

Computation as Dynamic Topography

Gabriel Menotti – gabriel.menotti@gmail.com

In order to think about the relations between computation and technical images, it is necessary to approach very carefully the conceptual corpus that ties them together. These two fields have much more in common than the theories about them may suggest.

The issue is that the study of communication systems and language, in a wider sense, is often reduced to the study of meaning, following parameters that originate from textual analysis. Screen studies has since long suffered from this vice, which results (for example) in the assumption that *reading a movie* is a better way to apprehend the cinematographic work's true structure and signification. But this is much more misleading in the case of digital media, given that the code underneath it is apparently similar to text, but works in a very different way.

Due to this tension, it is not surprising that a great deal of theory on the specificities of code as language comes from the field of electronic literature and hypertext: for example, on the seminal works of N. Katherine Hayles, Jay David Bolter and Florian Cramer, among others. But it is not surprising, either, that their ideas are not capable of dealing with all the implications of computation. They prove insufficient to handle the overlapping between code and systems of signification other than text – and even the peculiar qualities and operation of code by itself.

The reason is that their very concept of code is biased, built in opposition to that of text. For instance, even though Hayles stated intention in the book *My Mother Was a Computer* is to create "a theoretical framework that enables code and language to be thought together" (HAYLES: 16), she only focuses on the interactions between speech, writing and code, i.e. how "each successor

reinterprets the system(s) that came before, inscribing prior values into its own dynamics" (HAYLES: 39).¹

Albeit fairly all-inclusive, Hayles' work restricts *language* to this pair of semiotic structures she is most interested in, ignoring nearly completely other systems for creating signification (and value and effect) like cinema, architecture or curating – which are, in fact, the subjects of the present work. Since it would be plainly wrong to reduce such complex systems to particular cases of writing or speech, Hayles' framework cannot be considered universal (since her take on language is not).

In fact, by articulating the metaphorical dialectics between code and text, Hayles misinterprets the very nature of computation. On the one hand, she is very sensible in assuming that computation is information processing that "connotes far more than the digital computer" (HAYLES: 17). Indeed, computation can be found at work on primitive devices such as an abacus, or even on macrobiological processes like the evolution of successive generations of bacteria – as we can literally see in Eduardo Kac's transgenic art piece *Genesis* (1998/ 1999).²

But, if so, why is it that Hayles takes writing and speech as "legacy systems" to code, as if the latter was a maturation of the former (HAYLES: 40)? If code is, as she says, "the language in which computation is carried out" (HAYLES: 17), it is at least as old as computation itself. To situate code historically is also to situate computation historically. To consider code as a semiotic system that *exceeds* – and, hence, somewhat *supplements* – text and speech (HAYLES: 40) is also to presuppose that computation depends on the development of these two systems to exist. In fact, that goes against Hayles' own quasi-transcendental concept of the Computational Universe – "the claim that the universe is generated through computational processes running on a vast computational mechanism underlying all of physical reality" (HAYLES: 3).

¹ Not to mention that her arguments are constantly punctuated by whole chapters consisting solely of the analysis of literary texts, from which she also distils meaningful concepts.

² <<http://www.ekac.org/geninfo2.html>>.

This paradox shows how deceptive it can be to take text as a synonym to language, as well as to believe in the priority of the past in the development of systems for creating signification. If historical prerogatives are to be accepted, we might as well take images as the parameter. As signifying structures, images historically precede texts, since humankind used drawing and painting prior to writing. In fact, writing started as a particular modality of drawing, not because of the creation of new symbols, but as the image was fractured and unfurled in lines (FLUSSER: 132). If we were explain text based solely on this other major system for creating signification, we could say that writing is but a form of arranging images in a standardized way, which allows for a more sophisticated abstraction of meaning – but we all know that writing is more than that.

In the same manner, architecture and other forms of spatial organization were employed to convey messages before words ever came into existence, and in a much more pragmatic way. If some tree branches were piled up in the middle of a path, in the guise of a wall that poses difficulties to trespassers, their effect of meaning would be quite obvious even before we had to physically engage with them – as well as if they were put in a hole, drawing attention to it. Spatial arrangements can even be thought of as *executable*, a property Alexander Galloway considers exclusive to code (GALLOWAY: 165). The organization of the territory can “do what it says” – from simple effects such as projecting a shadow to protect us from the scorching heat, to even more refined ones like imprisoning animals (in a booby trap).

Therefore, there is no reason to privilege text and speech in creating theories that entwine code and language. This is particularly true if our main interest is to analyze how computer algorithms interact with visual or spatial systems for creating signification, whatever they are. The best approach would be first to consider *code* in relation only to *computation* and *algorithms* – which, together, constitute the framework with which digital computers operate.

So, for the present work, considerations about the most symbolic levels of the computer system will be avoided, in order to allow for an almost laboratorial

analysis of its processing core. It is undeniable that these layers – including social, economical and cultural macrostructures – are a constitutive part of computation, being even responsible for its ultimate understanding and development. But it also seems that the processes that interlace code, algorithms and computation can be more clearly comprehended if they are not engaged with any other, highly abstract, symbolic system.

The digital computer itself seems to be the most appropriate model to ground this analysis, due to its popularity and the way it shapes the broad understanding of computation. It may help us ponder what kind of problems the abuse of the metaphors that relates code to text can cause to the semiotic analysis of computation.³

There is no Text

Once we take code for text, the first mistake we can make is to think that, as the distinguished electronic poet Loss Pequeño Glazier believes, “programming is writing” (*in* HAYLES: 108). We do not have to go very deep in the history of computation to contest this bold statement. The first programmable computers, like the Colossus (UK, 1944) and the ENIAC (US, 1945), were operated using punched-hole paper tapes or patch cables and switches, in a way that resemble more page designing (or *braiding*) than writing. These systems depend on the position of objects in a field. They create algorithms as patterns that can be apprehended at a glance, like images.

Even today, in the lowest levels of computing, programming is closer to queuing beads in a chain (that is, *bits in a stream*) than articulating meaningful signifiers. And, of course, there are some high-level programming languages that work according to completely different systems of signification,

³ What will be done here is an exercise on how computation can be conceptualized in a different frame, which is not at all meant to be totalizing. Particularly, it is an early attempt of defining it according to certain parameters that will highlight how it fits in a universal theory of the *dispositif*.

such as drawing abstract pictures (like the esoteric language *Piet*)⁴ or building diagrams (in a dataflow environment as *Pure Data*).⁵

If we consider that programming is to implement an algorithm in a particular way so that it can be instantiated in a given system, then the code to be employed depends mainly on the system we want to affect. It must be something the system can be informed with. Either way, code is just an arbitrary interface, language as operational abstraction, whose desired characteristics are but formal economy and semiotic convenience. For this practical reason, it is no surprise that we came to use alphanumeric strings that resemble words, arranged under logic similar to that of natural languages, as our main technique for programming computers.

Programmers *want* programming to be like writing. That is what Donald Knuth literally asks for, in a 1983 essay: to consider programs to be “works of literature” (KNUTH, 1). His main reason for proposing this new attitude (as he puts it) is for the sake of a universal comprehensibility. This methodology is not natural for computation, though. Quite the contrary, it drives programming away of computation, focusing only on its results: “instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do” (idem).

Some of the most evolved programming languages, object-oriented codes like C++, even “create a syntax using the equivalent nouns (that is, objects) and verbs (process in the system design)” (HAYLES: 57). How can a code be more convenient than this? Since they comprise a familiar set of elements and rules, these languages “allow programmers to conceptualize the solution in the same terms used to describe the problem” (HAYLES: 57) – in other words, they bridge the gap between the engineering and the efficient logic.

⁴ <<http://www.dangermouse.net/esoteric/piet.html>>.

⁵ <<http://puredata.info/>>.

That means computer algorithms does not have to be composed in incomprehensible machine language. High-level code makes it possible for them to be created as abstract instructions that will be later interpreted as binary formalities. By using intermediating mechanisms – *compilers, interpreters* or *virtual machines* – the abstract orders are translated into bit patterns to be applied directly to the processor, where computation really happens. Then, as the system runs, the bit patterns are re-abstracted and the algorithm manifests its results as surface effects.

As we will see, these translation mechanisms are a constitutive part of the computer itself, an effect of the system's organization in dynamically engaged levels of abstraction that computation produces. So, the hidden layer of text is not really the *source* behind the graphical interface (GALLOWAY: 65). It is actually another cover, hiding deeper layers of materiality. Only if we are naïve enough to ignore this whole structure is it possible to consider programming and writing as equivalent, as if programs were some kind of magical incantations that make computers alive.

This mistake is slightly reinforced by the coincidence of interfaces for the production of both structures, and how we similarly engage with them. Since text and code alike are usually made of alphanumeric characters, they are created in the same way: by typing in the keyboard – this vestigial typewriter.⁶ But, in the end, every key stroked, be it to produce a letter or to shut the computer down, will have the same effect on the computer's heart. All the high-level commands are built on a binary base, and end up being converted into machine language as code goes through all the levels of abstraction in the computer and reaches its processing core (HAYLES: 57). So, a computer program has less to do with the text it resembles than with the image it produces in the surface of the screen, upon running.

⁶ But we should stress that typing is not always a form of writing, especially considering the scarcity of input interfaces of a multimedia digital computer. Pressing keys in a QWERTY keyboard can as well be a form of playing first-person shooter videogames, activating MIDI signals or toggling effects for image processing. For all those matters, typing can even be a kind of performance in itself, as some practitioners of livecoding suggest. See for example the Toplap collective "Livecoding Grades", at <www.toplap.org/index.php/LivecodingGrades>.

The other frequent mistake caused by taking too seriously the superficial similarities between algorithm and text is to believe that the machine that is running a code is, as Hayles proposes, its *primary reader* (HAYLES: 50). This analogy is not valid because, since code is ultimately converted into formal states of the processor's discrete elements, the machine never truly *decodifies* a program; it *incorporates* it. The code *informs* the machine. The algorithm is the computer's *form*, impossible to tell apart from its *operation*. So, even though the machine does *interpret* the code in a way, it is not as a *reader* in as much as an *actor* or an *involuntary performer*.

As Hayles herself states just before making that claim, "a computer program has only one meaning: what it *does*" (HAYLES: 48) – or, we would better say, *what it makes the computer do*. The code is applied to the system as a script of actions, patterns or movements, which the machine follows as if it was performing some kind of *mimics* with discrete binary states. As the dilemma of machine consciousness remains unsolved, we should assume that it acts like possessed furniture, giving yet another explanation for its *ghost* – which, as we can see, is *computation* itself: a poltergeist phenomenon.

Following the same logic, when Hayles claims that "the machine is the final arbiter of whether the code is intelligible" (HAYLES: 50), the problem she is addressing is actually *functionality* – and, in this regard, the final arbiter is always an external agent, such as the human operator in front of the screen.⁷

To the system, the appropriated code is always intelligible. Provided it is physically possible (i.e. there is enough memory, processing power, etc), binary information will reach the processor, and the computer will be informed. If that does not mean information will be satisfactorily carried through all levels of system abstraction, it is because the judgement on the

⁷ This is also reflected by the different concerns people may have about programming. On the one hand, there are savvy experts who pursue the concept of *elegant code* – that is, formally economical and stylistically sophisticated programs. On the other, you have the *script kids*, low-level hackers who are just interested in what the code can do, and built programs like Frankenstein monsters, indiscriminately copying parts from other people's source codes. Even though it may seem that just these neophytes are concerned with functionality, in fact elegant coding is also more about competence than aesthetics: its ultimate goal is to create software that uses the system resources wisely and runs in the smoothest way.

code's efficiency comes not from the machine, but from an external subject who, anxiously regarding output interfaces, expects something from computation.⁸

But, from the machine's standpoint, and considering the computer as a whole, a system crash or an error message is just a side effect of computation, as arbitrary (and as *correct*) as an animation playing due to the coordinated flickering of colour pixels. Digital artist Cory Arcangel abuses this arbitrariness of system abstractions in his *Data Diaries*, a collection of memory data dumps played as Quicktime movies.⁹ The data can be rendered as moving images because the machine doesn't actually judge its content – it just processes it.

Hence, contrary to what Galloway states in the presentation of this work,¹⁰ the Quicktime player is not being fooled by this procedure. The software never expects the files it plays to be movie files – it cannot even recognize what is a movie, actually. It does not make any difference for the software which data it processes. It makes a difference for *us*. It is we that have always been the fools, believing that what is on the screen is anything more than the echoes of distant, convulsive bits.

The Spatial Logic of Computers

The first step to understand how code operates and creates signification is to go deep into the heart of the machine, and look into the processes that build the system, where code is ultimately addressed: computation. Roughly speaking, computation is to process information – that is, to *inform information*. We can better understand that definition if we borrow the concept of information Vilém Flusser developed in the series of essays in which he attempted to sketch a philosophy of design. To him, *in-formation*

⁸ Some high-level programming frameworks are built so that code they consider to be *wrong* cannot reach the CPU, presuming that that would cause problems to the machine and (supposedly) that is not what the user wants. It is like some VCRs that disguise white noise – video signal in its purest form – with a placid blue screen.

⁹ <<http://turbulence.org/Works/arcangel>>

¹⁰ <<http://turbulence.org/Works/arcangel/alex.php>>

itself can be seen as a process: the act of *giving form* to the *material* (FLUSSER: 12). Flusser thinks of both concepts in a complementary opposition: “[form] is the *How* of the material, and material is the *What* of the form” (FLUSSER: 27). So, they cannot be told apart from each other.

According to this notion of information, communication becomes the process of transmitting forms from one medium to another. In a very basic example, it can be said that the broadcast antenna would shape the electromagnetic spectrum frequencies in a particular way, which would in tandem shape the flow of electrons in the cathode ray tube – consequently, shaping the pixels in the monitor screen. As Hayles says, “making, storing and transmitting [...] help to constitute the bodies of subjects and texts” (HAYLES: 7).

Hence, Flusser’s concept of information ultimately refers to the very materiality of the medium. But it is a *relative* materiality, of the same order as Hayles’: “an emergent property created through dynamic interactions between physical characteristics and signifying strategies” (HAYLES: 3). This idea of materiality is somewhat contextual: it comprises how the system is both *understood* and *used* (which will, in the end, direct the way the it is *designed and re-designed*). If this scheme were to be applied to a computer, computation would take the place of the *dynamic interactions* that create the interplay between algorithms (form) and the computer (matter). Immediately, the materiality would be manifested in the guise of hardware state updates, just like in the Universal Turing Machine.

Conceived by mathematician Alan Turing in 1936, the Universal Machine is a theoretical model for all computers: it is capable of every kind of information processing, including of its own rules of operation, in a pure formal way. This reflexivity contributes to its universality – “no computer that has been built or ever will be built can do more” (KITTLER: 1999, 18). It is a very simple device, consisting of a *tape* divided into discrete cells and a *read/write head* that moves it according to the rules prescribed in an *action table*. Each cell of the tape can be either occupied or not, holding a binary state. The tape reads the

cell over which it is positioned one at a time and, depending on the rules it is following, changes the state of the cell and move on to another.¹¹

It is important to stress that that is all the universal machine can do: move the tape and update its states – i.e. set references for future movements and updates. This “economy of operation” is what makes Turing’s device so powerful (HAYLES: 176). With just this small set of pieces and capacities, it is able to perform every kind of computation because it *discretizes* forms and information: trim them down to a series of presences that can be recorded into the tape’s cells and processed in steps. It is the same logic behind digital processors and their machine code.

Moreover, the Universal Machine also brings form and information down to one and the same thing: “monotonous rows of binaries” (KITTLER: 1999, 247). In it, the difference between data and operation is null. As Kittler says, a command “is neither a human enunciation nor a letter symbol, but just one of many series of bits” (KITTLER: 1999, 247).

These concepts are very well illustrated by Dave Griffiths’ livecoding environment *Betablocker* – an “acid techno machine” consisting of 256 cells (bytes) that can be occupied either by instruments (commands) or parameters (data).¹² Since the space is limited, as the performer goes changing the contents of each cell, commands start to destroy one another, and residual bytes gain new functions: an instrument can suddenly become the pitch value of another, or amplitude can be turned into reverb, depending on where it is in the grid.

In any computer system, just like in *Betablocker*, what defines what a bit stands for (either commands or data – and of which kind) is its *relative position* (KITTLER: 1999, 247). For that reason, it is important that the platform in which computation is instantiated (as abstract as it can be) is “a

¹¹ For a more detailed description of the Universal Turing Machine, see KITTLER, 1999: 18.

¹² <<http://www.pawfal.org/index.php?page=BetaBlocker>>.

measured environment" (KITTLER: 1999, 243). If there is no references, no ratio, no directions, processing is impossible.

So, truth be said, machine language is not really made of numbers – of zeroes and ones, as it is usually believed. It is more like *positions in a territory*. The zero-level materiality produced by the binary processing of information works like *topography*, the distribution of discrete volumes in relation to each other.¹³ Such a configuration drives the computational processes, whose results are fed back into itself, updating the topography and so on and so forth.

Space, under the concept of Brazilian geographer Milton Santos, satisfies all conditions Hayles states as necessary for computation: "a parsimonious set of elements and a relatively small set of logical operations [...] instantiated into some kind of platform" (HAYLES: 18). According to him, space is precisely an inseparable set of a system of actions and a system of objects (SANTOS: 51): "the system of objects stipulate the way the actions occur and, on the other hand, the system of actions leads to the creation of new objects or is instantiated over pre-existing objects" (SANTOS: 52).

This *spatial logic* behind computation can be clearly noticed in more elementary devices. The first programmable computer, build by Konrad Zuse two years after Turing publicized his ideas, was basically made of *relays* – switches that let electricity pass or not depending on the state of other switches (KITTLER: 1999, 18). In Zuse's computer, the binary state of the relays (open or closed) defined the trajectory of the electrical current throughout the circuit, which would change the configuration of the relays with every iteration. The form of the circuit was analogous to the configuration of switches, which could be simply read as a numerical pattern. Modern-day microprocessors, like Intel's *Core 2 Duo*, behave in a likewise fashion. They are nothing more than integrated circuits comprising hundreds

¹³ It is important to stress that topography is not employed here in the sense of the written description of a place, nor even in that of mapping or charting – which would be signifying systems in themselves. In this work, it refers to the *relief of a terrain*, that is, a purely formal organization of space.

of millions of transistors, small semiconductors that have a similar function to that of the relay: direct electrical voltage one way or another.

Code is Not Computation

The computer's form keeps changing while it is computing – information is processed as the system's relief updates itself. But the machine's discrete signifiers are not satisfactory for human subjectivity, which tends to deal with continuous forms, and creates consistency (like the segregation between figure and background) even where there is none. The hardware complex states cannot be plainly understood (or subtly affected) by human operators. We need some calculated distance to turn the binary territory into a practical landscape – something that gives materiality (and *usability*) to computing. In the words of Hayles, "signifying strategies" – that is, *code*.

Just as form obfuscates matter, high-level languages obfuscate machine code, and software obfuscates hardware; signifying strategies usually cover computation for *operational* reasons. But even the most inhumane numerical code, be it binary or hexadecimal, is a metaphor of the spatial arrangement – an abstraction of computation.

Early programming languages like BASIC and FORTRAN, called *procedural*, still retain marks of the spatial logic of computation. They "conceptualize the program as a *flow* of modularized procedures" (HAYLES: 57) using Boolean operators and *directional* commands, such as GOTO, BACK, BREAK, STOP. But, as we go up in the layers of code, programming languages start to get closer to natural ones. The logic of the space is slowly substituted by the logic of the text.

Hayles states that another advantage of object-oriented languages is precisely to hide the lower levels of coding, resulting in their "superior flexibility and ease of design" (HAYLES: 58). To Friedrich Kittler, that is actually software's unique purpose and reason of existence: to bridge between the environment of

everyday languages and the computer system – but in a negative way, rendering the machine transparent. In his essay *There is no Software*, Kittler makes clear that code is but a metaphor that “hides the machine from its users”.

These views cannot let us deny that, in the computer, there is no room for abstractions – no room for *language*. Just like the phonograph “can record all the noise produced by the larynx prior to any semiotic order and linguistic meaning” (KITTLER: 1999, 17), the computer system is absolutely physical. In its core, signifying strategies can barely be called so – “all code operations”, as we have already seen, “come down [...] to signifiers of voltage differences” (KITTLER: 1995).

Hence, code cannot be “the language in which computation is carried out”, as Hayles believes (HAYLES: 17), because *computation is not carried out in any language*. Binary states are not signifiers; they are not symbols of anything. They are *presence* and *absence*. So, although computation may only become fully material once it goes through some kind of semiosis, it has nothing to do with it in the first place.

It would be better not to think of computation in terms of language altogether, but in terms of *trajectories* – a flow that at the same time follows and materializes the insubstantial contours of the computer, giving it its particular form, the *algorithm*. This configuration sprouts from the processing core and produces the system’s structure up till its uttermost levels of abstraction.

Computation only exists while the system *runs*. The computer and computation are inseparable, because computation is the process of the computer itself. It is not something the computer *does*. It is the *manifested* computer. Likewise, software is never running *in* a computer. Software is the computer running *in a particular way*, which ends up being manifested in *surface effects*.

From the mechanic level upwards, increasingly abstract structures are being articulated in “interlocking chains of signifiers and signifieds, with signifieds of one level becoming signifiers on another” (HAYLES: 45). It could be said that this happens every time, in both directions – from the system’s core processor to its interfaces and back again, in a dynamic interaction that defines the ways the computer can be used and understood.

So, like keystroke signals and flickering pixels, code is an aspect of the computer’s materiality – it is how computation demonstrates itself and make itself analogically available. The same processes informed in the code are the ones instantiated in the computer, but in a different level of abstraction.

The way computation structures these layers could be described by the process Hayles calls *intermediation* – speaking in strictly formal terms, “the capturing of the lower level patterns as the primitives of the second level on top”.¹⁴ In the active heterarchy structured by intermediation, it is as if every form was the matter of a higher-level one; every pattern, just an object in a more complex visual field.

As computation, intermediation is discrete, in the sense that it is prompted within contiguous levels of abstraction. But, ultimately, it addresses the whole web of “entangled interactions between different system’s worldviews” (HAYLES: 31), i.e. the different ways analogous and digital subjects condition one another through feedback and feedforward loops.

The way intermediation creates a flow of subjects works similarly to Paul Virilio’s notion of *trajectivity*. According to Virilio, this vectorial condition is provoked by the relation of the nomad with the world, where the experience of movement predominates over the static dichotomy between object and subject (VIRILIO: 108). In intermediation as well, bodies are better viewed as processes that shape each other (HAYLES: 210). Albeit apparently whole and

¹⁴ HAYLES, N. Katherine. **Intermediation: In Pursuit of a Vision**. Conference at the University of California, Los Angeles. Departments of English and Design/ Media Arts. September 2006.

coherent, all layers of language and coding in the system are produced as the contours of the computer excited materiality, sustained by this fragile equilibrium of distances and positions we call *algorithm*.

Algorithms as *Dispositives*

Basically, an algorithm is a series of instructions that instantiate a given process. Flusser openly refers to them as *forms* (FLUSSER: 78), and indeed they are: albeit an algorithm may look different depending on the matter it is applied to (i.e., what kind and level of code it organizes and in which it is materialized), ultimately, it is the form of the computer. The algorithm structures the computer in a particular way that, once activated, produces computation.

Given the particular materiality of the universal computer, the form conveyed by an algorithm is never a *continuous* one. Even though it can be seen that way from a distance, it is just like *relief*: a dynamic arrangement of objects (*discrete presences*) in space (which, in binary level, is manifested as the sum of all *discrete absences*). So, the algorithm works less like a *text* than a *dispositif* – that which establishes a particular *disposition of terms* in a field.

Dispositif comes from the Latin *dispositus*, meaning literally *to put in place*. The Oxford Latin Dictionary is even more precise about it, referring to what is “placed at intervals, spread out” or “that is properly or regularly arranged or laid out (in order or formation)”.¹⁵ So, just like the computer cannot be told apart from computation, neither matter from the form in which it appears (since it is its very *appearance* – FLUSSER: 32), the *dispositif* is precisely this configuration of terms, which produces a particular effect once the system is activated.

¹⁵ It is also a term often employed by Michel Foucault to address the different mechanisms and structures that exercise power within the society. Alternatively, Foucault talks about *apparatus*, one of the central concepts to Louis Althusser’s structural Marxism, that gained much popularity because of its employment in the cinema theory of Jean-Louis Baudry, Christian Metz and Jean-Louis Comolli. We shall develop the implications of this coincidence in a future essay.

In the same way cellular automata manifests its successive generations of cells in the guise of an evolving pattern, once the system is informed by an algorithm and starts to run, the arrangement reconfigurates itself following what seems to be a logical behaviour. But, as we have already seen, in the universal discrete machine that is considered the generic model for computers (and whose logic all digital microprocessors follow), program and data – every action and object (SANTOS: 51) – are but the same thing: a stream of bits (KITTLER: 1999, 18). And the stream of bits, which in the lowest level of abstraction cannot be told apart from the processor, is also the input and output of computation.

Hence, simply by putting things into place, by creating a configuration of zeros and ones, the device produces its own updates, in a kind of first-order emergence – a behaviour that cannot be found in the system's individual components, but arises from their interaction (HAYLES: 198). And it would not be wrong to say that the particular effect of an algorithm is computation itself, recognized as it goes through all levels of coding and reaches the system's interfaces, eventually producing what Kittler calls *surface effects*: sound and image, voice and text (KITTLER: 1999, 1) – that is, the emergence of a *gestalt*, too: a form that surfaces.

This means that the system's behaviour is not produced by the actuation of forces from yonder. Quite the contrary, it is *immanent*: a manifestation of the computer's arrangement once activated, producing new states of the dispositif which are developed in more abstract levels of the system. The question that remains to be answered is what it is that activates the system, giving life to the algorithm.

Once again, it might be enlightening to evoke the notion of trajectory. The interesting aspect of the term coined by Virilio is how it emphasizes the *trajectory* – that is, the path inherent to the topography, coincident to the way through which it is traversed. The trajectory is a kind of negative of the

topography and, indeed, one of the ways it is experienced as a form.¹⁶ To run through space is also to *scrutinize* it, in a process that could be approximated to that of *parsing* – the analysis of a structure such as a program according to formal parameters.

In the case of Turing's device, this is made possible by the competence of the machine to slide the tape and to record/read positions in it (KITTLER: 1999, 33), creating this interplay of presences-absences we call *computation*. Ultimately, these operations could be explained in terms of the capacity of movement and the capacity of setting references in the field for future movement. In the digital microprocessor, as we have seen, it is not much different: the flow of electricity opens and close paths that will redirect the flow in future parsings.

So, as the name implies, to *run a program* is literally to put the algorithm (as a *form*) in motion. Or, following a reversal we already employed, *to put the computer in motion*. The form that thus surfaces is another one: not that of the territory's arrangement, but the contours of the movement within the territory – i.e. an *abstraction*. In the computer, spatial logic, as discretization, permeates every process, even the organization of the layers of code. In the end, movement constitutes the *system of actions* that complements the *system of objects* (program and data) under the dispositif: the effect of lower-level materiality on more abstract arrangements.

High-Level Topographies

As we have seen, the production of these high-level frameworks is usually designed to facilitate machine operation. The computer is throughout organized so that it can be informed by processes as close as possible to natural human communication, and produce equally sophisticated symbolic effects. Therefore, a final evidence of the arbitrary textuality of code might be that even the most natural-looking object-oriented languages only *seem* to

¹⁶ The other being its apprehension at distance, flattened, as a *surface*.

behave according to a natural grammar. Highly abstract code based on different degrees of spatial organization is in fact the norm.

This can be inferred by a closer look on *obfuscated code*, source code intentionally written to be very hard to be understood, usually to difficult reverse engineering. Even though it is not meant to be *read*, obfuscated code can still be compiled without any problems. Thus, it is revealing that some of the most often obfuscated languages are high-level (and easily legible) ones such as *C*, *C++* and *Perl*. Obfuscating dispose code of its apparent semantics, destroying the textual look of programs, and leaving behind just the field of characters the system runs through.

A work that exercises this tension between the readability of code and its effects is Jaromil's `:(){ :/:& }::` forkbomb.¹⁷ Although it may look like some sort of creepy emoticon, this small string of characters is actually a computer script. Once entered in an UNIX terminal, it starts to produce copies of itself in the computer's memory, eventually overloading the system and bringing it down. Its aesthetical interest comes mainly from the fact that, for most of the audience, the code has been stripped down of its semantic qualities – but not of its formal aspects nor of its capacity to inform the machine.

Other cultural practice that reveals the insuperable distance between code and text is the creation of *esoteric programming languages* – or *esolangs*, for short. They are usually made as a hacker hobby, either to test limits of design or as a form of entertainment. All of them are more or less *functional*, and some are even turing-complete, meaning they can simulate a Universal Turing Machine and, supposedly, instantiate any kind of computation. But they are not meant for real-world programming, because they are far from being *practical* – several are even naturally obfuscated.

Some esolangs just substitute the name of popular programming functions for more funny equivalents, and ask for the syntax of a specific kind of text –

¹⁷ < http://www.digitalcraft.org/?artikel_id=292>.

making programs look like recipes (as with *Chef*¹⁸), theatre plays (in the case of the *Shakespeare Programming Language*¹⁹) or maniac screams (AAAAAAAAAAAAAAAAA!!!²⁰), for example. But most of them are also based on a very reduced set of commands and operands, making programming works more as organizing elements in a grid than articulating signifiers.

A famous case is *brainfuck*,²¹ which uses just eight one-character instructions that either a) change the value of a *pointer*; or b) move the pointer in one direction or the other of the code. Although brainfuck code is usually broken in lines for readability, the instructions can just be aligned one after the other, making programs look surprisingly alike the one-dimensional discretized field of the turing machine.

Other esolangs are even more explicitly anti-textual, if not visual. *Whitespace*,²² for instance, is one that uses only whitespace characters (spaces, tabs and lines breaks) as instructions. But the most compelling example for us would be *Piet*,²³ a language inspired by Piet Mondrian's paintings. Programs written in Piet do not only look like abstract graphics: they are so. The basic unit of Piet code is a *colour block*. Thus, Piet interpreters do not recognize *ascii text*, like the other language's ones, but *bitmaps*. Moreover, they recognize them as fields: when the program is interpreted, the colour field is literally traversed by a pointer, which take colour blocks as references to update its value and decide where to go next. So, if programs written in brainfuck behave as turing-tape-like one-dimensional trajectories, those "painted" with Piet are more like two-dimensional paths.

A relatively different kind of programming paradigm that also follows spatial logic is *dataflow*. In dataflow, algorithms are organized as a diagram of instructions through which information circulates. But, contrary to esoteric languages, dataflow architectures are highly functional. In fact, they are

¹⁸ <<http://www.dangermouse.net/esoteric/chef.html>>

¹⁹ <<http://shakespearelang.sourceforge.net>>

²⁰ <<http://esolangs.org/wiki/AAAAAAAAAAAAAAAAA!!!>>

²¹ <<http://esolangs.org/wiki/Brainfuck>>

²² <<http://compsoc.dur.ac.uk/whitespace/index.php>>

²³ <<http://www.dangermouse.net/esoteric/piet.html>>

becoming increasingly well known among amateur programmers, such as artists, precisely because of their high usability. For that, we could even say that the way they model computation is turning into an important standard for the creation of artworks and installations.²⁴

One of the big responsables for this popularity is Miller Puckette, who developed the patcher system Max/MSP²⁵ in the 1980s. Max/MSP was originally created as an “authoring system for interactive computer music”, and that is why it employed a dataflow architecture in the first place. The dataflow paradigm is strikingly similar to that of electronic audio synthesis: the modulation of signal through a circuit made of different components assembled in series. Therefore, we could say that dataflow programming languages bring back to the surface of the system the electronic logic that work on the core of the digital computer. It is not a coincidence that dataflow programs – normally called *patches* – look a lot like electronic circuit schematics.

Due to its fluid structure, Max/MSP later became used also for video processing, and inspired the creation of similar environments such as *Pure Data*, *vvvv* and *Isadora*. Nowadays, all of these environments are able to handle devices external to the computer as well, making them widely used for all kinds of interactive systems. It is not improbable that this proneness for controlling multimedia installations comes from their inherent spatial logic. After all, since installations are likewise dispositives, they should be rather analogous to dataflow programs. This becomes easier to perceive in intermediate systems such as the *Reactable*,²⁶ an interface where the physical and the computational overlaps, producing something like a concrete dataflow environment.

²⁴ A good evidence that dataflow programming is entering domestic-consumer-level is that *Quartz Composer*, an environment for multimedia creation similar to Max/MSP, has been packed with all OS X releases since version 10.4.

²⁵ <<http://www.cycling74.com/products/max5>>

²⁶ <<http://mtg.upf.edu/reactable>>

But the complexity of these intersections between different-level arrangements is much evident concerning data matrixes, another spatial-based code. A data matrix is a kind of two-dimensional barcode, used to transmit digital information through physical or optical means. It consists of black and white cells arranged in a square or rectangular grid, which can measure from 8x8 to 144x144 units. Its storage capacity is relative to this resolution, up to a maximum of 2.335 alphanumeric characters.

One of the most common applications of data matrixes is to easily identify objects that go through different territories' borders, speeding up their circulation. A data matrix is placed on the surface of the object, to be later scanned by normal digital cameras, in order to be immediately processed by specific software. Often, it contains a website address about the tagged object. Therefore, a data matrix bridges different levels of spatial arrangement – the flow of consumer goods in the market, data processing in the digital computer, and the world wide web network topology – based on another, rather classical, one: the depth of field between camera and subject.

These imbrications can be taken to literally global scales, as in Bernd Hopfengärtner's piece *Hello, world!*. Proposed as "a real installation for the virtual globe of the software Google Earth",²⁷ it consists of the famous "Hello, world!" statement encoded in a data matrix of 170 meters side cut in a wheat field. Like a crop circle, or the notorious Nazca lines, this data matrix can only be seen as so from above. It is supposed to be captured by satellite lens and, sooner or later, get into the Google Earth software mapping system.

Hopfengärtner's work illustrates two important mechanisms underlying the spatial logics of the dispositif in the production of surface effects. First, how the interplay of presence and absence (of wheat) in the field is abstracted within a more complexly structured arrangement. Through distance, the relief is flattened and turned into an informed surface, an image pertaining to yet another architecture. At this layer, the trajectory of an immersed body (the

²⁷ <<http://hello.w0r1d.net>>

lawnmower man) is supplemented by that of the hovering gaze (the satellite lens).²⁸

On the other hand, it also stresses how these different trajectories are interconnected in a subtly balanced circuit, whose pivot is the multilayered code itself. This is manifested in the relation between the data matrix resolution and its physical size: the later has to be abstracted by the distance of the scanner (from space to image), so that the former can be processed by the computer system (from image to algorithm).

All of these instances are different surfaces of the same dynamic form, relative to different positions within the dispositif. Code is simultaneously instantiated in several arrangements, which together constitute the dispositif that will ultimately produce the desired information – a cheerful greeting to the whole planet, and everything else that might means.

²⁸ It might be good to point out that this gaze is also embodied, and this body is on its turn immersed in another dispositif.

REFERENCES

FLUSSER, Vilém. **O Mundo Codificado – Por uma Filosofia do Design e da Comunicação**. Cosac & Naify: São Paulo, 2007.

HAYLES, Katherine. **My Mother was a Computer – Digital Subjectivity and Literary Texts**. USA: University of Chicago, 2005.

KITTLER, Friedrich. **Gramophone, Film, Typewriter**. Stanford: USA, 1999.

KITTLER, Friedrich. **There is No Software**. Available at www.ctheory.net/articles.aspx?id=74. Originally published in 1995.

KNUTH, Donald. **Literate Programming**. Available at www.literateprogramming.com/knuthweb.pdf. Originally published in 1983

SANTOS, Milton. **A Natureza do Espaço – Técnica e Tempo, Razão e Emoção**. 2ª ed. Hucitec: São Paulo, 1997.

Virilio, Paul. **O Espaço Crítico e as Perspectivas do Tempo Real**. Editora 34: São Paulo, 1993.