Risking Code

Software Art – Dilemmas and Possibilities

Brogan Bunt

PhD Dissertation

School of Art & Design

Faculty of Creative Arts

University of Wollongong

2007

Abstract

This thesis focuses on the emerging field of software art. It is concerned with questions that arise in relation to efforts to think the field of software, and software programming particularly, in aesthetic terms. Centrally, how can software as a technical field of production, as a form of engineering and as a space of abstract, instrumentally-oriented, system elaboration, possibly correspond to art? What are the dilemmas that the notion of software art confronts? How can its space of opportunity be conceived? These questions are pursued not only at a general theoretical level but in terms of aspects of my own software art practice.

The thesis begins by considering the ambivalent character of software, examining how it mediates between dimensions of machine process and human agency and how this potential has been conceived in cultural theoretical terms. It then outlines the specific formal features of software programming and reviews competing perspectives of software practice. This description of the software medium establishes a foundation for a specific consideration of the field of software art. I trace the historical emergence of the genre, examine how it has been theoretically conceived and consider a range of exemplary works. I then specify three key dilemmas that confront software art: the dilemma of position (how can software art conceive its relation to the larger economic and discursive space of the software industry?); the dilemma of visibility (how can software art conceive its efforts to make code visible when software itself determinedly, structurally, hides?); and the dilemma of recursion (how is software to avoid an exclusive and disabling emphasis on self-reflection?). In the remaining portion of the thesis, these dilemmas are considered within the specific context of examining issues and aesthetic strategies within my own work.

My overall argument is that software art represents a permeable discursive space that discovers an aesthetic potential less by resisting the spectre of conventional software than by risking an intimate relation to this alien terrain. Rather than a calm appropriation of software by art, it represents an unsettling of art by means of software.

Thesis Certification

I, Brogan Bunt, declare that this thesis, submitted in partial fulfillment of the requirements for the award of Doctor of Philosophy, in the School of Visual Arts and Design, in the Faculty of Creative Arts, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification at any other academic institution.

Brogan S Bunt
21 January 2007

Acknowledgements

I would like to thank my supervisors Professor Diana Wood Conroy and Professor Amanda Lawson for their advice and encouragement. They managed to combine general patient indulgence with the odd timely poke in the ribs – both strategies proving very helpful.

I would also like to thank my partner, Deborah Jenkin, for reading all the drafts, gently making suggestions and being very kind about my various 'scholarly' idiosyncracies. Probably also sensible to thank my youngest son, Axel, for getting on with Christmas in Perth without me. And my elder sons, George and Sam, deserve thanks for not complaining about still not receiving Christmas presents.

I have also appreciated the support of my father, Professor John Bunt, and stepmother, Eleanor Bunt, who have come along to my exhibitions and provided very useful sounding boards for ideas.

This dissertation is dedicated to my mother, Joan Aileen Vance, who was always keen to see me get my doctorate and who died in late 2005. I regret that I did not finish this sooner.

Table of Contents

List of Figures

Chapter 1: Introduction

Research Question

Software surrounds us. It appears as an industry, a terrain of functional tools and a complex space of technical production. *How can software as a form of instrumental engineering possibly correspond to art?* This thesis addresses this question, focusing particularly on the writing of software – the various ways in which programming can be conceived as a mode of creative practice. My argument is that the thinking of code practice fundamentally unsettles conventional notions of art as autonomous, non-instrumental and reflective. In its engagement with the language and institution of software – especially in the necessity that software art literally function, that it operate – art is compelled to re-evaluate its sense of cultural identity. This encounter between two conventionally distinct modes of making takes shape not only as a set of dilemmas – of position, of association, of orientation and of delineation – but also, in an integral manner, as a field of creative possibility. Software programming discovers a 'speculative' (Fuller, 2003: 29) aesthetic dimension and art discovers a mode of practice that cannot be reduced to the twin removes of rarefied formalism or purely exterior critique.

This question of the relation between software and art is addressed not only at a general theoretical level but also in terms of my own work. A consideration of a number of my software art projects provides a means of illustrating a conception of software art as a permeable discursive space that engages fundamental cultural tensions.

Project Background

This project emerges from just over a decade's sustained creative engagement with the field of computer programming. My background is in the field of media theory and production. Sometime in the mid-1990s I became interested in the then expanding field of multimedia. Fiddling around with 'creative software'

applications such as Macromedia *Director* and *Flash* led me from basic slideshows and interactive websites to the reinvention of simple arcade games and more ambitious projects. As my programming skills improved, I found myself moving away from proprietary multimedia software applications to more general, system-level programming languages such as Java. I was no longer a media producer who dabbled in programming, I had become a programmer who dabbled in media production. While I produced all kinds of projects during this time – the interactive documentary project, *Halfeti – Only Fish Shall Visit* (2001), the experimental game project, *Hotel* (2002), the large oral-historical website, *Midland Railway Workshops* (2003), and the meditation on decomposed video time, *Cropper_Propper_Gridder* (2005) – I was increasingly aware that my practice was awkward to describe. It could be called 'multimedia' or 'new media', but this seemed to ignore the vital dimension of code. My work was very much about re-imagining media in terms of the language and discursive forms of computer programming. Lacking a better term, I tended to describe my more experimental, less professionally-geared projects as 'code-based art'. I was very pleased a few years ago when I came across the term 'software art' because it provided a context for my work that extended beyond simple technical determination and beyond the endless effort to distinguish 'old media' from 'new media'. The notion of software art, as articulated by authors such as Florian Cramer (2002) and Mathew Fuller (2003), opened up a vital means of making sense of my work within a larger art-historical and conceptual context. This thesis emerges as an attempt to elaborate this understanding in a formal and systematic manner.

It is worth noting, however, that my work veers from the ordinary conception of software art, which tends to take characteristic shape in work of formalist abstraction or political meta-commentary (Cramer, 2002: 10). My work retains an interest in media and mediation that links back to my earliest interests in photography, video art and experimental audio. One of my key arguments is that this should not be regarded as altogether antithetical to the concerns of software art – that software art can do more than simply reflect upon its apparent self-contained

material and cultural conditions, that it is constituted by openings rather than by any horizon of self-collected closure.

The creative portion of my dissertation includes the following works:

- *Halfeti – Only Fish Shall Visit* (2001): interactive documentary project focusing on a small Turkish town on the banks of the Euphrates River prior to its flooding by a large hydro-electrical project.
- *Hotel* (2002): experimental game project that represents an ironic reflection on notions of generative space.
- *Cropper_Propper_Gridder* (2005): a suite of tools for decomposing video sequences and playing them back as independent sections. When first exhibited the project employed footage from the Ross Sea region of Antarctica.
- *Anachronism* (2006): an anachronistic 3D graphics engine.
- *Paphos* (2006): a DVD of shots from the margins of an Australian archaeological project in Paphos, Cyprus. This is not a literal code-based project but represents a thinking of video in terms of code.

These works are discussed in Chapters 5, 6 and 7 of this dissertation. I enclose the projects on two DVDs. The first DVD, marked 'A', contains copies of the first four projects listed above. It is a data DVD and must be accessed on a computer. At the top level of this disk, you will find a file named 'index.html'. Open this file to obtain instructions about how to view the various projects. This file also links to a range of background information concerning the production and creative conceptualization of the works. The second DVD, marked 'B', contains the *Paphos* project and is viewable on a standard DVD player (as well as on a computer with DVD playback software).

Theoretical Background

I deliberately restrict my discussion, as far as is possible, to work that engages with

the culture and aesthetics of software. It is worth indicating, however, that my approach is strongly informed by strands of post-structuralist philosophy which question the traditional opposition between the human and the technological. Rather than being cast as an altogether alien phenomenon, which appears either (and both) as a subservient tool or as an exterior threat, technology comes to be associated, more positively, with a dimension of enabling non-identity that provides the basis (and non-basis) for any space of manifestation. Most famously, the French philosopher Jacques Derrida (1976) argues that 'speech' is actually preceded by the logical possibility and material operations of 'writing'. If there is the possibility of speech, according to Derrida, then it is not as the index of a pure domain of self-present thought but of the play of (technological) signs. More recently, another French philosopher, Bernard Stiegler (1998), argues that human culture is characterized precisely by the elaboration of a technological exterior ('As a "process of exteriorisation," technics is the pursuit of life by means other than life' (Stiegler, 1998: 17)). From this critical perspective, technology appears as an intimate human potential which serves also to announce the limits of the human (as traditionally conceived).

For my purposes, this unsettling of the boundaries between the human and the technological provides a means of questioning the relation between art and software. Software programming projects a space of communication with the technological nature of software. The latter entails processes both of rational, instrumental abstraction and of unconscious, non-reflective operation which work together to unsettle the reflectively constituted autonomy of critical avant-garde art. To engage aesthetically with software is not only to summon it to the table of art, it is to risk a passage into the aesthetic alienation and unconsciousness of technical process.

I should note that I adopt a broadly cultural theoretical approach, but draw upon aspects of computer science to describe the technical nature of software and programming. There is a need, in my view, to appreciate the specific discursive

character of software as a basis for properly examining its cultural and aesthetic implications.

A note on terminology: I often speak of 'code' and 'coding' instead of referring, more formally, to 'computer programming'.

Outline of Dissertation

Chapter 2 examines the notion of software, considering it at both a technical and a cultural theoretical level. It examines how the software-hardware relation is conceived within computer science and also how four key new media and software art theorists conceive the cultural and aesthetic character of software. Overall, I stress that software represents a mixed technical and cultural space which opens up a dialogue between machine and human and combines elements of abstraction and materiality.

Chapter 3 addresses computer programming specifically. It provides an overview of the main formal features of high-level language programming and considers a variety of competing – at times opposed – perspectives concerning the nature of programming practice.

Chapter 4 focuses on the contemporary field of software art, discussing its history and theoretical conception and examining the split between formalist and critical-cultural tendencies. The final portion of the chapter specifies three dilemmas that confront software art as a mode of creative practice: the dilemma of position (how can software art conceive its relation to the larger economic and discursive space of the software industry?); the dilemma of visibility (how can software art conceive its efforts to make code visible when software itself determinedly, structurally, hides?); and the dilemma of recursion (how is software to avoid an exclusive and disabling emphasis on self-reflexivity?).

Chapter 5 takes up the first dilemma of position in the specific context of considering a range of strategic means of engaging with the industrial-discursive space of the 3D graphics engine. I focus on the strategy of anachronism, which

involves resisting novelty and deliberately re-working aspects of the coding tradition. My 3D graphics engine, *Anachronism*, provides an illustration of this strategy.

Chapter 6 confronts the second dilemma of visibility, which is linked to the instrumental character of software. Rather than resisting the instrumental, I argue that software art should acknowledge its inevitable relation to the functional dimension of software and engage with it as a field of poetic potential. As a means of highlighting problems associated with the repression of the instrumental within software art, I consider my work, *Cropper_Propper_Gridder*, which is constituted vitally as a suite of tools.

Chapter 7 addresses the third dilemma of recursion. This dilemma relates to the sense of closure in software art – the sense that it can do nothing but reflect upon its own conditions. Opposing this model of disabling self-reflection, I describe a range of specific possibilities of opening that are relevant to my own work. I focus specifically on strategies evident in *Halfeti – Only Fish Shall Visit* and *Hotel*.

Chapter 8 is the conclusion. I summarize my argument and suggest the need for a new conception of software art practice. Rather than adhering to the twin fantasies of pure software formalism or pure software critique, there is a need to explore a messier, more complicit and more open space of creation. The various dilemmas of software art, at their limits, suggest areas of aesthetic possibility.

Chapter 2: Software

Introduction

Software is an ambiguous space, partly entwined in the alien complexity of binary processes and partly shaping multiple layers of human access. It abides within the machine but is also abstracted from it. As a vital context for choreographing the logic of technical systems and dimensions of human interaction, software programming mediates between the mechanical and the human, the abstract and the material, and the instrumental and the aesthetic. In its refusal to fall on one side or the other of these conventional oppositions, the work of software creation appears as a space of dialogic exchange and unsettling.

Software shares the suffix 'ware' with its semantic other, hardware. Conventionally, the term 'ware' denotes 'articles of merchandise or manufacture, or goods' (Macquarie, 1992). It suggests a context of physical manufacturing and sale. The metaphor of traditional material-economic relations provides then a means of conceptualizing the less directly accessible processes and products of the information economy. For my purposes, the presence of 'ware' within 'software' serves as a sign of an element of semantic and cultural-contextual tension. Despite the importance of the commercial art market, works of art tend not to be conceived as 'wares' but as critical, expressive, non-instrumental things. Alongside any thinking of the relation between technical computer science and the sphere of art practice, the juxtaposition of 'software' and 'art' also brings into play broader issues of art's notional differentiation from the 'non-art' spheres of industry and commerce (and instrumental rationality generally).

This chapter considers how software is conceived within computer science, as a basis for then examining various cultural theoretical perspectives on software. My particular interest is in how the contemporary critical and creative concern with software programming emerges as a reaction to the notion of new media.

Software and Computation

In the early 19th century, the British industrial inventor, Charles Babbage, designed three

(partly realized) calculating engines. The first two engines, Difference Engines no.1 and no.2, were machines for calculating fixed tables of values. The last machine, the Analytical Engine, was a more ambitious project which was never actually completed. It was envisaged as a general purpose machine that could perform any specific mathematical algorithmic process. The structure of this flexible, programmable machine was articulated in terms of analogies from the field of industrial textile manufacturing. It was composed of a 'store' that held values (raw yarn and finished textiles), a 'mill' that processed (wove) the yarn into textiles, and 'trains' of logical procedures that controlled the functioning of the 'mill' (Babbage, 2005: 282-293). In terms of contemporary computation, the 'store' represents memory, the 'mill' represents the central-processing unit, and the 'trains' (of thought) represent software. While there was no direct analogy from textile manufacturing to describe software, this vital work of logical coordination was to be implemented through punch cards borrowed from the technology of the Jacquard Loom (an early technology for mechanizing the production of patterned textiles). In this sense, the binary, array-based patterns of the Jacquard Loom punch cards provide a link between long cultural traditions of textiles practice and the emerging technology of computation. For me, this is suggestive of the opening of computation to a legacy of logical and aesthetic pattern-making that long precedes the specific forms of digital media.

The design of the Analytical Engine indicates the centrality of software to the conception of modern computing. A computer is conceived as a flexible machine rather than as a traditional, single-purpose mechanical device. The dimension of programmable software is what enables a computer to be reconfigured as a typewriter, a darkroom, a video-editing system, a public forum, and so on. If industrialization produced a plethora of complex single purpose machines, post-industrialization produces a core generic machine that can adopt multiple forms - that can simulate other machines via the dimension of programmatic abstraction.

In 1936, exactly one century after Babbage began work on his Analytical Engine, the British mathematician Alan Turing (1995) described the structure and discrete functioning of a 'universal machine' (Agar, 2001; Copeland, 2004). This virtual machine

could be programmed, like the Analytical Engine, to perform any specific mathematical operation. There was, however, a crucial point of difference. Whereas Babbage had struggled to implement decimal numerical representation, Turing represented dimensions of value and procedural process in a common binary numerical format. Both instructional code and calculated results were conceived as sequences of 0s and 1s, which provided the crucial key to constructing an actual computer. This synthetic representation worked simultaneously to abstract dimensions of material value and to materialize dimensions of symbolic logic. The richness of complex things (texts, images, sounds, etc.) finds itself amenable to binary articulation while processes of logical abstraction (mathematical algorithms) become embedded within a common material-digital substance. Furthermore, the latter gain the capacity to execute, to directly affect the realm of digitally articulated things.

Computer science withdraws from this dimension of (metaphysical) uncertainty to the extent that it conceives software as opposed to hardware. In conventional terms, hardware refers to the physical-mechanical and electronic aspects of computation (physical memory, input-output devices, etc.) whereas software designates the set of formal instructions that direct machine processes and that are abstracted from the material layer of digital circuitry (see, for example, Farouzan, 2003). Software, as a logical algorithmic system, is conceived as floating above the hardware layer. Relative to the material solidity of bits and pieces of metal and silicon, software appears virtual and immaterial. While justifiable as a means of relative differentiation, this common sense distinction obscures the vital sense in which computation works to unsettle the opposition between the abstract and the material.

When Turing first described the modern computer in his 1936 paper 'On Computable Numbers, with an Application to the *Entscheldungsproblem*', he described a hypothetical machine that was designed, in fact, to pinpoint the limits of mechanical computation (via the strategy of a recursive *reductio ad absurdum*) (Feynman, 1996: 81). In this sense, the computer was originally a speculative, immaterial, 'soft' machine that 'functioned' to provide a critique of the possibility of absolutely determinable (mechanical) outcomes. At a more general level, the modern computer and its underlying digital circuitry is the

material embodiment of the abstract system of Aristotelian logic. In the 19th century the mathematician, George Boole, represented this binary logical scheme in algebraic terms and in the 20th century the telecommunications engineer, Claude Shannon, represented it in physical electronic terms. The positive and negative charges of digital circuits are 'hard' to the extent that they are physical, material and machine-based, but their material operation is thoroughly determined by a corresponding space of 'soft', speculative abstraction.

Similarly, although software can be regarded as immaterial in its abstract, symbolic character and in its emphasis on the virtual and potential nature of any given operational system, it remains materially bound. The underlying instructions must be written. Text files are compiled into binary files which are then executed or interpreted to create patterns of discrete voltages in RAM. The code layer is scarcely ever visible to the end-user (it is hidden beneath the interface) and the speed of software operations (at the machine level) typically exceeds the capacities of human perception, but material processes are nonetheless still at play.

Overall, then, the neat distinction between hardware and software misses the point. Computation, as both software and hardware, highlights a dimension of ambiguity in the relation between matter and non-matter. It structures permeable relations between spaces of virtual, symbolic abstraction and material, instrumental functioning.

Alongside its notional immateriality, software also represents a channel of 'soft', human access to the invisible and arcane functioning of the digital machine. Access is enabled at a number of different levels. At the lowest levels, machine code and assembly language provide very intimate (relatively 'hard') access to machine functioning, doing very little to protect the programmer from the demands of binary and hexadecimal modes of representation, the intricacies of central processor functioning and the complexities of system architecture (buses, memory registers, devices and the like). Higher-level language computer programming, while obscure to non-programmers, provides a more accessible and natural language-based means of choreographing digital processes. Underlying mechanisms are abstracted, opening up a space of communication between

human and machine modes of representation and processing. C, C++ and Java are common examples of high-level programming languages. At the top (most abstract) level, graphic user interface (GUI) application software provides the 'softest' point of access to the computer. The conventional and never quite satisfactorily attainable aim of GUI developers is to make software operation seem as natural, human and intuitive as possible – to make the machine, and the relation to the machine, effectively disappear.

In slipping between the technical 'back-end' of coded machine instructions and the experiential 'front-end' of the graphic user interface, software emerges within computer science as a multi-layered and elusive concept, resisting all attempts to pin it down to any one essential material or virtual state and to any one side of the human/machine divide. My specific interest is in the intermediary space of high-level language programming. In the following chapter I offer a non-technical overview of the field, considering both its formal features and its character as a specific mode of creative practice. In the remainder of this chapter I consider a variety of contemporary conceptions of the creative space of software.

Conceiving Software

The German media theorists, Florian Cramer and Ulrike Gabriel, argue that the sphere of software production 'has a long history of being overlooked as artistic material and as a factor in the concept and aesthetics of a work' (Cramer and Gabriel, 2001: 1). The contemporary concern with software art emerges as a reaction to the dominant conception of digital art as a new form of media. The jury statement for the 2001 Berlin *transmediale*.01 festival software art award pointedly distinguishes their concern with software from a more conventional concern with media:

> This award is not about what is commonly understood as multimedia – where the focus is on data that can openly been [*sic*] seen, heard and felt. This award is about algorithms; it is about the code which generates, processes and combines what you see, hear and feel. (*Transmediale*.01 Media Arts festival jury, 2001)

Software can slip into the background of multimedia/new media practice not only due to its invisibility but due to its awkward instrumental character. Programming is all too often relegated to a sphere of technical implementation (indeed many new media artists outsource their coding to professionals).

Despite this sense of resistance to the notion of new media, it needs to be acknowledged that the possibility of software art has emerged from within the thinking of new media. The 'newness' of the latter has inevitably demanded a close consideration of the computational conditions of contemporary media. The creative field of software production has gained prominence as artists have developed increased computational literacy and new media theory has engaged more thoroughly with the implications of linking media and computation.

The work of Lev Manovich provides the clearest example of how a thinking of software can emerge from a thinking of new media.

From Media to Software

In *The Language of New Media* (2001), Manovich provides an influential formal-materialist account of new media. He explains the development of new media in terms of the convergence of 'two separate historical trajectories: computing and media technologies' (Manovich, 2001: 20). Both can trace their origins to the early part of the nineteenth century. Babbage's Analytical Engine provides the point of entry to the computing tradition, while Daguerre's daguerreotype provides the initial scene for modern media (Manovich, 2001: 20). The synthesis of these two traditions renders all media in the common form of computable data:

> The translation of all existing media into numerical data accessible through computers. The result is new media – graphics, moving images, sounds, shapes, spaces, and texts that have become computable; that is, they comprise simply another set of computer data. (Manovich, 2001: 20)

Drawing explicitly upon the discipline of computer science, Manovich (2001) defines new media in terms of a set of five logically-ordered principles. The first and most

fundamental principle is **numerical representation**. Digital processes represent all media in binary numerical form. This opens up the potential for algorithmic manipulation. The second principle is **modularity**. From the tiniest level (bits and bytes) to higher level structures, media elements have an independent status; they are not tied up in a continuous and fixed whole, but can be disassembled and are subject to many varied processes of recombination. The third principle is **automation**. The algorithmic character of crucial processes of new media 'creation, manipulation and access' (Manovich, 2001: 32) unsettles and partially removes the necessity for human agency and duplicates (simulates) many of the latter's characteristic features (perception, intelligence, memory). The fourth principle is **variability**. Due to their underlying numerical status and structural modularity, new media objects are not restricted to a single fixed, formal identity, but can take multiple forms and meet all kinds of specific contextual needs. Variability is vital to Manovich's definition of new media:

> It becomes possible to separate the levels of 'content' (data) and interface. A number of different interfaces can be created from the same data. A new media object can be defined as one or more interfaces to a multimedia database. (Manovich, 2001: 37)

The fifth and final principle is **transcoding**. Computer based structures and processes affect the 'traditional cultural logic of media' (Manovich, 2001: 46). Manovich argues that 'what can be called the computer's ontology, epistemology, and pragmatics – influence the cultural layer of new media, its organization, its emerging genres, its contents' (Manovich, 2001: 46). For this reason, Manovich suggests that:

> To understand the logic of new media, we need to turn to computer science. It is there that we may expect to find the new terms, categories, and operations that characterize media that become programmable. From media studies, we move to something that can be called 'software studies' – from 'media theory' to 'software theory.' (Manovich, 2001: 48)

Gradually, then, through the process of describing the fundamental principles of new media, Manovich moves away from a traditional conception of media towards a conception of software. This passage, this transition, is highly useful, but it is also unstable. The relation between media and software (as a form of computation) slips uncertainly between historical confluence, base-superstructure determination, analogy and identity. Manovich typically associates new media with a potential that takes shape on the basis of computation but that is not identical with computation itself. He argues, for instance, that '[a] new media object can be defined as one or more interfaces to a multimedia database' (Manovich, 2001: 37). Here new media appears as the interface to computational forms and processes. It is the visible and logical consequence of an underlying work of variability rather than itself coextensive with this space of abstraction. At another moment, however, he argues that 'a new media object typically gives rise to many different versions' (Manovich, 2001: 36). In this instance the object itself partakes of variability and hence appears more closely linked to the possibility of software. New media is cast ambivalently as both the experiential product of an anterior work of generative abstraction and as itself intimately engaged in variability.

Although the overall structure of Manovich's conception of new media suggests a notion of media reconceived and reshaped in terms of the model of computation, a sense of lingering distance remains. For example, in relation to the principle of modularity, Manovich argues that structured programming serves as an analogue for the modularity of new media. He suggests, however, that this analogy should not be taken literally:

> If a particular module of a computer program is deleted, the program will not run. In contrast, as with traditional media, deleting parts of a new media object does not render it meaningless. (Manovich, 2001: 31)

Within this context new media appears distinct from software. It is analogous to software but differs to the extent to which modularity is enforced.

At one level, these strands of ambivalence can be regarded as a means of teasing out a theory of software from the strangeness of digital media. However, at another level they

can be regarded as a means of subsuming software under the traditional sign of visible, audible media. From my perspective, the instability of his conception of new media highlights the instability of software itself, which encompasses dimensions both of abstraction and of experiential engagement. Manovich makes a vital contribution towards elaborating a theory of software by describing the limits of media, the moments in which it passes away from itself.

Rejecting Software

In a deliberate, playful provocation**,** German media theorist, Friedrich Kittler, recognizes software but subjects it to a strategic disavowal. Adopting a strongly technological determinist standpoint, Kittler argues that software constitutes an illusory means of human control over technological systems that have already usurped our human powers – that are no longer secondary tools but primary agents. In a 1999 message to the *nettime* mailing list, Kittler argues:

> The billion-dollar business called software is nothing more than that which the wetware [human beings] makes out of hardware: a logical abstraction which, in theory – but only in theory – fundamentally disregards the time and space frameworks of machines in order to rule them. (Kittler, 1999)

He regards software as a mystification – seeming to facilitate human creative-instrumental agency over the realm of machines but actually only a secondary result of underlying relations that are determined at the hardware level. Kittler deliberately reverses the humanist paradigm and embraces the 'pure' exteriority of the machine. It is the lowest-level instrumental means that shape systems of functioning and communication, as well as the imaginative possibility of abstraction, goals and, ultimately, human subjectivity itself:

> When texts, images, and sounds are no longer considered the impulses of brilliant individuals but are seen as the output of historically specified writing, reading, and computing technologies, much will already have been gained. (Kittler, 1999)

Kittler's argument reveals an affinity with post-structuralist critiques of human subjectivity in terms of the motif of language. The French philosopher Jacques Derrida's (1976) insistence on the priority of writing (material, technical, secondary and exterior) over speech (abstract, human, primary and interior) provides a clear example. However, if writing gains priority over speech within Derrida's philosophical scheme, it is not in order to produce a simple reversal; rather, the priority of writing serves as a means of unsettling the thinking of priority itself and of the categorical differentiation between technical exterior and human interior. It represents an effort to think 'beyond' or 'across' conventional categories. Kittler's reversal asserts the priority of the machine, but leaves the machine/human opposition intact. His devaluation of software is significant. Rather than regard software as an indeterminate space of dialogue, he aligns it altogether with the human. Only the machine itself – in its notional purity – is sufficient to represent the other of the human.

Despite this tendency to allow the machine its utter inhumanity, Kittler's reversal has considerable value as a means of countering the common tendency to regard the field of computation as simply a 'tool' that human agents master to realize their creative ideas. He suggests the need to consider how the dimension of the creative concept is shaped by the material and immaterial conditions of computation. The problem, however, is that he reduces this relation to one of determination when something more subtle and complex is occurring, when it is a matter of the blurring and unsettling of boundaries. Software is neither human nor inhuman but a curious terrain in which technical syntax and protocols become mingled with human cultural-imaginative concerns.

Neglecting Software

In *New Philosophy for New Media* (2004), Mark Hansen contrasts traditional media to digital media in terms of their differing modes of embodiment. He argues that traditional (particularly cinematic) media separates the viewer from the objective character of the work (rendering the viewer immobile), whereas digital media makes the body the new medium for the work – the formal and material basis for the work's concrete appearance. Hansen draws upon Rosalind Krauss's notion of the 'post-medium condition' (Krauss,

1999: 32) to describe the aesthetic implications of digitization. Just as contemporary 'post-media' art is aggregative and exceeds its technical support, so the digital image as a collection of 'numerical fragments' (Hansen, 2004: 35) represents an aggregate that lacks any necessary technical frame. He argues that '[r]egardless of its current surface appearance, digital data is at heart polymorphous: lacking any inherent form or enframing' (Hansen, 2004: 35). No longer, in his view, constituted by an external material support (a traditional medium), the digital image gains provisional coherence through an embodied work of interactive perception. In relation to Manovich's definition of new media, Hansen emphasizes the features of numerical representation, modularity and variability in order to position the new media art work as a liquid phenomenon that only properly coheres as it is humanly experienced.

The problem for me here is the lack of recognition of intervening technical layers. Hansen fails to acknowledge that we never actually encounter strings of bits and bytes directly. Digital data is subject to many layers of abstraction and systematic organization prior to becoming 'humanly' accessible. The body (or notions of interactive perception) are certainly implicated in the work of engaging with the digital image (particularly in relation to game environments, virtual and augmented reality systems and the like), but this sense of embodiment is constituted via vital layers of software that mediate between data and experience, that represent and frame data to facilitate relations of embodied perception. While acknowledging the abstract and variable nature of the underlying data structure, Hansen fails to adequately consider how this is articulated, choreographed and conceived to shape any specific possibility for embodied interaction. In this manner, the crucial abstract-conceptual and material-discursive space of software is ignored.

In their famous 1977 anticipation of contemporary forms of new media, 'Personal Dynamic Media', Allen Kay and Adele Goldberg (2003) describe computing as a 'metamedium':

> Although digital computers were originally designed to do arithmetic computation, the ability to simulate the details of any descriptive model means that the computer, viewed as a medium, can be *all other media* if the embedding

> and viewing methods are sufficiently well provided. (Kay and Goldberg, 2003: 393-4)

The crucial insight here is not that the computer is polymorphous (a multimedia chameleon) but that it transcends the dimension of sensible, formal, medium-based identity. A computer can simulate any media because it is defined by an order of abstraction, of flexible logical-systematic articulation. It is defined, in other words, by the possibility of software. The problem with Hansen's conception of embodied interaction is that it ignores the underlying layers of organization that structure and facilitate the field of sensible appearance.

Hansen's preference is for large-scale installation new media work in which the computer and processes of computation are largely hidden away out of view. The emphasis is upon images, spaces and contexts of free kinaesthetic interaction rather than upon the mixed material and abstract space of coded instructions or the sensibly restricted space of conventional human-computer interaction (monitor, keyboard and mouse). Criticizing the conservative character of the conventional human computer interface (HCI), Hansen argues:

> The fact that the HCI extends the sway of immobility must be seen as an occasion for criticism of the cinematic heritage of new media, and beyond that, for exploration of unheeded or unprecedented alternatives. (Hansen, 2004: 35)

Conventional screen-based software appears as a dull relic, inevitably compromised by its adherence to the (non) interactive conventions of cinema. This represents a privileging of physical, sensible modes of engagement over abstract-symbolic and cognitive forms of interaction. For example, in considering Paul Pfeiffer's digital video work, *The Long Count* (2001) – a representation of a boxing match in which the boxer's bodies are removed, leaving only cheering crowd, stretching ring ropes, etc. – Hansen argues that 'it is the viewer's body in itself

(and no longer as an echo of the work's "content") that furnishes the site for the experience of the "work's" self-differing medial condition' (Hansen, 2004: 34).

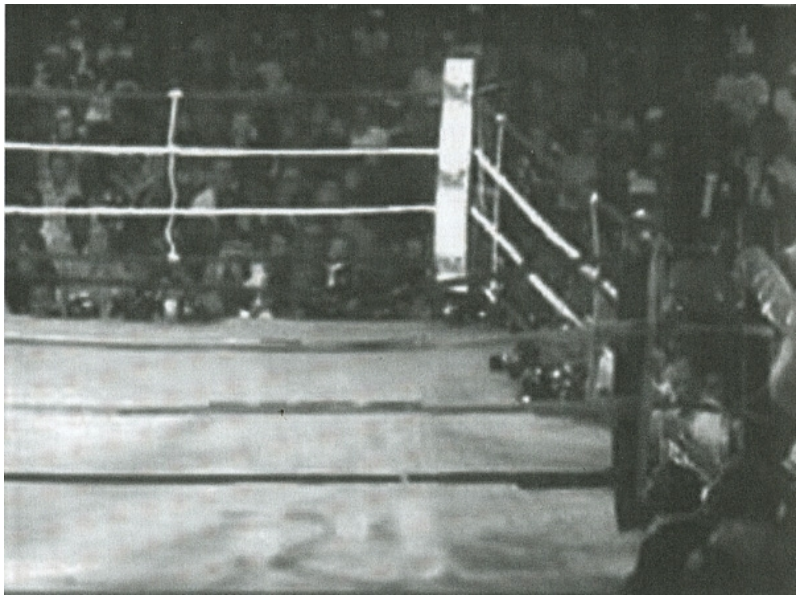

Fig.1: Paul Pfeiffer, *The Long Count* (2001)

Whereas another reading may have stressed the conceptual point of deliberately displaying an absence, of erasing the central point of interest while leaving in place its visible support and effects, Hansen focuses entirely on the embodied apperception of the work. This devaluation of cognitive, semiotic modes of interaction explains his lack of interest in genres that are of central concern to my study, political net art and conceptually inclined software art. While his notion of new media embodiment may provide a refreshing antidote to the discourse of post-humanism, it obscures the underlying conditions of new media - its status not only as experiential field but as the trans-sensible space of software.

Acknowledging Software

Cramer (2005) avoids the kind of polarized view of digital processes that is characteristic of Kittler and Hansen, and very explicitly makes a shift towards 'software theory' (Manovich, 2001: 48). One of the central theoretical voices in the development of the contemporary notion of software art, Cramer conceives software as a mixed technical and cultural phenomenon in which contradictions are

encompassed and conventional opposites intersect and coincide. Against narrowly technical perspectives, he argues that software is a mode of cultural practice that includes algorithms, coded machine instructions, human interaction and a more general space of 'speculative imagination' (Cramer, 2005: 124). Rather than restrict software (and the computational imaginary) to digital computation per se, Cramer sketches a much broader history that takes in traditions of religion, magic, mathematics, combinatory aesthetics and philosophy. Some of these traditions are concerned with divine algorithms, others with natural ones. Some conceive computation as an ecstatic practice, others as a rational, pragmatic one. Some project the possibility of systematic unity, others disassemble and deconstruct. Linking all of these tendencies together, in his view, is an underlying concern with negotiating a passage from the abstract to the material:

> Computation and its imaginary are rich with contradictions, and loaded with metaphysical and ontological speculation. Underneath those contradictions and speculations lies an obsession with code that executes, the phantasm that words become flesh. (Cramer, 2005: 125)

Cramer is explicit about the Biblical reference. He quotes from the Gospel of John in the New Testament: 'In the beginning was the Word […] And the Word was made flesh' (Cramer, 2005: 14). Whereas Kittler strictly limits this magical capacity to digital machine processes, arguing that '[t]here exists no word in any ordinary language which does what it says' (Kittler, 1999), Cramer conceives the link between the domains of abstraction and materiality as just as significant in its speculative and imaginary aspect as in its material technical implementation. In this respect, Cramer brackets the necessity that software be executed. Apart from permitting him a much wider historical-cultural compass, this enables a notion of imaginary, desiring execution (software as 'phantasm' (Cramer, 2005: 125)).

While this strategy has considerable merit, opening the notion of software up to much wider currents of cultural practice and philosophical reflection, it also runs the risk of losing sight of the specificity of contemporary software practice. For my purposes,

software art involves a vital dialogue with the event-space of machine execution. The intimate, dialectical relation between processes of abstraction and the field of execution, of defining algorithmic systems and then setting them to work or at play, is constitutive of my experience of programming software. It is worth noting that the relation between 'words' and 'flesh' has another dimension within software. It is not only that abstract becomes material, but also that the material becomes abstract. Programming code, as I have suggested, is material text that describes and shapes the functioning of abstract systems. 'Words' in this sense can also be regarded as a kind of 'flesh' (or as the coincidence of the technical-material and the technical-schematic). There is a passage back and forth between abstract and material within software practice – the relation is not at all unilateral.

My other qualification regarding Cramer's conception of software is that it pays much more attention to magical and aesthetic conceptions of software than to rational instrumental ones. Pythagorus, Kabbalah, Lull, and the Oulipo figure as key historical antecedents, whereas Aristotle, Boole, Babbage and Turing appear as bit players or are neglected altogether. This imbalance is prompted, no doubt, by Cramer's central concern with elaborating a notion of software art, but it has the consequence, once again, of belittling the space of execution. Everything that shapes software as instrumental, as a work that is crucially concerned with issues of generic and actual functioning, becomes secondary. In the process, the vital dialectic within software art between art and the 'non-art' realms of engineering and technical implementation slips into the thematic background. As I have suggested, one of my major goals in this project is to consider the contours of this awkward relation.

Conclusion

In its capacity to take in both the technical space of algorithms and the human space of interface and interaction, the notion of software is difficult to pin down to any particular mode of material or conceptual being. Cramer offers a cultural conception of software that encompasses algorithmic technical functioning, dimensions of human meaning and use, and a wider context of philosophical, speculative imagination. More specifically, he links software to a self-conscious work of algorithmic invention and interaction. He is

far less concerned with software as a 'transparent' vehicle for human instrumental or creative intention than as a terrain of reflective engagement with aspects of algorithmic system and process. Whereas Kittler associates software with a dimension of illusory (non-technical) human agency and control, Cramer associates it with a critical engagement with typically hidden dimensions of technical process. It is within this context that he and Ulrike Gabriel employ the notion of 'software art' as a means of indicating a realm of creative practice that is all too often obscured by the conventional emphasis on the front-end of 'new media' (Cramer and Gabriel, 2001).

For my purposes, finally, the value of the term 'software' hinges less on its rigorous conceptual specificity than on its ambiguity and connotative richness. Although software can appear opposed to hardware in the same way that the mind is opposed to the body and the abstract is opposed to the material, from another perspective software holds this distinction within itself. It is not simply 'soft', it is also a 'ware' – a hammered, battered, commercial-manufactured thing. Software suggests a mixed conceptual, cultural, economic-industrial and technological space.

Chapter 3: Code

Introduction

In the previous chapter I examined the general notion of software, stressing its vital ambiguity – its capacity to engage both human and machine dimensions of computational process. In this chapter I consider the specific field of high-level language programming. This is the field in which I produce my own software art work. It is constituted precisely as a space of intimate communication between human and machine processes and its key formal discursive features of abstraction, disguise and instrumental operation will prove vital to my questioning of the conventional conception of software art in subsequent chapters. I begin by reviewing these features, with a particular emphasis on examining the principles that inform the structure of contemporary object-oriented programming, and then go on to consider programming as a specific mode of making. If code represents an abstraction of industrial forms of production, then it also passes beyond the model of the assembly line, opening up a mode of practice that encompasses contradictions – appearing at once both precise and logical, and experimental and 'speculative' (Fuller, 2003: 29).

Abstraction

Abstraction is fundamental to all forms of programming. Programming involves abstracting states, processes and systems so that they can be represented in logical, symbolic terms. It is a form of model-making in which every component is ultimately decomposable into discrete binary states and logical operations. Abstraction not only affects how problems are represented, it also shapes the structure of programming itself. High-level language programming represents an abstraction of low-level computing processes. Instead of wrestling with bits and bytes, memory registers and the like, the high-level language programmer tends to deal with pre-defined data types that represent, for instance, numbers, whole words or images, and with sophisticated algorithmic functions that serve as abstracted 'wrappers' to more fundamental logical procedures. Relations of programmatic abstraction are characteristic of every dimension of computer processes.

Data and Algorithms

Programming is based on step by step procedures, algorithms. Algorithms can be regarded as sets of instructions that manipulate data. Data represents the dimension of content, while algorithms represent the dimension of process. There is nothing, however, at the lower level that materially or symbolically separates them. Following Turing's model of the universal machine, they are both represented in memory as strings of 1s and 0s. High-level programmers, as I have suggested in the previous chapter, rarely deal with binary data directly. In fact they rarely even deal with data directly. On the whole they represent data as variables. The following variable 'a' is set to contain the integer value, 49.

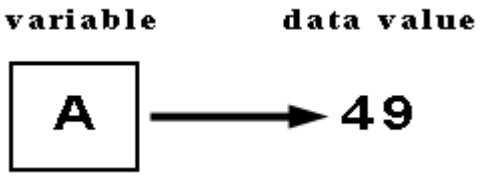

Fig. 2: The notion of a variable

Variables may contain data or they may simply refer to memory addresses where data is stored. Algorithms typically manipulate data through long chains of variable-based mediation. Great care is taken to ensure that algorithmic processes themselves are unaffected by any reference to actual data. Algorithms are conceived as generic machines. They abstract some specific dimension of functionality. They process data but do not allow the particularity of any specific data to structure their operations. The following diagram of a percentage calculating algorithm (function) indicates that specific values pass through the algorithm and specific output values emerge, but that the internal procedures of the algorithm are generic.

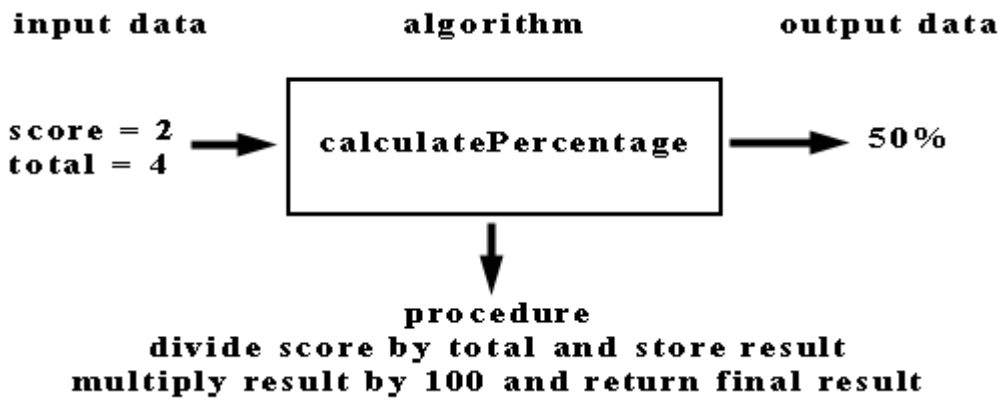

Fig. 3: The notion of a generic function

It is not always possible to produce fully generic code, but typically the more generic an algorithm the better its functioning has been abstracted and properly understood.

Variables are not restricted to holding single values. They can often refer to more complex sets of data – arrays of values, for instance, that represent a collection of potential states. The spatial structure of a chessboard can be represented, for example, in terms of an array of 64 values, one for each square on the board. The array is an abstract conception that bears no necessary relation to perceptible grid-like space, but the values it contains can be interpreted mathematically in terms of logical spatial relations. Numerical positions in the array can be calculated as references to specific row and column positions. I remember how powerfully this recognition of the potential to represent dimensions of perception in an imperceptible logical format affected me when I first learnt about it. The gap between underlying data structure and interface suggested all kinds of creative possibilities. For me, it was not simply a matter of recognizing a work of logical abstraction, it was about the capacity to regard the visible interface as an apparition – a guise – floating above the protean, re-combinatory potential of arrays of data. Overall then a programmer defines not only generic algorithmic procedures but also data structures that represent specific logically defined universes of manipulable elements.

Program Structure

Line by line, programming inevitably involves processes of calculation, but there is more

to programming than just the discrete manipulation of data. Larger structures are employed to choreograph program flow. The two fundamental forms are iteration and conditional branching. For example, a 'for-loop' is an iterative form that repeats some process a specific number of times, while an 'if-statement' is a conditional form that selects an appropriate process depending upon specific conditions. These are formulaic compositional features that shape the overall running of programs.

Beyond these syntactical features there are larger dimensions of program structure. A typical program, say a simple game, functions in the following manner. The program begins with an initialization phase in which technical display and interaction contexts are established and fundamental data structures are constructed and assigned relevant values (involving, perhaps, the loading of relevant game resources). The program then proceeds to the phase of core game operation. This is typically represented as a 'main loop', which involves rapidly repeating the following steps:

- Checking for user input
- Updating relevant data structures
- Updating aspects of game display

The main-loop is a macro-level construct that coordinates the overall running of the game. Each of its steps is likely to involve numerous sub-steps. Programs are structured as hierarchical systems of algorithmic process. There are typically loops within loops within loops.

At the end of the game the program passes into a shut down phase, exiting the main loop, disposing of data structures and returning the display context to the background operating system.

At one level programs have a linear aspect. They begin, do something and end. Yet at another level, in their looping, modular and interactive aspect they can be regarded as non-linear. From this perspective, beginning, running and exiting represent less a linear trajectory than a set of discrete states. A program can be conceived as a system of

interacting components rather than as a fixed sequential thing. This is the perspective that contemporary object-oriented programming adopts.

Object-Oriented Programming

My first programming efforts in the early 1980s were with the Basic language for the Commodore 64. Basic employs a very sequential, imperative style. All the code for a program is written in one unbroken text file. Every line is numbered and the code proceeds from the first line through to the last unless some control structure intervenes to point the processor elsewhere. For all sorts of good reasons, this process of jumping around and cycling here and there is often necessary.

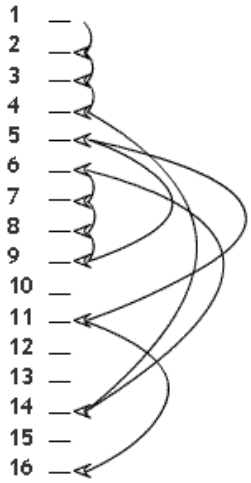

Fig. 4: 'Spaghetti' code

This mode of organization clearly echoes the structure of a basic Turing machine – it jumps discretely back and forth on a strip of tape. While this is fine for shorter programs it quickly becomes a nightmare for longer ones. The continuous strip of tape is simply not a good metaphor for complex non-linear processes.

The problem of so-called 'spaghetti' code was addressed by the next generation of code that established a non-linear and analytical organizational model. The monolithic continuous slab of code disappears, to be replaced by a more lightweight main loop that communicates with data and algorithms as necessary. This structure is often called

'structured' or 'procedural' programming because the emphasis is upon creating generic algorithms that link to specific data structures.

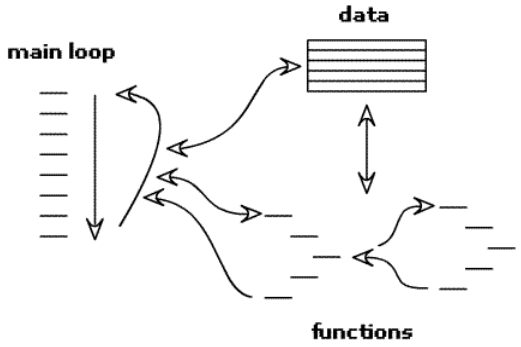

Fig. 5: Structured programming

Like linear imperative programming, procedural programming tends to work best with small programs. Large programs can lose structure and coherence, establishing a huge global pool of data and a large, undifferentiated mass of functional algorithms. The newer paradigm of object-oriented programming (see, for example, Horstmann and Cornell, 2005) arguably provides a more elegant and sophisticated means of developing clear, easily maintained and re-useable code. Object-oriented programming works to model an overall system of functional components rather than simply the passage of data through algorithms. It describes classes of entity that have both attributes and capacities. In programming a racing car game, for instance, there is likely to be the need for a class that represents the relevant characteristics of a racing car. It may have attributes such as make, colour and maximum speed, as well as capacities such as starting, turning and braking. The essential characteristics of this notional car then combine dimensions of both data (attributes) and algorithms (capacities). Instead of separating these dimensions, object-oriented programming combines them into a single logical entity. The programmer writes blueprints for potential objects. The focus shifts from describing data and abstracted generic functionality towards envisaging systems of objects that communicate. It is not that the concern with generic algorithms or data structures disappears. It is that they are encompassed within another level of abstraction.

Four principles guide the conception of object-oriented programming:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

In the case of object-oriented programming the principle of abstraction involves defining the essential characteristics of any specific problem in terms of logically distinct and communicating classes of modular functionality. The curious thing about this is how a concern with system can be linked to a mechanistic notion of discrete objects. Just as emerging modes of virtual interaction suspend our allegiance to the solidity of the real, and contemporary science and philosophy question the possibility of entirely discrete things (stressing instead relational identity), we suddenly discover, precisely within the texture of the virtual, a new realm of discernible objects. Object-oriented programmers work to define the essential characteristics of a 'thing' precisely as 'thing-hood' itself becomes problematic.

The principle of encapsulation is related to that of abstraction. To abstract is to represent in other terms. This involves not only a motion of manifestation but also a motion of hiding. Whatever it is that is abstracted disappears from view. The complexity of modern computer systems depends upon the construction of modular 'black-boxes' that can be pieced together as necessary into functional systems. We are not encouraged to look inside these boxes and are certainly not meant to tinker with their internal states and procedures. Encapsulation refers then to this deliberate work of hiding in which we engage with objects as abstracted functional entities and are not permitted to delve into the internal details of how they function. The same principle enables us to drive a car without requiring any thorough mechanical understanding of how it works. We engage with the mediating abstraction of the steering wheel rather than the messy complexity of contemporary steering technologies.

Object-oriented programming languages such as Java enforce strict protocols of encapsulation. The internal state of an object should only be accessible through designated 'public' procedures. The latter constitute an interface to the object's internal 'private' attributes (fields) and algorithms (methods). Apart from protecting data from inappropriate modification, encapsulation makes code modular. Classes can be written, tested and maintained in isolation from the functioning of the program as a whole.

The principle of inheritance refers to the capacity for classes to inherit the characteristics of other classes. The programmer defines general classes that provide the basis for more specialized classes. An abstractly defined 'moving object' class may then be extended to create a 'car' or a 'plane' or a 'spaceship' class. The various inherited classes take on the attributes and algorithms of the base class and can add their own additional features as necessary. An interesting consequence of inheritance is that it enables a class to encapsulate dimensions of its own identity and functionality – to become abstracted, as it were, from aspects of itself.

If inheritance enables logical-hierarchical and encapsulated differentiation, the final principle of polymorphism provides a means of managing multiplicity. Although cars, planes and objects are all different things (with their own independent class descriptions) they can still all be accessed as 'moving objects'. They share a common interface yet can still process differentiated input and produce appropriately differentiated results. Polymorphism serves to mediate complexity through a vital dimension of structured commonality.

It is interesting to note the variety of metaphors that inform the conception of object-oriented programming, drawing, for example, on engineering (the notion of an object as 'black-box' machine), politics (encapsulation as means of securing the boundaries between public and private) and biology (the notion of inheritance and of morphology). The concept of class interestingly relates across all three of these conceptual domains (blueprint, social class, biological class), although its main derivation is from mathematics (set theory). This multiple set of references seems pertinent to the expanded space of possibility that object-oriented programming engenders. Despite its atomic view

of real world things and processes, it enables a modular, mutable, decentralized approach to issues of structure that has all kinds of creative implications. The emphasis it places on the interaction of objects – on the elucidation of a system – suggests a very different field than that described by linear forms of media.

With this overall description of the formal features of high-level programming in place, we can now move on to consider programming as a mode of creative practice.

Code Practice

Programming is a form of writing. The programmer writes text files. However these files are not primarily intended for human reading, but for machine-reading. Unlike natural language that can cope with huge elements of ambiguity (meaning emerging through a complex process of human negotiation and inference) the artificial languages of computer programming demand an absolute semantic and syntactical precision. A single misplaced semi-colon or wrongly positioned space is enough to make the machine reader reject the code or possibly even to collapse (crash) on attempting to make sense of it. This horizon of machine reading, of execution – the necessity that code run, that it pass from text-based instructions into logically legible sequences of byte-code – vitally affects the nature of code practice. The programmer moves tenuously between two planes of visibility – the plane of written code and the plane of output (the sensible interface) – while the middle space, the space of code running occurs invisibly, as an elsewhere. As one monitors and observes the consequences of the specific play of calculations, there is both an intimate sense of engagement with code processes and an equally compelling sense of distance – of separation, of mediation.

How is this curious realm of writing – involving the passage between the human and the mechanical, the material and the abstract and the hidden and the manifest – to be conceived? As Cramer (2005) suggests in relation to the multiple, often contradictory, conceptions of the culture of software, there is a need less to elaborate a single coherent view than to acknowledge a diversity of perspectives. Programming has been variously conceived as an abstraction of the factory assembly line and as a speculative activity that bears a relation to painting and literature, as a dull motion of technical implementation

and as an experimental, questioning practice, as a form of solipsism and as a medium of social collaboration. My aim in the remainder of this chapter is to explore some of these competing, often opposed, perspectives. I should note that while Cramer adopts a very inclusive notion of software that includes not only executable code but also non-executable code, reflections on code and a more general context of speculative imagination, my interest is more specifically in modes of programming practice – with programming conceived as a form of writing that necessarily charts a relation to machine execution, that summons, and disappears within, a space of running.

Freedom and Restriction

Programming is both a highly circumscribed activity and one that enables substantial freedom. In terms of its restrictive aspect, programming is bound by precise syntactical and semantic rules, employs formulaic modes of expression and tends at the macro-level to follow recognizable 'design patterns' (Gamma et al, 1995). At the same time, however, there is a tremendous freedom to elaborate systems as one likes. Programming works at a meta-level. It allows the programmer to articulate the conditions of a medium. It permits a fundamental work of design and articulation that can establish, for instance, the nature of time, space and interaction. Programming enables a free, playful engagement with the generative, speculative character of abstraction.

Mediation in Reverse

Immanuel Kant's 1790 *Critique of Judgment* (Kant, 1980) provides a classic description of the nature and role of aesthetics. According to Kant, aesthetics works to mediate a relation between the abstract, *a priori* space of pure reason and the realm of sensible experience (Kant, 1980: 493). Aesthetics represents a moment of rationality that is pre-conceptual, that is constituted both as an intuitive recognition of a dimension of abstract order within the phenomenal (the notion of the beautiful) and as a sense of awe and wonder before vast, chaotic sensible prospects that serve to echo and metaphorically manifest the protean, infinite horizons of the *a priori* (the notion of the sublime). Experimental artistic programming practice, however, works differently – almost in reverse. It represents an aesthetic engagement with abstraction and with the trans-sensible space of computation. The latter appears as the mechanization and

exteriorization of pure reason; regarded no longer as an immediately accessible human resource but as a space of otherness that must be summoned through code. Programming represents the effort then to humanly, sensibly, engage with the terrain of the insensible. Code and the labour of coding appear as a flesh-like interface to an intractable space of abstraction.

The French philosopher Henri Bergson (1911) is famously critical of film for the manner in which it represents time. Film, he argues, decomposes time. It breaks it up into discrete instants and then attempts to reassemble continuous time from sequences of successive still images (Bergson, 1911: 306). This is to neglect the fundamental character of time – its intrinsic continuity, its duration. If film is problematic, then how much more so are binary, digital processes? The time of computation is sublimely opaque. It is not composed of perceptible instants, but of machine cycles that exceed human resolution. The speed of a standard repeat loop, even if it involves thousands of iterations, typically occurs so quickly that it cannot be registered as a temporal process; it appears instantaneous. Similar points can be made about computational space. We know that computer memory is ultimately physical but the combination of immense magnitude and microscopic scale makes it humanly inaccessible. But by actually practically engaging with this space, by allocating arrays of data, rapidly searching though them and destroying them, the practice of programming provides some kind of purchase on this terrain of abstraction. It provides a close, flickering sense of the inhuman otherness of computational processes.

Magic and Manufacturing

Cramer relates programming to the casting of spells – the magical potential to affect the material world through the agency of arcane procedural utterances (Cramer, 2005: 14). The Australian new media theorist Chris Chesher (2001) examines how the metaphor of 'invocation' informs notions of programming. He demonstrates that the relation of the programmer to the operating system and the abstract space of computer memory is conceived as a magical calling forth of spirits. Programming is described as a form of conjuring and as a means of accessing secret powers. As an example, a textbook by the artist Peter Small on the Lingo scripting language is titled *Lingo Sorcery: The Magic of*

Lists, Objects and Intelligent Agents (1996). Small introduces the field of object-oriented programming in the following hyperbolic terms:

> This book goes beyond syntax, into a conceptual world, where Lingo is used to construct strange interacting forms and structures in the mind as well as within the memory of the computer. It is a world which has to be discovered, not learned. (Small, 1996: 1)

Object-oriented programming appears as a mystical realm of understanding and the reader is positioned as an initiate into a dark art.

While the metaphor of magic is very evident within programming and suggests the fascination of a medium that links language to visible and invisible aspects of process, it is, for me, always qualified by an equally clear sense of programming as a logical, skilful practice of making. Programming is as much a work of manufacturing as of casting spells. I am very interested in this notion of programming as a work of rational, analytic construction, centrally concerned with issues of efficiency and running. The issue of how such a conception can correspond to a notion of critical art is one that I return to in later chapters. My aim there, very briefly, will be to examine the complex relation between critical questioning and building in software art, suggesting the potential for questioning to take the form of making and making the form of questioning within strands of experimental code practice.

General and Particular

In their classic 1947 critique of 20th century mass culture, 'The Culture Industry: Enlightenment as Mass Deception', Theodore Adorno and Max Horkheimer argue that culture has become thoroughly affected by capitalism. All culture now adopts the form of products to be consumed and obeys rules of standardization that take archetypal shape in the factory assembly line. Any appearance of difference is entirely superficial and illusory. There is a gesture of 'pseudo-individuation' that works only to disguise the genuine logic of uniform cultural identity: **'**[w]hat is individual is the generality's power to stage the accidental detail so that it is accepted as such' (Adorno & Horkheimer, 1982:

374). How does this critique affect our understanding of programming practice, specifically in terms of how the relation between the general and the particular is conceived within object-oriented programming? Object-oriented programming makes a fundamental distinction between classes and objects. A class is a general abstraction (a blue print) while an object is a particular thing (an instance of a class in memory). But how genuine is this dimension of particularity? Although a spaceship class may breed all manner of particular spaceship objects which vary in terms of specific designated parameters (length, colour speed, etc.), can this constitute anything more than a merely cosmetic level of difference? Does this abstractly determined particularity represent the very essence of the domination of the particular by the general?

These are precisely the questions that confront creative artificial life projects as they struggle to fashion emergent effects within the finitude of digital agar. In my view it is less an issue of insisting upon the generation of utterly unpredictable phenomena than of recognizing an overall orientation towards differentiation rather than uniformity. Object-oriented programming works at multiple layers to structure individuation in terms of the interaction of abstractly defined constitutive elements and rules. The potential for differentiated complexity is facilitated by object-oriented processes. The field of generative artificial life envisages classes as evolutionary templates – DNA style genotypes for 'breeding' particularized phenotypes (Whitelaw, 2004). Within the field of Genetic Algorithms even abstract classes themselves can be evolved. Artificial life conceives programming not in terms of the monotony of the assembly line but in terms of the rich variety of organic life. Overall, then, differentiation is less an external effect within object-oriented programming than an integral structural feature. This is clearly linked to the conceptual shift that object-oriented programming represents from a linear and centralized notion of process – to a systemic and distributed one.

Solipsism and Sociality

As a venerable convention, the first exercise in a programming textbook involves writing 'Hello World!' to the console output. This exercise has always raised questions for me. Who is speaking? Who is this greeting directed towards? Is it the thinking machine adopting a human guise and making a comic overture to a space of exteriority – the world

– that can never be its proper focus of concern? Or is it a completely uncertain gesture of communication from the programmer to a world that can only be accessed through an endless chain of mediation? Is there an irony in the tenuousness of this greeting and its obvious solipsism, or could it also express a utopian hope for social interaction? Programming represents at one level a withdrawal from the world into a closed cycle of cybernetic exchange in which the risk of human communication is replaced by the consoling rhythm of instructions and feedback, and in which human control and agency can only ever be contradictorily affirmed (as a work of both constructing a technological apparatus and being enframed within it). At another level, however, it represents a new avenue of social speech and engagement. It is both of these possibilities at once. This is very evident within the culture of hacking, which involves both a relentless concern with the arcane, technical details of computation and a determined effort to establish and participate in collaborative social networks.

The culture of hacking grew up in the late 1950s as large mainframe computers became accessible to graduate students in universities and the like. It received further impetus with the growth of the UNIX operating system, personal and networked computing, and the web. Richard Stallman is the archetypal hacker (Stallman, n.d.; Levy, 1984). A crucial contributor to the GNU/Linux project and an old-school 1960s radical, Stallman is the founder of the Free Software Foundation (Free Software Foundation, n.d.). He regards the practice of coding not as a work of commercial manufacturing but as a fundamentally political act. It is both an expression of freedom and a practical means towards it. Here a vital conceptual link is evident to the linguistic status of code. Code is a form of speech. The Free Software Foundation argues:

> Free software is a matter of liberty, not price. To understand the concept, you should think of "free" as in "free speech", not as in "free beer". (Free Software Foundation, n.d.)

However silent, however much it is spoken into the ears of a machine – code and the practice of programming somehow makes a return to the promise of the social. It is a paradoxical manifestation of the social.

Competing Methodologies

Traditional approaches to software engineering – so called 'waterfall' methods – represent the practice of software development as a predictable, linear process. Software development is conceived as a large-scale, long-term endeavour that is structured in terms of a fundamental division between processes of design and implementation. An initial stage involves producing user case scenarios and analyses to define the specific software requirements. This is followed by an exhaustive software design process in which every facet of the proposed software system is described. In relation to the technical system, thorough visual models of the object-oriented design are developed, with all attributes and capacities of individual classes indicated, as well as the complex system of inter-class relations. Only after everything has been determined in advance and meticulously documented does any actual programming begin. As a result the practice of programming is reduced to a work of technical implementation. The model of predictable industrial engineering banishes any sense of experimental design practice from software programming. It also relegates the texture of code, as writing, secondary and insignificant. Software development is regarded less as a textual process than as a conceptual one; the fully designed concept precedes the text and dominates it.

More recently, in 1996, an alternative paradigm has emerged, termed 'agile' software development (Agile Alliance, n.d.). This methodology rejects the model of large-scale modernist industrial engineering, preferring an approach in which software development proceeds in much shorter, socially collaborative and textually implicated development cycles. If software engineering adopts a linear and strictly hierarchical approach, agile practices follow a looping, modular pattern that mirrors the conception of object-oriented programming. Crucially, design becomes a part of programming rather than appearing as an altogether anterior space of determination. A problem – a possibility – is conceived, then it is swiftly clarified and bits of pieces of the problem are tackled. Software prototypes are produced which suggest further design issues and the need for further prototypes. Gradually, in an organic and iterative fashion, an overall piece of working software is developed.

Agile methodologies draw inspiration from the model of free and open-source software development. The GNU/Linux operating system was produced entirely as a community-based collaborative project. The complex cultural and technical assemblage of GNU/Linux emerged without any reference to a design blueprint. Indeed the phenomenon is simply too complex to be susceptible to exhaustive preliminary description. One of the major problems with the waterfall software engineering approach is that it attempts to manage complexity by stepping outside of it – by denying its real force. In the process the method tends to endlessly stumble across things that it had not adequately anticipated. Agile development methodologies suggest that only by moving design within software, within the complexity of programmatic textual relations, is a more coherent and pragmatic process of software development possible.

Despite their differences, both waterfall and agile software development methodologies represent programming as a form of engineering. They maintain an allegiance to logical, procedurally rigorous processes of construction. Hacking, however, is less immediately responsible and task-oriented. It begins with the wayward practice of tinkering, of messing about with bits of code. It is playful at the outset rather than directed. In his introduction to programming the Arduino microcontroller, Massimo Banzi describes an alternative embedded programming philosophy that is clearly based on the culture of hacking:

> The classic engineering approach relies on a strict process for getting from A to B while the Arduino way is based on maybe getting lost in the way and finding C instead; playing with the medium in an open-ended way, finding the unexpected. (Banzi, 2006: 7)

This methodology clearly also bears a relation to aspects of traditional aesthetic practice. In his article ‘Hackers and Painters’, the programmer Paul Graham rejects the classical engineering paradigm and searches for analogues from the creative arts to describe programming practice:

> [T]here was a name for what I was doing: sketching. As far as I can tell, the way they taught me to program in college was all wrong. You should figure out programs as you're writing them, just as writers and painters and architects do. (Graham, 2003)

This paradigm has considerable currency within contemporary software art (see Malina (1979), Reichardt (1971) and Druckrey (1999) for earlier precedents within 1960s and 1970s art and technology experimental practice). Ben Fry's and Casey Reas's experimental digital arts programming environment, *Processing* (Fry and Reas, 2001, continuing), deliberately encourages an exploratory approach. It is concerned not with the creation of finished, fully elaborated pieces of useful software but with playing with dimensions of structure and process. Programming projects are called 'sketches' and are conceived as mixed technical and conceptual-aesthetic entities. In my view, however, there is a need for some qualification. It is not a matter of reaffirming the conventional opposition between the instrumental and the aesthetic. Rather, alternative programming practice has the potential to suggest something more radical: a rethinking of the instrumental from within its own formal devices. Although it resists the culture of pragmatic software, software art nonetheless intervenes in its language and speaks its language, even when it casts its work in negative terms, even when it envisages its work solely as one of corruption. This is evident in a manifesto of the New York based group, 8-bit Collective (or Beige programming ensemble):

> Our primary foundation of Post Data is then this: the conscious corruption of data, the releasing of bits from their imprisonment within the restrictive, limiting boundaries of corporate software applications, and the exploitation of the extreme complexity of computer systems paired with the extreme intentionality of artist(s) who seek to engage the computing process at a fundamental level. It is at this point that the machine 'speaks' to us, revealing a more honest representation of the technological extremity. (Beige, 2001)

8-bit collective set out both to corrupt data and to engage 'the computing process at a fundamental level.' How is the latter possible without speaking the language of the computation, without adhering to the forms of abstraction and functioning that provide the basis for everything they set out to resist? In my view, experimental software art opens up a context in which the languages of critique and instrumental-making enter into a new relation that affects the nature of both. This is an issue that I pursue more thoroughly in Chapter 6.

Conclusion

This chapter has considered the formal characteristics of high-level programming and discussed the complex and often contradictory ways in which programming is conceived as a mode of practice. The aim has been to provide a basis for a more specific consideration of the genre of software art in the following chapter. Software art is centrally concerned with reflecting upon its status as code – with envisaging a critical meditation on code that is conducted through the mechanisms of code. What does this mean and how is it enabled? What are the specific issues and tensions that arise in attempting to position software as critical and reflective? How does this relate to programming as a practice of speculative manufacturing?

Chapter 4: Software Art

Introduction

While aesthetic experimentation with programming has been evident since the 1960s, it has tended to slip into the background – to be regarded as simply one, among many, ways of producing new media. The emphasis, as we have seen, has been on the sensible work rather than the underlying code that structures the possibility of any specific digital interface (Cramer and Gabriel, 2001). The question of the aesthetic status of this mixed space of writing and abstraction has never been quite so directly posed until the advent of contemporary software art. This chapter examines the genre of software art, considering how it has been conceived, the historical forces that have shaped it and the specific dilemmas that confront it. I begin by sketching a general context of aesthetic engagement with the field of programming.

Exclusion

The early history of computer art is typically described in terms of a rapid rise and an equally rapid fall. The British theorist Charlie Gere (2002:102-109) describes how the late 1960s avant-garde fascination with the aesthetic possibilities of the computer lasted only till the beginning of the 1970s when Conceptual Art grew suspicious, rejecting it as politically and aesthetically naïve. After the flurry of excitement and disappointment that surrounded Billy Kluver's 'Experiments in Art and Technology' (EAT), the 1968 London 'Cybernetic Serendipity' exhibition and the 1970 New York 'Software' exhibition, avant-garde art and technological experimentation went their separate ways. Conceptual Art emerged as dominant while computer art was marginalized to the aesthetic ghetto of separate art-science-technology festivals (Ars Electronica, Inter-Society for the Electronic Arts (ISEA) and Association for Computer Machinery's Special Interest Group on Graphics and Interactive Techniques (ACM SIGGRAPH)). Gere (2002: 104) argues that a primary reason for this exclusion was the growing sense that technology could no longer be regarded as an innocent mechanism of social progress. Instead, at the radical tail end of the 1960s, it came to be associated with the regime of exploitative

instrumental rationality. Aspiring to create art with a computer seemed at best wishful thinking and at worst an alibi for power.

Of course, this is to render the 1960s Art and Technology movement thoroughly unambiguous, when its relation to technology was actually more complex. For a start, the aesthetic questions that the Art and Technology movement raised were clearly continuous with dominant strands of avant-garde practice. Their reflections on the mechanical, the automatic, and the serial traced very legible links to central themes within Dada, Constructivism, 12-tone music, the OuLiPo, etc. They embraced the technological not simply as another (mistaken) sign of the aesthetic, but as a means, in classic avant-garde fashion, to undermine the notional autonomy of art. The alliance between art and engineering served both to question art's complacent distance from the world of technologically manufactured things and engineering's lack of concern with issues of human value and imagination. In this sense, technological art maintained a critical dimension. It was never simply affirmative. Edward Shanken argues that the reflective and parodic character of the Art and Technology movement's engagement with technology is often overlooked. Actually their work is 'infused with irony, their technological or pseudo-technological components must be interpreted as parodies of scientific structures of knowledge and their uncritical application in art and society.' (Shanken, 2004: 246).

Programming, as a specific form of technological engagement, was awkwardly positioned in terms of the split between technological and conceptual tendencies. Unlike heavy bits of machinery, it could hardly be condemned as basely material. Indeed the curator of the 1970 'Software' exhibition, Jack Burnham, explicitly positioned software as a metaphor for the properly conceptual character of art (Shanken, 1998: 1). The latter was modelled on the immaterial and system-focused realm of computer software. The notion of software provided a bridge between Cybernetics (with its concern with the techno-ecology of systems), Structuralism (with its concern with language and culture as an abstract combinatory system) and Conceptual Art as a formal and critical interrogation of dimensions of meaning and process. Yet despite its status as a metaphor for art and as a bridge between a variety of emerging modes of reflecting on systems, software

programming – in its clear orientation towards the dimension of machine process – could hardly avoid the consequences of the more general technological exclusion. The 'good' abstract conceptual stuff of programming was sublated into the algorithmic play of Conceptual Art (the instructional drawings, for example, of Sol LeWitt), while the 'bad' material residue of executable code was abandoned.

The consequences of this split were felt for more than two decades. The experience of the British artist, Paul Brown, is representative. As a young artist he was inspired by the provocative blurring of boundaries between the fields of art and science in the London Cybernetic Serendipity exhibition (Brown, 2003: 1). He became interested in computers, particularly as a means of exploring issues of generative structure and process related to the emerging field of artificial life. For this reason, he argues, his work attracted little mainstream interest:

> Thanks to my longstanding interest in computational systems as a medium for the visual arts I have been relegated to the fringes of the arts mainstream for most of my career. (Brown, 2000: 2)

He was not altogether unhappy about this exclusion. Alienation had its perks (alternative festival circuits and funding sources). When he detected a new interest in digital art from the 'global art mafia' (Brown, 2000: 2) in the early 1990s, he was relieved to find that it was thoroughly superficial 'as a unique new paradigm for the arts quickly fell prey to the […] "no skills please – we're postmodernists" kind of rhetoric that the international contemporary arts scene use to defend their position whenever it is threatened' (Brown, 2000: 2). Very apparent is the continuing sense of slight associated with Conceptual Art's rejection of the technical aspect of process – its attempt to reflect critically on the nature of systems without literally partaking of their material technical forms. While on one level Paul Brown's stance is practice-focused and vehemently anti-theoretical, at another level it represents a meta-critique of the notion of critique itself – of the assumption that critique can be purely articulated apart from dimensions of technical process.

In his short essay 'The Death of Computer Art', Manovich (1996) reflects upon the incompatibility between the art world and alternative networks of creative computing. The art world, which he terms 'Duchamp-land', is characterized by a concern with conceptual 'content' and critique, while the sphere of creative computing ('Turing-land') focuses on technological novelty and upon exploring the aesthetic possibilities of the computing medium. According to Manovich, the two tendencies are inevitably opposed and any attempt to integrate them is bound to fail:

> What we should not expect from Turing-land is art which will be accepted in Duchamp-land. Duchamp-land wants art, not research into new aesthetic possibilities of new media. (Manovich, 1996)

Some five years later, Cramer (2001) reassesses Manovich's claims and argues that the distinction is overly simplified. He rejects the association of 'techno-positivist' computer art with the founder of modern computer science, Turing, arguing that the latter's work, the discipline of computer science and wider hacker culture can be deeply reflective and ironic. While acknowledging the point that dominant strands in computer art, associated with exhibition and research contexts such as *Ars Electronica* and *Zentrum fur Kunst und Medientechnologie* (ZKM), reveal utopian tendencies, he argues that there are 'subtle transitions between both options' (Cramer, 2001). How did this more nuanced perspective develop? What had changed during those five years to prepare the way for software art?

New Relations

Very generally, the period between 1996 and 2001 was one of massive growth and consolidation of digital culture within Western societies. The proportion of home ownership of computers in the US, for instance, increased from about thirty to fifty percent (Leigh and Atkinson, 2001: 3) – a faster rise than had occurred in the whole past decade. The web also experienced exponential growth, gaining a strongly commercial focus and becoming much more technologically sophisticated. If early web sites were mainly static html pages, now they became dynamic, database driven sites, linking complex back-end technologies to increasingly self-consciously designed and interactive

front-ends. The computer game industry also gained vital impetus. ID Software's seminal first-person shooter, *Quake*, was released in 1996, pushing the real-time display of 3D computer graphics to an unprecedented level. Furthermore, a whole suite of 'creative software' and hardware appeared that enabled the first wave of accessible digital photography, video, audio and multimedia.

Within this context, it was no longer possible, as an artist, to ignore digital culture and technology. It demanded attention and engagement. The web became a key area of concern. If a major prior focus of cultural critique had involved exploring the dimension of language within the apparently intuitive media of photography and film, the internet represented a far more literally code-based medium. Engaging with code was no longer a matter of exposing hidden ideological mechanisms, but one of deconstructing the actual, material – abstractly inclined and pragmatically procedural – texture of power relations. Net art drew upon the radical traditions of Dada, Situationism and Fluxus to provide a deliberately low-tech critique of the politics and aesthetics of the mainstream web (Weibel and Druckrey, 2001; Paul, 2003; Tribe and Reena, 2006). Olia Lialina's *My Boyfriend Came Back from the War* (1996) provides a classic example, rethinking the web page as a strange conjunction of hyper-text and film montage. The field of net art played a crucial role in highlighting the space of code and established a vital context for artists to develop relevant technical skills. In writing their own html pages, artists gained insight into the technical nature of the web and a pathway to more dedicated forms of programming. Another pathway was provided by proprietary multimedia software packages, such as Macromedia *Director* and *Flash*, that included powerful and accessible scripting languages.

Net art grew increasingly technically ambitious during this period. No longer content to adhere to the conventional notion of the web as a network of tidy discrete pages, artists began to process html code themselves, creating alternative browsers that represented the web as dynamic, code-centred space. The London collective IOD's *Web Stalker* (1997) software is a famous alternative browser. The software provides no coherent assembled pages, only visible html code and maps of data and their networked relationships. It portrays the web very explicitly as a coded abstraction. Another example is Mark

Napier's *Shredder* (1998), which creates a jumbled collage of page elements. The work emphasizes that code and interface are not the same thing – code opens up a space of playful re-combinatory possibility that ordinary browsers repress. In their emphasis on deconstructing code processes through the mechanisms of a code work – as well as in their combination of conceptual-political themes and technical engagement – these works appear as seminal, anticipatory pieces of software art.

Alongside artists intervening in the technology, technologists were doing things that strangely resembled critical art; specifically in terms of modeling and realizing the kinds of collaborative social relations that radical art was intended to foster. In 1999, the *Ars Electronica* festival awarded their Grand Prize to the Linux operating system. Many artists were appalled, questioning how a purely instrumental technological project could possibly warrant an art award (especially when net artists were still struggling to obtain mainstream recognition) (*nettime* mailing list, 1999). Others, however, recognized it as useful provocation; a gesture from the aesthetic ghetto concerning the nature of genuine radicalism. This award also suggested the increasing confidence and cultural relevance of alternative contexts such as *Ars Electronica* which were no longer so clearly marginal and out of step with currents of cultural critique. According to Jacob Lillemose (2004: 141) a third wave of Conceptual Art developed in the early 1990s that was characterized by a heightened sense of social and political activism. It was concerned to intervene within structures of power rather than to enigmatically reflect upon systems of meaning within the autonomous and inevitably compromised space of avant-garde art. These currents of conceptualism recognized GNU/Linux as a highly relevant paradigm for alternative cultural-technological practice.

If *Ars Electronica* was established in the late 1970s to question the neat divisions between art and science, now this work was being undertaken at a far more general cultural level. The web provided a context – and a multitude of particular contexts – in which artists and technologists could meet beyond the straightjacket of conventional disciplines. Another context was the rapidly expanding field of computer gaming, which led programmers to take an increasing interest in issues of graphic visualization and interaction, and artists to take a more dedicated interest in the potential for non-linear,

generative abstraction that programming enables. Multimedia production served as another important interdisciplinary context. Neither art nor science, it appeared as a messy, non-rigorous, exploratory field in which artists could pretend to be technologists and technologists could pretend to be artists. All manner of significant exchange occurred on the basis of the inevitable and uncertain relation between the programming 'back-end' and the visual 'front-end'. Specifically, the technologically-oriented web forums provided an important model for social collaborative communication and practice. Unlike many of the agonistically conceived cultural theoretical forums, the technological forums were generally friendly places structured to provide information, answer questions and facilitate community.

One other development is worth mentioning. The first decades of personal computing had focused on making the computer as humanly accessible as possible. They had been dominated by the holy grail of the natural, intuitive interface. In the late 1990s, however, there was a shift in orientation back to the paradigm of text. This was associated particularly with the rise of the GNU/Linux operating system, which placed the emphasis on acknowledging computational complexity – allowing end users to engage with their systems in sophisticated ways via the command line interface. This made computers accessible in a different sense, not so much as simulated worlds and transparent (encapsulated) tools but as spaces of meta-level organization that demanded human intervention. A similar philosophical stance led many artists to shift away from the use of commercial 'creative software' towards an engagement with the medium of programming itself as a more open and genuinely creative space. Within the overall context of commercial computational complexity, this represented a deliberate anachronistic turn. Artist-based programming integrated development environments such as John Maeda's *Design By Numbers* and Ben Fry's and Casey Reas's *Processing* were deliberately structured as code-based interfaces. Running against the trend to make everything accessibly visible, they made the ordinarily invisible space of code manifest and accessible.

Much had changed then in the five years that separate Manovich's and Cramer's views on the relation between computer art and mainstream art. Mainstream art had re-engaged

with technology and technological art had become more mainstream – broaching questions that critical art itself wished to pose. The genre of software art appears within this context – at least initially – as a moment of reconciliation of conceptual and technological traditions. At the same time, however, it also provides a context for the reaffirmation of traditional distinctions.

Software Art

Contemporary software art achieved its first notable recognition in 2001 when a prize for 'artistic software' was awarded at the Berlin *Transmediale* media arts festival. Subsequent key events included the 2002 Read_Me 1.2 Software Art/Software Art Games festival (Moscow) and in the same year the Whitney Museum's CODEeDOC exhibition (New York). It is worth examining the critical statements and work associated with these events as a means of indicating the overall characteristics of software art, as well as specific tensions that constitute the field.

As I suggested in the previous chapter, the jury for the *Transmediale* festival define software art precisely in terms of its difference from new media. Software art, the jurors argue, shifts the focus from the visible surface of digital art to the constitutive space of code. They regard programming code as fundamentally different from traditional media in that the former is not a passive intermediary; code does something, it is executable, it performs actions:

> Perhaps the most fascinating aspect of computing is that code – whether displayed as text or as binary numbers – can be machine executable, that an innocuous piece of writing may upset, reprogram, crash the system. (Transmediale.01 Media Arts festival jury, 2001)

Programming represents a new condition of writing, in which written abstraction obtains powers of literal agency. On this basis, the jury rejects the conventional notion of software as a tool. It is less the instrumental character of a tool, in this instance, that is at issue than its passivity. They argue that 'digital code is virulent' (Transmediale.01 Media Arts festival jury, 2001) and that it can only appear as a tool by disguising its actual

operations. Software art then has the potential – and crucial aesthetic duty – to expose the machinations of code, to make code visible. At the same time they suggest that 'it is itself a ground for creative practice' (*Transmediale*.01 Media Arts festival jury, 2001). This is an interesting addition, because it suggests, already, an area of tension and a lingering basis of distinction; at one level software art is represented as a form of critical revelation – as a meta-level reflection on code processes – while at another level it constitutes a basis for creative experimentation. Beyond a motion of critique of dominant discursive and operational regimes of software, it seems that software art can also explore the formal conventions of the medium. Both tendencies are founded upon the making visible of code that is constitutive of software art. Software art as a form of critique necessarily takes shape as a coming to critical consciousness of code, while software art as aesthetic experimentation depends upon the medium of code distinctly appearing. The jury makes a deliberate effort to be inclusive, to encompass formal, critical-contextual and other tendencies:

> Software art could be algorithms as an end to themselves, it could subvert perceived paradigms of computer software or create new ones, it could do something interesting or disruptive with your computer, it could be creative writing, it could be science. (*Transmediale*.01 Media Arts festival jury, 2001)

The 2001 software art prize was shared between three works. Two were visually abstract, generative software projects (Antoine Schmitt's *Vexation 1* and Golan Levin's *Audiovisual Environment Suite*) while the third project, Adrien Ward's *Signwave Auto-Illustrator*, included aspects of both software criticism and formalist experimentation. A semi parodic re-make of the commercial vector drawing package, Adobe *Illustrator*, the interface represents an ironic reflection on the idiom of instrumental software, while the actual drawing operations reveal a concern with procedural pattern-making and generative artificial life.

In a subsequent article, two of the festival jurors, Cramer and Gabriel, describe the historical context of the emerging genre in terms of its links to the tradition of Conceptual Art and, more specifically, to strands of socially-critical, meta-reflective net art (Cramer

and Gabriel, 2001). They distinguish two tendencies within Conceptual Art, one that focuses on issues of immaterial structure and system (associated with artists such as Henry Flynt, Sol LeWitt, John Cage and La Monte Young) and another that is more socially and critically focused (the later Joseph Kosuth, Hans Haacke and Vito Acconci). The former appears as a mode of formalism, while the latter has a cultural, contextual and activist orientation. Cramer and Gabriel lean towards the latter, suggesting that software art 'has become less likely to emerge as conceptualist clean-room constructs than reacting to these stereotypes' (Cramer & Gabriel, 2001: 3). The Read_Me 1.2 and CODeDOC exhibitions represent two different responses to this heritage.

Leaning Toward Critique

The Read_Me 1.2 festival can be interpreted as a development of the critical-cultural orientation within software art. However, the jury's classic definition of software art maintains the emphasis on an inclusive approach:

> Since read_me 1.2 is one of the pioneering festivals of software art we felt it necessary to open up the field rather than to prematurely narrow it down. We consider software art to be art whose material is algorithmic instruction code and/or which addresses cultural concepts of software. (Read_Me 1.2 jury statement, 2002)

If strands of aesthetic experimentation and critical-cultural practice can coexist within software art it is because both entail a close engagement with the labour of programming. The crucial difference between the contemporary situation and the earlier moment in which computer art and Conceptual Art diverged is that contemporary conceptualism, whether formalist or critical, is determined to get its hands dirty – to engage with code practically. For critical software art this entails an effort to intervene within the culture of software rather than to remain at a safe distance. The call for entries to the Read_Me 1.2 festival articulates this engaged perspective:

> In order to stay current, an artist must acquire new methods of working in social spaces and react to the questions that concern society. An artist immersing

> himself in the production and software development spheres – areas allegedly intended to facilitate our lives through "progress" – has a chance to find his audience and to actually influence culture. (Read_Me 1.2 festival organizers, 2002)

The critical-cultural orientation is made very explicit in the three styles of code entry that the festival invites: code that leads standard software 'astray'; deconstructive code; and '[w]ritten from scratch' software that resists the model of software as a rational, pragmatic tool' (Read_Me 1.2 festival organizers, 2002).

The overall prize was split between three software projects: Mark Daggett's *Deskswap*, which allows users to load other people's desktops from around the world as a means of reflecting upon issues of globalization; Eldar Karhalev and Ivan Khimin's *ScreenSaver*, which is a set of simple (non-technical) instructions for altering the Microsoft operating system screensaver; and Joshua Nimoy's *Textension*, which playfully re-conceives word-processing as the creation of concrete-poetry style visual patterns of text. The emphasis is clearly on work that is critical rather than formalist, although *Textension*, like *Signwave Auto-Illustrator*, manages to bridge both strands of practice – shaping critique as a form of speculative aesthetic enquiry.

Leaning Toward Formalism

At least in terms of its overall curatorial conceit, the Whitney CODe_DOC exhibition (2002) represented a more formalist approach. It provided its twelve participating artists with a single conceptual exercise: 'This code should move and connect three points in space.' In its minimal abstraction, this clearly recalls the procedural rhetoric of formalist Conceptual Art. The *Transmediale*.01 jury had lamented the lack of visible source code amongst the submitted works (*Transmediale*.01 Media Arts jury, 2001). CODe_DOC responded to this concern by placing a primary emphasis on displaying source code. Project links on the website (http://artport.whitney.org/comissions/codedoc/index.shtml) contained a thumbnail image of the work, the artist's name and their specific programming language. Clicking on a link led directly to the artist's source code – only there, perhaps, in the code comments could you actually find the title of the work, and

only there, by scrolling to the end of the code, could you discover a link to 'the work' itself. One of the works, Alex Galloway's *What You See Is What You Get*, provided no visible interface whatsoever. A collection of short, illicit, apparently malevolent scripts, the project was not run but read, legibly demonstrating how easy it is to programmatically choreograph social (informational) disorder.

```
############################################################
# ASSIGNMENT: Move and connect three points in space #
# RESULT:     What You See Is What You Get (5.1 K)   #
#                                                    #
# BY:         RSG (feat. Alex Galloway)              #
############################################################



#ENTRY POINT
$first_point[0] = "Universal Email Header Spoofer"; #PHP script, 753 bytes
$first_point[1] = "How To Hack Hotmail Passwords";  #Text file, 933 bytes
$first_point[2] = "Confidential Business Proposal"; #Text file, 1682 bytes
```

Fig. 6: Alex Galloway, *What You See Is What You Get* (2002)

Galloway's work appears as the most pointedly critical-activist. Other works in this camp included Sawad Brooks collage of global news sites, *Global City Front Page* (Brooks, 2002), and Golan Levin's humorous reflection on the absurdities of US foreign policy (paranoia), *AxisApplet* (Levin, 2002). Levin's work provides a map of the world. The user clicks on any three countries and a text note explains the nature of the axis.

Clicking on Brazil, Russia and Australia, for example, produces an 'axis of huge, oil-producing, vodka-exporters'. In his code comments, Levin explains:

> President Bush's assertion that North Korea, Iraq and Iran form an 'Axis of Evil' […] was more than a calculated political act – it was also an imaginatively formal, geometric one, which had the effect of erecting a monumental, virtual, globe-spanning triangle. (Levin, 2002, code annotation)

Here then a strange link is opened up between cultural and formal interests. Culture appears as metaphor for a dimension of abstraction, which seems a back-to-front way of regarding things but also indicates the charged, metaphoric character of coded abstraction. It is less about the immediacy of vision (in a classic modernist sense) than the instantiation of aspects of structure and system. I will come to the more explicitly abstract works (which constitute half of the overall works) shortly, but three other works represent an interesting transition to more general formal concerns.

Fig. 7: Maciej Wisniewski, *The Meaning of Life Expressed in Seven Lines of Code* (2002)

Maciej Wisniewski 's *The Meaning of Life Expressed in Seven Lines of Code* is an enigmatic work that addresses issues of time and distance through the combinatory simultaneity of grid-based display. It establishes a poetic friction between the array-

based structures of code and dimensions of human experience. Wisniewski suggests that the work 'depicts a skewed view of geography, time and history, whose space and time elapses during the day and at night and stretches itself at sunrise and sunset' (Wisniewski, 2002, code annotation). The concern here is not with the politics of software but with its poetic formal imaginary.

Brad Paley's *CodeProfiles* reveals a strongly self-reflexive concern with software. It is a work in which code seems to display itself. The code appears in a semi-abstract manner as sets of programmatically formatted lines. Moving the mouse across specific lines displays the actual code. The code's operations are depicted as waves of changing line colour and white arcs that communicate between one code module and another. Paley describes the project in the following terms:

> The code reads in its own source and displays it in a tiny font, then moves three points in "code space". It essentially comments on itself. (Paley, 2002, code annotation)

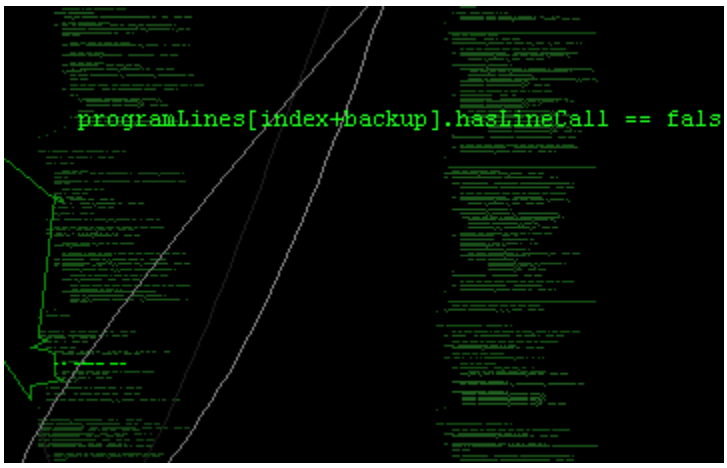

Fig. 8: Brad Paley, *CodeProfiles* (2002)

CodeProfiles is cast then as a work of recursive meta-reflection. It charts associations not only to Conceptual Art but to the recursive irony of Turing and hacker culture (the latter is perhaps most famously evident in Richard Stallman's acronym GNU, which stands for GNU is Not Unix). The work shapes reflection then in formalist rather than critically engaged terms.

John Klima's *Jack and Jill* (Klima, 2002) is a playful meditation on game-based parametric characterization and life. Jack and Jill represent two points and the bucket at the top of the hill represents the third. A set of simple radio buttons allows the user to control whimsical aspects of each character's behaviour, from how eager they are to get the bucket to whether or not a male chauvinist or feminist approach is preferred. The characters then respond appropriately – typically frenetically crashing into one another as they run up and tumble back down the hill. The references here are to the cultural field of gaming; however the work represents less an example of cultural critique than a whimsical meditation on aspects of conventional game form and modes of programmatic identity.

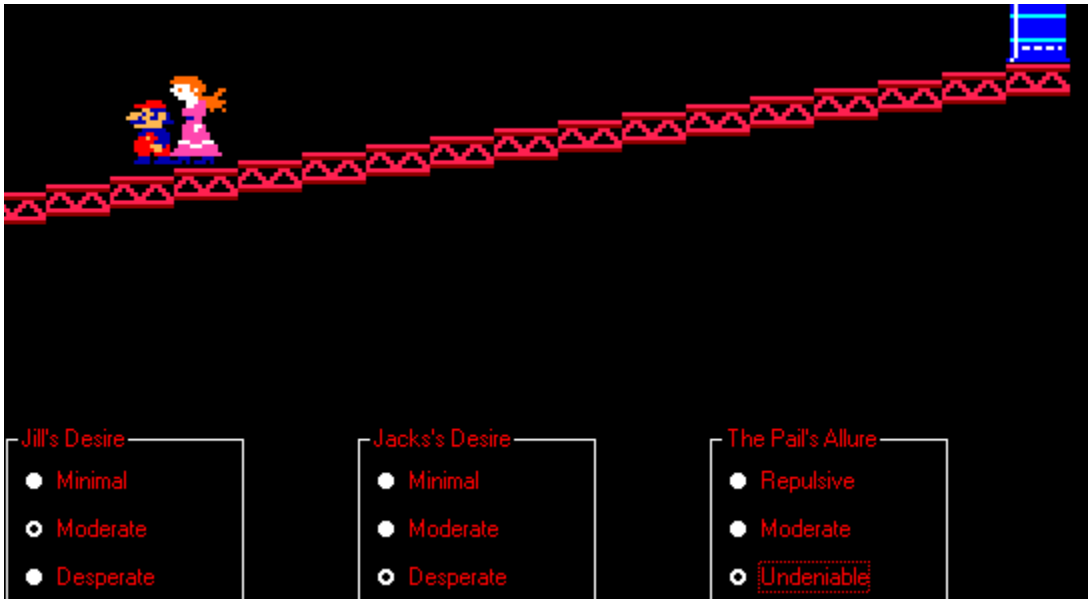

Fig. 9: John Klima, *Jack and Jill* (2002)

The remaining six works are much more explicitly formal in orientation, adopting a strategy of visual abstraction. Martin Wattenberg's *ConnectApplet* provides a representative example. A set of three points form a triangle hemmed in by rippling lines. The points can be moved to alter the triangle, which slowly transitions to its new state. Within this context, visual abstraction serves as a means of signalling the structuring force of code as another order of abstraction. As Brad Borevitz suggests, '[t]here is a way in which the basic programmatic logic of the work is as clearly evident in its visual presentation as it is in the code itself' (Borevitz, 2004: 304). The work

represents a reflective circle, with the visualization summoning an awareness of the code and the code structuring the visual display.

Fig.10: Martin Wattenberg, *ConnectApplet* (2002)

Other works by Mark Napier, Kevin McCoy and Camille Utterback work in a similar fashion (Napier, 2002; McCoy, 2002; Utterback, 2002). Mark Napier's *SpringyDotsApplet* is interesting because it introduces a third dimension of conceptual abstraction. Alongside the code and the softly transparent algorithmic drawing there are dots and lines that serve to further reinforce the relation between the conceptual layer of mathematical abstraction and the manner of visualization. The dots and lines make the underlying constraints that structure the formal exercise lucidly apparent.

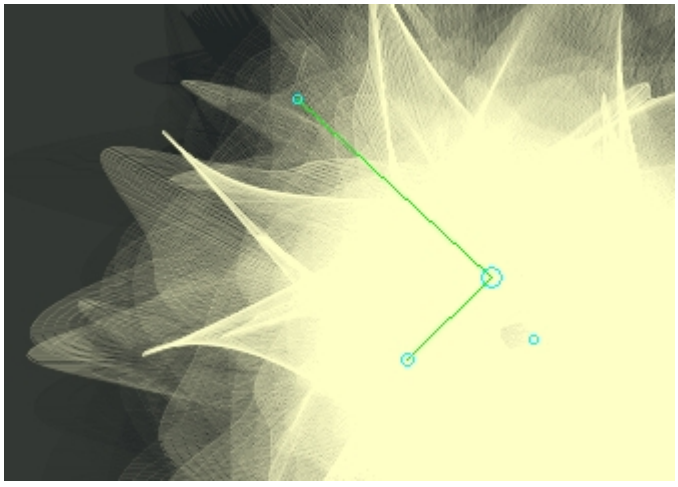

Fig. 11: Mark Napier, *SpringDotsApplet* (2002)

Scott Snibbe's *Tripolar* draws flower-like squiggles based on a chaos algorithm; it simulates a pendulum swinging above three magnets. This is the stuff of school science fairs, except that Snibbe's code comments suggest another level of concern:

> The source code demonstrates the "meta-chaos" of the program itself. A set of key variables defines all the parameters of the simulation. Changing any one of the parameters radically alters the artwork, in most cases making it non-functional – in some cases the program will hang, in others the paths will explode, implode or oscillate. (Snibbe, 2002, code annotation)

The software reveals, once again, a recursive aspect. It stages chaos in terms of its own parametric deconstruction. Evident here is a peculiarly abstract rendering of critical-cultural concerns, in which culture is replaced by a cosmological perspective on the behaviour of systems. Deconstruction becomes less political than mathematical and scientific.

Fig. 12: Scott Snibbe, *Tripolar* (2002)

A More Fundamental Tension

I have described a context for software art, some crucial initial events and two broad aesthetic tendencies. It is beyond the scope of this study to offer a detailed survey of the development of software art since then. Both formal and activist tendencies are still very much in evidence, although the former probably has greater prominence. The formalist-inclined *Processing* integrated development environment won the Gold Nica in the Net Vision category at the 2005 *Ars Electronica* and has provided an important context for

experimental digital arts practice. Many art schools around the world have adopted it as a teaching tool. *Processing* has also moved beyond an exclusive focus with code to engage with the sphere of physical computing. A number of sister projects such as the *Arduino* microcontroller/IDE provide an accessible means for artists to engage with the creative possibilities of electronics and kinaesthetic-interactive new media. Another area of expansion has been into the field of mobile device computing and locative media. These are efforts to move experimental software art beyond the narrow scene of conventional desktop computing. Critical-cultural software art lacks this technically-oriented identity. It coheres instead in events such as the annual Read_Me festival, community sites such as the Run_Me software art archive, and a range of publications, such as the proceedings of the 2004 Software Art & Cultures Read_Me festival (Goriunova and Shulgin, 2004) and the writings of Fuller (2003) and Cramer (2002, 2005).

While I have focused on the difference between formalist and critical-cultural software art, and the distinction certainly has currency within the field, the distinction is also, in my view, deeply problematic. It works to obscure and repress more fundamental tensions that affect the character of software art. Apart from any critical questioning of the nature of coded systems or of the culture of software, software art suggests even more basic questions concerning the relation between software (as a space of instrumental making) and art (as a space of aesthetic making). The current emphasis on formalism versus culturalism (Cramer, 2002; Lillemose, 2004) works to neglect the relation to software precisely. It describes a tension within art, within the space of art's conventional imaginary. Formalist software art, it would seem, sublates the discursive practice of technical software into a properly aesthetic, purely conceptual, space, while critical-cultural software distances itself from the rational and instrumental dimensions of software to focus entirely on gestures of critique and deconstruction. In both instances, software is lifted up to art. Software art is regarded as software stripped of its awkward and embarrassing features to become art, abandoning its base concern with efficiency, disguise and tool-based functioning to appear either as a space of formal purity or of socially engaged action. But this is to ignore that both formalist and critical-reflective software art engages with code at a practical level and that this engagement has

consequences that affect art as well as software. Formalist software art does not simply deal at the level of the rarefied aesthetic concept; it writes software, it engages with the medium as a specific discursive field. Similarly critical-cultural software must actually work (if it is to be literally software and not an aesthetic commentary on code). It must function. It must adhere to the same regimes of abstraction, procedure and rational efficiency that constitute the language of instrumental software production.

My interest in the second half of this thesis is to explore various aspects of this tension between software and art. The tension takes shape in terms of three basic dilemmas that confront software art: the dilemma of position; the dilemma of visibility; and the dilemma of recursion.

The Dilemma of Position

How is software art to conceive its relation to more general practices of software production? Particularly, how is it to conceive its relation to the spectre of commercial software production? Programming, after all, is not simply a language to be freely spoken; it is socially and economically situated. There are specific conditions of speech. In old-fashioned Marxist terms, the dilemma here is of conceiving the relation to the means of production.

In their recent survey of currents within contemporary digital art, *At the Edge of Art*, Joline Blais and Jon Ippolito (2006) employ biological metaphors to describe the strategic position of digital art. Society is conceived as a body that is really and imaginatively assaulted by the virus of technology. Digital art is cast as a protective antibody that adopts the form of viral technology in order to defeat it (Blais and Ippolito, 2006: 8-13). This conception seems very problematic. How can the integrity of the social be conceived apart from the thought of technology? If, as Blais and Ippolito argue, technology is not only the exteriority of machines and code, but also something that is vitally socially constructed (Blais and Ippolito, 2006: 10) – that exists as much ideologically as materially – then how can technology be regarded as a virus? How can it be conceived as an exterior threat? Furthermore, if art serves as an antibody then how can it possibly match the power of the technological virus? How does it find the means

to match its rapid mutations? How does it hope, within the fragile and contradictorily autonomous space of art, to actually defend this society (stripped of its technological basis, conceived in vague, romantic humanist terms) from such a huge and abiding threat? In my view, Blais and Ippolito's conception of digital art oversimplifies the complex relations between society, technology and art, accords digital art an impossible measure of social responsibility, and reduces art to a role of amelioration and protection (neglecting its irruptive potential).

A more persuasive model of art's social positioning is provided by the Frankfurt School theorist Theodor Adorno. In his last and unfinished work, *Aesthetic Theory*, Adorno (1997) emphasizes the contradictory character of art's autonomy, which at once resists regimes of instrumental rationality and is necessarily implicated within them; not only in terms of its commodity status but also in terms of its legibility as a spiritual alibi for capitalist social relations. While Adorno approves art's capacity to model alternative social relations and non-exploitative forms of labour and manufacturing, he is also keenly aware that this space of autonomy – in which alternatives are projected – is illusory, compromised, and thoroughly determined by the wider social relations that enable and circumscribe it. In this tension, in this friction, in this sense of contradiction, Adorno discovers art's genuine critical capacity. Art must tease out these social tensions in the very texture of the work – less at the level of shaping explicit political messages than in terms of engaging rigorously with the technical, formal and material conditions of a specific medium.

In this light, if software art wishes to regard itself as a form of reflective critique, then it must acknowledge its dependence on the broader institutional and discursive space of conventional software, its involvement in its language, mechanisms and cultural forms, and its relative lack of social purchase and cultural power. It is this awareness of dependence, marginality and inevitable association that lends software art the capacity to pose worthwhile questions for both art and the realm of a narrowly conceived instrumental rationality.

My aim in chapter 5 is pursue this question of position within the specific context of software art's relation to the 3D graphics engine. I argue for a notion of anachronistic experimental practice that both draws upon and questions the technological and aesthetic models provided by commercial software production.

The Dilemma of Visibility

Software art insists upon making code visible. This is its primary demand. Only by making code visible can it constitute programming as aesthetically reflective. Yet code itself would seem to resist visibility, and to resist it in a least two distinct ways: firstly by deliberately, structurally, hiding; and secondly by disappearing within the context of machine functioning. Let us consider each of these two modes of resistance.

Hiding

The first is classically evident within object-oriented programming in terms of the principles of abstraction and encapsulation (see chapter 2). Abstraction is not only the positive representation of something in a symbolic form; it also indicates a motion of leaving behind. That which is abstracted no longer itself appears. It is replaced by the abstraction. While this work certainly has a reflective aspect, its consequences are to make reflection itself more difficult. The many layers of computational process work precisely to make lower layers disappear. The principle of encapsulation denotes a particular form of this disappearance in which specific internal features of an object are deliberately hidden from view in order to protect them from unwarranted interference and to enable the simplicity of a general public interface. Encapsulation works both to protect the integrity of individual objects and to enable them to be treated as simple building blocks in more complex structures. A work of hiding then is implicit within the linguistic structure of contemporary programming. Object-oriented programming involves choreographing a play of hiding and manifestation. Within this context, how is the relation to art to be described? What does it mean to insist that code be visible, that it resist its own discursive strategies, and plainly appear?

The approach, adopted by the CODeDOC exhibition, of literally displaying code may have been appropriate within the context of drawing attention to this normally neglected

space, but seems hardly adequate as a typical strategy. For a start, code is simply not legible to non-programmers. Visible code makes the very general point that code is significant, but beyond that it serves as little more than a connotative surface – indeed it can quickly work to mystify software art, to suggest some realm of arcane, abstract power that bears little relation to actual, practically-directed programming. Even programmers have difficulty simply reading code. Even the person who actually wrote the code can have trouble making sense of it (especially after a few days or weeks away from it). Code is most legible as it is being written, especially in the alternation between writing and execution. In this sense, it resists entirely contemplative visibility (the traditional form of the aesthetic). Code is engrossing within the overall event space of writing, performance and debugging. To some extent, the notion of software art represents the artist-programmer's fantasy that this space of creation may somehow take literal, exhibitable shape for an audience. But this is not really possible. Programming is essentially participatory rather than something to be seen (an aesthetic spectacle). In order to become visible it has to persist with abstraction. It has to hide and shape disguises. It has to render the dimension of code metaphorically apparent. Cramer and Gabriel acknowledge this point when describing the *Web Stalker* alternative browser:

> The code of the Web Stalker may dismantle the code of the Web, but does so by formatting it into just another display, a display which just pretends to "be" the code itself. (Cramer and Gabriel, 2001: 2)

In this sense, the notion of rendering code visible entails something other than a puritanical resistance to code's processes of disguise and layering. Revelation is itself a staging, a manifestation, a motion away from origin.

Functioning

The second mode of disappearance is operational. Code disappears in the motion of its running. It passes away from itself. This reveals a fundamental, existential form of the instrumental, for the instrumental is that which does not stop, which does not rest upon itself, or imagine itself as an end, but that rather directs attention elsewhere, shaping, as Heidegger argues in relation to technology, a 'bringing forth' (Heidegger, 1978: 293). So

to make code visible is ultimately to insist that it stop, that it no longer be constituted as a passage into functioning. Beyond this global context of functioning, there is the whole question of how art can engage with code's integral and intricate instrumental character, of the way it is structured at all levels in terms of the demands of efficient running. Could this actually represent the genuine critical challenge of software art: to somehow find the means to reflect at the limits of reflection (within the blindness of process)?

The dilemma of visibility raises then a broader issue: strategies of disguise and disappearance are aspects of the instrumental character of software. Inasmuch as software art produces 'working' software it inevitably engages with issues of silent, hidden and efficient operation. I explore the issue of software art's relation to the instrumental character of software in chapter 6.

The Dilemma of Recursion

In an article on Cramer's conception of software art, Troels Degn Johansson argues that Cramer paints software art into a corner (Johansson, 2004: 151). All it can do is reflect on its own conditions of formal or cultural-critical being. It is stuck in a recursive loop of endless and ultimately disabling self-analysis. This is evident in the strategies of visual abstraction within formalist software art, as well in the emphasis on meta-level critical commentary within culturalist software art. Partly because the mechanisms of code are so hidden within conventional software, software art must devote all of its energies to the contrary work of reflective exposure. Is this inevitable or is there the potential to represent software art differently, to imagine code – the mechanisms of code – as a form of opening? If so, how can this work of opening be enabled without compromising the work of reflection? One mode of opening is the relation to the discursive space of instrumental software that software art maintains as an inner tension. My aim in Chapter 7 is to explore three other possibilities of opening.

The first is straightforward; it involves tracing relations beyond the apparent aesthetic enclosure of software art, exploring its links to wider aesthetic concerns and forms of media. Rather than constituting an entirely unique and autonomous discursive space, software art is shaped by and enters into relation with all manner of other traditions and

technical forms of art. I am particularly concerned with how aspects of my video art practice anticipate and are affected by my software art practice.

The second form of opening involves conceiving a relation between software and the alterity of the real. In its abstraction, in its systematic character, software may seem altogether removed from a traditional concern with the otherness of real things, but precisely through its difference it can also manifest a return; one that takes shape as a friction and poetic summoning. This concept is explored in relation to my interactive documentary project, *Halfeti – Only Fish Shall Visit*.

The final possibility of opening involves questioning the thinking of recursive closure itself. A genuine reflection upon software reveals a dimension of non-identity within the texture of computation. Rather than a repetitive finitude that is absolutely clear, transparent and reflective – there is a dimension of alterity that breaches software from within. This is evident for me less in programs that deliberately conjure up the illusion of organic difference than in works that pursue the binary to the point that it suddenly becomes mysterious.

Conclusion

This chapter has explored the emergence, bifurcated conception and dilemmas of software art. My argument is that the opposition between formalist and culturalist software art works to articulate the field in safely aesthetic terms, ignoring the more fundamental tension between instrumental software and reflective art that provides its genuine impetus. The three dilemmas that I have described are all in various ways dilemmas of reflection – of the reflective sense of formal or critical autonomy, of the aesthetic need to make code reflectively visible and of reflection as a recursive trap. The question of reflection lies at the heart of this thesis. I wonder about the viability of this relentless demand for reflection, the refusal to countenance the possibility that reflection may be better subsumed within a process of making rather than regarded as purely, autonomously constituted and all consuming. The following three chapters represent an attempt to sketch aspects of this alternative conception.

Chapter 5: Oblique Reflections: Software Art and the 3D Games Engine

Introduction

This chapter addresses the dilemma of position discussed in the previous chapter. It is concerned with how the field of software art conceives its relation to the industrial-technological infrastructure that surrounds and enables it. Specifically, how does it reflect upon the phenomenon of the 3D games engine? This chapter considers a range of tactical responses to the dilemmas of scale, encapsulation and conventional aesthetics that the game engine raises for software art. The main focus is on the strategy of anachronism. Anachronism resists the rhetoric of technological novelty, working instead to discover areas of creative purchase within the detritus of industrial (commercial gaming) progress.

Let us attempt to clarify some of the problems that commercial games technologies raise for experimental software art practice.

Scale and Complexity

The development of a cutting-edge commercial game depends upon huge financial investment and a large-scale, multi-tiered production process that involves work at the hardware, software and creative design levels. Despite the heroic myth of its cottage-industry genesis – stories of John Carmack and John Romero laboring away in relative isolation on *Wolfenstein 3D* (1992) and *Doom* (1993), hacking together the architecture and iconography of first-person game navigation through sheer force of geeky genius and popular cultural will – the 3D gaming engine is clearly the complex product of decades of military, scientific and computing/entertainment industry research into the visual and interactive possibilities of computer graphics. This is not to deny the visionary role Carmack and Romero played in developing specific technological solutions and, more generally, in linking emerging trends in computer graphics to the genre of visceral shoot-em-up gaming, but it is to insist that 3D gaming engines, like factories and telephone systems, are sophisticated industrial forms that resist efforts at individual authoring. Their sense of alienating technological scale and complexity presents strategic problems for a field such as experimental new media art where the ideology of individual (or local

level) creative control remains important. How can artists engage with the potential of technologies such as the 3D gaming engine when the scale of the technical apparatus radically exceeds the space of individual creative effort?

Encapsulation

As discussed in chapter 3, the notion of encapsulation within the field of object-oriented programming refers to the principle whereby code modules – specific areas of data and functionality – are protected from unwanted external modification through the creation of explicit public interfaces and a formal etiquette of access definition. At the general strategic level, encapsulation represents an effort to manage technological complexity, to enable systems to be pieced together in a modular fashion. Code modules are positioned as black boxes that take input and produce output while the details of implementation can safely be ignored. The whole conception of a gaming engine is based upon this principle. Contemporary game engines often encapsulate their functioning to such an extent that it is possible to develop original games without any dedicated work of coding at all. However, this raises a problem of aesthetic purchase. Is it acceptable to bracket the problem of the engine and focus exclusively on the dimension of alternative game content, or is a closer engagement with the underlying technology necessary? There is no single answer to this question. While some so-called 'art games' represent a fairly straightforward work of game 'mapping' (creating new levels and game art for an existing engine), software art takes a greater interest in the dimension of code. Software art is concerned with the experimental possibilities of code-based generic abstraction and spatial-interactive representation that the game engine represents. It wants to crack the game engine open and reinvent it. The encapsulated character of gaming technologies, the many layers of abstraction and hiding that enable their functioning are a source of both frustration and inspiration, suggesting all kinds of opportunities for experimental intervention and revelation (uncovering).

Conventional Aesthetics

Closely related to the issue of encapsulation is the sense that gaming technologies (particularly game engines) are not neutral entities. They encode specific aesthetic assumptions. For example, the emphasis upon perspective, back-face culling, naturalistic

shading algorithms and the like within 3D graphics engines reveals a clear orientation towards visual realism. While the tradition of avant-garde experimental art is suspicious of spatial illusion, commercial media (films and games) position perspective-based immersion as the essential axis of representational and interactive aesthetics. The gliding optical vector of the first-person shooter represents space as utterly seamless. None of this comes easily. The technical problem of stitching a three-dimensional game world together – of enabling smooth movement from one space to another and of managing the display of large complex spaces – is a vital one in 3D engine design. It typically involves the crafty use of portal systems so that the player constantly moves between partial worlds (from one rapidly loading data structure to another). The whole world, as such, never exists. The illusion of holistic space is a bubble with the player at its center. The bubble changes as the player moves about and everything else is darkness (or the barest map). This is, of course, conceptually very interesting, suggesting links to notions of the Cartesian subject, etc., but it is not explicitly highlighted within commercial games. It is represented as a technical problem rather than as a creative option worth exploring. There is a vital need then to engage with the mechanics of the engine to open up other aesthetic possibilities.

Tactical Responses

These problems mean that the commercial gaming engine is positioned awkwardly for software art practice. It is tempting (constituting an iconic popular form of virtual interaction and suggesting vital areas of cultural and aesthetic enquiry), but at the same time arcane, inaccessible and hidden. How have software art and the broader tradition of experimental new media dealt with this problem? How have they conceived and practically negotiated a relation to the technical means of production? Five strategies seem evident: **alliance** (artists and scientists working together); **abstraction** (artists determining a specific conceptual-aesthetic space independent from the necessity of technical engagement); **aggregation** (artists working together to match the scale of industrial production); **appropriation** (artists co-opting mainstream technologies as a mode of critical-deconstructive practice); and finally, and most relevantly for my purposes, **anachronism** (artists abandoning any claims to technological novelty and re-

working aspects of the technological heritage). Let us consider each of these strategies more closely.

Alliance

The first strategy involves an alliance between artists and computer scientists. In their famous 1825 article, 'The Artist, the Scientist, and the Industrial: Dialogue', the social philosophers Henri Saint-Simon and Leon Halevy suggest the possibility of a utopian accord between art, science and industry, in which these traditionally separate disciplines form an alliance to advance progressive societal interests (Saint-Simon, 1975). From a contemporary perspective, informed by the legacy of critical theory, post-structuralism and postmodernism, this early vision of the avant-garde is likely to appear naïve. We are less confident about the benign legacy of Enlightenment reason and very suspicious of the rhetoric of progress. Nonetheless, contemporary new media often summons up the rhetoric of a progressive alliance of artistic, scientific and industrial interests. The British collective Blast Theory provides an example of this approach (Blast Theory, 2004). Their augmented reality games such as *I Like Frank* (Adelaide Fringe Festival, 2004) explore the poetic relations between real and virtual spaces and players. Produced in collaboration with Nottingham Mixed Reality Lab, Blast Theory supplies the creative vision, while the Mixed Reality Lab researchers provide the cutting-edge technical infrastructure. Blast Theory is also involved in a larger research initiative, 'Integrated Project on Pervasive Gaming' (IPerG, n.d.) which links together a range of creative and scientific organizations with the aim of developing 'a radically new game form that extends gaming experiences out into the physical world' (IPerG, n.d.). This involves exploring 'new technologies to support the creation of new compelling forms of content' (IPerG, n.d.). It is worth noting that this strategy of alliance preserves a very traditional distinction between creative and technical contributions. Art focuses on the conceptual imaginative realm, while science focuses on the underlying engineering. This strategy maintains its critical aesthetic credibility by occurring at a slight remove from the realm of industrial, commercial application. Alliance is pursued in the guise of art-science collaboration, rather than as industrial R&D (research and development).

Abstraction

Like the previous strategy, abstraction accepts the conventional distinction between the field of creative design and technical implementation, but rather than entering into an alliance with the technological vanguard (whether conceived in scientific or industrial-commercial terms), it operates in isolation from them. Instead of portraying the possibility of an avant-garde that combines cultural-aesthetic and technological novelty, the sphere of the cultural-aesthetic becomes separated and abstracted from the technical. The focus shifts to the game concept as an abstract space that precedes any particular form of implementation and that can take shape experimentally without the machinery of cutting-edge gaming technology. This is the approach that Katie Salen and Eric Zimmerman (2004) adopt in their innovative account of the field of game design, *Rules of Play: Game Design Fundamentals*. They open up the potential for an alternative, creative and theoretical space of game design by deliberately bracketing issues of implementation. This has considerable value, especially as commercial gaming works within such an impoverished conceptual space, but also clearly represents a strategic withdrawal from the problems of scale and complexity that the technological dimension of contemporary gaming presents. So while strategies of abstraction risk devaluing the creative, imaginative dimension of technical implementation, they play a key role in delineating avenues of dedicated conceptual-aesthetic interest. The politically oriented web games of Gonzola Frasca provide an example of this approach. *September 12, A Toy World* (Frasca, 2003) employs the model of a simple isometric shooting game to make a critical point about the uselessness of addressing the 'war on terror' via missiles. The originality of this game lies not in its technical features, nor even in its mode of game play, but hinges instead upon a work of conceptual recontextualisation. It is an experiment in game-based political commentary. The technical and generic remain important as ironic points of reference, but the key creative work is abstracted from issues of implementation.

Aggregation

The sense of effective exclusion from the arena of cutting-edge technological development leads to another strategy: the formation of communities of cottage-industry

level producers who together build alternative game-related graphic engines and the like. The *Ogre* (n.d.), *Blender* (n.d.) and *Xith3D* (n.d.) communities provide examples of this approach. While certainly building sophisticated pieces of graphics technology, their collaborative work is not positioned as technologically cutting-edge. Instead the emphasis is upon access, upon providing means for small independent producers to engage with the esoteric and typically proprietary space of contemporary gaming technology. However, for my purposes, very few of these communities are oriented towards the sphere of experimental new media arts. Rather than questioning the aesthetic assumptions of commercial gaming technologies, they are more likely to provide ever so slightly pale copies. Undoubtedly the engines can be put to other uses, but any work of fundamental modification is likely to occur in a less collective context. Paradoxically, more relevant communities have a less explicit relation to the realm of gaming. The *Processing* community (Fry and Reas, 2001, continuing.), as we have seen in the previous chapter, focuses on providing artists with access to the creative space of programming, supplying technologies and a supportive context for the development of experimental projects that explore alternative possibilities for 3D rendering and the like. It supplies nothing like a game engine, because the focus is not upon games as such. The *Processing* environment is much more concerned with enabling artists to engage with code (and the aesthetic possibilities of code) at a more fundamental level. It deliberately strips away scale, complexity and encapsulation in order to establish a technical context in which genuinely creative questions can be posed. In its tactical effort to simplify, *Processing* has conceptual affinities with the previous strategy of abstraction. *Processing* aggregates precisely in order to enable individual, cottage-level experimental practice.

Appropriation

The popular practices of ‘modding’, ‘mapping’ and hacking bits of commercial game technology to produce alternative critical or whimsical pieces of new media art provide examples of appropriation. Gaming engines have been ‘appropriated’ to enable, among other things, abstract animation and drawing, data visualization, performance, political satire and film-making (*machinima*) (see the alternative games website, selectparks, n.d.). At times appropriation can represent a deliberate assault on proprietary game formats.

Cory Arcangel's (aka 8-bit Collective or Beige programming ensemble) *Super Mario Clouds* (2003) hacks into the hardware of the Nintendo game cartridge to strip away everything in the game but the floating background clouds. However, appropriation can also work in more agreeable harmony with the interests of commercial gaming. Many mainstream game developers allow and encourage efforts at creative modification and reconfiguration. They release source code and mapping tools to facilitate grass-roots production of new versions of an original game. They deliberately position their products as emergent cultural and technological phenomena. The interesting implication is that commercial games, as abstract engines and as generic fields of parametric possibility, may be said to logically anticipate all their various aesthetic appropriations. The game engine is a protean meta-level space of aesthetic potential that imaginatively encompasses all of its specific creative instances – even those that criticize and deconstruct it.

Anachronism

This strategy, specific to software art, engages closely with technology but in a distinct, deliberately 'out-of-time' critical-aesthetic manner. Excluded from – and avoiding – the rhetoric of technological novelty, anachronism tinkers, reflects, reconstructs and re-imagines aspects of the computational heritage. It overlaps to some extent with strategies of appropriation but places a greater emphasis on 'original' coding.

Sketching a cultural context for software art, Manovich (2006) suggests that whereas the postmodern media artist (of the 1970s-1990s) engages in a work of pastiche and appropriation, the software artist (of the late 1990s and early 21st century) insists upon a creative *tabula rasa*. The emphasis shifts to coding things from scratch, avoiding both the tools and illusory mimetic rhetoric of contemporary commercial new media (animation, games, etc.). According to Manovich, this represents a return to an earlier model of artistic practice – the model of the 'romantic/modernist' genius. Instead of drawing cynically upon the available media culture, iconography and creative tools (with the sense that there is no viable aesthetic space beyond), the software artist 'makes his/her mark on the world by writing the original code.' He stresses that '[t]his act of

code writing itself is very important, regardless of what this code actually does at the end' (Manovich, 2006: 211).

While this work of writing is clearly very significant, I am not convinced that it summons a pure terrain of original expression. Indeed the small qualification that Manovich makes – the acknowledgement that this code may be inconsequential, that it may not do anything especially significant or novel – suggests a tension and uncertainty surrounding the nature of 'original coding'. The blank sheet of code is not a simple surface. It is both a veil and an unveiling. It is both clean (creatively open) and thoroughly inscribed. It floats above a framework of encapsulated processes that extend down to the hardware level and is structured as a palimpsest, in which the software artist repeats, writes and interrogates the coding tradition. The software artist makes his/her 'original marks' in the space that is left once technological progress has moved on. Originality lies in summoning up a dimension of alterity within this abandoned landscape, discovering through self-conscious anachronism ('non-original' coding) a field of aesthetic possibility. Whereas the direction of commercial technological development is to develop more and more sophisticated layers of abstraction that work to make human engagement with computer processes as intuitive and kinaesthetically engaging as possible, software art deliberately returns to the earlier, retro model of arcane text-based interaction. As I argue in chapter 4, the GUI (graphic user interface) – and the dream of the GUI – disappears to be replaced by the IDE (integrated development environment) and the text console. Software art partakes of anachronism in its very concern to structure human-computer interaction in terms of the traditional metaphorics of programming.

The strategy of anachronism then engages creatively with the technological tradition by deliberately withdrawing from any attempt to appear at the cutting-edge. The aim is less to project an unseen future than to re-imagine and re-invent the computational wheel, to work over the detritus of technological progress searching for points of creative intervention. Anachronism acknowledges the asymmetry between art and the space of technological development, but insistently searches for means to reflect upon the technical, to open it up to a process of critical-creative enquiry. In the process, the

relation to gaming technologies often becomes indirect. While there are many artists producing alternative games, there are significantly more engaged in formal experimentation with aspects of 3D drawing and rendering. This genre of creative practice (very prominent in the *Processing* community) represents a response, at least partly, to the conventional illusionistic assumptions that inform the structure of the commercial game engine.

I have described these strategies separately, however it needs to be acknowledged that they very often communicate and overlap. For example, although abstraction, as I have defined it, resists engaging with the sphere of technical implementation, it has vital importance in terms of describing a dedicated space for conceptual-aesthetic reflection. Anachronism, at its best, incorporates a dimension of abstraction; it distills the conceptual-aesthetic relevance of specific technical processes rather than simply reconstructing them. Similarly, strategies of appropriation can often blend into strategies of 'original authoring' (anachronistic re-invention). Appropriation finds itself opening on to an original space while attempts to code from scratch discover a relation to the legacy of coding achievement. Even alliance can reveal other dimensions. The Blast Theory augmented reality projects, despite the rhetoric of avant-garde interdisciplinary accord and technological novelty, represent an impressive effort to re-orient aspects of standard industrial R&D, to gently appropriate science towards an investigation of critical-poetic issues related to virtual identity and emplacement.

The five strategies represent less a static set of antagonistic options than a dialectical constellation. Together they constitute a field of productive tension. The key tension concerns how the relation between art and *techne* is conceived, but there are also more subtle tensions concerning issues of originality and the tactical relation of art to the broader sphere of technological progress.

Software Art Works

With this general scheme in place, let us consider how software art reflects on the 3D game engine. My interest is in how the various tensions that I have described above are played out within a specific new media arts context and in the texture of specific software

art works. My initial focus is on two exemplary projects of appropriation (resource hacking) – JODI's *SOD* (1998) and *Untitled Game* (2002). These visionary works reconfigure the *Doom* and *Quake* engines and anticipate vital paths of investigation for contemporary software art. If *SOD* and *Untitled Game* address the 3D games engine directly, the relation is more oblique within contemporary software art. A consideration of one of my own recent projects will provide a means of clarifying the nature of this relation.

JODI Game Modifications

JODI is the Dutch-Belgian duo of Joan Heemskirk and Dirk Paesmans. Their modifications of the *Doom* and *Quake* engines are sublimely deconstructive reflections on the formal architecture of the first-person shooter. Although their work involves code-based intervention, it is clearly not software art that begins with a blank page (the imaginary, theatrical scene of a blank page). It is work that explicitly highlights the slippage between postmodern strategies of appropriation and (undecideable) strategies of 'original authoring'.

SOD

SOD hacks the *Doom* engine to represent the grim corridors of the original game as abstract black and white shapes. Stripping away the illusion of figurative, textured, shaded space and maintaining only minimal perspectival cues, *SOD* highlights the underlying architecture and artificiality of first-person space. Structural features such as the option screens, HUD (heads up display), portals and targeting system gain a new and uncanny visibility. Whereas in ordinary game play, these features support the game play and remain subservient to it, here they are foregrounded through deliberate strategies of abstraction. Option screens become lists of semantically void geometric shapes. The HUD displays numerical information about a game space that we can only marginally engage with. Doors (portals) float in space, manifesting forms of transition that undermine any naturalistic conception of a doorway and that very evidently involve the sudden loading of new spatial data. The only element that remains largely unchanged from the original game is the sound; it provides a residual sense of spatial integrity and

indicates that despite the obvious work of modification we remain in the *Doom* engine space.

Untitled Game

JODI's deconstructive strategies become even more radical in *Untitled Game*. Working now with the *Quake* engine, *Untitled Game* is a set of fourteen game variants that explore the coded-ness (or metaphysics) of 3D games. Gone, on the whole, is any lingering concern with maintaining aspects of three-dimensionality. The focus is on the pre-space of conceptual abstraction that shapes the underlying possibility of perceptible game space. This is evident in the title itself which playfully employs the archetypal name within abstract art ('untitled').

The names of the individual game variants are also worth considering. Many indicate what appear to be logical ranges within the alphabet – *A-X*, *G-R*, *M-W* – however, the ranges clearly overlap and bear no relation to the content of each game. Other games are named after command key combinations – *Ctrl-9*, *Ctrl-F6* and *Ctrl-Space* – however the games make no apparent use of these combinations. Only three of the games are named in a more ordinary descriptive manner (*Arena*, *Slipgate* and *Spawn*). The generally arbitrary character of the names makes the gulf between the sphere of language and reference (engine processes and 'game play') very explicit. The names are indicative of the central critical, deconstructive concern with the disjunction between the spheres of coded representation and spatial perception.

Turning now to a brief analysis of three of the game variants:

Arena

Arena (Figure 13) represents a sublime near-zero point of the *Quake* engine. There is the sound of attacking enemies and the player can click and fire, but 3D space itself has been altogether eliminated, leaving only a framed white screen and the HUD. This variant points to the non-space at the heart of 3D simulation and stages it literally, visually.

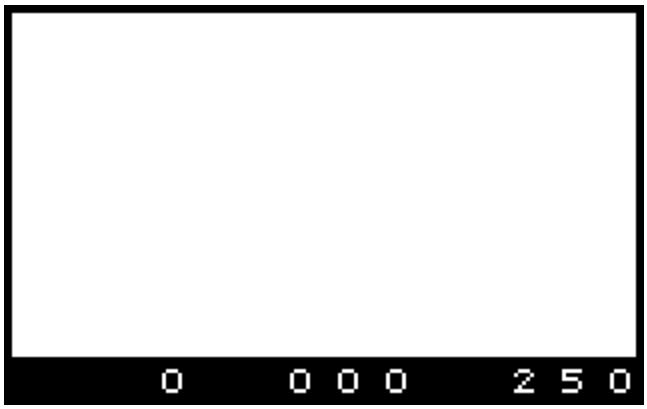

Fig. 13: JODI, *Untitled Game – Arena*

A-X

A-X (Figure 14) dispenses with even more features of the original game. There is no longer even the frame or the HUD. Players encounters a cascade of data; they encounter 3D space as the engine (at some relatively high-level) conceives and processes it. This is a particularly clear example of the critical focus on the discontinuity between code and the illusion of space.

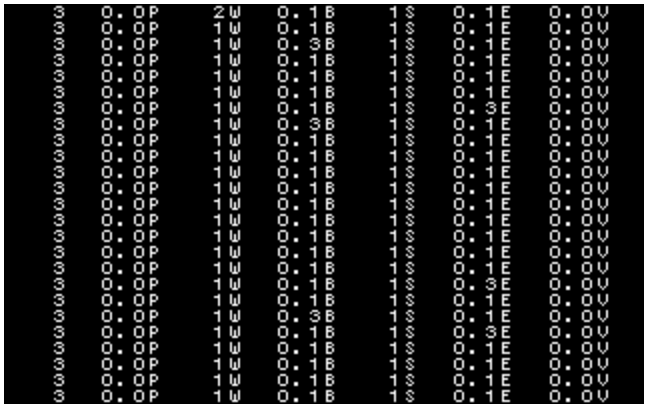

Fig. 14: JODI, *Untitled Game – A-X*

Q-L

Q-L includes recognizable aspects of *Quake* 3D space, but in a totally vertiginous manner. Instead of predictable visual orientation and motion, the camera spins wildly out of control and none of the usual interactive controls work as expected. This game variant unsettles the sense of continuous 3D space and confident first-person motion through

space. It suggests that spatial continuity and first-person interaction are only a fragile mathematical fiction.

The exit screen for *Q-L* (Figure 15) indicates a characteristic deconstructive strategy employed in *Untitled Game*. Instead of the usual set of options - 'New Game', 'Single Player', 'Exit', etc. – the user encounters a jumbled set of letters that are semantically meaningless but that adhere to the formal layout of the ordinary options screen. Anyone with experience of *Quake* can infer that 'PTJS' signals 'EXIT' because it is four letters long and positioned where 'EXIT' would normally be.

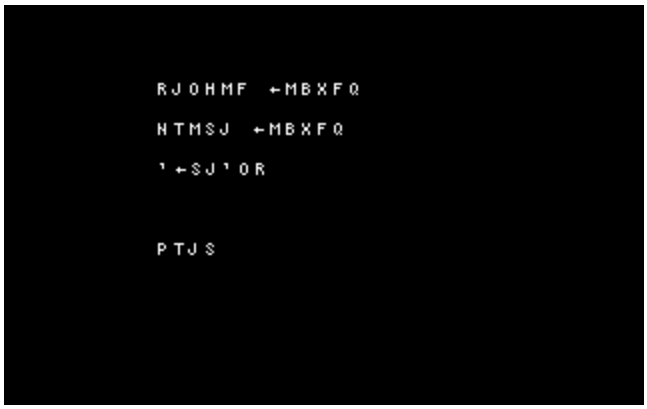

Fig. 15: JODI, *Untitled Game – Q-L*

This strategy of maintaining formal identity while simultaneously engaging in a work of semantic scrambling is a key characteristic of the JODI modifications. However much they deconstruct the *Doom* and *Quake* engines, they also remain true to aspects of their underlying structure. JODI's work obtains its critical imaginative (and political) force precisely inasmuch as it develops a tension between the formal mechanics (and cultural imaginary) of game engines and a more open space of creative possibility. Their work represents a limit form of game modification. While remaining within the orbit of *Doom* and *Quake*, they signal concerns that extend beyond the field of 3D games as such. The *Untitled Game* variants, for example, are very clearly not games. They are unplayable critical interventions that focus upon the underlying logic of spatial representation – upon the engine as a conceptual system rather than the structure and articulation of game-play. As much as they play upon the formal features and iconography of *Quake*, they also manifest a fundamental and more general concern with the abstract logic of code.

Working at the limits of appropriation, they indicate the necessity of other approaches; ones that not only modify engines but also imagine them differently. In this sense the JODI modifications anticipate contemporary strands within software art that focus on the creation of alternative, typically abstract, graphic engines.

Contemporary Experiments

Trawling through the *Processing* forums it is possible to find many examples of posts such as the following:

> Hello, As my brain is starting to smoke, and google cant seem to give me an understandable answer, turn to you for a possible solution. my problem is as follows: I have a point defined in polar coordinates (Zrotation, Yrotation and distance), now i need to find out what the absolute rotation is, by that i mean a single rotation around a defined axis that returns the same point in space. I've been speculating that this angle is sqrt(Zrotation^2 + Yrotation^2) but i havent been able to verify this. Is there anyone out here that have a better understanding of this than me? also i would like to know how to calculate the axis of this rotation. hope my explanation is understandable: Regards, Henrik, IP Logged (Fry and Reas, 2001, continuing))

Along with Henrik, many software artists are increasingly absorbed in the technical intricacies of 3D graphics. A huge amount of energy is devoted to solving entirely trivial mathematical and programming problems, ones that have been solved innumerable times in the past (and much better). To code with this sense of larger irrelevance, with this awareness of stupidity and anachronism – this is a substantial part of what it means to be a contemporary software artist. Typically there is no explicit aesthetic rationale; ostensibly it is just about tinkering around with the mathematical and technical infrastructure of 3D graphics. However, this may actually indicate the key point; this process of tinkering is about bringing the problem of technological complexity down to a human scale. Each specific technical problem is tantalizingly soluble at a local, human level. In this sense, each question projects a horizon of distant but attainable technical competence. The legible hope is that artists can obtain the relevant skills and

understanding to direct code in their own aesthetic interests rather than inevitably be swept along by existing technical regimes and aesthetic assumptions.

The technological infrastructure of the 3D games engine provides a crucial reference point for this work of contemporary experimental practice. The explosion of sophisticated graphics technologies over the past two decades is closely linked to the increasing economic and cultural sway of real-time 3D games. OpenGL, DirectX, superbly fast graphics cards – all bear the imprint of the commercial gaming industry. If software artists now have access to aspects of this technology it is substantially due to its popularization within games. The JOGL (Java bindings to Open GL) API, which is vital to *Processing* and more general Java-based software arts 3D experimentation, provides a clear example of this debt. Although the API can certainly be applied in contexts that extend beyond games, it is nonetheless the product of the Game Technology Group at Sun Microsystems and can be found at the java.net site by following the following set of hierarchical links; projects, games, games-core, jogl (JOGL, n.d.). While software art makes use of this technological infrastructure, gaming itself, as a cultural form, is often only obliquely acknowledged. Whereas the JODI projects explicitly confront the culture and aesthetics of commercial gaming, software art imagines a fragile space of autonomy. Gaming is positioned, in a contradictory fashion, as both a necessary foundation and an extraneous imposition.

Anachronism (Again)

To illustrate this tension within the self-identity of software art, its sense of relation and non-relation to the larger technological and cultural infrastructure, I will consider the development of one of my own recent software art projects. This project is entitled *Anachronism* in order to highlight the awkward relation to the means of production that is constitutive of software art. I am focusing on my own project here not with a sense of its aesthetic importance but because I can provide an under-the-hood explanation of how the work is informed by a relation to the 3D games engine.

Anachronism began as a kind of perverse Java 2D sketching program. The idea was to eliminate all sense of an analogue relation to manual drawing. The user would quite

literally draw with numbers; defining shapes by writing a series of x and y coordinates (and control points for Bezier curves) to a text file. An additional configuration file would describe sprites, motion and rendering styles. It is worth noting that although none of this engaged closely with the possibility of the 3D games engine, many of the fundamental concepts informing the structure of this experimental drawing program have their basis in gaming technologies. The whole conception of a 'sprite' as a screen instance of a graphic data structure stems from gaming, initially as an aspect of graphics hardware and then as a software abstraction.

Having created a version of this initial – deliberately contrary – drawing program, I became more interested in the creative possibilities of code drawing itself, and especially in the potential to draw with animated 3D shapes. It quickly became evident that it was impossibly slow and difficult to manually define 3D shapes, so I switched to the algorithmic definition of simple shapes and the parsing and loading of Alias Wavefront 'obj' files. The latter represent shapes as lists of vertices and polygonal faces and can be created in a variety of 3D modeling applications. My aesthetic rationale was to explore alternative, non-figurative means of 3D rendering. This represents a characteristic gesture of resistance to the predominant focus upon visual realism within commercial games and animation. It follows the trajectories suggested by the JODI projects, but would seem to articulate them in a less politically pointed and deconstructive manner. If my first concept playfully juxtaposes code and the ideology of intuitive aesthetic perception, my second encapsulates the dimension of code drawing in order to elaborate a wider space of visual possibility.

However there are also more subtle implications. The shift to 3D prompted a more explicit concern with the graphic-related structure of the 3D games engine. My interest was in stripping back the graphic operations to a bare minimum. There would be no back-face culling, no painter's algorithms, no binary partition trees, it would simply be sets of polygonal objects that could be animated and drawn as points, lines or filled shapes. I actually avoided OpenGL (JOGL) and worked with simple Java 2D drawing methods. This deliberate work of bracketing core areas of functionality was the key to opening up original creative possibilities. Suddenly in the interstices of the conventional

engine (here re-written from scratch) there was the potential to explore something other than the simulation of space; something that entered into to dialogue with traditional drawing, that was concerned with the deliberate fashioning of shapes, iterative patterns and conceptual series (Figure 16).

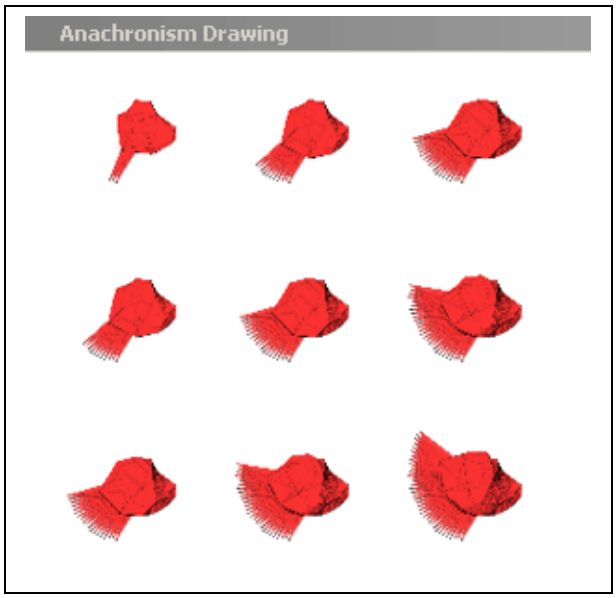

Fig. 16: Brogan Bunt, *Anachronism* (2006)

The creative work emerges then in the friction between the conventions of the 3D graphics engine and the experimental agendas of software art. *Anachronism* is interesting precisely in terms of the tensions that structure its autonomy and originality. However, there is the difficulty that this underlying dynamic may not be directly evident in the work. It figures as a background and is articulated obliquely. Perhaps the title makes the point, but the question remains a vital one for software art: how can the concepts and contextual constellations that inform the creative work of programming become lucid at the level of the perceptible work?

Conclusion

This chapter has attempted to describe the awkward relation between contemporary software art and the 3D games engine. It has considered the broad dilemmas of scale, encapsulation and conventional aesthetics that the 3D games engine presents as well as suggesting a range of specific strategic responses. My key interest has been in the

ambivalent character of strategies of anachronism. Anachronism appears both original and non-original. It both imagines prospects of creative autonomy and acknowledges relations of dependence and dialectical differentiation. If there is a problem in all of this, it is less in terms of issues of logical contradiction than in the all too common failure to tease out the conceptual implications of anachronistic practice. There is a crucial need for the critical character of 'technological tinkering' to be elaborated within software art. Processes of interrogation that may be apparent to the programmer need not be apparent to the user/viewer. If experimental graphical software art is to avoid being interpreted as apolitical and blandly decorative, then it needs to discover ways to articulate underlying conceptual concerns (and the politics of its problematic creative positioning) more explicitly.

Chapter 6: Software Art and the Instrumental

Introduction

The second dilemma of software art (see chapter 4) relates to the demand that code become visible, that it abandon its 'natural' tendencies to hide and proceed silently – that it become instead reflectively manifest. However, code insistently, structurally withdraws as part of its overall instrumental orientation. In resisting the invisibility of code, software art resists the instrumental character of conventional software. My aim in this chapter is to question this rejection of the instrumental – to suggest an intimate relation between the aesthetics of software and its functional character.

This issue takes shape for me in relation to the uncertainty of one of my own works. I describe it as a software art work, but with some hesitation. The work lacks an adequate aesthetic manifestation – either as code or as visible interface. It is a set of tools and an engine. It is concerned with the representation of time and the pragmatics of enabling a temporal display. The title of the work is *Cropper_Propper_Gridder*. If the work is of any interest, it is because it pursues a poetic idea through instrumental means – or better, it struggles to discover a potential for poetry in the aesthetic estrangement of software. While software art conventionally resists the instrumental character of software – struggling to make software aesthetically, reflectively appear – *Cropper_Propper_Gridder* deliberately engages with the aesthetic blindness of instrumental functioning.

The Problem of the Instrumental

Within the tradition of critical theory, the notion of the instrumental is associated with a specifically modern mode of rationality that is oriented towards the purposive accomplishment of tasks, in the process deliberately bracketing questions of human value. Instrumental rationality addresses issues of efficiency and running, ignoring wider ethical, political and cultural concerns. The sociologist Max Weber (1946) argues that this mode of reason takes characteristic form in the mechanisms of modern bureaucratic administration and industrial capitalism. This broadly social conception of the instrumental is predicated on a more fundamental notion of the nature of an instrument.

An instrument is a device that moves but lacks free being. It produces results but without any awareness of cause or result. It functions unreflectively. It proceeds blindly. In this sense, despite its status as a technical contrivance, an instrument – in its motion, in its running – comes to resemble the deterministic processes of nature. At the very outset of his discussion of art in his 1790 *Critique of Judgement*, Immanuel Kant explains that '*Art* is distinguished from *nature* as making (*facere*) is from acting or operating in general (*agere*); and the product of the result of the former is distinguished from the latter as work (*opus*) from operation (*effectus*)' (Kant, 1980: 523). In the same manner, an instrument can be regarded as performing operations which produce effects rather than performing actions which shape (aesthetic) works. This indicates the obvious dilemmas that confront any attempt to chart an association between the instrumental and the aesthetic. Conceived as intermediary, mechanical and unreflective, the instrumental appears directly opposed to the finality, freedom and reflective nature of art.

How, in this context, can software, as a discursive space that is substantially shaped by the logic of the factory assembly line (Gere, 2002: 17-46; Manovich, 2005: 5), and that is centrally concerned with issues of abstraction, procedure and function, possibly be aesthetic? How is software art to conceive its relation to the instrumental dimension of software? As a basis for addressing these questions, it is worth briefly sketching a more general context of debate concerning the relation between the instrumental and the aesthetic.

Art and Engineering

In 1920 the Russian Constructivist artist, Vladimir Tatlin, produced a proposal for the Monument to the Third International (Tatlin, 1920). He envisaged a 400 metre high steel and glass tower that incorporated a dynamic spiral structure and rotating internal rooms. It was utopian art adopting the guise of an architectural plan and was criticized for its impracticality by revolutionary artists and politicians alike. Another Constructivist artist, Gabo, cautioned Tatlin to 'either create functional houses and bridges or create pure art, not both' (*Wikipedia*, n.d.). The work was condemned for confusing two distinct languages and modes of making: art and engineering. Furthermore it transgressed the conventional boundaries between the

aesthetic 'end-in-itself' and the sphere of useful things. These 'flaws' are also the basis for its lasting significance as an icon of avant-garde art. The monument posed the essential problem concerning art's relation to modern forms of making and, more generally, art's relation to industrial modernity.

Two broad historical strategies emerge in relation to this challenge. On the one side there is the model supplied by Dada of incorporating technology and technological forms of making as a means of waging a multi-pronged assault on autonomous art, bourgeois humanism and instrumental rationality. This approach takes archetypal form in Marcel Duchamp's *Large Glass – The Bride Stripped Bare by Her Bachelors, Even* (1915-23), which provides an ironic take on our pleasant fictions of love, free will and organic, human difference by representing human courtship and erotic coupling in mechanical terms. In a parodic reference to the history of industrial machinery, the processes proceed upwards from steam, to internal combustion engine, to electricity (Duchamp, 1973: 39). This is not, of course, a working machine. It is a playful, subversive, metaphorical apparatus. It functions as a piece of critical commentary rather than as a literal instrumental device. A crucial distance then is maintained between art and engineering so that art, however fractured, however affected by industrial modernity, can shape a properly aesthetic space of critique.

The other strategy, evident especially in Constructivism and the Bauhaus, strives towards a unity of art and industry. It projects an integration of the aesthetic (as mode of formal appearance) and the instrumental (as sphere of functional, mass-produced products). Art abandons its reflective autonomy to enter into the texture of practical things. While crucial as a critique both of art and alienated labour (Burger, 1984; Huyssen, 1986: 12), this strategy runs the risk of providing an aesthetic sheen for forces that actually undermine the potential of art to suggest alternative social and imaginative possibilities. Furthermore, this effort to draw a close association between the aesthetic and the instrumental is much easier to manage with simple, everyday things – coffee cups, tables, light fittings, etc. – in which form and function share a common immanent material being. Software

programming is harder to conceive in these terms because it institutes a separation between the domain of instrumental instructions and the visible interface. The former indicates a space of symbolic abstraction and functioning that is hidden from view – that is not instantly coextensive with the terrain of user interaction. Although authors such as Donald Knuth (1973 – 1998) portray programming as a practical art which can be regarded aesthetically in terms of values such as economy and elegance, this makes the aesthetics of code only accessible to programmers and represents a return, as Cramer argues, to a very traditional neo-classical aesthetic space.

In short, neither critical nor integrative strategies genuinely engage with the instrumental in its non-aesthetic distance. Critical avant-garde art resists literal instrumental functioning while modernist design works to aestheticize the functional. Neither provides an adequate means of conceiving the field of software programming, which refuses to adopt a conventional aesthetic form, which is directed elsewhere, which shapes instructions rather than an easily critical or conciliatory work. If there is anything unique about the situation of software art it lies precisely in this search for an aesthetic rationale without the possibility of any recourse to the non-instrumental or the consolation of immanent form.

A Problematic Definition

The jury for the 2002 Moscow Read_Me 1.2 festival offer an influential definition of software art:

> We consider software art to be art whose material is algorithmic instruction code and/or which addresses cultural concepts of software. (Read_Me 1.2 festival jury, 2002)

Although intended to be inclusive, this definition works to obscure the key issue of the relation to the instrumental. Instead it focuses on distinguishing two strands of software art practice – formally oriented code-based experimentation and culturally oriented software critique. The formalist option is expressed in terms that recall the language of

high modernism; the focus is upon defining the material essence of the software medium, which here takes the form of 'algorithmic instruction code.' In this manner, a complex cultural assemblage – a language and a field of discourse – is reduced to the status of a simple material, like paint or clay. This reduction of code to the simplicity of an aesthetically malleable material is what enables formalist software art to be represented as a purely conceptual meditation on aspects of system without any integral concern with dimensions of culture. However, a close engagement with the medium of code can have other implications. It can have a cultural dimension. It can represent an engagement with a specifically culturally determined discursive space. More particularly, it can represent an interrogation of the instrumental language and strategies of conventional software. But unfortunately, by positioning code as a base aesthetic matter, formalism loses sight of this possibility. It is left to the other side of the definition to engage with software as a cultural phenomenon.

But correspondingly, although the culturalist option 'addresses cultural concepts of software' it seems to lack a specific point of discursive purchase. How is the nature of this mode of address to be described? Is this critique spoken in the language of code as actual functioning software or is it expressed in other terms? There is a need to explain how critical software art relates to the layer of instrumental, non-reflective language that provides the basis for its operations. There is a need to think through the engagement with the material language of code. In this sense, the cultural critique of software cannot be conceived apart from the apparently formalist option. The distinction between formalist and cultural tendencies obscures this vital issue.

If this bifurcated notion of software art is ultimately disabling, working to impoverish both formal experimentation and cultural critique, it is because it misconceives the field in terms of a tension between contrasting aesthetic tendencies rather than in terms of a more constitutive tension between art and the non-aesthetic, instrumental dimension of software.

Software Becoming Art

The notion of software art appears at one level as a transgression of ordinary aesthetic

proprieties. In a traditional avant-garde spirit, it seems to unsettle the complacent autonomy of art, insisting that art engage with a space of non-art – a realm of engineering and technical implementation. Yet at another level it proceeds in an opposite fashion. Rather than genuinely risking a relation to the alterity of another cultural and discursive space, it conceives software in terms of art. It dialectically subsumes those aspects of software that are aesthetically useful and digestible, while discarding everything else. This is evident inasmuch as the specific characteristics of software art correspond to a very conventional aesthetic scheme. It is worth briefly outlining the contours of this scheme in terms of Kant's classical model of aesthetics and fine art.

According to Kant (1980), aesthetics denotes a realm of non-instrumental engagement with things. It is a sensuously enabled mode of reflective judgement that rises above the dimension of sense to enter into dialogue with the *a priori* space of conceptual understanding (Eagleton, 1990: 85; Kant, 1980: 484). The experience of beauty, for instance, relates to the recognition of order in the symmetrical forms of nature – mineral and organic forms that are not themselves conceptual but that nonetheless reveal a systematic, formal logic (pattern, unity and harmony) – an order that is apprehended through the senses but that instantly summons an awareness of the universal and the metaphysical (Kant, 1980: 493). Fine art, as a specific experience of the beautiful, manifests a purposiveness without purpose, a disinterested, non-utilitarian demonstration of the felt rightness of the conceptual (Kant, 1980: 524-5). It strips real objects of their ordinary reality, their contextual significance as objects that are practically desired, manipulated and used. Art objects suspend the dimension of conventional instrumental utility in order to attain a higher conceptual utility as signs of an ultimate reconciliation of human faculties. Their lack of instrumental utility takes the form of an organic finality, a dimension of formal coherence without goal. The production of art depends upon genius; an 'innate mental aptitude (*ingenium*) through which nature gives the rule to art' (Kant, 1980: 525). Unlike instrumental craft, which is the product of practical, formulaic labour, fine art is conceived as a generative expression of the soul as a protean 'second nature' (Kant, 1980: 528). Kant's aesthetic scheme is representative of a

classical Enlightenment conception of art as non-instrumental, final, reflective and the product of genius.

Now, without trying to suggest that contemporary conceptions of software art are strictly-speaking Kantian, there are curious affinities linked to how issues of instrumental function, reflection and artistic subjectivity are conceived.

Instrumental Function

Software art characteristically resists the notion of software as a tool. The jury for the 2001 Berlin *Transmediale*.01 artistic software award suggest that '[e]very program that pretends to be a tool disguises itself'. Here they are referring to the fiction of a passive tool – of something that is altogether controlled by human beings and subservient to their interests, but it spills over into a more general rejection of pragmatic, instrumental software. There is a strong preference for work that undermines utility and suspends ordinary functioning. Adrien Ward's *Signwave Auto-Illustrator* (Ward, 2001) provides the iconic example, although it is a work that represents, in my view, an ambivalent relation to the instrumental. In its adherence to the interface conventions of commercial creative software, *Auto-Illustrator* at once deconstructs and delights in the notion of software as tool. While the deconstructive orientation is emphasized, the manner in which the work draws inspiration from the conventional language of tool-based software escapes explicit attention.

Software art's suspicion of tools connects to the classical aesthetic bracketing of the instrumental, although clearly the aim is less to determine a pure space of disinterested perception than to critically respond to the dominant models of commercial application software. Yet it seems to me that the rejection of the notion of the tool – as well as the rejection of the tool's effort to disguise itself – creates fundamental problems for software art. Even if a piece of software is not ostensibly a tool, it must speak the language of tools. It is devised as a system, an apparatus. It functions. As languages and discursive forms, programming languages bear the necessary imprint of the industrial forces that have shaped them. The concept of a tool is implicit within programming structure – in the notion of an algorithm that processes data, an object that performs a specific

(encapsulated) task and a procedure that runs more or less efficiently. In bracketing all of this, in trying to think algorithm and procedure beyond the instrumental space of tools and tool functioning, software abandons a crucial point of aesthetic purchase. The goal in my view is not to resist the notion of the tool, but to engage with issues of abstraction, disguise and efficiency, to somehow re-imagine the aesthetic in an alien terrain.

There are already models from within software – works that may not be primarily aesthetically constituted but that have aesthetic, poetic implications, that reveal the potential for an instrumental imaginary. Just to briefly mention three: Ivan Sutherland's 1962 *Sketchpad*, which was not only the first graphic drawing program but which, more particularly, as Allen Kay argues (2003), re-invents drawing in terms of the conceptual structures of object-oriented programming; Richard Stallman's *Emacs* (1975) which is a bizarre jalopy-style software, defiantly resisting task specialisation and ordinary boundaries between work and play; and finally even the modern integrated development environment, *Eclipse* (2004), which is utterly generically conceived – which can be radically reconfigured to accomplish different programming tasks and which appears as a kind of meta-tool, a tool for creating tools. For me these software tools are as much a source of inspiration as is work that is specifically (safely, neatly, clearly) positioned as software art.

Reflection

The primary motivation of software art is to encourage reflection upon underlying programmatic software processes. This notion of reflection is hardly the affirmative, grandly reconciling reflection of Kantian aesthetics – it is often, for instance, critical and deconstructive, but it nonetheless privileges code that does more than simply operate – that somehow finds the means to reflect upon its own operations. Without wishing to altogether question this orientation towards reflection, it seems to me that the issue is more complex. Programming entails relations that extend beyond the fantasy of visibility and self-collected reflection.

This is evident at the very outset of modern computer science in Alan Turing's model of computation (Turing, 1995; Feynman, 1996). If Turing chooses to compute, it is because

computation is mechanical, it proceeds stupidly step by step. Computer programs may represent brilliant efforts of reflective analysis, abstraction and design, but program operations at the atomic level of specific digital events are utterly simple and unambiguous. Reflection constitutes a problem for underlying digital processes, a quandary that suspends their functioning. It is worth examining the play of reflection and machine unconsciousness in Turing's famous 'halting problem'. Turing reflects upon the mechanism of computation, upon its procedural logic. He sets computation a reflective trap. The universal machine is programmed to halt if it is stuck and proceed if it is not. Then, in a crucial reflective step, Turing makes the computer process its own code. Now, it seems, it must halt if it proceeds and proceed if it halts. Unable to decide whether to proceed or to halt, the mechanism comes undone precisely through a motion of reflection.

The jury for the *Transmediale*.01 festival suggests that the fascination of computer programming depends precisely upon code's capacity to function, the passage it makes from a reflective conceptual state to one of actual machine processing:

> Perhaps the most fascinating aspect of computing is that code – whether displayed as text or as binary numbers – can be machine executable, that an innocuous piece of writing may upset, reprogram, crash the system. (*Transmediale*.01 Media Arts festival jury, 2001)

Turing's example suggests that this necessitates a relation of reflection to something other than reflection – to a space of blind motion that functions only on condition that it does not reflect. Programming demands a close engagement with this other space. It opens up a vital relation to the blindness of machine processing. The aesthetics of code is as much about the unseen, the hidden and the disguised as it is about the reflective and the visible. In this context, strategies of abstraction and encapsulation are also relevant – as indeed are all of the strategies that structure programming as a work of inscribing layers and guises above an unreflective foundation. So while software art expresses a fascination with the executable character of code, it withdraws from the thinking of this space to the extent that it insists upon a purely reflective conception of software art.

Artistic Subjectivity

Software art associates the aesthetic character of code with a dimension of personal inflection. Cramer and Gabriel argue that:

> [C]oding is a highly personal activity. Code can be diaries, poetic, obscure, ironic or disruptive, defunct or impossible, it can simulate and disguise, it has rhetoric and style, it can be an attitude. (Cramer and Gabriel, 2001: 3)

This is hardly the concept of aesthetic genius (which is actually much more ambiguous, which actually deeply problematizes issues of agency) but it places a similar emphasis upon the expressive potential of code. Code that is impersonal and formulaic appears less aesthetic. In my view, however, code is inevitably formulaic. There are all kinds of standard idioms, patterns and stylistic conventions. It is less by resisting these and affirming some notion of personal, differentiated expression that code becomes aesthetic, than by pursuing the formulaic closely and intimately. Rather than asserting subjectivity, it is a matter of finding it elsewhere, of re-inscribing it at a distance. Code is only personally inflected within the texture and through the agency of impersonal formula.

There is a vital need then to consider the instrumental character of software beyond the conventional framework of Enlightenment aesthetics. As Derrida (1976) argues, the tool is never a mere subservient vessel but always appears as a force that intimately affects and undermines the notion of human agency. Writing appears as an aid to human memory but actually destabilizes human memory and renders it in other, alien terms. Re-conceiving the instrumental character of the software tool depends upon considering the nature of a tool more closely, rather than turning away with a sense of traditional aesthetic disdain.

Heidegger – Technological Revealing

In his famous 1953 article, 'On the Question Concerning Technology', Heidegger begins by suggesting that 'the essence of technology is nothing technological' (Heidegger, 1978: 287). He is determined to reinterpret technology, to discover within it another meaning. Heidegger questions the common sense view of technology as a neutral means to an end

and as an expression of human agency. He describes this view as 'the instrumental and anthropological definition of technology' (Heidegger, 1978: 288). This would seem to represent a similar rejection of human instrumental agency that we find within software art, yet the notion of the instrumental makes a strange return as the argument proceeds – a return in which the notion of a subservient means is thought apart from the necessity of original agency or determined end.

Re-examining the nature of technological making as traditionally conceived (in the Aristotelian conception of *techne* (Aristotle, c. 350BC)), Heidegger finds that it involves a motion of "bringing-forth" that is aligned with *poesis* (Heidegger, 1978: 293). It also summons a more complex sense of causality which eludes the modern sense of means-end rationality and engages with processes of revealing – the manifestation of truth (Heidegger, 1978: 294). A classic instance is evident, perhaps, in Michelangelo's conception of uncovering figures in marble; he less makes the figure (*ex nihilo*) than releases and reveals the inherent potential of the figure from within the marble. In this sense the artist lacks absolute agency, appearing instead as a mechanism for an overall process of revealing (he is caught up in the mystery of Being).

Heidegger argues that while this model appears applicable to traditional handicraft, modern technology radically changes things. Rather than adapting to implicit nature – tending it and gently bringing it forth – modern technology exploits materials; it extracts from them and transforms them. Materials become bare functional resources that are never revealed as such but that are instead stored up, ordered and operationalized (Heidegger, 1978: 298). Traditional processes permit the object its distinct appearance, autonomy and finality, whereas modern modes of technological manufacturing enable no space of rest or of contemplative existence:

> Unlocking, transforming, storing, distributing, and switching about are ways of revealing. But the revealing never simply comes to an end. (Heidegger, 1978: 298)

This operational system affects not only natural materials and the technological devices but also the human beings that 'run' them. All elements become regulated components within an overall mechanical constellation; none of them can ever be revealed in themselves – instead they constantly point elsewhere and only gain meaning in their systematic (differential) functioning.

Surprisingly, rather than altogether rejecting this prospect of systemic displacement and human alienation, Heidegger discovers within it a sense of strange hope. This hope is linked precisely to the instrumental character of modern technology; specifically to the ambiguous relation it opens up between revealing and hiding. Rather than directly, un-problematically, displaying Being in a natural and organic fashion (as evident in the model of traditional handicraft), modern technology shapes a blindness, a layering, a system of guises. For Heidegger this has the potential to provide access to a deeper layer of revealing – the truth, precisely, that truth can never appear as such, that it is inevitably in disguise – dissembling and adopting the form of copy, metaphor and sign. Heidegger argues that humanity 'keeps watch the unconcealment – and with it, from the first, the concealment – of all coming to being on this earth' (Heidegger, 1978: 313). If 'the essence of technology is nothing technological', it is because it is actually about the ultimate mystery of being and revealing:

> The question concerning technology is the question concerning the constellation in which revealing and concealing, in which the coming to presence of truth comes to pass. (Heidegger, 1978: 315)

If this alternative thinking of technology appears dangerous it is because it risks becoming lost in the tissue of concealment – truth is no longer co-extensive with direct, lucid appearance. It passes away from itself and beyond the control of self-collected, critical, human consciousness. The anthropocentric dream of human control and mastery is abandoned in order to conceive technology in radical instrumental terms as an opening and a displacement. Hence, for Heidegger, the importance of art as both a species of *techne* and as a means of maintaining a human, reflective element:

> [E]ssential reflection on technology and decisive confrontation with it must happen in a realm that is, on the one hand, akin to the essence of technology and, on the other, fundamentally different from it. (Heidegger, 1978: 317)

However, art can only perform this task of maintaining reflection within the space of semblance and loss if it takes technology seriously, if it 'does not shut its eyes to the constellation of truth concerning which we are questioning' (Heidegger, 1978: 317). The catch, however, is that this also necessitates a questioning of the nature of art as critique: 'the more questioningly we ponder the essence of technology, the more mysterious the essence of art becomes' (Heidegger, 1978: 317). This mystery takes shape precisely as the risking of critique – art itself appears as a passage away from truth as simple revealing. This is not a consequence of its inevitable opposition to technology (the conventional romantic aesthetic attitude that contrasts irrational, sensible-material art to the rational abstraction of technology), but instead arises from a fundamental engagement with the problem of technology. The mystery of art lies in its participation within the problematic of the instrumental.

Heidegger's conception of technology has clear relevance to the nature of software which is characterized by an enframed writing, a motion of functioning without human agency and by endless processes of structural hiding (abstraction and encapsulation). Art cannot resist these processes by simply projecting a naïve opposite. There is instead a need to insert itself within software, to partake of its processes, to follow its complex system of layering and dissembling.

Plato – Inspiration and Mimesis

Heidegger's perspective emerges as a creative response to a specifically modern concern, yet there are also ancient models for this view. It is easy to imagine an ancient unity of *techne* and *poesis* that is split apart within modernity, yet this sense of division is also apparent within the ancient world. Plato, for instance, writing around the same time as Aristotle, is adamant that *techne* and *poesis* are fundamentally opposed (Plato, c. 380 BC). Whereas Aristotle (1965), in his *On the Art of Poetry*, positions (dramatic) *poeisis*

as a domain of conceptually-guided skill, Plato, in his dialogue *Ion* (c. 380BC), casts *poesis* as form of sympathetic magic, of intoxication. The discussion between Socrates and the Homeric rhapsodist Ion sets out to establish that the latter rhapsodizes not through the mechanism of clear aesthetic precepts and skills but through the agency of divine inspiration:

> For all good poets, epic as well as lyric, compose their beautiful poems not by art, but because they are inspired and possessed. And as the Corybantian revelers when they dance are not in their right mind, so the lyric poets are not in their mind when they are composing their beautiful strains: but when falling under the power of music and metre they are inspired and possessed. (Plato, c. 380 BC: 5)

For my purposes, what is interesting here is that inspiration renders the artist an instrument. They are no longer in control, they can no longer entirely reflect upon, or claim essential priority for, the processes in which they are involved. They are caught up in operations that exceed them. Plato describes inspiration in terms in terms of the metaphor of a magnet:

> This stone not only attracts iron rings, but also imparts to them a similar power of attracting other rings; and sometimes you may see a number of pieces of iron and rings suspended from one another so as to form quite a long chain: and all of them derive their powers of suspension from the original stone. In like manner the Muse first of all inspires men herself; and from these inspired persons a chain of other persons is suspended, who take the inspiration. (Plato, c. 380 BC: 5)

The metaphor suggests a chain of inspiration that takes shape as a set of mediated relations. The 'original stone' can not itself be seen – it passes away from itself in order to manifest its attractive force. Each iron ring – Homer, the Homeric rhapsodist Ion and the audience – is linked together instrumentally as an ordered sequence and as a chain of unconscious attraction. However the chain also gives rise to apparitions, because

inspiration becomes manifest through appearances, through mimetic guises. If Plato (1955) ultimately expels the poets from his ideal republic, it is not only because they encourage dimensions of irrational and emotional excess but because they produce beguiling appearances that are a 'third remove' (Plato, 1955: 425) from truth. Genuine truth has its home in the sphere of abstract, mathematical form, whereas human beings live in the world of appearances (dark and shadowy and yet visible), and artists create appearances of appearances. The fundamental paradox is that inspiration has its basis in the revelatory experience of music and metre (which traces intimate links to the realm of ideal truth), but instead of producing truth it gives rise to falsehoods. Just like technology (conceived in Heidegger's terms) mimetic art renders revealing as concealing.

We find then that Plato's rejection of the mechanism of art (*techne*) only enables its more thorough grounding within art – not as conceptually guided, skill-based practice, but as the instrumental character of *techne* which here informs the nature of poetic inspiration and mimetic form; taking shape as the suspension of self-collected human agency and in systems of dissembling that chart an undecidable relation between the revealing and concealing of truth.

How can Plato's scheme, in which the aesthetic and the instrumental discover a surprising space of association, contribute to a re-evaluation of the status of the instrumental within software art? The layers of abstraction that characterize code operations are certainly not mimetic, but they obey the fundamental form of mimesis inasmuch as they involve a motion away from self-present origin. Similarly, although Plato's conception of poetic intoxication may seem very distant from rational software processes, the notion of involuntary *poesis* – in its automatism and blind pull – summons a sense of Turing's concern with the universal machine's dumb mechanical functioning. Within this context it is worth recalling that Adorno and Horkheimer conceive instrumental rationality precisely in terms of a limit point of reason in which rationality and irrationality coincide (Adorno and Horkheimer, 2000: 172). If instrumental rationality reveals an irrational dimension, it is not only in terms of the division it opens up between *episteme* (knowledge of invariable principles) and *phronesis* (morally guided

practical action), but also in terms of its suspension of human agency, its orientation towards an automatism that inevitably comes to resemble intoxication.

Cropper_Propper_Gridder

Overall then the genuine aesthetic potential of software lies in engaging with everything within software that seems most intrinsically inimical to the aesthetic – dimensions of instrumental function, non-reflective process and formulaic expression. Rather than struggling to find means of lifting up software to the status of art, there is a need to delve into the instrumental character of software, to genuinely engage with this space of risk and aesthetic alienation. This is what *Cropper_Propper_Gridder* attempts. The work provides an example of an effort to conceive the relation to the instrumental differently. If it does not take adequate shape as either a piece of software art or genuinely useful tool, then it is because it is concerned to explore a space of tension and awkward possibility.

The name is enough to suggest a dimension of awkwardness. *Cropper_Propper_Gridder* refers to three separate pieces of software that together form an apparatus for decomposing video and playing it back in discrete, grid-based portions. When it was exhibited, however, the work had a different name. It was called *Ice Time*, which related to a specific instance of the work which focused on video sequences from the Ross Sea region in the Antarctic. This suggests another dimension of awkwardness; the awkwardness of the distinction between the visible interface with its specific instances and the generic character of the work as an engine, as a mechanism of decomposition, choreography and display. Which of these demands attention? Which of these has a properly aesthetic character? Or is it both? And if it is both then how are they to appear simultaneously? What would this mean? The work raises these kinds of awkward questions. Prior to considering its uncertain status as a piece of software art, there is a need to provide more detail about the work itself, considering the underlying concept, the technical system and the exhibition context.

Concept

The project had its basis in the philosopher Henri Bergson's (1911) condemnation of the

'cinematographic' representation of time (which I mentioned in chapter 3). According to Bergson, time as duration represents a qualitatively whole motion that cannot be subdivided without fundamentally altering its character:

> All is obscure, all is contradictory when we try, with states, to build up a transition. (Bergson, 1911: 313)

Film, as a technology for cutting up time into frames and reassembling it for illusory temporal display, appears as a metaphor for the modern alienation from the genuine experience of duration. In response to this, I wondered, perversely, whether time was not better experienced through a mechanism; not as a predictable linear sequence, but as a work of setting time astray, of manufacturing, emphasizing and exacerbating its obscurity. I was thinking of projectors that run too slowly, in which the individual frames are visible, in which a sense of time emerges precisely through the disengagement of actual continuous time, in which time is manifest not as a single flow but as a set of flickering instants which serve both as an alienated reminder of some other time and as an immediate, yet dislocated, perception of current duration.

This interest, this thematic space, is clearly not unique. It charts relations to long-standing aesthetic concerns within avant-garde film and video art, from the exploration of aspects of temporal sequence in Dziga Vertov's 1929 *Man with a Movie Camera*, to Chris Marker's concern with the invisible time of the black film leader in *Sans Soleil* (1983), to the Australian artists, Rodney Glick and Lynette Voevodin's display of columns of hours from a single day in *24Hr Panoramas* (1999-2006). It also connects to the tradition of experimental new media which explores issues of time in terms of the re-combinatory possibilities of computation (Jaschko, 2003). Some influential works include Joachim Sauter's and Dirk Lusebruk's *Invisible Shape of Things Past* (1995) which reconstitutes time slices as peculiarly non-temporal, sculptural entities, Martin Reinhart's and Virgil Widrich's *tx-transform* (1992-2002) which swaps the axes of temporal and spatial representation, and most relevantly Camille Utterbach *Liquid Time* (2001-2) which enables portions of the video frame to play at different speeds and in different directions. My work explores similar possibilities. It decomposes the video

frame into rows and columns of independently playing image sequences – in an effort to stage both the deconstruction of ordinary time and a summoning of temporal alterity.

It is at this point that the conventional aesthetic idea necessarily engages with a technical imaginary. There is a need to consider how the various aspects of the system can be implemented. There is a need to devise systems, tools, engines. There is a temptation to disregard this as a work of subsidiary technical implementation, but for me it indicates the vital process in which the aesthetic concept takes practical and poetic shape as an instrumental apparatus.

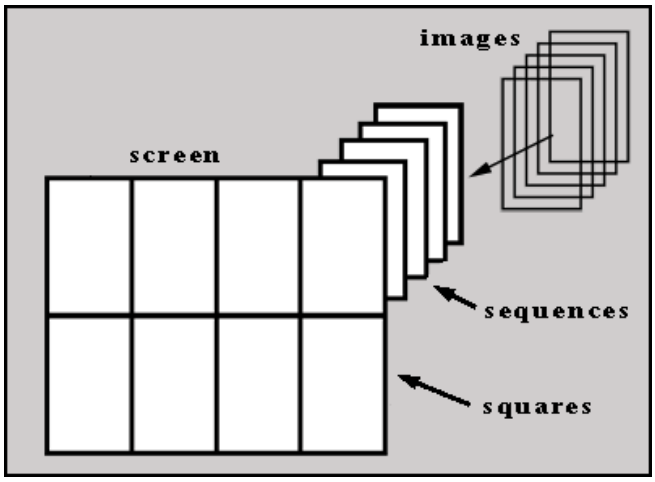

Fig. 17: Brogan Bunt, *Cropper_Propper_Gridder* (2005) design concept

Technical System

Above is a diagram of the display system:

The screen is composed of any number of squares which may or may not be arranged in a grid based manner. Each square is composed of a set of sequences of still images. Each sequence may be played independently and in various ways (in terms of speed, direction, etc.). Sequences may have associated sound files which may loop or play a specific number of times. Finally aspects of playback may be choreographed in advance or enable live interactive control.

The basic technical challenge involved finding means to decompose a video sequence into a set of independently playing image sections. The neatest and most logical approach was to employ a single video file and dynamically decompose and reassemble frames from the cube of spatio-temporal data. For long sequences, however, this was likely to demand retaining a very large number of frames within RAM and a constant process of multi-frame analysis to constitute any specific display frame. This is probably technically feasible but seemed beyond my means. Another obvious approach was to cut up the video in advance and play back any number of independent video streams. This proved unworkable due to the considerable overhead that each stream of video imposed on the overall system. It was not possible to play back more than a couple of video files at once. My only other option was entirely simple, even anachronistic. It involved conceiving the video sequences as game-style sprites. Video sequences were decomposed into sets of video stills and then decomposed again in to sets of cropped images. Represented as sprite arrays, these sets of cropped images could be played back in conventional sequential order, randomly or in any number of specific algorithmic ways. This was the approach I adopted and miraculously it seemed to work even for a finely articulated grid (60 or so sequences running simultaneously), but it had one major drawback. Instead of a single video file or a relatively small number of cropped video files, I had multiple directories filled with innumerable tiny image files. In this sense, it was a plainly awkward and inefficient solution. Moreover, in its literal complexity, in its fragmentation of data, it opened up the necessity for a set of specific tools to handle aspects of decomposition, choreography and display.

Cropper

Cropper is a small and unassuming utility program that handles the process of first cutting up sequences of video stills into rows and columns of cropped images and then saving them within an appropriate directory structure. It obscures the major part of its underlying functioning, merely displaying dynamic information concerning the percentage of images processed.

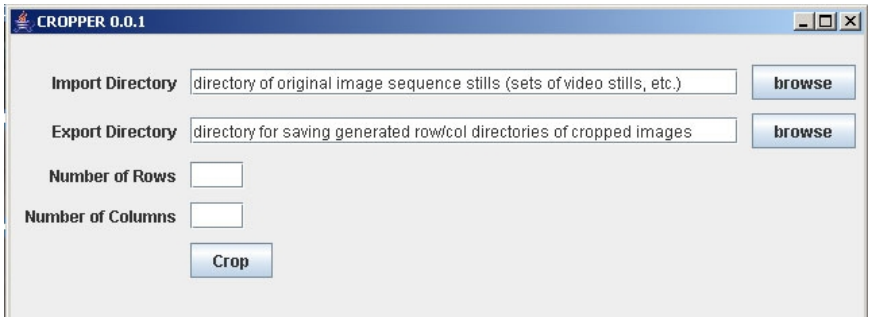

Fig. 18: Brogan Bunt, *Cropper* (2005) interface

Propper

Propper is a much more ambitious program. The role of this tool is to produce the underlying score that choreographs aspects of display. It builds XML (Extensible Markup Language) description files that the display engine, *Gridder*, reads in order to know what media to load when and where. XML makes dimensions of logical structure visible, legible and easily accessible (within text editors, browsers and so on); however, it can be slow to prepare manually. *Propper* provides a rapid, visual means of writing these files.

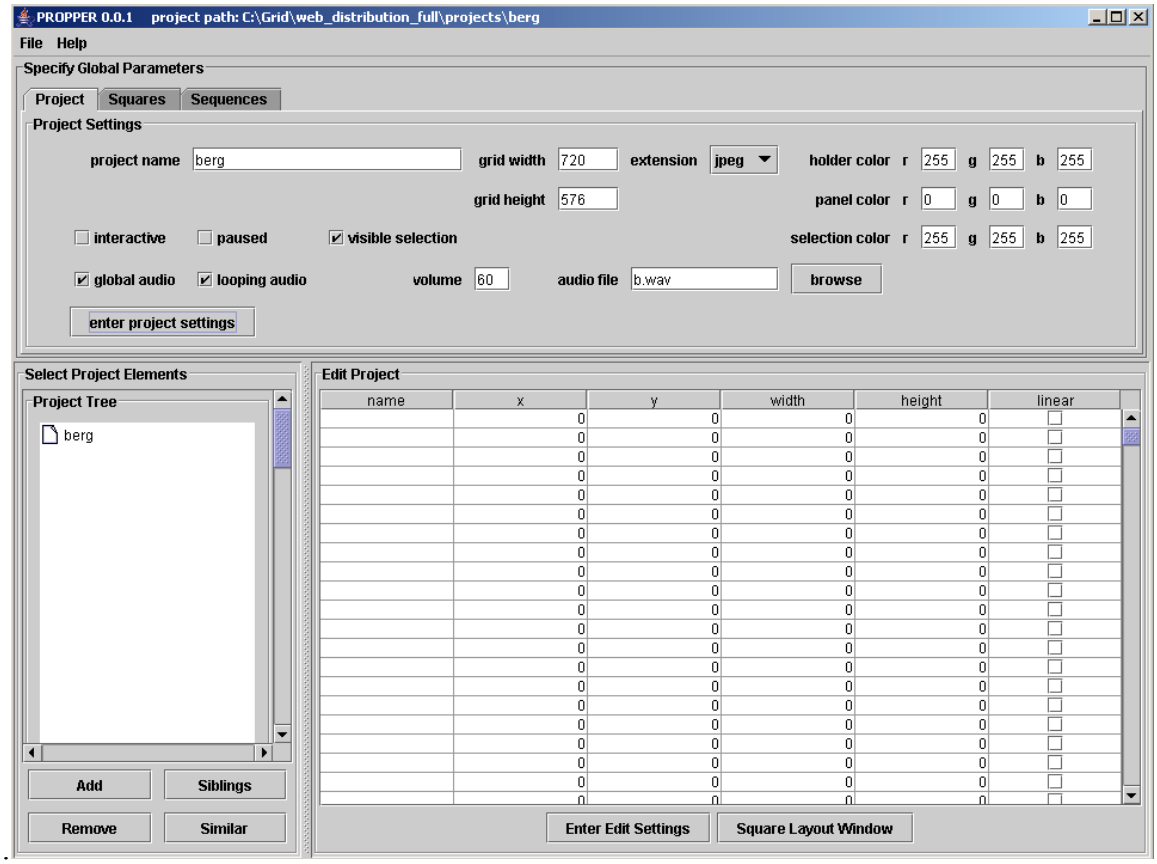

Fig. 19: Brogan Bunt, *Propper* (2005) interface

Gridder

Gridder is the display engine. A dialogue opens requesting that the user point to a relevant project directory. *Gridder* reads the project xml description file and commences media display. Additionally, the software enables interactive control of the playback parameters of image sections. *Gridder* displays in a screen window with a standard title bar. This is intended to remind the viewer/user that the work is a piece of software, not a piece of linear, pre-constructed video.

Fig. 20: Brogan Bunt, *Gridder* (2005) interface

Here is a screen shot without the title bar to provide a sense of the visible output of a more complex piece (a 9X3 grid with multiple 'video' sequences):

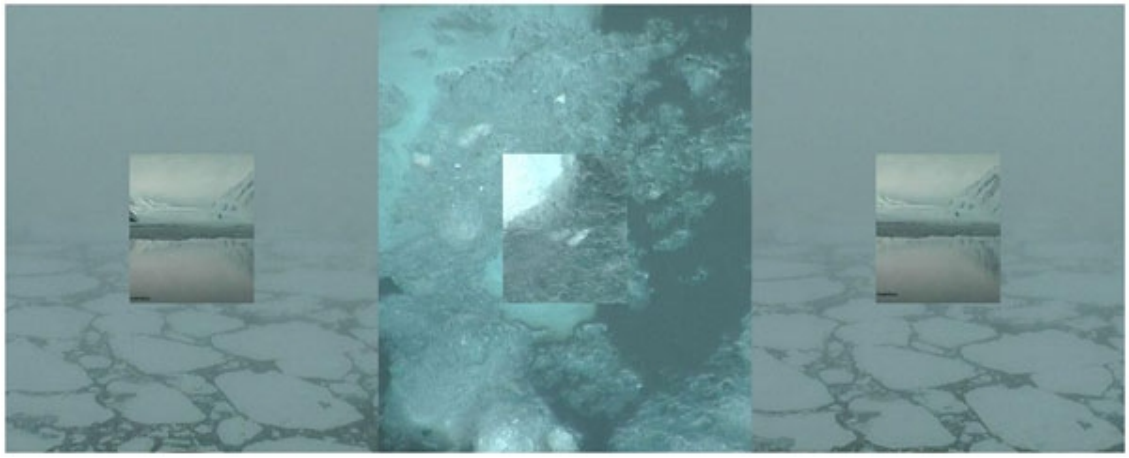

Fig. 21: Brogan Bunt, *Gridder* (2005), *Ice Time* exhibition (2005)

Exhibition

The screen shot above (Figure 21) is from the *Ice Time* exhibition. The emphasis was upon the display of fragmented video sequences of the Antarctic. The choice of footage from the Antarctic was deliberate. Antarctica is generally perceived as a space of pure glacial duration, yet we have recently become aware of its extreme temporal fragility; the Antarctic is entering another time – a time of division, of breaking up, even of imminent

catastrophe. Without pursuing this point in an explicit political manner, the conjunction of the software apparatus (as a work of temporal decomposition and the flickering re-summoning of time) and the samples of an entirely fragile realm of duration suggests a dimension of temporal uncertainty that has political implications.

Leaving aside the specific thematic issues addressed in this exhibition, the issue that mainly concerned me was the near invisibility of the *Cropper/Cropper/Gridder* apparatus in the exhibition display. Although I had spent close to two months producing the software and perceived it as the vital context in which the aesthetic concept took generic shape, there seemed no satisfactory way of acknowledging the apparatus, of making it appear aesthetically. I was very aware that the project may appear as a work of video compositing rather than software art. It is this sense of uncertainty (and frustration) concerning how to adequately exhibit the work that has prompted this specific reflection on the instrumental.

Software Art?

As I have suggested, the vital problem that the project raises for me is in identifying the properly aesthetic character of the work. A conventional view would distinguish between the aesthetically significant exhibited work and the aesthetically inconsequential background technical infrastructure. The contemporary notion of software art seems to provide a corrective to this view but ends up insisting upon a non-instrumental model of software as a form of abstract formal enquiry and/or self-reflexive software critique that has the unfortunate consequence, once again, of positioning the instrumental component of *Cropper_Propper_Gridder* as work of mere technical implementation. At the same time the software art status of the exhibited interface is questionable because it is less about reflectively revealing the dimension of code than about setting code into relation with the particularity of specific temporal samples. On what basis then do I regard the overall project as a work of software art?

The project represents a meditation on issues of the coded, discontinuous character of represented time that is conducted through the medium – the linguistic and discursive forms – of software. In this sense, it represents an example of what Fuller describes as

'speculative software' (Fuller, 2003: 29). Although the boundaries blur, Fuller distinguishes speculative software from 'critical software' (Fuller, 2003: 22) in that the former is oriented less towards deconstruction than making; it engages with 'the havoc of invention' (Fuller, 2003: 32). *Cropper/Cropper/Gridder* may be regarded as a speculative apparatus; it takes shape as a perverse media player, one in which the dimension of time is disarticulated and re-composed.

The work gains aesthetic coherence as an overall system that includes an element of generic operation and specific instantiation. In terms of the generic character of the work, the underlying poetic idea is realized as a linked system of functional tool-based operations which together form an abstract machine, an engine. *Cropper* represents the motion of decomposition, *Propper* the work of reassembling, and *Gridder* the rendering of an unnatural spatio-temporal logic in actual time. In its operation, the engine inevitably structures a moment of instantiation. The aesthetic dimension of the latter emerges in the friction between coded time and the particularity of sampled actual time. Without an awareness of the background software, this sense of friction is lost. The dimensions of interface and implementation are integrally aesthetically related.

The instrumental orientation represents an important aesthetic choice. The strangeness and technically anachronistic character of the project is heightened precisely by pursuing it through the agency of the instrumental, by discovering means to realize a perverse, absurd, idiosyncratic idea as efficiently as possible. Accordingly, at a stylistic level the software resists adopting a conventional aesthetic guise; it is deliberately blankly ordinary. The Java Swing style interface elements – menus, tabbed windows, hierarchical lists, radio buttons, etc. – interested me particularly in their anonymity and their embeddedness in the logic of instrumental software production and use. If the explicit conceptual theme is the alienation of time via mechanical division then the choice of a blank instrumental style works in my view to heighten the sense of alienation.

Of course, the problem is that none of this was seen by the exhibition audience. This is written then as a critique of the work's original mode of exhibition. The work needed to demonstrate both the engine and the interface in order to properly address the conceptual

them of the coded representation of time, as well as the equally important theme of the relation between an instrumental apparatus and an aesthetic concept.

Conclusion

My overall argument is that rather than positioning the instrumental, tool-based character of software as some grim fact that must be rigorously resisted, there is vital need to work through the instrumental, to explore its possibilities. This entails a risk – the risk of facilitating software functioning, of engaging with its work of abstraction, encapsulation and disguise. It projects a space of uncertain creation that cannot altogether shake of a relation to the blindness of mechanical process, that must find means to reflect amidst a work of operational making. The clear difficulty is in finding adequate means to conceive a work of critical reflection within the texture of instrumental relations when the self-consciousness of critical awareness is precisely what is put at risk. In my view there is no easy solution to this dilemma. Instead there is a constant work of negotiation – of engagement and distanciation with whatever it is that an instrumental device and an aesthetic work represents. This would seem to demand a re-examination of the relation between 'software culturalism' and 'software formalism' (Cramer, 2002:10). It may be that it is precisely at the level of form (regarded as a material discursive fact and experimental space) that the most profoundly cultural questions are raised. Of course, how these questions are to be articulated – how they are to take constitutive shape as processes, engines and interfaces – remains an open question.

Chapter 7: Openings

Introduction

The third and final dilemma of software art that I mentioned at the end of chapter 4 is that of recursion. This is once again a problem of reflection. Software, it is argued, can only aspire to art if it is reflective. Faced with conventional software's tendency to hide and its insistent instrumental orientation, software art attains properly aesthetic shape in its work of reflection. The revelation of aspects of code process and the critique of the culture of software becomes the dominant thematic concern. As a result, software art is caught within a recursive cycle. It is doomed to endlessly reflect upon its own formal and cultural conditions. To pursue any other concern appears as an affirmation of the invisibility of instrumental functioning.

On this basis, Johansson argues that software art is bound to a *mise-en-abyme* aesthetics (Johansson, 2004: 151) that leaves little room for aesthetic opening and Cramer (2002) writes of the apparent lack of future for both formalist and culturalist tendencies:

> If Software Art would be reduced to only the first [formalism], one would risk ending up with a neo-classical understanding of software art as beautiful and elegant code along the lines of Knuth and Levy. Reduced on the other hand to only the cultural aspect, Software Art could end up being a critical footnote to Microsoft desktop computing, potentially overlooking its speculative potential at formal experimentation. (Cramer, 2002, 10)

I have questioned the value of conceiving software art entirely in terms of an opposition between formal and cultural orientations. Apart from obscuring a more significant dialogue between instrumental and aesthetic discourse, it also renders formalism and culturalism as caricatures. A concern with form, for instance, need not entail a lack of concern with anything else. It can encompass interests that extend beyond the self-collections of form – that seek out the non-identical precisely through the paradoxical agency of form. This is evident, for example, in the tradition of Minimal Art, in which an extreme awareness of form and an extreme formal reduction are oriented towards the

staging of *aporia* rather than the formal integrity of a composition. Similarly, culturalism need not take exclusive shape as a concern with the culture of software. Instead of single-mindedly looping back on itself, software may engage with culture in all manner of different ways. I think of the example of the radical documentary tradition – filmmakers such as Chris Marker, Jean Rouch and Georges Franju – who manage to reflect upon the epistemological and aesthetic conditions of documentary film at the same time as exploring other issues. Their work of reflection avoids disabling solipsism and becomes integral to pursuing a broader set of cultural concerns.

This chapter aims then to resist this sense of reflective closure – to suggest possibilities of opening. In the previous chapter I discussed a specific, and particularly important, form of opening – the relation to the instrumental. Rather than possessing an entirely internally coherent identity, software art engages with a space of instrumental functioning that both shapes its interior dynamic and exceeds it. Here I wish to suggest three additional forms of opening: the first hinges on questioning software art's self-enclosed generic integrity, suggesting instead a permeable relation to a wider universe of aesthetic practice; the second takes shape in terms of software art's potential to engage with the alterity of the real; and the third concerns software art's relation to the aporetic opacity of finite computational processes. This chapter considers each of these forms of opening in terms of my own work. My argument is that while maintaining a reflective aspect, these openings signal the limits of conceiving reflection as an autonomous, constitutive, aesthetic ground. Rather than a pure space of self-present critical-aesthetic thought, reflective software art practice demands a thinking of dimensions of both exterior and interior non-identity.

Permeable Relations

Despite its technological specificity – the specificity, for instance, of its executable framework – software is not aesthetically isolated. Even were it to hope that it could endlessly reflect on its own processes, it would discover traces of other traditions and genres. This is hardly an original observation. Cramer (2005) does an excellent job of describing the rich cultural heritage that informs software art practice – from medieval rhetoric to OuLiPo poetic constraints and the language games of Conceptual Art. In a

more wayward fashion, Fuller traces links to the deconstructive architectural practice of Gordon Matta-Clark (Fuller, 2003: 39-49) and obscure disciplines such as Nomography ('This lost art is essentially that of producing gridded visual diagrams showing the results of what would otherwise be mental calculations' (Fuller, 2005: 161)).

While the permeable character of the genre is emphasized, specific sets of association have been avoided. I am thinking particularly of the relation to traditional 'mechanically reproducible' media such as film and photography. It seems that these are too obviously material, too apparently passive and too clearly tied to dimension of sensible perception to adequately engage with software art's constitutive abstraction and executable nature. While Manovich (2001), Geoffrey Batchen (2006: 27-44) and others have traced all kinds of links between traditional media and computation, these have occurred within the context of theorizing new media rather than software art. As we have seen, the fundamental effort to distinguish the specific character of software art depends upon bracketing new media, upon presenting new media as a kind of blindness to underlying code processes. This seems very unfortunate. Certainly my own software art practice is vitally constituted by a dialogue with aspects of traditional media. There is a tendency to associate the latter with a linearity that is inevitably opposed to the re-combinatory spatiality of software, yet this is to ignore strands of traditional media practice that anticipate the conceptual patterns of software. Consider, for example, the formal device of film montage, which in Eisenstein's (1986: 181-183) classical conception is not about tying aspects of time and space together into a seamless and coherent narrative whole, but about placing paradigmatic elements side by side – suggesting something new in their friction, in their charged juxtaposition. Even a formal device such as looping repetition, which seems so intimately enmeshed in the language of computation, is anticipated by traditional media. I can remember spending many (highly aesthetically derivative) hours working with audio tape loops and producing scratch-style video in the early 1980s. In this sense, software programming represents another and more dedicated means of exploring themes of re-combinatory structure and iterative pattern that are co-extensive with strands of experimental media production.

A number of my software art projects are fundamentally concerned with the relation to traditional media. As I mentioned in the previous chapter, *Cropper_Propper_Gridder* is an alternative media player, reconceptualising the representation of filmic time in terms of the possibilities of computation. *Halfeti – Only Fish Shall Visit* (2001) connects a concern with code to the investigation of real social space. It explores the potential to develop links to film and photographic documentary traditions. It is worth examining the latter's relation to software art.

Fig. 22: Brogan Bunt, *Halfeti – Only Fish Shall Visit* (2001)

Halfeti – Only Fish Shall Visit is an interactive documentary focusing on a small Turkish town in the months just prior to its flooding by the waters of a large hydro-electrical project (the Birecik dam just north of the Syrian border on the Euphrates River). At a conceptual level, the work explores relations between the formal articulation of space found within the adventure game and the cultural concerns and representational aesthetics of the experimental documentary tradition. The work establishes a spatial-navigable interface to a large set of documentary data – several thousand photographic images, a large number of ambient audio files and close to an hour of video (incorporating

interviews and observational footage). The user gains the sense of 'wandering' about the town, following lanes and pathways, entering open doors, and here and there coming across people who explain aspects of their lives and their responses to an uncertain future.

Described in these terms, the work may seem to bear very little relation to the field of software art. The emphasis on interaction, display and media instantly suggests a work that is insufficiently self-reflectively code focused. Yet, alongside its documentary focus, *Halfeti – Only Fish Shall Visit* is very much concerned with code. Indeed it explicitly aims to re-think the documentary genre in terms of the possibilities of code. One of the specific challenges of the work was to discover an appropriate means to represent the complexity of a real social space. I had previously produced simple navigable games in which space was represented as a grid and movement from one place to another was calculated mathematically. But the winding, irregular space of Halfeti demanded other strategies. There was a need to develop an abstract data structure that could somehow encompass complexity within a simple hierarchical framework. While the work makes no effort to literally display this data structure and indeed deliberately disguises it in the interactive interface, the work is reflective in another sense. A major aim was to qualify and unsettle the sense of immersive engagement that typical games establish. The user moves between static images, hears looping ambient sounds and encounters sudden, montage-like transitions from morning to afternoon, day to night, sunshine to snow. In this manner there is an explicit acknowledgement that the work is a coded mechanism that summons the past not in the guise of a fictional, available present but as a space of incomplete recollection and loss. Roland Barthes' (1981) account of photography in *Camera Lucida* provides some inspiration for this approach. Barthes represents photography as a wound that summons and manifests the intractable otherness of past events (Barthes, 1981: 77). I was attempting something similar in *Halfeti – Only Fish Shall Visit* – not simply at the level of individual images but through the fragile artifice of the navigational engine, which suggest less immediacy than the inevitably flawed retracing of steps within mechanical memory.

In this respect, *Halfeti – Only Fish Shall Visit* shapes a process of reflection which engages with the paradoxical character of traditional media; the curious combination of intimacy and distance that photography, for instance, projects. This motion of reflection, however, is not properly manifest at the level of source code (which can only appear opaque and mystifying to non-programmers) but functions instead at the level of the sensible interface – explicitly teasing out the friction between an artificial apparatus and the texture of the real. Veering from the standard conception of software art, the work resists an autonomous reflection on underlying software processes and instead posits one that is enmeshed with issues of representation. This dialogue with traditional media represents less a failure to properly conceive the aesthetic possibility of software, than an effort to genuinely think its permeable discursive character – its intrinsic dimension of opening.

Mechanism and Alterity

If traditional media focuses on issues of representation, describing a relation to the world and a space of encounter, computation typically suggests something different. It is about abstracting rules and conjuring simulations. Rather than the indexical trace, there is the numerical sample and the algorithmically constituted semblance. In this sense, computation less represents the world than abandons it and recreates it elsewhere. Consequently, it appears unaffected by the longing that Barthes describes – the endless play of summoning and deferral that characterizes traditional media. Instead of exterior relations, computation establishes a space of finite autonomy and self-functioning. Yet this can be interpreted differently. As I suggested in relation to *Halfeti – Only Fish Shall Visit*, an engine need not only remain focused on its own operations. It can become a means, paradoxically, of enabling a relation to the alterity of the real. Its motion of apparent turning away can enable a strange return.

In his discussion of Henri Bergson's notion of 'cinematographic time' (Bergson, 1911: 306) Deleuze describes the specific novelty of film representation: movement is no longer '*recomposed from formal transcendental elements (poses), but from immanent material elements (sections)*' (Deleuze, 1986: 4). Instead of human vision and deliberate framing and composition, the emphasis shifts to the mechanical sampling of instants of

time. The choice of instant is motivated by the logic of the camera apparatus rather than by any specific conscious representational agenda. In this sense, Bergson regards the film image as having an arbitrary temporal aspect; it is an 'any-instant-whatever' (Deleuze, 1986: 6). Precisely by suspending a level of human decision-making and intervention, cinema discovers a means to access something beyond the *a priori* conception of time; to figure time in its alien, material aspect. Film appears then as a mechanism of abstraction and division which enables paradoxical access to an ordinarily inaccessible dimension of sovereign particularity. In this manner, Bergson and Deleuze associate the specific representational power of film with features that anticipate core aspects of computation (automation and sampling). My interest here, however, is less in re-considering the nature of traditional media in terms of computation, than in recognizing that computation, and software art specifically, can also engage with a representational imaginary. My aim is to consider how the thinking of code shaped my processes of documentary production in *Halfeti – Only Fish Shall Visit* and provided a means of engaging with the specific texture of a historical place, but it may be worth considering another relevant example first.

The field of 'psychogeography', which draws on the concept of the *flaneur* and the *derive* (drift), is associated with Situationism and Fluxus. It is a mode of experimental practice that aims to encourage an open discovery of urban space, a re-interpretation of space in non-habitual and non-instrumental terms. Whereas this may have initially summoned a thinking of highly subjective, idiosyncratic forms of wandering, more recently there has been an emphasis on 'algorithmic psychogeography' (Crystalpunk, n.d.). Practitioners follow simple algorithms such as 'second right, second right, first left, repeat' (Crystalpunk, n.d.). Rather than remaining stuck in an abstract geometrical rut, the combination of a mechanical method and the complexity of real urban space opens up the potential for spatial discovery.

While *Halfeti – Only Fish Shall Visit* was much more directed towards evoking elements of spatial continuity, a similar friction between the demands of coded system and actual space is evident. If the normal photographic strategy is to frame an image to make a specific conceptual or aesthetic point, here my approach was circumscribed by the need

to record logical views that made sense within the overall spatial-interactive framework. Over a period of a few weeks, I followed a network of paths that led out from the center of the town, at regular intervals recording images forward, back, left and right. As a result, the town is documented in a curious semi-automatic fashion. I captured views that nobody else would bother to photograph – a faint path, a close-up of a wall, a dark room, half a tree. These banal, uninteresting, particular images have a positive value for me. They are the product of a process of spatial discovery that combines elements of schematic necessity and slight aesthetic mediation. They chart an association between hierarchical order and wayward wandering and engender a curious tension between the abstract and the particular. Abstraction becomes a means of staging an opening, of establishing relations to the alterity of the real.

Fig. 23: Brogan Bunt, *Halfeti – Only Fish Shall Visit* (2001)

Finitude and Non-Identity

It is not only by looking outside code that openings are discovered. They also appear within code, within the very motion of finitude that appears to constitute a stable basis for software art's reflective self-identity. Here it is less a matter of pursuing algorithms that

generate the semblance and underlying structure of organic complexity than of focusing upon basely mechanical operations. Programming shapes a relation to the machine that constantly hits up against the mystery of execution, of a motion of becoming that is manifest but not reflective.

This is a possibility that I explore in *Hotel* (2002). Hotel is an ironic generative game space. Instead of shaping an endless world of artificial wonder, it renders space explicitly Lego-like and repetitive. *Hotel* represents a response to *Halfeti – Only Fish Shall Visit*. It withdraws from the picturesque particularity of a real social place to explore a space of total artifice (the images are not photographs but renderings of an artificial 3D world). Instead of a pre-determined data structure representing every aspect of space, it begins with a void and a set of combinatory spatial elements. The space is built up in a dynamic random manner as users wander about. Once an option is selected, however, it is locked in place (hence users can always retrace their steps to the notional origin). Crucially then, rather than an overall map, the space is represented as a fragile and abstract tissue of associations between one spatial node and another (a dynamic linked list rather than an exhaustive hierarchical array).

Fig. 24: Brogan Bunt, *Hotel* (2002)

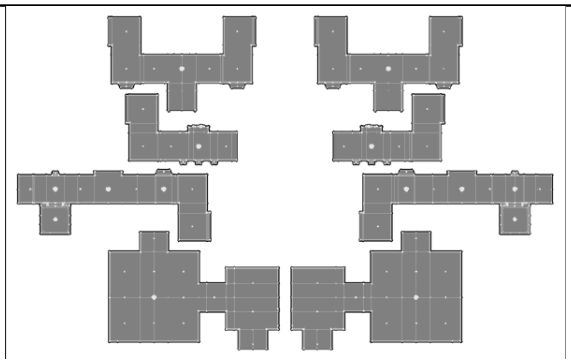

Fig. 25: Brogan Bunt, *Hotel* (2002) spatial elements

Users find themselves stuck in a hotel corridor with no other option than to wander about. They can wander as far as they like but only ever discover the same kinds of corridors, lifts and vestibules. If they do manage to find their way into a secret room which provides a kind of primal (very conventionally surreal) mythological scene, then they are

straight away transported back to the starting moment and an even clearer recognition of the recursive, cyclical nature of the space. To relieve the tedium, the hotel corridors contain distractions and strange clues which point to the secret room and aspects of the underlying mythological narrative. It may be worth briefly describing this narrative because it is specifically concerned with issues of finitude and opening.

Fig. 26: Brogan Bunt, *Hotel* (2002)

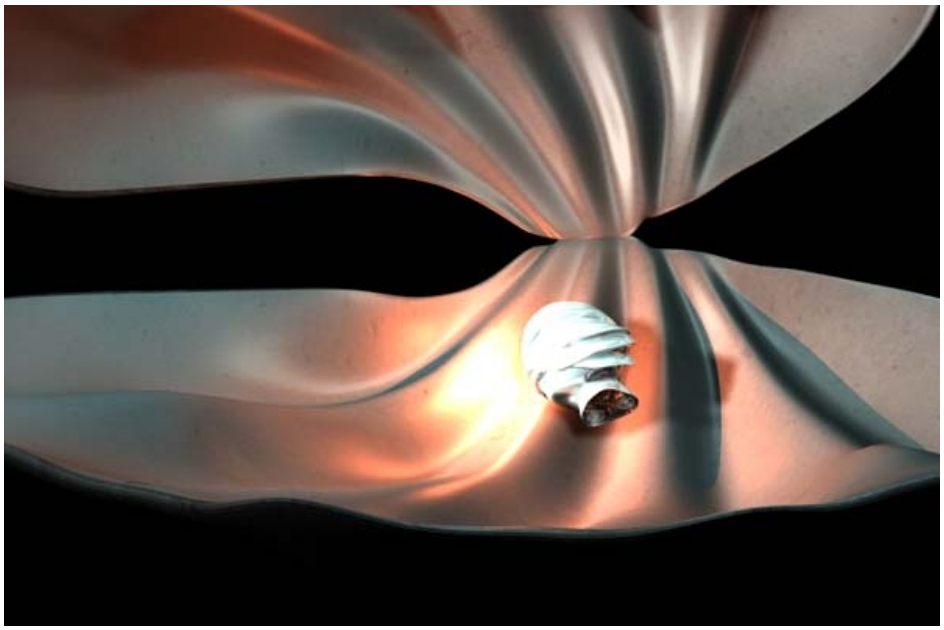

Fig. 27: Brogan Bunt, *Hotel* (2002)

The hotel appears as a space of order and light. It appears as the product of rational consciousness. Here and there, in images on walls, in voices associated with particular room and objects, there is the sense of some human architect. It appears visibly as a detached head. This head imagines that it is responsible for this space, that it has made it and manages it. Yet despite its brightness, there are elements that suggest that something is amiss. Apart from the oddness of the spatial repetition, there are inexplicable pictures, bizarre intercoms and televisions, and alcoves containing chromed body parts, sea creatures and household utensils. Here and there, as well, weird fish swim along the corridors. The secret room (hidden in a keyhole) provides an alternative account of this place. Rather than the product of reason, it is a bubble produced from the mouth of a fish. It is the product of darkness and chaos and the human head, endlessly dreaming, only imagines its constitutive power. This is intended as a metaphor for the apparent closure of computational systems, which is always precarious and, in its most fundamental processes, reveals traces of non-identity that elude self-collected consciousness.

My aim in this project was to ironically reflect on the closure of the digital, yet at the same time, strangely and unexpectedly, it came to reveal other possibilities. Abject processes of looping, repetition and random recombination engaged with the curiosity of mechanical manifestation. The system became strange in its operation, in its play. Something as simple as the random choice of one of four rooms, one of seven sound files or one of twelve video files suggested a dimension of non-identity within digital processes themselves. This can be linked to Nietzsche's notion of the 'eternal return', in which the metaphor of the dice throw suggests a relation between chance and necessity and in which finitude becomes the basis for a thinking of the openness of becoming. Nietzsche describes the motion of the dice in terms of 'dynamic quanta, in a relation of tension to all other quanta' (Nietzsche, 1968: 339). Software art can choreograph forces and set them at play, but the actual motion of execution constitutes a domain of estrangement and opening.

It is not as though this mode of opening ever appears sufficient. Simple random operations often appear as a cliché, yet at the same time, despite their over-use, they

retain a curious sense of wonder which is precisely linked to a non-reflective dimension of performance. They engage the creative capacity of the machine, which is far less about staging generative complexity than manifesting the most basic decisions.

Conclusion

These three forms of opening shape an important field of magnetic attraction within my software art practice. At times I pursue openings in the relation to traditional media, at times in relation to the alterity of the real, and at other times within the non-identity of programmatic processes. The various poles of attraction can both stray apart and coincide. I would like to conclude by mentioning two recent works that pursue radically different directions.

Walk

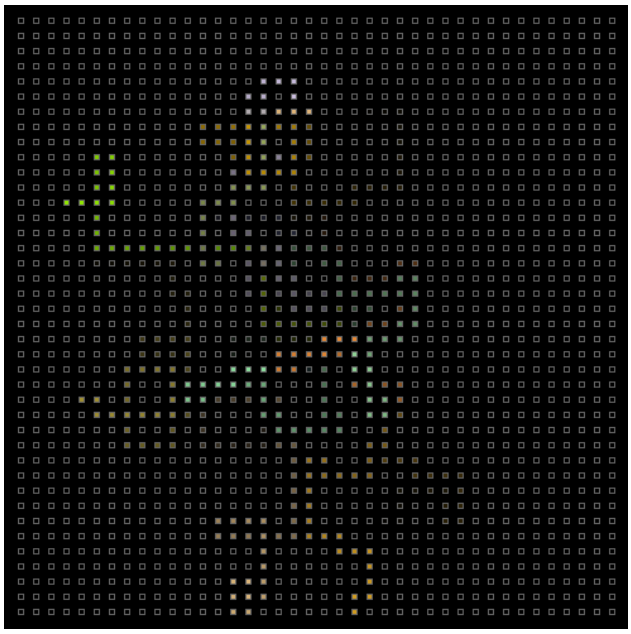

Fig. 28: Brogan Bunt, *Walk* (2006)

Walk is a perverse work that draws the field of psychogeography back within computation. Algorithms geared to the discovery of real space, now return to code and flicker about on a strictly-defined grid. A set of walk objects follow mathematically determined paths across the grid. In standard game style, if they encounter an edge then space wraps to the opposite side of the grid. If the objects meet on the same grid square

while making a navigational decision then they are randomly allocated new colours and algorithms. This project represents less a withdrawal from the real than an attempt to think it askance, to transpose it back within abstraction. At one level, it is a parody of human psychogeographical motion. At another level, it is a work of visualization, enabling code to become metaphorical and thus be seen. At another level again, it engages a play of finite wonder – it is fascinating (for me) as a repetitive and yet inexplicable space of manifestation.

Paphos

Fig. 29: Brogan Bunt, *Paphos* (2006)

Another recent work follows an almost opposite trajectory. *Paphos* is a piece of documentary video art, exploring moments at the margins of an Australian archaeological dig in Paphos, Cyprus. Despite its apparent distance from software art, the work – like *Halfeti – Only Fish Shall Visit* – conceives documentary in terms of the thinking of code. The process of recording followed a roughly determined set of constraints. Each sequence is about one minute long and the camera remains motionless throughout. The sequences represent samples of Paphos time. I determine the framing but cannot predict what will happen in front of the camera during the minute of recording. Individual samples are strung together into a vague and inconsistent temporal sequence

(representing a gradual shift from morning to night), but the collection of samples are essentially constituted as a paradigmatic set, representing mythological, contemporary, iconic and interstitial aspects of Paphos.

These projects indicate that software art appears to me not as an autonomous space but as a motion back and forth between code and other media, code and the world. It describes dynamic trajectories and systems that inevitably inscribe relations beyond the fantasy of closure.

Chapter 8: Conclusion

This dissertation has conceived software art as a space of tension which mediates new relations between machine and human, instrumental and aesthetic and abstract and particular. I have argued that these oppositions have both formal-aesthetic and cultural-institutional implications. A close concern with the language of code entails engagement not only with a formal linguistic and conceptual space but also with a broader social space. Software art cannot escape its relation to the more general culture of software – the sphere of industrial making and instrumental functioning which provides a vital ground and point of reference for contemporary experimental practice. Rather than shying away from these relations and withdrawing into the safety of traditional conceptions of the aesthetic, software art does much better to acknowledge and pursue its genuine space of risk.

Engaging with Code

I have considered the notions of software and software programming. I have traced the historical emergence of software art and have examined specific dilemmas that confront the genre in terms of issues that arise within my own work. I have not, however, offered a clear alternative definition of software art practice. Pressed on this issue, I would argue that software art represents a close engagement with the language and discourse of software production. It emerges through a work of coding, through an imaginative relation to the field of computer programming. This takes characteristic form in work that is literally coded and that bears a fundamental and intimate concern with the field of machine functioning, but it can take other forms as well. Manovich speaks of processes of 'transcoding', in which the forms of code come to shape the 'cultural layer' (Manovich, 2001: 46) and Cramer argues that 'software is no longer just machine algorithms, but something that includes the interaction, or, cultural appropriation through users' (Cramer, 2005: 122). For me, a work such as *Paphos*, which is not literally a piece of software, is shaped nonetheless by a coding imaginary. While only loosely a piece of software art, it highlights a wider space of conceptual-aesthetic possibility and suggests an inevitable dialogue with other modes of creative practice.

Some critics, however, advocate a more restricted conception of software art. Despite the inclusive rhetoric of the early *transmediale* and Read_Me festivals, critics such as Inke Arns (2004) associate software art exclusively with practices of cultural critique and bracket formalist genres such as generative art due to their apparent lack of critical self-reflexivity. Arns argues that whereas generative art focuses on the performance and visible results of generative (artificial life) algorithms, software art focuses upon exposing underlying machinations (Arns, 2004: 183). My problem with this restricted definition is that it not only oversimplifies the field of generative art – which includes practitioners such as Paul Brown whose work demonstrates, as Mitchell Whitelaw argues, an explicit concern with 'purely formal structures, templates for computation, patterns of rules' and an 'unhinging of figure and mechanism' (Whitelaw, 2004: 147) – but, more significantly, imagines that the engagement with code can constitute a pure moment of critical revelation. I have argued, on the contrary, that programming necessarily entails processes of hiding – that its operations elude and undermine simple visibility. Rather than literally exposing code, software art inevitably shapes metaphors and disguises, abstractions and interfaces. Indeed, as I suggested in Chapter 4, many pieces of formalist software art – and generative software art specifically – are precisely constituted as metaphors for underlying code processes. John Conway's *Game of Life*, the founding work of generative artificial life, provides an exemplary instance (Conway, 1970). The visual elements – the abstract grid of squares and the flickering play of 'gliders', 'blinkers' and other forms of 'life' – provide a means of explicating the rules, of playing them out in a sensible, visible manner.

In my view, the exclusive emphasis on critical self-reflection serves as an escape from the risk of software art, a disengagement from the dimension of process. It represents an effort to portray the relation to software in terms that entirely correspond to a fantasy of assured radical aesthetic practice. In this manner, it disregards software art's dependence upon the discursive space of conventional software and its necessary complicity with the language of instrumental functioning. Overall, there is a need to conceive the medium of software and the critical character of software art in more subtle terms. Software art can

enable neither a pure gesture of formalism nor an entirely self-collected moment of reflective critique. It inhabits a messier and more uncertain terrain.

In relation to formalism, although at one level software art manifests a return to the notion of a medium (a space of writing, of craft), at another level it abandons the reassuring character of a traditional medium; form and matter are no longer immanently combined, they slip apart. Instead of a complex single aesthetic substance (the organic unity of a work), there is a play of abstraction, layering, disappearance and disguise. Furthermore, the material space of software writing demands a thinking of the cultural dimensions of order and algorithmic process. It is not easily or adequately reducible to a nakedly formal potential.

Similarly, critical-cultural software cannot maintain the pretence of pure opposition. Its engagement can never be simply critical. Fuller's notion of 'speculative' software practice (Fuller, 2003: 29) provides a means of describing an exploratory, typically anachronistic work of tinkering with aspects of the software heritage that can potentially shape all kinds of relations between processes of imaginative making and critical reflection.

Overall, my enquiry suggests that the genuine political potential of software art emerges less in terms of explicit efforts at deconstruction than in terms of a rigorous (and self-consciously anachronistic) engagement with the technological tradition. It is by finding one's way (and becoming lost) in this complex technical-discursive space that other possibilities emerge. Software art, in my view, opens up an intimate relation to processes of operation and making and unsettles narrowly critical views of art practice. It risks the unconsciousness of mechanism. It pursues and reflects upon this risk. It discovers in this risk a source of inspiration.

Bibliography

8-Bit Collective (Beige programming ensemble) (n.d.). [website]. http://www.post-data.org/beige/ (accessed 23 January 2007).

Adorno, T. (1997). *Aesthetic Theory*. London & New York, Continuum. First published in 1970.

Adorno, T. and Horkheimer, M. (1982). 'The Culture Industry: Enlightenment as Mass Deception', in Curran, J., Gurevitch, M. and Woolacott, J. (eds). *Mass Communication and Society*. London, Open University Press, pp. 349-383. First published in 1947.

Adorno, T. and Horkheimer, M. (2000). 'The Concept of Enlightenment', in O'Connor, B. (ed). *The Adorno Reader*. London, Blackwell Publishing, pp. 156-173. First published 1944.

Agar, J. (2001). Turing *and the Universal Machine: The Making of the Modern Computer*. Cambridge, Icon Books.

Agile Alliance (n.d.). [website]. http://www.agilealliance.org (accessed 8 January 2007).

Aristotle (350 BC c.) *Nichomachean Ethics*. http://classics.mit.edu/Aristotle/nicomachean.html (accessed 4 January 2007).

Aristotle (1965) *On the Art of Poetry*, in Dorsch, T.S. (ed). *Classical Literary Criticism*. Great Britain, Penguin Books, pp. 31-75. First published: undated.

Arns, I. (2004). 'Read_Me, Run_Me, Execute_Me: Software Art and its Discontents, or: it's the Perfomativity of Code, Stupid!', in Goriunova, O. and Shulgin, A. (eds). *read_me: Software Art & Cultures*. Arhus, Aarhus University Press, pp. 176-188.

Babbage, C. (2005). 'Of the Analytical Engine', in Norman, M. (ed). *From Gutenberg to the Internet: A Sourcebook on the History of Information Technology.* Novato, California, thehistoryofscience.com., pp. 281-293, First published 1864.

Banzi, M. (2006). 'Getting Started with Arduino'.

http:www.arduino.cc/en/uploads/Booklet/Arduino_Booklet02.pdf (accessed 15 January 2007).

Barthes, R. (1981). *Camera Lucida*. New York, Hill and Wang.

Batchen, G. (2006). 'Electricity Made Visible', in Chun, W. H. K. and Keenan, T. (eds). *New Media/Old Media*. New York & London, Routledge, pp. 27-44.

Beige (programming ensemble) (2001). 'Deadtech: Post-Data in the Age of Low Potential'. http://www.epidemic.ws/biennale_press/deadtech.htm (accessed 16 December 2006).

Bergson, H. (1911). *Creative Evolution.* New York, Henry Holt and Company.

Blais, J. and Ippolito, J. (2006). *At the Edge of Art.* London, Thames & Hudson.

Blast Theory (2004). http://www.blasttheory.co.uk/ (website, accessed 17 December 2006).

Borevitz, B. (2004). 'Super-Abstract: Software Art and a Redefinition of Abstraction', in Goriunova, O. and Shulgin, A. (eds). *read_me: Software Art & Cultures*. Arhus, Denmark, University of Aarhus, pp.298-312.

Brown, P. (2000). 'Stepping Stones in the Mist'. http://www.paul-brown.com/WORDS/STEPPING.HTM (accessed 20 December 2006).

Brown, P. (2003). 'The Idea Becomes a Machine: AI and Alife in Early British Computer Arts'. *Consciousness Reframed: Art and Consciousness in the Post-biological Era* (5th CAiiA International Research Conference), Caerleon UWCN Wales UK. http://www.paul-brown.com/WORDS/CR2003.PDF (accessed 20 December 2006)

Burger, P. (1984). *Theory of the Avant-Garde.* Minneapolis, Minnesota, University of Minnesota Press.

Chesher, C. (2001). *Computers as Invocational Media* (unpublished PhD thesis). Sydney, Macquarie University.

Chun, W. H. K. and Keenan, T. (eds) (2006). *New Media, Old Media*. New York and London, Routledge.

Copeland, J. (2004). 'Computation', in L. Floridi (ed). *Philosophy of Computing and Information*. Malden, Massachusetts, Blackwell Publishing, pp. 4-6.

Cramer, F. (2001). untitled comment on Manovich. http://absoluteone.ljudmila.org/cream.php (accessed 21 December 2006).

Cramer, F. (2002). 'Concepts, Notations, Software, Art'. *Seminar for Allegmeine und*

Vergleischende Literaturwissenschaft.

Cramer, F. (2005). *Words Made Flesh.* Rotterdam, Piet Zwart Institute Media Design Research.

Cramer, F. and Gabriel, U. (2001). 'Software Art'. Transmediale.01 Arts Festival, Berlin.

Crystalpunk (n.d.). 'Crystalpunk: The Chain Reaction Glitterati'. [website] http://socialfiction.org (accessed 13 January 2007).

Deleuze, G. (1986). *Cinema 1: The Movement Image*. Minneapolis, Minnesota, The Athlone Press.

Derrida, J. (1976). *Of Grammatology*. Baltimore & London, The Johns Hopkins University Press.

Druckrey, T. (ed.) (1999). *Ars Electronica: Facing the Future*. Cambridge, Massachusetts, The MIT Press.

Duchamp, M. (1973). "The Green Box", in Sanouillet, M. and Peterson, E. (eds). *The Writings of Marcel Duchamp*. New York, Da Capo Press, pp. 26-71. First published 1934.

Eagleton, T. (1990). *The Ideology of the Aesthetic*. Oxford and Cambridge, Massachusetts, Basil Blackwell.

Eisenstein, S. (1986). *The Film Sense*. London, Faber and Faber Ltd. First published in 1943.

Feynman, R. P. (1996). Lectures on Computation. Edited by Hey, A. J. G. and Allen, R. W. London, Penguin Books.

Farouzan, B. (2003). *Foundations of Computer Science: From Data Manipulation to Theory of Computation*. Canada, Brooks/Cole - Thomson Learning.

Free Software Foundation (n.d.). http://www.fsf.org/ (website, accessed 3 January 2007).

Fuller, M. (2003). Behind *the Blip: Essays on the Culture of Software*. Brooklyn, New York, Autonomedia.

Fuller, M. (2005). 'Freaks of Number', in Cox, G. and Krysa, J. (eds). *Data Browser 02: Engineering Culture - On 'the Author as (Digital) Producer'.* London, Autonomedia, pp. 161-175.

Gamma, E., et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Upper Saddle River, New Jersey, Addison & Wesley.

Gere, C. (2002). *Digital Culture*. London, Reaktion Books.

Goriunova, O. and Shulgin, A. (2004). *read_me: Software Art & Cultures*. Denmark, Aarhus University Press.

Graham, P. (2003). 'Hackers and Painters'. http://www.paulgraham.com/hp.html (accessed 15 November 2006).

Hansen, M. (2004). *New Philosophy for New Media*. Cambridge, Massachusetts and London, MIT Press.

Heidegger, M. (1978). 'The Question Concerning Technology', in Krell, D. F. (ed). *Martin Heidegger: Basic Writings*. London, Thames & Hudson, pp. 284-317. First published 1953.

Horstmann, C. and Cornell, G. (2005). *Core Java (Vols. 1 & 2)*. Santa Clara, California, Sun Microsystems Press.

Huyssen, A. (1986) *After the Great Divide: Modernism, Mass Culture, Postmodernism*. Bloomington and Indianapolis, Indiana University Press

IPerG. (n.d.). *Integrated Project on Pervasive Gaming*. http://iperg.sics.se/ (accessed 25 November 2006).

Jaschko, S. (2003) 'Space-Time Correlations Focused in Film Objects and Interactive Video' in Shaw, J. and Weibel, P. (eds.) *Future Cinema: The Cinematic Imaginary After Film*. Cambridge, Massachusetts, ZKM and The MIT Press, pp. 430-435.

Johansson, T. D. (2004). 'Mise en Abyme in Software Art: A Comment to Florian Cramer', in Goriunova, O. and Shulgin, A. (eds). *read_me: Software Art & Cultures*. Arhus, Denmark, Aarhus University Press, pp. 150-159.

Kant, I. (1980). *Kant.* 'Great Books of the Western World' Series, edited by Adler, J. and Brockway, W. Chicago, University of Chicago Press (Encyclopedia Britannica, Inc.). First published 1790.

Kay, A. (2003). From video 'The History of the Personal Workstation' (1986), in Wardrip-Fruin, N. and Montfort, N. (eds). *New Media Reader*. Cambridge, Massachusetts, The MIT Press, pp. 393-404..

Kay, A. and Goldberg, A. (2003). 'Personal Dynamic Media', in Wardrip-Fruin, N. and Montfort, N. (eds). *The New Media Reader*. Cambridge, Massachusetts, The MIT Press, pp. 393-404. First published in 1977.

Kittler, F. (1999). 'On the Implementation of Knowledge - Towards a Theory of Hardware'. http://www.hydra.umu.edu/kittler/implement.html (accessed 15 January 2007).

Knuth, D. (1973 -1998). *The Art of Computer Programming.* Reading, Massachusetts, Addison-Wesley.

Krauss, R. (1999). *A Voyage on the North Sea: Art in the Age of the Post-Medium Condition*. London, Thames & Hudson.

Leigh, A. and Atkinson, R. D. (2001). 'Clear Thinking on the Digital Divide'. http://www.ppionline.org/documents/digital_divide.pdf (accessed 23 December 2006)

Levy, S. (1984). *Hackers: Heroes of the Computer Revolution*. New York, Anchor Press/Doubleday.

Lillemose, J. (2004). 'A Re-Declaration of Dependence - Software Art in a Cultural Context It Can't Get out of', in Goriunova, O. and Shulgin, A. (eds). *read_me: Software Art & Cultures*. Arhus, Denmark, University of Aarhus, pp. 136-149.

Macquarie (1992). *The Macquarie Dictionary* (Second Edition). Sydney, Macquarie Library Pty Ltd.

Malina, F. (1979). *Visual Art, Mathematics & Computers.* Oxford, Pergamon Press.

Manovich, L. (1996). 'The Death of Computer Art'. http://www.thenetnet.com/schmeb/schemb12.html (accessed 21 December 2006).

Manovich, L. (2001). *The Language of New Media*. Cambridge Massachusetts, MIT Press.

Manovich, L. (2005). 'Remixability & Modularity.' http://www.manovich.net/ (accessed 21 December 2006)

Manovich, L. (2006). 'Flash Generation', in Chun, W. H. K. and Keenan, T. (eds). *New Media/Old Media: A History and Theory Reader*. London & New York, Routledge, pp. 209-218.

Nettime mailing list (1999). www.nettime.org (accessed 12 December 2006).

Nietzsche, F. (1968). *The Will to Power*. New York, Vintage Books, Random House.

Norman, J. (2005). *From Gutenberg to the Internet: A Sourcebook on the History of Information Technology*. Novato, California, historyofscience.com.

Paul, C. (2003). *Digital Art.* London, Thames & Hudson.

Plato (c. 380 BC). *Ion*. http://classics.mit.edu/Plato/ion.html (accessed 6 January 2007).

Plato (1955). *The Republic*. Translated by D. P. Lee. London, Penguin Books. First Published c.360 BC.

Read_Me 1.2 festival jury (2002). 'Read_Me 1.2 Software Art/Software Art Games festival jury statement'. http://www.macros-center.ru/read_me/adden.htm (accessed 13 December 2006).

Read_Me 1.2 festival organisers (2002). 'Read_Me 1.2 Software Art/Software Art Games festival call for entries'. http://www.macros-center.ru/read_me/abouten.htm (accessed 13 December 2006).

Reichardt, J. (ed) (1971). *Cybernetics, Art and Ideas*. New York, New York Graphic Society.

Saint-Simon, H. (1975). *Henri Saint-Simon 1760-1825: Selected Writings on Science, Industry and Social Organisation*. Edited by K. Taylor. London, Croom Helm Ltd.

Salen, K. and Zimmerman, E. (2004). *Rules of Play: Game Design Fundamentals*. Cambridge Massachusetts, The MIT Press.

Selectparks (n.d.). [website]. http://www.selectparks.net (accessed 21 December 2006).

Shanken, E. A. (1998). 'The House that Jack Built: Jack Burnham's Concept of "Software" as a Metaphor for Art'. *Leonardo Electronic Almanac* 6(10). http://www.duke.edu/~giftwrap/House.html (accessed 23 July 2006).

Shanken, E. A. (2004). 'Art in the Information Age: Technology and Conceptual Art', in Corris, M. (ed). *Conceptual Art: Theory, Myth, Practice*. Cambridge, England, Cambridge University Press, pp. 235-250.

Small, P. (1996). Lingo *Sorcery: The Magic of Lists, Objects and Intelligent Agents*. Chichester, England, Wiley.

Stallman, R. (n.d). [website]. http://www.stallman.org (accessed 2 January 2007).

Stiegler, B. (1998). *Technics and Time, 1: The Fault of Epimetheus.* Stanford California, Stanford University Press.

Transmediale.01 Media Arts festival jury (2001). 'Transmediale.01 Media Arts festival jury

statement'. http://transmediale.de/01/de/s_juryStatement.htm (accessed 15 December 2006).

Transmediale.01 Media Arts festival organizers (2001). 'Call for Papers'. http://transmediale.de/01/de/s_juryStatement.htm (accessed 15 December 2006).

Tribe, M. and Reena, J. (2006) *New Media Art*. Edited by U. Grosenick. Koln, Germany, Taschen.

Turing, A. (1995). 'On Computable Numbers, with an Application to the Entscheidungsproblem' (extract), in Norman, M. (ed). *From Gutenberg to the Internet: A Sourcebook on the History of Information Technology*. Novato, California, historyofscience.com. First published in 1936.

Weber, M. (1946). *Max Weber: Essays in Sociology*. Edited by H. H. Gerth. and C. W. Mills. New York, Oxford University Press.

Weibel, P. and Druckrey, T. (eds) (2001). *Net Condition: Art and Global Media*. Cambridge, Massachusetts, ZKM and The MIT Press

Whitelaw, M. (2004). *Metacreation: Art and Artificial Life*. Cambridge, Massachusetts, MIT Press.

Wikipedia (nd). 'Constructivism'. http://en.wikipedia.org/wiki/Constructivism_(art) (accessed 23 December 2006).

Creative Works

Adobe (n.d). *Illustrator.* [Computer program].

Arcangel, C. (2003). *Super Mario Clouds*. http://www.beigerecords.com/cory/21c/21c.html (accessed 22 November 2006).

Banzi, M. (n.d.) *Arduino Microcontroller and Integrated Development Environment.* [computer hardware and software]. http://www.arduino.cc (accessed 22 December 2006).

Blast Theory (2004). *I Like Frank*. Adelaide, South Australia, Adelaide Fringe Festival. [new media work]. http://www.ilikefrank.com (accessed 16 December 2006).

Blender (n.d.). [computer program]. http://blender.org (accessed 12 December 2006).

Brooks, S. (2002). *Global City Front Page.* New York, Whitney Museum CODeDOC Exhibition. [software art work]. http://artport.whitney.org/commissions/codedoc/Brooks (accessed 12 December 2006).

Carmack, J. and Romero, J. (1993). *Doom*. [computer game].

Carmack, J. and Romero, J. (1996). *Quake*. [computer game].

Conway, J. (1970). *Game of Life*. First published in Gardner, M. mathematical games column, *Scientific American*, October 1970. [paper-based mathematical artificial life project].

Dagget, M. (2002). *Deskswap*. [software art work]. http://www.markdaggett.com/get.php?page=deskswap (accessed 23 December 2006).

Duchamp, M. (1915-23). *Large Glass – The Bride Stripped Bare by Her Bachelors, Even*.

Eclipse Foundation (2004). *Eclipse 3.0*. [computer program].

Frasca, G. (2003). *September 12, A Toy Story*. [computer game].

Fry, B. and Reas, C. (2001, continuing) *Processing*. [computer program] http://processing.org (accessed 3 December 2006).

Galloway, A. (2002). *What You See is What You Get*. New York, Whitney Museum CODeDOC Exhibition. [software art work]. http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Glick, R. and Voevodin, L. (1999-2006). *24Hr Panoramas*. [video art project].

IOD (1997). *Web Stalker*. London. [software art work]. http://bak.spc.org/iod/ (accessed 3 January 2007).

JODI (1998). *SOD*. [software art work]. http://sod.jodi.org/ (accessed 21 December 2006).

JODI (2002). Untitled Game. [software art work]. http://www.untitled-game.org (accessed 21 December 2006).

JOGL (Java Bindings to Open GL API). (n.d.). [3D library]. https://jogl.dev.java.net/ (accessed 18 December 2006).

Karhalev, E. and Khimin, I. (2002). *ScreenSaver*. [software art work]. http://www.404pro.com/desoft (accessed 16 December 2006).

Klima, J. (2002). *Jack and Jill*. New York, Whitney Museum CODeDOC Exhibition. [software art work]. http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Levin, G. (2000). *Audiovisual Environment Suite*. [computer program]. http://acg.media.mit.edu/people/golan/aves (accessed 2 January 2007).

Levin, G. (2002). *AxisApplet.* New York, Whitney Museum CODeDOC Exhibition. [software art work]. http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Lialina, O. (1996). *My Boyfriend Came Back from the War*. [net art work]. http://myboyfriendcamebackfromth.ewar.ru/ (accessed 16 December 2006).

Macromedia (n.d.). *Director.* [computer program].

Macromedia (n.d.). *Flash.* [computer program].

Maeda, J. (n.d.). *Design by Numbers*. Cambridge Massachusetts, MIT. [computer program].

Marker, C. (1983). *Sans Soleil*. [film].

McCoy, K. (2002). *Circler*. New York, Whitney Museum CODeDOC Exhibition. [software art work]. http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Napier, M. (1998). *Shredder*. [software art work]. http://www.potatoland.org/shredder/ (accessed 6 December 2006).

Napier, M. (2002). *SpringyDotsApplet*. New York, Whitney Museum CODeDOC Exhibition. [software art work]. http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Nimoy, J. (2002). *Textension*. [software art work]. http://www.jtnimoy.com/textension/ (accessed 2 January 2007).

The *Ogre* (n.d.). [3D library]. http://ogre3d.org (accessed 14 December 2006).

Paley, B. (2002). *CodeProfiles*. New York, Whitney Museum CODeDOC Exhibition. [software art work]. http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Pfeiffer, P. (2001). *The Long Count*, in Hansen, M. (2004). *New Philosophy for New Media*. Cambridge, Massachusetts and London, MIT Press, p. 30. [new media artwork].

Reinhart, M. and Widrich, V. (1992-2002). *tx-transform*, in Shaw, J. and Weibel, P. (eds). (2003). *Future Cinema: The Cinematic Imaginary After Film*. Cambridge, Massachusetts, ZKM and The MIT Press, pp. 442-443. [new media art work].

Sauter, J. and Lusebruk, D. (1995). 'Invisible Shape of Things Past', in Shaw, J. and Weibel, P. (eds.) (2003) *Future Cinema: The Cinematic Imaginary After Film*. Cambridge, Massachusetts, ZKM and The MIT Press, pp. 436-439 [new media art work]

Schmitt, A. *Vexation 1*. [software art work].http://www.gratin.org/as/ (accessed 21 December 2006).

Snibbe, S. (2002). *Tripolar*. New York, Whitney Museum CODeDOC Exhibition. [software art work] http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Stallman, R. (1975). *Emacs*. [computer program].

Sutherland, I. (1962). *Sketchpad*. [computer program].

Tatlin, V. (1920). *Monument to the Third International*. [sculpture/architectural design].

Utterbach, C. (2001-2). *Liquid Time*. in Shaw, J. and Weibel, P. (eds.) (2003) *Future Cinema: The Cinematic Imaginary After Film*. Cambridge, Massachusetts, ZKM and The MITPress, p.434. [new media art work].

Utterback, C. (2002). *Linescape*. New York, Whitney Museum CODeDOC Exhibition. [software art work].

http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Vertov, D. (1929). *Man with a Movie Camera*. [film]

Ward, A. (2001). *Signwave Auto-Illustrator*. [software art work].

http://www.gratin.org/as/ (accessed 12 November 2006).

Wattenberg, M. (2002). *ConnectApplet*. New York, Whitney Museum CODeDOC Exhibition. [software art work].

http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Wisniewski, M. (2002). *The Meaning of Life Expressed in Seven Lines of Code*. New York, Whitney Museum CODeDOC Exhibition.

http://artport.whitney.org/commissions/codedoc/index.shtml (accessed 6 December 2006).

Xith3D (n.d.). [3D library]. http://www.xith.org (accessed 14 December 2006).