Programming with a Game Pad

Dave Griffiths

Introduction

In recent years »live coding« has emerged as a practice of improvised musical and visual performance. Watching programming seems an unlikely thing to do whilst listening to music or watching a performance, but it's proving popular as a way of reconnecting the computer performer with their audience.

Game interfaces can also provide flexible new ways of interacting with software, and the rich metaphors, techniques and hardware developed for game playing are waiting to be used for new purposes.

Two new live coding languages, »Betablocker« and »Al Jazar«, have been developed to bring together these ideas to make live coding more accessible and moving it away from the realms of traditional programming.

Live Coding

Live coding is generally understood to be writing code in front of an audience, and has partly come about as a reaction to existing laptop performance, which has a tendency to hide the performers actions away, leading to a disconnection of performer and their audience. Such a disconnection is apparent in contrast to performances with traditional instruments.

For this reason, the projecting of screens during performances is an important aspect of live coding. It must be stressed that, although programming code is projected, it is considered enough that the audience understand that something is being created in front of them, and relate the changes in the output with the changes of the structure of the visible code – a literal understanding of the code is not important.

Live Coding History

The essence of live coding is rule writing and modification at the same time as the rules being carried out. This doesn't necessarily have to be realized using a computer, and some offshoots of live coding research are dealing with the possibilities of rule based choreography acted out by human participants capable of rewriting their own rules.[1]

Some of the precedent for live coding comes from experiments of this nature in the 60s and 70s (and probably long before that). The first recorded live coding on computer was done by Ron Kuivila in 1985 at STEIM. Another group live coding at this time were the Hub[2] who programmed Forth during live performances and encouraged audience members to look at what they were doing as part of the performance.

TOPLAP (see below) is always looking for early references to live coding performance, so let them know if you know of any similar work.

---

1 Non-computer live coding: http://www.toplap.org/index.php/Live_Coding_Without_Computers

2 The Hub: http://hub.artifact.com/

We now have to jump nearly 20 years to find the next forms of live coding, as computer technology enabled the rise of portable laptops with CPUs powerful enough to process audio or video. The audio synthesiser software »SuperCollider« brought a high level language with sound processing capability to the fore, and gigs around 2000 by James Mc Cartney, Julian Rohrhuber, Nick Collins and Fabrice Mogini started to incorporate elements of code improvisation.

At the same time, London based group »slub« were introducing elements of live coding using home made terminal based networked software for collaborative music making. On the other side of the Atlantic, Ge Wang and Perry Cook were developing »ChucK«, a highly malleable language and environment for »on the fly« audio and graphics programming.

TOPLAP

According to the official histories, at 1am on Sunday 15th February 2004, in a smoky Hamburg bar some members of this embryonic live coding community formed TOPLAP[3] (a Temporary Organisation for the Promotion of Live Algorithm Programming) in order to promote and cement the ideas of live coding. Later that day, on a Ryanair transit bus from Hamburg to Lübeck, the TOPLAP manifesto was born:

We the digitally oversigned demand:

- Give us access to the performer's mind, to the whole human instrument.
- Obscurantism is dangerous. Show us your screens.
- Programs are instruments that can change themselves
- The program is to be transcended – Language is the way.
- Code should be seen as well as heard, underlying algorithms viewed as well as their visual outcome.
- Live coding is not about tools. Algorithms are thoughts. Chainsaws are tools. That's why algorithms are sometimes harder to notice than chainsaws.

We recognize continuums of interaction and profundity, but prefer:

- Insight into algorithms
- The skillful extemporization of algorithm as an impressive display of mental dexterity
- No backup (minidisc, DVD, safety net computer)

We acknowledge that:

- It is not necessary for a lay audience to understand the code to appreciate it, much as it is not necessary to know how to play guitar in order to appreciate watching a guitar performance.
- Live coding may be accompanied by an impressive display of manual dexterity and the glorification of the typing interface.
- Performance involves continuums of interaction, covering perhaps the scope of controls with respect to the parameter space of the

3 TOPLAP: http://www.toplap.org/index.php/Main_Page

> artwork, or gestural content, particularly directness of expressive detail. Whilst the traditional haptic rate timing deviations of expressivity in instrumental music are not approximated in code, why repeat the past? No doubt the writing of code and expression of thought will develop its own nuances and customs.

Performances and events closely meeting these manifesto conditions may apply for TOPLAP approval and seal.

A website and mailing list were founded and the fledgling community of live coders began. Now live coding had become more clearly defined with some core values.

Programming as thought process

One of the interesting elements to have arisen from the TOPLAP manifesto is contained within the lines:

»The program is to be transcended – Language is the way.« and »Live coding is not about tools. Algorithms are thoughts. Chainsaws are tools. That's why algorithms are sometimes harder to notice than chainsaws.«

Which interestingly, draws parallels with Abelson and Sussman's famous textbook, »The Structure and Interpretation of Computer Programming«: »Programs must be written for people to read, and only incidentally for machines to execute.« (Abelson/Sussman #: #)

This highlights an often misunderstood facet of programming, that it doesn't directly concern computers at all, but is mainly used as a communication tool for the programmer to understand what they are constructing, in order to debug and reuse their work later – or for other programmers to pick up, use and understand.

Programming languages also communicate something more subtle, which is a way of thinking, or a space for solving problems. Different programming languages offer different philosophies, or solutions to the problem of solving problems, with broad categories like »object orientation«, »functional«, »imperative« or »declarative«, and many subcategories and styles making up the range of languages available today. This goes some way to explaining why programmers get very attached to languages they know – as it can be difficult to think about problems in a different way, after the many years required to learn a language well.

Live coding languages are generally »high level languages«. These closely exhibit Abelson and Sussman's observation as they are designed to be easy for a human to use, rather than a computer to run. Due to the nature of live coding, these languages also tend to be run in specially designed environments, or editing applications, which allow the code to be running at all times, with edits being incorporated into the running program in various ways (so as not to interrupt the flow of the process). Here are some example environments and the languages they use:

Environment	Medium	Language
SuperCollider	Primarily music, some visuals	Specially designed language based on Smalltalk and C
Impromptu	Primarily music, some visuals	Scheme
Feedback.pl	Music	Perl
ChucK	Music and visuals	Specially designed language »ChucK«
Fluxus	Primarily visuals	Scheme
Thingee	Visuals	Specially designed language based on lingo
Quoth	Music	Natural langauge parsing
Pure Events	Music	Specially designed language

As shown above, a lot of live coding environments are neither wholly restricted to audio or visual, and this exposes an interesting feature of live coding – it tends to blur the distinctions between different media. A good musical live coder will find it easy to transfer their ability to write music generating code to that of movements of shapes and colours. In the same way, a visual live coder will be tempted to try their hand at triggering sounds instead of visual events.

In a deeper way, the exact same code can be used to simultaneously control the audio and visual, leading to exciting new possibilities in integrating and controlling the two in an improvised manner during a performance.

Malleability of code

The structure and interpretation of computer programs also contain this quote from John Locke in »An Essay Concerning Human Understanding« (1690):

»The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.«

One of the advantages of programming as a performance is that it is possible to quickly make sweeping changes to the output during live coding. This is about the expressivity of a language.

Programming languages allow you to »abstract« ideas into components which are shared between different sections of the code. Changes to core components are reflected everywhere simultaneously, amplifying the effect of the programmer in ways they can control.

The simplest example of this is the use of a global variable. A variable is a way of naming a value, for instance a programmer can in one place define 3 to the name »size«. The programmer has abstracted 3 with the concept of size and given it a meaning. The size variable maybe referenced in many places, so changing it's value at the one definition later on will take effect everywhere it has been used.

This kind of abstraction can be applied to processes too – where sequences of operations can be named and reused in many places. Once we start layering these abstractions on top of each other, we get a description of a process which is very easy to control and change in predicable and expressive ways. This is one way a live coder can keep the pace needed for an engaging performance.

Fluxus and live coding in Scheme

The live coding language/environments described later on are written in fluxus. This is potentially confusing, as fluxus is itself a live coding environment in its own right.

As mentioned above, fluxus is programmed in Scheme, which is a language dating back to 1975 when it was invented by Jerald J. Sussman and Guy L. Steel Jr. Scheme is a dialect of »Lisp« which is one of the first programming languages to be invented, and dates back to the 1950's.

Fluxus is, in essence a 3D graphics engine, similar to one you would find inside a 3D computer game. It extends the language of Scheme with commands for describing three-dimensional scenes to a computer's graphics card, and contains features normally found in modern computer games, such as a selection of geometry types, the ability to control surface appearance with textures or hardware shaders, and rigid body dynamics for physics simulations. Fluxus also includes functionality for audio input (for controlling animations with sound) and network IO using the popular Open Sound Control format. It is free software, available under the GLPv2 licence for Linux and OSX. Version 0.1 was released Tuesday, August 5th 2003 at 17:29.

Fluxus also comes with a live coding interpreter, utilising PLT Scheme's[4] mzscheme – allowing you to see the code »float« above the graphics as you create them. This makes fluxus useful as a rapid prototyping tool for learning or playing with 3D animation.

The main focus of fluxus's interface is for live coding though, and Scheme is a good candidate for a live coding language. It has a very concise and elegant style, meaning that it takes very little effort (in terms of key presses or amount of text) to come up with interesting results.

```
(define (render) ; makes a function called "render"
        (scale (gh 1) (gh 3) (gh 9)); scale using harmonic levels from
the sound input
        (draw-cube)); draw a cube - which takes on the scale set above

(every-frame (render)); every frame call render
```

4 PLT Scheme: http://www.plt-scheme.org/

An example fluxus script to squash and stretch a cube in time to the music.

Keyboards

The TOPLAP manifesto talks of »glorification of the typing interface«, and clearly for a programmer, text editors and keyboards are of fundamental importance. However, keyboards and other peripheral devices commonly used by programmers are responsible for physical ailments such as RSI and Carpal tunnel syndrome. Should we really be glorifying something which can do us harm in this way – or should we be looking for something different?

Visually programmed, largely mouse controlled languages such as Pure Data and MAX/MSP are valuable to people who are not so attached to their text editor. Some recent developments in visual programming languages are making them possible for live coding[5] but still the mouse and keyboard is the prevalent input method.

The languages we are using for live coding are not generally designed for purpose, and while employing languages meant for slow careful construction of software for ad hoc improvisation adds to the spectacle of live coding – it seems we could do better.

Programming games

Computer game input devices provide readily accessible hardware for experimentation. Also, programming environments which use the visual language of computer games allows live coding to appeal to wider audiences. There are some precedents for programming forming part or the entirety of a game play mechanic. I will cover two such examples here.

Corewars

Corewars[6] is a game where player/programmers write programs which attempt to take control of as much of the memory of a virtual machine (the core) as possible. A virtual machine is a program which mimics another (possibly fictional or real) device. They are wholly contained within the program which »virtualizes« them, and so are safe places to do things which couldn't be done with a real hardware device.

The core war programs written by players take the form of assembly language code (one such langauge is called »Redcode«), and have the ability to write and modify themselves and other programs in the quest to copy themselves over as much memory as possible. The simplest redcode »warrior program« is called the »Imp«:

MOV 0, 1

---

5 Desire Data: https://devel.goto10.org/desiredata is a variant of Pure Data which includes support for live coding. Robert Atwood (http://robert.lurk.org) was the first person to use it in a live coding performance at the LOSS livecode festival 2007, http://livecode.access-space.org/.

6 Core wars homepage: http://www.corewars.org/

This program simply copies itself to the next instruction (redcode addresses relatively, so »1« means the address after the current one), which is then run, copying itself until memory is used up. More sophisticated programs will »bomb« memory in calculated patterns, or hijack each others code for parasitic strategies.

Carnage Heart

Carnage Heart[7] is a Japanese Playstation game developed in 1995. The player has to program robots for battle using a visual language akin to a flow diagram. In this way, the robots (known as »OverKill Engines«) cannot be controlled once battle has been started – the player has defined the behavior and ultimate success of the robot entirely in their program.

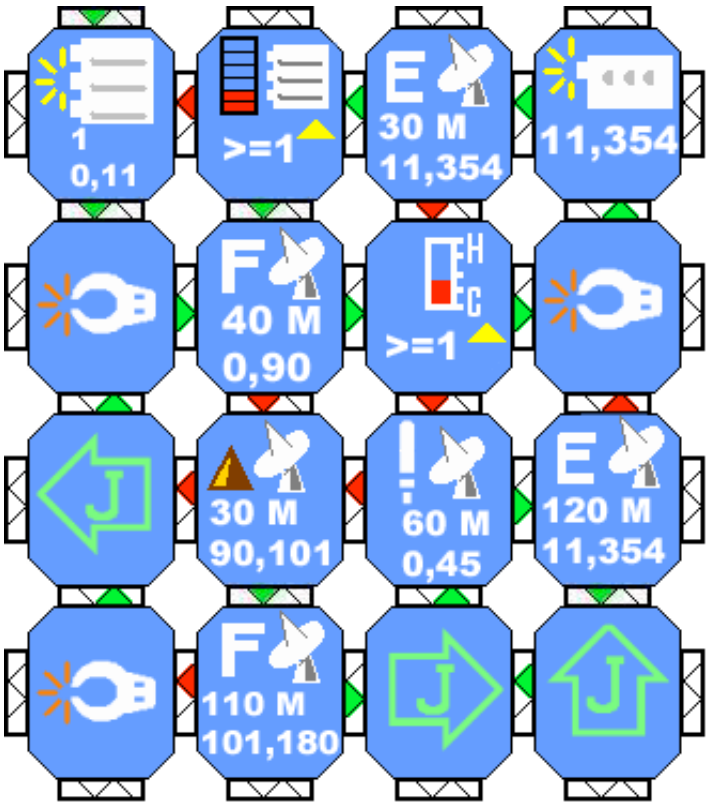

Figure 1: A carnage heart OKE program, courtesy of Wikipedia

7 Carnage heart fan site: http://www.carnageheartdepot.blogspot.com/

Gamepad live coding

Taking these and other games as influence, Betablocker and Al Jazari are languages written in Scheme for fluxus which can be programmed using a game pad only – no other input device is required during performance. This in itself is an interesting restriction, and requires a specially designed language and a usable interface mechanic to allow code to be written. These projects also focus on the visual aspect of live coding, and make more of an attempt to make the code interesting to the audience than simply a text display would.

Ring menus

These interface elements are key to making game pad live coding possible. They allow items to be selected from a large range of options, and also utilise »muscle memory« in remembering the direction of different selections. They only need the use of one analogue stick, which can be used both to activate and operate the menu. Game pad shoulder buttons are also used to determine and switch between different menu types.

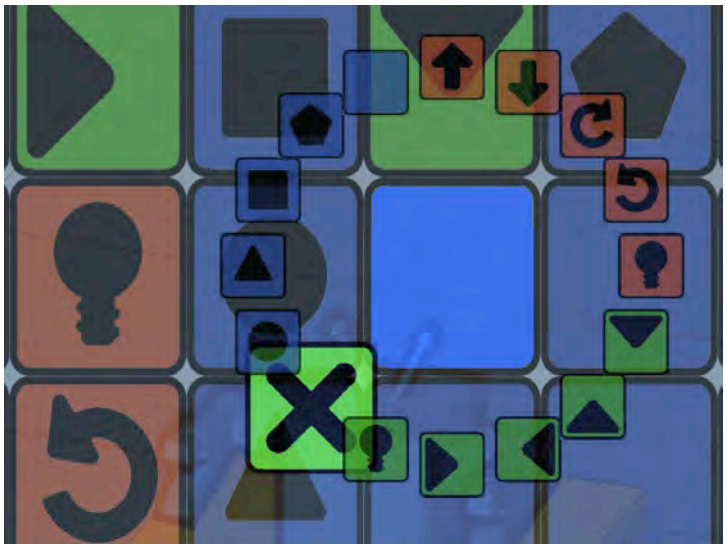

Figure 2: Al Jazari's programming ring menu

Betablocker

Betablocker was the first attempt at a game pad programmable live coding system in fluxus. It was inspired by a discussion on the TOPLAP mailing list about virtual machines, and also visually by games such as »Mr Driller«, in terms of colorful blocks interacting with each other and setting up chain reactions as a game mechanic.

The program is a virtual machine running inside fluxus. It visualizes the simulation of a fictional CPU with 256 bytes of memory, and allows multiple threads of execution to be run sharing the same address space.

The Betablocker language is very much based on the ideas in core wars, but Betablocker also visualizes the threads of execution. This is different to other live coding language/environments, as the process itself as well as the code and entire contents of memory are visible.

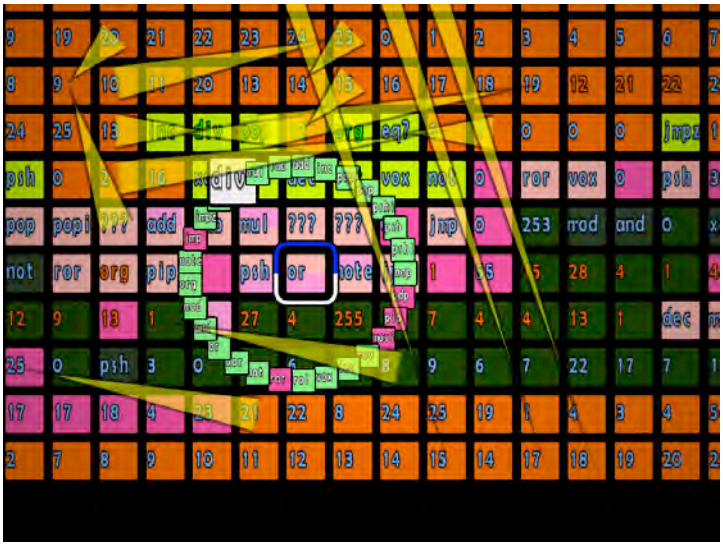

Figure 3: A betablocker screenshot

Betablocker programs never crash – this means that while the virtual machine is running, they will never stop executing. Crashes and exceptions are important in most situations as they prevent unintended code from being run, which is usually essential, preventing dangerous things from happening. Uncrashable languages are used in situations where this isn't important, usually safely as virtual machines – for applications such as genetic programming. To make this possible, instructions are able to return arbitrary values for invalid input (divide by zero, or referencing non-existent memory) so programs keep on running even if an error has been detected.

A performance of Betablocker consists of writing instructions and data to the memory cells with a game pad, which is used to navigate memory with the direction buttons, and select instructions with a ring menu using the analogue sticks. Instructions are included to trigger sounds, and change instruments for each thread. Small pre-written library code segments can also be loaded in and pasted over memory using more ring menus.

As programs run, they can write over each other, or modify themselves. Although Betablocker is deterministic (there are no calls to random functions) this generally results in a chaotic situation after a few threads of instructions have begun running. As they will never crash, these threads will run off the end of their programs if they do not loop, and run through memory executing anything they find.

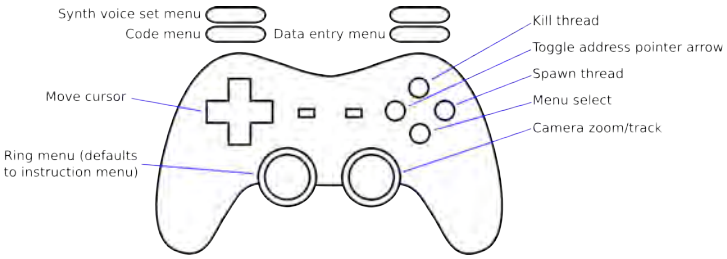

Figure 4: The gamepad layout for betablocker

Al Jazari

Ibn Ismail ibn al-Razzaz al-Jazari[8] was an influential scholar and engineer who lived at the beginning of the 13th century. Along with inventing and making detailed plans for many currently used mechanical devices (the crankshaft, mechanical clocks, combination locks, segmental gears and valves to name a few) he also worked on plans for automatons and humanoid robots. The robots he designed were intended to be used for playing music at royal drinking parties.

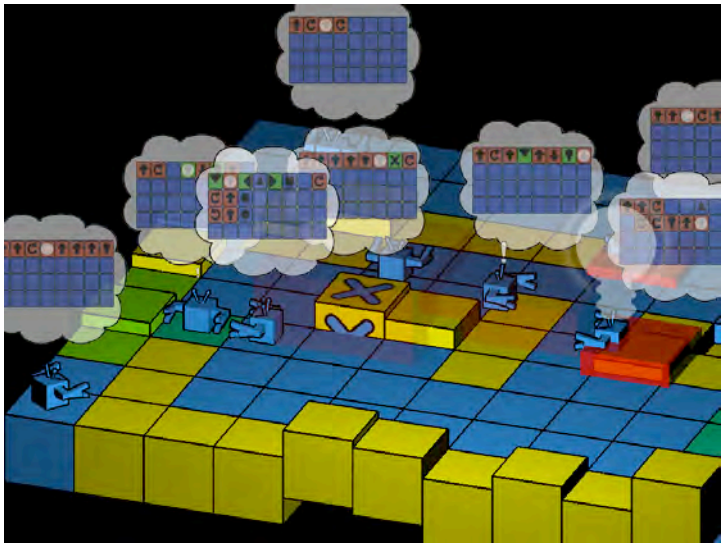

Figure 5: Al Jazari in action

This project was inspired and named after Al Jazari and the idea of live coding robots for royal drinking parties. It also has elements of the computer game »The Sims« – the idea of instructions and states visible floating above a characters head. Also Gullibloon's »Army of Darkness«[9],

8 More on Al-Jazari here: http://en.wikipedia.org/wiki/Al-Jazari

9 The army of darkness http://gullibloon.org/mediawiki/index.php/Gbot

a robotic installation piece, where robots chaotically explore an environment which happens to be populated by strategically positioned electric guitars.

Al Jazari was developed for audiences who may not be expecting to witness live coding. It was premièred and developed during a series of events called »Ravage me Savagely« at a pub in London's New Cross, in which laptop performances were inserted between rock and punk bands.

Playing Al Jazari is similar to Betablocker, as it uses the direction buttons to navigate a grid, this time in isometric projection as well as flat grids. Again, ring menus are used for inserting instructions. New robots are placed on the isometric grid and then programmed by writing code in their »thought bubbles«. The instructions command the robot to move, turn, trigger signals or inspect their local space and conditionally execute further instructions. The musical events are generated in an indirect manner, triggers can be placed on the isometric grid, which are activated when a robot moves over the trigger. Robots cannot exist in the same grid positions and will block or move each other out of the way.

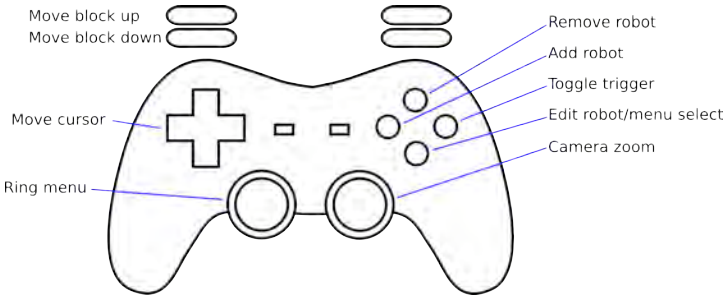

Figure 6: The gamepad layout for Al Jazari

Both these programs/languages are designed to be synched with other systems for network performances, usually other live coding languages for collaborative performances as part of the »slub« live coding group.

Conclusions

Betablocker and Al Jazari are simple languages designed for live use. Betablocker is heavily inspired by existing languages (it's more of a novel visualisation and interaction experiment) and although it's designed to look different, using Al Jazari is the same as programming other languages – robots can be programmed to follow or avoid each other, send messages and generally exhibit complex behavior. They both rely on small programs interacting with each other to provide the amplification of effort, and result in controllable, but difficult to predict situations.

The use of a game pad allows the performer to leave the confines of the screen and keyboard, and means the computer is less of a central point compared to other laptop performances. The only objects you need to interact with live are the game pad and projection.

The language is the instrument when coding live, and the advantage of developing a language specifically for live coding is that it can be

influenced in response to performance practice. There is a lot of work to do in finding the correct balance between simplicity and complexity when considering the pressures of live performance. Live coding languages need to avoid unnecessary detail while keeping interesting possibilities open.

Bibliography

Abelson, #/Sussman, # (#), Structure and Interpretation of Computer Programs, #, http://mitpress.mit.edu/sicp/#

Collins, Nick/McLean, Alex/Rohrhuber, Julian/Ward, Adrian (2003), »Live Coding Techniques for Laptop Performance«, *Organised Sound*, #8,#3, #321, p. 30.

Locke, John (1690), »An Essay Concerning Human Understanding«,#

#»Live Algorithm Programming and a Temporary Organisation for its Promotion«, http://www.toplap.org/index.php/Read_me_paper

Slub: http://slub.org/

Fluxus: http://www.pawfal.org/Software/fluxus/

Al Jazari: http://www.pawfal.org/index.php?page=BetaBlocker

Betablocker: http://www.pawfal.org/al-jazari