Fig. 1 *qrpff tie with DeCSS algorithm*

Fig. 2 *DeCSS, The Movie*

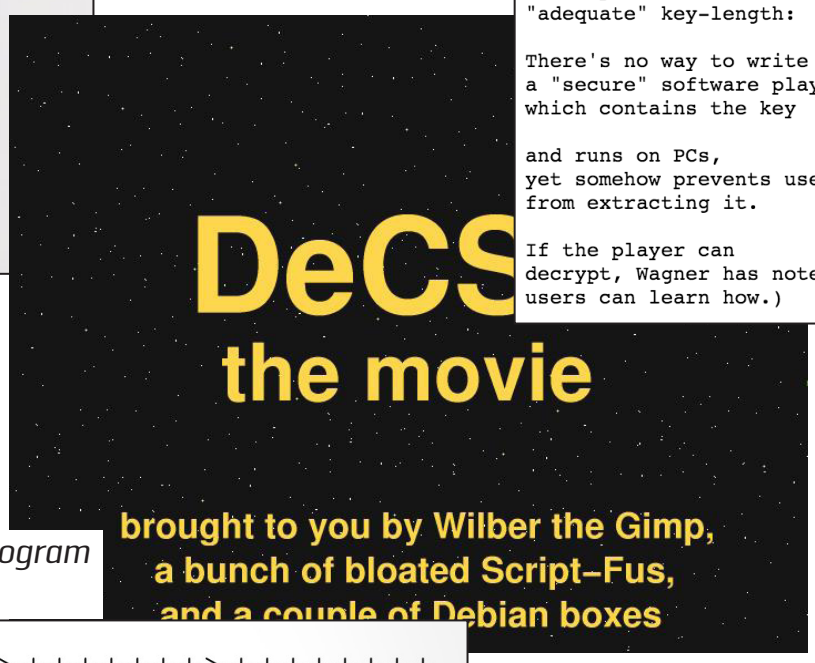

Fig. 3 *DeCSS Haiku*

Here's a description
of how a player key will
decrypt a disk key.

You need two things here:
An encrypted disk key, which
is just six bytes long.

(Only five of those
are the _key itself_, because
"zero" marks the end.

So that's five real bytes,
and eight times five is forty;
in the ideal case,

forty bits will yield
just short of two trillion
possible choices!

Ian Goldberg once
recovered a key that long
in seven half-hours.

But his office-mate
David Wagner points out that
it's _impossible_

to achieve what the
DVD CCA seems
to want to achieve,

even by making
the key some reasonable,
"adequate" key-length:

There's no way to write
a "secure" software player
which contains the key

and runs on PCs,
yet somehow prevents users
from extracting it.

If the player can
decrypt, Wagner has noted,
users can learn how.)

Fig. 5 *An excerpt from "Hello World" program in Shakespeare*

```
                    Act II: Behind Hamlet's back.

                    Scene I: Romeo and Juliet's conversation.

[Enter Romeo and Juliet]

Romeo:
 Speak your mind. You are as worried as the sum of yourself and the
 difference between my small smooth hamster and my nose. Speak your
 mind!

Juliet:
 Speak YOUR mind! You are as bad as Hamlet! You are as small as the
 difference between the square of the difference between my little pony
 and your big hairy hound and the cube of your sorry little
 codpiece. Speak your mind!

[Exit Romeo]

                    Scene II: Juliet and Ophelia's conversation.

[Enter Ophelia]

Juliet:
 Thou art as good as the quotient between Romeo and the sum of a small
 furry animal and a leech. Speak your mind!

Ophelia:
 Thou art as disgusting as the quotient between Romeo and twice the
 difference between a mistletoe and an oozing infected blister! Speak
 your mind!

[Exeunt]
```

Fig. 7 *"Hello World" program written in brainfuck*

```
++++++++++[>+++++++>++++++++
++>+++>+<<<<-]>++.>+.++++++
+..+++.>++.<<+++++++++++++++
.>.+++.------.--------.>+.>.
```

Fig. 8 *Screenshots of programming performance videos in bodyfuck*

Fig. 10 *"Hello World" program in Piet*

Fig. 11 *A program in Piet calculating the day of the week*

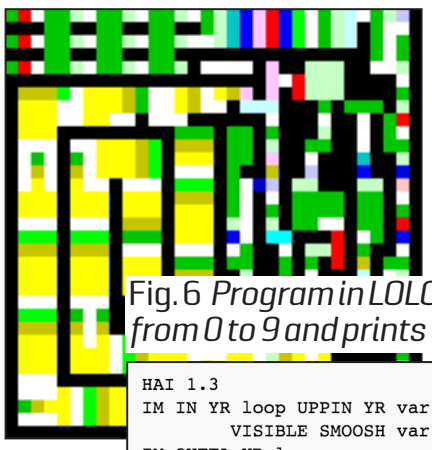

Fig. 6 *Program in LOLCODE that counts from 0 to 9 and prints out the numbers*

```
HAI 1.3
IM IN YR loop UPPIN YR var TIL BOTH SAEM var AN 10
        VISIBLE SMOOSH var AN " " MKAY!
IM OUTTA YR loop
KTHXBYE
```

Fig. 9 *A program that prints out the word "Piet"*

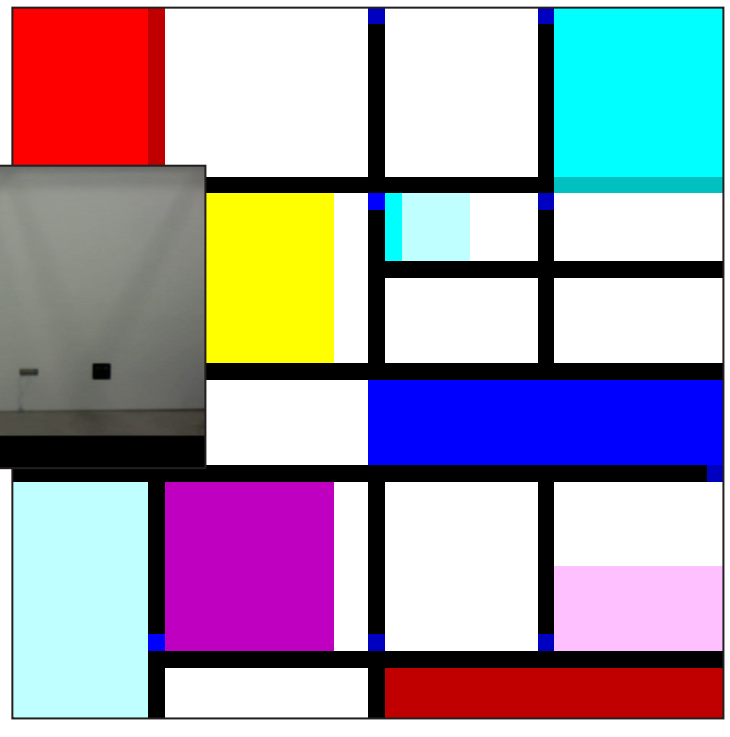

Fig. 12 *Code written in Whitespace*

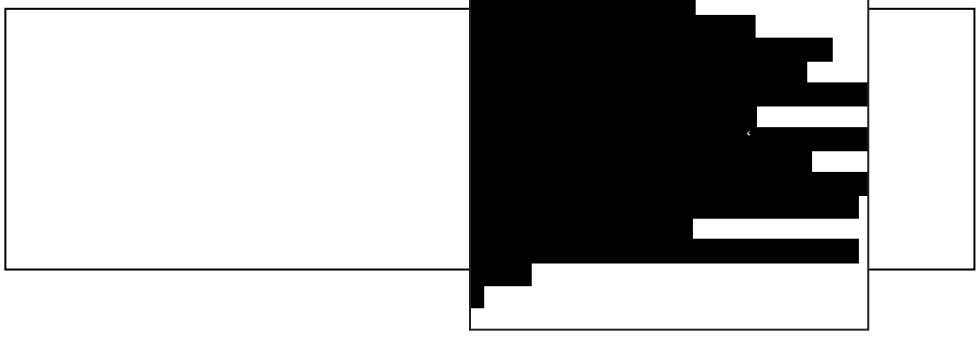

Fig. 13 *Whitespace code when all document is selected*

Programmning Language as a System of *Thought*

Thesis by Medeina Musteikytė
Design by Dirk Verweij and Medeina Museikytė
Graphic Design Department
Gerrit Rietveld Academie 2016

Introduction

Algorithmic/Code art is often addressed as one of the digital art forms. Formally this is where it belongs together with all sorts of works produced by generative digital software. Although algorithms clearly function as our controlled instruments, they often outgrow their primary function. Algorithmic art can be experimental, explorative, provocative and conceptual; on a bigger scale it can also change the way we think, who we vote for, the things we buy, music we listen to... or even determine who we date. In this way, cultural processes are becoming products of algorithms while global society is being shaped and formed by the hands of programmers.

> "If we live in the machine age, then algorithms are the soul of the machine. Every time we commune with a computer, there's an algorithm dancing within, leading us along a winding, pre-ordained route. Tweak the algorithm and you've toppled a dictator. Modify another and a baby is born in Berlin. There's even an algorithm, rendered optically, that lets color blind people see the color red in all its shocking glory for the first time." [1]

Algorithm is a part of how we create our culture over time and it is not bound to the limits of computer, its existence dates back to the first Babylonian calculation systems carved on clay plates and expands beyond the software we use nowadays. The definition of an algorithm describes it as

> "a self contained step-by-step set of operations to be performed". [2]

It is a certain procedure that can be expressed in variety of languages or systems while remaining always active and result-producing process. In any existing form, algorithm has its inherent functionality and is rarely dissociated from it. However, not every code is meant to be run or executed; it might be as well perceived as text or object, language structure or a creative system of thought. The form of abstraction it translates to in order to perform certain commands deserves a separate reflection disconnected from the end result of its function.

> "Coders are deconstructing and re-constructing language structures – quite similar to what many conceptual artists do in their works." [3]

If algorithm is not only a tool, the relationship between its language and performance of given commands gains certain complexity and opens new directions in the process of programming.

Code as a form of free speech

In March 2015 the world's first Algorithmic Auction was held in New York to showcase the aesthetic beauty and unique influence of algorithms. The auction includes software licenses related to a range of significant and archival historic code works. All the works were successfully sold marking a key moment of pure code officially entering the sphere of art.

> "The beauty of the code is that it is a way of thinking"

stated Daniel Benitez, a chief technology officer of Bardan Cinema in Miami, who bought a cobalt blue silk necktie for $2,500 in the Algorithm Auction (Fig. 1). This original piece of neck-wear

in white frame is inscribed with six lines of qrpff algorithm that decodes CSS, the DVD Content Scrambling System.

Moreover, this object embodies an act of civil disobedience; a code as a form of free speech. When MIT student Keith Winston and his friend Marc Horowitz were writing this code in 2001, they were not thinking about the role their work would get in the art world, neither they were considering it as performative piece or an artefact. In K. Winstein's words for them it was

> "programming to make a point. That point was about code as creative means of expression." [4]

In their case, the creative expression was to distill a complex algorithm to its essence for demonstrating the simplicity of secret scrambling method Hollywood used to protect their DVDs. The restricting software wouldn't allow to fast-forward past previews, copy movies or play DVDs in a country where it wasn't intended to sale. The secrecy also gave the studios leverage over manufacturers making them agree on certain conditions in order to produce DVD players.

> "I believed, then and now, that it's a lousy idea to restrict innovation and tinkering just to big manufacturers with lawyers. We all benefit when creativity can take flight anywhere." [4]

– said Keith Winstein who just wanted to learn to build his own player at that time and decided to pick up the DVD descrambling code, called DeCSS which already circulated in the underground. The DeCSS procedure became publicly known

in 1999 and spread out widely with at least 42 different versions and encryptions. The algorithm sets out a procedure that copyright holders still regard to as a criminal act. The entertainment industry responded with lawsuits against hundreds of people who had republished DeCSS, arguing that it revealed a trade secret and was a prohibited circumvention "device" or "machine". The efforts programmers put to make the point that "code is speech" made the algorithm impossible for lawyers to eradicate. Out there was a haiku poem explaining the method, Starr-wars like animation, an alternative rock song and a voice recording of dramatic reading – all representing the same source code. DeCSS movie file (Fig. 2) was created by Samuel Hocevar, an engineering student in France. He applied the "Star Wars" introduction part aesthetics to the lines of DeCSS code and made them scroll off into space throughout his "movie".

Another example- DeCSS Haiku (Fig. 3)

> "A program is a literary work. The idea was to show how strange and difficult it is to classify computer programs and technical information as something other than speech" [5]

This is how Seth Schoen described his work – 456 stanzas of haiku poem explaining the DeCSS algorithm. His goal for writing this poem was to prove that source code should be considered as speech and hence should be given the same legal protections as free speech. Not meant to be interpreted by an average viewer, this haiku can easily be transformed back into a

functioning program if read by a programmer. DeCSS, DVD logo (Fig. 4) formed out of the DeCSS source code characters was generated using the MosASCII tool created by Robert DeFusco. To view the full code (some of the characters are made invisible by having 0 opacity) one has to "select all" and copy-paste the code to another document.

The rest of the examples together with the ones described above are gathered in the online gallery called "Gallery of CSS descramblers" that computer science professor Dr. David S. Touretzky created in reaction to the charges put against him by copyright holders. This is how Touretzky explained his point in court:

> "I think there are three ideas I'm trying to communicate. The first is that computer code written in high level language, such as C, has expressive content. The second is that it's not possible to distinguish between computer code and other forms of expression, such as plain English. And the third point is that it's not possible to distinguish between what people call source code and what they call object code" [6]

Many Internet activists considered this criminal prosecution an attack on the constitutional freedom of expression and by encrypting the algorithm in many forms helped to convince the American Court that the distinction between code and speech was false. Consequently, qrpff tie was not the first example of creative means of DeCSS encryption. The significance of K. Winstein's and M. Horowitz's work was

the way they condensed it and made publicly available. They rewrote the code 77 times each time simplifying the program further and further, until from several hundred lines it shrank down to only six. It was small enough to fit on a necktie, coffee mug or a teeshirt and available not only to programmers: “If you type that in, it’ll let you watch movies”. Because of its minimal appearance and ease of use this necktie became a symbol of a public protest against copyright restrictions.

These restraining conditions provoked a mayor turn regarding the computer code as conceptual language. The hackers were not aiming for the artistic exploration just for its own sake, they were pushed into this experimental zone trying to circumvent the restrictions. However, it was a remarkable start of the whole new dimension being added to an art world.

Esolangs: nonfunctional, impractical and useless

More recent code-art works do not prioritize functionality of an algorithm, but rather treat it as a conceptual process, a system of thought. Programming languages, same as regular human languages, are transforming and developing over time based on what people perceive as necessary to communicate ideas and on the abstraction methods they apply.

Apart from huge variety of languages designed for practical computing, growing number of “useless” experimental programming systems appear online. Often referred to as jokes, these pieces of weird programmers’ art are joined together under the name of “Esolangs” an abbreviation that stands for Esoteric

Programming Languages. As practical use is never a goal, an esolang creator experiments with programming concepts and explore the edges of a programming language with all the creative freedom. Often esolangs function as immersive works and can be perceived as experiential or even performative pieces.

> "Like a Fluxus score, one can choose to actually carry out the instructions, but it can sometimes be understood by its instructions alone. This is true as well for the rules that make up a programming language – they are both primarily dematerialized forms." [7]

In general, a programming language is designed to communicate instructions to the machine, particularly a computer. However, it can exist as a concept outside or without a computer, leaving compiler and software as secondary things. That doesn't make much sense in function oriented practical computing, but serves well the idea implemented by esolangs. A programming language is firstly a set of rules of how to combine and interpret a given set of symbols, therefore it can be also understood by these instructions alone. Even though most of the esolangs can be used in writing working programs, running the code is not the main objective. The attention is pointed to the creative process of programming, construction of the syntax – a layer that is considered as transparent or neutral in traditional computing. Despite of the given form, there is no ambiguity in the way how computer interprets commands of the program. For this reason the only

touching point between human thought and computer logic is the programming language and it is definitely something more than just a communication tool.

There are various strategies of creating an esolang. One of the possible categories involves text based programming languages that deconstruct the regular vocabulary or create their own syntax and set of rules. Most of these esolangs function as parodies or jokes, providing playful ways of giving instructions to the computer. For instance, "Shakespeare" (2001) created by Karl Hasselström and Jon Åslund, a language designed to look like Shakespeare play (Fig. 5). The source code resembles a structure of a play, with acts and scenes serving as labels and characters standing for variables.

Another example of the same kind – "LOLCODE" (Fig. 6) (2007 by Adam Lindsay) uses the distorted spelling manner of internet lolcat memes. The traditional commands are replaced by misspelled or ridiculing words, but the code can be understood quite simply when being familiar with Internet slang.

This kind of esolangs is usually nothing more than funny ways of diminishing the authority of the computer by using the means of language. They certainly give a feeling of more familiar communication adding some human expression, but the structure of the code remains untouched – the only change is how we read it and see it in on the screen. Consequently, these simple vocabulary-based esolangs are more of the formal experiments of translating the symbols from one language to another (some esolangers

even suggest a term of “esolexicon” for this type). However, a straightforward method like this makes as much sense to a non-programmer and might provoke the first attempt of seeing the code out of its traditional definition.
Even though most of the esolangs use text and vocabulary as a tool of expression, the concept behind can be based not only on ciphering practice. What makes an esoteric programming language truly intriguing is the confrontation between computer logic and human thinking. The esolangs that explore a language as a system of thought invite their user to an experiential process leaving unlimited space for different interpretations. Nevertheless, getting the idea of these works is rarely possible by only looking to an outcome (if it has one at all). As if someone would try read a poem in a language they never heard before – it’s only possible to state that it is a language and probably has a structure of a poem. Accordingly, the full involvement is crucial and inevitable condition for being able to navigate in the strange logic of esolang and make one’s own discoveries. In other words, taking the time to learn the set of rules and syntax of a useless “fictional” language.
One of the incentives for doing so is a chance of challenging the brain and its settled conventions of problem solving by finding different methods of communicating the ideas in a newly learned logic system. Probably the easiest way to get a grasp of the concept is by studying one of the pioneering and most well-known esolangs called “brainfuck” (it is also reffered to as brainf***,

brainf*ck, brainfsck, b****fuck, brainf**k or BF due to an offensive connotation of the word). Brainfuck was created in 1993 by Urban Müller and became a significant inspiration for later development of many other esolangs (Fig. 7). The author released it together with a README file, which provoked the reader "Who can program anything useful with it?:)". It should be mentioned, that despite of its weird syntax, this language is Turing-complete, meaning that it is theoretically capable of computing any computable function and any working program can be written using it. Nevertheless, it is hard to come up with more complicated and unpractical way of programing. What makes it so fascinating, is that the language is composed of extremely minimal syntax and has a very simple idea, though the perception of it is complex and highly challenging. As Daniel Temkin puts it in his article on brainfuck :

> "Programming languages are perhaps the most direct conduit between human and machine: here our commands translate into machine instructions. Brainfuck <...> uses this process of translation to explore the breakdown of communication and expose how computers train us to think. It is an experiential piece rather than a practical tool to create working programs." [8]

The way how "computers train us to think" or else plain computer logic without any facilitations for the human mind is materialised in the language of brainfuck. Its minimal structure consists only of eight commands, each represented by a single character: > < + - . , [ ] .

> Increment the pointer (move to the right)
< Decrement the pointer (move to the left)
+ Increment the byte at the pointer.
- Decrement the byte at the pointer.
. Output the byte at the pointer.
, Input a byte and store it in the cell at the pointer.
[Jump forward past the matching] command if the byte at the pointer is zero.
] Jump backward to the matching [command unless the byte at the pointer is zero.

Brainfuck has no variables, no conditionals or functions — any word commands like "print" or letter symbols and numbers are ignored allowing the program to be written only in punctuation marks. Instead of usual constructs, this program has a byte pointer which can be moved in the array of 30000 bytes by incrementing or decrementing the value of a cell. The data pointer is initially set to zero and it sequentially moves from one command to the other. In his paper on brainfuck, Daniel Temkin gives a non-programmer friendly explanation and example:

> "In practice, a brainfuck programmer can't write "x = 1" to assign 1 to x, since "x," "=," and "1" have no meaning, nor do the spaces between them. Instead, he must traverse memory manually, using angle brackets to find a location one can think of as "x," and then populate that memory cell by incrementing the value (initialized to zero) with a single plus sign:
>
> +

To store the number five, he would do this:

+++++

While this may seem intuitive, like scratching consecutive lines to keep score in a card game, complexity multiplies quickly. To write the number thirty-six, rather than writing thirty-six plus signs, one can step back and forth between memory cells, incrementing one value repeatedly and thereby producing that number more compactly. The top line below multiplies six by six, but the other lines also arrive at a value of thirty-six using other programmatic tricks:

++++++[>+++++++<-]>

++[>++<+++++++]>

++[>–<——-]>

–[>++<——-]>

–[>–<+++++++]>

+[->-[—<]>-]> [1]

Thirty-six is no longer just a number, it is a set of steps to build a number, and, without careful reading, could easily be mistaken for stored text, a set of instructions, or anything else representable in programming." [8]

As the example above has shown, to complete a simple task like writing a number 36 requires a logic strategy and the result can be achieved in various ways. Therefore, number 36 becomes a resource which allows original problem solving approaches to be developed. Obviously it is highly time consuming and has a reason to be considered completely useless.

Traditional programming languages are meant

to make the code more human-friendly by having descriptive names and commands. They serve as abstractions simplifying an exchange between human mind and logic of the computer. Brainfuck suggests completely opposite strategy. It not only refuses any means of making programming easier, but it constrains the user to communicate commands the way machine perceives it:

> "We're forced to think in terms closer to the machine, and to construct lengthy algorithms for simple tasks." [9]

Since there are no cutting corners or workarounds, the pure computer logic behind the code is exposed in all its glory. This is the moment when the human-machine confrontation reaches its extreme while the process itself turns out to be somewhat irrational – it is getting the logic to the point where it becomes not logic anymore. Brainfuck erases the boundary between human and computer communication by immersing our mind in simulative machine's system of thought. Useless as a programming language, it serves as an experiential piece which alters the concepts of programming. Similarly, limitation and constraints have been explored in various artistic practices. For instance, "La disparition" ("A Void") a 300 page-long novel without a single letter "e" by French writer Georges Perec published in 1969. He applied a method of lipogram – one of constrained writing techniques that Oulipo collective ("workshop of potential literature") was delving into. Instead of having the story overwhelmed by the word puzzle, the author finds the obstructions inspiring and liberating his

creativity in new ways. In the postscript of the book Perec describes his experience as amusing:

> "it took my imagination down so many intriguing linguistic highways and byways, I couldn't stop thinking about it <...>." [10]

What starts as a formal set of rules forbidding the easiest and most familiar ways of communicating, triggers original forms of expression and unique creative solutions. And again, this is how programming in brainfuck works. This approach must have been powerful enough to provoke countless adaptations and interpretations derived from brainfuck, that later joined together under the name of esolangs. Till nowadays, it is the programming language to be mentioned as the main inspiration among the community of esolangers.

One of the cases of taking the same idea further is Bodyfuck (2010)(Fig. 8). It is Nik Hanselmann's extension of brainfuck which translates the language directly to physical gestures. The act of programming gains performative qualities by pulling out the process out of the screen. Accordingly, programming in this way becomes even more absurd, as there is no backspace and a single error would ruin the whole thing. In order to perform, one has to be as precise and mechanic as computer processes are. In bodyfuck, traditional text based input methods are replaced by physical gestures.

Although the translation from brainfuck commands to their motion equivalents is quite direct, the act of programming gets more abstract form of expression and introduces new ideas. The "dance" still consists of the same

eight steps that are represented as punctuation marks in their textual version and the syntax remains minimal and unmodifiable. The dialog between human and the machine becomes more provocative as the performer is forced to work out the code physically. While the method might be used to communicate simple instructions as “copy“, “smiley“, “hello world“ etc; performances like “Walk to 68“ , “going nowhere“, “endless“ and “undefined walk“ expand the borders of interpretation.

Code made beautiful

Bodyfuck introduces well an idea of unconventional input methods in programing by replacing symbols with motion gestures. In addition to the concept behind, the way “how“ the commands are communicated is certainly what esolangs are built on. In general, most of the programming languages rely on text-based input, like the examples mentioned above.
With this in mind, how would a programming language look like, if it had no characters or textual symbols? What if the code could not be read, but rather judged by the way it looks as any other image? And on the whole, how relevant or significant that is if the code is beautiful or ugly?
A “Hello World” program written in Piet could clearly stand for beautiful, except the fact that it doesn’t look anything like code (Fig 9, 10, 11).
A composition of colourful pixels literally prints out the words “Hello world” to the screen, and the image itself is nothing more than the source code of this output.
Piet was created in 2001 by David Morgan-Marr and named after the painter Piet Mondrian

because of visual resemblance to his abstract paintings. In Piet, the code which is usually behind and never seen t is meant to have a certain aesthetic value.

Programming in Piet is not only about the best rational strategy to get the desired output, but also about composing a nice looking image – source code. Disregard for functional aspects of programming guarantees Piet a steady spot in the realm of esolangs. The language's visual qualities are so dynamic because the colour pixels do not resemble exact commands. As a matter of fact, colour itself doesn't signify anything at all – the code is built on transitions between hue and darkness of the pixels. The change or transition works as signifier, not a symbol, so editing one command - pixel group requires modifications in all the following chain. This feature gives a possibility for each programmer compose original looking images even when working with the same commands.

Empty programs

Esolangs like brainfuck, bodyfuck or Piet demonstrate that programing language can be built on a complex idea and acquire any form. Although this might seem perplexing enough, some esolangs can be so nonfunctional that computer is only a secondary thing they need to exist. Moreover, they can even have no code, or consist of a file of zero instructions. But can it still be considered a program when lacking the main component? A definition offered in U.S. Copyright Act of 1976 describes computer program as

> "a set of statements or instructions to be

> used directly or indirectly in a computer in order to bring about a certain result."

Referring to this explanation an empty set or an empty sequence can still be seen as computer program if it is valid within a language. By all means, the esolang community is the one exploring this notion to its extremes. For instance "Compute" – a programing language with no syntax and no result created by esolangs.org user nicknamed Orange:

> "Compute is esoteric programming language that "has no required syntax and has the power to solve any and all problems. It is smart enough to interpret any human language (English, Spanish, Latin, etc), any programming language (C++, Java, brainfuck, etc), or any kind of data you can think of. The only downfall is that there is absolutely no I/O." [11]

In other words, the language doesn't actually allow interchange between an information processing system-computer and the outside world-human. However, it implies a fictional execution of any possible commands communicated in a natural way of speaking. Telling it to print a sample "Hello World"command would end up with the same result as asking the program to become self-aware – Compute replies "Done" while no output is actually produced. It goes without saying, that Compute is very memory efficient and runs extremely fast. Nevertheless, it is all based on empty promises and requires total trust in its ability to "compute". This language is also questioning ways of human-computer

communication by applying the opposite strategy to brainfuck. It simulates an Utopian situation of absolute apprehension when no compromises or abstractions are needed – machine understands our language structures directly like another human being.
Another twist of the concept of emptiness is reflected in an uncommon syntax of "Whitespace" esolang written by Edwin Brady and Chris Morris in 2003 (Fig. 12, 13). In contrast to the most of text-based languages where the spaces between words are ignored (the same way as white pixels in Piet), Whitespace interpreter considers all the letters and visible symbols as null. As a result, only spaces, tabs and line-feeds have meaning and are read by the compiler. So the code written in whitespace looks like an empty sheet—the inscribed code is invisible even though it can be read or executed.
The idea of emptiness has been a source of explorations for many artists and thinkers. But could a work of programing be seen together with other famous art pieces like Nam Jun Paik's "Zen for Film" unexposed film strip, Robert Rauschenberg's "White Paintings" or John Cage's "4'33" silent piece? It might be compellingly easy to draw direct comparisons between concepts behind esolangs and other artworks, for example, non-text characters in Whitespace and silence as non-sound object in John Cage's work or quine —

> "a self-replicating program that takes no input and produces its own source code (its medium) as its only output"[12]

and Nam Jun Paik's film depicting only its own material qualities. Although this may make sense

and provoke new insights, the works of code, in contrast to art pieces mentioned, have only one and always the same way of interpretation when run by the machine. For that reason concept of the blankness has different qualities when extended to computational medium, where everything is rational and calculated. Computer's performance is based on definite logic and programming languages are perceived as formal systems. Most of the esolangs use confusing commands and unconventional syntax, but these instructions are never confusing to the machine, they get executed the same way as other procedural programming languages.

Program never meant to be run

Is it possible to confront computer logic by designing a programming language which can never be actually performed and produces no functional programs at all? "Unnecessary"— an esolang created in 2005 by anonymous user nicknamed Keymaker—is challenging the conventions of programming to a level of absurd. The esolangs.org website describes it as a programming language

> "where the existence of a program file is considered an error".[13]

It means, that running this program on any existing file, even an empty one is impossible — the program reports an error and terminates immediately. Only the file that cannot be found — the one that does not exist leads to successful execution of a program. The output consists of a file with a single command "NOP" (sometimes spelled no-op for "no operation") which is an operation in a programming language that does

nothing according to esolangs.org definition. In Keymaker's words, the main principle of Unnecessary is that

> "Every working program is a null quine. <...> The main idea was that the language could not have programs, other than the kind that don't exist. (Can it have those then if they don't exist?) Then I noticed that every valid program (whatever that is) is a/the null-quine but that was more of a by-product of the main idea. Fitting nonetheless!" [14]

The null quine that Keymaker mentions is

> "a special quine that does nothing, created from nothing. Because the only successful program is one that doesn't exist, this output (of nothing) is identical to its input (of nothing). Unnecessary is like a language equivalent of the null quine itself." [15]

Unnecessary outplays the computer logic in the most rational way that leads to completely irrational processes. To understand this paradox no computer needs to be actually used, as the outcome would not make any sense. Programming language that is not meant to be executed by a computer, reshapes the boundary between machine and human thought.

Conclusions

Algorithms are powerful and can trigger processes, change forms of perception, ways of thinking and modes of acting on a larger scale. Usually we don't think of programs as art because of their inherent imperative, a practical potential to be run or executed, to produce a certain result.

Looking at computer programs as always active bring them close to Fluxus performance scores. Even though computer always executes the same commands the same way, conceptual esolangs offer the possibility of different interpretation to the process of programming. Experimental language systems open new passages of navigating in strange logic constraints and even make possible to outgrow them. Designed for experience of thinking through them, esolangs can be taken away from the computer and exist as set of rules and instructions describing the behavior of their syntax (if it has one). Like some of the Fluxus event scores or Oulipean systems, conceptual esolangs contain the performance in the text itself, while actual execution depends on the interpretation and can only be an imaginary one. It seems controversial that the programming languages that are not intended to be actually run by a computer are the most compelling in the realm of esolangs. While this term itself gathers together works of different weights like superficial joke language LOLCODE and complex logic systems like brainfuck at the same time, their references to the art world might still be a question.

Meanwhile, the world's first Algorithm Auction is throwing works of plain code to the art market and erasing traditional boundaries by marking a turning point of approach towards programming. All things considered, programming languages are not bound only to computing anymore; they become figure of thought and reflection in artistic practice.

Works cited

1. Laboratories, Ruse. "This is The Procedure." Artsy. N.p., 19 Mar. 2015. Web. *<https://www.artsy.net/article/ruse-laboratories-this-is-the-procedure>.*

2. Bronson, Gary J., and David Rosenthal. "Introduction to Programming with Visual Basic.Net" Google Books. Jones and Bartlett Publishers, 2005.

3. Pereira, Lorenzo. "Computer Codes Art : A New Form of Creativity ?" Widewalls. N.p., 2015. Web. *<http://www.widewalls.ch/computer-codes-as-new-form-of-art-computer-arts/>.*

4. Winstein, Keith. "Qrpff." Artsy. Ruse Laboratories, 17 Mar. 2015. Web. *<https://www.artsy.net/article/ruse-laboratories-keith-winstein-creator-of-qrpff>.*

5. Schoen, Seth. "The History of the DeCSS Haiku." . N.p., 2001. Web. *<http://www.loyalty.org/~schoen/haiku.html>.*

6. "David S. Touretzky Deposition, in MPAA v. 2600." (PA; July 13, 2000). United States District Court, 2000. Web. *<http://cyber.law.harvard.edu/DVD/NY/depositions/touretzky.html>.*

7. Lindgren, Chris. "Interview with New Media Artist Daniel Temkin on Esolangs." Code Work, 15 June 2015. Web. *<http://umncodework.github.io/past_events/codework-interview-temkin-esolangs/>.*

8. Temkin, Daniel. "Brainfuck." NMC MediaN. N.p., 07 May 2013. Web. *<http://median.newmediacaucus.org/tracing-newmediafeminisms/brainfuck/>.*

9. Lindgren, Chris. "Interview with New Media Artist Daniel Temkin on Esolangs." Code Work, 15 June 2015. Web. *<http://umncodework.github.io/past_events/codework-interview-temkin-esolangs/>.*

10. "A Void - Post VII." Inside Books. N.p., 26 May 2008. Web. *<https://insidebooks.wordpress.com/2008/05/26/a-void-post-vii/>.*

11. "Compute." - Esolangs.org N.p., 29 Sept. 2011. Web. *<https://esolangs.org/wiki/Compute>.*

12. "Quine (computing)." Wikipedia. Wikimedia Foundation, 8 Dec. 2015. Web. *<https://en.wikipedia.org/wiki/Quine_(computing)>.*

13. "Unnecessary." Esolangs.org. N.p., 17 Dec. 2014. Web. *<https://esolangs.org/wiki/Unnecessary>.*

14. Temkin, Daniel. "Interview with Keymaker." N.p., 6 May 2014. Web. <http://esoteric.codes/post/84939008828/interview-with-keymaker>.

15. Temkin, Daniel. "Unnecessary: Purely Conceptual Languages." N.p., 11 Nov. 2014. Web. *<http://esoteric.codes/post/102380982203/unnecessary-purely-conceptual-languages>.*