating the work to a programmer. Some unanswered questions: What is it that a programmer does when forming structure? (How) can information visualisation techniques be used to assist in engaging with the four Ss?
2. The supportive role of technology

2.1. Expressivity support

An important issue at stake is language expressivity (expressivity presumably being valued in tools for artists). Broadly speaking, popular programming languages allow us to make “big brush strokes”, and achieve impressive things with not much effort, but at the expense of flexibility – popular languages have easily-reached limitations. Less limited are deeper programming languages, but it is difficult to construct large systems from small articulations. A truly expressive language would allow both large and small granularity, both obvious gestures and subtle nuances.

Figure 2 is a simplified sketch of a learning curve for someone moving from popular programming to deep programming, based on a similar sketch in [9]. The bumps in the curve indicate the size of, and the gaps the necessity for, conceptual leaps in order to overcome different types of ‘walls’, or limitations of tools, applications, languages, operating systems, and so on. The gradient of the curve indicates difficulty. For example, the diagram shows that the difficulty of changing to another command is less than the difficulty of changing to a new language or API. It also indicates that the first new language (for example) is harder to learn than subsequent new languages, which are made easier with experience. Most importantly, the diagram shows that there are currently no smooth ways of overcoming the limitations of a given toolset.

The dashed line shows an ideal learning curve, where the minimum time is taken to learn new technology, and effort is consistently rewarded. Approaching this line, and making the curve continuous would ease the transition from popular to deep programming, and would have definite learning, exploratory and expressivity advantages.

One way of achieving this is to have the entire computer system built around a single, 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 single 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 an example of particular power is Squeak Smalltalk [10], because it is written in itself, which means that the language and environment themselves can be modified.

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 [11] 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”: Searching (for knowledge and inspiration), Visualising, Consulting, Thinking, Exploring, Composing, Reviewing, Disseminating [12]. 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 [13].

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 [14].

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 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…” [15].

2.4. Social support – communication and collaboration

The close communication implicit in Mamykina et al.’s ‘full partnership’ or ‘partnership with artist control’ styles of art-technology collaboration goes beyond that which is well supported by typical communication tools [16]. A number of studies [17, 18] have, however, reported on the use of the visual programming language Max/MSP in close collaborations. Weakley et al. [18] 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. [17] report seeing “… a radical shifting and sharing of responsibility” with both parties able to contribute to the other’s domain of interest. This is an instance of technology in support of...
close collaboration, but the technical system was not specifically designed for this purpose; instead the support for collaboration was integrated with the tool itself. The popularity of visual programming tools in the COSTART (Computer SupporT for ARTists) residencies [19] is testament to the fact that although there are many tools available to support communication, we do not yet have dedicated tools to explicitly support collaboration in 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 programmers, 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 [20-22] 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.

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 [20] 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 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 [19], which involved nine further residencies, bringing the total to 16 artists, 6 technologists and 4 observers. These were coded using NVivo software.

It was decided that an appropriate way to gather some of the data needed to saturate some of the categories which had emerged 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, transcribed 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 [23]), based upon the concepts that were popular, but as yet unsaturated. The lengths of the interviews ranged from 30 minutes to 1 hour and 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 codes which did not fit well into the emerging theory, such as those relating to ownership and quality issues. To avoid repeating ourselves, we also do not discuss codes that only further validate 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.

4. Findings

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, ideally 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 a very top-level description, but then is rapidly transformed into low-level constructs by the programmer, which are attuned to, then abstracted and built into higher-level structures.
We will now explore the theory in detail.

4.1. 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 him or herself to create. The relationships aligned with the models of Mamykina et al. (assistant, full partnership with artist control, and full partnership) [16], although we couldn’t find an explicit example of full partnership which 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 prerequisite for attuning.

We found that programmers work with artists for different sets of stated reasons, primarily satisfaction (fulfilling goals, such as money, technical challenge, ownership) and enjoyment (with no well-defined goals, for example creative urges, ‘soulfulness’, to teach, friendliness).

4.2. Approach to problem-solving

In all cases where it was mentioned, artists presented, or were characterised as initially presenting, very imprecise descriptions of the systems they envisaged. The form of these high-level descriptions ranged from using ‘vague language’ through describing it ‘in terms of effect’ to communicating a ‘metaphorical understanding’. There was some evidence of dismay at this approach amongst programmers who wanted more specificity, logic and sequencing. However, this metaphorical communication is the first stage of the attuning process.

What then appears to immediately take place is a process of transforming this high-level conceptual description directly into low-level terms. This was mainly achieved by question asking on the part of the programmer – about critical detail, what-if scenarios, and so on. We speculate, with two data points, that this is not just translation of the problem to machine language, but also, because of the early timing of such questions, it could be a comfort-bringing process for the programmer of changing the problem domain from an open world into a ‘hermetic’ system with fewer unknowns.

The low-level result 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 initial development of a mathematical process was characterised as ‘an engine’ around which input and output could be added). This bottom-up approach seems to be because technologically the system is so dependent on these things. This causes the artist in turn to ask the technologist several questions about this low-level technology. The artist and technologist get attuned over the low-level technology.

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 for them 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 analysis versus synthesis.

4.3. Developing subsystems

Regardless of the goals of the approach, 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 to find 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 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 top-down or bottom-up approach here.

Once sufficient understanding of the sub-problem has been acquired, the programmer begins to generalise and 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.”

4.4. Intimate Iteration vs. Play and Language-Learning

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 can be triggered by any of the artist, programmer or computer (via the programmer). This decision-making process was described by one programmer as a way of adding the ‘character’ to the system.

In many cases, the decisions were made by working closely 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 attuned manipulation of a system’s language produces meaning both technically and aesthetically.

5. Implications for Information Visualisation

The technological support needs identified in the first part of the paper show that artists will want to engage with the speed, the slavery, the synaesthesia and the structure of the computing medium. The modus operandi of technologists studied in the second part shows that engaging with input and output is also an early, crucial part of interactive media. Tools that allow artists and programmers to visualise and explore these capabilities will be of fundamental importance. Speed, and slavery are built-in and self-evident, and need little visualisation support, beyond computing resource meters. Designing visualisation tools for synaesthesia (in our use of the word) is an interesting notion, because any given data may need to be visualised in any way. There is therefore a need for synaesthetic programming environments to support all of the computer’s output media, and to provide a range of tools for transforming the data into different forms for visualisation.

As we said earlier, structure is both the most unique and most difficult-to-learn aspect of current computing environments, and our social study has indicated that the only way that programmers learn structure is a combination of instruction and use. We propose a number of things to make learning structure easier. Firstly, that having a single universal concept to base programming on will mean a simpler structure paradigm to learn and explore, as opposed to the chimeric amalgamation that today’s applications, languages and operating systems impose. We can thus transform our thinking about the way computational artefacts (e.g. algorithms) work in a way that aligns with this concept, situated within a system of similar concepts, rather than to rely on limited metaphors (an algorithm is very often not a ‘recipe’) in the incoherent assemblage of conflicting technologies that exists today. Secondly, that programming environments need to represent structure (in terms of concrete objects, not just abstract classes) in an appropriate, manipulable, navigable form (in other words, not as static diagrams, or pages of ASCII) Messy programming should look messy, clean programming should look clean. Thirdly, in the case of interactive art, structure proceeds bottom-up, so it would be useful to visualise ways of making low-level toys that are easily abstracted within the environment. Fourthly, ways of seeing programming elements already situated within a structural context would be useful, so interactive working toys within the help system for the environment would assist in discovering the use of structure (one interview
respondent mentioned that the quality of the help infrastructure was a crucial element of his decision to use any given tool).

Which brings us on to the next important finding – that ‘play’ is crucial to the development of interactive art systems – for finding rules, developing the ‘character’ of the system, learning the ‘language’ of the system, making the toy’s place within the system apparent, producing technical and aesthetic meaning simultaneously (which also assists transdisciplinary collaboration), and making the artist feel more comfortable and empowered. So making ‘toys’ is useful, and consequently should be easy, and therefore well visualised. One way this could happen is to automatically generate a control panel for a chunk of code as the development of that code chunk progresses (but, having been generated, the control panel should be extensively reconfigurable for reasons of learnability, simplicity and synaesthesia). The control panel would act as a ‘toy’ for that code chunk, allowing people to play with the toy and become acquainted with the behaviour of the code chunk, particularly in terms of its inputs and outputs. We suggest that initially, because of the intuitive nature of play, data would be visualised in an intuitive way by using low-impact techniques such as Tufte’s sparklines (e.g. the word-size graphic as a trace of time-varying data) [24], and equivalents in other visualisation media. As play progresses, more cerebral, high-impact techniques can be used to visualise the more interesting components of the programming.

The recommendations we have made also align well with the creativity support guidelines listed in section 2.2. Support for Shneiderman’s ‘visualisation’, ‘exploration’ and ‘composing’ is obvious; we have proposed a way to simplify the technological aspects of the ‘thinking’ operation; ‘toys’ facilitate ‘composing’ and ‘reviewing’ and so on. Edmonds and Candy’s guidelines are largely satisfied by implementing an expressive system and its technological consequences, as described in section 2.1: For example, having no technological distinction between programming and using the system mean that there is no distracting code-compile-execute cycle, allowing ‘immersion in the activity’; having the environment written in itself allows the ‘breaking’ of its own ‘convention’, and ‘breaking of convention’ is supported within a system where only fundamental limits exist; a single universal concept aids the production of ‘holistic views’. Terry and Mynatt’s ‘near-term experimentation’ and ‘evaluation of actions’ are also addressed with the support of toy creation.

Collaboration support is provided as a consequence of toy-generation support – the process of mutual attuning, production and manipulation of the toys can be seen as the production of boundary objects and shared meaning amongst the agencies in the collaboration.

Finally, we can use several more isolated incidents of artists and programmers in the study speculating about what they would like to see in future technology, to propose some potential avenues for future exploration. One in particular said it was a ‘great goal’ to produce transparent code – in other words, code which does not need support. He currently gets as close as he can by extensively annotating the code, but perhaps with the inclusion and dissemination of ‘toys’ for exploring and familiarising with the code, transparency can be achieved in more immediate ways.

Also along the lines of annotation, another programmer speculated about the potential advantages of behaviour recording and annotating tools, which could record and play back sequences of calculation, in order to help replicate and analyse the behaviour of a complex system. This recording, and visualisation thereof, could potentially be associated and disseminated with program ‘toys’, to demonstrate features and interesting scenarios within the code’s situation.

Casting our net wider, it seems that the importance of the program as a situated artefact within an art system may be of interest to a wider cultural society. Tools for recording and visualising the development of interactive art could be of interest not only to programmers and artists, but also to curators and art historians. One example of such interest is Christiane Paul’s CODEDOC exhibition [25].

6. Conclusion

The topics we have dealt with in this paper have important high-level implications for designers of programming support environments. Such designers must learn to free themselves from thinking of programming as the manipulation of a one-dimensional ASCII language; abandon the concept of algorithms as ‘recipes’, and treat them more as situated actors in a system; abandon the code-compile-execute cycle; and to become aware that the programming environment is not hermetic, but situated within an interactive art system, a transdisciplinary collaboration, and the wider cultural society.

7. References


# Table of Contents

M2Vis Information ........................................................................................................... xv
Preface ............................................................................................................................... xvi
Acknowledgments ............................................................................................................. xvii
Committees ....................................................................................................................... xviii
Reviewers ......................................................................................................................... xxi

### Session MMViz 1_1: Plenary Discussion – Non-visual Visualisation: Learning from the Visual

#### Session MMViz 1_2: Sonification

- Polyrhythmic Organization of Coupled Nonlinear Oscillators ........................................... 5
  
  G. Baier, T. Hermann, and M. Müller

- User Evaluations of Interactive Multimodal Data Presentation ............................................. 11
  
  V. Salvador, R. Minghim, and H. Levkowitz

- MotionLab Sonify: A Framework for the Sonification of Human Motion Data ......................... 17
  
  A. Effenberg, J. Melzer, A. Weber, and A. Zinke

- Mode Independent Interaction Pattern Design ....................................................................... 24
  
  C. Frauenberger, T. Stockman, V. Putz, and R. Höldrich

### Session MMViz3: Haptics/Touch & Multimodal Interfaces

- A Comparison of Visual and Haptic Object Representations Based on Similarity ...................... 33
  
  T. Cooke, C. Wallraven, and H. Bülthoff

- Design of a Pressure Sensitive Floor for Multimodal Sensing ................................................ 41
  
  P. Srinivasan, D. Birchfield, G. Qian, and A. Kidané

### Session MMViz4: Non-visual & Multimodal Visualisation

#### Keynote Lecture

- Multi-modal Interfaces – Challenges and Opportunities ......................................................... 51
  
  C. Harding

### Session IV1_1: GeoViz – GeoVisualisation

#### Keynote Lecture

- Research Advances in Geovisualization and Remaining Challenges ....................................... 57
  
  M. Jem

### Session IV 1_2: KViz – Knowledge Visualisation in Organizations

- Visualization and Visual Modelling for Strategic Analysis and Problem-Solving .................... 61
  
  E. Kazancioglu, K. Platts, and P. Caldwell

- Knowledge Visualization in Practice: Challenges for Future Corporate Communication ........... 70
  
  S. Klein