Artist-Programmers and Programming Languages for the Arts

Christopher Alex McLean

Thesis submitted to Goldsmiths,

University of London,

for the degree of Doctor of Philosophy.

October 2011

Dedicated to Jess, Harvey, Mum, Dad, Andrea and Stuart and the rest of my family, to whom I owe everything, and with the memory of those who supported this work but passed away before seeing its end, namely my Father-in-Law Dr. David Elmore, and my cat Olga.

Abstract

We consider the artist-programmer, who creates work through its description as source code. The artist-programmer grandstands computer language, giving unique vantage over human-computer interaction in a creative context. We focus on the human in this relationship, noting that humans use an amalgam of language and gesture to express themselves. Accordingly we expose the deep relationship between computer languages and continuous expression, examining how these realms may support one another, and how the artist-programmer may fully engage with both.

Our argument takes us up through layers of representation, starting with symbols, then words, language and notation, to consider the role that these representations may play in human creativity. We form a cross-disciplinary perspective from psychology, computer science, linguistics, human-computer interaction, computational creativity, music technology and the arts.

We develop and demonstrate the potential of this view to inform arts practice, through the practical introduction of software prototypes, artworks, programming languages and improvised performances. In particular, we introduce works which demonstrate the role of perception in symbolic semantics, embed the representation of time in programming language, include visuospatial arrangement in syntax, and embed the activity of programming in the improvisation and experience of art.

Acknowledgements

I must first thank Geraint Wiggins for his attentive supervision, mentorship and insights, helping me along a fascinating journey. I have greatly benefited from his striving focus on honest, rigourous research, and I do not doubt that his support through MSc and PhD research required a leap of faith on his part, showing both an open mind and deep generosity. My thanks extend to all the other members of the Intelligent Sound and Music Systems group for daily lunchtime discussions on all things, I thank you all and must name my co-supervisor Mark d'Inverno, my collaborator Jamie Forth, and also Daniel Jones for his generous and detailed criticism. My thanks extend further to envelop the department as a whole, including Tim Blackwell, Mick Grierson, Janis Jefferies, Frederic Leymarie and Robert Zimmer for their encouragement and support. It has been a great privilege to work amongst such creative minds.

The path leading to this thesis really began in the year 2000, when Adrian Ward encouraged me to experiment with algorithmic composition. We formed the generative music (and later, live coding) band Slub, joined by Dave Griffiths. Many of the ideas explored here have roots in this collaboration, and I cannot bear to imagine life without Slub. My thanks to both.

My forays into Electronic and Software Art also paved the way to the present work, made possible with the support, encouragement and collaboration of Saul Albert, Amy Alexander, Geoff Cox, Olga Goriunova, Douglas Repetto, EunJoo Shin and Alexei Shulgin. My journey into research also received great support from my former colleagues at state51, particularly Paul Sanders, who has always been generous with rich ideas, encouragement and criticism. Thank you all.

He may not remember, but my first thoughts about a research degree were planted by Rob Saunders, the AI and Creativity researcher. We have hardly crossed paths since, but without his encouragement I may not have made the leap. Following this, Simon Emmerson was extremely supportive, including helping secure funding for my MSc, making the whole thing possible. Thanks both.

Speaking of which, I must also sincerely thank the PRS Foundation for funding the fees for my MSc in Arts Computing, and the EPSRC for the full funding of my PhD studentship. It is wonderful that these great institutions fund unusual areas of research.

The final stage of writing up the present thesis was completed within the OAK group at the University of Sheffield, special thanks go to Simon Tucker and Fabio Ciravegna for helping make the transition from research student to researcher so enjoyable.

I have met with many influential minds over the last four years, more than I could list here. But I really must thank Sam Aaron, Sarah Angliss, David Bausola, Renick Bell, Vaughan Bell, Ross Bencina, Alan Blackwell, Graham Booth, Ollie Bown, Andrew Brown, John Burton, Arturo Castro, Graham Coleman, Christof Damian, Eleanor Dare, Ben Fields, Philip Galanter, Joel Gethin-Lewis, Bruno Gingras, Sam Freeman, Matthew Fuller, Alex Garacotche, Enrico Glerean, Jake Harries, Cormac Heron, Scott Hewitt, Tom Holley, Sampath Jagannathan, Sergi Jorda, Ryan Jordan, David Lewis, Thor Magnusson, Charles Matthews, Jon McCormack, Andrew McFarland, Daniel Müllensiefen, Rob Myers, Eric Namour, Ashraf Nehru, Julian Oliver, Fredrik Olofsson, Irini Papadimitriou, Rob Partington, Aneeta Patel, Marcus Pearce, Hester Reeve, Christophe Rhodes, Chris Roast, Julian Rohrhuber, Olivier Ruellet, Ash Sagar, Andrew Sorensen, Dan Stowell, Andy Studer, Patrick Tresset, Kassen Oud, Michele Pasin, Juan Romero, Ge Wang, Mitchell Whitelaw, Scott Wilson, Matthew Yee-King, and everyone in Dorkbot, eugene, openlab, POTAC and TOPLAP.

I must also thank those who having bought me drinks, proofread my work, shared their ideas and/or offered helpful advice, have failed to find their name above. My special thanks to you, and I apologise for forgetting about you at this crucial moment, *especially* my friend and collaborator Nick Collins, I have reached this point only by relying upon Nick as a deep well of inspiration and enthusiasm over the past 10 years, and he really deserves better.

Finally I would like to thank Jess Elmore for her understanding and support, and our son Harvey McLean. Harvey is one month older than my PhD studentship, and his speedy development into a walking, talking, loving and complaining human being has been extremely useful in keeping me alert to the passing of time.

Contents

Research outputs

The present thesis has been developed and disseminated through the following activities and publications.

Practice

Acid Sketching (§2.6). Exploring drawn analogue symbols in digital art. Installed in the Dorkbot tent at the Big Chill Festival, August 2010.

Microphone (§2.8). Exploring vocal analogue symbols in digital art. Created with EunJoo Shin, installed at the "Unleashed Devices" group show at Watermans Gallery, September/October 2010.

Babble (§3.5.1). System for improvising musical timbre with written words. A permanent on-line work commissioned by Arnolfini, Bristol in November 2008.

Tidal (§4.4) and Texture (§5.6). Computer language designed for the improvisation of musical pattern, with visuospatial notation. Released with a free/open source license, and used by the present author in public music performances.

Live Coding Performances (§6.8). The following is a list of live coding performances featuring the present author during the period of research. Those marked with an asterisk were as part of a collaboration with Adrian Ward and/or Dave Griffiths as Slub, which is discussed in §6.10.

- Area10, Peckham, London, 12th April 2008 *
- Ivy House, Nunhead, London, 4th April 2008 *
- Thursday Club, Goldsmiths, London, 5th June 2008 *
- Public Life, London, 18th July 2008 *
- Secret Garden Party, Cambridgeshire, 26th July 2008 *

- Immersion, Flea Pit, London, 7th August 2008 *
- Placard Headphone Festival, Cafe Oto, 20th September 2008 *
- Noise=Noise, Goldsmiths, 21st January 2009 *
- Worm, Rotterdam, 9th April 2009 *
- STRP, Eindhoven, 10th April 2009 *
- pubcode1, Roebuck, 29th May 2009 *
- pubcode2, Roebuck, 5th August 2009 *
- spacecode, Plymouth Immersive Vision Theatre, 20th August 2009 *
- Shunt Lounge, London, 1st October 2009 *
- Transfer, Goldsmiths 16th October 2009 *
- Lambda Festival, Antwerp, 1st May 2010 *
- Performance with Scott Hewitt, Access Space, Sheffield, 29th October 2010
- With Matthew Yee-King, Jamie Forth and Scott Hewitt, FACT Liverpool, 3rd September 2010
- Piksel Festival, Bergen, 19th November 2010 *
- Performance with Nick Collins, Thursday Club, London, 15th December 2010
- Performance with Nick Collins, LAB, Nottingham, 17th December 2010
- VEX, Portland Works, Sheffield, 11th June 2011
- Placard Headphone Festival, Access Space, Sheffield, 16th July 2011
- Performance with Samuel Freeman, Dorkcamp, Watermillock, 20th August 2011
- Noise Box, York, 23rd September 2011
- Sony CSL Paris, 30th September 2011 *

Refereed Publications

Forth, J., McLean, A., and Wiggins, G. (2008). Musical creativity on the conceptual level. In *Proceedings of International Joint Workshop on Computational Creativity 2008*.

Forth, J., Wiggins, G., and McLean, A. (2010). Unifying conceptual spaces: Concept formation in musical creative systems. *Minds and Machines*, 20(4):503–532.

McLean, A., Griffiths, D., Collins, N., and Wiggins, G. (2010). Visualisation of live code. In *Proceedings of Electronic Visualisation and the Arts London 2010*.

McLean, A. and Wiggins, G. (2008). Vocable synthesis. In *Proceedings of International Computer Music Conference 2008*.

McLean, A. and Wiggins, G. (2009). Words, movement and timbre. In *Proceedings of New Interfaces for Musical Expression 2009*.

McLean, A. and Wiggins, G. (2010a). Bricolage programming in the creative arts. In *22nd Psychology of Programming Interest Group 2010*.

McLean, A. and Wiggins, G. (2010b). Live coding towards computational creativity. In *Proceedings of the 1st International Conference on Computational Creativity 2010*.

McLean, A. and Wiggins, G. (2010c). Petrol: Reactive pattern language for improvised music. In *Proceedings of the International Computer Music Conference 2010*.

McLean, A. and Wiggins, G. (2010d). Tidal - pattern language for the live coding of music. In *Proceedings of the 7th Sound and Music Computing conference 2010*.

McLean, A. and Wiggins, G. (2011). Texture: Visual notation for the live coding of pattern. In *Proceedings of the International Computer Music Conference 2011*.

McLean, A. and Wiggins, G. (in press). Computer programming in the creative arts. In McCormack, J. and d'Inverno, M., editors, *Computers and Creativity*. Springer.

Stowell, D. and McLean, A. (2011). Live music-making: a rich open task requires a rich open interface. In *Proceedings of BCS HCI 2011 Workshop - When Words Fail: What can Music Interaction tell us about HCI?*

Chapter 1

Introduction

The history of computation is embedded in the history of humankind. Computation did not arrive with the machine, it is something that humans do. We did not invent computers, we invented machines to help us compute. Indeed, before the arrival of mechanical computers, "*computer*" was a job title for a human employed to carry out calculations. In principle, these workers could compute anything that modern digital computers can, given enough pencils, paper and time.

The textile industry saw the first programmable machine to reach wide use: the head of the Jacquard loom, a technology still used today. Long strips of card are fed into the Jacquard head, which reads patterns punched into the card to guide intricate patterning of weaves. The Jacquard head does not itself compute, but was much admired by Charles Babbage, inspiring work on his mechanical *analytical engine* (Essinger, 2004), the first conception of a programmable universal computer. Although Babbage did not succeed in building the analytical engine, his design includes a similar card input mechanism to the Jacquard head, but with punched patterns describing abstract calculations rather than textile weaves. While the industrial revolution had troubling consequences, it is somewhat comforting to note this shared heritage of computer source code and cloth, which contemporary artists still reflect upon in their work (Carpenter and Laccetti, 2006).

This early computer technology was later met with theoretical work in mathematics, such as Church's lambda calculus (Church, 1941) and the Turing machine (Turing, 1992, orig. 1947), which seeded the new field of computer science. Computer programmers may be exposed to these theoretical roots through their education, having great impact on their craft. As it is now practised however, computer programming is far from a pure discipline, with influences including linguistics, engineering and architecture, as well as mathematics. Digital computers now underpin the operation of business, military, academic and governing institutions, with impacts across human activity. These diverse backgrounds bring different approaches to programming, a great challenge for computer programming education.

As abstract machines, computers are multi-purpose, and are used in many ways towards many different ends. Judging by the contents of newstand magazines dedicated to them, the computer arts are most often framed as the use of software applications as design tools. Here software is produced by software houses, and bought and used by creative professionals. This situation has its merits, but is a diversion from our theme: we are interested in artists who write programs, not in those who only use programs written by others. Neither are we greatly concerned with the notion of computer programs as autonomous creative agents, although we will touch on this within broader discussion of programmer creativity (ch. 6). Instead we are interested in the practice of artists who get directly involved with computer languages as environments in which to create. They are end-user programmers, in that they create software not for others to use as tools, but as a means to realise their own work. We refer to such people as *artist-programmers*.

1.1 Artist-Programmers

The use of the term *artist-programmer* could be seen as over defensive. Alone, the word *programmer* is often used to imply a technician, tending a computing machine, or realising a designer's dream. We could be more assertive, and use the word *programmer* to establish a similar context effect to that which the word *painter* enjoys in the fine arts. But for the present thesis we keep the arts context explicit, while confronting the singular identity of the programmer as artist.

1.1.1 Computer art

We situate artist-programmmers within the *computer arts*, and so inherit important context from this field. We will first examine the role of industrial and military institutions in computer arts, before moving on to the issue of authorship and autonomy in the following section.

In Great Britain, computer art became established following the *Cybernetic Serendipity* exhibition shown in the Institute of Contemporary Arts, curated by Jasia Reichardt in 1968. On the whole this exhibition was well received, both in terms of reviews and the number and diversity of visitors. However as Usselmann (2003) notes, while the exhibition was successful in bringing some of the possibilities of computer art to public consciousness, it was significantly compromised. Despite the turbulence of the late sixties, there was no political dimension apparent in the exhibition, which Usselmann attributes to the inclusion of sponsoring corporations in the exhibition itself. He argues that this exhibition cast computer art into a form which later proved to be well suited for interactive museum exhibits, but has contributed little to critical debate around technology. As Usselmann notes, visitors were compelled by the ICA to "lose their fear of computers", whereas dissenting voices would advise otherwise, even then.

The model described by Usselmann persists far beyond Cybernetic Serendipity, for example the *Decode* exhibition at the Victoria and Albert museum in 2009 was sponsored by SAP AG, who commissioned the *Bit.Code* artwork greeting visitors to the exhibition. In their press release, SAP noted that "Bit.Code is themed around the concept of clarity, which also reflects SAP's focus on transparency of data in business, and of how people process and use digital information." In a 40 year echo of Cybernetic Serendipity, the artworks were used to promote humanising aspects of technology, with the artists tacitly taking on the political stance of their sponsor.

Despite such compromises in the public presentation of computer art, there have always been computer artists who engage closely with the sociopolitical context around their work. A few years after Cybernetic Serendipity, Nake (1971) published his essay "There should be no computer art" in response to the political compromises he then saw as implicit in computer art. Nake also took aim beyond computer art, giving a leftist perspective decrying the wider model of art dealer and art gallery, where art is sold for the aesthetic pleasure of the ruling elite. Taking the perspective of Usselmann (2003) alone, we might consider computer art as compromised, but Nake suggests that it is the whole art world that is compromised, and that the new computational media should establish alternative practices.

In a second 40 year echo, this time of Nake's essay, Oliver et al. (2011) focus on the social rather than artistic role of the *critical engineer*, quoting from their manifesto:

> The Critical Engineer notes that written code expands into social and psychological realms, regulating behaviour between people and the machines they interact with. By understanding this, the Critical Engineer seeks to reconstruct user-constraints and social action through means of digital excavation.

Oliver et al. (2011) are established artists, and exemplify the strong theme of activism present in contemporary digital arts. They highlight the unique opportunity for engaging with political themes where the boundary between digital arts and activism is blurred. This boundary has been explored by artists from the outset, with their work well charted by Neural magazine (`http://neural.it`).

Perhaps this dichotomy between politically engaged and disinterested art could reflect two distinct views of the relationship between programmer and software. This brings us to the subject of authorship in computer art.

1.1.2 Generative vs Software art

The different approaches to digital art described above can be understood in terms of a dichotomy drawn between *generative art* and *software art*. Arns (2004) describes generative art as work which approaches use of technology as a black box, with focus on end results. In contrast, she describes software art as focusing on technology and technological culture, where the software itself holds meaning.

Some argue that comparing generative and software art in this way is a category error. The most commonly referenced definition of generative art, and the one addressed by Arns (2004), is provided by Philip Galanter:

> Generative art refers to any art practice where the artist uses a system, such as a set of natural language rules, a computer program, a machine, or other procedural invention, which is set into motion with some degree of autonomy contributing to or resulting in a completed work of art. (Galanter, 2003, p.4)

Galanter (2003) states that his definition of generative art does not say anything about *why* an artwork might be made or anything about its content, as a definition of software art might. It is instead concerned only with a high level aspect of *how* art is made. He argues that the questioning theme of software art runs orthogonal to the definition of generative art, and therefore that the two do not bear comparison. However, central to Galanter's definition is the issue of autonomy: "Generative art must be well defined and self-contained enough to operate autonomously." (Galanter, 2003, p.4). Contrary to Galanter's assertion, the requirement of autonomy offers a strong constraint to the *why* of generative art; its operation must be considered separable from the programmer, who is in the business of creating activity external to their own influence. Software art by contrast admits views of software as an extension of the human, where the computer provides language which allows human expression to reach further.

According to Galanter's definition, we view the generative artist as emphasising distance between themselves and their work, and the software artist as embedding the activity of their software in their own actions. We position the artist-programmer towards the latter, directly engaging and interacting with their code as an integral aspect of their work. In the case of generative art, authorship becomes a fundamental question, who or what produces the art: is it the programmer or their autonomous process? In the case of software art, this question need not arise, as the process may just be viewed as part of the human artist's activity.

From the above we can see evidence of a divide throughout the history of computer art, between focus on aesthetic output, and focus on the processes of software, including its role in society. These are not mutually exclusive, and indeed Nake advocates both in balance. However, there is a tendency for computer artists and their audiences to focus on processes as disembodied, autonomous activity (§4.1). By exposing the activity of programming, perhaps we can adjust the balance towards focus on human interaction rather than autonomous processes.

1.1.3 Discussion

In the following chapters, sociopolitical context and critical frameworks around computer art are not our focus. Instead we turn our attention inward towards the intimate relationship between artist-programmers and their systems of language, understanding programming as a human interaction. The artist is considered for their role as a visionary, in exploring and extending their human experience. We consider technological support in terms of extending their vision through processes of perception and reflection in bricolage creativity. In particular, we expand upon the remarkable notion of programming language, to consider the role of these languages in the activity of creative art.

1.2 Programming Languages for the Arts

The 'tools' that artist-programmers use to make their work are formal, artificial languages. These languages are artificial in that they are constructed by individuals, rather than emerging from a wider cultural process as with natural languages. The word *artificial* is problematic in implying fakery, but as we will argue in §4.1, computer languages are only artificial in the sense that a desk fan produces artificial wind – the air still moves.

There are many thousands of programming languages, but they tend to fall within a small number of functional, structural and object oriented programming paradigms largely developed between the 1950s and 1970s. The most widely used programming languages[1] across institutions include Java, C, Basic and Python. These are all *general purpose* languages, with a core definition abstract from any particular task, albeit with add-on libraries which may target a particular problem domain. In using general purpose programming languages, artists must build their work using technology ostensibly designed for the general case, but in practice designed for the expression of discrete, logical structures, in an abstract medium ungrounded in human experience.

In the present thesis we question the extent to which contemporary, general purpose programming language environments are suitable for Artist-Programmers. We will argue for new approaches to the design of programming languages for the arts, with human cognition and

[1]An index of programming language popularity is maintained at `http://www.tiobe.com/`.

perception as primary motivating factors. In particular, language environments that relate both discrete and continuous representations, that consider visuospatial and temporal experience in the design of their notations, and that support software development as creative activity.

1.3 Aims

The work described by the present thesis is conducted towards two primary aims.

- To characterise human-computer interaction as a means for the improvisation of music and art, from the viewpoint of cognitive psychology, with strong focus on the human role in computer programming.
- To develop and demonstrate the potential of this theoretical understanding to enrich arts practice, through free/open source software applications and programming languages, installation art, live coding performances and workshops.

1.4 Structure

The following five chapters provide the core of the present thesis, where each successive chapter builds upon the chapter before, each time broadening in scope. Firstly *Symbols* (ch. 2) will provide a representational basis for the thesis, exploring units of representation and their role in the production and experience of computer art. The following chapter *Words* (ch. 3) considers the composition of symbols into words, and the expressive role of words in speech, music and computer programs. Next we consider the composition of words into the structures of *Language* (ch. 4), in particular the expression of pattern and meaning in natural languages, computer languages, and in music. Zooming out once more we view *Notation* (ch. 5), looking at the perceptual and temporal practicalities of how programmers may write programs. Finally our discussion will take in the wider context of *Creativity* (ch. 6), the role of programming in a creative exploration, with particular focus on musical improvisation.

Overall, the focus will be not on digital representations within computation, and not on analogue expression either, but on the interactions between the two. Computers give us privileged access to the digital realm, but we must not lose sight of the analogue, because humans experience and interact with the world as an amalgam of both.

This journey will be led by practice-based research, and research-based practice, in mutual support. Each of the following five chapters will introduce works informing and informed by the thesis, as listed in the above preface.

1.5 Original contributions to knowledge

The key contributions of this thesis are:

1. The development of *vocable synthesis*, a system for the terse description of a particular conception of timbre through the articulation of phonetic symbols informed by music tradition (§3.5).

2. The representation of music patterns in live music performance, as functions of events over time, demonstrated by Tidal (§4.5).

3. An approach to visuospatial syntax in pure functional language, based on relative distance and type-compatibility, demonstrated by Texture (§5.6).

4. A characterisation of the creative processes of the artist-programmer in relation to established theoretical frameworks (§6.3 – 6.5).

Chapter 2

Symbols

Our discussion begins in earnest with symbols, starting with discrete units of digital representation. For our purposes, a *symbol* is simply something that is used to represent, or signify, something else. At their most basic level, computers use only two discrete symbols, 1 and 0, which in combination represent other symbols, such as the letters of an alphabet. For the present discussion, there is not a great deal to say about discrete symbols in isolation. However we are not only interested in digital computers, but also the artists who work with them. Accordingly our focus through this chapter will be on comparing the internal symbol systems of humans with those of computers, looking for correspondences and differences upon which to build the higher order concerns of later chapters.

While discrete computational representations are fully observable and comparatively well understood, neuroscience is some way from providing a clear picture of the representations underlying human thought. The pervasive cognitivist approach attempts to address this by postulating a central role for discrete computation in human cognition, likening the logical operation of electronic hardware to that of the carbon wetware of our brains. We will build an alternative view based upon Dual Code theory, which admits a role of discrete symbols in cognition, but places greater emphasis on analogue symbols. This will provide a base from which we later consider the role of human and computer symbol systems in the practice of computer programming.

2.1 Situating symbols

In taking a human oriented view of computer programming, we seek to understand the relationship between digital symbols and the analogue world in which they are notated. We have already noted that a *computer* was once a job title (§1.1), since then however, computation has largely been mechanised, and perhaps to some extent dehumanised. The job of 'computer' is now taken by electronic, digital machines, bringing increased speed and accuracy by several orders. However the human has not been entirely factored out of the process; on the contrary the role of writing program(me)s for computers to follow has developed into a rather unique profession, where programmers often work in large teams writing huge tracts of code defining the underlying logic of modern institutions. Computer languages have been designed for parsimony by human programmers to support this work, allowing new approaches to human artistic expression as we will see in chapter 5. However, the underlying functional representation, of operations over sequences of discrete symbols, remains the same as when computation was a wholly human task.

In Babbage's Difference Engine (§1.1), 1s and 0s are represented by the punched/not punched states of paper cards. In modern computers, these binary states are provided by the binary on/off (or high/low) states of analogue electronic components. All discrete data may be represented within binary states, for example the true/false values of Boolean logic, or the ones and zeros of binary (base two) numbers. Such numbers may represent the instruction set of the computer processor, allowing a symbolic sequence to be interpreted as a sequence of operations over itself. This is the fundamental view of a computer program given to us by Turing (1992, orig. 1948) – a sequence of symbols, read by a machine, which interprets them as operations over that same sequence of symbols.

A discrete symbol system may represent aspects of the continuous world using a process called *analogue to digital conversion*. For example, sampling and storing sound input from a microphone is a process of averaging sound pressure over a given sample period, and approximating the average as a discrete number. The conversion cannot be perfect, as its accuracy depends on the granularity of the sample period and sample value. For example a period of one sample per 44100th of a second, with a range of 16 bits (65536 possible values) is used on standard Compact Disc recordings, and is taken to be in the same order as the distinguishing limits of human perception. We experience such a recording through inverse *digital to analogue conversion*, by sending the sampled data as stepped pulses of electricity to an electromagnet, moving the membrane in a loudspeaker to push sound pressure waves across the room, to be felt out by our ears.

Having digitally represented a sound signal, we may wish to perform some computation over it. In order to do so, some aspect of the continuous world it was sampled from is often modelled or simulated. For example to store digital sound efficiently, psychoacoustic models are often used, so that information beyond the ability of hearing is discarded. Likewise, to apply a *reverb* effect, a physical room may be modelled. It is apt to extend this concept of simulation to the function of computer hardware itself. If software can simulate a continuous

analogue	digital
real	integer
continuous	discrete
image	language
imagen	logogen
smooth	striated
amorphous	pulsating
neumatic	structural
plane	grid
articulation	sequence
nonmetric	metric
modal	amodal
grounded	ungrounded
specific	general

Table 2.1: *analogue and digital - analogous antonyms.*

domain in a discrete domain, then the job of hardware is to simulate a discrete domain in the continuous domain of analogue electronics. We construct computers to simulate a digital world, within which we may then construct a simulation of an analogue world. This simulation within simulation takes a recursive, fractal form, which may continue to arbitrary depth.

The pattern of analogue and digital representations supporting one another also runs across *human* experience. By way of illustration, table 2.1 shows pairs of related antonyms used across the arts and sciences. The antonyms *smooth* and *striated* bring to mind the smoothness of a pebble and the striated lines across weathered rock, one perceived as a continuous texture and the other as a series of discrete boundaries. But a stone can be simultaneously smooth and striated, perhaps marked by layers of limestone but washed smooth over millennia. The interdependence of the striated and smooth is described by Deleuze and Guattari (1987, p. 480) as the relation between points and lines; "in the case of the striated, the line is between two points, while in the smooth, the point is between two lines." This is a necessarily circular definition, as we understand one in relation to another.

The distinction and relationship between the analogue and digital carries through to our perception of time, including within perception of music. Boulez (1990) relates striated time with *pulsating*, regular rhythm, and smooth time with an *amorphous*, irregular flow. In discussing the notation of music, he contrasts *neumatic* and *structural* representations, where neumes are analogue lines, in contrast to the discrete structures of staff notation. Boulez tends towards the use of discrete symbols in notation, considering them more general and accurate, but of course discrete symbols are only more accurate if you wish to notate a pulsating rather than amorphous temporal structure in the first place.[1] The Western Classical tradition focuses on the former, but amorphous time is nonetheless present as an important aspect of dynamic performance, even when not notated. There are however strategies for notating amorphous time in discrete computer language, exemplified by the constraint-based time setting used in the Bol Processor language and inspired by Indian Classical music (Bel, 2001).

The distinctions in table 2.1 between *articulation* and *sequence*, and *modal* and *amodal* are different aspects of the distinction between *grounded* and *ungrounded*. Our environment, and our bodies moving through it, are by nature analogue, but our experience of our environment is both analogue and discrete. We may perceive a movement as a smooth articulation, while simultaneously abstracting it into a discrete sequence of events, by segmenting it at perceived points of discontinuity. Where we abstract it, we evoke a representation that is to an extent *amodal*, freed from the qualities of a particular mode or sense. However 'ungrounded' is perhaps too strong a term; just because we name a hue as *red*, it does not mean that we have lost all connection with perception; we have simply gone from a particular experience in colour space, to a discrete value that symbolises a region of possible experiences in colour space. We will expand on perceptual and conceptual spaces later in this chapter (§2.2.5).

In summary then, discrete and analogue domains are distinct, but hosted within one another. They also interconnect, supporting and enriching one another. This is of great importance to the programming interface between humans and computers, profoundly so when we consider how the analogue/digital interconnection extends into the mind of the human programmer.

2.2 Symbols in cognition

From a computer science perspective, digital symbol systems are shown to be remarkably elegant and with enormous practical use, so it is unsurprising that they are often used as a descriptive model for the basis of complex phenomena, such as in biology and the cognitive sciences. Wolfram (2002) shows that it is surprisingly likely for general computation to emerge from what would otherwise seem to be trivial rule systems. In particular, his exploration of one dimensional cellular automata led to the discovery of *Rule 110*. This rule is one of the family where a cell's state is based only on its preceding state and that of its two immediate neighbours. It has been proved that despite this simplicity, Rule 110 is a universal computer in that it can carry out all the operations of the Turing machine, and therefore any other digital computer (Turing, 1992, orig. 1948). Wolfram takes his observation that computation falls out of

[1]Accuracy need not be the goal when notating amorphous structures however. The use of discrete symbols to denote *classes* of amorphous structure is of course possible, and is often the most satisfactory choice.

such trivial interactions as evidence that the workings of nature are computational. We are interested in a more specific hypothesis than Wolfram; are digital symbols the representational basis of cognition? As computation arises from the simplest of rule sets, we certainly cannot discount it on the basis of Occam's razor.

Great works on the Language of Thought by Fodor (1980) and Pylyshyn (2007) do indeed point towards a discrete representational basis for cognition. Cognition is described by these thinkers as the subconscious workings within an innate language often dubbed *mentalese*. This is not a natural language as we speak it, but an internal, universal language structuring subconscious thought. This presumed language is represented using discrete symbols, with cognition characterised as a process of computational operations over those symbols. Such computational accounts have strong traction across the cognitive sciences, and also support understanding in computer science; we understand the syntax of computer languages with reference to Chomskian transformational grammar (§4.1).

We return however to the notion of digital supported by analogue, and analogue by digital. To focus solely on discrete computation is in denial of its essential interplay with analogue movement and shape. In comparison with humans, digital computers appear to lead a rather impoverished existence in terms of engagement with their analogue environment. Could this difference be due to differing symbolic foundations? There is a line of thought which allows us to admit computational accounts of the mind, but in addition consider a second, complementary system of representation. This second system is *mental imagery*; an analogue symbol system grounded in perception.

2.2.1 Mental imagery

When presented with certain kinds of problems, a quasi-perceptual *mental image* may be consciously manipulated to solve a task. We can examine these subjective experiences through objective experiment, for example Shepard and Metzler (1971) identified that when matching rotated objects, subject reaction times have a strong, linear correlation with relative degree of rotation. This suggests that imagery is being rotated in the mind. It would appear that these quasi-perceptual states are an analogue system of symbolic representation, used here in cognitive problem solving.

The *image* in mental imagery is not specific to vision, but a broad term related to any quasi-perceptual state. For example the use of prosodic intonation in speech is *paralinguistic*, not entirely notated in written text, yet symbolising meaningful content. Indeed the meaning of a spoken text can be twisted, perhaps negated with sarcasm, through subtle paralinguistic phrasing. Furthermore, intonation is not just a concern in communication with others, the phenomenon of *inner speech* while reading silently is understood to be a function supporting working memory, providing analogue, prosodic cues useful in comprehending text (Rayner and Pollatsek, 1994, p. 216).

Mental images then, are analogue symbols underlying visuospatial cognition. An image symbolises an object, using properties mapped directly from perception. For mental rotation tasks, the symbol may itself be rotated, as it shares essential geometric features with the object it symbolises. Some higher order tasks, such as the analysis of complex symmetries, may however be beyond the capabilities of mental imagery. For such tasks we may abstract properties of imagery to create a formal language, by which we mean a system of discrete symbols governed by grammatical rules. In the case of mental rotation, we identify *group theory* as a linguistic counterpart, developed to gain mathematical understanding of symmetries (du Sautoy, 2008). Group theory allows us to extend our understanding of symmetry with discrete logic, but does so in relation to the mental images we experience.

Empirical understanding of the psychology of analogue and discrete symbols is provided by Paivio (1990) through his *Dual Coding* theory. His contention is not that there are two codes, but rather that there is a hierarchy of codes, which branch at the top into discrete linguistic codes and continuous perceptual codes, which Paivio names *logogens* and *imagens* respectively. This split is shown in their concurrent processing; humans are able to comprehend language while simultaneously attending to imagery. Continuing with our earlier example of phrasing in speech, humans find it easy to simultaneously process and integrate prosodic and linguistic information, but rather more difficult to simultaneously read text and listen to speech. The explanation offered by Dual Coding theory is that there are distinct, yet integrated symbol systems for imagery and language. This theory sits well against the background of digital and analogue interdependence we noted in §2.1.

Neuropsychology provides support for Dual Coding through research into the fundamental brain structure of the two distinct hemispheres. Very broadly speaking, the left hemisphere is specialised for language, and the right for visuospatial tasks (Martin, 2006, pp. 128-129). We must be careful however not to over-simplify this relationship: the more closely it is examined, the less clear it gets. For example, from a meta-analysis of cerebral lateralisation of spatial abilities (Vogel, 2003) we see there is a strong sex difference in lateralisation of spatial tasks, where female subjects tend to show no hemisphere dominance and males tend to exhibit strong right hemisphere dominance. The same meta-analysis finds subjects classed as 'high imagers' also show no hemisphere dominance in visuospatial tasks, suggesting that high imagers are not those with a dominant 'visual' right hemisphere, but rather with high integration between both hemispheres. This latter finding has statistical significance, but is based on few studies and so cannot be considered robust. However while it stands, it supports the view that the usefulness of mental imagery comes from integration between discrete and analogue codes.

2.2.2 Mental Imagery and Programming

Before we go any further, we should address Dual Coding theory to our overall theme. It may seem that computer programming is an overwhelmingly linguistic task in a wholly discrete domain, but if we focus on the programmer rather than the computer, we find this is not the case. Programmers may deal with formal, discrete and textual language, but they support and structure their work in a variety of ways external to computer language itself. For example diagrammatic representations of code structure are widespread in the teaching and practice of software development, such as those standardised in the Structured Systems Analysis and Design Method (SSADM) and Unified Modelling Language (UML). These are highly formalised languages, but use spatial arrangement and connecting lines to present the structure of a program in a visual manner. Further, the structure of programs is often introduced in programming textbooks using visual metaphor, for example as interconnected roads (Griffiths and Barry, 2009, pp. 13–21) . We assert then that despite the discrete nature of computation, programmers often use mental imagery to support their work.

Petre and Blackwell (1999) look for, and find reports of mental imagery during programming tasks. They conducted verbal interviews with ten expert programmers, while they were in the process of solving programming tasks, prompting them to explain what they were 'seeing' in visual or auditory terms.[2] We highlight and comment on a few reports from these interviews:

> … no place holders, no pictures, no representation … just the notion, the symbol entities, semantic entities and the linguistic token … atomic notions. They just 'are' (Petre and Blackwell, 1999, p. 17)

We contend that this is not a report of mental imagery as Petre and Blackwell (1999) imply. This is imaginative use of language, rather than modality-specific quasi-perceptual states.

> … it moves in my head … like dancing symbols … I can see the strings [of symbols] assemble and transform, like luminous characters suspended behind my eyelids … (Petre and Blackwell, 1999, p. 14)

This is again a report featuring discrete symbols, but augmented with visual imagery. This appears to be simultaneous activation of both language and imagery, in line with Dual Coding theory.

[2]To counter known problems with interview protocols, the study was also complemented and to an extent confirmed by a second study, based on an undirected questionnaire of 227 programmers of the LABView visual programming environment.

> It buzzes … there are things I know by the sounds, by the textures of sound or the loudness … it's like I hear the glitches, or I hear the bits that aren't worked out yet … (Petre and Blackwell, 1999, p. 15)

This is an intriguing example of a reported sonic image, again highlighting that mental imagery is modality specific, but not necessarily of the visual sense. Indeed the previous quote could be interpreted as simultaneously having both kinaesthetic and visual features. Petre and Blackwell (1999) report that all ten experts reported sound as an element in their imagery, although not in general as a typical element. It is worth noting however that subjects were specifically probed for sonic imagery, which may have influenced the programmers' mental imagery and reports.

> It's like driving across a desert looking for a well. What you actually have is good solutions distributed across this desert like small deep wells and your optimizer trundling along looking for them… (Petre and Blackwell, 1999, p. 17)

This image appears to be visual-specific although could in addition be interpreted in a kinaesthetic sense. That the programmer reports imagining a three-dimensional problem space is of relevance to the theory of Conceptual Spaces, which we will examine later in §2.2.5.

Again, it is wise to be cautious of introspective reports. Introspection is a subject of research in its own right, with one theory being that introspections are nothing other than reconstructed, 'dramatised' perceptions (Lyons, 1986). Nonetheless this should not discourage us from treating mental imagery seriously as a system for symbolic reference, and these reports give us cautious vantage over the cognitive processes of computer programming, towards understanding how programmers may use mental imagery to support their work.

2.2.3 Dual Coding in Source Code

At base, the source code for a computer program is a one dimensional sequence of discrete symbols, ready for interpretation by a digital computer. How then could this relate to the Dual Code in human cognition, when we assert in the previous section that programmers employ mental imagery in their work? In answer we find that despite the discrete underlying representation, imagery is present both in the perception and organisation of source code.

In the Psychology of Programming field, a discipline bridging Psychology and Human-Computer Interaction, it is widely understood that there are notational features of source code besides formal discrete syntax, and further that these aspects are vitally important to human understanding of programs. Such features are known as *secondary notation*, which includes spatial layout, comments, colour highlighting and variable names. Secondary notation, and its place within the Cognitive Dimensions of Notations framework will be further expanded upon in §5.1, but for now we focus upon the use of spatial layout in code, with respect to mental imagery. Consider the following code fragment written in the C programming language (Kernighan and Ritchie, 1988):

```
if (condition == true) {
    display("Welcome.");
    beep();
}
```

All spaces in the above example could be discarded, giving this:

```
if(condition==true){display("Welcome.");beep();}
```

As far as a C language compiler is concerned, these code fragments are equivalent, as all whitespace is discarded during tokenisation, an early stage of computer language parsing. For a human however, the former version of the code is much easier to comprehend, and for a much bigger program, the latter form would be close to impossible to understand on sight alone. The computer 'sees' a one dimensional string of symbols, parsed into the higher order data structure of a syntax tree. Humans however are able to perceive and navigate the fragments and wholes of a code structure in a manner analogous to a visual Euler diagram. By way of illustration, our code becomes even more difficult to read if we replace characters to remove the visual prompts of containment given by the symmetry of matching brackets:

```
if$condition==true@^display$"Welcome."@;beep$@;&
```

From this we argue that while both humans and computers understand programs as discrete symbols, humans use secondary notation to augment this representation with visual imagery. As such, source code is an amalgam of two symbol systems, one of which is discarded in the computational parsing process.

Programming languages with particularly visual notations are known as *Visual Programming Languages.* Programming has a reputation for being rather difficult to teach and learn (e.g. Lister et al., 2004), and so a recurring theme in language design is finding forms of notation that are more 'natural' or 'intuitive'. A particular hope is that Visual Programming will lead to languages grasped easily by end users with little or no training. Early enthusiasm has however not led to wide success, on the whole VPL has only taken hold for certain target domains, for example LABView in engineering and Patcher languages (§5.3.1) in audio/visual digital signal processing. The lack of success of general purpose Visual Programming suggests that relationships between discrete symbols and mental imagery are by nature particular to the task at hand. We should look then not for ways of adding 'artificial' visual metaphor to programming as has been unsuccessful in HCI in general (Blackwell, 2006e), but for ways of supporting a programmer's own system of imagery, integrated with the text of source code. We will revisit and expand upon this point in our view of the notation of programs in §5.3.

We view source code then as supporting both language and imagery. Use of discrete symbols expressed within grammar rules are linguistic[3], but are arranged to allow the support of visuospatial cognition in understanding and writing programs.

2.2.4 Language and Situated Simulation

Language and Situated Simulation (LASS) theory is a recent development of Dual Coding theory introduced by Barsalou et al. (2008). LASS adds empirical support broadly in agreement with Dual Coding, but with different detail and a strong change of emphasis. Whereas Paivio (1990) deals with imagens and logogens even handedly, Barsalou et al. place far greater emphasis on imagery, which they refer to as the *simulation* system, reflecting its active role in cognition. According to LASS, the linguistic system is *superficial*, acting as little more than a control mechanism for the simulation system, which is responsible for deep conceptual structure grounded in experience.

Simulations in LASS are described as *perceptual symbols*, but this does not imply conscious awareness of their use. Indeed in a break from Dual Coding theory, Barsalou (1999, §2.1.1) asserts that perceptual symbols are a neural representation with only limited correlates in consciousness. This proposal is similar to mentalese, in that it is an internal representation underlying subconscious cognition. The difference is that perceptual symbols are analogue and highly modal, being grounded in sensory-motor neural systems, whereas the notion of mentalese is as discrete and amodal.

The programmers' reports from research reviewed in §2.2.2 were the result of probes for imagery in consciousness while programming, in keeping with the focus on conscious states by Paivio (1990). However we share the view of LASS, that analogue representations extend beyond the confines of consciousness and attention, to structure the bulk of *unconscious* thought. Therefore if anything, these reported experiences of mental imagery are only the tip of the iceberg.

Table 2.2: *Levels of representation according to the theory of Conceptual Spaces (Gärdenfors, 2000), aligned with terms from Dual Coding theory (Paivio, 1990)*

Experience	Gärdenfors	Paivio	Structure
Language	Symbolic	Logogens	Discrete symbols
Perception	Conceptual	Imagens	Low dimensional geometry
Sensation	Sub-conceptual	–	High dimensional neural nets

2.2.5 Conceptual Spaces

We have discussed perception with respect to analogue symbols at length, but how does this relate to concepts? In a major review of the field of concept theory, Murphy (2002, p. 5) defines the term *concept* as "a mental representation of a class of things". So concepts allow us to structure our experiences through generalisation, but how is this done? The predominant view is that perception and cognition are independent functions of the brain, but an alternative view holds that functions of perception and concepts are integrated, relying upon shared neural resources.

How then could concepts, abstractions from the world, be structured in the same way as perception? Firstly it is important to recognise that perception is itself not a straightforward reflection of the world, rather a low-dimensional representation of high-dimensional sensory input, giving us a somewhat coherent, spatial view of our environment. By *spatial*, we do not only mean in terms of physical objects or visual perception, but rather in terms of features in the analogue spaces of all possible tastes, sounds, tactile textures and so on. This scene is built through a process of dimension reduction from tens of thousands of chemo-, photo-, mechano- and thermoreceptor signals. Gärdenfors (2000) proposes that this process of dimension reduction is behind the construction of *Conceptual Spaces*. Moreover, he takes this as the primary model of conceptual representation, including that of higher-level concepts somewhat abstract from perception. That is, Conceptual Spaces are grounded in the cognitive resources of mental imagery, but are repurposed to represent conceptual relationships in geometric spaces.

The theory of Conceptual Spaces has great explanatory power in resolving conflict in the field of artificial intelligence (AI) between proponents of the high dimensional, statistical graphs of artificial neural networks (ANNs) and the discrete symbols of good old-fashioned artificial intelligence (GOFAI). ANNs represent concepts through trained networks of connections between cells, and GOFAI through the discrete constructs of language. In other words, ANNs work in the realm of sensation, and GOFAI in the realm of computation. Gärdenfors seeks to unite these approaches by identifying a level of representation that mediates between

[3]See §4.1 for discussion of the relationship between computer and natural language.

them: the low dimensional, geometric realm of the conceptual level.

Table 2.2 illustrates how the three levels of Conceptual Space theory align with the two channels of Dual Coding. Note some discrepancies in terminology; Gärdenfors refers to linguistic representation as *symbolic* whereas we use the term *discrete symbolic* to distinguish from analogue symbols in Dual Coding theory. Note also that Gärdenfors describes ANNs as *sub-conceptual* in place of the more generally used terms *non-* or *sub-symbolic*, to properly situate them beneath the geometry of the conceptual layer.

Conceptual Space theory has some agreement with LASS, in that conceptual representation is considered to be largely outside the grasp of conscious introspection. Although these theories use different terms and have different emphases, they agree that the primary form of conceptual representation is within spaces, rather than within the syntax of the discrete symbolic layer. However whereas LASS focuses on simulations of physical objects, movements and interactions, Conceptual Space theory focuses more on the geometric structure of mental spaces.

Conceptual spaces are structured by *similarity*; concepts that are closer together are more alike. Furthermore, the dimensions of conceptual spaces are aligned with particular conceptual qualities. This is most easily explained with the example of colour space, where more similar colours are closer, within the quality dimensions of hue, chromaticism and brightness. As already alluded to in §2.1, the concept *red* is not a point in colour space, but rather a *convex region*, although we may pinpoint a *prototypical* red as the centroid, or perhaps more holistically as the Voronoi generator (Okabe et al., 2000) of its region. The significance of convexity is clear if one considers that for any two hues of red, all the hues along the path between them will also be red. *Red* as a property is a concept in its own right, but can also form part of a more complex cross-domain concept, such as *red ball*.

The example of colour is straightforward, but it is rather harder to apply the theory to concepts without clear correlates to perceptual spaces in consciousness. However the theory of Conceptual Spaces makes the bold claim that the same processes of dimension reduction and spatial representation are applied to the majority of concepts. Despite the lack of subjective conscious experience, we can hope to objectively identify the dimensions of higher order concepts through psychological experiment, such as those based on multi-dimensional scaling (MDS). In classic MDS experiments, human subjects are asked to grade pairs of stimuli by similarity, with their judgements interpreted as distance measurements, and then used to reconstruct the conceptual space evoked by the stimuli. We will appraise MDS approaches in the context of musical timbre in §3.3.1. We may also construct conceptual spaces based on established theory: for example Forth et al. (2010) identify dimensions of musical metre based upon music theory.

The theory of Conceptual Spaces is particularly useful for our present discussion in providing an account of how linguistic and spatial representations may interact in cognition. Rather than placing the entire burden of higher-order representation on discrete, linguistic structure, it instead places emphasis on metaphor, both for relating spaces together and for grounding higher order concepts in embodied, perceptual spaces. Conceptual domains are low dimensional, in general having from one to three dimensions, in common with mental imagery and human perception in general. Thus, they may be explored and mapped through spatial reasoning, as if they were physical spaces. This includes relating features of two spaces together, in other words drawing metaphorical relations between conceptual domains.

2.2.6 Metaphor

Metaphor is often considered to be a form of poetic wordplay, a window dressing on language. In introducing their Conceptual Metaphor theory, Lakoff and Johnson (1980) argue on the contrary that metaphor is of central importance to language. Conceptual Metaphor theory is a defining contribution to the field of cognitive linguistics, and places metaphor in the role of structuring concepts relative to one another within a coherent system of meaning. For example, Lakoff and Johnson claim that "Well, that *boosted* my spirits!" is a linguistic phrase structured by the underlying conceptual metaphor HAPPY IS UP[4]. A single conceptual metaphor may be expressed through many linguistic phrases, for example "I am *depressed*", "I am feeling *down*" and "My spirits *sank*" are instantiations of the same HAPPY IS UP metaphor.

Lakoff and Johnson (1980) divide conceptual metaphors into two types; *structural* metaphors such as TIME IS MONEY (e.g. "He is living on *borrowed time*") and ARGUMENT IS WAR (e.g. "His claims are *indefensible*"), and *orientational* metaphors such as CONSCIOUSNESS IS UP (e.g. "He is *under* hypnosis") and the aforementioned HAPPY IS UP. Structural metaphors structure one concept in terms of an other, while orientational metaphors structure a whole system of concepts relative to fundamental directional and spatial relationships. Lakoff and Johnston assert that the majority of conceptual metaphors are orientational, coherent within a large system of metaphors (Lakoff and Johnson, 1980, p. 17). Orientational metaphors are organised relative to the body, often with physical motivation, for example HAPPY IS UP relates to the erect posture of a happy person versus drooping posture of a depressed person.

Gärdenfors (2000) builds the notion of orientational conceptual metaphors into his theory of conceptual spaces (§2.2.5), asserting, like Lakoff and Johnston, that they form the pri-

[4]Conceptual metaphors are by convention denoted with small block capitals, to differentiate them from the linguistic metaphorical phrases which refer to them.

mary structure of a human conceptual system. Orientational metaphors ground Conceptual Metaphor theory, and therefore the theory of Conceptual Spaces, in human physical and cultural experience. Gärdenfors proposes that these metaphors, and the geometric spaces they relate together, are the basis of semantics. We will return to this subject in discussion of music and language in §4.2.

2.3 Anthropomorphism and Metaphor in Programming

Metaphor appears to permeate our understanding of programming, as evident in the varied reports of mental imagery in programming tasks (§2.2.2). Perhaps this is due to the abstract nature of computer language syntax, requiring metaphorical constructs to ground programming language in everyday reasoning. Blackwell (2006d) used techniques from corpus linguistics on programming language documentation in order to investigate the conceptual systems of programmers, identifying a number of conceptual metaphors listed in Table 2.3. Rather than finding metaphors supporting a mechanical, mathematical or logical approach as you might expect, components were instead described as actors with beliefs and intentions, being social entities acting as proxies for their developers.

Blackwell (2006d) classifies orientational metaphors into the metaphors Programs operate in a spatial world with containment and extent, along with the related metaphor of movement in spaces, Execution is a journey in some landscape. These metaphors are reported to occur regularly, and reflecting on Conceptual Metaphor theory, we would expect orientational metaphors to provide the primary structure of programming concepts. Perhaps these metaphors could be broken down and related in a coherent, spatial system of metaphors such as Abstraction is Up and Progress is Forward. A preliminary examination of the corpus indicates that this may be feasible, however further work is required.

It would seem then that programmers understand the structure and operation of their programs by metaphorical relation to their experience as a human. The notion of including a computer in a creative process (§6.3) is by nature anthropomorphic; by embedding the development of an algorithm in a human creative process, the algorithm itself becomes a human expression. However, Dijkstra strongly opposed such anthropomorphic approaches in computer science:

> "I have now encountered programs wanting things, knowing things, expecting things, believing things, etc., and each time that gave rise to avoidable confusions. The analogy that underlies this personification is so shallow that it is not only misleading but also paralyzing." (Dijkstra, 1988, p. 22)

Dijkstra's claim is that by focusing on the operation of algorithms, the programmer submits to a combinatorial explosion of possibilities for how a program might run; not every case can be

Components are agents of action in a causal universe.
Programs operate in historical time.
Program state can be measured in quantitative terms.
Components are members of a society.
Components own and trade data.
Components are subject to legal constraints.
Method calls are speech acts.
Components have communicative intent.
A component has beliefs and intentions.
Components observe and seek information in the execution environment.
Components are subject to moral and aesthetic judgement.
Programs operate in a spatial world with containment and extent.
Execution is a journey in some landscape.
Program logic is a physical structure, with material properties and subject to decay.
Data is a substance that flows and is stored.
Technical relationships are violent encounters.
Programs can author texts.
Programs can construct displays.
Data is a genetic, metabolizing lifeform with body parts.
Software tasks and behaviour are delegated by automaticity.
Software exists in a cultural/historical context.
Software components are social proxies for their authors.

Table 2.3: *Conceptual metaphors derived from analysis of Java library documentation by Blackwell (2006d). Program components are described metaphorically as actors with beliefs and intentions, rather than mechanical imperative or mathematical declarative models.*

covered, so programmer errors prevail. He argues for a strict, declarative approach to computer science and programming in general. Dijkstra views computer programming as such a radical activity that we should not associate it with our daily existence, or else limit its development and produce bad software.

The alternative view presented here is that metaphor necessarily structures our understanding of computation, as it provides the basic structure of our conceptual system (§2.2.6). Software now permeates Western society, and is required to function reliably according to human perception of time and environment. Metaphors of software as human activity are therefore becoming ever more relevant.

2.4 Synaesthesia

Synaesthesia is a condition whereby modes of perception are *cross-activated*, where activation in one modality stimulates experiences in a second modality. A classic example is of the colour→taste synaesthete, who experiences a particular taste whenever they see a particular colour. Such activations are uni-directional – a colour→taste synaesthete will generally not also be a taste→colour synaesthete. Synaesthetic experience differs from that of conceptual metaphor, in that for a colour→taste synaesthete, red may *literally* taste of earwax, as opposed to a temperature-colour metaphor aligning the spatial dimensions of redness with warmth, by linguistic reference to orientational metaphor.

Historically synaesthesia has not been taken particularly seriously by psychologists, dismissed as an insignificant function of memory, metaphor or illicit drug use (Ramachandran and Hubbard, 2001b, p. 4). Over the last decade however a number of experiments have confirmed that synaesthesia is a real phenomenon, showing for example that number→colour synaesthetes perform better than controls at certain identification tasks, aided by perceptual 'pop-out' provided by their condition (Ramachandran and Hubbard, 2001a). This clear experimental support for synaesthesia as a medical condition emboldened Ramachandran and Hubbard (2001b) to speculate on a role for synaesthesia in the evolution of language, connecting vocal articulations and sounds with concepts, and providing a neural basis for metaphor. In the same paper, a causal link is claimed for the high incidence of synaesthesia reported in artists, where synaesthetes are more able to make creative metaphorical connections. These claims are tempered by a review by Ward et al. (2008), showing that evidence surrounding artists and synaesthesia is unreliable, with the additional observation that the condition is automatic and inflexible, whereas creative metaphor requires divergent thinking of a qualitatively different nature. However the automatic cross-activation of synaesthetes is at least a metaphor for the optional cross-domain metaphors drawn by artists, a meta-metaphor perhaps.

While the word *synaesthesia* is generally reserved for abnormal phenomena, there are many cross-modal 'illusionary' experiences that are experienced by the majority of the population. Sound symbolism does not sit well with contemporary linguistic theory, but certain effects are undeniable. In his treatise on Gestalt Psychology, Kohler (1930) notes that subjects readily associate spiked and loopy doodles with the nonsense words *takete* and *maluma* respectively, an observation repeated in formal experiment by Ramachandran and Hubbard (2001b) with the words *kiki* and *bouba.* Another illusion is where a single flash of light, when accompanied by a double sonic click, is perceived as a double flash of light, a phenomenon confirmed by fMRI evidence (Zhang and Chen, 2006). A third example is the McGurk-McDonald effect (§3.1), which demonstrates the influence of lip reading on auditory perception. These strong illusions provide ground where artists may play with the senses of their audience.

2.5 Artistic synaesthesia?

In the arts, synaesthesia is frequently alluded to where a work crosses multiple media. This is not intended to be an automatic process as with clinical synaesthesia, but the extension of an artistic theme across modalities, in order to create a rich experience. At times this may seem a mundane aspiration dressed in the clothes of psychological disorder. It is after all normal to be able to both see and hear an action in perceptual unity; modes of perception are by nature integrated. Interest comes however where technology allows modalities to be related in a novel manner.

The phrase *algorithmic synaesthesia* is coined by Dean et al. (2006) to describe artistic attempts to connect vision and sound (or more specifically, film and music) using digital computers. The word *algorithm* is used rather loosely, to indicate any use of computers to connect modalities, even if this only amounts to sharing of data between outputs; artistic license is applied to both terms of the algorithmic-synaesthetic juncture.

Dean et al. (2006) report several interesting approaches to cross-modal integration in performance. However we treat with scepticism their claim that the addition of a computer necessarily brings mixed media into a new realm. They state that their new digital 'algorithmic synaesthesia' is more precise than past analogue efforts, but it is not clear why; if we are concerned with art that plays with analogue perception, then surely an analogue approach would notionally be more precise. Instead we see computational developments as a continuation of an ever-present tendency for artists to look for new correspondences between perceptual domains, through both analogue and digital means.

The use of mixed media in the arts is not new, and technology has been employed for centuries in finding novel ways to connect the senses. For example the inventor Mary Hallock-Greenewalt developed a colour organ (as in, a keyboard instrument to 'play' colour projections) in a bid to realise her conception of *Nourathar*, an artform based on fluctuating transpositions of brightness and hue, with a scoring system to unite it with musical staff notation (Greenewalt, 1946). Many colour organists, Hallock-Greenewalt included, have presumed their colour organs to be novel, however experiments in this area have been a longstanding theme before and since. Light shows and video displays have become an integral part of the live music stage, from the psychedelic light shows of the 1960s to contemporary live generative visual art. The latter is exemplified by United Visual Artists (UVA), who began by creating stage visuals to accompany and react to the live music of Massive Attack in 2003. UVA have since brought their work to national galleries in the form of interactive installation art. The artistic focus of both the work of Hallock-Greenewalt and UVA is the fusing of modes into an whole, audio-visual experience.

Hallock-Greenewalt invented analogue electric components to support her work, and UVA work with digital computation, but the artist John Whitney is a case of an artist who has worked through both analogue and digital technology. Whitney produced experimental films from the 1950s, first using hand drawn animation, then analogue computers and in later years digital computers as they became available (Youngblood, 1970, pp. 207–228). There is little discontinuity between his works as he progresses from analogue to digital methods; all are explorations of form through geometry. The biggest difference is in the amount of time taken to produce the films, a decade for his first hand drawn film, but much less for his later works. The depth of his first digitally produced film *Arabesque* (1975) is breathtaking, the influence of his earlier analogue approaches still showing in leading the viewer's perception to an experience that is both abstract and coherent. This continuity is shown too in the development of programming languages for music; in particular Miller Puckette has said he thinks of his Patcher languages (§5.3.1) as more like analogue synthesisers than programming languages (Lewis, 1993). We will have much more to say about analogue representation in the notation of computer programs in chapter 5.

Digital computers too often lead artists into a trap of applying digital transformations to data without consideration of the analogue source and destination of the data, ultimately leading to output that human perception cannot decode. While this is not always the case, analogue methods tend towards straightforward mapping between two quality dimensions in a straightforward, perceptually salient manner. Digital methods on the other hand can too easily result in transformations divorced from human perceptual processing, such as reading an image as a one dimensional scan-line, as Dean et al. (2006) report of the MetaSynth software. The result is too often perceptually incomprehensible. Nonetheless at particular scales humans have strong acuity for recognising discrete patterns, and indeed we will assert that this is an important aspect of music cognition in §4.4.

The work of Whitney shows how the analogue and digital aspects of a work support one another, and also that while electronic computers are often extremely convenient, they are not necessary. There are dangers in working with electronic computers, in that by not taking full consideration of orientational metaphor, or coherent pattern, the results of digital transformation are likely to be incomprehensible. However we should certainly not downplay the potential importance of computer languages in a rich creative process, a subject we will explore in chapter 6.

2.6 Acid Sketching – Semantic Imagery

We have discussed two flavours of symbols in Dual Coding theory, analogue imagens and discrete logogens (§2.2.1), as important to understanding human perception and cognition. We will develop an argument for the relevance of this to programming languages in later chapters, for now however we ground the discussion so far in practice, with a working prototype that demonstrates the use of symbolic, analogue imagery in a user interface for music. A video of Acid Sketching in use is contained within the accompanying DVD.

The *Acid Sketching* system was developed through the present research, to understand how geometric forms and relationships can be meaningfully used in a computer system. The Acid Sketching interface consists of an ordinary piece of paper, which is also a projection surface. When shapes are drawn on the paper with an ink pen, they are identified and analysed using computer vision. Their shapes are translated to sound synthesis parameters, and their relative positions translated into a polyphonic sequence.

The procedure to turn shapes into a sequence of sound events is as follows. Shapes are identified from a digitised image of the paper via a consumer grade webcam. This is done using the OpenCV (Open source Computer Vision) library developed by Intel Corp. Using this library, the contours of the hand drawn lines are identified, and the shapes which they describe are enumerated. The centroids of the shapes are calculated, and a minimum spanning tree connecting them is constructed. The result is the graph containing exactly one path between any two shapes, whose spanning traversal is minimised. Starting from the most central shape, the minimum spanning tree is followed, to place the shapes in a sequence. Time intervals between each pair of shapes are given by the length of the edge connecting their centroids, which are scaled relative to an adjustable beats-per-minute value. Because the minimum spanning tree branches, events may co-occur, which in musical terms results in polyphony.

This use of a minimum spanning tree turns a visual arrangement into a linear sequence of events, a kind of dimension reduction. In terms of computational complexity, a simpler approach would have been to simply disregard one dimension, for example by reading the events from left to right. However we argue that greater richness is achieved by using this graph structure built from the perceptually salient measure of relative distance in 2D space. After all when we view a picture, our eyes do not generally read from left to right, but instead jump around multiple fixation points influenced by the structure of the scene (Henderson, 2003).

Sound synthesis is provided by *nekobee*, a free/open source emulator of the analogue Roland TB-303 Bass Line synthesiser. The nekobee software is no longer actively maintained, but continues to be available in Linux operating system distributions. The TB-303 is best known for its use in the *Acid House* genre, hence the name *Acid Sketching* (we discuss perception of synthesis in acid house music in §3.3.3). The nature of each sound event is given by morphological measurements of its corresponding shape, where each measurement is mapped to sound synthesis parameters. Specifically, *roundness* is calculated as the ratio of a shape's perimeter length to its area, and maps to envelope modulation; *angle* is that of the shape's central axis relative to the scene, and maps to resonance; and finally, the shape's *area* maps to pitch, with larger shapes giving lower pitched sounds.

Visual feedback is projected back on to the paper using a standard data projector, with the camera input aligned to projector output in software. This feedback takes the form of moving circles, tracing the path from one shape to the next along the edges of the minimum spanning tree, flood-filling each shape as its corresponding sound event is triggered.[5]

While software based, Acid Sketching aims to avoid some of the traps of digital representation. One such trap is placed by the digital representation of an image, generally a two dimensional array with x and y as indices, or the underlying single dimensional representation of a scan-line. In Acid Sketching, what is significant is placement not relative to two dimensions, but relative to all the other marks on the page. Further, the geometric calculations used in this Acid Sketching prototype are not formally tested, but illustrates our hypothesis that correspondence between shape and timbre are straightforwardly learnable. If this hypothesis holds, this prototype system could be developed further into an engaging interface for live music. Work inspired by Acid Sketching is underway at the Computer Laboratory at Cambridge University, investigating such correspondences from the perspective of human-computer interaction.

The symbolic nature of the shapes in Acid Sketching gives particular insight to our overall theme. It demonstrates a use of analogue symbols which have morphological properties con-

[5]For the purposes of demonstration, the video for Acid Sketching on the DVD shows this feedback mixed directly with recorded video, rather than projected into the scene.

tinuously mapped from those of what is represented. A criticism may be that these symbols are not truly analogue as they are represented digitally; from the perspective of the computer, we can think of there being many millions of discrete sounds to represent, and potentially at least the same number of discrete symbols to represent them. It is more useful however to think of these as analogue symbols with a continuous, perceptually salient mapping to the represented sounds. At a particular scale the underlying granular, discrete representation begins to show, but as with atomic units in physics, this is outside of the normal limits of human perception. The simulation is accurate enough to be perceived as analogue.

Acid Sketching demonstrates how analogue symbols may be interpreted through the use of existing computer vision libraries. We can think of these libraries as models of perception, which while impoverished compared to human perception, are nonetheless close enough to be useful in interface design. While creating these perceptually salient continuous mappings is relatively trivial, the challenge comes when we try to integrate analogue and discrete symbols in a mutually supporting manner. How may we relate abstract, discrete symbols with perceptually grounded, continuous symbols? For one answer, we look to the vocal tract.

2.7 Phonemes

For most people, the instrument that most directly grounds symbols is the vocal tract, with paralinguistic support from their hands. For Deaf people, it is the other way around, with hand gestures providing the 'atoms' of language and the mouth and face adding phrasing. The expressive equivalence of signed and spoken languages (Sutton-Spence and Woll, 1999) demonstrates dependence of language on movement rather than sound, a point we will return to in §3.1.

In spoken form, language is represented within the discrete symbols of phonemes, the units which we represent with sound. The phonetics of English has a comparatively poor correspondence to the letters of its alphabet, and so to unambiguously transcribe English pronunciation, the International Phonetic Alphabet (IPA, see Ladefoged, 1990) is generally used. The IPA includes 107 letters and 52 diacritics, split into two tables, one for consonants and one for vowels. Both tables are organised according to place of articulation, and so we can say that these symbols use physical positions to classify sounds. That is, phonetics grounds speech not in sound images, but in bodily movement. This is an important distinction, which we mark now to be built upon later through chapter 3.

A phoneme exists in three modalities: as a discrete symbol, as a configuration of the vocal tract, and as a sound. Phonetics binds these very different modalities together into a whole, allowing us to communicate discrete structures of language through movement, whether its the hand that writes or signs, or voice that sounds. The following section introduces an artwork that brings the role of movement in communication and sound perception to the fore.

2.8 Microphone

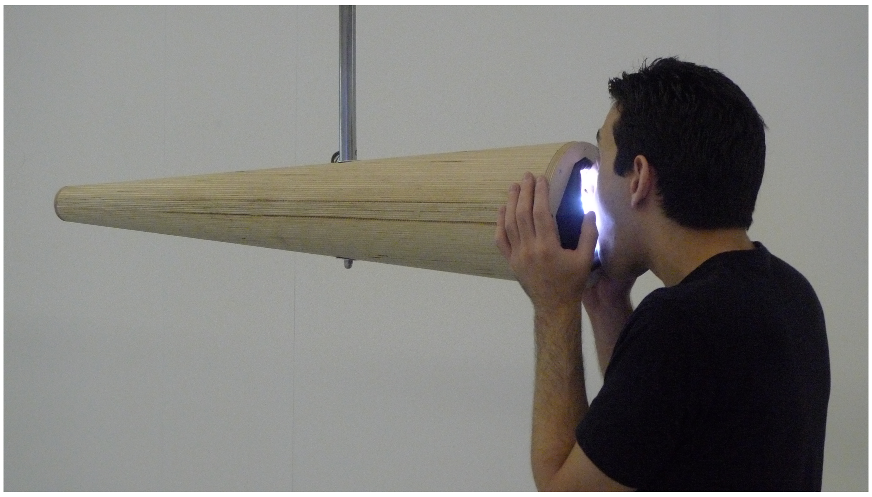


Figure 2.1: *Microphone, by Communications, 2010. Media: CNC milled plywood, webcam, speaker array, software.*

Microphone is an artwork by *Communications*, which is a collaboration between EunJoo Shin and the present author. Microphone was installed at the Unleashed Devices group show at the Watermans gallery London in Autumn 2010. Microphone invites participants to communicate with each other across a gallery, using two large microphones. The devices in Microphone do not operate in the same way as conventional microphones as we described in 2.1. The sounds are captured not with a conventional electronic transducer but with a digital camera, with software trained to produce vowel formants from mouth shapes.

The work invites participants to communicate using vocal sounds as a medium for gesture without language, bringing focus on the role of movement in communication. It evokes a feeling that is literally visceral, of vocal organ encoding patterns of movement into sound, and being perceived as movements.

Microphone uses computer vision in a similar manner to *Acid Sketching* described above (§2.6), in that it uses OpenCV blob detection to identify a polygon representing the shape of the mouth. From this polygon the parameters of roundness and area are derived as with Acid Sketching, along with the aspect ratio (height/width ratio of the minimum enclosing rectangle), and the convex hull area. These measures are not used directly, but instead used in combination to calculate a measure of similarity between the given mouth shape and the five vowels *a*, *e*, *i*, *o*

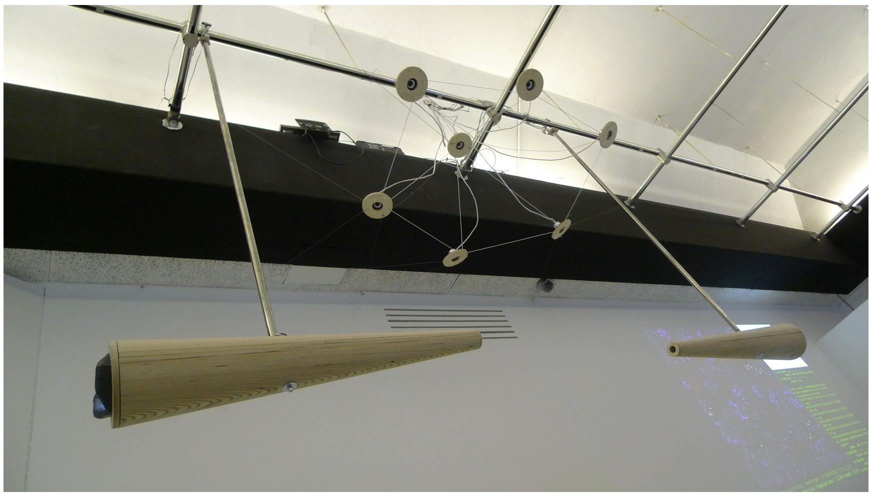


Figure 2.2: *Microphone, showing a speaker array which carried sounds between the two devices. The distance between the devices was constrained by installation in a group show.*

and *u*. The average formant values of the three closest vowels are then calculated, weighted by their similarity to the target shape. Although five discrete vowels are used in this calculation, a range of sounds between them are mapped continuously, so for example the neutral ə (schwa) vowel would be a point between them.

Microphone applies digital technology to map between modalities, but entirely analogue means may be used to much the same ends. There are long traditions of using the mouth as paralinguistic, musical instrument, either alone or augmented with instruments such as the Jew's harp, as we will see in the following chapter. The digital foundations of this artwork allows expectations to be confounded in interesting ways, but outwardly, Microphone is an entirely analogue artwork.

2.9 Discussion

We have taken a fairly unconventional view of symbolic representation, but one that we have suggested to be plausible in the context of cognitive psychology. In particular, we have taken a view of symbolic representation that takes both analogue and discrete symbols into account. Computation may at base be defined by the manipulation of discrete symbol sequences, but may be applied to simulate analogue systems, including aspects of the outside analogue world. Further, human cognition, including that of computer programmers, involves both analogue and discrete processing, lateralised and integrated.

Against this background we view programming as a human activity that spans both analogue and discrete domains. The two works we have presented, Acid Sketching and Microphone have in part been developed to explore this view. However both show the discrete aspects of programming as somewhat subservient to analogue aspects in the final artwork. They are 'interactive' works, where participants engage with the work by making analogue gestures which are digitised, transformed, then transduced back to analogue. In the case of Acid Sketching it is hand drawn gestures which are digitised, and in the case of Microphone it is mouth movements. The purpose of the digitisation however is to allow transformation from one analogue domain to another.

There is a further process of discretisation which happens in the ear and brain, perceiving Microphone's output as vowel categories, and Acid Sketching's output as discrete sound events in pitch classes. This categorical perception happens in tandem with the expressive spatial perception, where a listener might attend to how the participant transitions from one vowel to the next, or plays with different timbral parameters.

The computation in Microphone and Acid Sketching is hidden behind analogue interfaces based upon gesture, visual feedback and sound. This is the case in a great deal of 'interactive' computer artworks, where a great deal of energy is put into finding novel means of analogue expression. This is also very much true in computer music interfaces, such as those demonstrated at the New Interfaces for Musical Expression conferences.[6] Analogue expression is often a focus in the arts, and so research exploring new analogue interaction is of course very welcome. It has however led to some questionable claims being endemic in the electronic arts; that analogue interfaces are somehow more advanced than discrete ones, that computation necessarily *should* be hidden in art, and that computation is secondary to analogue experience. For example the artist Simon Penny has the following to say about his piece "Fugitive 2":

> The intervention of fugitive is to present a mode of interaction which is predicated on the system interpreting a person engaged in normal human bodily behaviors. This is in stark contrast to the conventional notion of interface, in which ideas and concerns must be encoded, usually as alphanumeric data, demanding the sequential pressing of little buttons on a board. This ut[t]erly impoverished interface functions, in fact, as a filter, excluding all rich and diverse aspects of human intelligence which cannot be encoded alphanumerically.[7]

This may seem a reasonable point of view until we notice that he is arguing against the writing of novels as impoverished compared to (in this case) moving around before a camera. To argue that either analogue or digital representations are more advanced is equivalent to arguing whether novels or paintings are a more advanced form of expression. Indeed we might argue that novels are 'more advanced' as print is a later technological development, and indeed builds upon the accomplishments of paint. This would however be a weak position to

[6]`http://nime.org/`

[7]Retrieved from `http://ace.uci.edu/penny/works/fugitive2.html`, July 2011

take; as we have argued through this chapter, humans are marked by the integration between linguistic and spatial forms. It follows then that whether discrete representations are shown in works depends entirely on the focus of the artist, whose engagement may include source code, written natural language, musical notation, low-resolution lights arranged in grids, or any other discrete, linguistic code.

From here then, we look for ways in which computer art can *integrate* the various codes of language and perception, in new works that engage with full, rich experiences across both domains.

Chapter 3

Words

Symbols are to words as positions are to movement. These words that I am writing, and you are reading, are sequences of letters written in the Latin alphabet, from left to right. They are members of a lexicon of many hundreds of thousands of words, which includes much apparent redundancy, although every synonym has its own pattern of usage. Before words came to be written like this, they were only spoken. A pencil leaves a line, but air pressure waves from the moving vocal tract settle quickly to the mean, leaving no trace behind. Except of course the subjective trace, brought by hair movements in the cochlea, reduced by the listening brain into a perceptual image, then perhaps reduced further into a form held in memory.

As we have seen, the English alphabet has a loose phonetic mapping. Children are conventionally taught to read using phonics, teaching letters of the alphabet by typical sounds they represent, which are strung together to pronounce words. However, there is a significant leap from theory to practice; this is a natural language, with much ambiguity and exceptions for every rule. Nonetheless once learned, natural language can feel effortless to the point of invisibility. We may feel completely absorbed in a text, which is remarkable as literacy is a comparatively recent, wholly cultural development; we have not evolved to read and write.

In programming, words are used rather differently. Aside from comments written in natural language, words in source code are either language keywords, or names given to variables and functional abstractions by the programmer. As with the use of spatial arrangement examined in §2.2.3, through the process of tokenisation, the words are ignored by the language interpreter as secondary notation. Words are not treated as having morphology, and are instead reduced to unique, shapeless tokens. Even the practice of capitalising names of libraries of code is by convention, rarely enforced by the interpreter; the naming is left entirely to the programmer. This differs from natural language, where morphological word structure relates important meaning, such as the suffix -s or -es to indicate plurals in English. Furthermore, programming language does not have the sound symbolism we noted in §2.4, such as the onomatopœic words clutter, quack and bang, or the perceptual relation between fricative consonants and sharp shapes (Ramachandran and Hubbard, 2001b). In natural language the morphology is not only syntactic, but has structure which mirrors what may be represented. Such relations are generally not considered important to modern study of linguistics, but nonetheless illustrate a richness of human language, which we may draw upon in the design of human-computer interactions.

While word morphology is absent from programming languages, allusions to it may occasionally be found. In the Ruby language it is a convention for destructive methods, as in those that directly modify mutable variables, to end in an exclamation mark (Flanagan and Matsumoto, 2008). Regular Expression (regex) language is designed for matching strings of text according to rules defined by terse, often single character operators and modifiers (Friedl, 2006). The result is a language on the word level; for example the following matches some variants of the root word *colour*:

```
/\bcolou?r(s|ist|ing)?\b/
```

Despite word morphology not featuring significantly in programming syntax, it is of course important for a chosen name in source code to reflect what it symbolises, including morphological aspects such as $-s$ suffixes on methods that return multiple values, and appropriate use of nouns and verbs to describe object classes and methods. This is convention in secondary notation which we will cover in greater detail in §5.1.

The general absence of word structure in the syntax of source code is indicative of the lack of articulation in the activity of programming. The particular movements of a programmer's fingers are left behind at the keyboard, translated there into discrete on-off states. Our question for artist-programmers working in computer music is, how can we relate our discrete representations back to analogue movement? Words have life as articulations, as well as sounds and symbol sequences, and so artist-programmers have much to learn from their study. In the following we will look for greater understanding of this issue by examining the perception and structure of words, finding motivation in vocal traditions of music.

3.1 Perceiving Speech as Movement

We begin our examination of words with a reduction of the spoken word to minimal components. *Sine wave speech* is where the complex, time-varying properties of a speech sound signal are reduced to a few sine waves (Remez et al., 2001). Typically, the frequency and amplitude of three sine waves are mapped from the lowest three formant frequencies, and a fourth sine wave from a fricative formant. The result is a bistable illusion, where an untrained human subject initially perceives sinewaves as separate, burbling artificial sounds. However once they are directed to attend to the sounds as a human voice, they are able to perceive a single stream of intelligible speech. It is surprising that speech is perceivable at all from these simple modulated tones, with all hisses, pops and clicks removed. Despite the short-term acoustical properties of speech being absent we can still perceive speech in the variance of pure tones, even identifying particular speakers by it.

The influence that non-acoustic cues hold over speech perception is also demonstrated by the McGurk-McDonald effect (McGurk and MacDonald, 1976). A classic demonstration of this effect is where a subject is simultaneously presented with the sound /ba/, and a video of a face mouthing the syllable /ga/, but 'hears' neither, instead experiencing the illusionary syllable /da/. This perceptual effect is strong for the majority of test subjects, stable even when subjects are made fully aware of what the audio stimulus really is. What is more, the illusion persists even when subjects are not consciously aware of what they are looking at; Rosenblum and Saldaña (1996) reproduced the effect without showing the face, but instead only point-light movements taken from a moving face. This is a closely related result to that of sine-wave speech; both cases show that features of the signal are not as important as how those features vary. In other words, movements derived from the signal source are more important than recognition of its short-term properties.

Such speech illusions have been taken as support of the Motor Theory of Speech Perception (Liberman and Mattingly, 1985). According to this theory, a 'special module' has evolved in the human brain for speech, responsible for both speech production and perception. Despite having some popular notoriety, motor theory is not widely supported within the speech perception field, as the notion of this special module is not well defined (Mole, 2009), and is not supported by the evidence (Galantucci et al., 2006). While the concept of a special module for language is generally no longer taken seriously, other aspects of the theory have become widely accepted. Motoric contribution to speech perception is well supported, and sits well with broader literature demonstrating strong motoric contribution to perception in general (Galantucci et al., 2006). It would seem that rather than speech being 'special' as Liberman and Mattingly originally suggested, that meshing of action and perception is a feature of human perception in general.

The meshing of perception and action has enjoyed renewed interest since the existence of *mirror neurons* was identified by di Pellegrino et al. (1992). Mirror neurons were identified as a class of neurons in the F5 region of the premotor cortex in a laboratory experiment with a Macaque monkey. These neurons were seen to fire both when the Macaque observed an assistant grabbing food, and when the Macaque himself grabbed the food. The explanation offered by di Pellegrino et al. (1992) is that these neurons represent *action understanding* in motor areas. The hypothesis is that because the same neuron fires regardless of whether the subject or an observed individual performs an action, that this is the basis of *social* cognition. This experiment captured the imagination of many researchers interested in embodied social cognition, not least Ramachandran (2000), who was driven to exclaim that the discovery of mirror neurons was on the same level of the discovery of DNA. This enthusiasm has reached the field of music psychology, with Leman (2007) citing mirror neurons as a basis for social understanding of music cognition based on gesture/perception couplings. However despite all this enthusiasm, mirror neuron theory has a number of problems. For example, several of its original assumptions have not held, and it fails to explain a wealth of evidence from brain lesion studies (Hickok, 2009). A neural system underlying social cognition is certainly cause for great interest, but it is early days for mirror neurons, and care should be taken not to extrapolate from findings which remain controversial and unclear (Dinstein et al., 2008).

It is early days for neurobiology in general, but controversy over these initial claims does not mean that a human mirror system does not exist in some form, with or without mirror neurons. While fMRI studies do not give data on the level of individual neurons, a number show overlap in performing and observing actions (e.g. Calvo-Merino et al., 2005; Chong et al., 2008), suggesting motor simulation does have a role in action understanding. In any case, we have seen evidence for strong motoric contribution to speech perception, demonstrating the influence our bodies have over what we experience. This continues the theme of interaction of continuous and discrete representations from the previous chapter; speech is a medium for discrete units of language deeply intertwined with the continuous movement that carries it.

3.2 Vocable Words in Music Tradition

Like speech, instrumental sounds are produced via a series of articulations of the human body. Modern digital computers allow us to synthesise sound with algorithms, but nonetheless we perceive the result with a brain evolved and developed for deriving movement from sounds. We have examined the role of articulation in the perception as well as production of speech, and now consider the same or analogous relationship with the sounds of musical instruments. In particular, we consider articulation as a perceptual bridge between musical instruments and the human body, and will later propose a means for this bridge to be used in the design of computer music notation.

A *vocable* word is simply a word that is able to be spoken and recognised, according to a system of phonetics (§2.7). The word *vocable* is generally used to describe 'nonsense' words that are non-lexical, but nonetheless readily pronounceable using the phonetic sounds of a particular language. In musical tradition, vocable words are often used to describe an articulation of a musical instrument. For example a music instructor may use their voice to describe a sound their student should try to make on their violin, perhaps by singing a pitch contour while using a consonant-vowel pattern to indicate a particular bowing technique. Over time the student will learn to perceive the phonetics of their instructor's voice as the sound categories of their instrument.

Vocable words can be found in use across the continents and in many musical traditions. Indian classical music has *Bol* syllables to 'speak the drums' where, for example, *ṭe* represents a non-resonating stroke with the 1st finger on the centre of the *dāhinā* (right hand) drum (Kippen, 1988). Bol syllables are often used in refrain, where the tabla player switches from playing the drums to *speaking* them with vocables. In the Scottish Highlands we find *Canntaireachd* of the bagpipes (Campbell, 1880), for example *hiaradalla* represents an echo of the D note in the McArthur Canntaireachd dialect. As with Tabla, Canntaireachd is used in performance, and the Indian Dhrupad singer Prakriti Dutta and piper Barnaby Brown have collaborated to unite the forms in vocal performance[1].

Canntaireachd and Bols are largely vocal traditions, which have come to be written down in recent times. Some vocables have traditionally been transcribed however, such as the *chien-tzû* notating the delicate finger techniques of the *guqin* (*Chinese zither*). For example *ch'üan-fu* indicates that the index, middle and ring finger each pull a different string with a light touch, making the three strings produce one sound 'melting' together.

In her doctoral thesis "Non-lexical vocables in Scottish traditional music", Chambers (1980) divides the use of vocables as either being culturally *jelled* or *improvisatory*. Jelled vocables, such as Bol or Canntaireachd vocables, are part of a formal system, where particular vocables represent particular articulations of an instrument. Improvisatory vocables on the other hand are made up during a performance, such as scat singing in Jazz. Chambers acknowledges that the line between jelled and improvisatory is often blurred, where for example a player formalises some aspects of their 'diddling' over time. Another distinction made by Chambers is between *imitative*, onomatopoeic vocables and *associative*, arbitrarily assigned vocables. This is a perceptual distinction, as Chambers reports: "Occasionally a piper will say that a vocable is imitative (indigenous evaluation) when analysis seems to indicate that it is actually associative (analytic evaluation) because he has connected the vocable with the specific musical detail for so long that he can no longer divorce the two in his mind" (Chambers, 1980, p. 13). In other words, a vocable may appear to mimic an instrumental sound on the perceptual level, without

[1]A video of this collaboration is available online; `http://www.youtube.com/watch?v=I_7wh_ClamA` (accessed March 2011)

having similarity on the level of the sound signal. This seems to be true in the general context of onomatopoeia – for example where a native English speaker hears a hen say “cluck”, their German neighbour may perceive the same sound as “tock” (de Rijke et al., 2003). Research into tabla Bols have however found them to be genuinely imitative, sharing audio features with the instrumental sounds they represent, identifiable even by naive listeners (Patel and Iversen, 2003).

A third distinction can be made between vocable words in *spoken* and *written* form. A reader of a vocable applies *paralinguistic* phrasing not derived from the text, but nonetheless with great musical importance. Conversely a transcriber may resolve ambiguity in a spoken vocable by writing a precise interpretation of the intended discrete pattern. These issues are of course common to all notation systems, including those of natural language (§2.2.1). We can say however that to some degree a written vocable may capture the discrete essence of a sound, or at least a mnemonic focus to the whole articulation. From our discussion of Dual Coding (§2.2), we can understand a written vocable as less accurate in capturing the full expressivity of a sung vocable, but *more* accurate in precisely capturing aspects of its discrete structure. This is simply a focus on either discontinuities or on smooth transitions, both important musical features, and as such a sound may be fully understood in terms of both.

The popularity of staff notation in Western Classical music has, at least in the case of Canntaireachd, led to a reduction in the teaching of jelled vocables (Chambers, 1980). However in reaction to the lack of standard means to notate articulation in staff notation, Martino (1966) proposes his own method grounded in phonetics. Each of his notational marks represent a vocable syllable, for example the mark ′ has the phonetic representation *tat*, described as a “incisive, crisp attack with similarly dosed decay”. Martino asserts that the articulatory parameters of any musical instrument can be understood as a subset of those of the voice, and as a result that his notation applies to all instruments.

Vocables in music are often referred to as being *non-lexical* (Chambers, 1980). It would seem that the definition of jelled vocables implies a lexicon, a dictionary of words that symbolise particular articulations of an instrument. On closer examination however vocable words are sub-lexical. Canntaireachd is not so much structured by a lexicon but by a system of phonetics, which in combination, generates a broad range of possible vocable words, with no lexical reference apart from their phonetic structure. As such, improvisatory vocables are pure sound symbolism, with reference occurring on the phonetic, and not lexical level.

3.3 Timbre

Timbre is an important component of music, yet is little understood. We have seen how vocable words allow instrumental articulations to be related to articulations of the vocal tract. Articulation is closely related to timbre, where movements of instrumentalists such as of plucking style, breath control and after-touch have strong influence over the resulting timbre. In the present section we will build an argument that this influence is crucial to the experience of music, asserting that in perceptual and conceptual terms, articulation *is* timbre.

The American Standards Association (ASA) defines timbre in their Acoustical Terminology standards as "...that attribute of auditory sensation in terms of which a listener can judge that two sounds, similarly presented and having the same loudness and pitch, are different" (Bregman, 1994). Where the ASA standard is quoted it is often derided, for example Bregman (1994, p. 92) paraphrases it as "We do not know how to define timbre, but it is not loudness and it is not pitch", in other words an "ill-defined wastebasket category". There have been several research attempts to establish a better definition of timbre, but with little success, and in a broad review Hajda et al. (1997) highlight this lack of definition as the most lasting obstacle in the research of timbre. Indeed without being able to define timbre, it is difficult to know how to look for other obstacles. Timbre seems to be an important part of music, but we cannot agree on what it might be, or even how to approach defining it.

For context, the same ASA document as above is also quoted as defining pitch as "... that auditory attribute of sound according to which sounds can be ordered on a scale from low to high." We can restate this in our terms, that ASA define pitch as an orientational metaphor of the sort examined in §2.2.6. We infer then that the ASA believe that other aspects of sound are *not* orderable along such a scale. Pitch certainly has a clear relationship with the physical world, being the perceptual counterpart to frequency of waves of air pressure. Pitch also correlates with physical body size, as smaller animals tend to have shorter vocal tracts with higher resonant frequencies, resulting in higher pitched vocalisations. This makes pitches amenable to being ordered along a quality dimension. However, the perceptual quality of loudness has a similar relationship with wave amplitude and physical body size; smaller things tend to be quieter. The ASA definition of pitch is under-specified; it fails to exclude orientational metaphors which we claim structure much of the experience of music, timbre included. To look for support for this claim, we turn to psychology of timbre literature, where the search for the dimensions of timbre has been a common research aim.

3.3.1 Multi-Dimensional Scaling

Unlike pitch and loudness, timbral differences do not have direct relationships with the fundamental features of air pressure waves, hence their consignment to a wastebasket category (Bregman, 1994). However a number of researchers have attempted to identify distinct quality dimensions of timbre. The hypothesis is that there are a certain number of quality dimensions defining a space in which timbre is perceived. These attempts have in general applied Multi-Dimensional Scaling (MDS; Shepard, 1962) or closely related approaches of dimensionality reduction. In MDS experiments, similarity judgements are collected from human subjects and interpreted as distances, which are then used to reconstruct the perceptual space in which the judgements are assumed to have been made. We can express this aim as capturing the dimensions of mental imagery (§2.2.1) or conceptual spaces (§2.2.5) at play when listeners attend to musical timbre.

MDS similarity judgements are classically given in response to pairs of stimuli, as a mark along a scale from very dissimilar to very similar. They may alternatively be derived from sorting tasks or closest pairs in triplets (Weller and Romney, 1988; Hajda et al., 1997), with various effects on the scope and reliability of the results (Bijmolt and Wedel, 1995). An early attempt at applying MDS to instrumental sounds was performed by Grey (1977), where human subjects were asked to rate pairs of sounds on a similarity scale. The stimuli were synthesised based upon the physical properties of orchestral instruments. However Grey interpreted the MDS solution as showing clustering that went beyond familial groupings of instruments to showing articulatory features. Furthermore Wessel (1979) used MDS techniques to produce control structures for synthesis, so that additive synthesis of timbre could be manipulated according to movements in a perceptually salient space resulting from the scaling.

Wessel interpreted his MDS solution as having two dimensions, namely *brightness* and *bite*. Brightness is generally quantified as the frequency of the spectral centroid, and Wessel (1979) relates 'bite' to the nature of the sound onset or *attack*. A review of MDS timbre studies performed since Wessel's early work shows the majority of studies have these two dimensions in common (Caclin et al., 2005), however besides this there is a great deal of inconsistency, with many other dimensions identified in individual studies without broader agreement. These include spectral flux, spectral spread, spectral irregularity and harmonic onset asynchrony.

In summary, the broad finding of MDS studies of timbre thus far is that spectral centroid and attack time seem to be important. Little more is agreed upon except for one important point; inter-individual differences are significant (Caclin et al., 2005). From our perspective, we would certainly expect it to be so, perception being not directly against the background of physical reality, but of personal experience of it (§4.2). Some of the inconsistency in the results of MDS timbre studies may be methodological, MDS is known to be highly sensitive to optional cognitive processes influenced by test conditions, even when similarity measures are based on Stroop interference (reaction times). For example, MDS is often used to find whether two dimensions of perception are related, by looking for either a city block or Euclidean distance metric in the MDS solution (e.g. Gärdenfors, 2000, p. 25). However, Melara et al. (1992) found that such metrics were entirely down to optional processes employed by test subjects, and could be straightforwardly manipulated by altering the wording of test instructions. Such optional cognitive processes are a serious and difficult problem to deal with.

Krumhansl (1989) looks beyond methodology in suggesting that the very notion of universal dimensions of timbre is flawed. Although there may be one or two dimensions with general salience, different sound sources have particular characteristics resulting in their own dimensions. Instead, timbre should be understood not just as frequency analysis, but also relative to the dynamics of a real, simulated or imagined physical source behind the sound. It is clear that MDS studies have focused on the former (Hajda et al., 1997), but we join Krumhansl in arguing that the strong role of physical movement in the perception of timbre has too often been overlooked.

3.3.2 Grounding Timbre in Movement

Many of the words used to describe instrumental sounds are metaphors for the physical manifestations of movement, shape and the body; we have already seen that Wessel (1979) used *bite* to describe sound onset quality, an oral metaphor indicative of the close perceptual relationship between sound and bodily movement. This theme is explored by Traube and D'Alessandro (2005), who finds that vocal articulation has a strong role in guitarists' timbre perception. Traube and D'Alessandro investigate both the lexical words and non-lexical vocables that guitarists use to describe guitar sounds. In the former case, They asked guitarists to choose words to describe guitar notes. Amongst other words, sounds obtained by plucking the string close to its middle were ascribed words such as *closed* and *damped* whereas plucking close to the bridge were described as *thin* and *nasal*, and plucking midway, over the sound hole was described as *large*, *open* and *round*. This is in apparent subconscious reference to mouth shapes for the vowels that have similar formant structure to the resulting guitar sounds. Traube suggests that this shows use of mental imagery to communicate timbre, relating it to the use of vocable words by tabla players. Indeed through experiment Traube demonstrates that when guitar players improvise vocables, there are positive correlations both between plosive consonant and plucking angle, and between vowel and plucking position.

3.3.3 Music of Timbre

In Western culture, the primary basis of music is widely considered to be that of pitch contrasts over time. This is evident in the clinical definition of those with deficits in music perception; *amusia* is generally considered primarily as a deficit in pitch perception popularly known as 'tone-deafness' (Pearce, 2005). There are musics however where contrasts of discrete tones is unimportant, which we describe as *music of timbre*. While we argue that perceived movement is of universal importance to music, to some musics, it would seem that this is almost all there is. The question is, can music achieve the same levels of complexity without tonality, or does the focus on movement involve a trade off, a shift from musical contemplation, perhaps towards something with a broader 'non-musical' function in culture?

In his book "Music, Language and the Brain", Aniruddh Patel notes that despite much musical experimentation, music based on timbral rather than tonal contrasts is rare (Patel, 2007, pp. 30–37). This is explained twofold; firstly, timbral changes often require instrumental manipulations that are physically difficult to perform in quick succession. The second, cognitive reason he gives is that timbral contrasts cannot be perceived in terms of intervals, so the higher order relationships associated with tonal music are not supported by timbre. A given counter-example is music of the tabla drums in Indian classical music, which Patel (2007, pp. 62–67) explains by looking at the music culture around the tabla. In particular, he notes the extensive use of Bol syllables, a jelled system of vocable words (§3.2). Patel concludes that these vocables allow perception of complex timbral contrasts to be aided by cognitive resources developed for linguistic structure.

We propose an alternative hypothesis to Patel (2007), that it is not specifically linguistic sound categories which support timbre perception in Tabla music, but more generally categories of articulation, which just happen to be of the vocal tract in this case. The human ability to perceive and categorise movement is used in the transmission of language, both in vocal and sign languages (Sutton-Spence and Woll, 1999), but we contend that this is a more general cognitive rather than linguistic resource; speech is not 'special' (§3.1) as Patel implies. Audiences can perceive tabla music because when performers speak the drums, listeners are able to relate the sounds to the movements of their own vocal tracts. In the case of electronic dance music, it is not the movement of the vocal tract, but of the whole body dancing that opens the door to broad appeal found within large audiences.

The strong implication made by Patel (2007) is that without co-opting linguistic resources, music of timbre would be impossible, or at least rare. Together with the lack of clarity found in MDS study, it would be too easy to conclude from this that timbre is therefore unimportant in music cognition, being either too complex or too formless to provide structure for music. Electroacoustic music has not found popular audiences beyond its academic base. However Patel overlooks the existence and popularity of electronic dance music, which often centres around manipulation of synthetic timbre with apparently no supporting use of vocable words. An example of this is the *acid line*, a musical style exemplified by the 1995 recording of "Higher state of consciousness" by Josh Wink. This piece revolves around slow manipulation of the timbral parameters of a repeating motif on the Roland MC-202, a monophonic, analogue subtractive bass line synthesiser. Synthesis filters are manipulated on this machine during the piece, with low-pass cutoff slowly increasing tension until releasing into a single, final crescendo. The timbral changes of this example are slow, but one does not have to look far to find timbral manipulation at a speed and complexity comparable with that of tabla virtuosos. For example Autechre's *Gantz Graf* (2002) has little in the way of discernible melody, but manipulates sound events at a speed on the boundary between percussion and metallic drone. Autechre regularly attract audiences of thousands across Europe and the USA, who dance to fast-changing, complex rhythms in the dark, with no visual accompaniment.

The role of movement is unsurprisingly central to dance music, but to the extent that aspects of music cannot be understood without dancing, or at least imagining oneself dancing. The Clave rhythm is a case in point, where the main beats are emphasised not through musical accents, but in the associated dance steps (Agawu, 2003, p. 73). Indeed in many cultures the concept of music encompasses both the sounds and the dance, and one is not understood without the other (Agawu, 2003, p. 264). In the case of Autechre, the musicians provide rich and complex timbral structures, for which audience members create their own reference points through their own bodily movements.

Some musicians attempt to produce *acousmatic* music free from physical manifestation and constraints. Sounds may include those recorded from recognisable sources, but are used for their sonic properties. This music is not performed, only existing in recorded form, and in a concert setting is played over loudspeakers. It is of course possible to enjoy this music through physically static, deep listening, but we contend that it is difficult to do so, and that general audiences find it troublesome to relate ungrounded timbre to their own bodily experience. This difficulty is expressed well by Smalley (1994, p. 39); "In electroacoustic music where source-cause links are severed, access to any deeper, primal, tensile level is not mediated by source-cause texture. That is what makes such types of acousmatic music difficult for many to grasp. In a certain physical sense there is nothing to grasp - source-cause texture has evaporated." Smalley concludes that to free timbre from source-cause, the composer must confront and enjoy the dissolution of timbre.

Emmerson (2007a) describes a dichotomy between dance and art music in electronic music as being between either a focus or freedom from repetition. Emmerson relates repetitive music to a grounding in the movements of the human body, and amorphous music to movements in the environment. It should be noted however that while the celebrated electroacoustic composer Stockhausen disliked the “repetitive language” in electronic dance music (Emmerson, 2007a, p. 62), he also compelled students of music to “go dancing at least once a week. And dance. Please, really dance: three or four hours a week.” (Stockhausen and Maconie, 2000, p. 170). We share the conclusion with Emmerson (2007a), that musicians who work on the boundary between these dance and art musics are reconciling the dual themes of body and environment. That these two groups are in many cases already using the same tools and languages offers such musicians a unique opportunity.

3.3.4 Defining Timbre

We have focused on the role of movement in timbre perception, but until now have avoided explicitly defining timbre. In contrast to the approaches of trying to define timbre in terms of spectral features, or the ASA approach of defining timbre by what it is not, we define timbre more broadly in terms of mental imagery:

> Definition: *Timbre* is sound as it is perceived, as mental imagery of positions, regions and their variance in action spaces.

We need to be careful in interpreting this definition. Musical events are by definition discrete, not only notes but also vocable words such as tabla Bols. However we characterise such discrete symbols as signposts in continuous timbre space. We may notate timbre using discrete symbols, but the timbre we notate is understood in terms of points, areas or movements in perceptual spaces. Thus *brightness* names an analogue dimension, *bright* names a range within it, and *brighter* a direction along it.

Our definition of timbre does not exclude pitch, and we contend that it is not possible to do so – for example pitch and brightness are so interdependent as to be inseparable. When pitch scales are used, attention is shifted away from the continuous timbral quality of pitch and towards discrete relationships between notes. This does not mean that mental imagery of pitch height is ignored, but rather that the balance shifts in some measure from analogue, timbral movement to structures in a discrete domain. The development of staff notation in the Western tradition has de-emphasised the older oral tradition of vocable words, distancing music from the body (§3.2). This distance can be considered a freedom, allowing us to explore and contemplate the multi-dimensional complexity of discrete tonality. However we contend that what makes a melody affecting is its *integration* of both discrete notes and analogue timbral experience. The same can of course be said for integration between discrete pattern and timbral movement in Tabla playing.

Integration between the continuous and discrete in music is the normal case, but it is possible to perceive timbre without associated discrete coding. For example an elongated drone, with smoothly fluctuating timbre, does not have discontinuities that would evoke discrete segmentation. Likewise, if an event is repeated in performance with little variation, such as a repetitive kick drum, then it may be attended to primarily as a discrete pattern, but still, its pulse is likely to be perceived and understood relative to sounds around it in spatial terms, in relation to its low position on the pitch dimension.

3.3.5 Timbral analogies

If timbre is a product of movement in mental imagery, then we might expect orientational metaphor (§2.2.6) to play a large part in its structure. This is what the notion of *timbral analogy* amounts to, the idea that timbral differences are perceived as distances, and related to one another as such in perception. Wessel (1979) applied a perceptual space identified through MDS (discussed in §3.3.2) to test the existence of timbral analogy and found encouraging results; subjects would tend to choose the D in the form A is to B as C is to D that results in the distance AB being closest to CD. A similar experiment is given by McAdams (1999), which also showed some support for the timbral analogy hypothesis, however also showed strong individual differences and irregularities. As Patel (2007, pp. 30-34) points out, this indicates that unlike tonality, music of timbre does not easily support intervals as a shared category system. However while McAdams (1999) does find individual differences, electroacoustic musicians made judgements more consistent with one another, suggesting that this could just be a matter of cultural exposure. It would be interesting therefore to apply similar experimental design to the perception of timbral analogies between the parameters of bass line synthesisers (such as the Roland MC-202, §3.3.2), and compare the results between those of varying exposure to acid house music.

On another level, we may question whether shared category systems are necessarily important to music. As we have argued, music can be grounded and creatively interpreted in the dance of the listener, and so the job of the composer or improviser is in such cases not to encode a musical 'message' to be unambiguously decoded. Rather, it is to sculpt music with structures of interest, that may be interpreted according to analogies dynamically constructed by the listener; music that sounds different on each listen, being framed by the particular state of the listener's mind and body.

3.3.6 Sound source modelling

In relating mental imagery to action spaces, our definition of timbre implies that sound source modelling is the basis of sound perception. An expanded form of this assertion is that through listening to a series of sounds we build a model of aspects of a physical object and its excitation, and how it reacts to actions to produce sounds. This model is then used to structure perception of the sounds which gave rise to it.

The idea that we perceive a sequence of events relative to a model built from those events may seem far-fetched, but such dynamic relationships are far from alien to music psychology. For example it is widely accepted that underlying metre is inferred from rhythm, and that the rhythm is then perceived relative to the metre (London, 2004). This is particularly explicit in Indian classical music, where the metric *tāl* is a clapping pattern inferred from the rhythm that is based around it (Clayton, 2008). This can happen within the time frame of the psychological present, so that the perception of metre and rhythm are unified, appearing to influence one another simultaneously. It would seem that the same is true of timbre; humans are able to identify instruments within a mixed auditory scene effortlessly (Bregman, 1994), and so at least some source modelling is at play. The questions are, to what extent does this ability contribute to timbre perception, and to what extent are neural motor circuits involved?

Some studies have found correlations between deficits in music perception and of spatial ability, with Douglas and Bilkey (2007) making the argument that amusia is part of a general deficit in spatial ability, supporting the assumption that music perception is a form of spatial perception. Such findings are controversial, and difficult to interpret; among other issues it is unclear whether spatial ability improves music perception, or listening to music improves spatial ability (Stewart and Walsh, 2007), although we would suggest a third option that both cases are true. That there is *some* relation seems clear, for example Cupchik (2001) finds that mental rotation of figural drawings (after the classic mental imagery work discussed in §2.2.1) predict ability to discern analogous musical permutations. It is interesting however that this debate in the literature is made solely on the basis of pitch perception. If, as we suggest, music is perceived simultaneously in terms of discrete events and analogue movements, and pitch tends towards the former use, then perhaps more conclusive results would be found if wider timbral aspects of music were examined with these experimental designs.

3.3.7 Universality of Timbre

Music is popularly described as a universal language, the implication being that music is not culturally specific, and does not need to be learned. There is support for some interpretations of this in the literature, Fritz et al. (2009) found that listeners in Mafa, Cameroon judge the emotion of Western pieces of music in agreement with Western listeners, at above chance levels. We may imagine how the Mafa achieved this feat, despite lacking prior exposure to Western culture. *Emotion* is defined by Fritz et al. somewhat coarsely as being either happy, sad or scary, to which they report clear correlations with basic musical features; for example 'happy' music has a faster tempo and 'sad' a slower tempo. However these universals are not necessarily musical in nature. It is universal human behaviour for happy people to move faster than sad people, due to physical effects on motor control. While it is unsurprising that Western musicians choose to take advantage of this in their work, it is quite a stretch to make the broad claim that music is a universal language on this basis, or even that music has universal components. It is equivalent to observing that when people are sad, they speak slower, and arguing on this basis that speech is a universal language. While humans are universally driven to make music, as they are driven to speak, the result is many musics, not one.

So, naive listeners may be able to pick up references to mood in Western music, through certain coarse universals of an analogue code. Overall though, music is defined within a particular human culture. Traditional dance is a visible aspect of this, not just accompanying music but an intrinsic part of the musical experience (§3.3.2). Indeed we agree with the analysis of Western *musicking* by Small (1998), that music is an activity that encompasses not only playing instruments and dancing, but also concert ticket sales, dressing up, the hubbub in concert hall atria prior to the concert, the clearing of throats, and the striding on stage of the conductor. Music is a cultural activity in the deepest sense, and by taking an external perspective on an otherwise familiar Western activity, Small shows what strange creatures we are. It is no surprise then that Western ethnomusicologists in Africa have found music grounded in movement, embedded in culture, and not focused on sound; this is the general case across cultures (Agawu, 2003, Ch. 5), including in Western music.

3.4 Articulation

Having defined timbre primarily as perceived articulation, we examine the articulation of words as discrete symbol sequences. This will complete the context required for the practical connection between timbre and computation that concludes this chapter.

We have already examined the use of symbols to notate positions of articulators in the vocal tract, as well as the related issues of embodiment and mental imagery through chapter 2. Articulation of words is in theory a simple case of moving from one position to another, but the timing and phrasing of articulation forms an additional, analogue channel of prosodic communication. Furthermore physical constraints lead to interactions between points of articulation known as co-articulation, a challenge for computational speech recognition.

3.4.1 Rhythm in Speech

We saw Patel (2007) draw a structural connection between music of the tabla and speech in §3.2. Patel goes further in the same volume with research comparing timing in speech with music performance. In speech research, a dichotomy was long accepted dividing languages into stress-timed and syllable-timed languages. It was generally accepted that stress-timed languages tend towards equal duration between stressed syllables, and syllable-timed languages tend towards equal duration between every syllable (isochrony). However, cracks appeared in this theory when agreement could not be met on which languages were which. When computers allowed measurements unaffected by hidden bias, it became clear that the stress/syllable timing dichotomy had no basis in the cold reality of the sound signal – no languages have isochronous timing.

The distinction between stress- and syllable-timing was however saved, or perhaps replaced, by Ling et al. (2000), through introduction of the normalised Pairwise Variability Index (nPVI). This provided a measure for quantifying something that was intuitively felt, but falsely attributed to stress/syllable timing. As the name suggests, nPVI measures the "degree of contrast between successive durations in an utterance" (Patel, 2007, p. 131), with a higher nPVI indicating greater contrast between neighbouring durations, and a perception of stress rather than syllable timing. In his search for correspondences between music and language, Patel applied the nPVI to music. Unlike speech, much music is isochronous, but by applying nPVI to both speech and music, Patel found that relationships could be found in the non-periodic structures of both. In particular, that cultures with spoken languages with a higher nPVI had a style of playing music that also had a higher nPVI. Because notes are normally played much faster than syllables are spoken, this is an inconclusive result, but does support a universal link between musical and prosodic expression. Furthermore, in certain cases this link is made explicit and undeniable, as we will see in the following section.

3.4.2 Sound poetry

Sound poetry is an artform closely related to the use of vocable words, in that speech is used for its timbral quality rather than as a linguistic medium. There is no reference to another instrument, instead attention is turned solely on the expressive abilities of the vocal tract itself. Among the most celebrated sound poems is the Ursonate by Schwitters (1932), with themes introduced, explored and expounded upon through four movements over twenty nine pages. Only non-lexical words are used, although the work includes instructions to recite the poem with German intonation.

The practice of sound poetry has related forms in music. For example the Italian composer Luciano Berio's *Sequenza III* is a vocal piece featuring mutterings, clicks and shouts of the female voice, notated with a unique system of symbols including vocables. The manipulation of the voice is taken to different extremes in *human beatboxing*, which emerged from the Hip Hop genre. Beatboxers employ extended vocal techniques to produce convincing impersonations of drum machines and bass lines (Stowell, 2008). Beatbox rhythms may be notated with a system of jelled vocables called *standard beatbox notation* (Tyte, 2008), where written syllables represent the different sounds. Although the notation is vocable, beatboxers generally intend to produce sounds that do not suggest a vocal source. Beatboxers anecdotally report that their hand gestures help them do this, imagining instruments in a spatial arrangement in front of them, and moving their hand to find the different timbres of their voice; a mapping from space to articulation.

3.4.3 Words in Music Technology

We have seen that vocable words are used across diverse music cultures. However at certain points, music culture has embraced technological advance changing how music is conceived. This includes the development of staff notation, a form of musical literacy, changing the way music is composed, taught and distributed. More recently, electronic and computational technology has provided radically new ways of describing musical sounds, particularly through understanding of the frequency domain and the development of new analogue and digital sound synthesis methods. In all these cases the result has been new music activity signalling a move away from oral tradition. In the case of staff notation vocable words have been replaced with notation focused on pitches, and in the case of synthesis the transmission of music has moved to tape, disc and computer networks. In the case of electronic dance music, human movements are often factored out through *quantisation*, normalising input data so that events fall into a precise, coarse grid. If the composer feels that the result is too robotic, human-like articulations are then synthesised, adding subtle time phrasing to events to give 'human feel' through performance rules (e.g. Friberg et al., 2006). We can say that for some popular computer music interfaces, control over the kind of expression that we have compared to prosody is abstracted away.

Although in some cases technology has replaced use cases for vocables, in others it has blurred the distinction between vocable words and the instrumental sounds they evoke. Indeed it has ever been thus; the Jew's harp, one of the world's oldest instruments, locates a reed inside the mouth, which is twanged while the mouth is articulated, allowing the musician to create timbral expressions based on vowel formants. A more recent technological counterpart to the Jew's harp is the vocoder, which when applied to speech imposes articulations of the vocal tract upon another sound source, thus allowing any instrument to 'speak'. We may also consider *Microphone* (§2.8) in this light. In such cases a spoken vocable word is not a symbolic representation signifying another sound, but is the sound itself.

Computer music technology has employed a wide range of human articulations in the specification of sound. Many alternate means of articulating a sound to a computer system have been developed, with many hundreds of examples in the annals of the New Interfaces for Music Expression conference. Gestures of the hands are sonified to shape timbre, equivalent to the vocal articulation of words, especially when we consider sign languages of the Deaf (§2.7).

The voice is a recurrent theme in the control of new music interfaces, hardly surprising as the vocal tract evolved to support articulations of great complexity, for the purpose of communication (Boer, 2010). We discussed the bistability of auditory and speech perception in §3.1 in relation to sine-wave speech, and this bistability has itself been treated as a musical medium. Jones (1990) describes how he leads the listener to perceiving non-speech as speech, and vice-versa, through timbral manipulation with the CHANT software (Rodet et al., 1984). Speech synthesis has also featured as a source of musical timbre in electronic dance music, for example the largely unintelligible singing synthesis software written by Chris Jeffs for use in his compositions under the "Cylob" moniker (Jeffs, 2007).

Perhaps a central problem underlying new music interface research is that composers need to describe sounds in ways which have some connection to the outside world. Reflecting on the above, we propose that one answer is to build models of human perception into software, so that the software may plausibly relate sounds to experience, as Wessel (1979) attempted with some success using MDS (§3.3.1). An alternative approach of building physical models, and controlling them with discrete symbol systems analogous to phonetics, is introduced in the following section.

3.5 Vocable synthesis

Vocables are words which musicians use to represent instrumental articulations (§3.2). Vocable synthesis then is the use of physical modelling sound synthesis in this process, to automatically render words into timbre.

Pioneering research into voice-control of synthesisers with vocable words is introduced by Janer (2008), where syllables are sung into a microphone and the resulting sound signal analysed and mapped to instrumental parameters. This work is inspired by the tradition of *scat* *singing* vocables in Jazz culture. Scat syllables are not jelled but improvisatory (§3.2), and perhaps as a result Janer is interested not in *which* syllables are sung, but *how* they are sung. As such he uses a digital computer to translate from one analogue form to another, a similar approach to the Acid Sketching and Microphone projects introduced in chapter 2; the underlying process is digital, but in terms of inputs and outputs it is perceived as a continuous mapping. Janer's approach allows realtime use, where a singer's syllables are detected, analysed and re-synthesised on-line. The result is a system which is very easy for singers to learn hands-on, through use.

The approach to vocable synthesis introduced here differs from that of Janer, in that we consider words in written rather than spoken form. That is, we use words for their discrete phonetics rather than analogue prosody, so that we may translate from a discrete form to a continuous one. The motivation for taking this approach is to provide a representation of timbre for live coders (§6.8), computer musicians who notate music in the digital domain under tight time constraints. Such musicians are often driven to specify large synthesis graphs to describe synthetic timbre, and so the promise of instead describing complex timbres with short words should be very attractive.

For vocable synthesis to be usable, it should include a coherent mapping, with a perceivable connection between symbols and sound categories. This is facilitated through the articulation of a computer model of an analogue sound source. The use of physical modelling synthesis promises that even a naive listener can perceive time variance of perceived audio features as physical movement. The musician then describes articulation with symbols, which the listener experiences through the music of timbre. The aim is that for the musician, the experience of using vocable synthesis should feel as natural as using onomatopoeia in spoken words.

3.5.1 Babble - vocable Karplus-Strong synthesis

An early implementation of vocable synthesis was introduced by the present author (McLean, 2007), and since adapted into the artwork *Babble*, commissioned in 2008 by the Arnolfini gallery in Bristol, accessible at `http://project.arnolfini.org.uk/babble/`. This artwork was inspired by sound poetry (§3.4.2), in which speech is used as structured sound within a poetic composition, rather than as a linguistic medium. The aim for this was to bring the attention of participants to the paralinguistic connection between symbols and sound.

In Babble, the consonants and vowels map to different aspects of a physical model. This is analogous to a system of phonetics, but is not intended as an approximation of the phonetics of a natural language. In particular, this is not intended to be a system for speech synthesis, but rather a system for speech-like composition of sounds. Similar results could have been found by taking a speech synthesiser and 'breaking' it, by adjusting the parameters just outside the point where the sound could be perceived as speech. The advantage of the Babble however is that it uses a computationally simple model, and is therefore able to run in a web browser.

The physical model used in Babble is *Karplus-Strong* synthesis, a trivial model which produces surprisingly realistic synthesis of strings, by simulating wave propagation using a circular sample buffer (Karplus and Strong, 1983). Each vowel corresponds to a size of sample buffer, analogous to the length of a physical string. A second parameter to the model is the probability of sample values being inverted, controlling how aperiodic, or *percussive* the results are. Each consonant corresponds to a different value of this parameter. That vowels and consonants control different aspects of the model is analogous to the organisation of the International Phonetic Alphabet (Ladefoged, 1990), where vowels relate to mouth shape and pulmonic consonants to the place and manner of articulation (§2.7). In addition, to give the sound a speech-like quality, a formant filter is applied to the synthesis, which is also controlled by the vowels.

The words typed into the online interface were recorded, although participants were able to opt-out with a privacy mode. From the resulting logs over three years of use it would seem that the work has been successful in bringing participants to attend to musical rather than lexical features of their words. Sessions often begun with meaningful passages, but quickly turned to 'nonsense' syllables, such as the following short excerpt, shown with time-stamps from the log file:

```
[2009-12-05 07:20:29] this really makes some interesting noises
[2009-12-05 07:20:55] ko doo ko doo - ko doo ko doooiiiT!
[2009-12-05 07:21:07] ko doo doo - ko doo ko doooiiiT!
[2009-12-05 07:25:14] ntdedvoxi hso - - lbpxuerohzyi - - jolkui
```

While Babble is successful as an artwork, and shows promise as an approach of timbral control, the range of expression by the two parameters of the Karplus-Strong model is too limited to meet the needs of a musician. For this purpose, vocable synthesis was re-applied to a more complex physical model.

3.5.2 Mesh - vocable waveguide synthesis

The *Mesh* vocable synthesis system is an extension of Babble, towards a greater range of timbre. It uses waveguide synthesis (Van Duyne and Smith, 1993) which is inspired by Karplus-Strong synthesis, but uses bi-directional delay lines. These waveguides are connected together to simulate strings, tubes, surfaces and volumes of arbitrary complexity, within whatever the current limits of on-line computation are. Mesh models a drum head as a two dimensional mesh of waveguides, using a triangular geometry for maximal accuracy (Fontana and Rocchesso, 1995).

The drum head is excited through interaction with a simulated drumstick, using a mass-spring model developed by Laird (2001). The drum head has parameters to control the *tension* and *dampening* of the surface, and the drumstick has parameters to control its *stiffness* and *mass*. The drumstick is thrown against the drum head with parameters controlling the *downward velocity*, *starting x/y position* and the *angle and velocity of travel* across the drum skin.

Table 3.1: *Mapping of consonants to mallet property (columns) and movement relative to drum head (rows).*

	heavy stiff	heavy soft	light stiff	light soft
across	q	r	y	s
inward	c	m	f	w
outward	k	n	v	z
edge	x	d	t	b
middle	j	g	p	h/l

Table 3.2: *Mapping of vowels to drum head tension (columns) and dampening (rows).*

	tense		loose
wet	i		u
		a	
dry	e		o

As with Babble, vocable words for Mesh are composed from the 26 letters of the modern English alphabet. The consonants map to the drumstick and movement parameters, and vowels to the drum head parameters, shown in Tables 3.1 and 3.2.

As an example, consider the following articulation:

> Hit a loose, dampened drum outwards with a heavy stiff mallet, then hit the middle of the drum with a lighter mallet while tightening the skin slightly and finally hit the edge of the skin with the same light mallet while loosening and releasing the dampening.

This above may be expressed with the single vocable word:

> kopatu

Polymetric vocable rhythms may be described using syntax inspired by the Bol Processor software (Bel, 2001). This was later rewritten for use in the string parser of the Tidal pattern DSL (Domain Specific Language), described in detail in §4.5.4.

3.5.3 Vocable manipulation and analysis

Our system is a discrete representation, but is defined in relation to simulated continuous movement. This means that we may musically manipulate vocable words as either sequences of discrete symbols or as continuous movements.

In the discrete domain, we have a wide range of techniques from computer science available to us. For example we may analyse sequences of vocables using statistical techniques such as Markov models. Such an approach to modelling structures of vocable rhythms in order to generate rhythmic continuations was introduced in earlier work (McLean, 2007). We may also use standard text manipulation techniques such as *regular expressions* (regexes). Regexes are written in concise and flexible language allowing general purpose rule-based string matching (Friedl, 2006). A regex parser is embedded in Mesh, allowing operations such as the following:

```
~%3=0 /[aeiou]/to/ fe be
```

This replaces the vowels of every third vocable with the string "to", resulting in the following sequence:

```
fto be fe bto fe be
```

Mesh vocables are direct mappings to the simulated physical space of a drum and its articulation. It is therefore straightforward to operate in a simulated analogue domain, in order to perform geometrical analyses and manipulations. For example combining vocables in polyphonic synthesis is straightforward, and implemented in our current system as follows. As consonants control the movement and mallet material, we allow two consonants to be synthesised concurrently simply by using multiple mallets in our model. Currently we allow up to five active mallets per drum, allowing five consonants to be articulated at the same time. As vowels control the properties of a single drum head, we combine them simply by taking the mean average of the values they map to.

We may exploit both symbolic and geometric vocable representations in one operation. For example we could estimate the perceptual similarity of two vocable words of different lengths with an approach similar to the symbolic Levenshtein edit distance (Levenshtein, 1966), with edits weighted by phonemic similarity in the simulated analogue domain.

3.6 Discussion

In this chapter we have studied words as sequences of symbols which notate discrete points of articulation, and therefore analogue movements between them, in writing, speech and song.

For humans, words evoke movement, a bridge between discrete and analogue modes that lies at the heart of our experience. We have defined musical timbre in terms of imagined movement, and noted the use of vocable words in describing movements within musical tradition. While noting that words are generally treated as shapeless tokens in computer programming, we have proposed a use of words in notating timbre within computer programs, allowing treatment of timbre through both digital and (simulated) analogue manipulations.

Chapter 4

Language

Thus far we have been occupied with issues of symbolism and movement, taking an embodied view of discrete computation integrated with analogue experience. In the present chapter we will examine the higher order concerns of the design of programming languages for computer music. Firstly, we will tackle the issue of what we mean by the word *language* in this context, by relating natural and computer languages. We will then approach the meaning of *meaning* in computer music, this time by relating natural language with music. We then examine notions of abstraction in programming language design, by contrasting imperative and declarative approaches. This will lead into the introduction of Tidal, a language designed for the live improvisation of musical pattern.

4.1 Natural and Computer Language

Comparing natural and computer language brings up thorny issues. How does writing a program compare to writing a poem: does it even make sense to speak of these activities in the same terms? They can at least in an arts context; computer language poetics has been a running theme in software arts discourse (Cox et al., 2000, 2004). Indeed computer programs were first conceived in terms of weaving (§1.1), and perhaps the same is true of writing, as the word *text* is a dead metaphor for cloth:

> An ancient metaphor: thought is a thread, and the raconteur is a spinner of yarns – but the true storyteller, the poet, is a weaver. The scribes made this old and audible abstraction into a new and visible fact. After long practice, their work took on such an even, flexible texture that they called the written page a *textus*, which means cloth. (Bringhurst, 2004, p. 25)

We can also relate natural and computer languages in a more scientific context. There is a great deal of family resemblance between programming and natural languages, in syntax rules, semantics, words, punctuation, and so on. Their similarity is also reflected in statistical measures, for example the words of natural and programming languages both conform to a Zipfian distribution, and comparable long range power law correlation (Kokol and Kokol, 1996).

The Chomsky hierarchy organises language grammars according to their expressive power, in terms of recursion in production rules (Chomsky, 1956). The grammars for programming languages are *context-free*, where production rules specify a single symbol with a corresponding string of symbols which may recursively include *nonterminal* symbols with their own production rules. The grammar rules for natural languages remain an active area of research, but are also considered to be almost entirely context free (Pullum and Gazdar, 1982). In these syntactical terms then, both programming and natural languages have the same expressive power. However language is not defined by its syntax alone, but also its semantics and pragmatics. In §4.2 we will examine this issue from the viewpoint of cognitive semantics, which strongly de-emphasises the role of syntax in language.

Another test of expressive equivalence between languages is *translatability*, however this is somewhat problematic as natural languages have close ties with their modes of expression. For example, a joke may be told in British Sign Language (BSL) that is untranslatable to English, if the joke included reference to similarities between signs. Despite the existence of untranslatable phrases both BSL and English are natural languages with equivalent expressive power (Sutton-Spence and Woll, 1999). Furthermore the same difficulty of translation may occur when taking a spoken joke and trying to write it down using the same language – often jokes are all in timing, intonation or sound symbolism that is lost on the page. The question then is not whether it is possible to find phrases which are not translatable, but to what extent phrases generally are translatable.

Leaving natural languages aside for the moment, translating between computer languages is certainly possible, and may be done either literally or idiomatically. We may translate a program from C to Perl literally, by translating each control flow and data structure, preserving the structure of the original program. Alternatively we may adapt a C program to the idioms of Perl, by using Perl's syntax for data structures and string handling. However because Perl is a higher level language the inverse of translating Perl code to C is more difficult, as C lacks many of Perl's language features. Likewise, it is fairly straightforward to translate from programming languages to natural languages, and indeed this is what we do when explaining how source code operates to another programmer. Translating from natural languages to computer languages is however much more challenging.

"When Joana Carda scratched the ground with the elm branch all the dogs of Cerbère began to bark, throwing the inhabitants into panic and terror, because

> from time immemorial it was believed that, when these canine animals that had always been silent started barking, the entire universe was nearing its end." (Saramago, 2000, p. 1)

The above quote stands as the first sentence of the novel *The Stone Raft* by José Saramago, Nobel Laureate for Literature. It was originally written in Portuguese; the above is taken from the authorised English translation. In all his novels, Saramago limited his punctuation only to the comma and full stop, which allowed him to compose long sentences constructed as patterns of sound, with distinctive rhythm (Saramago, 2000, translator's note). He constructed his sentences as a continuous flow, experimenting with timbre and resonance, treating words as sounds. The translator's great challenge then is to translate the *music* of this text as well as the linguistic content. In this case it seems that this difficult task has been achieved to a large degree, as Saramago approved this translation.

We find Peter Naur, the Turing Award winning computer scientist, to be in agreement with Saramago in considering language a primarily spoken form. He makes this point by transcribing one of his talks as it was spoken, and without punctuation, this time using only em dashes to indicate pauses. The following is an excerpt:

> "The other notion – well – language – that is when one talks – and if one says this one will immediately be taken to task – he has not gone to school – doesn't he know – language cannot be a *when* – shame – it must be a thing surely – and we are thus tangled up into the claim that there are concepts denoted by the words" (Naur, 1992a, p. 524).

The similarity between this and Saramago's writing style is striking, although Naur's transcription is at times rather harder to read, with repetitions and transgressions included.[1] But this difficulty is Naur's point, to demonstrate the difference between spoken language and written text. As he tries to get across in the above quote, words do not denote precise, external concepts, which are rather held by individuals as mental imagery. In this he is in agreement with the point of view of cognitive linguists such as Barsalou, Gärdenfors, Lakoff and Paivio related in chapter 2. Language is not a thing that is written down, but rather a habit or activity that is done; meaning is personal, and not containable within grammar rules. Naur asserts that in order to really understand a speaker's words, you must absorb their language on a number of topics, so that you may begin to understand their frames of reference, and pick up the patterns behind what they are trying to say.

Naur (1992b) applies all this to our understanding of programming languages in his paper "Programming Languages are not Languages – Why 'Programming Language' is a Misleading

[1]As an added complication, Naur's original talk was in Danish, although he asserts his translation is useful in representing a talk that *might* have happened.

Designation.", this time in conventionally written English. His conclusion is that Programming Language is a "special, limited part of the linguistic possibilities, deliberately designed to cater for certain limited situations and purposes." From this we understand that while Naur finds that the term *programming language* is misleading, he still believes that programming language is a certain, specialised area of language. This has echoes of Wittgenstein on formalised language:

> "Don't let it bother you that languages (2) and (8) consist only of orders. If you want to say that they are therefore incomplete, ask yourself whether our own language is complete – whether it was so before the symbolism of chemistry and the notation of the infinitesimal calculus were incorporated in to it; for these are, so to speak, suburbs of our language. (And how many houses or streets does it take before a town begins to be a town?) Our language can be regarded as an ancient city: a maze of little streets and squares, of old and new houses, of houses with extensions from various periods, and all this surrounded by a multitude of new suburbs with straight and regular streets and uniform houses." Wittgenstein (2009, 18)

It seems that even though there are ways in which programming languages are not languages, they at least operate within the landscape of language. How then, could Saramago's text, in its speech-like form, be translated into the language suburb of source code? Transliteration is hardly possible, and so we must look to translate particular aspects of the situation represented in the text. One approach would be to model discrete entities and relationships in the text, of the town, its dogs, and the dependency of the existence of the town on the dogs' behaviour. A human reader would then get some sense of what is written in this programming language either by reading it, or perhaps by using it within a larger computer program. Alternatively we could try to capture a sense of expectation and dread within a temporal structure, and output it as sound, thereby creating a musical theme. This kind of translation is analogous to the live coding of sound tracks relating the narrative structure of silent films, practised by Rohrhuber et al. (2005).

We argue then that translation from natural language to programming language is possible, as long as one accepts that languages with strengths designed for particular purposes also have particular limitations. For example, programming languages are designed to be typed and not spoken, and so bringing the spoken prosody of language (§3.4) directly to computer language seems impossible (although see vocable synthesis, §3.5). However, the same is true of some natural languages. Nicaraguan Sign Language is a particularly interesting case, emerging naturally from a Deaf community of young school children over two decades – a span comparable with the overall development of computer language. It has no spoken or indeed written form, but does already have complex, spatial grammar, and of course prosodic rhythm, being based on movements of the body.

Despite the lack of spoken form, code does have its own spatial features, as we will see in later discussion of secondary notation (§5.1). A programmer also has choice about how to express an algorithm, to a degree depending on the language in hand. A strong example is Perl, which has the rather unwieldy motto There is More Than One Way To Do It (TMTOWTDI). Indeed Larry Wall, the creator of Perl, has a background in linguistics, and lists several features that Perl borrows from natural language in his 1995 on-line essay "Natural Language Principles in Perl". As well as the freedom of TMTOWTDI, it includes Perl's manner of learning, its ambiguity, import of features from other languages, topicalisation and pronominalisation.

Perhaps the key difference between programming and natural language is that the former is formalised and abstract, whereas the latter has developed with a closer relationship to its speakers. Words in a natural lexicon are grounded in human experience of movement and relationships of the body in its environment. Computer languages are not based around these naturally developed words, but we may still maintain the same semantic references for human readers, by using those words in the secondary notation of function and variable names, or even by working with an encoded lexicon as data (such as WordNet; Fellbaum, 1998). In doing so we borrow from the lexicon of a natural language, but we could just have easily used an invented lexicon such as that of Esperanto. Regardless, a computer program is ultimately grounded in the outside world when it is executed, through whatever modalities its actuators allow, usually images, sound and/or movement. At the point which a program is executed, it becomes clear that the its source code is full of a particular kind of linguistic reference known as *performative utterances*. Due to the power that humans wield over computers to do their bidding, by describing an action in a computer language, we cause it to be performed.

Natural and computer languages are developed under different pressures with very different results. However they have sufficient family resemblance to both be considered to be aspects of human language activity. By appraising the differences between natural and computer languages, we may look for ways in which features of natural languages could be appropriated for use in programming languages for the arts, an approach we have demonstrated with Vocable Synthesis (§3.5).

4.2 Music, Language and Cognitive Semantics

We have already seen some of the complex relationship between music and language, while examining integration between the dual codes of language and mental imagery in chapter 2, and arguing in chapter 3 that timbral and prosodic articulation share a grounding in movement. The Chomskian linguistic notion of semantics as it is generally understood excludes any notion of musical meaning. The hierarchical structure of music holds some similarity to linguistic syntax, but the lack of reference to a real or imagined world makes a semantics of music untenable (Wiggins, 1998). As a result, discussion of meaning in music often lacks a formal underpinning, resulting in a broad spectrum of parallel discussions in the literature, each under its own terms (for a broad review see Cross and Tolbert, 2008). However the alternative view of *conceptual semantics* provided by the theory of Conceptual Spaces (§2.2.5) puts things in a rather different light. In the following we show that unlike Chomskian semantics, Gärdenforsian conceptual semantics is applicable to music as well as language, by summarising its main tenets in relation to music.

"Semantic elements are constructed from geometrical or topological structures (not symbols that can be composed according to some system of rules)." In other words, semantic meaning is primary to the conceptual, analogue level, and not the discrete level as with Chomskian semantics. The common view in music theory characterises musical structure as being discrete and syntactic (Lerdahl and Jackendoff, 1983), however conceptual semantics allow us to consider music structure as spatial and geometric in addition. In the case of music of timbre (§3.3.3), we claim that conceptual semantics is the primary structure.

"Semantic meaning is a conceptual structure in a cognitive system." Meaning does not exist through links to the world (or a possible world), but in the body (and in particular, the brain), of an individual. However the conceptual structures of a group of individuals may reach accordance through communication. We can consider a musical improvisation in terms of such a process of communication, where two or more improvisers begin with individual conceptual structures at the beginning of a piece, which are manipulated towards accordance and discordance during a performance. Of course while situated in an individual, the conceptual structures are informed by previous performances and higher cultural effects such as musical genre. It is also possible, through aberration or inference, that new conceptual structure is created during an improvisation that did not exist at the beginning (Wiggins, 2006b). Such conceptual structure could be deemed valuable and kept for reuse in future improvisations. In such a case we can say that an improvisation created new meaning, and was therefore a particularly *creative* performance. This manner of creative search is discussed in greater detail in §6.3.

"Conceptual structures are embodied (meaning is not independent of perception or of bodily experience)." This tenet connects cognitive semantics to its roots in theories of embodied cognition. Instruments and the voice require movements of the body in order to make sound, and the constraints and vagaries of motor action within the tight feedback loop of action and reaction are an important component of musical improvisation (Pressing, 1984, 1987). However the influence of bodily experience goes beyond actual motor action to suggest that semantic meaning is dependent on motor and perceptual circuits in the brain.

"Cognitive models are primarily image-schematic (not propositional). Image-schemas are transformed by metaphoric and metonymic operations (which are treated as exceptional features on the traditional view)." Image schemata are abstract diagrams of spatial relationships and actions, representing notions such as 'over', 'containment' and 'attraction' (Lakoff, 1997). A metaphorical operation is where two concepts are related via common image schemata, most commonly relative to orientations such as 'UP' (§2.2.6). It is through metaphorical structure that a conceptual system can be grounded in perception and action, yet represent meaning abstract from it.

"Semantics is primary to syntax and partly determines it (syntax cannot be described independently of semantics)." This is another tenet in opposition to the widely held view of Chomskian linguistics, where syntax is primary and independent of semantics. It implies that when composing a piece of music, the cognitive semantic structure is more important than grammatical rules. That is, any grammatical rules underlying a piece of music are placed in support of the geometry of the semantic structure, rather than a precursor for it (Forth et al., 2010).

"Concepts show prototype effects (instead of following the Aristotelian paradigm based on necessary and sufficient conditions)." The Aristotelian paradigm has not been taken seriously for several decades and current theories of concepts do not depend upon it (Murphy, 2002, p.16). The subscript to this tenet therefore is weakened by not showing consideration for theories competing with the prototype view such as those of the exemplar view and knowledge approach (Murphy, 2002, pp. 41–71). However prototype effects, such as a robin being judged a more typical bird than a penguin, are indeed easily accounted for within the theory of conceptual spaces. Gärdenfors does so using the Voronoi diagram (Okabe et al., 2000), where a conceptual prototype is a Voronoi generator for the geometrical regions of conceptual properties. Prototype effects are observed in music, for example where pieces are judged as greater or less typical examples of a musical genre. In the Voronoi diagram of genres, a typical piece would be near to the region's generator, and a difficult to define or 'crossover' piece would be near a boundary between two or more genres. In practice, musical genres are impossible to define universally, which points again to the relativist position stated in the first tenet.

In summary, in Chomskian terms music cannot be understood in terms of semantics, only syntax. However if one is prepared to take a Gärdenforsian view, a discussion of musical meaning can proceed with a formal underpinning, where meaning exists within individuals' conceptual structures of music, within the structures shared by the members of a music culture, and within the grounding relationships between musical structure and universals of human perception and movement.

4.3 Declarative vs Imperative

Declarative and imperative programming are competing paradigms, an opposition that raises interesting issues for the time-based computer arts. Declarative programming is the coding of *what* should happen, and imperative programming is the coding of *how* something should occur. This distinction is frequently used, and gives a sense of what is meant, but lacks practical detail. This situation is reminiscent of that of the definition of timbre in music §3.3; an important distinguishing feature within a domain, that is often defined in vague and sometimes conflicting terms.

The use of computer language can be broadly divided into the description of algorithmic control, and of problem logic (Kowalski, 1979). Imperative programming languages support the former, where programmers describe a method for solving a problem, rather that the problem itself. Declarative programming languages support the latter, allowing programmers to focus on the description of problems, by leaving the interpreter to find the algorithmic solution. Well-known examples of declarative programming languages include the logic language Prolog, the database query language SQL, and the string matching language Regular Expressions (ch. 2). These are as close as we get to the declarative promise, of programmers concerning themselves only with what they want to do, and not how it should be done. This promise holds for simple examples, but unfortunately hardly at all in practice. SQL database query optimisers are in constant development, and programmers must keep abreast of exact implementation details, structuring their queries and indexing their datasets in a way that allows their SQL queries to operate efficiently. Upgrading to a new version of an SQL engine is then a serious matter, as optimisations for general cases may have led to serious regressions in edge cases. Likewise, regular expressions must be carefully crafted for a particular interpreter, where implementation details such as backtracking or determinism can impact computational complexity by several orders of magnitude (Friedl, 2006). Prolog programmers have much the same problem, classically requiring manual search space reduction by inhibiting backtracking, a technique known as the *cut* (Sterling and Shapiro, 1994).

While the imperative frequently intrudes, declarative programming styles nonetheless maintain a certain clarity of expression, giving high level descriptions largely separate from implementation details. This is related to the idea of purity in *pure functional* languages such as ML (Milner et al., 1990), where functions have no side-effects. This means a pure functional program cannot interact with the outside world while it is executing, beyond the singular, orderly flow of input and output. Haskell is a pure functional programming language from the ML family, but gets around this limitation through modelling side-effects by chaining together pure functions. The result is a declarative description of an imperative program, which the interpreter then takes care of executing.

Another distinction made between declarative and imperative programming (e.g. Dijkstra, 1985) is concerned with the passing of time. In declarative programming, time is a concern of optimisation, to be separated as much as possible from the problem description. In imperative programming one statement follows another, describing a sequence of operations, each with its own time 'cost'. In the general case this makes a great deal of sense; the imperative 'how' approach is concerned with algorithms as programmes of work over time, and the declarative 'what' approach with logical relationships abstract from time. However within our theme of the design of programming languages for artists, the focus on time is problematic. In particular, when we consider the time-based arts, the unfolding of an algorithm over time is *both* the imperative how *and* the declarative what. In this case, the distinction between declarative and imperative programming appears to not apply, as the particular operation of how an algorithm works has strong influence over what we experience. If the distinction is to make any sense in this context, it must be defined carefully.

We argue that a declarative approach to programming is indeed desirable for artists. We define it however not in general terms, or even relative to algorithms, but as the **closeness of mapping** (§5.1) of the programming notation to the target domain. If a programmer is only interested in logic, then a declarative approach is to choose a programming language that takes care of how a program is solved over time. However if our programmer is interested in the problem of how to complete a piece of music, then they are concerned with how the solution is arrived at as well as the problem; the how *is* the what. A declarative approach in this case would then be concerned with how events are ordered to reach a musical resolution. This does not however mean precisely specifying an algorithm, certain aspects of operation may be musically inconsequential, and so may be specified in an ambiguous manner.

Our characterisation of declarative programming is closely related to the concept of Domain Specific Language (DSL) (van Deursen et al., 2000). To a limited extent, a straightforward programming library is DSL, in that it provides functions particular to a domain. However in a fuller sense, DSL goes further to provide higher order means of abstraction and combination tailored to the target domain, allowing programmers to find the level of abstraction that best mirrors the structure of their problem. For example, a computer musician working with patterns may create language for combining functions which represent generation and transformation of pattern. We will approach this subject in depth in the following section, before introducing Tidal, DSL for live coding of pattern.

4.4 Domain Specific Language for Pattern

When we view the composed sequence "abcabcabc…" we quickly infer the pattern "repeat abc". This inference of hierarchy is known in the psychological literature as 'chunking' (Miller, 1956), and aids memory of long sequences, prediction of future values and recognition of objects. Pattern pervades the arts; as Alfred Whitehead (2001) eloquently puts it, "Art is the imposing of a pattern on experience, and our aesthetic enjoyment is recognition of the pattern."[2] This communicates a role of pattern supported here; one individual encodes a pattern and another decodes it, both actively engaged with the work while creating their own experience. In the present section we examine the encoding of pattern in particular, introducing Tidal, a computer language for encoding musical patterns during improvised live coding (§6.8) performances.

Pattern has been of great interest throughout the history of art. The patterned walls and floors of the Alhambra in Spain are an extraordinary example, where Moorish artists have captured all seventeen types of symmetry, centuries before their formalisation by group theorists (du Sautoy, 2008). The technological exploration of musical pattern also has a long history, extending back to well before electronic computers. For example, Leonardo da Vinci invented a hurdy gurdy with movable pegs to encode a sequence, played back using multiple reeds at adjustable positions, transforming the sequence into a canon (Spiegel, 1987). Hierarchies of repeating structure run throughout much of music theory; computational approaches to music analysis, indexing and composition all have focus on discrete musical events and the rules to which they conform (Rowe, 2001, §4.2). From this we infer that the encoding and decoding of pattern is fundamental to music making. We review support given to musical pattern making by computer language in this light.

The literature on pattern DSLs (Domain Specific Languages) is mainly concerned with *analysis* of composed works relative to a particular theory of music. For example Simon and Sumner (1992) propose a formal language for music analysis, consisting of a minimal grammar

[2]To our shame, these words were background to Whitehead lambasting those taking quotes out of context.

for describing phrase structure within periodic patterns. Their language allows for multidimensional patterns, where different aspects such as note value, onset and duration may be expressed together. The grammar is based on a language used for description of aptitude tests which treat pattern induction as a correlate with intelligence. Deutsch and Feroe (1981) introduced a similar pattern DSL to that of Simon and Sumners, for the analysis of hierarchical relationships in tonal music with reference to gestalt theory of perception (Kohler, 1930).

The analytical perspective shown in the pattern DSLs discussed thus far puts focus on simple patterns with unambiguous interpretation. We assert that music composition demands complex patterns with many possible interpretations, leading to challenged, engaged listeners in a state of flow (see analytic listening; Csikszentmihalyi, 2008, p. 111, and our discussion on timbre, §3.3.3). Therefore pattern DSL for synthesis of music requires an approach different from formal analysis. Motivation for the design of pattern DSL for music composition is identified by Laurie Spiegel in her paper "Manipulations of Musical Patterns" (Spiegel, 1981). Twelve classes of pattern transformation, taken from Spiegel's own introspection as a composer are detailed: transposition (translation by value), reversal (value inversion or time reversal), rotation (cycle time phase), phase offset (relative rotation, e.g. a canon), rescaling (of time or value), interpolation (adding midpoints and ornamentation), extrapolation (continuation), fragmentation (breaking up of an established pattern), substitution (against expectation), combination (by value – mixing/counterpoint/harmony), sequencing (by time – editing) and repetition. Spiegel felt these to be 'tried and true' basic operations, which should be included in computer music editors alongside insert, delete and search-and-replace. Further, Spiegel proposed that studying these transformations could aid our understanding of the temporal forms shared by music and experimental film, including human perception of them.

Pattern transformations are evident in Spiegel's own Music Mouse software, and can also be seen in music software based on the traditional studio recording paradigm such as Steinberg Cubase and Apple Logic Studio. However Spiegel is a strong advocate for the role of the musician-programmer, and expresses hope that these pattern transformations would be formalised into programming libraries. Such libraries have indeed since emerged. Hierarchical Music Specification Language (HMSL) developed in the 1980s, and includes an extensible framework for algorithmic composition, with some inbuilt pattern transformations. The Scheme based *Common Music* environment, developed from 1989, contains an object oriented pattern library (Taube, 2004); classes are provided for pattern transformations such as permutation, rotation and random selection, and for pattern generation such as Markov models, state transition and rewrite rules. The SuperCollider computer music language (McCartney, 2002) also includes an extensive pattern library, benefiting from an active free software development community, and with advanced support for live coding (§6.8). What all these systems have in common is a desire to represent the structure of music in the structure of code. As such, they adhere closely to our definition of declarative language (§4.3), working at the same level of abstraction of the target domain of musical pattern. Inspired by this, we now move to introduce our own pattern DSL, *Tidal*.

4.5 Tidal

Tidal is a pattern DSL embedded in the Haskell programming language, consisting of pattern representation, a library of pattern generators and combinators, an event scheduler and programmer's live coding interface. This is an extensive re-write of earlier work introduced under the working title of *Petrol* (McLean and Wiggins, 2010). Extensions include improved pattern representation and fully configurable integration with the Open Sound Control (OSC; Freed and Schmeder, 2009) protocol.

4.5.1 Features

Before examining Tidal in detail we first characterise it in terms of features expected of a pattern DSL.

Host language Tidal is a Domain Specific Language embedded in the Haskell programming language. The choice of Haskell allows us to use its powerful type system, but also forces us to work within strict constraints brought by its static types and pure functions. We can however turn this strictness to our advantage, through use of Haskell's pioneering type-level constructs such as applicative functors and monads. Once the notion of a pattern is defined in terms of these constructs the expressive power of Haskell's syntax becomes available, which can then be explored for application in describing musical pattern. Haskell's syntax is very terse, thanks in part to its declarative approach (§4.3) and notational conveniences. For example in Haskell all functions are implicitly *curried*, that is all Haskell functions only take a single argument, but may return another function that takes a further argument and so on. This allows partial application to be very tersely expressed, for example we may specialise the function `+` to `+ 1`, and map the resulting function over a list of values without explicit use of lambda; `map (+1) [1, 2, 3]`. Such tersity removes barriers from the expression of ideas, and therefore allows a tighter creative feedback loop (§6.3).

Pattern composition In Tidal, patterns may be composed of numerous sub-patterns in a variety of ways and to arbitrary hierarchical depth, to produce complex wholes from simple parts.

This could include concatenating patterns time-wise, merging them so that they co-occur, or performing pairwise operations across patterns, for example combining two numerical patterns by multiplying their values together. Composition may be heterarchical, where sub-pattern transformations are applied at more than one level of depth within a larger pattern.

Time representation Time can be conceptualised either as *linear change* with *forward order* of succession, or as a repeating cycle where the end is also the beginning of the next repetition (Buzsaki, 2006). We can relate the former to the latter by noting that the phase plane of a sine wave is a circle; a sine wave progresses over linear time, but its oscillation is a repeating cycle as shown in Figure 7.1. Music exhibits this temporal duality too, having repeating rhythmic structures that nonetheless progress linearly. Tidal allows both periodic and linear patterns to be represented.

Another important distinction is between discrete and continuous time. In music tradition, time may be notated within discrete symbols, such as Western staff notation or Indian Bol syllables, but performed with subtle phrasing over continuous time. Tidal maintains this duality, where patterns are events at discrete time steps. However phrasing may be specified as patterns of floating point onset time deltas (§4.6.1).

Random access Both Common Music and SuperCollider represent patterns using lazily evaluated lists, where values are calculated one at a time as needed, rather than all together when the list is defined. This allows long, perhaps infinitely long lists to be represented efficiently in memory as generator functions, useful for representing fractal patterns for example. In some languages, including Haskell, lists are lazily evaluated by default, without need for special syntax. This is not how patterns are represented in Tidal however. Lazy lists are practical for linear evaluation, but you cannot evaluate the 100th value without first evaluating the first 99. This is a particular problem for live coding (§6.8); if you were to change the definition of a lazy list, in order to continue where you left off you must regenerate the entire performance up to the current time position.[3] Further, it is much more computationally expensive to perform operations over a whole pattern without random access, even in the case of straightforward reversal.

Tidal allows for random access by representing a pattern not as a list of events but as a function from time values to events. A full description is given in §4.5.2.

[3]SuperCollider supports live coding of patterns using PatternProxies (Rohrhuber et al., 2005). These act as placeholders within a pattern, allowing a programmer to define sub-patterns which may be modified later.

Ready-made generators and transforms A pattern library should contain a range of basic pattern generators and transforms, which may be straightforwardly composed into complex structures. It may also contain more complex transforms, or else have a community repository where such patterns may be shared. Tidal contains a range of these, some of which are inspired by other pattern languages, and others from Haskell's standard library of functions, including its general support for manipulating collections.

Community

> "Computers're bringing about a situation that's like the invention of harmony. Sub-routines are like chords. No one would think of keeping a chord to himself. You'd give it to anyone who wanted it. You'd welcome alterations of it. Sub-routines are altered by a single punch. We're getting music made by man himself: not just one man." John Cage (1969)

John Cage's vision has not universally met with reality; much music software is proprietary, and in the United States sound synthesis algorithms are impeded by software patents. However computer music languages are judged by their communities, sharing code and ideas freely, particularly around languages released as free software themselves. A pattern DSL then should make sharing abstract musical ideas straightforward, so short snippets of code may be easily used, modified and passed on. This is certainly possible with Tidal, although this is a young language which has not yet had a community grow around it.

4.5.2 Representation

In Tidal, patterns are represented by the `Pattern a` data type, which is defined as follows:

```
data Pattern a =
  Pattern {at :: Behaviour a, period :: Period}

type Behaviour a = Int → [Maybe a]

type Period = Maybe Int
```

This `Pattern a` data type is composed of two further types; `Behaviour a` which is a type synonym for `Int →[Maybe a]`, and `Period` which is a type synonym for `Maybe Int`. They are accessible via their field names `at` and `period` respectively. The *behaviour* represents the structure of the pattern, and the *period* how often it repeats, if at all.

The name of the `Behaviour a` type is taken from functional reactive programming nomenclature (Elliott, 2009; Hudak, 2000), a *behaviour* being the term for a time-varying value. Its form `Int →[Maybe a]` is a function from `Int` (integer), to a list of values of type `Maybe a`.

The integer function parameter represents discrete time steps, and the result of the function, `[Maybe a]` represents a list of events. In other words, a behaviour represents a pattern as a time-varying list of events.

`Pattern` and `Behaviour` share the polymorphic type parameter `a`, which is encapsulated within `Int →[Maybe a]`. The type parameter `a` is specialised to whatever the event type of a particular pattern is, for example a pattern of musical notes could be of type `Pattern String`, where pitch labels are represented as character strings, or alternatively of type `Pattern Int` for a pattern of MIDI numbered note events.

We have not yet explained the purpose of the `Maybe` type which encapsulates `a`. The `Maybe` type is a standard feature of Haskell, and has two constructors, namely `Just a` and `Nothing`. The reason for using it here is to allow events which do not have a value in terms of `a` to be represented as `Nothing`. This is particularly useful for representing musical *rests*. Further motivation is shown in §4.6, where `Nothing` is shown to have different meaning in different situations.

As we see in the above definition, the *period* of a pattern – the duration at which it repeats – is represented as an `Int`. It is also encapsulated within the `Maybe` type, used for specifying aperiodic patterns. The periodic pattern "*abcdefgh*, repeated" would have a `Period` of `Just 8`, and the aperiodic pattern "*a* followed by repeating *b*s" would have a `Period` of `Nothing`.

The following example shows how a pattern may be constructed, in this case the integer pattern of the repeating sequence "0, 2, 4, 6":

```
p = Pattern {at = λn → [Just ((n `mod` 4) * 2)],
             period = Just 4}
```

To map from the time parameter to this trivial sequence, the `Behaviour` simply takes the modulo of four, and multiplies it by two.

We access events in a pattern by evaluating its `Behaviour` function with a time value. Continuing with the pattern defined above, `at p 1` evaluates to `[Just 2]`. As this is a cyclic pattern of period 4, `at p 5` would give the same result, as would `at p (-3)`.

The above pattern is expressed as a function over time. An alternative, recursive definition would be more idiomatic to Haskell, in this case defining `at p 0` to return `Just [0]` and subsequent `at p n` to return the value at `n - 1` plus two. However great care must be taken when introducing such dependencies between time steps; it is easy to produce incomputable patterns, or as in this case, patterns which may require whole patterns to be computed to find values at a single time point.

4.5.3 Pattern generators

A pattern would not normally be described by directly invoking the constructor in the rather long-winded manner shown in the previous section, but by calling one of the pattern generating functions provided by Tidal. These consist of generators of basic repeating forms analogous to sine, square and triangle waves, and a parser of complex sequences. The following is the definition of Tidal's `sine` function, which produces a sine cycle of floating point numbers in the range -1 to 1, with a given period:

```
sine :: Int → Pattern Double
sine l = Pattern f (Just l)
  where f n = [Just ( sin (
                 fromIntegral n ∗ (pi / fromIntegral l ∗ 2)
               ))
              ]
```

The `sine1` function does the same as `sine` but in the range from 0 to 1. It is defined relative to `sine`, by using the functor map operator `<$>`. The values of a pattern are mapped over a scaling function in the following definition:

```
sine1 :: Int → Pattern Double
sine1 l = ((/ 2.0) ∘ (+ 1.0)) <$> sine l
```

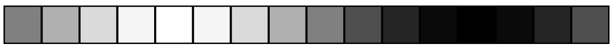
We can use this functor mapping because `Pattern` is an instance of Haskell's `Functor` class, with the following definition:

```
instance Functor Pattern where
  fmap f (Pattern xs p) =
    Pattern (fmap (fmap (fmap f)) xs) p
```

This definition simply maps over each of `Behaviour`'s enclosing types. This is all that needs to be done, because Haskell already defines the function application, `Maybe` and list types as instances of the `Functor` class.

The following is an example of the `sine1` function in use, here rendered as a sequence of grey values by the `drawGray` function:

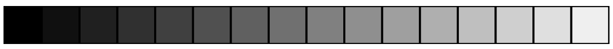
```
drawGray $ sine1 16
```



Tidal is designed for use in live music improvisation, but is also applicable to off-line composition, or for non musical domains. We take this opportunity to illustrate the following examples with visual patterns, in sympathy with the present medium. For space efficiency the above cyclic pattern is rendered as a row of blocks, but ideally would be rendered as a circle, as the end of one cycle is also the beginning of the next.

Linear interpolation between values, somewhat related to musical glissandi, is provided by the `tween` function:

```
drawGray $ tween 0.0 1.0 16
```



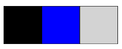
4.5.4 Parsing strings into polymetric patterns

A pattern may also be specified as a string, which is parsed according to the context, as defined by Haskell's type inference. In the previous two examples we have used the drawGray function which requires a pattern of floating numbers, in particular one of type `Pattern Double`. In the following example the `draw` function requires a colour pattern, and so a parser of colour names is automatically employed by the function `p`.

```
draw (p "black blue lightgrey")
```

Thanks to the string overloading extension to Haskell provided by the Glasgow Haskell Compiler, a string is automatically coerced into a Pattern using the function `p`. This is so that `p` does not need to be explicitly specified:

```
draw "black blue lightgrey"
```



The parser allows terse description of polymetric rhythms, inspired by the syntax of the Bol Processor (Bel, 2001). We will first describe the parser in detail before moving on to examples, which the reader may wish to refer to in advance.

The parser rules are implemented using Haskell's *Parsec* library, but are illustrated in the following using Extended Backus-Naur Form (EBNF). We diverge from standard EBNF to parameterise rules, in order to show that the four different parsers operate in the same manner, but with differing rules to parse atomic events within rhythms.

Four different parsers are created by parameterising general parser rules with a rule to match an atom, i.e. a rhythmic event at the lowest level of granularity:

```
colourRhythm         = rhythm(colourname) ;
doubleRhythm         = rhythm(double) ;
stringRhythm         = rhythm(string) ;
intRhythm            = rhythm(int) ;
boolRhythm           = rhythm(bool) ;

colourname           = "blue" | "red" | "green" | ... ;
bool                 = "t" | "1" | "f" | "0" ;
```

These parsers allow Tidal to parse `Pattern`s of the basic types `String`, `Bool`, `Int` and `Double`, as well as `Colour`. It is straightforward to define more as needed.

A rhythm consists of a sequence of rhythmic parts:

```
rhythm(atom)         = whitespace, sequence(atom) ;
sequence(atom)       = { part(atom) } ;
whitespace           = { " " } ;
```

A rhythmic part may consist of a single atom, a rest (denoted by ~), a polymeter, or a sequence of atoms:

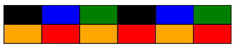
```
part(atom)           = atom | "~" | poly(atom) | polypad(atom)
                       | sequenceParens(atom) ;
sequenceParens(atom) = "(", sequence(atom) , ")" ;
```

Two different methods are provided for combining rhythms into polymeters, denoted with either square or curly brackets.

```
poly(atom)          = "[", rhythm(atom), {",", rhythm(atom)}, "]" ;
polypad(atom)        = "{", rhythm(atom), {",", rhythm(atom)}, "}" ;
```

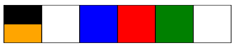
The `poly` rule indicates combination of rhythms by a method of *repetition* and `polypad` by *padding*. In both cases the result is a pattern with the same period, being the lowest common multiple of the constituent pattern periods. We illustrate their operation by example. Firstly, in combination by repetition, the first rhythm is repeated twice and the second thrice:[4]

```
draw "[black blue green, orange red]"
```



Combining by *padding* interleaves the internal structure of a rhythm with rests. In the following example the first part is padded with one rest every time step, and the second with two rests every step:

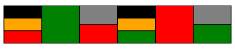
```
draw "{black blue green, orange red}"
```



In the above there are time steps where two events co-occur and the block is split in two, steps where one event takes up the whole block, and steps where no events occur, shown by a blank block.

Polymeters may be embedded to any depth (note the use of a tilde to denote a rest):

```
draw "[{black ~ grey, orange}, red green]"
```



4.5.5 Pattern combinators

Once we have a basic pattern, using either a generator such as `sine` or the above parser, we can transform it with functions, each one adding a layer of rhythmic structure.

If our underlying pattern representation were a list, a pattern transformer would operate directly on sequences of events. For example, we might *rotate* a pattern one step forward by *pop*ping from the end of the list, and *unshift*ing/*cons*ing the result to the head of the list. In Tidal, because a pattern is a function from time to events, a transformer may manipulate time as well as events. Accordingly the Tidal function `<~` for rotating a pattern to the left is defined as:

```
(<~) :: Int → Pattern a → Pattern a
(<~) p n =
  Pattern (λt → at p (t + n)) (period p)
```

[4]Note that co-occurring events are visualised by the `draw` function as vertically stacked colour blocks.

Rotating to the right is simply defined as the inverse:

```
(~>) :: Int → Pattern a → Pattern a
(~>) p n =  p <~  (0 - n)
```

The `append` function appends one pattern to another timewise. If the first pattern is aperiodic, then the second pattern is never reached, so the result is the first pattern unchanged.

```
append :: Pattern a → Pattern a → Pattern a
append a@(Pattern f Nothing) _ = a
```

If the first pattern is periodic, and the second is aperiodic, then the second is appended to a single cycle of the first, and the resulting pattern is aperiodic.

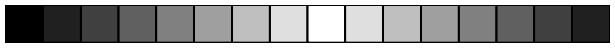
```
append a@(Pattern _ (Just l)) b@(Pattern _ Nothing) =
    Pattern newF Nothing
  where newF n | n < l = at a n
               | otherwise = at b (n - l)
```

If both patterns are periodic, then the resulting pattern alternates between them, with a period that is the sum of those of the constituent patterns.

```
append a@(Pattern f (Just l)) b@(Pattern f' (Just l')) =
    Pattern newF (Just newL)
  where
    newL = l + l'
    newF n | cycleP < l = f ((loopN * l) + cycleP)
           | otherwise = f' ((loopN * l') + (cycleP - l))
      where cycleP = n `mod` newL
            loopN = n `div` newL
```

The following illustrates `append` in use:

```
drawGray $ append (tween 0 1 8) (tween 1 0 8)
```

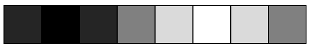


The `cat` function for joining together a list of patterns is trivial to define as a fold over `append`:

```
cat :: [Pattern a] → Pattern a
cat = foldr append nullPattern
```

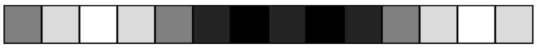
Further combinators are defined for reversing a pattern with `rev`:

```
drawGray $ rev (sine1 8)
```



Or alternatively playing a pattern forwards and then in reverse with `palindrome`:

```
drawGray $ palindrome (sine1 8)
```



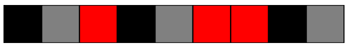
The `every` function allows transformations to be applied to periodic patterns every n cycles. Its definition simply multiplies the period of the given pattern by `n - 1`, then appends the transformed pattern to it:

```
every :: Int → (Pattern a → Pattern a) → Pattern a
           → Pattern a
every 0 _ p = p
every n f p = (p ~* (n - 1)) `append` f p

(~*) :: Pattern a → Int → Pattern a
(~*) p n = Pattern (at p) (fmap (* n) (period p))
```

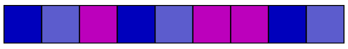
The following demonstrates how the `every` combinator may be used to rotate a pattern by a single step every third repetition:

```
draw $ every 3 (1 ~>) "black grey red"
```



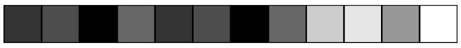
Because the Pattern type is defined as a functor, we may apply a function to every element of a pattern using the `fmap`, or its operator form `<$>`. For example, we may add some blue to a whole pattern by mapping the `blend` function (from the Haskell Colour library) over its elements:

```
p = every 3 (1 ~>) "black grey red"
draw $ blend 0.5 blue <$> p
```



We can also apply the functor map conditionally, for example to lighten the pattern every third cycle:

```
drawGray $
    every 3 ((+ 0.6) <$>) "0.2 0.3 0 0.4"
```



If we were doing something similar to a sound rather than colour event, we might understand it as a musical transposition.

As Haskell is a functional language, it is possible to have higher order patterns, that is patterns of functions. For example, the following would result in a pattern of type `Pattern (Colour →Colour)`, a pattern of color blends alternating between red and blue:

```
(blend 0.5) <$> "red blue"
```

We can hardly visualise a pattern of functions, but such patterns are of use, as we shall see shortly.

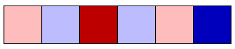
Haskell has a superclass of functor called the *applicative* functor, which defines the `<*>` operator, allowing us to apply functions 'inside' Patterns to other values.[5] Tidal patterns have the following instance definition:

```
instance Applicative Pattern where
  pure x = Pattern (pure (pure (pure x))) (Just 1)
  Pattern fs pf <*> Pattern xs px =
      Pattern (liftA2 (zipCycleA2 (<*>)) fs xs) (lcd pf px)

-- lowest common duration
lcd :: Period → Period → Period
lcd Nothing _ = Nothing
lcd _ Nothing = Nothing
lcd (Just n) (Just n') = Just (lcm n n')
```

The definition of `<*>` allows a new pattern to be composed by taking a function with multiple parameters, and mapping it over combinations of values from more than one pattern. The `<*>` operator is defined for Patterns so that all events are used at least once, and no more than necessary to fulfil this constraint. For example, the following gives a polyrhythmic lightening and darkening effect, by blending colours from two patterns:

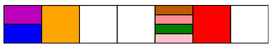
```
draw $ (blend 0.5) <$> "red blue" <*> "white white black"
```



[5]I am grateful to Ryan Ingram for his help with this applicative functor definition.

The Tidal `onsets` function filters events, only allowing through those which begin a phrase. Here we manipulate the onsets of a pattern (blending them with red), before combining them back with the original pattern.

```
draw $ combine [blend 0.5 red <$> onsets p, p]
  where p = "blue orange ~ ~ [green, pink] red ~"
```



The `onsets` function is particularly useful in cross-domain patterning, for example taking a pattern of notes and accentuating phrase onsets by making a time onset and/or velocity pattern from it.

4.6 Open Sound Control patterns

Tidal has no capability for sound synthesis itself, but instead represents and schedules patterns of Open Sound Control (OSC; Freed and Schmeder, 2009) messages to be sent to a synthesiser. Below we see how the 'shape' of an OSC message is described in Tidal:

```
synth = OscShape {path = "/trigger",
                  params =
                    [ F "note" Nothing,
                      F "velocity" (Just 1),
                      S "wave" (Just "triangle")
                    ],
                  timestamp = True
                 }
```

This is a trivial `"/trigger"` message consisting of two floating point parameters and one string parameter. Each parameter may be given a default value in the `OscShape`; in this case velocity has a default of `1`, wave has a default of `"triangle"` and note has no default. This means that if an OSC pattern contains a message without a note value set, there will be no value to default to, and so the message will be discarded. Pattern accessors for each parameter are defined using names given in the `OscShape`:

```
note     = makeF synth "note"
velocity = makeF synth "velocity"
wave     = makeS synth "wave"
```

4.6.1 Scheduling

As `timestamp` is set to `True` in our OscShape example, one extra pattern accessor is available to us, for onset deltas:

```
onset = makeT synth
```

This allows us to make time patterns, applying subtle (or if you prefer, unsubtle) expression. This is implemented by wrapping each message in a timestamped OSC bundle. A simple example is to vary onset times by up to 0.02 seconds using a sine function:

```
onset $ (∗ 0.02) <$> sine 16
```

The `$` operator does nothing except apply the right hand side to the function on the left. However it has very low precedence, and so is useful for removing the need for parenthesis in cases such as this.

Instances of Tidal can synchronise with each other (and indeed other systems) via the NetClock protocol (`http://netclock.slab.org/`). NetClock is based upon time synchronisation in SuperCollider (McCartney, 2002). This means that time patterns can notionally schedule events to occur in the past, up to the SuperCollider control latency, which has a default of 0.2 seconds.

It is also possible to create tempo patterns to globally affect all NetClock clients, for example to double the tempo over 32 time steps:

```
tempo $ tween 120 240 32
```

4.6.2 Sending messages

We connect our OSC pattern to a synthesiser using a `stream`, passing the network address and port of the synthesiser, along with the `OscShape` we defined earlier:

```
s ← stream "127.0.0.1" 7770 synth
```

This starts a scheduling thread for sending the messages, and returns a function for replacing the current pattern in shared memory. Patterns are composed into an OSC message Pattern and streamed to the synthesiser as follows:

```
s $ note ("50 ~ 62 60 ~ ~")
    ~~ velocity foo
    ~~ wave "square"
    ~~ onset ((* 0.01) <$> foo)
  where foo = sine1 16
```

The ~~ operator merges the three parameter patterns and the onset pattern together, into a single OSC message pattern. This is then passed to the stream `s`, replacing the currently scheduled pattern. Note that both `velocity` and `onset` are defined in terms of the separately defined pattern `foo`.

4.6.3 Use in improvisation

Music improvisation is made possible in Tidal using the dynamic Glasgow Haskell Compiler Interpreter (`http://www.haskell.org/ghc/`). This allows the musician to develop a pattern over successive calls, perhaps modifying the preceding listing to transpose the note values every third period, make a polyrhythmic pattern of wave shapes, or combine multiple onset patterns into a chorus effect. Tidal provides a mode for the iconic emacs programmer's editor (`http://www.gnu.org/software/emacs/`) as a GHCI interface, allowing patterns to be live coded within an advanced developers' environment.[6]

As well as to music, Tidal has also been applied to the domain of live video animation, in support of the musician Kirk Degiorgio in Antwerp, May 2010. This was inspired by the art of colour play developed by Mary Hallock-Greenewalt (§2.5), and focused on colour transitions and juxtapositions between colours. Patterns of colour transitions are specified in a manner analogous to musical events, with something like chords of colour provided by splitting the display into vertical bars. For example in order to move from three colours to two, four transitions are performed, as the central colour is split in two. Patterns may be applied to the three parameters of hue, transition type (either linear or sinusoidal) and transition speed. For the performance, beat tracking was employed so that the colours changed in time with the regular pulse of the music. The performance was not recorded or formally evaluated, but serves as an example of the applicability of Tidal to the visual domain.

4.6.4 Future directions

We have introduced Tidal, a language designed for live coding of musical pattern. Tidal has already been field tested through several performances, including to large audiences at international music festivals, informing ongoing development of the system. Although it is designed

[6]Projecting the emacs interface as part of a live coding performance has its own aesthetic, having a particularly strong effect on many developers in the audience, either of elation or revulsion.

for direct use, Tidal provides an ideal base on which to build experimental user interfaces. For example in §5.6, we will introduce a visual programming interface, with Tidal providing the underlying pattern DSL.

As noted in §4.3 time is in general not well supported in programming languages. In Tidal we have represented time in a function from discrete time to events, which works well within electronic dance music genres with minimal deviation from a steady pulse. While Tidal does allow expression through patterns of time, in a broader context however its sole focus on sound event onsets is a huge constraint on music expression. Following the Functional Reactive Programming literature to representing time with real or rational numbers is compelling. This would be towards supporting representations of time based not just on fitting events on to grids, but also arranging sounds relative to each other. That is, supporting smooth as well as striated time (§2.1). This would draw upon extensive work by Bel (2001) on constraint-based time setting.

4.7 Discussion

Language is a central issue in programming languages for the arts, and has taken a central position in the structure of the present thesis. It is of course a broad subject, within which we have visited a number of sub-themes. We have compared programming languages with natural languages, taking a view from Wittgenstein of both existing on the same landscape. We have taken a cognitive view of semantics and related it to spatial relationships in music, allowing us to take an unconventional position of relating meaning in music with that of natural language. We have made a distinction between declarative and imperative approaches to programming that focuses on levels of abstraction, where a declarative approach is one that matches the level of abstraction of the target domain. This has finally led us into the subject of linguistic descriptions of patterns of experience, and the introduction of Tidal, a pattern DSL for improvising music. Tidal provides a system allowing artists to potentially work at a level which is abstract from the surface of the target medium, in this case sound. This does not necessarily distance them from their work, but rather allows them to work directly with the structure of a piece, at a higher level of composition.

We draw these strands together to make the point that programming languages are human languages, for communicating with computers but also other programmers and with ourselves. This will be developed further in the following chapters, in focusing on the notation of computer programs (ch. 5) and their use in creative processes (ch. 6). Already though the part that the design of programming languages can have in human expression is clear, in allowing the development of languages that accord with the structure of our target medium. This promise has led many artists to be hooked by programming, and the wider public perception of programming as a technical endeavour may at last be lifting as communities of artist-programmers grow. From here we look for how programming languages, their notation and use may be reframed in response.

Chapter 5

Notation

Building upon our discussion of the syntax and semantics of language, we now examine programming language environments as designed user interfaces. Study of the notation of computer programs brings Applied Psychology to Human-Computer Interaction (Blackwell, 2006b), allowing us to take a human-centric view of the activity of programming. Formally speaking, all mainstream programming languages are Turing-complete, and so have equivalent expressive power (§2.2). However, in practice the design of programming notation makes certain ways of working easy and others difficult, or practically impossible. We argue that as the result of design processes over the last fifty years, computer languages have become specialised towards use in industry to the detriment of more freeform use. These constraints have become so embedded in languages and their use that it has become difficult to see beyond them. This is reflected in the way we describe mainstream languages as *general purpose*, assumed to be suitable for any purpose, or in other words, none in particular. However we contend that these languages are unsuitable for smaller scale, human-level domain of the artist-programmer.

In the present chapter we will examine and compare existent notations using the Cognitive Dimensions of Notation framework, briefly reviewed below. We will focus in particular on the use of time and space in notations, which we will assert is of particular concern to the arts. This will lead into the introduction of Texture, a visual programming language with novel use of Euclidean distance in the syntax of a pure functional language. This will provide foundations for the following chapter, in which will examine how the use of programming notations impact upon creative processes.

5.1 Cognitive Dimensions of Notation

The Cognitive Dimensions of Notation framework is designed to aid discussion of programming language design (Blackwell and Green, 2002). Rather than a checklist of good or bad design, it describes a set of features which may be desirable or not, and which interact with each other depending on the context. Further more they are known as *dimensions* because they are not absolutes but scales. An example of a common trade-off is one between the **viscosity**[1] dimension, how difficult it is to modify a program, and the provision for **secondary notation**. If we change a notation to extend the role of secondary notation, then **viscosity** will also increase, as the secondary notation must be changed to match any syntax change. A list of cognitive dimensions with short descriptions is shown in Figure 5.1.

Table 5.1: *Cognitive Dimensions of Notation, adapted from Church and Green (2008)*

Abstraction	Availability of abstraction mechanisms.
Hidden dependencies	Invisibility of important links between entities.
Premature commitment	Constraints on the order of doing things.
Secondary notation	Notation other than formal syntax.
Viscosity	Resistance to change.
Visibility	Visibility of components.
Closeness of mapping	Closeness of representation to target domain.
Consistency	Similar semantics are expressed in similar syntactic forms.
Diffuseness	Verbosity of language.
Error-proneness	Likelihood of mistakes.
Hard mental operations	Demand on cognitive resources.
Progressive evaluation	Temporal relationship between edits and their evaluation.
Provisionality	Degree of commitment to actions or marks.
Role-expressiveness	Extent to which the purpose of a component may be inferred.

The cognitive dimensions exhibit many inter-dependencies. These trade-offs are not fully formalised, and it is presumed that they differ depending on the task at hand. Indeed a dimension may be desirable in one context, but undesirable in another. For example an increase in **viscosity**, making programs more difficult to change, is generally seen as having negative impact. However Beckwith and Burnett (2004) report that increasing **viscosity** of certain notational aspects improved the performance of a class of subjects using their language.

[1]To clarify discussion, we will use a particular typeface to indicate where we refer to a particular dimension, for example as with **hidden dependencies**.

The Cognitive Dimensions of Notation are particularly useful in considering the design of Domain Specific Languages (DSLs), by providing standard terms for describing the particular demands of a task domain. In the following we apply it to the notation of time in the domain of live music improvisation.

5.2 Notation in Time

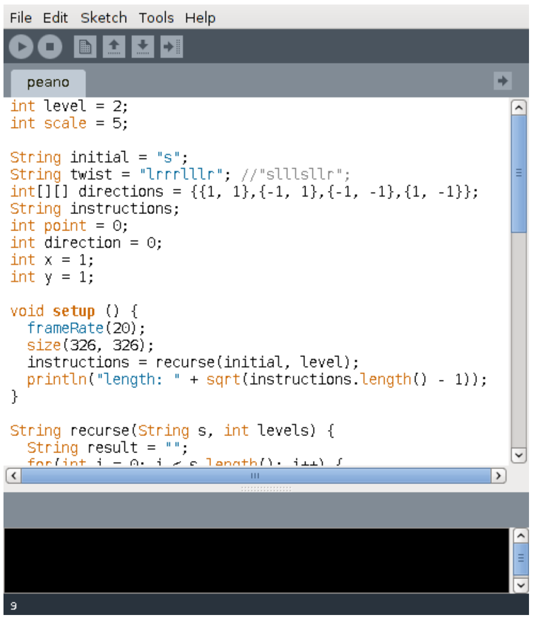
We have already seen problems that the march of time presents to computer language in the previous chapter (§4.3). It can be useful to separate the declarative *what* from the imperative *how*, but when the behaviour of an algorithm over time is important to the aesthetics of a digital artwork, the how *is* the what. We introduced Tidal, which illustrates this point well, being a pattern DSL solely concerned with the notation of events in time (§4.5).

Designers of conventional programming languages consider time in terms of efficiency, and as something to be saved or spent. For those designing languages for the time based arts, time is rather part of the structure of the experience that is being represented, a stratum of both digital pattern and analogue movement.

There is however an awkwardness about the relationship between the *what* of notation and *how* of the passage of time. On one level they seem separate: a declarative description of a time structure is one thing, and an imperative algorithm for accurately and efficiently producing that structure may be quite another. But in practice they rarely are completely separate, the algorithm leaves its imprint, patterning its output. On recognising the 'glitches' arising from the operation of a particular algorithm, artists bring them into the music itself (Cascone, 2000).

Time may always be relied upon to pass, and it passes not only during a program's interpretation but during the activity of notation itself. This is of concern to programming languages designed for the arts, where the separate timelines of development and execution effectively separates an artist from their target medium. Figure 5.1 shows the Processing programming environment, which is an adaptation of the general purpose Java programming language, and is designed for artists who are learning programming. To this end the interface has few distractions, and makes running a program very straightforward: the programmer simply presses the 'play' button marked with an icon familiar from music equipment. Nonetheless, the programmer must stop writing code, press the button and wait a little while before seeing the results of their work. This is not wholly bad, and may be thought in terms of an artist taking a step back from their work to pause in consideration. However, such periods of reflection should be taken on an artist's own terms, and not forced upon them by an external process.

Live coding is a movement that has emerged to investigate the removal of the compilation



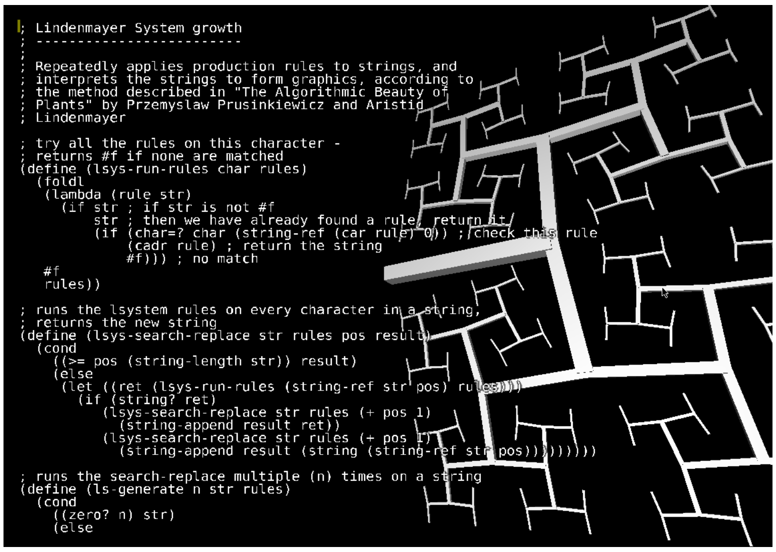
```
int level = 2;
int scale = 5;

String initial = "s";
String twist = "lrrrlllr"; //"slllsllr";
int[][] directions = {{1, 1},{-1, 1},{-1, -1},{1, -1}};
String instructions;
int point = 0;
int direction = 0;
int x = 1;
int y = 1;

void setup () {
  frameRate(20);
  size(326, 326);
  instructions = recurse(initial, level);
  println("length: " + sqrt(instructions.length() - 1));
}

String recurse(String s, int levels) {
  String result = "";
  for(int i = 0; i < s.length(); i++) {
```

Figure 5.1: *The Processing programming environment. Every time the 'play' button is pressed, the program is compiled and executed, with no state preserved between successive runs (unless explicitly defined).*



```
; Lindenmayer System growth
; -------------------------
;
; Repeatedly applies production rules to strings, and
; interprets the strings to form graphics, according to
; the method described in "The Algorithmic Beauty of
; Plants" by Przemyslaw Prusinkiewicz and Aristid
; Lindenmayer

; try all the rules on this character -
; returns #f if none are matched
(define (lsys-run-rules char rules)
  (foldl
   (lambda (rule str)
     (if str ; if str is not #f
         str ; then we have already found a rule, return it
         (if (char=? char (string-ref (car rule) 0)) ; check this rule
             (cadr rule) ; return the string
             #f))) ; no match
   #f
   rules))

; runs the lsystem rules on every character in a string,
; returns the new string
(define (lsys-search-replace str rules pos result)
  (cond
    ((>= pos (string-length str)) result)
    (else
     (let ((ret (lsys-run-rules (string-ref str pos) rules)))
       (if (string? ret)
           (lsys-search-replace str rules (+ pos 1)
             (string-append result ret))
           (lsys-search-replace str rules (+ pos 1)
             (string-append result (string (string-ref str pos)))))))))

; runs the search-replace multiple (n) times on a string
(define (ls-generate n str rules)
  (cond
    ((zero? n) str)
    (else
```

Figure 5.2: *The Fluxus programming environment. The editor containing the source code of a program is embedded in the same 3D scene as its output. The source is dynamically interpreted, so that the program may be changed during a live coding performance (§6.8).*

step in an arts context. We will investigate the creative processes of live coding more broadly in §6.8, and so for now focus on notational aspects, in particular, how the removal of the compilation step results in the intertwining of active notation with interpretation. Figure 5.2 shows the Fluxus interface, where the code editor is placed in the same scene as the process output. There is no 'play' button in Fluxus as such, instead the code is continuously playing. Incremental changes to the source code are reflected in the output whenever the programmer presses the F5 button. The visual form of the code is more literally part of its own output; the visual output is a 3D scene, of which the text editor is part. It is therefore possible to write a program that modifies the display of its own code.

How is live coding technically possible? Using the programming environments that have become conventional, artists cannot generally experience what they make at the same time as they are making it. This problem may seem insurmountable; while it runs, a program maintains *state*, its 'working memory', the data generated and collected during its execution. Such data may come with no description apart from its declaration in program code. If one then changes the program code, there may be no mechanism in which the new version of the program may continue where the old version left off, as the old state may be of no use to the new version of the program. The cognitive dimension of **progressive evaluation** therefore seems difficult to achieve. There are however a number of ways in which it can be done, although each case has significant impacts on other cognitive dimensions.

External grounding. For certain task domains, the internal state of a program has little or no significance. This is the case in Tidal (§4.5), where just about the only 'state' is the measurement of time kept by a separate process, the system clock. It is also the case in languages known as UNIX system *shells*. These developed from research into *conversational programming* (Kupka and Wilsing, 1980), where programmers interact with a language interpreter one statement at a time, able to consider systems feedback at each step. Conversational programming lives on in shell languages such as Bash, where the state is global to the whole computer system, with access control for security. In particular, state is almost exclusively held within the filesystem, a tree structure able to hold not only permanent files on disks, but processes, their environment variables and working 'core' memory (Filesystem Hierarchy Standard Group, 2003). As such a shell programmer tends to work with live data, making irreversible and at times drastic actions, an impact along the cognitive dimension of **premature commitment**. However some systems, including some file systems and relational databases, implement *transactions*, allowing a series of successive actions to be iteratively applied, but not committed. When live coders work before an audience, turning back performance time is impossible, but being able to roll back development to a previous version, for example for musical reprise, is certainly useful.

Internal grounding. This approach to **progressive evaluation** defines state in such a way that it may be passed from one version of a program to the next. This is conceptually similar to external grounding above, except the state is not shared with other processes. The programmer is left with the **hard mental operation** of being aware of certain conflicts that may arise, for example if a variable with the same name is used for different purposes in a successive version of the code, state will be carried across to unintended effect. In a strictly typed language such as Haskell, state is handled in a highly formalised manner, and so it is possible to arrange for a program to return its entire state for coercion into a form suitable for its successive version (Stewart and Chakravarty, 2005). Alternatively, dynamic scripting languages such as Perl make it straightforward to evaluate code that replaces existing functions, where state may be preserved using global variables.

Self-modification. A rather unusual approach, developed in the *Feedback.pl* live coding environment by the present author, is to store state in the source code itself (McLean, 2004). The programmer is put in the position of writing code that stores, reads and modifies values in its own description. This is done either by programmatically updating assignment statements, or storing values as program comments, so that **secondary notation** is parsed and modified by its own program. This creates an interaction where the source code is used to modify both the instructions and state of a process, and also to show output from its execution.

Temporal recursion. This term was coined by Sorensen (2005) in describing the operation of the music live coding system Impromptu, based around the Lisp derivative Scheme. A recursive function is one that calls itself, and a temporal recursive one is one that calls itself but with a scheduled delay. *Progressive evaluation* is then simply a case of swapping one function for another between self-calls.

Hybrid. A hybrid approach is certainly possible, exemplified by SuperCollider, a music programming language with two concurrent processes. One process maintains a pure functional graph for sound synthesis, while the other acts as a dynamic language interpreter. The language process manages such tasks as scheduling events and signalling graph changes of the synthesis process. The synthesis process is largely stateless, and care must be taken to avoid discontinuities when the graph is changed, audible as clicks. JITLib has been developed to aid this, providing a means to transition audio between successive versions of code (Rohrhuber et al., 2005).

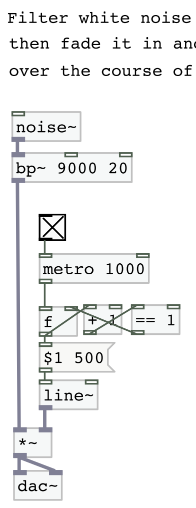
The above methods all act to embed program development in a timeline shared with its execution, with practical applications being explored in the arts. We now move from the role of time in notation, to examine the role of *space*.

5.3 Visual notation

Visual Programming Languages is an active research field, looking at how to design programming notations that go beyond conventional linear text. To head off confusion, note that the use of *visual* here is not the same as popular use, such as in "Visual Basic", to describe language environments based on GUI forms and event-driven programming. Instead, *visual programming languages* are those making heavy use of visual elements in the code itself, for example where a program is notated as a graphical diagram.

Research into visual languages is hampered by a definitional problem. In a well-cited taxonomy of visual programming, Myers (1990, p. 2) defines visual languages as "any system that allows the user to specify a program in a two (or more) dimensional fashion." Myers specifically excludes conventional textual languages, because "compilers or interpreters process them as long, one-dimensional streams." This exclusion is highly problematic however, as at base all a computer *can* do is process one-dimensional streams. Further, some textual languages such as Haskell and Python do indeed have two dimensional syntax, where vertical alignment as well as horizontal adjacency is significant; however no-one would call either language 'visual'. Worse still, for the majority of visual languages, 2D arrangement has no syntactical significance whatsoever, and is purely **secondary notation**. Often graphical icons are described as visual, but they too are discrete symbols, in other words textual.

This confusion stems from two misunderstandings, firstly that text is non-visual, and secondly that visual interfaces are necessarily a technological advance beyond textual interfaces. The former point is in denial of the heavy use of spatial layout in structuring text, and the latter point makes the same mistake as those assuming analogue expressions are more advanced than digital ones (§2.9). As a species we navigated spaces before we marked them out with sequences of symbols, and made marks on surfaces before we used those marks to transcribe words. Indeed it runs against implicit hierarchy in their representations; written words (ch. 3) are comprised of symbols (ch. 2), notated with images. From this perspective, textual interfaces are the more advanced, as they are developments of visual interfaces. However it does not follow that text is superior to visual interfaces, but rather that the whole dichotomy of visual and textual language is false. Language by nature involves discrete symbols, but may be integrated with analogue symbols towards a richer, dual code of expression.



Filter white noise at 900 hertz,
then fade it in and out every second,
over the course of a half second.

Figure 5.3: *The Pure Data visual programming environment. The locations of the boxes have no significance, but allow the programmer free rein in arranging their program in a manner meaningful to themselves. Image © Miller Puckette, used here under the terms of the BSD License.*

5.3.1 Patcher Languages

Research into visual programming languages has held much promise, but has so far had little in the way of broad industry take up. An intention of many visual language researchers has been to find ways of using visual notation that result in new, broadly superior general programming languages (Blackwell, 2006c). However this panacea has not been reached, and instead research around the cognitive dimensions of notations (§5.1), has identified inherent, inescapable trade-offs. There are however two domains where visual programming has been highly successful, one being engineering, exemplified by the LabVIEW visual programming language. The other, of special relevance to our theme, is computer music. In particular, *Patcher* languages have become a dominant force in computer music and interactive art since their introduction by Puckette (1988). Using a data flow model inspired by analogue modular synthesis, users of Patcher languages such as Pure Data or Max/MSP are able to build programs known as 'patches' using a visual notation of boxes and wires. Patches may be manipulated while they are active, a form of live coding predating the movement examined in §6.8 by well over a decade. The long-lived popularity of Patcher languages, the continued innovation within communities around them, and the artistic success of their use is undeniable.

Patcher languages allow programmers freedom in arranging their programs, to great advantage, a point we will come to later. However first we raise a point of contention; as with many visual programming languages, in terms of syntax, they are hardly visual at all. In the Pure Data patch shown in Figure 5.3, you could move the boxes wherever you like, or even place them all on top of one another, it would make no difference to the interpreter. Those familiar with Max/MSP may counter this line of argument by pointing out that right-left ordering signifies evaluation order in Max. This falls on two counts; first, Max programmers are discouraged from branching that relies on this, in favour of the *trigger* object. Second, having right-left execution order does not distinguish Max from any mainstream textual language. Indeed to say that Patcher languages are not themselves significantly textual is in blind denial of the large number of operators and keywords shown as editable text in a patch.

An advantage of Patcher languages is that they are unconstrained by visible dimensions, in that you can place objects anywhere with little or no syntactical significance. In conventional languages, to relate two words together, you must put them adjacent to one another, in a particular order. In Patcher languages, you instead connect them with lines, effectively defining syntax graphs of arbitrary dimension, hypercubes and up. As opposed to the definition related earlier, from this perspective it is actually 'textual' rather than 'visual' languages that are constrained by the two dimensions of the visual canvas.

The above is not an attack on Patcher languages – syntax is not everything. Well, to the interpreter syntax *is* everything, but to the programmer, who is our focus, it is only half the story. Because Patcher languages have such remarkably free **secondary notation**, they allow us to lay programs out however we like, and we may embrace this visual freedom in making beautiful patches that through shape and form relate structure to a human at a glance, in ways that linguistic syntax alone cannot do. That is why we call these visual languages; while the language syntax is not visual, the notation is very much so.

5.4 Notation and Mental Imagery

All programming languages can be considered in visual terms, we do after all normally use our eyes to read source code. Programming languages have context-free grammars, allowing recursive forms often encapsulated within round brackets, resulting in a kind of visual Euler diagram. We can also say that adjacency is a visual attribute of grammar; as we know from Gestalt theory, adjacency has an important role in perceptual grouping (Kohler, 1930). These visual features are used to support reading of code, where our eyes saccade across the screen, recognising discrete symbols in parallel, chunked into words (Rayner and Pollatsek, 1994). Crucially however, we are able to attend to both visuospatial and linguistic aspects of a scene simultaneously, and integrate them (§2.2.1). Discrete symbols are expressed within a language grammar, supplemented by visuospatial arrangement expressing paralinguistic structure. The computer generally only deals with the former, but the human is able to attend to both.

Magnusson (2009) describes a fundamental difference between acoustic and digital music instruments in the way we play them. He rightly points out that code does not vibrate, and so we cannot explore and interact with a computer music language with our bodies, in the same way as an acoustic instrument. However, programmers still have bodies which contain and shape their thoughts, and in turn, through **secondary notation**, shape their code. Programmers do not physically resonate with code, but cognitive resources grounded in perceptual acuity enables them to take advantage of visuospatial cognition in their work.

We have seen that visuospatial arrangement is of importance to the notation of Patcher languages, despite not being part of syntax. Our argument follows that if shape, geometry and perceptual cues are so important to human understanding, then we should look for ways of taking these aspects out of **secondary notation** and make them part of primary syntax. Indeed, some languages, including recent music programming languages already have.

5.5 Geometry in Syntax

In §2.2.3, we examined the role of analogue representation in computer language against the background of Dual Coding theory. Computer languages allow programmers to communicate their ideas through abstraction, but in general do not at base include visuospatial syntax to support this. Do programmers then simply switch off a whole channel of perception, to focus only on the discrete representation of code? It would appear not, we have seen that not only do programmers report mental imagery in programming tasks (§2.2.2), but that the use of spatial layout is an important feature of **secondary notation** in mainstream programming languages, which tend to allow programmers to add whitespace to their code freely with little or no syntactical meaning (§5.4). Programmers use this freedom to arrange their code so that its shape relates important structure at a glance. That programmers need to use spatial layout as a crutch while composing discrete symbolic sequences is telling; to the interpreter, a block may be a subsequence between braces, but to an experienced programmer it is a perceptual gestalt grouped by indentation. From this we conclude that concordant with Dual Coding theory, the linguistic work of programming is supported by spatial reasoning, with **secondary notation** helping bridge the divide.

The question is whether spatial reasoning can be successfully employed to work with primary syntax, where visuospatial layout is relevant to the computer as well as the human. We can find interesting perspectives on this question in the realm of *esoteric programming languages*.[2] Esoteric programming languages are those which demonstrate strange and ex-

[2]See `http://www.esolangs.org/` for a comprehensive catalogue of esoteric programming languages.

perimental ideas, for fun rather than practical use, and we approach them here in search for inspiration for practical applications in the arts. The use of spatial arrangement in primary syntax has become a popular game in the esoteric language community. *Befunge*, illustrated in Figure 5.4, is a particularly interesting example, a textual language with a highly two dimensional syntax. Control flow may proceed up, down, left or right, directed by single character instructions.

The instruction set of the *Piet* language is inspired by Befunge, but rather than the contents of individual cells, the Piet instruction set is given by colour relationships between neighbouring cells. For example, a multiply operation is symbolised by the colour hue changing a little and darkening considerably. The programmer is given a great deal of freedom to choose particular colours within these constraints. Much like Befunge, Piet instructions include directional modifiers, and control flow travels in two dimensions. Figure 5.5 contains an example program, showing its resemblance to the modernist paintings of Piet Mondrian which inspired the Piet language.

Musicians often lead the way in technology, and programming language design is no exception, as there are a number of examples of computer music languages which include geometrical measures of spatial arrangement in their primary syntax. Firstly, *Nodal* is a commercial environment for programming agent-based music (Mcilwain et al., 2005). Nodal has several interesting features, but is notable here for its spatial syntax, where distance symbolises elapsed time. As the graph is read by the interpreter, musical events at graph nodes are triggered, where the flow of execution is slowed by distance between nodes. Colour also has syntactic value, where paths are identified by one of a number of hues.

Al-Jazari is one of a series of playful languages created by Dave Griffiths, based on a computer game engine and controlled by a gamepad (McLean et al., 2010). In Al-Jazari, cartoon depictions of robots are placed on a grid and given short programs for navigating it, in the form of sequences of movements including interactions with other robots. As with Nodal, space maps to time, but there is also a mechanism where robots take action based on the proximity and orientation of a another robot. In programming Al-Jazari you are therefore put in the position of viewing a two dimensional space from the point of view of an agent's flow of execution. Indeed it is possible to make this literally so, as you may switch from the crow's nest view shown in Figure 6.1 to the first-person view of a robot.

Our last example is the ReacTable (Jordà et al., 2007), a celebrated tangible interface aimed towards live music. Its creators do not describe the ReacTable as a programming language, and claim its tangible interface overcomes inherent problems in visual programming languages such as Pure Data. But truly, the ReacTable is itself a visual programming language, if an

```
vv  <      <
    2
    ^  v<
 v1<?>3v4
    ^   ^
>  >?>  ?>5^
    v   v
 v9<?>7v6
    v  v<
    8
 .  >  >   ^
^<
```

Figure 5.4: *A pseudo-random number generator written in the two-dimensional language Befunge.*

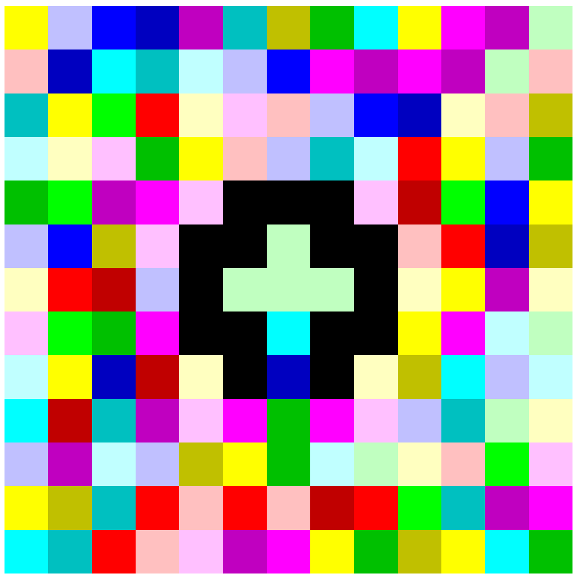


Figure 5.5: *Source code written in the Piet language with two dimensional, colour syntax. Prints out the text "Hello, world!". Image © Thomas Schoch 2006. Used under the Creative Commons BY-SA 2.5 license.*

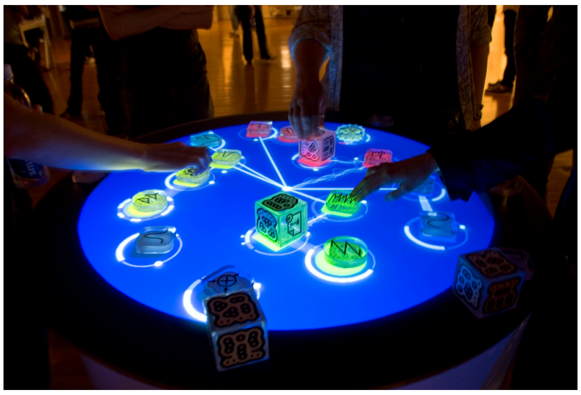


Figure 5.6: *The ReacTable music language, where relative distance and orientation is part of primary syntax. Image © Daniel Williams, used here under the terms of the Creative Commons BY-SA 2.0 License.*

extraordinary one. It has a visual syntax, where physical blocks placed on the ReacTable are identified as symbols, and connected according to a nearest neighbour algorithm, the results of which can be seen in Figure 5.6. Not only that, but relative distance and orientation between connected symbols are parsed as values, and used as parameters to the functions represented by the symbols. Video is back-projected onto the ReacTable surface to give feedback to the musician, for example by visualising the sound signal between nodes. The ReacTable has also been repurposed as an experimental interface for making graphics (Gallardo et al., 2008), which suggests that the ReacTable is not as far from a general purpose programming language as it may first appear.

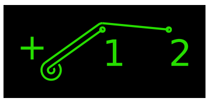
5.6 Visual Programming with Texture

We now introduce *Texture*, a visual programming language, based upon the Tidal language for the live coding of pattern introduced in §4.5. The name *Texture* is intended to accentuate the role of text in programming, as a structure woven into a two dimensional surface. This reflects Texture's novel use of spatial arrangement in the notation of a pure functional programming language. An important design aim for Texture is to create a programming notation suitable for short, improvised scripts, allowing fast manipulation by an improviser, where lay audience members may appreciate more of the structure behind the code, made explicit through cues that are both visual and syntactical. Here we describe Texture as an early prototype system with novel features, describing the thinking behind it and issues raised through its development and early use.

The Texture environment and parser is implemented in C, using the free/open source Clutter graphics library (`http://clutter-project.org/`) for its user interface. It compiles its programs into an intermediate form of Haskell (Jones, 2002) code, which is piped straight to a Haskell dynamic interpreter whenever a Texture program is modified. Haskell then takes care of the task of evaluating the code and scheduling sound events accordingly, using Tidal. Much of Haskell's type system is re-implemented in Texture, its contribution not being to provide a whole new language, but rather a complementary and novel visual syntax.

5.6.1 Geometric relationships

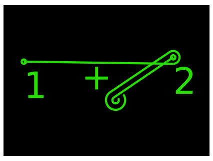
A program written in Texture is composed of strongly typed values and higher order functions. For example the function `+` takes two numbers as arguments (either integers or reals, but not a mixture without explicit conversion), and returns their sum. Here is how `+ 1 2` looks in Texture:

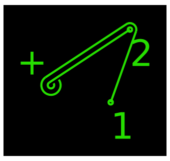


Two green lines emerge from the bottom right hand side of the function `+`, both travelling to the upper left hand side of its first argument `1`, and then one travelling on to the second argument `2`. This visualises Haskell's automatic *currying* of functions (§4.5.1), whereby each time a parameter is applied, a function is returned with arity reduced by one. This visualisation of arity makes it easy to perceive where functions are partially applied, which we will show later in our example of `fmap`.

The programmer types in functions and values, but does not manually add the lines connecting them, as they would with a Patcher language. The lines are instead inferred and drawn automatically by the language environment, according to a simple rule: the closest two type-compatible function-value pairs connect, followed by the next two closest, and so on. Values may be moved freely with the mouse, which may change the topology of the graph, which updates automatically.

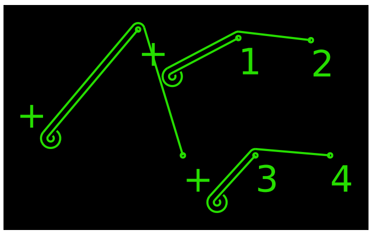
Texture has prefix notation (also known as Polish notation) where the function comes first, followed by its arguments. There is no distinction between functions and operators. The values can be placed and moved around the screen, with a line drawn linking the function to each of its arguments in turn. For example `+ 2 1` may be expressed as either of the following:



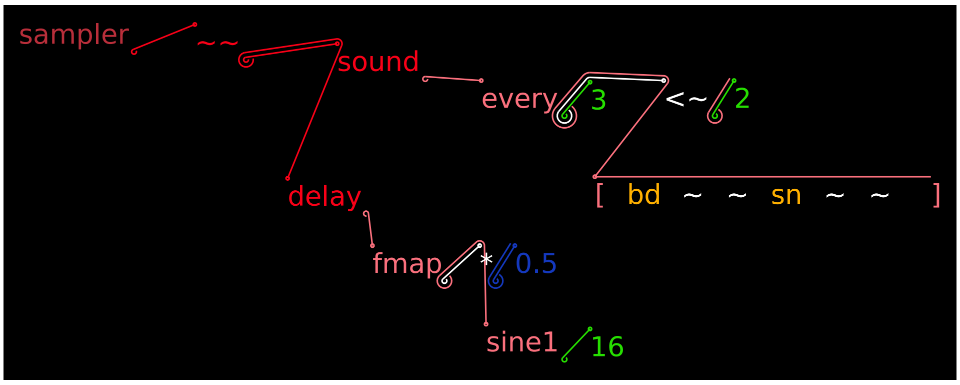


In both cases, two parallel lines travel from the bottom right hand corner of the function `+` to the top left of `2`, being the closest compatible value. As already mentioned, these lines represent the arity of the function. The line representing the parameter terminates at its argument `2`, and the remaining line continues to the top left of `1`. Again, these symbols connect automatically, closest first, where 'closest' is defined as Euclidean distance in two dimensional space. If a symbol is moved, the whole program is re-interpreted, with connections re-routed accordingly.

The functions may be composed together as you might expect. Here is `+ (+ 1 2) (+ 3 4)` in Texture:



The use of green in the above examples is significant, representing the integer types. The following example shows all the types currently supported by Texture at play:



Of the basic types, strings are golden yellow, integers are green and floating point numbers are blue. In general, Tidal `Pattern` s (§4.5.2) are pink, or the similar hues of red for patterns of OSC parameters or dark pink for patterns of OSC messages (§4.6). Functions assume the colour of their return value, or if they are not fully applied (i.e. return another function) then they are white. Lines connecting functions to their parameters are coloured in the same way, for example in the above example `every` takes an integer, a function and a pattern to apply the function to, so its lines are coloured green, white and pink respectively.

The following shows the output (re-formatted here to aid reading) from Texture given the above example.

```
sampler ((~ ~)
         (sound (every (3)
                 (<~ 2)
                 (lToP [Just "bd", Nothing, Nothing,
                        Just "sn", Nothing, Nothing])
                )
         )
         (delay (fmap (* 0.500000) (sine1 16)))
        )
```

The `lToP` function is of type `[Maybe a] →Pattern a`, and simply turns a list of events into a `Pattern`. The other functions are explained in §4.5.

The strong typing in Haskell places great restrictions on which arguments may be applied to which Tidal functions. This "bondage and discipline" works out well for Texture, as it limits the number of possible connections, making it easier for the programmer to predict what will connect where, supported by the colour highlighting. Furthermore, because Texture enforces type correctness, there is no possibility of syntactically incorrect code.

5.6.2 User Interface

The Texture user interface is centred around typing, editing and moving words. In fact that is all you can do – there are no menus or key combinations. A new word is created by moving the cursor to an empty part of the screen using the mouse, and then typing. The word is completed by pressing the space bar, at which point the cursor moves a little to the right where the next word can be begun, mimicking a conventional text editor. A word is edited by being given focus with a click of the mouse, or moved by holding down the shift key while being dragged with the mouse. A whole function tree (the function and its connected arguments) is moved by holding down control while dragging, although the arguments may connect differently in the new location according to the new context.

5.6.3 Texture in Practice

Having seen much of the technical detail of Texture, we turn to its musical context. By way of illustration, a video showing Texture in use is available on the enclosed DVD.

Texture is a prototype language that has not yet undergone full examination through HCI study, however preliminary observations have been conducted. In particular a small workshop for six participants was arranged with the Access Space free media lab in Sheffield, and led by the present author. A limit of six people was agreed as a good balance for a guided workshop, and the places were filled on first-come-first-served basis. The participants were found by the arts programme manager at Access Space, who advertised through their own website, through public, arts related mailing lists, as well as approaching regular Access Space participants directly. All participants were male, aged 23, 26, 28, 30, 42 and 42 years of age. Four lived locally to Sheffield, and two travelled from Liverpool. All identified as musicians, four using computers in their music, and five playing traditional instruments. Two had prior experience of programming, only one of which had experience of a functional programming language. The workshop was free of charge, being part of an arts programme funded by the Arts Council, England. Participants were free to leave at any time with no penalty, but all stayed to the end.

The workshop was in the form of an hour long presentation surveying live coding practice and other influences of Texture, followed by a three hour hands-on workshop. The first half of the hands-on section introduced techniques on a projected display, which participants, while listening on headphones, copied and adapted in exploration of their use. The second half was more freeform, where each participant had their own set of stereo speakers at their computers. The participants were playing to a globally set tempo, with accurate time synchronisation.[3] This meant that they were able to respond to each other's patterns, improvising music together; because of the layout of the room, it was only possible to clearly hear the music of immediate neighbours. Recorded video taken from this part of the workshop is available on the enclosed DVD.

The participants were the first people to use Texture besides the present author, and so there was some risk that the participants would be unable to engage with it due to unforeseen technical problems or task difficulty. However all showed enthusiasm, were keen to explore the language, and joined in with playing together over speakers.

The participants were surveyed for opinions through a questionnaire answered via the surveymonkey website, using computers at Access Space. This was done in two parts, immediately before and then immediately after the workshop. The participants were told that the survey was designed to "help in the development of the software used in the workshop" and were asked to "answer the questions honestly". To encourage honest responses further, the survey was conducted anonymously.

Participants were asked to rate their agreement with three statements both before and after the workshop, on a Likert scale from Disagree Strongly (1) to Agree Strongly (5). Although there is little statistical power for such a small group, feedback from these individuals was encouraging for a system at such an early stage of development. The results are visualised in Figure 5.7. Overall, participants tended to agree with "I am interested in live coding" and "I would like to be a live coder" both before and after the workshop. They largely disagreed with "I am a live coder" at the beginning but were less sure by the end, indicating they had

[3]Accurate time synchronisation was made possible by the netclock protocol.

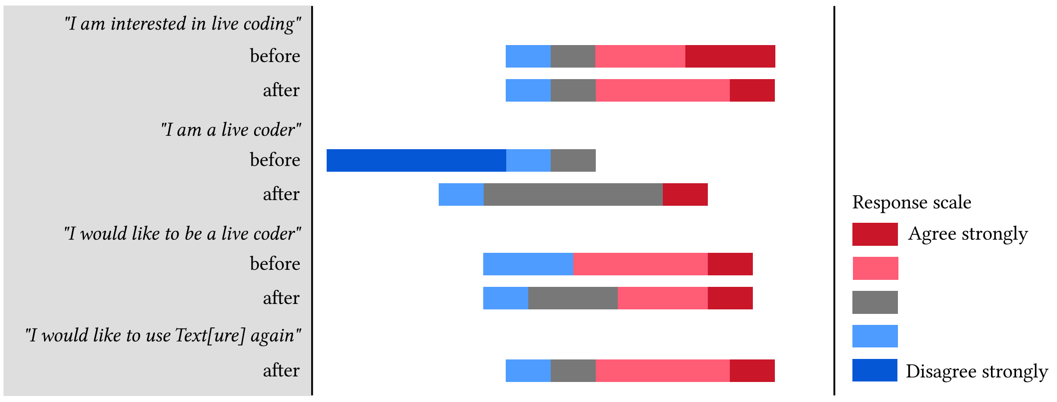


Figure 5.7: *Responses on a Likert scale from participants (n=6) in a workshop for the Texture visual live coding language. The graph visualises a straightforward response count for each question centered on the neutral responses. Responses to the first three statements were collected both before and after the workshop.*

achieved some insights and begun to identify with the practice. At the end, four out of the six participants agreed with a final statement "I would like to use Text[ure][4] again".

Participants were also given freeform questions asking what they liked and disliked about Texture, how much they felt they understood the connection between the visual representation and the sound, and soliciting suggestions for improvements. Dislikes and suggestions focused on technical interface issues such as the lack of 'undo', and three found the automatic linking difficult to work with. On the other hand, three participants reported liking how quick and easy it was to make changes.

5.6.4 Cognitive Dimensions of Texture

Texture is designed for the improvisation of musical pattern, as a visual programming interface to the Tidal pattern DSL. The result is a more tightly constrained system than many programming languages for music, which generally include extensive facilities for low level sound synthesis. While the ability to compose right from the micro-level of the sound signal offers great possibilities, it comes with trade-offs, in particular along the **hard mental operations** and **diffuseness** (verbosity) cognitive dimensions.

The **visibility** of Texture is high, where a complex rhythm can be notated on a single screen. We also argue that Texture has high **closeness of mapping**, as the visual representation of trees within trees corresponds well with the hierarchical structure of the pattern that is being composed. This echoes the tree structures common in music analysis, and indeed we would expect significant correspondence between the Texture structure and the listener's perception of it. The extent to which an untrained listener may relate the structure they hear with the

[4]At the time of the workshop, Texture had the working name of *Text*.

Texture program they see is an empirical question, but we suspect that further development is needed to support this.

Creative use of Texture is aided not only by high **visibility** but also aspects of **provisionality**. What is meant by "creative use" will be outlined in relation to *bricolage programming* in §6.3.1. A programmer may work on a section of code and drag it into the main body of the program when it is ready. They may also drag part of the code out of the main body and reuse it elsewhere later. The code must always be syntactically correct, but unless it connects to a function representing OSC messages sent to a synthesiser, it will have no effect.

The **error-proneness** of Texture is well positioned. It is impossible to make syntax errors in Texture, and while the automatic connection can at times have unexpected results, the result is at times musically interesting, but otherwise straightforward to reverse.

5.6.5 Future directions

Texture is a working prototype, in that it is fully functional as a live music interface, but is a proof of concept of an approach to programming that brings many further ideas to mind.

In terms of visual arrangement, Texture treats words as square objects, but perhaps the individual marks of the symbols could be brought into the visual notation, through experiments in typography. For example, a cursive font could be used where the trajectory of the final stroke in a word is continued with a curve to flow into the leading stroke of the word it connects to. This suggestion may turn the stomach of hardened programmers, although Texture is already unusual in using a proportional font, complete with ligatures.

Currently there is no provision in Texture for making named abstractions, so a piece of code cannot be used more than once without being repeated verbatim. Visual syntax for single assignment could symbolise a section of code with a shape derived from the arrangement of its components. That shape would become an ideographic symbol for the code, and then be duplicated and reused elsewhere in the program using the mouse.

Texture is inspired by the ReacTable, but does not feature any of the ReacTable's tangible interaction. Such tabletop interfaces offer a number of advantages over keyboard-and-mouse interfaces, in particular multitouch, allowing movement of more than one component at once. Multi-touch tablet computers share this advantage while avoiding some of the tradeoffs of tangible interfaces. Much of the ReacTable technology is available as an open research platform, and could be highly useful in this area of experimentation.

Currently the only output of Texture is music rendered as sound, with no visual feedback. There is great scope for experimentation in visualising the patterns in the code, making it easier for live coders and audience members to connect musical events and transformations with particular sections of the code. One approach would be to visualise pattern flowing between nodes, again inspired by the ReacTable, however as Texture is based upon pure functions rather than dataflow graphs, it presents a rather different design challenge. Texture could be adapted to allow any Haskell program to be written, and so could be applied to other domains, such as digital signal processing and visual animation. This would again place different challenges on the visualisation of results.

Most generally, and perhaps most importantly, we look towards proper analysis of lay audience code comprehension, grounding further development with better understanding of what the design challenges are.

5.7 Discussion

We have considered programming languages as rich notations with both visual and linguistic aspects. Many computer musicians write words to describe their music, for computers to translate to sound. The computer musicians have become comfortable with this rather odd process in private, but perhaps found it difficult to explain to their parents. Simply by projecting their interfaces, live coders have brought this oddity out in public, and must deal with the consequences of bringing many issues underlying computer music to the surface.

Live coding may be understood as a dual activity of language and spatial perception. We have seen how humans have the capacity to integrate both simultaneously, showing that the act of live coding, and perhaps the audience reception of it, can be realised and felt simultaneously as both musical language and musical movement.

By placing the design of live coding languages in a psychological and analytical context, we have aimed to support future directions in live coding design. The introduction of Texture takes a step in this direction, and we hope demonstrates the exciting ground waiting for future language designers to explore.

Chapter 6

Creativity

From symbols up, each of the preceding three chapters has built a layer of abstraction on top of the previous one. The next layer takes us above the activity of notation, to the broader context of creative artistic activity. How can programming fit into a creative process?

6.1 Programmer culture

From early beginnings programmers have pulled themselves up by their bootstraps, creating languages within languages in which great hierarchies of interacting systems are expressed. Much of this activity has been towards military, business or scientific goals. However there are numerous examples of alternative programmer subcultures forming around fringe activity without obvious practical application. The Hacker culture at MIT was an early example (Levy, 2002), a group of male model-railway enthusiasts and phone network hackers who dedicated their lives to exploring the possibilities brought by the new, digital computers. Many other programming cultures have since flourished. Particularly strong and long-lived is the *demoscene*, a youth culture engaged in pushing computer video animation to the limits of available hardware, using novel algorithmic techniques to dazzling ends. The demoscene spans much of the globe but is particularly strong in Nordic countries, where annual meetings attract thousands of participants (Polgár, 2005). Another, perhaps looser programmer culture is that of Esoteric Programming Languages discussed in §5.5. The authors of these languages push against the boundaries of programming, providing insight into the constraints of mainstream programming languages.

Members of the demoscene and esoteric language cultures do not necessarily self-identify as artists, despite their relentless search for novel approaches. However there are cultures of programmers who do call themselves artists, now extending into their second and third generations. Early on, communities of experimental artists looking for new means of expres-

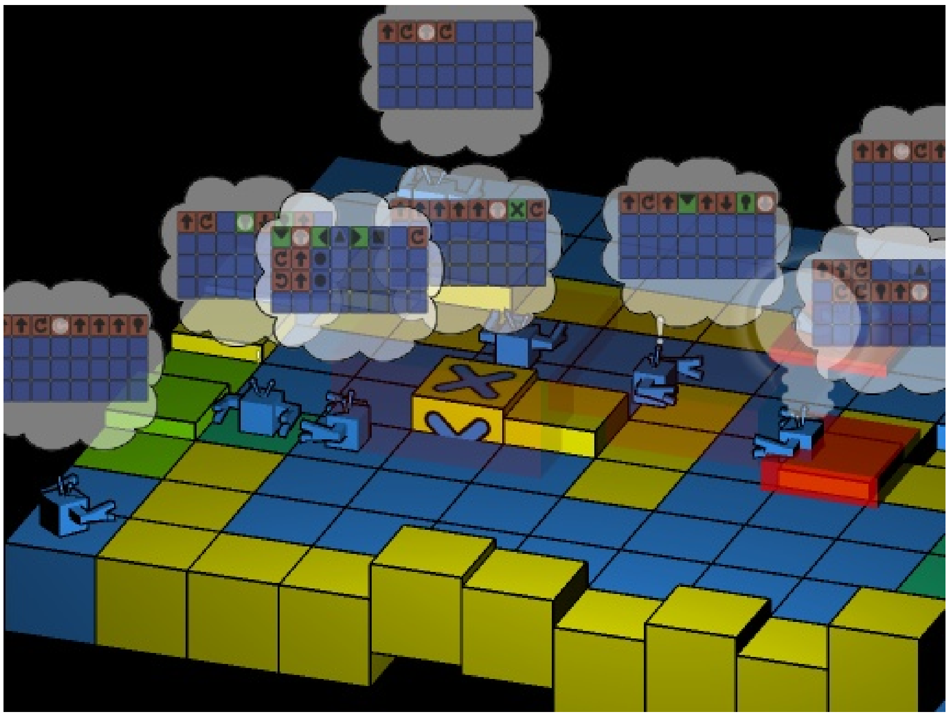


Figure 6.1: *The robots of the Al-Jazari language by Dave Griffiths (McLean et al., 2010). Each robot has a thought bubble containing a small program, edited through a game pad.*

sion grew around computers as soon as access could be gained. In Great Britain interest during the 1960s grew into the formation of the Computer Arts Society (CAS; Brown et al., 2009, `www.computer-arts-society.org`). However after a creative boom CAS entered a period of dormancy in the mid 1980s, perhaps drowned out by extreme commercial growth in the computer industry at that time. CAS has however been revived in more recent years, encouraged by a major resurgence of software as a medium for the arts. This resurgence has seen a wealth of new programming environments designed for artists and musicians, such as Processing (Reas and Fry, 2007), SuperCollider (McCartney, 2002), ChucK (Wang and Cook, 2004), VVVV (`http://vvvv.org`) and OpenFrameworks (`http://openframeworks.cc`), joining more established environments such as the Patcher languages PureData and Max (§5.3.1).

The creators of programming languages for the arts are often artist-programmers themselves, motivated to support their own work. This results in experimental features which resemble those of esoteric languages, for example unique representations of time are central features of ChucK and SuperCollider (§5.2). Languages created by artist-programmers have themselves been exhibited as interactive works of art, such as the *Al-Jazari* music programming environment shown in Figure 6.1 (McLean et al., 2010).

The design of programming languages for the arts is an active research domain, with new approaches still emerging, bringing important psychological issues to the fore. As computers enter almost every aspect of our lives, few would now deny that people can be creative using computer tools. Against all this context it is clear to see the activity of programming as potentially being highly creative. Before we discuss the creative processes of programming in depth however, we should describe what we mean by creative behaviour.

6.2 Concepts in creative behaviour

Creativity is assumed to be a mark of the human species, and something that we all do as part of a healthy approach to life. The alternative to a creative life is a mechanistic one, lacking in introspection and control over the self-imposed rules we live by. However, the word *creative* is used to define a wide range of behaviour, for example a creative person in the advertising industry is something rather more specific than a creative person in the arts. So what exactly do we mean by *creativity*? We might try to define it in terms of qualities of the artefacts that creativity produces. However we judge the creativity of an artefact not only by its intrinsic value, but also its novelty in a culture, and perhaps how much it surprises us. That is, we judge an artefact not just by any physical manifestation, but primarily the concepts behind it. To understand creativity then, we should focus on the conceptual activity behind creative works.

In §2.2.5, we took the definition of a concept as "a mental representation of a class of things" (Murphy, 2002, p.5). By asserting that creativity is a conceptual process, we therefore imply that creativity is not in the production of things, but rather in the organisation of things into novel classes. If we recognise creativity in an artefact, it is because its properties allow us to infer a novel conceptual class of which it is a member.

If concepts are the primary output of the creative process, we should define how and where we think they are represented. In our review of conceptual representation (§2.2) we shared the view that concepts are structured in relation to perception; that basic concepts arise from recurrent states in sensorimotor systems, which in turn form the building blocks of higher level abstract thought. When we creatively generate novel, valued concepts, we are literally altering our perception of the world and of ourselves.

Exactly how a concept is represented in the human mind is an open question. Here we take the view that a conceptual property is represented by a single best possible example, or *prototype*. In accordance with the theories reviewed in chapter 2, these prototypes arise through perceptual states, within the geometry of quality dimensions. To ground the discussion in music, consider a piece of jazz, where *jazz* is the concept and the particular composition is an instance of that concept. The musician, in exploring the boundaries of jazz, then finds a piece beyond the usual rules of jazz. Through this process, the boundaries of a music genre may be redefined to some degree, or if the piece is in particularly fertile new ground, a new sub-genre of jazz may emerge. Indeed a piece of music which does not break boundaries in some way could be considered uncreative. These changes in conceptual structure first happen in an individual, which in the case of music is the composer or improviser. Another individual's conceptual structures may be modified to accord with a composer's new concept by listening to the instance of the new musical concept, although success is only likely if the individual already shares some of the music cultural norms of the creator.

Wiggins (2006a,b) formalises the Creative Systems Framework (CSF) based on the work of Boden (2003), in order to provide a framework for discussing creativity and comparing creative systems. The CSF characterises creativity as a search in a conceptual space, according to a mutable set of rules. Treating creativity as a search is much the same as treating it as construction, but implies that creativity takes place within defined boundaries. Creativity is often discussed using the metaphor of exploration, around where boundaries are defined and broken, and the CSF allows us to talk about such behaviour within its well defined terms.

As a comparative framework, the CSF is agnostic to issues of representation, and so may be applied to both analogue and discrete conceptual representations. In taking the Gärdenforsian approach to conceptual representation (§2.2.5), we argue that creativity involves employing visuospatial cognitive resources to navigate an analogue search space with geometric structure. We will discuss creative processes of programming in terms of the mechanisms of the CSF later in §6.5.

The subject of creativity within the computer arts field is mired in confusion and misconception. The perennial question of *authorship* is always with us: if a computer program outputs art, who has made it, the human or the machine? Positions on creativity through computer programming tend towards opposite poles, with outright denials of creativity at one end and outlandish claims of unbound creativity in generative art at the other. Here we look for clarity through our anthropocentric view, with focus on programming as the key activity behind computer art. We view the artist-programmer as engaged in inner human relationships between perception, cognition and computation, and relate this to the notation and operation of their algorithms, particularly in the context of live coding.

6.3 Creative Processes

Creative processes are rather more mysterious than computer processes, which we like to think of as well defined, predictable and subservient to human control. For artist-programmers though, the computer process is a component of their creative process. What then is the relationship between an artist, their creative process, their program, and their artistic works? We will look for answers against the backgrounds of psychology, cognitive linguistics, computer science and computational creativity, but first from the perspective of an artist.

The painter Paul Klee describes a creative process as a feedback loop:

> "Already at the very beginning of the productive act, shortly after the initial motion to create, occurs the first counter motion, the initial movement of receptivity. This means: the creator controls whether what he has produced so far is good. The work as human action (genesis) is productive as well as receptive. It is **continuity**." (Klee, 1953, p. 33, original emphasis)

This is creativity without planning, a feedback loop of making a mark on canvas, perceiving the effect, and reacting with a further mark. Being engaged in a tight creative feedback loop places the artist close to their work, guiding an idea to unforeseeable conclusion through a flow of creative perception and action. Klee writes as a painter, working directly with his medium. Artist-programmers instead work using computer language as text representing their medium, and it might seem that this extra level of abstraction could hinder creative feedback. We will see however that this is not necessarily the case, beginning with the account of Turkle and Papert (1992), describing a *bricolage* approach (after Lévi-Strauss, 1968) to programming by analogy with painting:

> The bricoleur resembles the painter who stands back between brushstrokes, looks at the canvas, and only after this contemplation, decides what to do next. Bricoleurs use a mastery of associations and interactions. For planners, mistakes are missteps; bricoleurs use a navigation of mid-course corrections. For planners, a program is an instrument for premeditated control; bricoleurs have goals but set out to realize them in the spirit of a collaborative venture with the machine. For planners, getting a program to work is like "saying one's piece"; for bricoleurs, it is more like a conversation than a monologue.
>
> (Turkle and Papert, 1990, p. 136)

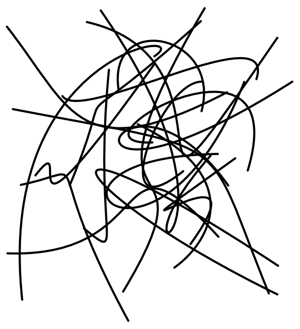
Turkle and Papert describe the bricolage programmer as forming ideas while working in the text editor, making edits in reaction to edits, rather than planning their work in advance. This accords with Klee's account, and may also be related to that of *Reflective Practice* from professional studies (Schon, 1984). Reflective practice distinguishes the normal conception of knowledge, as gained through study of theory, from that which is learnt, applied and reflected upon while 'in the work'. As such, practice is not ruled by theory, but embedded in activity. Reflective practice has strong influences in professional training, particularly in the educational and medical fields. This suggests that the present discussion could have relevance beyond our focus on the arts.

Although Turkle and Papert address gender issues in computer education, this quote should not be misread as dividing all programmers into two types; while associating bricolage with feminine and planning with male traits (although for critique of their method see Blackwell, 2006a), they are careful to state that these are extremes of a behavioural continuum. Indeed, programming style is clearly task specific: for example a project requiring a large team needs more planning than a short script written by the end user.

Bricolage programming is particularly applicable to our theme of the artist-programmer, writing software to work with media such as music and video animation. To ground the following discussion, we bring an image of a visual artist to mind, programming their work using the Processing language environment (§5.2). Our artist begins with an urge to draw superimposed curved lines, having been inspired by something they saw from the train the previous day. They quickly come up with the following program, shown with its output below:

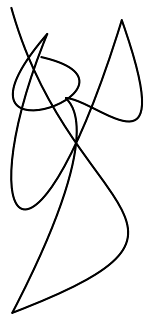
```
float rx() { return(random(width)); }
float ry() { return(random(height)); }

void draw() {
  background(255);
  for (int i = 0; i < 20; ++i) {
    bezier(rx(), ry(), rx(), ry(),
           rx(), ry(), rx(), ry());
  }
}
```



On seeing the output of the first run, our artist is immediately struck by how hairy it looks. Thanks to arbitrary placement of the curves through use of pseudo-random numbers, each time the artist runs the program it comes up with a different form. Over a few runs, the artist notices the occasional suggestion of a handwritten scribble. Intrigued, they change their program to join the curves together, in order to remove the hairiness and accentuate the scribble:

```
void draw() {
  background(255);
  float x = rx(); float y = ry();
  for (int i = 0; i < 5; ++i) {
    float x1 = rx(); float y1 = ry();
    bezier(x, y, rx(), ry(),
           rx(), ry(), x1, y1);
    x = x1; y = y1;
  }
}
```

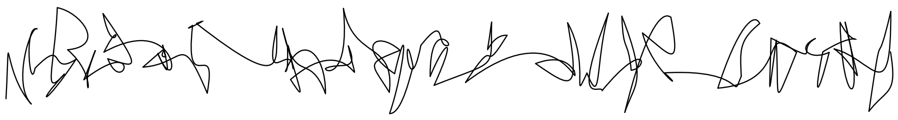


The end-points of the curves are still placed arbitrarily, but they now begin at the point where the previous curve ended, resulting in a continuous line. After a few more runs, our artist begins to perceive a letter-like quality in the scribble. They decide to try writing them across the page, grouped into word-like forms:

```
float letterSpace = 30;

float rx() { return(random(letterSpace + 10)); }
float ry() { return(random(height - 10)); }
int rWordlen() { return(3 + int(random(4))); }

void draw() {
  background(255);
  int letters = (int) (width / letterSpace) - 4;
  int wordLen = rWordlen();
  int word = 0;
  float x = rx(); float y = ry();
  for (int letter = 0; letter < letters; ++letter) {
    float ox = letter * letterSpace + word *
letterSpace;
    if (wordLen-- == 0) {
      wordLen = rWordlen();
      word++;
    }
    for (int i = 0; i < 3; ++i) {
      float x1 = rx() + ox; float y1 = ry();
      bezier(x, y, rx() + ox, ry(),
             rx() + ox, ry(), x1, y1);
      x = x1; y = y1;
    }
  }
}
```



The output has a handwritten quality, appearing almost readable, a quality of 'automatic writing' used by mystics to supposedly channel the spirit world. This may bring further conceptual development to our artist's mind, but at this point we will leave them pondering.

Our case study is somewhat simplistic, and is not intended to illustrate either great art or great code. However it does trace a creative process of sorts, as carried out by the present author. We are not suggesting that the algorithms themselves are creative, any more than we would suggest that paint is creative. Multiple executions helped our artist perceive qualities in the output, but it was the artist's perception, and not the algorithm that discovered value. It is clear that our programmer, like a painter, could not understand their code until they had perceived its output, that the act of perception was itself creative, and that the concept they were trying to encode was continually changing in response to their perception of the results. We seek to understand this process in greater depth in the following sections.

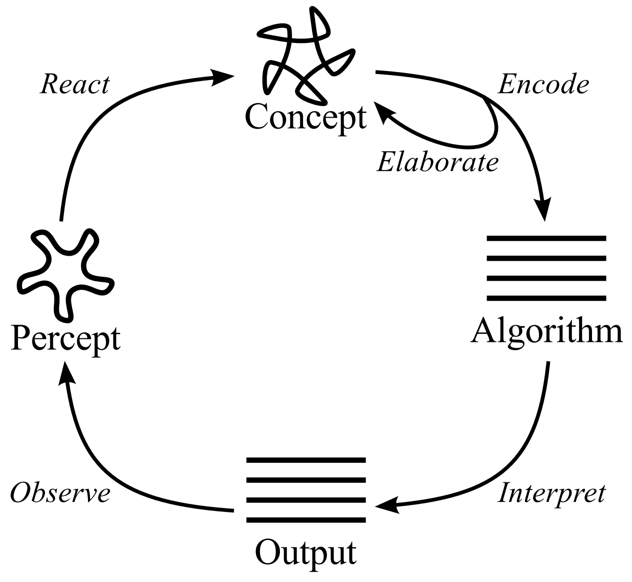


Figure 6.2: *The process of action and reaction in bricolage programming*

6.3.1 Creative Process of Bricolage

> "The source of the exhilaration associated with computer programming is the continual unfolding within the mind and on the computer of mechanisms expressed as programs and the explosion of perception they generate." Alan Perlis (foreword to Abelson and Sussman, 1996)

Figure 6.2 characterises bricolage programming as a creative feedback loop encompassing the written algorithm, its interpretation, and the programmer's perception and reaction to its output or behaviour. Creative feedback loops are far from unique to programming, but the addition of the algorithmic component makes an additional inner loop explicit between the programmer and their text. At the beginning, the programmer may have a half-formed concept, which only reaches internal consistency through the process of being expressed as an algorithm. The inner loop is where the programmer elaborates upon their imagination of what might be, and the outer where this trajectory is grounded in the pragmatics of what they have actually made. Through this process both algorithm and concept are developed, until the programmer feels they accord with one another, or otherwise judges the creative process to be finished.

The lack of forward planning in bricolage programming means the feedback loop in Figure 6.2 is self-guided, possibly leading the programmer away from their initial motivation. This straying is likely, as the possibility for surprise is high, particularly when shifting from the inner loop of implementation to the outer loop of perception. The output of a generative art process is rarely exactly what we intended, and we will later argue in §6.5 that this possibility of surprise is an important contribution to creativity.

Representations in the computer and the mind are evidently distinct from one another. Computer output evokes perception, but that percept will both exclude features that are explicit in the output and include features that are not, due to a range of effects including attention, knowledge and illusion. Equally, a human concept is distinct from a computer algorithm. Perhaps a program written in a declarative rather than imperative style is somewhat closer to a concept (§4.3). But still, there is a clear line to be drawn between a string of discrete symbols in code, and the morass of both discrete and continuous representations which underlie cognition (§2.2).

There is something curious about how the programmer's creative process spawns a second, computational process. In an apparent trade-off, the computational process is lacking in the broad cognitive abilities of its author, but is nonetheless both faster and more accurate at certain tasks by several orders of magnitude. It would seem that the programmer uses the programming language and its interpreter as a cognitive resource, augmenting their own abilities in line with the extended mind hypothesis (Clark, 2008).

6.4 Symbols and Space

We have seen argued throughout this thesis that human conceptual representation centres around perception. Algorithms on the other hand are represented in discrete symbolic sequences, as is their output, which must go through some form of digital-to-analogue conversion before being presented to our sensory apparatus, for example as light from a monitor screen or sound pressure waves from speakers, triggering a process we call observation. Recall the artist-programmer from our case study (§6.3), who saw something not represented in the algorithm or even in its output, but only in their own perception of the output; observation is itself a creative act.

The remaining component to be dealt with from Figure 6.2 is that of programmers' *concepts* (§2.2.5). Figure 6.2 shows concepts mediating between spatial perception and discrete algorithms, leading us to ask; are concepts represented more like spatial geometry, like percepts, or symbolic language, like algorithms? Our focus on metaphor leads us to take the former view, that conceptual representation is grounded in perception and the body. This view is taken from Conceptual Metaphor Theory (§2.2.6), in particular that concepts are primarily structured by metaphorical relations, the majority of which are orientational, understood relative to the human body in space or time. In other words, the conceptual system is grounded in the perceptual system.

Gärdenforsian conceptual spaces (§2.2.5) are compelling when applied to concepts related to bodily perception, emotion and movement. However, it is difficult to imagine taking a similar approach to computer programs. What would the quality dimensions of a geometrical space containing all computer programs be? There is no place to begin to answer this question; computer programs are linguistic in nature, and cannot be coherently mapped to a geometrical space grounded in perception. Again we return to the point that spatial representation is not in opposition to linguistic representation; they are distinct but support one another. This is clear in computing, hardware exists in our world of continuous space, but thanks to reliable electronics, conjures up a linguistic world of discrete computation. Our minds are able to do the same, for example by computing calculations in our head, or encoding concepts into phonetic movements of the vocal tract or alphabetic symbols on the page. We can think of ourselves as spatial beings able to simulate a linguistic environment to conduct abstract thought and open channels of communication. On the other hand, a piece of computer software is a linguistic being able to simulate spatial environments, perhaps to create a game world or guide robotic movements, both of which may include some kind of model of human perception.

6.5 Components of creativity

We now have grounds to characterise how the creative process operates in bricolage programming. For this we employ the Creative Systems Framework (CSF; §6.2), introducing its terms as we go.

Within the CSF, a creative search has three key aspects: the conceptual *search space* itself, *traversal* of the space and *evaluation* of concepts found in the space. In other words, creativity requires somewhere to search, a manner of searching, and a means to judge what is found. However, creative behaviour may make use of introspection, self-modification and need boundaries to be broken. That is, the constraints of search space, traversal and evaluation are not fixed, but are examined, challenged and modified by the creative agent following (and defined by) them. The CSF characterises particular kinds of *aberration* from a conceptual space, and approaches to addressing them.

Using the terminology of Gärdenfors (2000), the search spaces of the CSF are themselves concepts, defining regions in a universal space defined by quality dimensions. Transformational creativity then is a geometrical transformation of these regions, motivated by a process of searching through and beyond them. This means that a creative agent may creatively push beyond the boundaries of the search as we will see. While acknowledging that creative search may operate over linguistic search spaces, we focus on geometric spaces grounded in perception. This follows our focus on artistic bricolage described in §6.3, but for an approach unifying linguistic and geometric spaces see Forth et al. (2010).

We may now clarify the bricolage programming process introduced in §6.3.1 within the CSF. As shown in Figure 6.3, the search space defines the programmer's concept, being their current artistic focus structured by learnt techniques and conventions. The traversal strategy is the process of attempting to generate part of the concept by encoding it as an algorithm, which is then interpreted by the computer. Finally, evaluation is a perceptual process in reaction to the output.

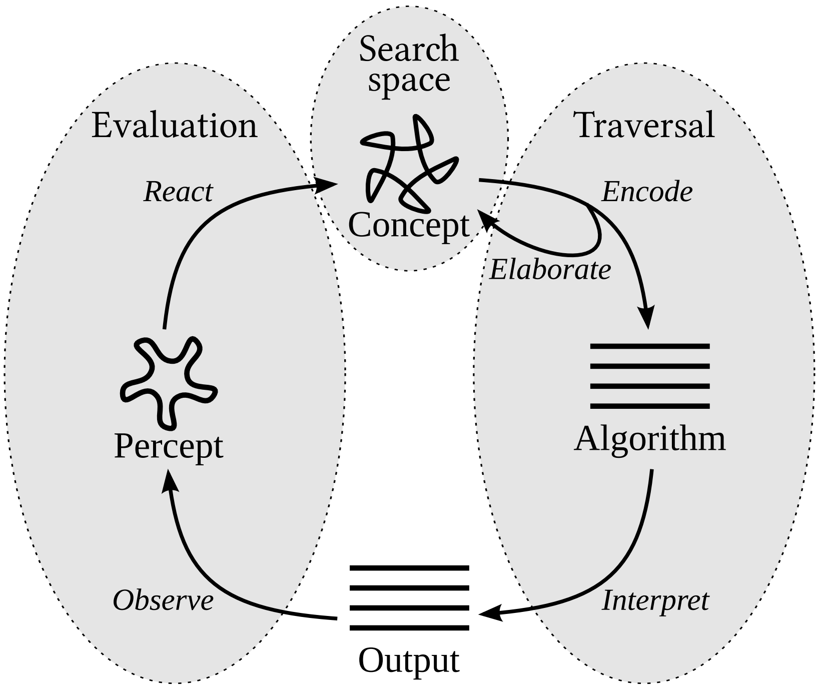


Figure 6.3: *The process of action and reaction in bricolage programming from Figure 6.2, showing the three components of the Creative Systems Framework, namely search space, traversal strategy and evaluation.*

In §6.3, we alluded to the extended mind hypothesis (Clark, 2008), claiming that bricolage programming takes part of the human creative process outside of the mind and into the computer. The above makes clear what we claim is being externalised: part of the traversal strategy. The programmer's concept motivates a development of the traversal strategy, encoded as a computer program, but the programmer does not necessarily have the cognitive ability to fully evaluate it. That task is taken on by the interpreter running on a computer system, meaning that traversal encompasses both encoding by the human and interpretation by the computer.

The traversal strategy is structured by the techniques and conventions employed to convert concepts into operational algorithms. These may include *design patterns*, a standardised set of *ways of building* that have become established around imperative programming languages. Each design pattern identifies a kind of problem, and describes a generalised structure towards a solution.[1]

The creative process is constrained by the programmer's concept of what is a valid end result. This is shaped by the programmer's current artistic focus, being the perceptual qualities they are currently interested in, perhaps congruent with a cultural theme such as a musical genre or artistic movement. Artists often speak of self-imposed constraints as providing creatively fertile ground. In terms of a creative search such constraints form the *boundary* of a search space. It is possible for a search to traverse beyond that boundary, thus finding *invalid* concept instances, a scenario called *aberration* (Wiggins, 2006a, §5.2.2). In such instances *transformational creativity* can be triggered. For example, if an invalid yet (according to evaluation rules) valued instance is found, then the concept should be enlarged to include the instance. An invalid concept instance which is not valued indicates that our traversal strategy is flawed and should be modified to avoid such instances in the future. A single traversal operation may result in both valid and invalid instances being found, indicating that both the traversal rules and the definition of the concept should be modified.

The artist in our earlier case study was working within the concept of bezier curves, but when curve endpoints happened to join, they perceived some things outside that concept – a squiggle and some hair. They made a value judgement, and decided to change their concept in response, which we consider as a case of transformational creativity. They then made edits to their source code (the traversal strategy), in order to try to generate output which evoked perceptual experiences closer to their concept. This may seem a minor case of transformational creativity, but indeed we contend that much creativity is quite ordinary human behaviour; humans apply creativity at all levels of life.

Our case study shows where a programmer may set themselves up for being surprised by the results. This is not only due to the use of pseudo-random numbers, after all noise is rarely a source of surprise; in information theoretic terms noise has maximal information content, but in practice the lack of form quickly results in fatigue or attention shift. It is rather due to the linguistic abstraction of an idea that consists of fragments of perceptual symbols. In other words, because the traversal strategy of a programmer includes external notation and computation, they are likely to be less successful in writing software that meets their preconceptions, or in other words more successful in being surprised by the results. A creative process that includes external computation will follow a less predictable path as a result. Nonetheless the process as a whole has the focus of a concept, and is guided by value in relation to a rich perceptual framework, and so while unpredictable, this influence is far from random, being meaningful interplay between language and perceptual experience. The human concepts and

[1]This structural heuristic approach to problem solving is inspired by work in the field of urban design (Alexander et al., 1977).

algorithm are continually transformed in respect to one another, and to perceptual affect, in creative feedback.

According to our embodied view, not only is perception crucial in evaluating output within bricolage programming, but also in structuring the space in which programs are conceptualised. Indeed if the embodied view of conceptual metaphor theory (§2.2.6) holds in general, the same would apply to all creative endeavour. From this we find a message for the field of computational creativity: a prerequisite for an artificial creative agent is in acquiring computational models of perception sufficient to both evaluate its own works and structure its conceptual system. Only then will the agent have a basis for guiding changes to its own conceptual system and generative traversal strategy, able to modify itself to find artefacts that it was not programmed to find, and place value judgements on them. Such an agent would need to adapt to human culture in order to interact with shifting cultural norms, keeping its conceptual system and resultant creative process coherent within that culture. For now however this is wishful thinking, and we must accept generative computer programs which extend human creativity, but are not creative agents in their own right.

6.6 Programming in Time

> "She is not manipulating the machine by turning knobs or pressing buttons. She is writing messages to it by spelling out instructions letter by letter. Her painfully slow typing seems laborious to adults, but she carries on with an absorption that makes it clear that time has lost its meaning for her." Sherry Turkle (2005, p. 92), on Robin, aged 4, programming a computer.

Having investigated the representation and operation of bricolage programming we now examine how the creative process operates in time. Dijkstra might argue that considering computer programs as operating in time at all, rather than as entirely abstract logic, is itself a form of the anthropomorphism examined in §2.3. However from the above quotation it seems that Robin stepped out of any notion of physical time, and into the algorithm she was composing, entering a timeless state. This could be a state of optimum experience, the *flow* investigated by Csikszentmihalyi where "duration of time is altered; hours pass by in minutes, and minutes can stretch out to seem like hours" (Csikszentmihalyi, 2008, p. 49). Perhaps in this state a programmer is thinking in algorithmic time, attending to control flow as it replays over and over in their imagination, and not to the world around them. Or perhaps they are not attending to the passage of time at all, thinking entirely of abstract logic, in a timeless state of building. In either case, it would seem that the human is entering time relationships of their software, rather than the opposite, anthropocentric direction of software entering human time. While programmers can appear detached from physical time, there are ways in which the time-lines of program development and operation may be united, as we saw in our discussion of notation in time (§5.2).

Temporal relationships are generally not represented in source code. When a programmer needs to do so, for example as an experimental psychologist requiring accurate time measurements, or a musician needing accurate synchronisation between processes, they run into problems of accuracy and latency. With the wide proliferation of interacting embedded systems, this is becoming a broad concern (Lee, 2009). In commodity systems time has been decentralised, abstracted away through layers of caching, where exact temporal dependencies and intervals between events are not deemed worthy of general interest. Programmers talk of "processing cycles" as a valuable resource which their processes should conserve, but they generally no longer have programmatic access to the high frequency oscillations of the central processing units (now, frequently plural) in their computer. The allocation of time to processes is organised top-down by an overseeing scheduler, and programmers must work to achieve what timing guarantees are available. All is not lost however, realtime kernels are now available for commodity systems, allowing psychologists (Finney, 2001) and musicians (e.g. via `http://jackaudio.org/`) to get closer to 'physical' time. Further, the representation of time semantics in programming is undergoing active research in a sub-field of computer science known as *reactive programming* (Elliott, 2009), ideas from which have inspired representation of time in the Tidal language (§4.5).

6.7 Embodied programmers

What we have seen provides strong motivation for addressing the particular needs of artist-programmers. These include concerns of workflow, where elapsed time between source code edits and program output slows the creative process. Concerns of programming environment are also important, which should be optimised for the presentation of shorter programs in their entirety to support bricolage programming, rather than hierarchical views of larger codebases. Perhaps most importantly, we have seen motivation for the development of new programming languages, pushing the boundaries to greater support artistic expression.

From the embodied view we have taken, it would seem useful to integrate time and space further into programming languages. In practice integrating time can mean on one hand including temporal representations in core language semantics, and on the other uniting development time with execution time, as we have seen with interactive programming. Temporal semantics and live coding both already feature strongly in some programming languages for the arts, as we saw in §6.6, but how about analogous developments in integrating geometric relationships into the semantics and activity of programming? It would seem the approaches shown in Nodal, the ReacTable and Texture described in chapter 5 are showing the way towards greater integration of computational geometry and perceptual models into programming language. This is already serving artists well, and provides new ground for visual programming language research to explore.

Earlier we quoted Paul Klee (§6.3), a painter whose production was limited by his two hands. The artist-programmer has different limitations, but shares what Klee called his limitation of reception, by the "limitations of the perceiving eye". This is perhaps a limitation to be expanded but not overcome, rather celebrated and fully explored using all we have, including our new computer languages. We have characterised a bricolage approach to artistic programming as an embodied, creative feedback loop. This places the programmer close to their work, grounding discrete computation in orientational and temporal metaphors of their human experience. However the computer interpreter extends the programmer's abilities beyond their own imagination, making unexpected results likely, leading the programmer to new creative possibilities.

6.8 Live Coding in Practice

Two live coding systems have been introduced in this thesis, the Tidal pattern DSL (§4.5) and the related visual language Texture (§5.6). We have discussed technical aspects of live coding in the process, but until now have not discussed the creative practice of live coding. The term *live coding* emerged around 2003, to describe the activity of group of practitioners and researchers who had begun developing new approaches to making computer music and video animation (Collins et al., 2003; Ward et al., 2004; Blackwell and Collins, 2005; Rohrhuber et al., 2005). It is defined by Ward et al. (2004) as "the activity of writing a computer program while it runs", where changes to the source code are enacted by the running process without breaks in musical or visual output. The archetypal live coding performance involves programmers writing code on stage, with their screens projected for an audience, their code dynamically interpreted to generate music or video.

Closely related terms are *interactive*, *on-the-fly* (Wang and Cook, 2004), *conversational* (Kupka and Wilsing, 1980), *just-in-time* (Rohrhuber et al., 2005) and *with-time* (Sorensen and Gardner, 2010) programming. Many of these terms are interchangeable, although there are differences of technique and emphasis, for example live coding is most often used in the context of improvised performance of music or video animation. This context of improvised computer music is adopted here, and although much of the following could be related to work in live video animation, focus on computer music is kept for brevity.

In live coding the performance is the *process* of software development, rather than its outcome. The work is not generated by a finished program, but through its journey of development from nothing to a complex algorithm, generating continuously changing musical or visual form along the way. This is by contrast to *generative art*, popularised by the generative music of Brian Eno (1996). Generative art may be understood by a gardening analogy, where processes are composed as 'seeds', planted in a computer and left to 'grow'. The random number generators often used to provide variation in generative processes have led to their being likened to the construction of wind chimes, in that they are structures that are 'played' by sources of noise. Like wind chimes, while generative art may constantly vary, generative systems which produce qualitative changes are rare. Output more or less follows the same style, with only a few permutations giving an idea of the qualities of the piece. This is well illustrated by our case study of an artist-programmer (§6.3), who ran their program a few times not to produce new works, but to get different perspectives on the same work.

With live coding, hands-on human involvement is essential to the development of a piece. Metaphorically speaking, rather than sowing seeds, live coders metaphorically construct plants from scratch by splicing different plants together, modifying their DNA while they grow, and experimenting with different ways of destroying them for artistic effect. With generative art, onlookers are often left to question whether the programmer or computer is the creative agent in the artistic process. However with live coding there is no question, the programmer very visibly provides all the rules, the human act of programming providing all creative impetus, and the computer process extending the human range of exploration.

Live coding allows a programmer to examine an algorithm while it is interpreted, taking on live changes without restarts. This unites the time flow of a program with that of its development, using dynamic interpretation or compilation. Using techniques outlined in (§5.2), live coding makes a dynamic creative process of test-while-implement possible, rather than the conventional implement-compile-test cycle. The creative processes shown in Figures 6.2 and 6.3 still apply, but are freed from the constraints of time, with the arrows now representing concurrent influences between components rather than time-ordered steps.

Live coding not only provides an efficient creative feedback loop, but also allows a programmer to connect software development with time based art. This is bricolage programming (§6.3.1) taken to a logical and artistic conclusion, particularly with archetypal 'blank slate' live coding. Here risk is embraced and pre-planning eschewed, the aim being to design a program 'in the moment' where it is implemented and executed in the expectant atmosphere created by an audience.

The primary research focus around live coding practice has been upon the integration of performance time with development time, for example in the live coding papers already cited. This is important work, as human perception of the progression of time during the evaluation of an algorithm has often been deliberately ignored in computer science (§6.6). This line of research is certainly not complete, but there are now several working approaches to improvising music through live code development. Some research emphasis has therefore moved from time to space, that is, to the consideration of visuospatial perception within the activity and spectacle of live coding performance.

6.8.1 "Obscurantism is dangerous. Show us your screens."

The present section title is taken from the manifesto drafted by the Temporary Organisation for the Promotion of Live Algorithm Programming (TOPLAP; Ward et al., 2004), a group set up by live coders to discuss and promote live coding. It neatly encapsulates a problem at the heart of live coding; live coders wish to show their interfaces so that the audience can see the movement and structure behind their music, however in positioning themselves against the computer music tradition of hiding behind laptop screens (Collins, 2003), they are at risk of a charge of greater obscurantism. Most people do not know how to program computers, and many who do will not know the particular language in use by a live coder. So, by projecting screens, do audience members feel *included* by a gesture of openness, or *excluded* by a gibberish of code in an obscure language? Do live coding performances foster melding of thoughts between performer and audience, or do they cause audience members to feel stupid? Audiences have not yet been formally surveyed on this issue, but anecdotal experience suggests both reactions are possible. A non-programmer interviewee in a BBC news item ("Programming, meet music", 28th August 2009) reported ignoring projected screens and just listening to the music, and less ambiguous negative reactions have been rumoured. On the other hand, a popular live coding tale has it that after enjoying a live coding performance by Dave Griffiths in Brussels (FoAM studios, 17th December 2005), a non-programmer turned to their lover and was overheard to exclaim "Now I understand! Now I understand why you spend so much time programming."

Partly in reaction to the issue of inclusion, a new direction of research into *visual programming* has emerged from live coding practice, evident in the systems reviewed and introduced in the previous chapter. The challenge is to find new ways of notating programs suitable not only for containing the expressions of a well-practiced live coder, but for doing so in a way meaningful to an audience.

6.8.2 Cognitive Dimensions of Live Coding

Blackwell and Collins (2005) have examined live coding with respect to the Cognitive Dimensions of Notation (CDN), using it to compare the ChucK language for programming on-the-fly computer music (Wang and Cook, 2004) with the commercial Ableton Live production software. ChucK, and by implication live coding in general, does not come off particularly well. It has low on the dimensions of **visibility**, **closeness of mapping** and **role-expressiveness**, is **error-prone** and requires **hard mental operations** in part to deal with its high level of **abstraction**. It would seem that the **progressive evaluation** and representational **abstraction** offered by ChucK come at a cost. Nonetheless, these are costs that many are willing to overcome through rigorous practice regimes reminiscent of instrumental virtuosos (Collins, 2007). They are willing to do so because abstraction, while taking the improviser away from the direct manipulation that instrumentalists enjoy, allows them to focus on the compositional structure behind the piece. Being able to improvise music by manipulating compositional structure in theoretically unbound ways is too attractive a prospect for some to ignore.

Established norms place the live coder in a stage area separate from their audience members[2], who depending on the situation, may listen and watch passively or interact enthusiastically, perhaps by dancing, shouting or screaming. We therefore have two groups to consider, the performers needing to work 'in the moment' without technical interruptions that may break creative flow (Csikszentmihalyi, 2008), and the audience members needing to feel included in the event, while engaged in their own creative process of musical interpretation (§4.4). There is a challenge then in reconsidering live coding interfaces, creating new languages positioned at a place within the CDN well suited for a broader base of musicians and audiences who may wish to engage with them. The question is not just how musicians can adapt to programming environments, but also the inverse; how may programming environments, often designed to meet the needs of business and military institutions, be rethought to meet the particular needs of artists? First, we should consider what those needs might be.

An interesting cognitive dimension with respect to live coding is **error-proneness**. There are different flavours of error, some of which are much celebrated in electronic music, for example the *glitch* genre grew from an interest in mistakes and broken equipment (Cascone, 2000). In improvisation, an unanticipated outcome can provide a creative spark that leads a performance in a new direction. We would classify such desirable events as semantic errors, in contrast with syntactic errors which lead to crashes and hasty bug-fixing.

In terms of the CDN, bricolage programming requires high **visibility** of components, in

[2]Performance norms are of course extensively challenged both inside (Rohrhuber et al., 2007) and outside (Small, 1998) live coding practice.

particular favouring shorter programs that fit on a single screen, and avoiding unnecessary **abstraction**. Here is a conflict – as noted above abstraction sets live coding apart from other approaches to improvisation in computer music, but also acts as an obstacle to bricolage programming. We are pulled in different directions, and so look for the happiest medium, a common result from taking a CDN perspective. Some programmers, known in some quarters as architecture astronauts, enjoy introducing many layers of abstraction that only serve to obfuscate (Spolsky, 2004). Bricolage programmers are the opposite in wanting to be as close to their work as possible. This is not however a case of removing all abstraction, but finding the right abstraction for the work. Programming after all is an activity that takes place somewhere between electric transistors and lambda calculus – the trick is finding the right level of abstraction for the problem domain (§4.3). Accordingly a computer musician may find having to deal with individual notes a distraction, and that a layer of abstraction above them provides the creative surface where they can feel closest to their composition.

6.9 Live coders on computational creativity

Creativity is often touched upon in the study of music, but rarely approached in detail. Musicians may worry that analysing their creative processes may somehow spoil them, as if self reflection can be destructive if approached in too formal a manner. The many points of view as to the nature of creativity, with common disagreement, may lead a scholar to lose interest and look for a better defined field of research. However such an important subject deserves attention, and light is being thrown by work within sub-fields of philosophy, psychology and artificial intelligence, which each contribute to form the cross-disciplinary field of computational creativity.

A survey was carried out with the broad aim of gathering ideas for study of computational creativity from a live coding perspective. An on-line discussion group for live coders hosted by the Temporary Organisation for the Promotion of Live Algorithm Programming (TOPLAP; Ward et al., 2004) was asked to fill out an on-line survey. To encourage honest responses the survey was anonymous, and demographic information was not collected. Discussion of creativity can revolve around ill-defined terms and invite prejudice. For this reason, the word 'creativity' was not used in the invitation or survey text, and questions addressing issues of creativity were mixed with general questions about live coding. A total of 32 completed the survey (although not all answered every question), and 30 of whom indicated at the end that they were happy for their answers to be published under a creative commons attribution license. The survey and responses were in English as the language used by the discussion group, although for several this was a second language.

6.9.1 The subjects

The respondents were a broad cross-section of live coders, with users of the six pre-eminent live coding environments represented, between five and fourteen for each system (many had used more than one). A large proportion (22/32) had used one of the listed classes of traditional musical instruments, with comments suggesting a higher percentage would have resulted if electronic instruments were included. From this we can assume a group with a generally rich musical background. Relatedly, almost all (30/32) of subjects live coded music, whereas only a small proportion (6/32) live coded video animation; live coding would appear to currently be a music led culture. There were a diverse range of approaches to the question of how to define live coding in one sentence, the results are rather unquantifiable but the reader is referred to Appendix A.1 to enjoy the responses. While the responses show some diversity of approach to live coding, because the subjects had all used at least one of the main languages it is safe to assume that they are working to largely the same technical definition.

6.9.2 Creating language

Computer users often discuss and understand computer programs as tools, helping them do what they need efficiently and without getting in the way. For a programmer it would instead seem that a computer language is an immersive *environment* to create work in. Indeed a suite of source code editing software is collectively known as an Interactive Development Environment (IDE). It is interesting then to consider to what extent live coders adapt their computer languages, personalising their environments, perhaps in order to aid creativity. Over two thirds (21/32) collected functions into a library or made an extensive suite of libraries. This is analogous to adding words to a language, and shows the extent of language customisation. A smaller proportion (6/32) had gone further to implement their own language interpreter and smaller number still (5/32) had designed their own language. That these artists are so engaged with making fundamental changes to the language in which they express their work is impressive.

6.9.3 Code and style

From the perspective of computational creativity, it is interesting to examine the relationship that live coders have with their code. An attempt at quantifying this was made by asking, “When you have finished live coding something you particularly like, how do you feel towards the code you have made (as opposed to the end result)?” Over half (17/32) indicated that code resulting from a successful live coding session was a description of some aspect of their style.

This suggests that many feel they are not encoding a particular piece, but how to make pieces in their own particular manner. Around the same number (15/32) agreed that the code describes something they would probably do again, which is perhaps a rephrasing of the same question. A large number, (24/32) answered yes to either or both questions. There are many ways in which these questions can be interpreted, but overall this suggests that many subjects feel they have a stylistic approach to live coding that persists across live coding sessions, and that this style is somehow represented in the code they make.

6.9.4 Live coding as a novel approach

The subjects were asked the open question "What is the difference between live coding a piece of music and composing it in the sequencer (live coding an animation and drawing one)? In other words, how does live coding affect the way you produce your work, and how does it affect the end result?" The answers are difficult to summarise, and so again the reader is directed to Appendix A.2 to read the full responses. Some interesting points relevant to computational creativity are selectively quoted for comment here.

> "I have all but [abandoned] live coding as a regular performance practice, but I use the skills and confidence acquired to modify my software live if I get a new idea while on stage."

This admission, that getting new ideas on stage is infrequent, makes an important as well as humble point. In terms of the Creative Systems Framework, we can say that live coding is useful in performance if you need to modify your conceptual space (the kind of work you want to make), or your traversal strategy (the way you try to search for or make it). If you are content with having both fixed in advance, then live coding is not warranted. In other words, live coding is useful for invoking transformational creativity, although as with this test subject, transformational creativity is not always desirable in front of a paying, risk-averse audience.

> "When I work on writing a piece … I can perfect each sound to be precisely as I intend it to be, whereas [when] live coding I have to be more generalised as to my intentions."

This respondent is making the point that live coders work at least one level of abstraction away from enacting individual sounds.

> "Perhaps most importantly the higher pace of livecoding leads to more impulsive choices which keeps things more interesting to create. Not sure how often that also creates a more interesting end result but at least sometimes it does."

This is interesting with reference to the creative feedback loop of bricolage programming (§6.3.1). Live coding allows a change in code to be heard or seen immediately in the output, with no forced break between action and reception. This is in stark contrast to those whose experience of software development as slow and arduous.

> "Live coding has far less perfection and the product is more immediate. It allows for improvisation and spontaneity and discourages over-thinking."

This come as a surprise, as live coding has a reputation for being cerebral and over technical. In reality, at least when compared to other software based approaches, the immediacy of results fosters spontaneous thought.

> "Live coding is riskier, and one has to live with [unfit decisions]. You can't just go one step back unless you do it with a nice pirouette. Therefore the end result is not as clean as an "offline-composition", but it can lead you to places you [usually] never would have ended."

This comment is particularly incisive; the peculiar relationship that live coders have with time does indeed give a certain element of risk. Riskier ways of making music are more likely to produce aberrant output, not of the type you were looking for. However, where such output turns out to be valuable, then you have the opportunity to redefine what you are looking for, transforming your conceptual space. If the output turns out to be poor, then you can at least change the way you work to avoid similarly poor output in the future; mechanisms of transformational creativity.

> "... while live coding is a performance practice, it also offers the tantalising prospect of manipulating musical structure at a similar abstract level as 'deferred time' composition. To do this effectively in performance is I think an entirely different skill to the standard 'one-acoustic-event-per-action' physical instrumental performance, but also quite different to compositional methods which typically allow for rework."

This really gets to the nub of what live coding brings to the improvising artist – an altered perspective of time, where a single edit can affect all the events which follow it.

6.9.5 Computational creativity

In using programming languages to make music, live coders have a unique perspective on the question of computational creativity. It is interesting then to measure the extent of optimism for computers taking a greater role in the creative process than they already do. Towards this the subjects were given a series of statements and asked to guess when each would become true, with the pertinent results in Figure 6.4. Regrettably there was a configuration error early on in the surveyed period, requiring the answers of two subjects to be discarded.

Optimism for the statement *"Live coding environments will include features designed to give artistic inspiration to live coders"* was very high, with just over half (14/27) claiming that was already true, and almost all (25/27) agreeing it would become true within five years. This indicates strong support for a weak form of computational creativity as a creative aid for live coders.

Somewhat surprisingly, optimism for the stronger form of creativity suggested by *"Live code will be able to modify itself in an artistically valued manner"* was also high, with two fifths (11/28) claiming that was already possible. If that is the case, it would be appreciated if the live code in question could make itself known. Perhaps they are referring to *feedback.pl*, a live coding editor for the Perl programming language. This editor does indeed allow self-modifying code, but we are some way off from seeing an artistic computational agent emerge from it.

More pessimism is seen in response to *"A computer agent will be developed that produces a live coding performance indistinguishable from that of a human live coder"*, with a third (9/27) saying that this will never happen. This question is posed in reference to the imitation game detailed by Turing (1950), however our version involving musical rather than language based imitation seems rather easier to fulfil. As one subject commented, "the test indistinguishable from a human is very loose and there can be some very bad human live coding music." That would perhaps explain why half (13/27) of respondents thought the statement was either already true or would become so within five years.

6.9.6 Discussion

What if a computational approach to the musicology of live coding were to develop, where researchers deconstruct the code behind live coding improvisations as part of their work? Correlations between expressions in formal languages and musical form in sound could be identified, and the development of new ways of expressing new musical forms could be tracked. If successful, the result need not be a new kind of music, but could be music understood in a novel way. Perhaps this new computational approach to understanding music that could prove invaluable in the search for a musically creative software agent.

In looking at creativity through the eyes of live coders, we can see some promise for computational creativity even at this early stage of development of both fields. Live coders feel their musical style is encoded in the code they write, and that their language interfaces provide them with inspiration. They are actively developing computer languages to better express the music they want to make, creating computer language environments that foster creativity.

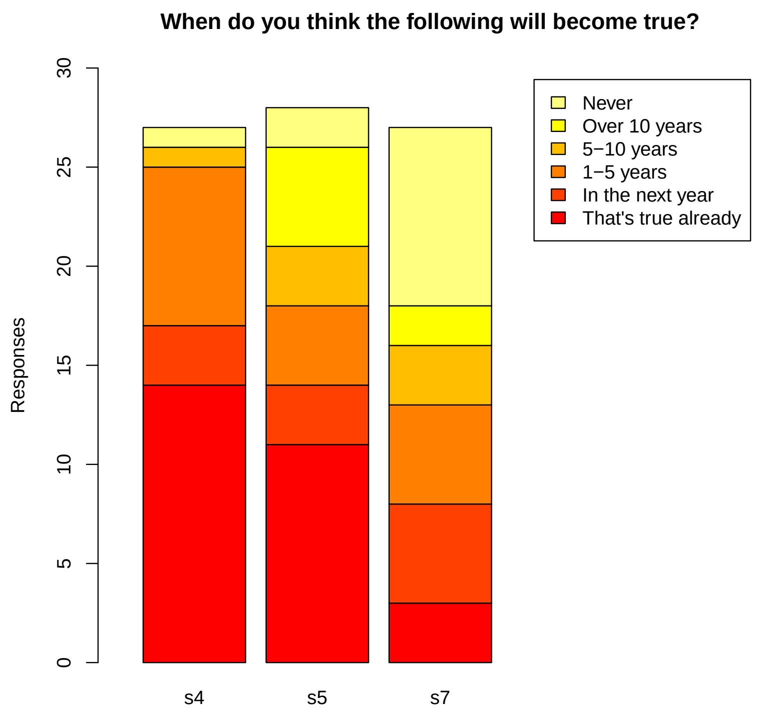


Figure 6.4: *Responses to statements s4: "Live coding environments will include features designed to give artistic inspiration to live coders", s5: "Live code will be able to modify itself in an artistically valued manner" and s7 "A computer agent will be developed that produces a live coding performance indistinguishable from that of a human live coder."*

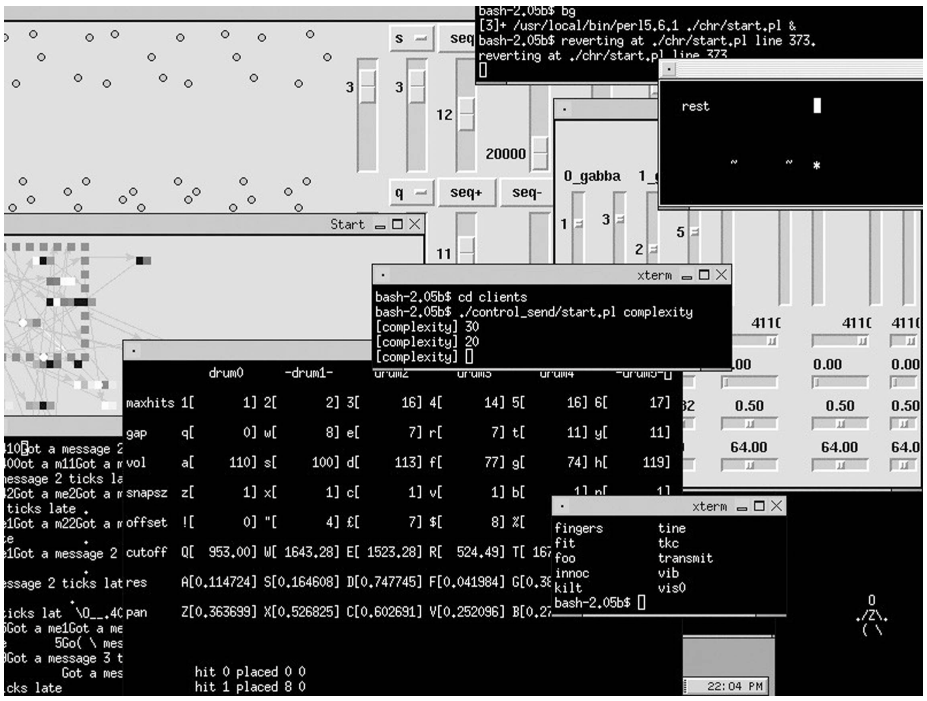
From here it is easy to imagine that live coding environments could become more involved in the creation of higher order conceptual representations of time-based art that live coders are concerned with. Perhaps this will provide the language, environment and application in which the creative processes of a computational agent will one day thrive.

6.10 Slub

The band *Slub* begun in the year 2000, as a collaboration between Adrian Ward (`http://www.adeward.com/`) and the present author. They both shared a desire to make music and enthusiasm for programming, and resolved to combine them.

> So if you've got programming skills and enjoy making music, it makes sense to combine them - just like combining pottery and integral mathematics can be stimulating. (Adrian Ward in interview with Shulgin, 2003)

Slub established a clear aim early on, to make people dance to their algorithms. They first met this aim in the Paradiso club during the 2001 Sonic Acts festival in Amsterdam, the first of many appearances at major European festivals.


```
bash-2.05b$ bg
[3]+ /usr/local/bin/perl5.6.1 ./chr/start.pl &
bash-2.05b$ reverting at ./chr/start.pl line 373.
reverting at ./chr/start.pl line 373
```

```
rest
```

```
bash-2.05b$ cd clients
bash-2.05b$ ./control_send/start.pl complexity
[complexity] 30
[complexity] 20
[complexity] 
```

```
          drum0       -drum1-
maxhits 1[      1] 2[      2] 3[     16] 4[     14] 5[     16] 6[     17]
gap     q[      0] w[      8] e[      7] r[      7] t[     11] y[     11]
vol     a[    110] s[    100] d[    113] f[     77] g[     74] h[    119]
snapsz  z[      1] x[      1] c[      1] v[      1] b[
offset  ![      0] "[      4] £[      7] $[      8] %[
cutoff  Q[ 953.00] W[ 1643.28] E[ 1523.28] R[ 524.49] T[ 167
res     A[0.114724] S[0.164608] D[0.747745] F[0.041984] G[0.3
pan     Z[0.363699] X[0.526825] C[0.602691] V[0.252096] B[0.27

          hit 0 placed 0 0
          hit 1 placed 8 0
```

```
10Got a message 2
00ot a m11Got a m
message 2 ticks la
2Got a me2Got a m
 ticks late .
1Got a m22Got a m
e
1Got a message 2
ssage 2 ticks lat
icks lat  \O__.4C
Got a me1Got a me
      5Go( \ mes
Got a message 3 t
       Got a mes
cks late
```

```
fingers        tine
fit            tkc
foo            transmit
innoc          vib
kilt           vis0
bash-2.05b$ 
```

Figure 6.5: *A screenshot showing a typical performance desktop used by the present author circa 2003.*

An early Slub system is described in detail by Collins et al. (2003). In brief it featured a synthesiser and early live coding system written by Ward, and a number of beat and bass-line generating programs written by McLean. Although their primary aim was musical, Slub enjoyed being faced with the challenge of being accepted as programmers who make music. To this end they began projecting their screens, confronting audiences with the conceptual overlap between their hand-crafted software and the music they produced using it.

> This whole network of software was written by slub to fulfil their individual needs, forming an environment ideal to their methods of working. This brings us to an important point – that the code is not just running in the computer. To explain; when programmers are watching a computer execute their own programs, the code is also executing in their minds. They have intimate knowledge of the process, and so can imagine it running. In this way, the code is alive in slub and in their computers, and hopefully also in their sound and the audience too. The music is also alive in these four places. As far as slub are concerned, code *is* music. (Collins et al., 2003, pp. 323–324)

Slub were not just challenging their audiences, but also the use of computers in the arts in general. Ward was instrumental in challenging the concept of software as a passive 'tool' with his software *Auto-Illustrator* (Ward et al., 2002). By drawing a satire on the widely used Adobe Illustrator software, Auto-Illustrator confounded expectations, making it difficult to for example draw a straight line. This brought attention to the aesthetic decisions built into all software, leading the Transmediale Software Art jury to award it first prize in 2001. This point was further explored by McLean with *forkbomb.pl*, a simple program which produced patterns while pushing a computer system to its limits; in effect visualising low-level interactions composed by human system programmers. This work was also awarded the Transmediale Software Art award, and both artworks were exhibited in the touring 2002/2003 *Generator* group show alongside works by Sol Lewitt and Yoko Ono.[3]

Following the formation of TOPLAP, Ward focused on live coding (creating *Pure Events*, based on the *Tracker* paradigm; Nash and Blackwell, 2011), and McLean created his own live coding environment, the *feedback.pl* editor (McLean, 2004) for self-modifying code (§5.2). From then on, Slub performed only using live coding interfaces. Having developed his own live coding practice in parallel, Dave Griffiths (`http://pawfal.org/dave/`) joined Slub following a key joint performance at Sonar festival in 2005. Dave took the Slub aesthetic in a new, audio/visual direction, by developing game-like live coding environments for music (McLean et al., 2010).

6.10.1 Reflections following a Slub performance

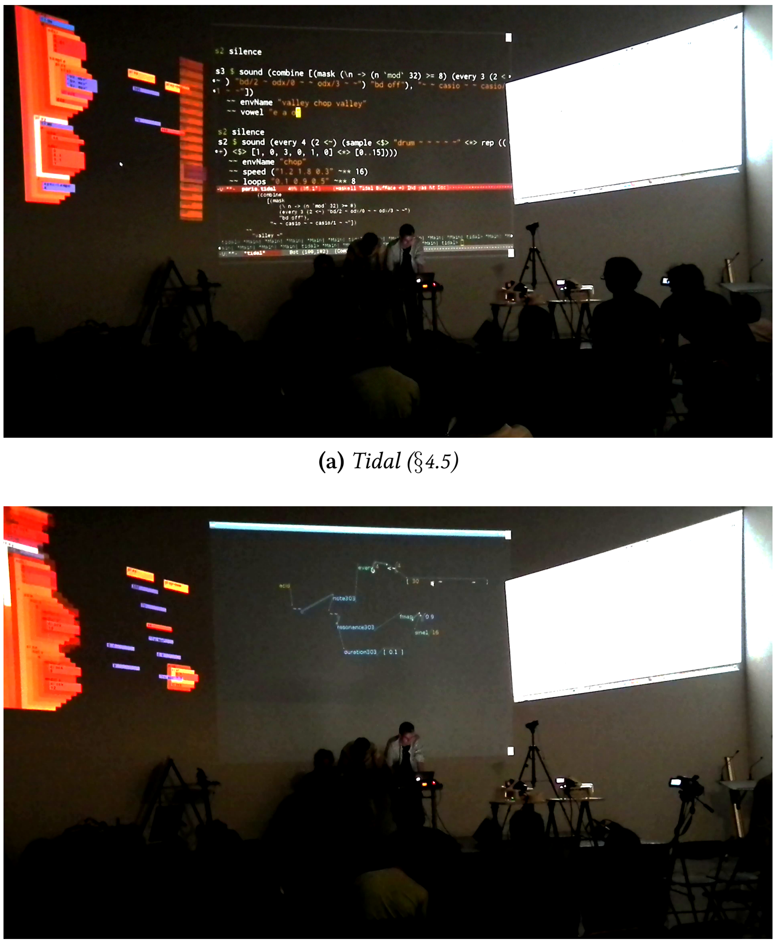
Slub performed at the Maison Rouge, Paris on the 30th September 2011, invited by the Sony Computer Laboratory in Paris as part of their 15th anniversary celebrations. Desk recordings of the performance are included on the enclosed DVD, which includes a mix of all three performers, and separate recordings of each performer. The present author interviewed Griffiths and Ward in the days following the performance, to reflect upon the performance and the journey that Slub had taken to their current practice.

> Dave Griffiths (DG): [Live coding] seems to define us at the moment, although it wasn't always that way. I think perhaps it will grow less important with time. The code we base performances on seems very unstable in that it's constantly being rewritten in different languages, we individually switch between visual and text based programming and back again - lots of little experiments. Despite this the music seems to grow in a independent manner, we somehow retain the knowledge of where we are going during a live performance and what we are doing as performers.

There are two strands of development then, musical and technological. But in Slub performances the code is displayed, and the two strands are intertwined. How important is this?

> Adrian Ward (AW): It's important to display our screens. But I'm not convinced projections are the best way to do this as they come with quite a bit of baggage –

[3]`http://www.generative.net/generator/`



```
s2 silence

s3 $ sound (combine [(mask (\n -> (n `mod` 32) >= 8) (every 3 (2 <
~ ) "bd/2 ~ odx/0 ~ ~ odx/3 ~ ~") "bd off"), "~ ~ casio ~ ~ casio/
1 ~ ~"])
   ~~ envName "valley chop valley"
   ~~ vowel "e a o"

s2 silence
s2 $ sound (every 4 (2 <~) (sample <$> "drum ~ ~ ~ ~ ~" <*> rep ((
+) <$> [1, 0, 3, 0, 1, 0] <*> [0..15])))
   ~~ envName "chop"
   ~~ speed ("1.2 1.8 0.3" ~** 16)
   ~~ loops "0.1 0.9 0.5" ~** 8
```

(a) *Tidal (§4.5)*

(b) *Texture (§5.6)*

Figure 6.6: *Slub at the Maison Rouge, Paris, 2011. Dave Griffiths and his SchemeBricks (McLean et al., 2010) interface are on house left, Adrian Ward and his Pure Events software are shown house right. The present author's screen is displayed in the centre, using software named in the above captions.*

> they're too passive, they reinforce the author/audience hierarchy, they're boring. However, technical and practical means make anything else unfeasible.
>
> DG: For me the projection is more important than the sound. The passivity is a problem, my favourite gigs are when people come up and me what I'm doing. I like the breakdown of the hierarchy very much, and make a point to explain and talk to them.

The role that projections play in live coding is problematic (§6.8.1), reflected here in Ward's unease, indicating that it is important to show screens, but that in doing so deep issues are unearthed. In stark contrast Griffiths, who has a background in fine art and computer games research, sees the projection of code as more important than the sound. This is a surprising statement, but when asked to expand upon his point, Griffiths showed fundamental agreement with Ward; what is of prime importance is not the projection itself, but that performances should centre around activity.

> DG: I would like to try being in the middle of the audience and projecting on the floor around us. Or maybe in small performances, make use of more of a physical material – electronics, tangible computing or some use of computer vision could be a way to remove the role of the screen, and projection. I think workshop-performances are something to follow up (thinking of your Textual workshop at Access Space, Alex) if you have 20 or so people joining in the projection is not important any more.
>
> In more usual situations, perhaps we think about what the absolute minimum we need to project would be - perhaps just the characters as we press the keys, or the expression the interpreter is currently executing. I have a feeling if we distill it down it could end up being more meaningful to an audience than seeing everything, all the time.

Slub ensure they perform side-by-side, allowing inter-performer communication which they deem important:

> AW: Yes. And [we] use gestures, too - pointing, laughing, dancing etc.
>
> DG: Agreed - I'm not sure what we talk about, it can be quite minimal like "shall we stop soon" or "why am I the only one making sound?" but I like it a lot and I think it's really important to be able to do that.

In live coding culture, "from-scratch" coding is held as an ideal, however Slub describe a more relaxed approach, with improvisation taking a starting point from pre-prepared or practiced elements:

> AW: My preparation usually takes the form of preparing sound samples, and re-learning what I've forgotten (awkward JavaScript syntax, mostly). The actual performance is a lot of improvisation but I'm not strict enough with myself to deny using previously written code, which I know is a bad habit.
>
> DG: I generally will have a couple of different starting points rehearsed, a set of possible things I will build and a familiarity with the interface built up before a gig. I think rehearsal in livecoding is really about making sure you can create a situation where you have lots of branching points for improvising from. It's important to be comfortable enough that your code will provide some predictable results when you need them, but flexible enough that you can dive into the unknown and see what happens.

This fits with the well established view of improvisation as being an exploration of pre-established musical schemata, which are not necessarily changed or deviated from during a performance (Pressing, 1987).

6.11 Discussion

Live coders such as Slub are shifting the discourse around the use of computers in music performance, for example taking the leading researcher in electronic music Simon Emmerson by surprise:

> The most unexpected interface for live sound control to have reemerged in recent years is the *lexical*; reading and writing. The increasing dominance of graphic interfaces for music software obscured the continuing presence of the command-line tradition, the code writer, the hacker. (Emmerson, 2007b, p. 115)

In an on-line research statement "Live electronics and the acousmatic tradition" (2001), Andrew Deakin reacted to a comment made by Emmerson during a radio discussion:

> Emmerson suggested that now we have affordable and 'realtime' computer-based systems there is really only one remaining area needing development – interface and/or instrument design. I could not agree more, a 400 year-old keyboard layout is surely not the answer.

What was missing from this discussion is recognition of the importance of *language*, which the computer allows us specialised access to. We must not allow the directness of embodied interfaces to mask the importance that the development of language, including computer language, has to the development of humanity. By nature, human culture continually re-creates itself, and as computers play an increasing role in society, including in music culture, programmers have found themselves in an increasingly privileged position. Douglas Rushkoff puts this well:

> Digital technology is programmed. This makes it biased toward those with the capacity to write the code. In a digital age, we must learn how to make the software, or risk becoming the software. It is not too difficult or too late to learn the code behind the things we use – or at least to understand that there is code behind their interfaces. Otherwise, we are at the mercy of those who do the programming, the people paying them, or even the technology itself. (Rushkoff, 2010, p. 128)

But even in 2011, the importance of programming languages are still not felt in music interaction research. While delivering a paper on live coding at the New Interfaces for Musical Expression conference (Aaron et al., 2011), Sam Aaron expressed shock that the word language was not amongst the top 200 conference paper keywords.[4] We are still developing programming as a creative process, there is still more to be done, and much more to be gained.

[4]A video recording of Sam Aaron's presentation is available at `http://vimeo.com/26905683/`.

Chapter 7

Future directions

The present research has been speculative and integrated with ongoing practice, and so as endings are beginnings, conclusions also serve as introductions. In the following we take stock of the present cycle of research, consider its impacts, and where the following cycle may take us next.

7.1 The Freedom of Interpretation

Where computation is at times hidden and forgotten in computer art, we forget some of its radical nature. Through programming, artists are able to create pure, linguistic abstractions, then ground them in human senses through actuators in striking ways. We have seen how a programming language interpreter can fit into a human creative process, performing the linguistic structures we give it, to generate output to evoke human perception. It seems however that the freedom to interpret programs is under threat, where consumer computers designed for entertainment are taking the place of general purpose machines. Just as underlying computer languages are often hidden in digitally-realised arts, they are being made increasingly inaccessible in end-user computing in general.

Without interpreters, we would not have software, but yet interpreters are also software. This is why we talk about *bootstrapping*, where software pulls itself from the floor by its bootstraps, a paradox settled by the existence of hardware microcode. This is also why Naur (1992a) prefers the word 'Dataology' to the phrase 'Computer Science'; programs are data, which operate over data.

Any piece of software exists as a combination of two parts, some instructions in a computer language and an interpreter of that language. Alone they do nothing, put them together and they can notionally do anything. Often there are intermediary steps, commonly compilation into *bytecode*, but these are just translations into another language, which still require interpreting as a sequence of instructions for the magic to happen.

Interpreters allow us to try out ideas beyond our imaginations, adding some instructions, interpreting them to get output rendered as sound or light to our senses, perceiving otherwise impossible worlds, and returning to the source code to twist the encoded structures into new contortions inspired by the results so far. We expand the realms of perception through computation, not creating things but writing about ideas in order to try to invoke them. We are only scratching the surface of what is possible, artistic and otherwise, from marrying high speed computation with embodied human experience.

It is of concern then that the freedom of thought given by interpreters happens to threaten business models of large companies, who are accordingly searching for the power to make free access to them illegal on the computers they produce. *Games consoles* are computers where the end user is not allowed access to an interpreter, unless they pay for an often prohibitively expensive license. You are not otherwise allowed to modify code, certainly not allowed to modify the interpreter, and so must be satisfied with using whatever programs the manufacturer allows you to.

Furthermore these business models are spreading, from computer games, to handheld computers and now to tablet computers. iOS, the operating system for the iPhone and iPad, was a particular shock as a device coming from Apple Computer, Inc., a company producing hardware traditionally marketed at the creative. However to develop and distribute software for these platforms you must pay an annual license. The iOS terms have now relaxed to the point that Apple have allowed the distribution of Codea (`http://twolivesleft.com/Codea/`), a tablet based programming environment for the Lua language. This is a welcome development, but still only exists within the terms of Apple's business model, and so the ability to share programs with others requires the purchase of Codea, and cumbersome transfer of source code listings.

This issue is related to the notion of *generativity* introduced by Zittrain (2009) to describe components of computer systems which are fundamentally incomplete.[1] Zittrain gives the operating system as one example, as a system which need not be tied to particular underlying systems (i.e. the hardware layer) or constrain systems running on it (i.e. the application layer). In other words, generative systems are designed to be used in ways which the original designers do not necessarily anticipate. Zittrain (2009) sees generativity as under threat due to security issues which generative systems are prone to, but the threat also comes from aesthetic and moral control; for example Apple have blocked software from distribution on the basis of aesthetic appearance, depiction of sexuality and for political commentary in the form

[1]This use of the word *generative* is distinct from the use in *generative art* (§1.1.2).

of ridicule of public figures. As Lessig (2006) argues, those in control of code effectively set the rules for everyone else. It is no surprise then to see that in following their vision for a hardware platform, vendors seek to limit and control the ability to program it.

Interpretation is of central concern in creative use of computers, and so the creep towards centralised, corporate control over interpretation is deeply worrying. The home computer revolution in the 1980s brought BBC Microcomputers and Sinclair Spectrums into schools and the home, which encouraged programming from the moment they were switched on. Using these computers empowered children to create with language under their own terms (Turkle, 2005). It would be a great shame if this exposure to programming was lost. One way to protect the freedom of interpretation may simply be through the development of novel approaches to programming language design. If new ways of programming engage end-users, then consumer pressure may be enough to preserve their freedom to program. This can already be seen in the development of Codea, which is hopefully just the beginning of a software genre, engaging end users in programming in and of these new platforms.

7.2 Software engineering standards

As software takes an ever increasing role in structuring life in the developing world, considerable work has been put into the processes of software engineering. Methodologies of software design promote methods to streamline the specification, planning, development, testing, and deployment of software. However like the design of programming notations (§5.1), standard development practices (such as the application of ISO 9000 to software engineering) are designed within particular constraints of commercial development which may not necessarily apply to more experimental or creative situations. Bricolage programming (§6.3.1) is a case in point, where specifications are defined by the act of programming, rather than vice-versa. As the programmer is engaged in end-user programming, there is no other user to satisfy, and formal discussions with audience members are hardly practical. The design goals are therefore internal and highly changeable.

Practices such as pair programming and test-driven development are intended to make development both reliable and responsive to requirements, and are collectively known as *agile* programming. However built-in assumptions of reaction to external requirements, rather than an individual's exploration and experimentation, mean they are inappropriate for the processes of artist-programmers, who may quite rightly resist formalisation of their personal creative processes, which may carry important aspects of their artistic style.

Bricolage programming is in stark opposition to the test-driven development movement, where a suite of small programs are written to test new features and fixes *before* they are implemented. A program is then judged to be feature complete when all the tests pass. This is akin to describing a program's behaviour twice, firstly to outwardly test it and secondly to inwardly implement it. Rather than developing programmatic definitions of behaviour before implementation, bricolage programmers make decisions on the basis of behaviour *while* implementing it. These are different ways of navigating a problem space, test-driven development is useful for arriving at a pre-arranged target, whereas bricolage programming is useful for exploring, looking for novel outputs that are valuable in potentially surprising ways.

Another point of diversion is the manner in which programmers collaborate. In industry, there is strong emphasis on group work in teams, with group overriding individual identity, particularly where individuals are employed on fixed term contracts. In the arts programmers may be employed on larger projects under similar terms, as technical, artist assistants. However the artist-programmer often works alone, to realise their own individual works. Practices such as *pair programming*, where two programmers work together on one computer, are rather alien to this situation. While artists and musicians do work together on large, free/open source projects, the development is usually centred around a particular individual. This is even the case in large projects, such as the development of programming languages for creative work. This was the case both for Impromptu and SuperCollider, where development followed a pattern where the first years of development is conducted by a single individual. Only when their vision has been fully scoped out is the codebase opened, and a wider free/open source software development community formed.

7.3 Cyclic revision control

We have seen how artist-programmers may have unconventional software design processes (§6.3), placing particular demands on their languages and tools. Revision control systems are among the most important members of a programmer's toolbox, allowing the history of a program to be annotated and managed. Modern revision control systems allow different programmers to work on their own 'branches' of code, which are then merged back into the main flow at a later date. Where problems arise in development, revision control systems can help developers understand the source of the problem, by providing historical reference. These systems already have much to offer artist-programmers, but could revision control be rethought to better meet their particular needs?

Consider a live coder, writing software to generate a music performance. In terms of revision control they are in an unusual situation. Normally we think of programmers making revisions towards a final result or milestone, at which point they 'ship', packaging and releasing their code for others to use. For live coders, every revision they make is part of the final result. In this case nothing gets shipped, as they are already the end users. We might somewhat crassly think of live coding in terms of shipping a product to an audience, but really what is being shipped is not software, but a software development process, as musical development.

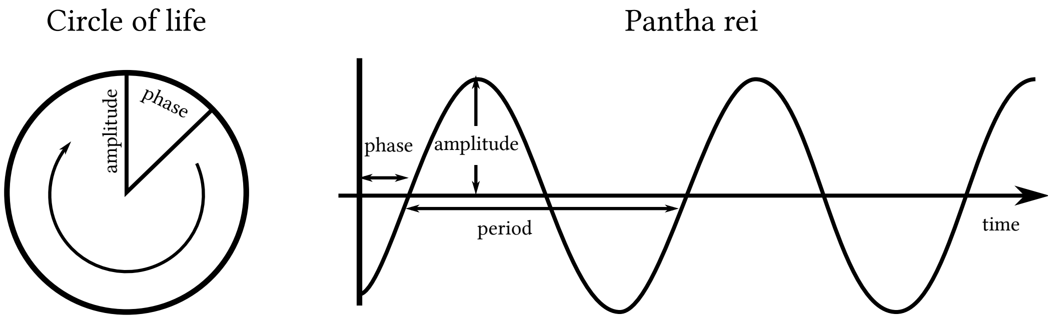
Also unusual in live coding revisions is that whereas conventional software development may begin with nothing, and finish with a complete, complex structure, a live coder both begins and ends with nothing. Rather than aim for a linear path towards a predefined goal, musicians instead are concerned with how to return back to nothing in a satisfying arc. A live coder may begin their performance with a blank editor, and be faced with an immediate decision about what to build, and how. As they improvise by encoding and modifying musical ideas, they eventually progress towards a conclusion. Their final challenge is how to end; some increase complexity to a crescendo and finish abruptly, and others decrease complexity to the minimum, before final reduction back to an empty editor and silence.

There are two ways of thinking about time, either as a linear progression, or as a recurrent cycle or oscillation, as shown in Figure 7.1. These approaches are not mutually exclusive, they rather provide different ways of looking at the same temporal processes. Conventional software design processes are characterised in terms of cycles of development, with repeating patterns between milestones. Nonetheless, it is not conventional to think of the code itself ending up back where it started, while during music performance, we often do return to prior states. We are all familiar with chorus and verse structure for example, and performances necessarily begin and end at silence.

It may be that if we reconsider code development in terms of time cycles rather than linear progression, then we could find new ways of supporting software development of music and other time based arts. Without further speculative research it is difficult to visualise what the outcome of this might be, but there is already some active, radical work in the representation of development time. For example revision control is a central part of the Field language environment (`http://openendedgroup.com/field/`), and allows the programmer to 'scrub' the development timeline. It seems however that in such experimental systems, the revision control timeline has so far been treated as a linear structure, with occasional parts branching and re-meeting the main flow later on. It is not unheard of for timelines to feed back on themselves in conventional software development, a process called *backporting*, but this is generally avoided, only done in urgent circumstances such as in applying security fixes to old software versions.

What if instead of being linear, software development timelines were of cycles within cy-

Figure 7.1: *Below figure and caption reproduced from Buzsaki (2006, pg .7) with permission*



Oscillations illustrate the orthogonal relationship between frequency and time and space and time. An event can repeat over and over, giving the impression of no change (e.g., circle of life). Alternatively, the event evolves over time (pantha rei). The forward order of succession is a main argument for causality. One period (right) corresponds to the perimeter of the circle (left).

cles, with revision control designed not to aid progression towards future features, but help the programmer wrestle their code back towards the state it was in ten minutes ago, and ten minutes before that? We leave this as a question for future research, but suggest the result could be a system that better supports thematic development in music and video animation.

7.4 Conclusion

"My view is that today's computer world is based on techie misunderstandings of human thought and human life, and the imposition of inappropriate structures ... on things we want to do in the human world." Ted Nelson (2008; `http://youtu.be/zumdnI4EG14`)

Day to day, it does not particularly matter that we do not know the mechanisms behind our own actions, we can simply learn through doing. The need for theory is sometimes derided in music culture, the long-lived phrase "writing about music is like singing about economics" traced back nearly 100 years (H. K. M., 1918), more recently taking the form of comparison with dancing about architecture. The sentiment is that music is about activity, and not theorising.

Theorising is difficult for artist-programmers to avoid however, including those making music. In order to write a generative music program, an artist-programmer must necessarily theorise and encode musical structure in language. Indeed writing computer programs to make music is a form of writing about music: it requires the introspection, abstraction and formalisation that many have derided by comparison with architectural dance. Not only is writing about the processes of art necessary, but in the process, coming to greater understanding of ourselves allows us to reach beyond what would otherwise be possible.

With introspection as our motivation, we have tried to characterise the inner processes of artist-programmers, in terms of intertwined analogue and digital representations. This leaves many questions unanswered and indeed unposed, but promotes a balanced view of the artist-programmer engaged closely both with their target medium, and the meta-medium of the source code. We propose that this creative approach should be embraced with the full range of human faculties available.

The ideas of the artist-programmer are wrapped in both analogue and digital packages, and as they are unwrapped, the programs that spring forth from fingers at keyboard have a trace of analogue relations in conceptual space, as well as high level, structural reflections cast from the language of the mind. Source code may be organised into discrete trees, but those trees sway in an analogue breeze from the activity of perception. We hope the artworks, languages and notations introduced through this thesis demonstrate the fertile ground available to the artist-programmer, and inspire greater works to follow.

References

Aaron, S., Blackwell, A. F., Hoadley, R., and Regan, T. (2011). A principled approach to developing new languages for live coding. In *Proceedings of New Interfaces for Musical Expression 2011*, pages 381–386.

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. The MIT Press, second edition.

Agawu, K. (2003). *Representing African Music: Postcolonial Notes, Queries, Positions*. Routledge, first edition.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, first edition.

Arns, I. (2004). Read_me, run_me, execute_me. code as executable text: Software art and its focus on program code as performative text. In Goriunova, O. and Shulgin, A., editors, *Read_me Software Art and Cultures*, pages 177–188.

Barsalou, L. W. (1999). Perceptual symbol systems. *The Behavioral and brain sciences*, 22(4).

Barsalou, L. W., Santos, A., Simmons, W. K., and Wilson, C. D. (2008). *Language and Simulation in Conceptual Processing*, chapter 13, pages 245–283. Oxford University Press, USA, first edition.

Beckwith, L. and Burnett, M. (2004). Gender: An important factor in End-User programming environments? In *Proceedings of IEEE Symposium on Visual Languages and Human Centric Computing 2004*, pages 107–114.

Bel, B. (2001). Rationalizing musical time: syntactic and symbolic-numeric approaches. In Barlow, C., editor, *The Ratio Book*, pages 86–101. Feedback Studio.

Bijmolt, T. H. A. and Wedel, M. (1995). The effects of alternative methods of collecting similarity data for multidimensional scaling. *Research in Marketing*, 12(4).

Blackwell, A. (2006a). Gender in domestic programming: From bricolage to séances d'essayage. In *CHI Workshop on End User Software Engineering 2006*.

Blackwell, A. (2006b). Ten years of cognitive dimensions in visual languages and computing. *Journal of Visual Languages & Computing*, 17(4):285–287.

Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of 17th Psychology of Programming Interest Group 2005*. University of Sussex.

Blackwell, A. and Green, T. (2002). Notational systems – the cognitive dimensions of notations framework. In Carroll, J. M., editor, *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, pages 103–134. Morgan Kaufmann.

Blackwell, A. F. (2006c). Metacognitive theories of visual programming: what do we think we are doing? In *Proceedings of IEEE Symposium on Visual Languages*, pages 240–246.

Blackwell, A. F. (2006d). Metaphors we program by: Space, action and society in java. In *Proceedings of the 18th Psychology of Programming Interest Group 2006*.

Blackwell, A. F. (2006e). The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction*, 13(4):490–530.

Boden, M. A. (2003). *The Creative Mind: Myths and Mechanisms*. Routledge, second edition.

Boer, B. (2010). Modelling vocal anatomy's significant effect on speech. *Journal of Evolutionary Psychology*, 8(4):351–366.

Boulez, P. (1990). *Orientations: Collected Writings*. Harvard University Press.

Bregman, A. S. (1994). *Auditory Scene Analysis: The Perceptual Organization of Sound*. The MIT Press.

Bringhurst, R. (2004). *The Elements of Typographic Style*. Hartley & Marks Publishers, third edition.

Brown, P., Gere, C., Lambert, N., and Mason, C., editors (2009). *White Heat Cold Logic: British Computer Art 1960-1980 (Leonardo Books)*. The MIT Press.

Buzsaki, G. (2006). *Rhythms of the Brain*. Oxford University Press, USA, first edition.

Caclin, A., McAdams, S., Smith, B. K., and Winsberg, S. (2005). Acoustic correlates of timbre space dimensions: A confirmatory study using synthetic tones. *The Journal of the Acoustical Society of America*, 118(1):471–482.

Cage, J. (1969). Art and technology. In *John Cage: Writer*. Cooper Square Press.

Calvo-Merino, B., Glaser, D. E., Grèzes, J., Passingham, R. E., and Haggard, P. (2005). Action observation and acquired motor skills: An fMRI study with expert dancers. *Cerebral Cortex*, 15(8):1243–1249.

Campbell, J. F. (1880). *Canntaireachd : articulate music*. Archibald Sinclair.

Carpenter, E. and Laccetti, J. (2006). Open source embroidery (interview).

Cascone, K. (2000). The aesthetics of failure: "Post-Digital" tendencies in contemporary computer music. *Computer Music Journal*, 24(4):12–18.

Chambers, C. K. (1980). *Non-lexical vocables in Scottish traditional music*. PhD thesis, University of Edinburgh.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Chong, T. T. J., Cunnington, R., Williams, M. A., Kanwisher, N., and Mattingley, J. B. (2008). fMRI adaptation reveals mirror neurons in human inferior parietal cortex. *Current biology : CB*, 18(20):1576–1580.

Church, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ, USA.

Church, L. and Green, T. (2008). Cognitive dimensions - a short tutorial. In *Proceedings of 20th Psychology of Programming Interest Group 2008*.

Clark, A. (2008). *Supersizing the Mind: Embodiment, Action, and Cognitive Extension (Philosophy of Mind Series)*. Oxford University Press, USA.

Clayton, M. (2008). *Time in Indian Music: Rhythm, Metre, and Form in North Indian Rag Performance (Oxford Monographs on Music)*. Oxford University Press, USA.

Collins, N. (2003). Generative music and laptop performance. *Contemporary Music Review*, 22(4):67–79.

Collins, N. (2007). Live coding practice. In *Proceedings of New Interfaces for Musical Expression 2007*.

Collins, N., McLean, A., Rohrhuber, J., and Ward, A. (2003). Live coding in laptop performance. *Organised Sound*, 8(03):321–330.

Cox, G., McLean, A., and Ward, A. (2000). The aesthetics of generative code. In *International Conference on Generative Art*.

Cox, G., McLean, A., and Ward, A. (2004). Coding praxis: Reconsidering the aesthetics of code. In Goriunova, O. and Shulgin, A., editors, *read_me Software Art and Cultures*, pages 161–174. Aarhus University Press.

Cross, I. and Tolbert, E. (2008). Music and meaning. In *The Oxford Handbook of Music Psychology*. Oxford University Press.

Csikszentmihalyi, M. (2008). *Flow: the psychology of optimal experience*. HarperCollins.

Cupchik, G. (2001). Shared processes in spatial rotation and musical permutation. *Brain and Cognition*, 46(3):373–382.

de Rijke, V., Ward, A., Stockhausen, K., Drever, J. L., and Abdullah, H. (2003). Quack project CD cover notes.

Dean, R., Whitelaw, M., Smith, H., and Worrall, D. (2006). The mirage of real-time algorithmic synaesthesia: Some compositional mechanisms and research agendas in computer music and sonification. *Contemporary Music Review*, 25(4):311–326.

Deleuze, G. and Guattari, F. (1987). 1440: The smooth and the striated. In *Thousand Plateaus: Capitalism and Schizophrenia*, chapter 14. University of Minnesota Press, first edition.

Deutsch, D. and Feroe, J. (1981). The internal representation of pitch sequences in tonal music. *Psychological Review*, 88(6):503–22.

di Pellegrino, G., Fadiga, L., Fogassi, L., Gallese, V., and Rizzolatti, G. (1992). Understanding motor events: a neurophysiological study. *Experimental brain research. Experimentelle Hirnforschung. Expérimentation cérébrale*, 91(1):176–180.

Dijkstra, E. W. (1985). On anthropomorphism in science.

Dijkstra, E. W. (1988). On the cruelty of really teaching computing science.

Dinstein, I., Thomas, C., Behrmann, M., and Heeger, D. J. (2008). A mirror up to nature. *Current biology : CB*, 18(1).

Douglas, K. M. and Bilkey, D. K. (2007). Amusia is associated with deficits in spatial processing. *Nature Neuroscience*, 10(7):915–921.

du Sautoy, M. (2008). *Finding Moonshine: A Mathematician's Journey Through Symmetry.* HarperCollins Publishers.

Elliott, C. (2009). Push-pull functional reactive programming. In *Proceedings of 2nd ACM SIGPLAN symposium on Haskell 2009.*

Emmerson, S. (2007a). The human body in electroacoustic music: Sublimated or celebrated? In *Living Electronic Music*, pages 61–87. Ashgate Pub Co.

Emmerson, S. (2007b). Postscript: the unexpected is always upon us – live coding. In *Living Electronic Music*, page 115. Ashgate Pub Co.

Eno, B. (1996). Generative music. *In Motion Magazine.*

Essinger, J. (2004). *Jacquard's Web: How a Hand-Loom Led to the Birth of the Information Age.* Oxford University Press, USA, first edition.

Fellbaum, C., editor (1998). *WordNet: An Electronic Lexical Database (Language, Speech, and Communication).* The MIT Press, illustrated edition.

Filesystem Hierarchy Standard Group (2003). Filesystem hierarchy standard.

Finney, S. A. (2001). Real-time data collection in linux: A case study. *Behavior Research Methods, Instruments, & Computers*, 33(2):167–173.

Flanagan, D. and Matsumoto, Y. (2008). *The Ruby Programming Language.* O'Reilly Media, first edition.

Fodor, J. A. (1980). *The Language of Thought (Language & Thought Series).* Harvard University Press, first edition.

Fontana, F. and Rocchesso, D. (1995). A new formulation of the 2D-waveguide mesh for percussion instruments. In *Proceedings of XI Colloquium on Musical Informatics 1995*, pages 27–30.

Forth, J., Wiggins, G., and McLean, A. (2010). Unifying conceptual spaces: Concept formation in musical creative systems. *Minds and Machines*, 20(4):503–532.

Freed, A. and Schmeder, A. (2009). Features and future of open sound control version 1.1 for NIME. In *Proceedings of New Interfaces for Musical Expression 2009.*

Friberg, A., Bresin, R., and Sundberg, J. (2006). Overview of the KTH rule system for musical performance. *Advances in Cognitive Psychology, Special Issue on Music Performance*, 2(2-3):145–161.

Friedl, J. E. F. (2006). *Mastering Regular Expressions.* O'Reilly Media, third edition.

Fritz, T., Jentschke, S., Gosselin, N., Sammler, D., Peretz, I., Turner, R., Friederici, A. D., and Koelsch, S. (2009). Universal recognition of three basic emotions in music. *Current Biology*, 19(7):573–576.

Galanter, P. (2003). What is generative art? complexity theory as a context for art theory. In *In GA2003 – 6th Generative Art Conference.*

Galantucci, B., Fowler, C. A., and Turvey, M. T. (2006). The motor theory of speech perception reviewed. *Psychonomic Bulletin & Review*, 13(3):361–377.

Gallardo, D., Julià, C. F., and Jordà, S. (2008). TurTan: a tangible programming language for creative exploration. In *Third annual IEEE international workshop on horizontal human-computer systems (TABLETOP).*

Gärdenfors, P. (2000). *Conceptual Spaces: The Geometry of Thought.* The MIT Press.

Greenewalt, M. H. (1946). *Nourathar, the Fine Art of Light Color Playing.* Philadelphia. Pa. Westbrook.

Grey, J. M. (1977). Multidimensional perceptual scaling of musical timbres. *The Journal of the Acoustical Society of America*, 61(5):1270–1277.

Griffiths, D. and Barry, P. (2009). *Head First Programming: A Learner's Guide to Programming Using the Python Language.* O'Reilly Media, first edition.

H. K. M. (1918). The unseen world. *The New Republic*, 14:63+.

Hajda, J. M., Kendall, R. A., Carterette, E. C., and Harschberger, M. L. (1997). Methodological issues in timbre research. In Deliège, I. and Sloboda, J., editors, *Perception and Cognition of Music*, pages 253–307. Psychology Press.

Henderson, J. (2003). Human gaze control during real-world scene perception. *Trends in Cognitive Sciences*, 7(11):498–504.

Hickok, G. (2009). Eight problems for the mirror neuron theory of action understanding in monkeys and humans. *Journal of Cognitive Neuroscience*, 21(7):1229–1243.

Hudak, P. (2000). *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press, first edition.

Janer, J. (2008). *Singing-driven Interfaces for Sound Synthesizers*. PhD thesis, Universitat Pompeu Fabra, Barcelona.

Jeffs, C. (2007). Cylob music system. `http://durftal.com/cms/cylobmusicsystem.html`.

Jones, D. E. (1990). Speech extrapolated. *Perspectives of New Music*, 28(1):112–142.

Jones, S. P., editor (2002). *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/.

Jordà, S., Geiger, G., Alonso, M., and Kaltenbrunner, M. (2007). The reacTable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of Tangible and Embedded Interaction 2007*.

Karplus, K. and Strong, A. (1983). Digital synthesis of plucked string and drum timbres. *Computer Music Journal*, 7(2):43–55.

Kernighan, B. W. and Ritchie, D. M. (1988). *C Programming Language*. Prentice Hall, second edition.

Kippen, J. (1988). *The Tabla of Lucknow - A cultural analysis of a musical tradiation*. Cambridge University Press.

Klee, P. (1953). *Pedagogical sketchbook*. Faber and Faber.

Kohler, W. (1930). *Gestalt Psychology*. Camelot Press.

Kokol, P. and Kokol, T. (1996). Linguistic laws and computer programs. *Journal of the American Society for Information Science*, 47:781–785.

Kowalski, R. (1979). Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436.

Krumhansl, C. L. (1989). Why is musical timbre so hard to understand? In Nielzén, S. and Olsson, O., editors, *Structure and Perception of Electroacoustic Sound and Music, Proceedings of the Marcus Wallenberg symposium 1998*, pages 43–53. Excerpta Medica.

Kupka, I. and Wilsing, N. (1980). *Conversational Languages*. John Wiley and Sons.

Ladefoged, P. (1990). The revised international phonetic alphabet. *Language*, 66(3):550–552.

Laird, J. A. (2001). *The Physical Modelling of Drums using Digital Waveguides*. PhD thesis, University of Bristol.

Lakoff, G. (1997). *Women, Fire, and Dangerous Things*. University Of Chicago Press, 1997 edition.

Lakoff, G. and Johnson, M. (1980). *Metaphors We Live By.* University of Chicago Press, second edition.

Lee, E. A. (2009). Computing needs time. *Communications of the ACM*, 52(5):70–79.

Leman, M. (2007). *Embodied Music Cognition and Mediation Technology.* The MIT Press.

Lerdahl, F. and Jackendoff, R. (1983). *A Generative Theory of Tonal Music.* MIT Press, Cambridge, MA.

Lessig, L. (2006). *Code: And Other Laws of Cyberspace, Version 2.0.* Basic Books.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, 10(8):707–710.

Lévi-Strauss, C. (1968). *The Savage Mind (Nature of Human Society).* University Of Chicago Press.

Levy, S. (2002). *Hackers: Heroes of the Computer Revolution.* Penguin Putnam.

Lewis, G. (1993). Max in a musical context. *Computer Music Journal*, 17(2).

Liberman, A. M. and Mattingly, I. G. (1985). The motor theory of speech perception revised. *Cognition*, 21(1):1–36.

Ling, L. E., Grabe, E., and Nolan, F. (2000). Quantitative characterizations of speech rhythm: syllable-timing in singapore english. *Language and speech*, 43(4):377–401.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '04, pages 119–150, New York, NY, USA. ACM.

London, J. (2004). *Hearing in Time: Psychological Aspects of Musical Meter.* Oxford University Press, USA.

Lyons, W. E. (1986). *The disappearance of introspection.* MIT Press.

Magnusson, T. (2009). Of epistemic tools: musical instruments as cognitive extensions. *Organised Sound*, 14(2):168–176.

Martin, G. N. (2006). *Human Neuropsychology.* Prentice Hall, second edition.

Martino, D. (1966). Notation in General-Articulation in particular. *Perspectives of New Music*, 4(2):47–58.

McAdams, S. (1999). Perspectives on the contribution of timbre to musical structure. *Computer Music Journal*, 23(3):85–102.

McCartney, J. (2002). Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68.

McGurk, H. and MacDonald, J. W. (1976). Hearing lips and seeing voices. *Nature*, 264(246-248).

Mcilwain, P., Mccormack, J., Dorin, A., and Lane, A. (2005). Composing with nodal networks. In Opie, T. and Brown, A., editors, *Proceedings of the Australasian Computer Music Conference 2005*, pages 96–101.

McLean, A. (2004). Hacking perl in nightclubs. http://www.perl.com/pub/a/2004/08/31/livecode.html.

McLean, A. (2007). Improvising with synthesised vocables, with analysis towards computational creativity. Master's thesis, Goldsmiths College, University of London.

McLean, A., Griffiths, D., Collins, N., and Wiggins, G. (2010). Visualisation of live code. In *Proceedings of Electronic Visualisation and the Arts London 2010*.

McLean, A. and Wiggins, G. (2010). Petrol: Reactive pattern language for improvised music. In *Proceedings of the International Computer Music Conference 2010*.

Melara, R. D., Marks, L. E., and Lesko, K. E. (1992). Optional processes in similarity judgments. *Perception & Psychophysics*, 51(2):123–133.

Miller, G. A. (1956). The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81–97.

Milner, R., Tofte, M., and Harper, R. (1990). *The definition of Standard ML*. MIT Press, Cambridge, MA, USA.

Mole, C. (2009). The motor theory of speech perception. In *Sounds and Perception: New Philosophical Essays*. Oxford University Press.

Murphy, G. L. (2002). *The Big Book of Concepts (Bradford Books)*. The MIT Press.

Myers, B. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123.

Nake, F. (1971). There should be no computer art. *PAGE*, 18.

Nash, C. and Blackwell, A. F. (2011). Tracking virtuosity and flow in computer music. In *Proceedings of International Computer Music Conference 2011*.

Naur, P. (1992a). Human knowing, language and discrete structures. In *Computing: A Human Activity*, pages 518–535. ACM Press.

Naur, P. (1992b). Programming languages are not languages – why 'programming language' is a misleading designation. In *Computing: A Human Activity*, pages 503–510. ACM Press.

Okabe, A., Boots, B., Sugihara, K., and Chiu, S. N. (2000). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Series in Probability and Statistics. John Wiley and Sons, Inc., second edition.

Oliver, J., Savičić, G., and Vasiliev, D. (2011). The critical engineering manifesto. http://criticalengineering.org/.

Paivio, A. (1990). *Mental Representations: A Dual Coding Approach (Oxford Psychology Series)*. Oxford University Press, USA.

Patel, A. D. (2007). *Music, Language, and the Brain*. Oxford University Press, USA, first edition.

Patel, A. D. and Iversen, J. R. (2003). Acoustic and perceptual comparison of speech and drum sounds in the north indian tabla tradition: An empirical study of sound symbolism. In *Proceedings of the 15th International Congress of Phonetic Sciences (ICPhS)*.

Pearce, J. M. (2005). Selected observations on amusia. *European neurology*, 54(3):145–148.

Petre, M. and Blackwell, A. F. (1999). Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51:7–30.

Polgár, T. (2005). *Freax: The Brief History of the Computer Demoscene*. CSW-Verlag.

Pressing, J. (1984). Cognitive processes in improvisation. In Crozier, R. and Chapman, A., editors, *Cognitive Processes in the Perception of Art*, pages 345–363. Elsevier Science Publishers.

Pressing, J. (1987). Improvisation: Methods and models. In Sloboda, J. A., editor, *Generative Processes in Music*, pages 129–178. Oxford University Press.

Puckette, M. (1988). The patcher. In *Proceedings of International Computer Music Conference 1988*.

Pullum, G. K. and Gazdar, G. (1982). Natural languages and context-free languages. *Linguistics and Philosophy*, 4(4):471–504.

Pylyshyn, Z. W. (2007). *Things and Places: How the Mind Connects with the World (Jean Nicod Lectures).* The MIT Press, first edition.

Ramachandran, V. S. (2000). Mirror neurons and imitation learning as the driving force behind ”the great leap forward” in human evolution. *Third Culture.*

Ramachandran, V. S. and Hubbard, E. M. (2001a). Psychophysical investigations into the neural basis of synaesthesia. *Proceedings. Biological sciences / The Royal Society*, 268(1470):979–983.

Ramachandran, V. S. and Hubbard, E. M. (2001b). Synaesthesia – a window into perception, thought and language. *Journal of Consciousness Studies*, 8(12):3–34.

Rayner, K. and Pollatsek, A. (1994). Word perception. In *The Psychology of Reading*, chapter 3. Routledge.

Reas, C. and Fry, B. (2007). *Processing: A Programming Handbook for Visual Designers and Artists.* The MIT Press.

Remez, R. E., Pardo, J. S., Piorkowski, R. L., and Rubin, P. E. (2001). On the bistability of sine wave analogues of speech. *Psychological Science*, 12(1):24–29.

Rodet, X., Potard, Y., and Barriere, J. B. (1984). The CHANT project: From the synthesis of the singing voice to synthesis in general. *Computer Music Journal*, 8(3).

Rohrhuber, J., de Campo, A., and Wieser, R. (2005). Algorithms today: Notes on language design for just in time programming. In *Proceedings of International Computer Music Conference 2005.*

Rohrhuber, J., de Campo, A., Wieser, R., van Kampen, J.-K., Ho, E., and Hölzl, H. (2007). Purloined letters and distributed persons. In *Music in the Global Village Conference 2007.*

Rosenblum, L. D. and Saldaña, H. M. (1996). An audiovisual test of kinematic primitives for visual speech perception. *Journal of experimental psychology. Human perception and performance*, 22(2):318–331.

Rowe, R. (2001). *Machine Musicianship.* The MIT Press.

Rushkoff, D. (2010). *Program or Be Programmed: Ten Commands for a Digital Age.* OR Books, first edition edition.

Saramago, J. (2000). *The Stone Raft (Panther).* The Harvill Press.

Schon, D. A. (1984). *The Reflective Practitioner: How Professionals Think In Action.* Basic Books, 1 edition.

Schwitters, K. (1932). Ursonate. *Merz*, 24.

Shepard, R. (1962). The analysis of proximities: Multidimensional scaling with an unknown distance function. i. *Psychometrika*, 27(2):125–140.

Shepard, R. N. and Metzler, J. (1971). Mental rotation of three-dimensional objects. *Science (New York, N.Y.)*, 171(972):701–703.

Shulgin, A. (2003). Listen to the tools, interview with Alex McLean and Adrian Ward. In *read_me 2.3 reader.* NIFCA.

Simon, H. A. and Sumner, R. K. (1992). Pattern in music. In Schwanauer, S. and Levitt, D., editors, *Machine models of music*, pages 83–110. MIT Press, Cambridge, MA, USA.

Small, C. (1998). *Musicking: The Meanings of Performing and Listening (Music Culture).* Wesleyan, first edition.

Smalley, D. (1994). Defining timbre refining timbre. *Contemporary Music Review*, 10(2):35–48.

Sorensen, A. (2005). Impromptu: An interactive programming environment for composition and performance. In *Proceedings of the Australasian Computer Music Conference 2005*, pages 149–153.

Sorensen, A. and Gardner, H. (2010). Programming with time: cyber-physical programming with impromptu. In *Proceedings of ACM OOPLSA*, pages 822–834.

Spiegel, L. (1981). Manipulations of musical patterns. In *Proceedings of the Symposium on Small Computers and the Arts*, pages 19–22.

Spiegel, L. (1987). A short history of intelligent instruments. *Computer Music Journal*, 11(3).

Spolsky, J. (2004). Don't let architecture astronauts scare you. In *Joel on Software*, pages 111–114. Apress.

Sterling, L. and Shapiro, E. (1994). *The Art of Prolog, Second Edition: Advanced Programming Techniques (Logic Programming).* The MIT Press, second edition.

Stewart, D. and Chakravarty, M. M. T. (2005). Dynamic applications from the ground up. In *Proceedings of ACM SIGPLAN workshop on Haskell 2005*, pages 27–38, New York, NY, USA. ACM.

Stewart, L. and Walsh, V. (2007). Music perception: sounds lost in space. *Current biology : CB*, 17(20).

Stockhausen, K. and Maconie, R. (2000). *Stockhausen on Music*. Marion Boyars Publishers Ltd.

Stowell, D. (2008). Characteristics of the beatboxing vocal style, C4DM-TR-08-01. Technical report, Queen Mary, University of London.

Sutton-Spence, R. and Woll, B. (1999). *The Linguistics of British Sign Language: An Introduction*. Cambridge University Press.

Taube, H. K. (2004). *Notes from the Metalevel: Introduction to Algorithmic Music Composition*. Swets & Zeitlinger, Lisse, The Netherlands.

Traube, C. and D'Alessandro, N. (2005). Vocal synthesis and graphical representation of the phonetic gestures underlying guitar timbre description. In *8th International Conference on Digital Audio Effects (DAFx'05)*, pages 104–109, Madrid, Spain.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, LIX:433–460.

Turing, A. M. (1992). Intelligent machinery. report, national physics laboratory. In Ince, D. C., editor, *Collected Works of A. M. Turing: Mechanical Intelligence*, pages 107–127. Elsevier, Amsterdam.

Turkle, S. (2005). *The Second Self: Computers and the Human Spirit*. The MIT Press, twentieth anniversary edition.

Turkle, S. and Papert, S. (1990). Epistemological pluralism: Styles and voices within the computer culture. *Signs*, 16(1):128–157.

Turkle, S. and Papert, S. (1992). Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1):3–33.

Tyte, T. (2008). Standard beatbox notation. online; `http://www.humanbeatbox.com/tips/p2_articleid/2`.

Usselmann, R. (2003). The dilemma of media art: Cybernetic serendipity at the ICA london. *Leonardo*, 36(5):389–396.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36.

Van Duyne, S. A. and Smith, J. O. (1993). The 2-D digital waveguide mesh. In *Applications of Signal Processing to Audio and Acoustics*, pages 177–180.

Vogel, J. (2003). Cerebral lateralization of spatial abilities: A meta-analysis. *Brain and Cognition*, 52(2):197–204.

Wang, G. and Cook, P. R. (2004). On-the-fly programming: using code as an expressive musical instrument. In *Proceedings of New interfaces for musical expression 2004*, pages 138–143. National University of Singapore.

Ward, A., Levin, G., Lia, and Meta (2002). *4x4 Generative Design (with Auto-Illustrator, Java, DBN, Lingo): Life/Oblivion.* Apress, first edition.

Ward, A., Rohrhuber, J., Olofsson, F., McLean, A., Griffiths, D., Collins, N., and Alexander, A. (2004). Live algorithm programming and a temporary organisation for its promotion. In Goriunova, O. and Shulgin, A., editors, *read_me — Software Art and Cultures.*

Ward, J., Thompson-Lake, D., Ely, R., and Kaminski, F. (2008). Synaesthesia, creativity and art: What is the link? *British Journal of Psychology*, 99(1):127–141.

Weller, S. C. and Romney, A. K. (1988). *Systematic Data Collection (Qualitative Research Methods Series 10).* Sage Publications, Inc, first edition.

Wessel, D. L. (1979). Timbre space as a musical control structure. *Computer Music Journal*, 3(2):45–52.

Whitehead, A. N. (2001). *Dialogues of Alfred North Whitehead (A Nonpareil Book).* David R Godine.

Wiggins, G. A. (1998). Music, syntax, and the meaning of ”meaning”. In *Proceedings of the First Symposium on Music and Computers*, pages 18–23.

Wiggins, G. A. (2006a). A preliminary framework for description, analysis and comparison of creative systems. *Journal of Knowledge Based Systems*, 19(7):449–470.

Wiggins, G. A. (2006b). Searching for computational creativity. *New Generation Computing*, 24(3):209–222.

Wittgenstein, L. (2009). *Philosophical Investigations.* Wiley-Blackwell, fourth edition.

Wolfram, S. (2002). *A New Kind of Science.* Wolfram Media, first edition.

Youngblood, G. (1970). *Expanded Cinema.* E P Dutton, first edition.

Zhang, N. and Chen, W. (2006). A dynamic fMRI study of illusory double-flash effect on human visual cortex. *Experimental Brain Research*, 172(1):57–66.

Zittrain, J. (2009). *The Future of the Internet–And How to Stop It.* Yale University Press.

Appendix A

Live coding survey

The following contains freeform answers given in response to the survey discussed in §6.9.

A.1 Definitions of live coding

The following are the twenty nine answers to the question "How would you define live coding, in one sentence?", presented here unedited.

- Live coding is instrument building, composing and performing in one performative act.
- Exposing, for an audience, the thought processes behind making music algorithmically.
- Coding as a performance art. (I'm not very good at it yet)
- a way to entertain, the geeks and the noobs.
- An audio visual performance practice where computer software that generates the audi visuals is written as part of the performance.
- A live performance practice which uses computer programming languages and environments as an interface for the interactive manipulation of media rich software runtime systems.
- Live coding is performing a work by writing and modifying computer programming code which is responsible for creating the resultant work.
- Live coding is a performance practice of constructing and interacting with algorithmic processes to create art.
- An exchange of knowledge and experience between two or more performers/coders.
- Creating logic, structure and meaning as part of a performance instead of beforehand.

- maing real-time changes to sound generation
- coding stuff .. live
- listening to changes
- Build from scratch, make it trandparent.
- live coding is improvisation on the ”instrument computer”
- I’m a bit hard core, and prefer live coding to be on the fly, and from scratch - none of this executing pre-built patches / code. I also have a tendency to think it should be ’code’ and not graphic apps such as MAX/ PD.
- Programming as creative journey or ritual.
- electronic improvising freedom
- Coding music/animation with direct sounding/viewable results.
- I often define it as an improvisional way of doing music with computers, thus being to electronic music what free jazz is to jazz. It’s also a way of considering the computer as an instrument. (sorry two sentences)
- The use of instructions and/or rules for the control of computer(s) (or person(s)) as a method of creative expression.
- several iterations of coding/executing within the span of a song/piece
- a more natural interface to the creative process to those of us that think more like computers than bipedal meatbags
- performace as in performance art
- building your instrument while playing it
- the art of creating music from a procedure yet to be started
- Reformalising an active formal system.
- Creating and sharing algorithms live.
- An improvised performance of music or animation by using interactive programming to describe and control process.

A.2 Comparing live coding

The following are the twenty eight freeform, unedited responses to the question "What is the difference between live coding a piece of music and composing it in the sequencer (live coding an animation and drawing one)? In other words, how does live coding affect the way you produce your work, and how does it affect the end result?"

- It is very much live. Temporality comes strongly in. The sequenced piece has its own formal structure defined in the sequencer. The formal structure of the live coded piece is defined by the complexity of the language. Live coding pieces can therefore be boring to listen to. :) But they can be fun to watch being programmed. Therefore, I find livecoding more interesting as a performative practice than a musical practice (if this distinction makes any sense).

- (This question might be too open to answer in a meaningful way.) Live coding is performing, composing is not.

- To date, my live coding has only been a process of assembling code already written, when to turn this on or off, tweak gain/pan, etc.

- it is more about the experience of the audience than about my own concepts. yes, sometimes you try to integrate them, but if it doesn't appeal it won't help you in the contest. while making fixed media music, one can choose for a certain amount of extremity, although one might not sell a lot of cd's than, but hey, that's up to you.

- Live coding is a meta compositional process that emphasises patterns and how they play out, while sequencing is often capturing of performed gestures or sound.

- Live coding is about abstractions. This is really the only difference between me live coding and me using a sequencer. A more interesting question is maybe what's the difference between using impromptu and using a guitar. Then it becomes about the symbolic vs the gestural as well as the many vs the few (concurrent activities that is).

- Live coding forces a work to be performed, and possibly (ideally?) improvised. I find this a very different prospect to 'composing' music in a sequencer (or even writing notation on paper) for a later performance. However, while Live Coding is a performance practice, it also offers the tantalising prospect of manipulating musical structure at a similar abstract level as 'deferred time' composition. To do this effectively in performance is

I think an entirely different skill to the standard 'one-acoustic-event-per-action' physical instrumental performance, but also quite different to compositional methods which typically allow for rework.

- Live coding encourages creative thinking at a more abstract level, for example, the parametrisation of large scale formal structures. Sequencing can provide a composer with a highly detailed representation of the music, which can be useful when trying to realise a very concrete idea. However, I would argue that sequencing is, on the whole, a more passive medium, in that it generally contributes less to the creative process than a live coding environment. Perhaps a reason for this is that sequencing aims to provide an analogical representation of music, where as there is much more scope for the representation of musical ideas in live coding. Serendipity is also more likely to figure in live coding, where subtle random processes or plain bugs might result in interesting unintended ideas. Plus the live aspect imposes its own set of constraints that are not present in off-line composition, as well as opens up possibilities for real-time collaboration.
- Live coding connects me with the music or sound the same way playing guitar does; it gives me a feeling of control and makes me more conscious of the now. My first concern is to produce a sound, much the same way as a jazz improviser's first concern is a melody. Structure is something to start working on once you have suitable sounds (or scales, melodies, harmonies, rhythms, whatever). Note: I am a beginning livecoder, so my view on livecoding could very well change over the years.
- Sequenced music isn't performed, it's just playback. Following this logic, and assuming I want to improvise in a performance, there's no way to properly perform using a computer if I had to sequence everything. The other part of me thinks that writing code is the same as sequencing, it's just a shorthand. To answer your question, this all hinges on the need to be improvising at a performance. If this wasn't necessary, and it was merely about a playback, then there would be no need for livecoding.
- I've only played around with this at home, but Live coding has far less perfection and the product is more ore immediate. It allows for improvisation and spontaneity and discourages over-thinking.
- As someone who isnt very apt in the manual world of realtime composition with physical, traditional instruments I probably struggle with both approaches. Live coding is just another instrument to me, just one which has a more significant setup time and linguistic interface. When I do use sequencers I frequently use a lot of embedded algo type

processes and code external apps (pd etc) to process alongside. Is the distinction more apparent when its live to a live audience? as that implies a deadline, expectations etc. If you compose music at home on your own gear, including a day long workout with sequencers and software (generally with littel pause in the actual loop) is that livecoding? semantics schemantix.....

- there is only a recording left, but i can't repeat the coding.
- Coding live: intuitive decisions with no pre-built sequences. Freedom, danger and punk.
- generally i find composing in sequencers very boring as a process (so i hardly do it nowadays), whereas live coding (LC) adds more excitement to the process ("will it actually work at all?"). whenever i do sequence-like structures in LC, they are usually generated (using e.g. randomness (with constraints to fit them into the musical context)), so they are not totally predictable. sequences will eventually be re-generated. however, i often don't use sequences at all, but rather try generate all the non-live structures algorithmically.
- Well, the live-coding software provides greater scope for sound manipulation than other audio editing software (eg pro tools etc), but doing it on the fly (live coding) vs writing code in my own time is more of a psychological hurdle - It's harder to be satisfied with the outcome with live coding. When I work on writing a piece, in my own time (and taking a long time) I can perfect each sound to be precisely as I intend it to be, whereas doing it on the fly , i.e. live coding I have to be more generalised as to my intentions.
- Essentially the main difference is the ability to improvise. The live aspect becomes more important than the end result. Collaboration becomes easier, and more natural - as it's easier to adapt the work to different people and new situations.
- I use live coding to develop ideas and test them then I would use them in composed pieces, but the process and imprvising is almost more interesting as the end result.
- Mostly the fact that while livecoding I always found sounds that I could never had found otherwise. Thus, when I find a sound that seems interesting to me, I try to tweek every parameter to see how better it could be. I am used to keep track of every session I am doing, but I often feel a bit disappointed by the static feeling of the last sound I get. But most importantly, even if sometimes I make sessions that is totally crap, I always learn something new. And as I come back to livecoding, all the knowledge I got from all sessions is here readily available to use in my head. Another advantage is the fact that

even if you use rather the same structures every now and then, it always sounds different. On a mental process way of thinking, livecoding gives the opportunity to always foster new ideas while not coding. It's a real creative way of making music.

- Live Coding is riskier, and one has to live with unfitting decissions. You can't just go one step back unless you do it with a nice pirouette. Therefore the end result is not as clean as an "offline-composition", but it can lead you to places you usally never would have ended.
- I'm not sure they need be different at all, as a basic sequencer is fairly easy to implement. Mostly I use larger gestures to place a whole series of notes at once, this is faster than clicking them all in and leads often leads to unexpected results. In homebrew software we can have a tighter link between the pattern generation and the signal generation processes but in livecoding creating this link may become a form of expression itself, which is of course quite exciting. Perhaps most importantly the higher pace of livecoding leads to more impulsive choices which keeps things more interesting to create. Not sure how often that also creates a more interesting end result but at least sometimes it does. The shorter setup time before we can get started with making music (less gear to turn on, no patch cables) makes it more suitable for impulsive playing as well.
- i don't think of it as different to a sequencer, it's just a different user interface. i think in the same way using both. i just often find the programmatic interface to fit my thought processes better.
- You don't have the opportunity to do re-takes
- the whole process becomes more improvisational. like sketching. never finished compositions (on a timeline / with a fixed form). for me it's more about having fun coding than the end result anyway.
- Live coding forces me to experiment and move faster within the compositional/ improvisational process. Personally I consider live coding to be the improvisation of the computer music world.
- I've never really done much with sequencers, so I don't know the difference. I like the fact that live coding is more sculptural and passive.
- Live coding exposes the beauty of programming languages (or programmatic practices) and the invention of interface live. Though you could do live coding with an established tool, I believe part of the interest is seeing the building of tools, even if they are meta

tools. I have all but abandonned live coding as a regular performance practice, but I use the skills and confidence acquired to modify my software live if I get a new idea while on stage.

- You have different kind of canvas, and it favors results within your technique.