Code Art Brutalism

low-level systems and simple programs

Simon Yuill, April 2004

A version of this essay was originally published in **read_me: Software Art and Cultures**, *edited by Olga Goriunova and Alexei Shulgin, Digital Aesthetics Research Centre: Aarhus, 2004.*

Introduction

fig.1.) le Corbusier, *Unite d'Habitation*, Marseille, 1952

Brutalism, more properly known as "New Brutalism" in its heyday, is arguably one of the most unpopular and least understood architectural styles of the 20th Century. It is mostly associated with rough-cast concrete buildings where its name is linked with the "beton brut" casting technique used by le Corbusier in the *Unite d'Habitation*, Marseille (1952) [fig. 1]. The term was adopted by young British architects of the 1950's, seeking a name to distinguish themselves from the prevalent style of their elders, one which harked back to traditional, pre-Modernist building styles [1]. Alison and Peter Smithson were its key proponents who described Brutalism as an ethic rather than an aesthetic. An attitude to how buildings and their materials should be used, it sought to re-think architectural practice through abandoning stylistic and metaphorical constructs. Brutalist designs often consciously exposed the basic, raw materials of industrial building, "demythologising concrete and recognising it for what it is." [2] Similarly, its proponents attempted to develop attitudes to urban planning that accepted the complexity and confusion of existing social situations rather than impose idealised models onto them [3].

The exposure of a raw materiality is prevalent in much current code art, such as JODI's *untitled game* and *JET SET WILLY Variations* [4], and Gameboy sound hacks [5] [fig. 2]. These works often bring the, normally hidden, basic materials from which digital works are made (code and data structures) into the foreground. Such works are often linked with, or perceived as, a form of nostalgia for "old skool" coding, but they are often as much an archaeology of the present, de-metaphorising interface-based software and reasserting the inherent aesthetics of pixels and code outwith any mimetic representational role. Yet these projects still accept a certain degree of interface as given, whether it is a Gameboy chassis or the familiar keyboard and monitor setup of the domestic PC. There is, however, a whole field of programming that deals much more fundamentally with the manipulation and execution of code by machine: assembly coding and the world of simple programs.

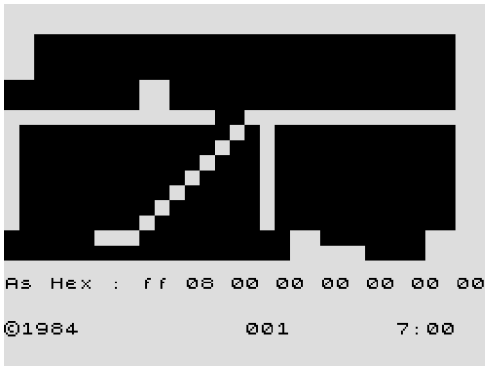

fig. 2a) JODI, *JET SET WILLY Variations*, 2003, interface

```
;
; Disassembly of the JSW engine
; under CP/M.
;
        .Z80    ;A:JSW.OVY 0000..FFFF

T4000   EQU     4000H
T4800   EQU     4800H
T481D   EQU     481DH
T481E   EQU     481EH
T4842   EQU     4842H
T4845   EQU     4845H
T4848   EQU     4848H
```

fig. 2a) JODI, *JET SET WILLY Variations*, 2003, assembly code

Assembly and simple programs

Assembly languages are a set of human-readable programming languages which relate closely to the mechanics of processor chips. Commands in assembly often refer directly to the physical actions of the processor, moving values in and out of memory locations, or turning the pins that connect processors to their host circuits on and off. Individual assembly commands can be related directly to the binary machine instructions, or opcodes, that the processor responds to, and many chip manufacturers provide tables that map specific binary values to assembly commands [fig. 3]. Because of this close proximity to the mechanics of the hardware, assembly languages are described as low-level languages in contrast to high-level languages, such as C or Java. These are more distant from the hardware and use commands that generally relate more to the human understanding of what a program does. Whilst low-level actions, such as bit operations, can be expressed in them, many high-level languages effectively clothe the machine operations in a set of metaphors which enable us to construct programs in accord with our own conceptual structures. Even basic functions such as subtraction and division are metaphorical expressions for particular machine actions which relate those actions to the entirely human concept of mathematics. The common conception of computers as essentially mathematical devices is, in this sense, inaccurate. It would be more precise to state that they are machines capable of manipulating binary patterns which simulate aspects of human mathematics. It is in those areas of coding that deal directly with this pattern-making process that we encounter a kind of Brutalism.

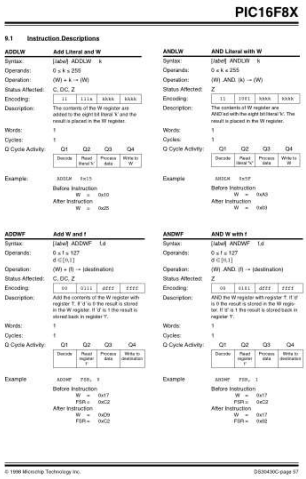

fig. 3a) PIC16F8X command specifications (extract)

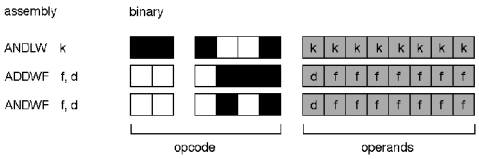

fig. 3b) assembly commands as binary patterns

This pattern-making process is often best expressed through "simple programs". Simple programs are those which perform a small set of basic operations on a set of constituent components producing some kind of output which may or may not have any functional value. A simple program applied to a lightbulb, for example, might take the form:

```
turn light on
wait 5 seconds
turn light off
wait 5 seconds
repeat
```

Indeed the most interesting simple programs are often those which have no obvious purpose yet which nevertheless exhibit particular distinctive behaviours. The physicist Stephen Wolfram has dedicated a large part of his life to the study of such programs, attempting to explain and catalogue their behaviours like a 19th Century plant collector [6]. One significant species of simple program that he has examined is that of cellular automata. A cellular automaton consists of a set of cells in a grid or line which may have two or more states. These are frequently represented as a series of squares which are coloured to indicate their current state. A typical program for a cellular automaton simply determines what state a cell should turn to given its current state and those of its neighbouring cells. Wolfram has catalogued a total of 256 such programs for just one particular type of cellular automata. He illustrates the rules of these programs in a simple graphical form with their output displayed on a grid in which the cells start in their initial states on the top line and show their changes over time as they move down the grid [fig.4].

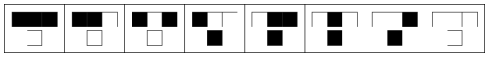

fig. 4a) diagram for linear cellular automata program
(copyright Stephen Wolfram, LLC, 2002)

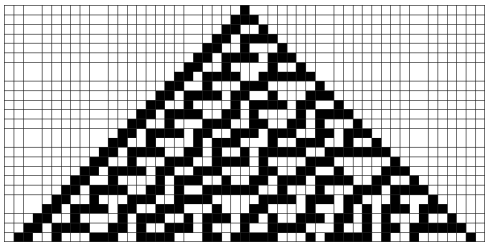

fig. 4b) output from program
(copyright Stephen Wolfram, LLC, 2002)

Cellular automata first developed as a past-time of the mathematician Stanislaw Ulam who, in the late 1940's, created games in which 2-D and 3-D structures were generated through simple rule systems [7]. John von Neumann adopted and formalised Ulam's games as a means of exploring the possibility of a self-reproducing machine. His work demonstrated that many mathematical and computational systems, such as Turing machines, could be simulated on them [8]. There are strong similarities between the operations performed in cellular automata and the basic instruction sets of assembly languages [9]. Wolfram has largely built on this work, simplifying von Neumann's automata and demonstrating that very complex behaviours can be generated from simple programs. Interestingly, many of the significant developments in cellular automata have emerged from purposeless play, such as Ulam's games and John Conway's life [10]. As Wolfram has stated, understanding of simple programs, such as the cellular automata, often develops out of a kind of "abstract aesthetic" interest in their innate properties and behaviours rather than functionally driven analyses [11].

HAKMEM

Distributed as an internal report in 1972 at the MIT AI lab, *HAKMEM* (short for "hack memo") is a classic compilation of early computer hacks, mostly designed for the then current PDP-10 system [12]. Its contributers include various luminaries of 1960's and early '70's computing such as Marvin Minsky, Bill Gosper and Richard Stallman. It includes mostly mathematical algorithms but also some plans for hardware devices, such as illegal radio packet transmitters, and a section called "Programming Hacks" which covers various tricks to produce visual and audio effects.

The PDP-10 (Programmed Data Processor model 10) was one of a series of mainframe machines produced by the Digital Equipment Corporation (DEC) in the 1960's and '70's and was popular with many of the computing research labs in the States at that time. The first distributed computer game, Spacewar!, was created for a PDP as were the first successful timesharing systems [13]. The PDP series were programmed in their own assembly language and many of the entries in *HAKMEM* are based on specific features of the PDP instruction set and binary system. Item 174, for example, points out that "21963283741 is the only number such that if you represent it on the PDP-10 as both an integer and a floating-point number, the bit patterns of the two representations are identical." In Item 154, Bill Gosper wryly explains how the particular form of binary representation used on a given hardware system can alter the results of certain types of calculations.

Many of the examples in *HAKMEM* are forms of "simple program", although often looser and more intuitive than Wolfram's cellular automata programs. One such example is Gosper's "display hack", Item 145, "proving that short programs are neither trivial nor exhausted." It uses just four lines of code, the second line of which can be substituted with various alternatives. The output produces "pretty pictures" on the display screen and can also be wired to a stereo amplifier to produce audible forms. Item 146 presents the *Munching Squares* algorithm, originally discovered by Jackson Wright in 1962. It consists of five lines of code which can produce a variety of different visual forms which relate closely to those of cellular automata [14] [fig. 5].

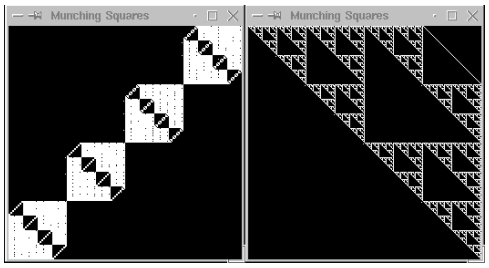

fig. 5) output from a *Munching Squares* program

Gosper was one of the pioneers of cellular automata research in the 1970's and implemented a version of Conway's life in the text editor program *TECO* [15]. *TECO* was a form of programmable text editor that ran on PDP's and has since been replaced by applications such as *Emacs* [16]. *TECO* made no distinction between content and program code, any text entered into it could be treated as a command. *TECO's* own command set was extremely concise, consisting largely of single character commands that often related to keyboard shortcuts. The following program takes a set of names, and lists them alphabetically according to lastname:

```
[1 J^P$L$$ J <.-Z; .,(S,$ -D .)FX1 @F^B $K :L I $ G1 L>$$
```

The ^P, for example, is the command for "sort" [17]. In many ways the syntax of regular expressions is a descendent of *TECO*, and such expressions can be seen as a form of "simple program" that operates on text rather than binary cell structures. Many codewurkers, such as Alan Sondheim and mez, as well as free-form ascii artists, use such devices to produce their work - a kind of raw "textual automata" [18].

PIC progs

Modern computer systems have grown so much in processing capability that the dexterity and genius that lay behind many of the *HAKMEM* examples is hard to appreciate. In many ways the descendents of this particular programming ethos are to be found not in mainstream software but in the work of microchip coders. Microchips provide basic processor systems on a single chip (such as are found in mobile phones and electronic toys) and many current processors are of similar capabilities as the early large-scale computer systems. The availability of cheap microchips like the PIC have made chip programming popular with hobbyists. A variety of high-level programming languages are available for microchip development, but the vast majority of programmers still use the native assembly languages of the actual chips. In many ways, a lot of these projects are even more reduced in material resources than those of the PDP developers, relying on breadboard circuits, LED lights and simple piezo-amplifiers. In terms of hardware and software these are the most stripped down, interface-bare realisations and therefore the most 'brutal' in terms of their raw simplicity.

slowcount.asm is a demo program that is distributed on the Yappa website. Yappa is a simple programming interface for PIC microchips developed by Mark Colclough [19]. In the world of microchip programming, the standard equivalent of "HelloWorld" is a simple program that controls an LED flashing on and off repeatedly. *slowcount.asm* is an extension of this designed for the PIC16f84 chip. The program runs a loop that treats the input/output pins of the chip as a binary counter, turning each pin on and off in representation of the current count. The chip is placed in a circuit in which the 8 of the pins can be connected to LED lights and a small audio speaker. The LEDs flash at different rates relating to the binary value they represent. Those representing the most significant bits being slowest, whilst the least significant bits, which change at a faster rate, produce an audible hum when connected to the speaker [fig. 6]. It has strong parallels with some of the sound and video hacks for the PDP-10. *HAKMEM* Item 168, for example, provides a program that sets up an endless loop of incremental bit operations, and suggests connecting speakers to the output pins and listening to "the square waves from the low bits of [pin] 0".

Wolfram's work with cellular automata is largely based around automata which expand infinitely as they progress but he has also looked into fixed-width automata which use the same number of cells throughout. *slowcount.asm* can be understood as a form of fixed-width automata and the code could be easily changed to express other fixed-width automata programs [20]. Fundamentally it is a raw expression of assembly code behaviour, and, like the *Munching Squares* algorithm, demonstrates the congruity of binary processor mechanics and cellular automata - both being forms of "simple program" and neither being developed for any purposeful mathematical exercise.

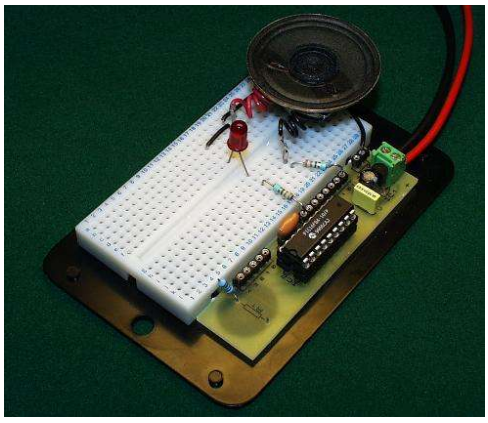

fig. 6a) *slowcount.asm* circuit

fig. 6b) binary patterns from *slowcount.asm* output, show start and end of sequence, a black square indicates an LED is on, white indicates that it is off

ap

Rather than designing one-off buildings, many Brutalist architects were interested in creating habitats, adaptable social complexes that were capable of supporting small communities. Examples include the Ivor Smith and Jack Lynn's *Park Hill* estate in Sheffield, Kunio Mayekawa's *Harumi* apartment block, Tokyo, and Moshe Safdie's *Habitat*, Toronto [21]. ap have taken a similar approach in their creation of an entirely new form of operating system, one which could be described more in terms of a habitat for code. Echoing the the Smithsons, ap argue that existing operating systems are restricted by "antiquated tool metaphors, and the limiting notions/divisions of system, user and programmer," and that instead data generation should be liberated from such imposed models [22].

Like the Brutalist habitats, ap projects emphasize modularity and topology as the principle design factors. The Smithsons cited the traditional Japanese house as a model, capable of changing the internal structure and function of its rooms over the course of a day [23]. The notion of modular design had developed through 20th Century architecture from its adaption of industrial process and in the 1950's and '60's was a key interest spreading from Brutalists to the Pop architecture of Archigram [24]. In Cedric Price's *Fun Palace* design the only permanent features were a skeletal grid and set of cranes which could place wall and floor components into an endless variety of structures, an approach which was applied to larger urban systems in many Archigram proposals [25]. The basic UNIX architecture is constructed on a modular principle that can facilitate temporary connections between different components to suit users' needs. The overall function of these is expressed in the topology of the pipelines and process forks that connect them. The ap OS takes this further, constructed as an environment of multi-purpose units whose individual behaviour is often determined by their connections to other units - depending on what types of input each provides. As the units have the capability to autonomously reconnect themselves to others, the behaviour of the system may constantly fluctuate in accord with its topological mutation. These units often operate as simple programs processing data for their own sake, as an expression of their inner structure rather than according to a purposeful end. In contrast to UNIX, which is a modular architecture of software applications, ap propose modular software that seeks, and perhaps denies, its own applicability.

In *ap0202* each unit runs as a small scale virtual machine that provides a basic series of bitshifting operations. As with *TECO*, the system makes no distinction between content data and program code. All input is treated as patterns of binary data which are processed according to simple programs analogous to those of cellular automata and forms of genetic splicing and mutation. Incoming patterns that match the instruction set for a given unit effectively reprogram it, thereby causing new behaviours to emerge within the system in an autopoeitic fashion. There are parallels with the Smithson's notion of cluster compositions: "a closenit, complicated, often-moving aggregation." [26]

One of the greatest problems of many of the Brutalist habitats was that they aspired, yet failed, to be complete self-contained environments that met their inhabitants' needs - insular, artificial paradises which often became urban hells [27]. The ap OS is, by contrast, open and "promiscuous" with its external environment [28]. Specialised units can take input from various sources, such as video cameras, sound and other sensors, all of which are potentially capable of reprogramming the units that receive their data. In the "self display devices" of *ap0201*, the system has been transfered from virtual machines running on desktop computers to microchips. A set of solar-powered versions of the device are currently installed at a location in the Mojave desert, California. One of the main aims of this project is to determine what new forms of program are generated in response to the extreme conditions of that environment [fig. 7].

Brutalism was not only a challenge to the conservative "garden city" suburbanism of post-war Britain. For practitioners, such as the Smithsons, whilst it followed on from Modernism, Brutalism also reacted against what they considered to be the over stylised "Machine Aesthetic" of Modernist design. This metaphorised buildings as ships and engines, subjecting materials, such as concrete, to highly refined finishing and rendering processes that suppressed their innate qualities in favour of sleek stylistic statements. There is a similar attitude underlying ap's raw data processing, their stated desire to avoid metaphorisation of this, and its contrast to the highly stylised "information design" of John Maeda and his students. In their move away from conventional computer media and the rehashing of existing interfaces, they also move beyond the dysfunctionalist rhetoric of much recent software art. They do not so much upset our expectations of what software should do as reveal that those expectations are still highly constrained.

fig. 7) *ap0201*, Mojave desert, 2004

Conclusion

Brutalist architecture was possibly more successful in its principles than its realisation. Despite the desire to abandon aesthetics, a distinctive aesthetic nevertheless emerges from Brutalist designs. At its worst this is merely a stylistic gesture, but at its best it relates the deployment of its materials back to something of Brutalism's ethical ambitions, emphasising both the materiality and contingency of constructed form. There is also something inherently 'difficult' about such works, they almost defy you to like them. Their ungainly rawness suggests an incompleteness rather than finality of design, and this is perhaps why a similar aesthetic/ethic is at play in so much classic hacker code and low-level programming. The examples discussed here are mostly neither complicated nor especially 'elegant' pieces of code and their aesthetic value, in many cases, arises directly from their lack of aesthetic intent.

A Brutalist approach to software demythologises code and recognises it for what it is. High-level programming approaches can be very successful in achieving certain ends, but the very imposition of higher-level constructs and metaphors also limits awareness of how code operates in and for itself and what may be achieved through that. Arguably it is the changes in low-level systems that have provoked the biggest paradigm shifts, such as the development of binary computation and Turing machines, and such as Wolfram is suggesting will be the case in fully understanding simple programs. What emerges from the Mojave desert may be a new form of software culture, or it may just be meaningless data, but what is most important is the underlying attitude which has enabled it.

Endnotes

1. Reyner Banham, **The New Brutalism: Ethic or Aesthetic?**, Architectural Press: London, 1966, p.10

2. Banham, ibid., p.17

3. Banham, ibid., pp.71 - 72

4. http://www.untitled-game.org, http://jetsetwilly.jodi.org

5. such as *Nanoloop*, http://www.nanoloop.de, and *Little Sound DJ*, http://www.littlesounddj.com

6. Stephen Wolfram, **A New Kind of Science**, Wolfram Media: Champaign, 2002

7. http://www.brunel.ac.uk/depts/AI/alife/al-ca.htm

8. John von Neumann, "Theory and Organization of Complicated Automata," in A.W. Burks (editor), **Theory of Self-Reproducing Automata**, University of Illinois Press: Urbana 1949, pp.29-8&, see also Wolfram, op. cit.

9. a range of different parallels are examined in Wolfram, *op. cit*.

10. Martin Gardner, "The fantastic combinations of John Conway's new solitaire game 'life'," in **Scientific American** 223, October 1970, pp.120-123

11. Wolfram, *ibid*., p.109

12. http://home.pipeline.com/~hbaker1/hakmem/hakmem.html

13. http://www.brouhaha.com/~eric/retrocomputing/pdp-10

14. its relationship to cellular automata is discussed in Eric W. Weisstein, "Munching Squares" from **MathWorld–A Wolfram Web Resource**, http://mathworld.wolfram.com/MunchingSquares.html

15. Eric Raymond, "life", **The Jargon File**, http://www.catb.org/~esr/jargon/html/L/life.html

16. Eric Raymond, "TECO", **The Jargon File**, http://www.catb.org/~esr/jargon/html/T/TECO.html

17. this is equivalent to the 'control - p' combination on the keyboard. The program itself is a bit more complex than described, see Raymond, *ibid*.

18. a repository of such work is maintained by Florian Cramer through his unstable digest postings to the nettime mailing list, http://amsterdam.nettime.org

19. http://www.cm.ph.bham.ac.uk/software/yappa/

20. see Wolfram, *op cit*., pp. 255 - 260, for an analysis of fixed width automata

21. for Park Hill see Banham, *op cit.*, pp.41 - 43, for Harumi, Banham, *op cit*., p.131, and for Habitat, Moshe Safdie, **Beyond Habitat**, MIT Press: Massachusetts, 1970

22. http://www.1010.co.uk

23. Banham, *op. cit*., p.45

24. Banham, *op. cit*., p. 18, Peter Cook, *et al*., **A Guide to Archigram 1961 - 1974**, Academy Editions: London, 1994

25. Banham, *op. cit*., p.43, and, Cedric Price, **Works II**, Architectural Association: London, 1984, pp.56 - 61

26. the Smithsons, quoted in Banham, *op. cit*., p.73

27. *Park Hill* was a disaster, Safdie's *Habitat*, however, has been highly successful, this may be in large part to the fact that the inhabitants are heavily involved in running and maintaining the building as a community

28. ap's own term for it, as in systems which are both open to all forms of incoming data and actively seeking other available data elsewhere on the network