$Id: concepts_notations_software_art.tex,v 1.1 2002/03/25 01:09:31 paragram Exp $

CONCEPTS, NOTATIONS, SOFTWARE, ART

FLORIAN CRAMER

SOFTWARE AND CONCEPT NOTATIONS

Software in the Arts. To date, critics and scholars in the arts and humanities have considered computers primarily as storage and display media, as something which transmits and reformats images, sound and typography. Reflection of the as such invisible layer of software is rare. Likewise, the term "digital art" has been associated primarily with digital images, music or audiovisual installations using digital technology. The software which controls the audio and the visuals is frequently neglected, working as a black box behind the scenes. "Interactive" room installations, for example, get perceived as a interactions of a viewer, an exhibition space and an image projection, not as systems running on code. This observation all the more applies to works in which it is not obvious at all that their production relied on programmation and computing. John Cage's 1981 radio play "Roaratorio", for example, appears to be a tape montage of a spoken text based on James Joyce's "Finnegans Wake", environmental sounds recorded in several cities of the world and Irish folk music, edited with analog recording technology. Yet at the same time it is an algorithmic artwork; the spoken text was extracted from the novel using a purely syntactical, formal method (mesostychs of the name "James Joyce"), and the montage was done according to a random score generated on a computer at the Parisian IRCAM studios. While the book-plus-CD set of "Roarotorio" documents the whole composition extensively, containing the audio piece itself, a recording and a reprint of John Cage's reading, a recording and a reprint of an interview, an inventory of the cities where sound was recorded, it includes the computer-generated score itself only in a one-page excerpt and nothing at all of the computer program code which generated the random score.[1]

The history of the digital and computer-aided arts could be told as a history of ignorance against programming and programmers. Computer programs get locked into black boxes, and programmers are frequently considered to be mere factota, coding slaves who execute other artist's concepts. Given

Date: March 23rd, 2002.

[1] [Cag82] — Regarding randomness generated with computers, the software artist Ulrike Gabriel says that it doesn't exist because the machine as a fact by itself is not accidental.

that software code *is* a conceptual notation, this is not without its own irony. In fact, it is a straight continuation of romanticist philosophy and its privileging of aisthesis (perception) over poeisis (construction),[2] cheapened into a restrained concept of art as only that what is tactile, audible and visible. The digital arts themselves participate in this accomplicity when they call themselves [new] "media art". There's nothing older than "new media", a term which is little more than a superficial justification for lumping together a bunch of largely unrelated technologies, such as analog video and computing, just because they were "new" at a particular time. If one defines as a medium something that it is between a sender and a receiver, then computers are not only media, but also senders and receivers which themselves are capable of writing and reading, interpreting and composing messages within the limitations of the rule sets inscribed into them. The computer programs for example which calculate the credit line of checking accounts or control medical instruments in an emergency station can't be meaningfully called "media". If at all, computer processes become "media" only by the virtue that computers can emulate any machine, including all technical media, and by the virtue of the analog interfaces which transform the digital zeros and ones into analog sound waves, video signals, print type and vice versa.

A Crash Course in Programming. A piece of software is a set of formal instructions, or, algorithms; it is a logical score put down in a code. It doesn't matter at all which particular sign system is used as long as it is a code, whether digital zeros and ones, the Latin alphabet, Morse code or, like in a processor chip, an exactly defined set of registers controlling discrete currents of electricity. If a piece of software is a score, is it then by definition an outline, a blueprint of an executed work?

Imagine a Dadaist poem which makes random variations of Hugo Ball's sound poem "Karawane" ("Caravan"):

> KARAWANE
> jolifanto bambla ô falli bambla
> grossiga m'pfa habla horem
> égiga goramen
> higo bloiko russula huju
> hollaka hollala
> anlogo bung
> blago bung
> blago bung

[2] A similar angle is taken in the paper "The Aesthetic of Generative Code" by Geoff Cox, Adrian Ward and Alex McLean, [CWM01]

bosso fataka
ü üü ü
schampa wulla wussa ólobo
hej taat gôrem
eschige zunbada
wulebu ssubudu uluw ssubudu
tumba ba-umpf
kusagauma
ba-umpf

The new Dada poem could simply consists of eight variations of the line "tumba ba-umpf". The author/performer could throw a coin twice for each line and, depending on the result, choose to write down either the word "tumba" or "ba-umpf", so that the result would look like:

tumba tumba
ba-umpf tumba
tumba ba-umpf
tumba ba-umpf
ba-umpf ba-umpf
ba-umpf tumba
tumba ba-umpf
tumba ba-umpf

The instruction code for this poem could be written as follows:

(1) Take a coin of any kind with two distinct sides.
(2) Repeat the following set of instructions eight times:
    (a) Repeat the following set of instructions twice:
        (i) Throw the coin.
        (ii) Catch it with your palm so that it lands on one side.
        (iii) If the coin shows the upper side, do the following:
            • Say "tumba"
        (iv) Else do the following:
            • Say "ba-umpf"
    (b) Make a brief pause to indicate the end of the line.
(3) Make a long pause to indicate the end of the poem.

Since these instructions are formal and precise enough to be as well executed by a machine (imagine this poem implemented into a modified cuckoo clock), they can be translated line by line into a computer program. Just as the above instruction looks different depending on the language it is written in, a computer program looks different depending on the programming

language used. Here I choose the popular language “Perl” whose basic instructions are rather simple to read:

```
for $lines (1 .. 8)
         {
         for $word (1 .. 2)
                  {
                  $random_number = int(rand(2));
                  if ($random_number == 0)
                           {
                           print "tumba"
                           }
                  else
                           {
                           print "ba-umpf"
                           }
                  print " "
                  }
         print "\n"
         }
```

The curly brackets enclose statement blocks executed under certain conditions, the $ prefix designates a variable which can store arbitrary letters or numbers, the “rand(2)” function generates a random value between 0 and 1.9, “int” rounds its result to either zero or one, “ ” stands for a blank, “ n” for a line break. This program can be run on virtually any computer; it is a simple piece of software. Complex pieces of software, such as computer operating systems or even computer games, differ from the above only in the complexity of their instructions. The control structures — variable assignments, loops, conditional statements — are similar in all programming languages.

Unlike in the instruction for throwing coins, the artists’ work is done once the code is written. A computer program is a blueprint and its execution at the same time. Like a pianola roll, it is a score performing itself. The artistic fascination of computer programming — and the perhaps ecstatic revelation of any first-time programmer — is the equivalence of architecture and building, the instant gratification given once the concept has been finished. Computer programming collapses, as it seems, the second and third of the three steps of concept, concept notation and execution.

Contrary to conventional data like digitized images, sound and text documents, the algorithmic instruction code allows a generative process. It uses computers for computation, not only as storage and transmission media. And this precisely distinguishes program code from non-algorithmic digital code, describing for example the difference between algorithmic composition on the one hand and audio CDs/mp3 files on the other, between algorithmically generated text and "hypertext" (a random access database model which as such doesn't require algorithmic computation at all), or between a graphical computer "demo" and a video tape. Although one can of course use computers without programming them, it is impossible not to use programs at all; the question only is who programs. There is, after all, no such thing as data without programs, and hence no digital arts without the software layers they either take for granted, or design themselves.

To discuss "software art" simply means to not take software for granted, but pay attention to how and by whom programs were written. If data doesn't exist without programs, it follows that the separation of processed "data" (like image and sound files) from "programs" is simply a convention. Instead, data could be directly embedded into the algorithms used for its transmission and output to external devices. Since a "digital photograph" for example is bit-mapped information algorithmically transformed into the electricity controlling a screen or printer, via algorithmic abstraction layers in the computer operating system, it follows that it could just as well be coded into a file which contains the whole transformation algorithms themselves so that the image would display itself even on a computer that provides no operating system.[3]

SOFTWARE ART

Executable Code in Art. If software is generally defined as executable formal instructions, logical scores, then the concept of software is by no means limited to formal instructions for computers. The first, English-language notation of the Dadaist poem qualifies as software just as much as the three notations in the Perl programming language. The instructions only have to meet the requirement of being executable by a human being as well as by a machine. A piano score, even a 19th century one, is software when its instruction code can be executed by a human pianist as well as on a player piano.

[3] I would not be surprised if in a near future the media industry would embed audiovisual data (like a musical recording) directly into proprietary one-chip hardware players to prevent digital copies.

The Perl code of the Dada poem can be read and executed even without running it on machines. So my argument is quite contrary to Friedrich Kittler's media theory according to which there is either no software at all or at least no software without the hardware it runs on:[4] If any algorithm can be executed mentally, as it was common before computers were invented, then of course software can exist and run without hardware. — A good example are programming handbooks. Although they chiefly consist of printed computer code, this code gets rarely ever executed on machines, but provides examples which readers follow intellectually, following the code listings step by step and computing them in their minds.

Instead of adapting Dadaist poetry as software, one could regard some historical Dadaist works as software right away; above all, Tristan Tzara's generic instruction for writing Dada poems by shuffling the words of a newspaper article[5]:

> To make a Dadaist poem:
> Take a newspaper.
> Take a pair of scissors.
> Choose an article as long as you are planning to make your poem. Cut out the article.
> Then cut out each of the words that make up this article and put them in a bag.
> Shake it gently.
> Then take out the scraps one after the other in the order in which they left the bag.
> Copy conscientiously.
> The poem will be like you.
> And here you are a writer, infinitely original and endowed with a sensibility that is charming though beyond the understanding of the vulgar.

The poem is effectively an algorithm, a piece of software which may as well be written as a computer program.[6]. If Tzara's process would be adapted as Perl or C code from the original French, it wouldn't be a transcription of something into software, but a transcription of non-machine software into machine software.

---

[4] [Kit91]

[5] [Tza75]

[6] My own Perl CGI adaption is available under `http://userpage.fu-berlin.de/{~}cantsin/permutations/tzara/poeme{\protect\T1\textunderscore}dadaiste.cgi`

Concept Art and Software Art. The question of what software is and how it relates to non-electronic contemporary art is at least thirty-two years old. In 1970, the art critic and theorist Jack Burnham curated an exhibition called "Software" at the Jewish Museum of New York which today is believed to be first show of concept art. It featured installations of US-American concept artists next installations of computer software Burnham found interesting, such as the first prototype of Ted Nelson's hypertext system "Xanadu". Concept art as an art "of which the material is 'concepts,' as the material of for ex. music is sound" (Henry Flynt's definition from 1961[7]) and software art as an art whose material is formal instruction code seem to have at least two things in common:

(1) the collapsing of concept notation and execution into one piece;
(2) the use of language; instructions in software art, concepts in concept art. Flynt observes: "Since 'concepts' are closely bound up with language, concept art is a kind of art of which the material is language".[8]

    It therefore is not accidental that the most examples of pre-electronic software art cited here are literary. Literature is a conceptual art in that is not bound to objects and sites, but only to language. The trouble the art world has with net.art because it does not display well in exhibition spaces is foreign to literature which always differentiated between an artwork and its material appearance.

    Since formal language is a language, software can be seen and read as a literature.[9]

If concepts become, to quote Flynt again, artistic"material", then concept art differs from other art in that it actually exposes concepts, putting their notations up front as the artwork proper. In analogy, software art in particular differs from software-based art in general in that it exposes its instructions and codedness. Since formal instructions are a subset of conceptual notations, software art is, formally, a subset of conceptual art.

My favorite example of both concept art in Flynt's sense and non-computer software art is La Monte Young's "Composition 1961", a piece of paper containing the written instruction "Draw a straight line and follow it". The instruction is unambiguous enough to be executed by a machine. At the same time, a thorough execution is physically impossible. So the reality of piece is mental, conceptual.

---

[7] [Fly61]

[8] ibid.

[9] But since formal language is only a small subset of language as a whole, conclusions drawn from observing software code can't be generally applied to all literature.

The same duplicity of concept notation and executable code exists in Sol LeWitt's 1971 "Plan for a Concept Art Book", a series of book pages giving the reader exact instructions to draw lines on them or strike out specific letters.[10] LeWitt's piece exemplifies that the art called concept art since the 1970s was by far not as rigorous as the older concept art of Henry Flynt, La Monte Young and Christer Hennix: While the "Composition 1961" is a concept notation creating an artwork that itself exists only as a concept, mentally, LeWitt's "Plan for a Concept Art Book" only is a concept notation of a material, graphic artwork. Unlike the concept art "of which the material is 'concepts"', LeWitt's piece belongs to a concept art that rather should be called a concept notation art or "blueprint art"; an art whose material is graphics and objects, but which was instead realized in the form of a score. By thus reducing its its own material complexity, the artwork appears to be "minimalist" rather than rigorously conceptualist.

A writing which writes itself, LeWitt's "Plan" could also be seen in a historical continuity of combinatory language speculations: From the permutational algorithms in the Sefer Jezirah and ecstatic Kabbalah to the medieval "ars" of Raimundus Lullus to 17th century permutational poetry and Mallarmé's "Livre". The combinatory most complex known permutation poem, Quirinus Kuhlmann's 1771 sonnet "Vom Wechsel menschlicher Sachen" consists of $13 * 12$ nouns can be arbitrarily shuffled so that they result in 10^{114} permutations of the text.[11] Kuhlmann's and La Monte Young's software arts meet in their aesthetic extremism; in an afterword, Kuhlmann claims that there are more permutations of his poem than grains of sand on the earth.[12] If such implications lurk in code, a formal analysis is not enough. Concept art potentially means terror of the concept, software art terror of the algorithm; a terror grounded in the simultaneity of minimalist concept notation and totalitarian execution, helped by the fact that software collapses the concept notation and execution in the single medium of instruction code. — Sade's "120 days of Sodom" could be read as a recursive programming of excess and its simultaneous reflection in the medium of prose.[13] The popularity of spamming and denial-of-service code in the contemporary digital arts is another practical proof of the perverse double-bind between software minimalism and self-inflation; the software art pieces awarded at the transmediale.02 festival, "tracenoizer" and "forkbomb.pl" also belong to this category.

---

[10] [Hon71], p. 132-140

[11] [Kuh71]

[12] ibid.

[13] As Abraham M. Moles noticed already in 1971, [Mol71], p. 124

La Monte Young's "Composition 1961" not only provokes to rethink what software and software art is. Being the first and still most elegant example of all artistic jamming and denial-of-service code, it also addresses the aesthetics and politics coded into instructions. Two years before Burnham's "Software" exhibition, the computer scientist Donald E. Knuth published the first volume of his famous textbook on computer programming, "The Art of Computer Programming".[14] Knuth's wording has adopted in what Steven Levy calls the hacker credo that "you can create art and beauty with computers".[15] It is telling that hackers, otherwise an avant-garde of a broad cultural understanding of digital technology, rehash a late-18th century classicist notion of art as beauty, rewriting it into a concept of digital art as inner beauty and elegance of code. But such aesthetic conservativism is widespread in engineering and hard-science cultures; fractal graphics are just one example of Neo-Pythagorean digital kitsch they promote. As a contemporary art, the aesthetics of software art includes ugliness and monstrosity just as much as beauty, not to mention plain dysfunctionality, pretension and political incorrectness.[16]

Above all, software art today no longer writes its programs out of nothing, but works within an abundance of available software code. This makes it distinct from works like Tzara's Dada poem which, all the while it addresses an abundance of mass media information, contaminates only the data, not its algorithm; the words become a collage, but the process remains a synthetic clean-room construct.

Since personal computers and the Internet became popular, software code in addition to data has come to circulate in abundance. One thus could say that contemporary software art operates in a postmodern condition in which it takes pre-existing software as material — reflecting, manipulating and recontextualizing it. The "mezangelle" writing of mez, an Australian net artist, for example uses software and protocol code as material for writings in a self-invented hybrid of English and pseudo-code. Her "net.wurks" are an unclean, broken software art; instead of constructing program code synthetically, they use readymade computations, take them apart and read their syntax as gendered semantics. In similar fashion, much software art plays with control parameters of software. Software artworks like Joan Leandre's "retroyou" and "Screen Saver" by Eldar Karhalev and Ivan Khimin

[14] knuth:art

[15] according Steven Levy [Lev84]; among those who explicitly subscribe to this is the German Chaos Computer Club with its annual "art and beauty workshop".

[16] which is why I think would be wrong to (a) restrict software art to only properly running code and (b) exclude, for political reasons, proprietary and other questionably licensed software from software art presentations.

are simply surprising, mind-challenging disconfigurations of commercial user software: a car racing game, the Microsoft Windows desktop interface. They manage to put their target software upside down although their interventions are technically simple and don't involve low-level programming at all.

Software Formalism vs. Software Culturalism. Much of what is discussed as contemporary software art and discourse on has its origin in two semi-coherent London-based groups. The older one around Matthew Fuller, Graham Harwood and the groups I/O/D and Mongrel is known, among others, for the experimental web browser "WebStalker", which instead of formatted pages displays their source code and link structures, the "Linker", a piece of "social software" (to use a term by Fuller) designed to empower non-literate users to design their own digital information systems, and "natural selection", a politically manipulated web search engine. Fuller also wrote a scrupulous cultural analysis of Microsoft Word's user interface and an essay with the programmatic title "Software as Culture". The other group involves the programmer-artists Adrian Ward (whose "Auto-Illustrator" won the transmediale.01 software art prize) and Alex McLean (whose "forkbomb.pl" won the transmediale.02 software art prize), the theoretician Geoff Cox and participants in the mailing list "eu-gene", the web site `http://www.generative.net` and the "DorkBot" gatherings in London (which involve poetry readings of program code). Both groups take exactly opposite standpoints to software art and software criticism: While Fuller/Harwood regard software as first of all a cultural, politically coded construct, the eu-gene group rather focuses on the formal poetics and aesthetics of software code and individual subjectivity expressed in algorithms.

If software art could be generally defined as an art

- of which the material is formal instruction code, and/or
- which addresses cultural concepts of software,

then each of their positions sides with exactly one of the two aspects. If Software Art would be reduced to only the first, one would risk ending up a with a neo-classicist 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. Formal reflections of software are, like in this text, inevitable if one considers common-sense notions of software a problem rather than a point of departure; histories of instruction codes in art

and investigations into the relationship of software, text and language still remain to be written.

REFERENCES

[Cag82] John Cage. *Roaratorio. Ein irischer Circus über Finnegans Wake*. Athenäum, Königstein/Taunus, 1982. 1

[CWM01] Geoff Cox, Adrian Ward, and Alex McLean. The Aesthetics of Generative Code, 2001. http://www.generative.net/papers/aesthetics/index.html. 2

[Fly61] Henry Flynt. Concept art. In La Monte Young and Jackson MacLow, editors, *An Anthology*. Young and MacLow, New York, 1963 (1961). 7

[Hon71] Klaus Honnef, editor. *Concept Art*. Phaidon, Köln, 1971). 8

[Kit91] Friedrich Kittler. There is no software, 1991. http://textz.gnutenberg.net/textz/kittler_friedrich_there_is_no_software.txt. 6

[Kuh71] Quirinus Kuhlmann. *Himmlische Libes=küsse*. ?, Jena, 1671. 8

[Lev84] Steven Levy. *Hackers*. Project Gutenberg, Champaign, IL, 1986 (1984). 9

[Mol71] Abraham A. Moles. *Kunst und Computer*. DuMont, Köln, 1973 (1971). 8

[Tza75] Tristan Tzara. Pour fair une poème dadaïste. In *Oeuvres complètes*. Gallimard, Paris, 1975. 6

C/O FREIE UNIVERSITÄT BERLIN, SEMINAR FÜR ALLGEMEINE UND VERGLEICHENDE LITERATURWISSENSCHAFT, HÜTTENWEG 9, D-14195 BERLIN, CANTSIN@ZEDAT.FU-BERLIN.DE, HTTP://USERPAGE.FU-BERLIN.DE/~CANTSIN

Florian Cramer, " Concepts, Notations, Software, Art ", 2002. Available: http://userpage.fuberlin.de/~cantsin/homepage/writings/software_art/concept_notations//concepts_notation s_software_art.pdf. Mise en Abyme in Software Art: A Comment to Florian Cramer " , in read_me Software Art & Cultures Edition Center for Digital AEstetik-forskning. Jan 2004. Troels Degn Johansson. Troels Degn Johansson, " Mise en Abyme in Software Art: A Comment to Florian Cramer ", in read_me Software Art & Cultures Edition, ed. Olga Goriunova & Alexei Shulgin, Center for Digital be called a concept notation art or â€œblueprint artâ€; an art whose material is not graphics. and objects, but which was instead realized in the form of a score.â€38 It is this idea of. LeWittâ€™s work taking the form of a concept notation instead of purely a concept, that I. would like to investigate more thoroughly. If we consider the work of LeWitt as a. notation of a concept, then it becomes more obvious that there is a direct relationship.Â Cramer makes a point in his essay â€œConcepts, Notations, Software, Art.â€46 Describing. how we make art, there are three steps that follow a logical progression in the. production of art they are; â€œConcept, Concept Notation and Executionâ€. He argues. that both conceptual art and code art, share an ability to combine the second and third.