# 1 Introduction

Today's new media theory increasingly invokes *materiality* as a significant, perhaps even *the* significant, mode of investigating digital objects and the media through which they are delivered. This thesis questions such a centrality of materiality through a practice-based, process-oriented approach. *Process* is proposed as the atomic unit of that which new media theory investigates. This is true on a formal material level: applications run as either as individual process or as assemblages of process which are managed by an operating system and through which the application's code is accomplishes all of its tasks, from memory and access to algorithmic execution on the central processing unit. A process-oriented approach will be shown to provide superior methodologies for engaging with and understanding software than material analysis alone provides. For instance, certain problematics within Lev Manovich's concept of 'media hybridity' will be resolved by a re-orientation towards process (Manovich 2008). Process also allows a fresh perspective for examining human-digital relations. Human processes and digital processes are seen as inextricably intertwined, leaving any discussion of digital process that excludes relevant dimensions of human process necessarily unfinished.

The method proposed to demonstrate these points is two-fold. The first is an analytic approach—the modes of operation of designers themselves are examined. Starting from the proprietary Mac OS X operating system, described here as a unique and powerful example of *process hybridity*, we progress to a discussion of the operations of designers as constrained by FLoSS (Free/Libre/open/Source Software). The second aspect of the method is a detailed interrogation of actual practice in the form of *digital typesetting*. This topic was chosen for several reasons. The first is a general lack of focus on the processes behind typesetting among new media theory—while the surfaces of text have been investigated in numerous ways (Bolter 2001; Fuller 2000), there has been a general lack of concern (or capacity) regarding the underlying processes of text in the metamedium (computers). This is especially evidenced as regards the *command line interface*, a realm where text becomes kinetic. Yet I found that very little theory has been written regarding the command-line, despite its place as the historical interface (once contemporary with batch punch cards) by which digital processes were initiated. Far from being obsolete, both Microsoft and Apple ship command line interfaces within their operating systems. In Microsoft´s case, significant money has been spent developing a new grammar and implementing new functionalities into their modern command line implementation Powershell (as opposed to the grammar and functionalities of DOS).

The second reason for choosing typesetting is the supposed lack of media hybridity of typesetting—according to Manovich's definitions of the terms, typesetting has failed

to move beyond ‘multimedia’ to a state of ‘media hybridity’ (this is opposed to typography, which undoubtedly has) (2008: 86). Media hybridity is Manovich´s formulation of the increasingly common ¨sharing of languages¨ between media. When media share language, they develop new dimensions (2008: 86). Language, then, demonstrates its capacity for modulation in a new context. While the proposition that ¨language can add dimensions to things¨ may at first consideration seem a bit too obvious for stating out loud, the kinetic properties of language within the context of the metamedium—that the code enabling the language sharing that enables media hybridity is *itself made of language and made executable by language*—seem to beg for consideration. Whereas much of the new media discourse relating to changes in media trends toward contemplating fast-paced visual cultures such as video games and cinema, this thesis aims to take the opportunity to contemplate the much slower-moving medium of text. This contemplation of screenic text leads to questions about the nature of media within a medium as well as to the introduction of a conception of processual hybridity that both underpins and exceeds the dynamics of media hybridity.

The third aspect is the allowance of a truly reflexive investigation in which multiple processes of digital typesetting are utilized to generate the thesis itself. This provides a means to integrate the process-oriented perspective into a software study of FLoSS typesetting software. Not only this, it provides a means to attempt what could be considered a *refractional* methodology. Inspired by Gilbert Simondon´s adoption of the language of chemistry in the formulation of *transduction* within his theory of ontogenesis, this thesis can be viewed as a distinct crystallization process, the composition of a whole from the process of that whole´s unfolding. The applicability of Simondon´s ontogenesis to matters of generative design will be interrogated in contrast to Jay David Bolter and David Grusin´s remediation theory (Bolter and Grusin 1996; Bolter 2001). Ontogenesis, albeit without Simondon, has already proven an effective angle for approaching Web 2.0 platforms (Langlois, McKelvey, Elmer, and Werbin 2009). Here the description of this thesis´own workflow will demonstrate Simondon´s ontogenesis as making unique contributions to the process-oriented perspective which this thesis attempts to invoke and instantiate.

The fourth is the simple fact that screenic text has not been interrogated on a *subtextual* level—surface analysis of text (and hypertext) have driven the discourse of screenic text in new media.

# 2 Screens

As digital typesetting provides the focus for the application of the process-oriented perspective, the point of origin is necessarily that of the screen. Information transmission is

increasingly screen-based, a fact that only intensifies with the exponentializing ubiquity of mobile devices such as the iPhone. The long-awaited advent of cheap "tablet" computers and e-readers is also now at hand. These devices may all be seen as mediums for *screenic processes* in that their entire configuration and all of its computation exists to serve as the basis for screenic interactions with *human processes*. These phrasings introduce the perceptual angle attendent with this thesis, namely the centering of *process* as the atomic unit of what is discussed in new media theory. The term *screenic* simply means 'screen-based,' or (perhaps) 'screen-native.' It is analagous to 'printed.'

One way to define screens is in terms of their interactivity. Some screens, such as television screens, offer very limited interactivity: the choice of content. This choice itself can be constrained by varying degrees, such as the number of available channels and playback formats (VHS, DVD, Xvid, etc.), even to the point of disappearing (in the case of many televisions that appear in public spaces).[1] The medium of the remote control should not be underestimated in its effects on human processes, to say nothing of the screens at which they are aimed. Indeed, they drive the interactivity of the video game consoles, an interactivity that clearly represents the cultural cutting edge of what a television screen can offer.

The computer screen, on the other hand, is defined by its seemingly limitless degree of interactivity. Remote controls can be run as screenic processes and can not only change television channels—processes on remote systems can be controlled with similar ease. Indeed, the entire screenic composition of one computer can be controlled over a network by a second computer using included, or easy to obtain, applications. Furthermore, the very interfaces to the screen (keyboards and mice) are examples of remote controls in cases where the screen has not itself become its own remote control (touch-screen devices). Typically the only element of a computer screen that the user does not effectively control are the structure and visual language of an operating system's graphical user interface (GUI). Even this, however, is generally accomplishable by a significantly informed user. In the case of GNU/Linux the task is not only accomplishable: in the case of a "from scratch" installation,[2] the user is literally forced to make a choice of GUI structure and visual styling. Microsoft has generally shipped their operating systems with multiple choices for widget[3] presentation, including re-mediations of widgets from previous

[1] Mobile devices are beginning to ship IR transeivers with full hardware access through software. That is, the *entire potential* of the IR spectrum is available to them.

[2] Such as is demanded by no-frills distributions such as Gentoo and ArchLinux, where manual installation and configuration of a GUI is required for use.

[3] A widget is the technical term for a GUI element. Scrollbars, titlebars, menus, and close/minimize/maximize buttons are widgets attached to most of the "windows" that appears on any given GUI-driven computer.

versions of Windows. Users also developed Apple, however, maintains strict control of widget presentation, especially on their mobile devices.

## 2.1 Screens as material, screens as process

Screens offer an ideal point of juxtaposition between the material and processual frames. From a material view, the very formulation of "screens" as *the* interface between humans and computers is problematic: what of the interfaces that have been developed to work around instances of blindness or other [disabilities] that prohibit visually screenic interaction?

From a processual orientation, the question becomes: how do interactions between humans and computers resolve themselves? The answer returns in the form of the *available* remote controls and the *available* response interfaces. The next step might be to investigate the degree of variance between these availabilities, and whether they problematize any umbrella-classification. While it would be *insensible* to argue that material differences in inputs and outputs can—or do—not lead to a huge amount of variation between experiences within humans. Such variation is likely to occur in differentials. That is to say, the spectrum of possible feedback occurs at the level of the human individual—one's experiences are functionally irrepresentable without translation of some kind. [We can choose to call these translations mediums, or we can choose to call these processes.]

At this point the question becomes, then, whether it is necessary to instantiate these inherent divergences in every evocation of a broad level discussion of input and output mechanisms or whether the inherent, *core* similarity between them all remains that in all instances they serve as *the point of contact* between human and digital processes. Does it make a processual difference if the output technology is a braille screen or an LCD screen? Only inasmuch as to what degree the process being examined is unique to, or highlights differences between, one or the other. From a discursive level, *controls* and *screens* can capture the essence of these dual "action spaces" that together form the single point of contact between human and digital process.

Is it possible to remediate of the term screen into discussions of previous mediums? For instance could one speak of the "screen" of a newspaper or the "screen" of a cave wall? What about the "screen" of a radio? From a linguistic-conceptual perspective the final example certainly pushes the limits. From a process perspective, though, the presence of the radio/what it is playing/what listening choices are available/how and to what degree does the hardware support frequency tuning: these questions can all be conceived in terms of 'control' and 'screen.' The sounds of a radio do emit, after all, from the vibrations of a stretched membrane.

This thesis proposes a conceptual-linguistic shift in the discussions of screens as the *site of discourse* through which digital processes yield the results of their execution. Likewise, the remote control, or simply *control*, is the site of discourse through which which human processes instigate and extend into the digital. There is no removing or reducing of this dyadic assemblage—even when the control and the screen are literally fused (as in most contemporary smartphones) the distinction between *control* and *screen* holds on both a conceptual and material level. Conceptually, human process still extends through the control into digital process, which still produces feedback through the screen. Materially, the screen is a Liquid Crystal Display driven by a graphics card that interfaces with coded drivers and display subsystems in the device's operating system. The control, on the other hand, is the glass suspended over the LCD which, through one or more of the multitude of available technical solutions for the process, reads point(s) of contact, pressure, and vectors (velocity and direction) of movement.

# 3 From Screens to Text

To discuss computer screens one must necessarily engage with the concept of *interface*, a topic that rightfully occupies a great deal of current new media discourse. Interface, then, represents one point of departure from our origin. While interfaces often utilize many visual metaphors (most of them inherited from the work done developing the first GUI at Xerox's Palo Alto Advanced Research Lab (PARC) in the 1970s), there are yet few computer interfaces that do not rely on text as their dominant mechanism for organizing and presenting a program's internal capabilities to a user. (Mobile screens, on the other hand, increasingly display developing trends of icon-only design, though the web browser remains a popular application). Despite the success of the GUI over the text-only command-line interface (CLI), text remains central to contemporary experiences of computer screens.

The command line is seen as a space of contestation for traditional modes of media analysis. Remediation, for instance, will be demonstrated as inappropriate for discussing the CLI. As Google has just recently released a command line interface for interacting with Google services, I believe a discussion of the command line is essential for new media (Holt and Miller 2010).

(Unfortunate to note, this historiographic aspect is still **'to-do'**:

The centrality of text to the experience of computer screens represents the main avenue by which we proceed from the origin, constituting a trunk from which many additional concerns fork away and then face examination. The arguments of the paper are augmented by the inclusion of a historiography of digital typesetting. Engaging critically

with the history of *software itself* is considered a requisite for responsible software studies: a full range of influences (economic, cultural, technological) should be considered in the re-telling of a given processual unfolding. In this aspect of focus, it extends Lev Manovich's admirable positioning of history as central to a software study by broadening the scope of historical considerations.[4] Inspiring this enagement is the work of Robin Kinross, whose *Modern typography: an essay in critical history* is one of but a few texts covering a history of typography to adequately engage with the influence of factors outside of that field on the field itself (Kinross 2004). By integrating a critical history of digital typesetting with a process perspective, an equilibrium between human and digital processes will be illustrated.

## 3.1 Recognizing the Ontogenesis in Generativity

In his text *The Position of the Problem of Ontogenesis*, Simondon writes,

> By transduction we mean an operation—physical, biological, mental, social—by which an activity propagates itself from one element to the next, within a given domain, and founds this propagation on a structuration of the domain that is realized from place to place: each area of the constituted structure serves as the principle and the model for the next area, as a primer for its constitution, to the extent that the modification expands progresively at the same time as the structuring operation. (Simondon 2009: 11).

Note the distinct lack of 'computational' in Simondon's list of operations. Written prior to the advent of Manovich's formulation of the age of cultural computing, this absence might simply be read as a matter of temporal context. Nevertheless, Simondon's solution to the ontogenesis problematic provides a framework for describing digital processes of a generative nature.

This leads to another important element of this thesis, one that runs throughout the entirety of itself—the underlying processes of presentation required to 'typeset' the text itself. Through the utilization of FLoSS software, multiple output formats will be not only be investigated but also materially instantiated through a designed mechanism of process—a *processual hybridity*. These output formats represent two of the top formats currently used to manage and display texts digitally: HTML and PDF.

The process(es) of their generation offers an attempt at mapping Gilbert Simondon's language of ontogenesis onto file format translation or, to begin the project immediately,

[4] **Note:** This work largely remains unfinished in this draft, as it became apparent that I needed to work back through more discussions of basic infrastructural elements such as operating systems in order to fully describe the assemblage of process upon which computer-based design is situated.)

*individuation*. Coupled with Simondon's individuation is this concept of *transduction*. Repurposed from the language of chemistry, Simondon's metaphorically images transduction with the example of a substrate—swelling with *metapotential*—that crystallizes. The final formation is the substrate fulfilling this metapotential, a fulfillment that arises only through an unpredictable unfolding involving emergent factors. (The language of chemistry was likewise appropriated for the term 'interface' (Cramer and Fuller 2008: 149)).

Through this mapping I hope to provide a convincing argument for shared properties between what I am calling process and individuation, and between transduction and what I am calling instantiation.

This relates with the increasingly generative nature of contemporary design. All of which are generated from a plain-text file whose syntax conforms to a format standard called 'markdown.' The polycephalous nature of *the text itself* thus demands further branching into a discussion of formats. What are the attributes of the class of process to which formats belong? Formats are seen as stable, yet they move like glass (or glaciar) in the nano-magnitudes of the digital. Formats provide another point of contrast between process and material perceptual orientations.

The discussion of generativity provides further means to demonstrate the equilibrium of human and digital processes. Analyzed materially, these processes are chunks of code electronically lifted from hard drive platters, loaded into system memory, and then executed via the assemblage of chips on the computer's motherboard by way of instructions from the operating system currently residing as a mass of memory heaps in RAM chips. Analyzed *processually*, however, these digital processes are properly seen as deriving from interactions with human beings. That is to say, digital and human processes are intimately intertwined, from the design of their physical landscape of execution (microcircuitry) to the instructions derived from the user. From a process angle the computer becomes something of an external nervous system, extending and modifying the realm of human potentiality even as it surpasses the capacity of a single mind to functionally comprehend the entirety of its workings.[5]

### 3.1.1 Print is static, code is process

The flat/deep distinction proposed by Hayles is, by its formulation, material. Problematizing this material focus is the interwoven history of text and code: the lens of typesetting allows us to focus on a unique intersection of the two. As the historiographic case will

[5] The chips produced by Intel, for example, are too complex for any single person to ever hope to entirely understand.

demonstrate, typesetting is a *non-reducible* process (NP-Complete). This non-reducibility of typesetting reflects the non-reducibility of computational processing of language, as well as the non-reducibility of language, as signifier, into that which is actually signified. This "turtles all the way down" scenario has intriguing implications from a process perspective as we investigate the methods that have been developed in order to work around this non-reducibility.

When Hayles states that "materiality thus cannot be specified in advance; rather it occupies a borderland—or better, performs as connective tissue," she is provisionally correct (Hayles 2004: 72). However, this metaphor-ization of process is exemplary of new media practices: reference the complex with an abstract metaphor, obscuring complex and important dynamics with a metaphor. The metaphor works, to be sure. One could even consider it an ideal formulation. At issue is the fact that this borderland is not discussed in a technically correct manner.

# 4 Remote Controls

I think it may be reasonable to take the remote control and use it to create a metaphor for all human-computer interaction.

Every digital process has, at its origin, a human. The rate of computation has increased the impact of human-digital processes in that the results deliver their results faster. The results will either match the intentions of the originating human process, or they will not. In the second case we can find the first evidence of the effects of digital process on human process: *the code behind the digital process will be re-arranged in an attempt to deliver an output that satisfies the intention of the human processes.* Whether this modulation of the executed code is through sliders/input boxes/etc within a GUI interface or through direct reworking of the source code itself, the effect is the same: the code executed has been re-configured according to the goal of human process. The result(s) of the digital process, experienced through a screen, can match, exceed, or fail this goal. In turn, human process is effected and the next move is made according to new goals or revised digital processes.

Video games, for example, can easily be represented by this model. Human process is obviously shaped by digital at the outset: there are a finite number of actions that a game offers within its context. In addition, these actions are often presented as pre-set mappings of action to controller button.

# 5 Attributes of Process

[It needs to be asserted that I am willingly engaging in my own appropriation of the term 'process' outside of any traditions other than my own. As the process oriented perspective arose under the looming shadow of the draft deadline, I admit to a lack of historical perspective on the use of this word in either new media or other contexts. Withstanding that, however, I sense a real applicability of this term in the discourse of new media. I'm looking forward to working on the final draft and using some of that time to construct historical perspective for this shift to process. One important angle is Ned Rossiter's work on "Processual Media Theory" in his book *Organized Networks* (Rossiter 2007: 166–192), which this draft does make use of but which I would like to interweave more deeply. As it stands, this introduction was left relatively alone for the sake of fleshing out the middle part of this thesis. This was at the suggestion of the second reader.]

Process is reflective. It's outputs reflect its inputs. Additionally, process reconfigures the metapotential in any given system. It's reflectivity, then, has material effect. As it reflects the inputs into the outputs, the outputs in turn reflect new (or else simply different) potentials back into the *context* which is the reciprocal contact point in which the processes began. This language is extrapolative into any set of intersections. This paper considers just the subset of human-digital recipricity, and within the relatively static domain of typesetting.

A new configuration of metapotential in any system results in the reconfiguration of (all) other systems as well. This fact reflects the *fractal* nature of process—there is a degree of non-reducibility inherent in any discussion of process, as ultimately certain factors in its functioning are unknown to us.

# 6 Why free software?

There are multiple points of consideration that lead me to concentrate on free software. The first is its relative lack of presence within new media circles. Time and again I arrive at a conference only to see a room full of computers booted into proprietary operating systems. While I am not a 'zealot' who disavows any potential use or need for proprietary software, I find the general population of new media's reliance on proprietary operating systems—chiefly, by way of personal and anecdotal evidence, Mac OS X—disturbing. Hans Magnus Enzensberger outlined in his "Constituents of a Theory of the Media" the importance of issues of control with relation to mediums. Let us move through the juxtaposed elements of repressive versus emancipatory uses of media which Enzensberger

provides and interrogate them in relation to Mac OS X and GNU/Linux (Enzensberger 1970: 269):

*Repressive versus Emancipatory*

**Centrally controlled program vs. Decentralized program** This question is answered by asking the question: "Where is the source code of the operating system?" In the case of OS X, the source code resides only within the confines of Apple's corporate computers. It is likely heavily guarded by multiple mechanisms. Whereas in the case of GNU/Linux, the operating system source code is spread across dozens of mirrors on the Internet as well as the computers of programmers and users around the world. Each of these copies can be readily modified to the designs of any given user, demonstrating decentralized (in fact, distributed) control. Apple maintains sole, central control of the code and thus fully determines the functional possibilities of the operating system.

**One transmitter, many receivers vs. Each receiver a potential transmitter** This is already demonstrated above: the code for GNU/Linux is globally distributed across hundreds of thousands of computers. Each one of these has the ability to modify the software and share those modifications with anyone who will listen. OS X can be modified by no one.

**Immobilization of isolated individuals vs. Mobilization of the masses** OS X encourages the use of proprietary applications. These applications have restrictive license that generally allow only one individual the right to run the application. GNU/Linux, meanwhile,

**Passive consumer behavior vs. Interaction of those involved, feedback** A major advantage for both users and developers in a free software ecosystem is the feedback that occurs between them. Users may suggest new features at any time. If they have the skill and/or time, they can add these features themselves. If the addition of the features is contentious in any way, the contributer can simply fork the codebase and continue evolving the software in new directions. In OS X, you run the binaries you are given.

**Depoliticization vs. A political learning process** Mac OS X is pro-capitalist and promotes consumer culture. It can probably be said that it is politically "neutral" in its codedness, but this very codedness remains obfuscated and proprietary. GNU/Linux, in conservative judgment, at least does not actively promote consumerism. In an idealistic formulation, it destabilizes the capitalist

ecosystem.[6] It's politics are as multifaceted as its user base. In its well-deserved reputation as 'taking some work to make it work,' GNU/Linux forces its users to become active in the system's administration. This induced learning of an open approach to computer systems could be said to have political dimension.

**Production by specialists vs. Collective production** This seems self-explanatory.

**Control by property owners or bureaucracy vs Social control by self-organization** Are you getting the picture?

In a presentation at the Libre Graphics Meeting 2010, Florian Cramer explains his theoretical positioning of free software as an entry point into media criticism. Aymeric Mansoux, also of the Networked Media design faculty at the Piet Zwart Institute and present with fellow faculty member Michael Murtough, describes the critical engagement in the error message common to GNU/Linux distributions, found in the Totem media player program complaining of a missing codec library that is required to decode common patent-encumbered media formats such as MPEG-Layer 3 (Cramer, Mansoux, and Murtaugh 2010). Behind the error message lies an assemblage of inter-related issues of intellectual property rights, cultural practices, and media accessibility. This is a clear instantiation of a "political learning process."

On 21 June 2010, Apple changed its privacy settings to allow the company to "collect store and share 'precise location data, including real-time the geographic location of your Apple computer or device' " (Marco 2010).

## 6.1 Caveats

Free software is not, however, a "magic bullet"—tied to the open systems theory which is philosophically related to the underpinnings of the Chicago school of economics, some of the philosophical foundations of free software, *and especially open source*, need to be interrogated (Cramer, Mansoux, and Murtaugh 2010; Pasquinelli 2008). Liberation does not automatically lead to a distribution of tools to all those that need them. However, even in this instance we see the power of FloSS in its capacity to inspire critical engagement with media.

[6] It is important to note that free software also plays a significant role in supporting this infrastructure, as the license provides no recourse on the terms of the softwares use (Pasquinelli 2008).

# 7 Substrates of Digital Process

In his outlining of the nature of a 'processual media theory,' Ned Rossiter asserts that "a processual media theory examines the tensions and torques between that which has emerged and conditions of possibility; it is an approach that inquires into the potentiality of motion that underpins the existence and formation of a system" (2007: 178). To that end I have deemed it important to begin at "a basis" of digital process as it stands today. Computers have multiplexed dependencies. The hardware is virtually non-functional without an operating system. Likewise operating systems, like all software, are non-functional, even "non-existent," without the presence of hardware. As the coded substrate for software, the operating system provides the sites of intersection between human and digital process.

# 8 Alan Kay and a `Metamedium´ Vision for Personal Computing

In his text *Software Takes Command*, it is Manovich's inclination to focus on the work of Alan Kay at Xerox PARC when discussing the development of *cultural software*. He notes that there are multiple entry potential entrypoints for consideration: the work of Douglas Englebart and his team, the development of the LINC computer at the MIT´s Lincoln Lab, and Donald Sutherland´s SketchPad. The development of the Xerox Alto, however, is unique in multiple ways. First and foremost is the architecture of the software: by developing and employing an object-oriented approach to software design, users were positioned as inventors of new media through their ability to design their own interfaces that both enabled and spurred new modes of creation native to the screen. These screenic modes of creation represented a new, vital dimension to computing—the willful, *shaping into existence* through design and implementation of new digital processes. Kay´s team was specifically dedicated to applying intersections between education and computation. In the process of teaching the system to children and adults alike, those they taught often ended up developing their own unique applications out of the *objects* that could be shared between applications as well as extended through the inheritance model of object-oriented programming.

Cultural software—and the *cultural computing* which it facilitates—is defined by Manovich as software that is "directly used by hundreds of millions of people" and that "carries 'atoms' of culture (media and information, as well as human interactions around these media and information)" (2008: 3). Alan Kay is a pioneer figure in the computing world, an

individual who not only theoritically formulated a vision of the computer as a 'metamedium' but also did a great deal of practical work in order to achieve this vision. Unfortunately, as is all too common in the lives of visionaries, key elements of Kay's ideal never breached into the mainstream even as other aspects where appropriated and commodified wholesale by Apple and Microsoft.

The first, and probably most important, element of Kay's original platform that failed to transfer from his ideal "Alto" personal computing platform to the commercial GUI-drive operating systems that now define the landscape of cultural software[^foss_exception] is the concept of *easy programmability*. To this end Kay founded the basis of all the Xerox PARC work in personal computing on a programming language called Smalltalk. In his text "The Early History of Smalltalk," he explains that the computer science work of the 1960s began to look like "almost a new thing" as the funding of the United States' Advanced Research Projects Agency (ARPA) led to the development of techniques of "human-computer symbiosis" that include screens and pointing devices (Kay 1993). Kay's focus became investigating what, from this standpoint of "almost a new thing," what the "actual 'new things' might be." Any shift from room-size, mainframe computers to something suitable for personal use presented new requirements of "large scope, reduction in complexity, and end-user literacy" that "would require that data and control structures be done away with in favor of a more biological scheme of protected universal cells interacting only through messages that could mimic any desired behavior" (Kay 1993)[smalltalk]. [smalltalk]: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

One of dynamic typing's greatest attribute—and perhaps at least a part of why some choose to object to it—is that it makes the programmer's job easy. Whereas *statically typed* languages such as Java or C require the declaration of the kind ('type') of data the programmer expects to store in a given variable, a dynamically typed language has no such limiting expectations. Instead anything can be stored within any variable, something which allows for a great deal of flexibility in the hands of a programmer.

Object-oriented programming, though, represents an even larger shift in the potentials presented to programmers. Object-orientation metaphorically maps the concept of 'objects' onto the practice of programming, providing a means to encapsulate sections of code in a way that encourages reusability and thus increases programmer productivity. It is also said to be easier for the human mind to learn to think in the metaphor of objects over, for instance, the functional programming paradigm that is heavily influenced by the lambda calculus and which represents relatively steep initial barriers to those for whom discrete mathematics is not a second language. Since objects can contain other objects as well as maintain inherited relationships with yet other objects (using the abstract form called a 'class,' which is a pre-instantiated description of what an object will contain once

it is called into existence), the potential for constructing programs out of the pieces of other programs was greatly increased.[^I will return to this topic when we reach discussions of 'process hybridity.']

The last attribute of Smalltalk on this list, its *reflectivity*, simply means that every object can explain its own properties, i.e. which "methods" (blocks of code oriented towards performing a specific task) it provides, what variables it is keeping track of, etc. The result is that "objects can easily be inspected, copied, (de)serialized and so on with generic code that applies to any object in the system" (*Smalltalk* 2010). This capacity for writing "generic" code once again has a positive impact on programmer productivity, as demonstrated by Kay and Goldberg's description of a hospital simulation designed to be able to compute "any hospital configuration with any number of patients" in their ground-breaking essay presentation of their work, "Personal Dynamic Media" (Kay and Goldberg 1977[^kg]: 400). [^kg]: The page numbers refer to the article's position within *The New Media Reader*, published in 2003. The original date of publication has been maintained in order to respect the publication's true chronological position in computer history.

This digression into the specific mechanisms by which Smalltalk eases the tasks of programming highlights the focus of Kay and his group (the Learning Research Group) at Xerox PARC. The intentions of their research—as Kay says in a passage quoted earlier—was a paradigm shift on the level of the printing press. Given the degree to which the mechanics of contemporary computing received their "start" within the Learning Research Group as well as the pervasive impact of that contemporary ("cultural") computing, it can reasonaly be said that they achieved this goal. Further testament to the success of their system was the fact that "children really can write programs that do serious things" (Kay and Goldberg 1977: 394). Indeed, a "young girl, who had never programmed before, decided that a pointing device *ought* to let her draw on the screen. She then built a sketching tool without ever seeing ours" (399).

There is, however, one major element of their implementation—central to the capacities for children programming—that did not find manage to find its way through the commercialization of their ideas via Apple and Microsoft: that of unlimited access to the building blocks of the system itself. While Kay clearly saw not only the potentials but also the real, invigorating effects of providing "exceptional freedom of access so kids can spend a lot of time proving for details, searching for a personal key to understanding the processes they use daily" (404), unfortunately for the un-folding of cultural computing neither the Apple Macintosh nor Microsoft Windows shipped with the capacity for unfettered tinkering with the platform itself. Even worse, they generally shipped without the programming tools required to simply make a program on one's own, irregardless of the

degree of availability of “building blocks.” Considering the enthusiasm with which children interacted with Kay’s system, it is hard not to question where computer literacy—the ability to read *and* write within the computer—in the age of the GUI might be should either of those seminally-positioned products have shipped with the kind of development environment to be found in Kay’s largely-forgotten prototype.

— kay and the ipad http://www.tomshardware.com/news/alan-kay-steve-jobs-ipad-iphone,10209.html

## 1 Operating Systems as the Substrates of Contemporary Design

This line of inquiry into lost historical possibilities serves only to highlight the serious degree to which history affects the unfolding of the metamedium that we call a computer. Historical developments have also lead to further opportunities, as well. The manifestation of the metamedium as a space operating nearly exclusively on proprietary software in the 1990s drove many freedom-minded individuals to begin, and contribute to, free and libre software. What at first approach looked like an impossible task (Richard Stallman’s pledge to implement a completely free implementation of Unix called Gnu’s Not Unix (GNU) OS) has resulted in a snow-ball effect that provides not only the backdrop for much of this thesis’ discussion but indeed the GNU project provides the compiler with which Apple builds its invisible, proprietary codebase into the commodity known as Mac OS X.[7]

==== The evolving nature of operating systems ====

The conceptions and roles of operating systems have been evolving along with the hardware. Originally the operating system of a computer was intimately tied to its hardware. It was the development and *dissemination* of Unix by software engineering luminaries Ken Thompson and Dennis Ritchie that sparked the ontogenesis of operating systems as a ¨hardware independent¨ process, enabling *cross-platform code* as a dimension in the software process. While operating systems had always existed to abstract the system to some extent, for instance to hold segments of code for handling routine tasks (incidentally, these segments of code are often called `code routines´) such that they need not be input every time that operation is to be performed, the introduction of Unix accelerated this process of abstraction.

[7] In his text “The Ideology of Free Culture and the Grammar of Sabotage,” Italian media theorist Matteo Pasquinelli provides a critical analysis combining Marxist discourse on *resource extraction* with Michael Serres’ conception of the *parasite* in an analysis of contradictory intersections between capitalism and FLoSS (Pasquinelli 2008). Apple´s investment in developing a competitor to the GNU compiler, LLVM, is the result of Apple´s decision to avoid the GNU Public License as much as possible (LLVM is licensed under an easily-coopted open source license).

In 1999, more than twenty years after this milestone in abstraction, speculative fiction writer and trained programmer Neal Stephenson released a novella-length essay entitled ¨In the Beginning, Was the Command Line¨ (Stephenson 1999). This essay delivers an analysis of (consumer) operating systems and a critique of the reign of the graphical user interface. By means of distinguishing the cultures and manifestations of the operating systems—Microsoft´s Windows, Apple´s original Mac OS, Be´s BeOS, and GNU/Linux—Stephenson offers cars as analogy. Mac OS becomes an expensive European car, streamlined but expensive to maintain. Windows becomes a station wagon, the un-sexy, utility-oriented choice of the masses. The (now-defunct) BeOS is a Batmobile. And GNU/Linux is a tank without a price tag.

Computer programmer Garrett Birkel, writing in his update/response to Stephenson—¨The Command Line in 2004¨—notes the extreme pace of evolution of computer hardware. Despite the expanding set of elements contained within a personal computer ¨we don't call our new Dell machine a computing collective´. We consider it one device, with a singular name. And our concept of anOperating System´ has evolved right along, neatly obscuring this complexity¨ (Birkel 2004). This leads Birkel to clarify that what Stephenson´s text concerns itself is not so much the distinctions and separations between hardware and software as the mode of interactions with which we engage information:

*****-T-he crucial separation here is not between the computer (hardware) and the Operating System (software). Those are so deeply integrated these days that they have effectively merged into the same beast. The crucial division is between ourselves, and our information. And how that division is elucidated by the computer, with hardware and software working in tandem, really determines how useful the computer is to us. In other words, it's all about User Interface, and even though "In the Beginning, There Was the Command Line", it's also true that In The Beginning, Information Took Fewer Forms. (Birkel 2004)

The operating system is the initial site of formation for the processes that define this division between ¨ourselves¨ and ¨our information,¨ but it is far from the end of story, and perhaps far from the most influential. I will return to Stephenson and Birkel in the next chapter, which begins with a discussion of the command line. For now, the point has been made that there is no true means of separating hardware from software—without an operating system, there is no means for hardware to access its own functionality. The materiality, then, of a personal computer is inherently tied to process—without an OS, the PC is not a metamedium, it is dead weight.

===== Unix and Code Drift =====

The capacity of Unix to migrate across hardware while maintaining a consistent interface to both users and programmers has been criticized as well as lauded, for example

in the *UNIX-HATER´s Handbook* wherein the authors liken it to ¨a computer virus with a user interface¨ (Garfinkel, Weiss, and Strassman 1994: 4). The author´s complain that the portability of Unix results from an under-designed infrastructure, which they call *incoherent*. The virus metaphor works, for them, because of this portability (whatever its origins), the ability to ¨commandeer the resources of hosts¨ (of which there were, even at the time of their writing, many), and Unix´s capacity for rapid mutation (Garfinkel, Weiss, and Strassman 1994: 4).

This capacity for rapid mutation is evidence of Unix´s *evolutionary superiority* over other operating systems, which the UNIX-HATERS note is not concomitant with *technical superiority* and, they assert, in the case of Unix the balance is far too the side of evolution over technical soundness (6). They decry the introduction of ¨junk genetics¨ into the Unix codebase, resulting in either vestigial code for devices that no one uses anymore or divergent implementations (mutations of mutations) (7). This idea of ¨code drift¨ has been recently invoked by Arthur and Marilouise Kroker as the ¨laws of motion¨ of a digital cosmology (Kroker and Kroker 2010). Unix, then, could be considered an early object governed by this law of motion, from its initial existence as a system shared freely in code across research institutions to its fractured existence as inconsistently implemented proprietary offerings to its resurgence as free (GNU/Linux) and open source (Free/Open/Net-BSD) software.

The introduction of an ontological premise for software, Richard Stallman´s *free software*, has multiplied the evolutionary superiority of operating systems by orders of magnitude. Though his GNU operating system project is, as a whole, unfinished after more than two decades of development, the environment surrounding that effort—the GNU compilers, the libraries of code upon which they depend, and a vast array of available commands and applications—provides a ¨genetic library¨ from which one can extract whatever processes are useful for one´s own ends (provided one follows the rules of the code´s licensing terms, the GNU Public License (GPL)). As a result, GNU code appears in not only virtually every non-proprietary platform, it also ships with, and provides important underpinnings to, that flagship of the designing class: Mac OS X.

## 8.1 Mac OS X: Process Hybridity in Action

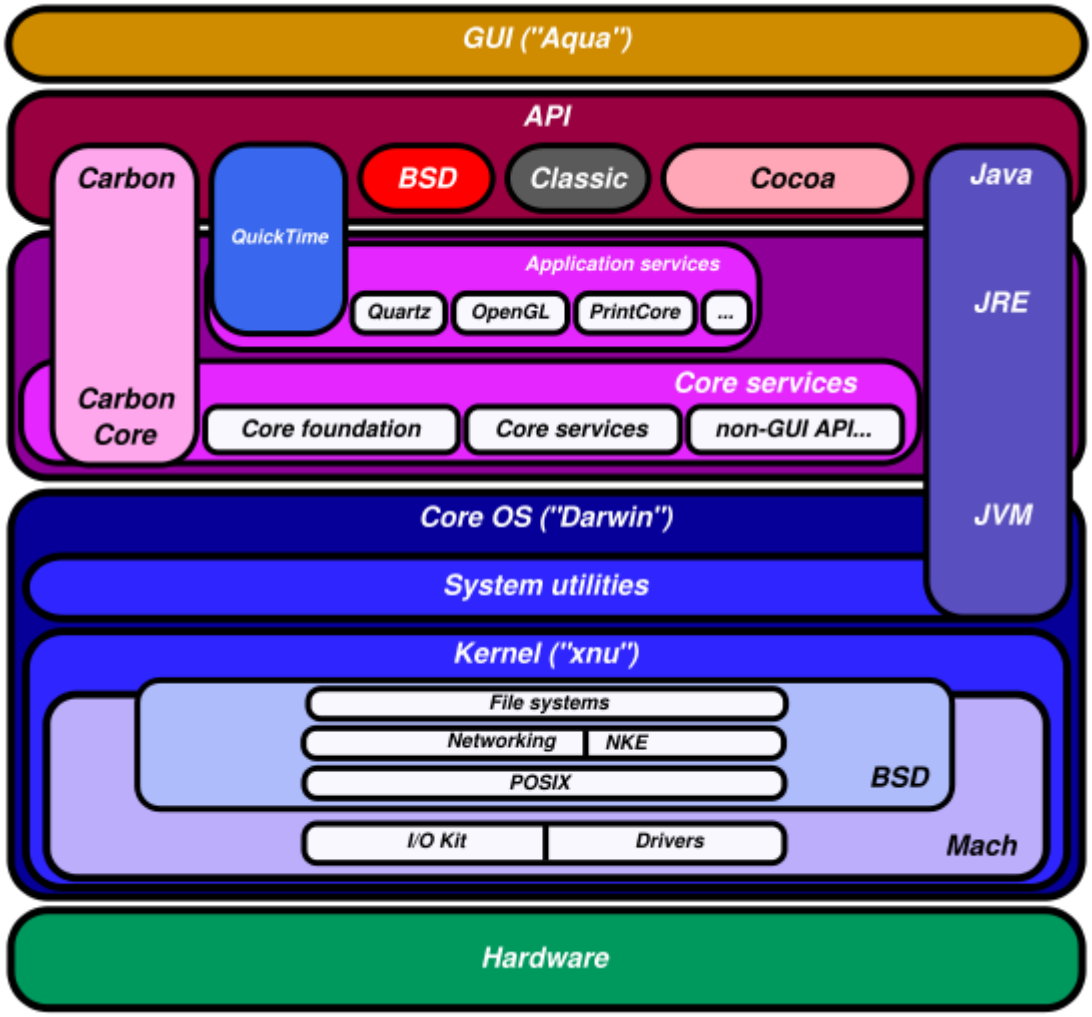
As the Apple platform has long been the preferred environment for professional designers—since before OS X's first release in 2001—it necessarily invokes itself as an important object of study. It allows us to investigate the concept of *process hybridity* that I introduced within the introduction. Mac OS X is a multi-layered system that contains both open and closed source elements with distinct historical lineages. As an assemblage, it can be

represented as containing a high degree of process hybridity as it combines not only various separate projects into unified commodity, it also bridges ideological boundaries by combining proprietary, open, and free code into a single commercial object.

This hybridity reaches into the very core of OS X, into the kernel itself. The kernel of an operating system handles the specific task of delivering the instructions deriving from software to the specific pieces of hardware that pertain to those intructions. A simplistic analogy can be made to a traffic cop, whose role it is to direct traffic in a way that the competing interests at an intersection proceed in an orderly fashion. Kernels come in two flavors: the *monolithic kernel* and the *microkernel*. The distinction between the two resides in the size of the tasks they are expected to perform, as well as a material distinction between where operating system code should be executed (*kernel* memory or *user* memory) (*Kernel (computing)* 2010).

Without entering into a lengthy discussion of the differences between the two—a discussion that has sparked a great degree of vitriol between their respective advocates—let us note that a monolithic kernel contains the code for everything from networking to video display drivers to encryption mechanisms. A microkernel, on the other hand, delegates these processes to *servers* that run outside of the kernel's memory space in the user memory (or *userland*, as it is often called). The result is that there is a great deal less chance of crashing a system that runs on a microkernel, as a bug in user space cannot affect the operation of the hardware whereas a bug in kernel space necessarily can. Microkernel's are also much more modular, as servers can be replaced with (theoretic) ease due to the fact that it sits outside the kernel's code. OS X's kernel, however, is a hybridization of the monolithic and microkernel designs (*XNU* 2010). Due to speed concerns, it was decided that elements

of the FreeBSD project's monolithic kernel would be welded onto the Mach microkernel.




Moving up from the kernel we encounter the system utilities. These are directly imported from BSD Unix, specifically FreeBSD. This is the last layer of what constitutes the "core OS", which is released under an open-source license. It must be noted, however, that the vast majority of code within this part of the operating system was licensed under terms that allow Apple to withhold any and all code whatsoever (i.e. *open source* licenses such as BSD or MIT). The major exception to this is the GNU C Compiler (GCC) and its attendant libraries, which are *free* (as in freedom) software and is licensed under terms that force Apple to make available the code of their modifications whenever they publicly release new binaries of their version of the GCC. On the other hand, specific improvements to BSD-licensed code is available only at Apple's whim. It is impossible to monitor whether the source code they make available represents all the improvements

they have made or whether they have been selective with their source in order to maintain their proprietary advantage.

Above the core OS level we find further evidence of OS X's storied history. The roots of Mac OS X reside in the operating system NeXTSTEP, developed at Steve Jobs' NeXT, Inc. after he left Apple in 1985. Like all GUI-based operating systems, NeXTSTEP necessarily incorporates a large number of metaphors developed first at Xerox PARC. In addition, however, it was built from the ground up to be easier to develop applications for than other operating systems at the time. To this end the Objective-C language, object-oriented and heavily influenced by Smalltalk, was adopted. While certain systems (such as the microkernel) might be written in C, applications were developed in Objective-C and all the servers above the kernel spoke exclusively with applications in this language. The programming environment was also a leap forward in ease of use: GUI windows could be designed in a WYSIWYG fashion, where their component widgets (buttons, menus, etc) could be easily tied to blocks of code which would inform the computer what was to be done upon a given widget interaction. NeXTSTEP thus represented somewhat of a resurgance for Alan Kay's original vision of personal computing. However, NeXT's first computers were priced at $9,999 dollars—resulting in slow adoption outside of academic or institutional contexts. It was within such a context that Tim Berners-Lee developed the first web browser, WorldWideWeb, on a NeXT computer.

Three other hybridities occur within this top level of Mac OS X: Carbon, Classic, and Java (the programming environment of NeXTSTEP was renamed Cocoa in OS X). Carbon was developed as a new way of writing Macintosh applications that would allow developers an easier transition path from Mac OS Classic to OS X. Applications written for Carbon could run on both the older Mac OS as well as OS X, allowing companies like Adobe to refashion their code without changing programming languages. Classic was an emulation layer that allowed "non-Carbonized" Mac applications to run on OS X (this hybridity has been removed since Apple's transition to Intel microprocessors). Java is a programming language and environment designed to allow universal execution of programs regardless of underlying architectures: if Java has been ported to an OS, then theoretically any Java program can then be run on that OS[8].

The hybridities present in Mac OS X have material effects from the execution of code to the variability of its cultural enablement. It allows for a vast assortment of applications from various lineages and paradigms to coexist and interoperate within a single operating

[8] However, there are many different versions of Java, making this ideal of "write once, run everywhere" somewhat problematic. Microsoft was even successfully sued for trying to hijack the language with by leveraging an incompatible implementation through the virtual industry standardization of its Visual Studio development environment.

system context. Short of labelling each of these hybridized elements a “medium,” there is currently no proposed language within new media for articulating the hybridity of the assemblage as a whole. In this fashion, *process hybridity* provides a mechanism with which we can describe the facets of OS X. Perhaps UNIX is a medium, perhaps not. However, as a standard, it can be recognized as a formalized resolution of a design process. In an environment where 3D cinema is not even presented as a new medium in popular discourse, the term itself appears to me to have lost its utility. Further in the thesis I will introduced more localized and dissectable instances of process hybridity. At this point it should be convincingly demonstrated that Mac OS X hybridizes not only material processes (code) but also ideological processes (licensing terms). This allows OS X its status as the only successful, commercial UNIX for a desktop as well as its status as an easy-to-use, GUI driven development platform (something no other UNIX can claim as convincingly), while maintaining a clean integration with the lineage of previous versions of Macintosh operating systems.

# 9 Design Beyond the Proprietary

Much of the prominent research within the field of software studies thus far has been dedicated to investigating the operation of various proprietary applications (Fuller 2000; Manovich 2008). Culturally speaking this may make sense: study what the users actually use. However, from a moral or political angle it can easily be seen as irresponsible or, at the very least, incomplete. Use of proprietary operating systems and application software still seems to dominate the field of new media. This is in spite of Hans Magnus Enzensberger’s classic critique of ‘repressive’ media—if you do not control the mechanisms of your medium, those mechanisms can be used to control you.

Whether for political or economic reasons, there is a small, yet increasing, number of designers and artists who have begun to use FLoSS either exclusively or at some stage in their workflows. Economic considerations should not be underestimated: Adobe charges no less than $599.99 for a new (non-upgraded) version of Photoshop, while bundles of their software sell for $1699 (Production Premium CS5) or $1899 (Design Premium CS5).[9] That makes Photoshop more expensive than an entry level desktop system, and the bundles each more expensive than a higher-performance computer! The long-standing solution to this situation is simply piracy. The newer solution, really only available for the last decade or less, is to use FLoSS software to accomplish these goals instead.

[9] All prices sourced from the Adobe online store on 18 June 2010 (**http://store1.adobe.com/cfusion/store/index.cfm?store=OLS-US&view=ols_cl&nr=0**)

Though at a superficial level one can highlight FLoSS design programs as mere reproduction of proprietary process—the GNU Image Manipulation Program (GIMP) reproduces the process of photo editing that is the domain of Adobe Photoshop—Florian Cramer declares that free softwares are not simply cheap imitations or reproductions of proprietary software (Cramer 2010). Instead, he argues, they represent new and unique avenues for accomplishing a given task.

The degree to which the shape of our tools shapes that which we create with those tools has long been established within media theory, though the focus on the dynamics of given medium specificities often overshadow the processes through which those media are filled with content.

Why do relatively so few professional designers use FLoSS? For designer and free software advocate ginger coons[10], the answer lies in a feedback loop between academia, industry, and the designers themselves:

Graphics professionals learn proprietary software in school because the industry runs on it. Employers demand knowledge of specific software because it is the norm. Connected industries, like printing, run on those de facto standards because of their clients. This feedback loop cements the place of proprietary software in the graphics industry. (coons 2010)

Part of the problem lies in the evolutionary nature of the open source development model. Whereas proprietary programs are expected to be "feature complete" with any given release, FLoSS integrates features as they are added. Any potential proprietary competitor to Adobe Photoshop, for instance, has to ship its first version under the expectations of proprietary software: a) it should offer at least the features that are used every day in Photoshop; and b) it should be stable. Otherwise there is no reason for a user to purchase this new proprietary option over the older, more mature one. Also, if coons' formulation of the feedback loop supporting proprietary software is correct, a new proprietary entrant into the field of image manipulation faces vertical economic barriers to adoption: it isn't an industry standard, so the industry isn't hiring those who only know that software. Thus, coons' formulation not only explains why professionals are slow to adopt FLoSS alternatives, it also serves to demonstrate why there is no competition in the space of professional proprietary image manipulation programs—competition, insofar as it exists, is solely within the consumer space where relatively few features are needed to meet user requirements.

[10] ginger coons does not capitalize her names for moral reasons related to the judgment implications of capitalizing some nouns over others.

## 9.1 Traditional Design Interfaces in FLoSS

In a personal conversation between myself and professional web designer Andy Fitz, he explained how he came to use FLoSS. In college, the Windows operating system on his computer broke down. His roommate restored the computer, but replaced the operating system with a variant of GNU/Linux. After realizing that this meant no more working with Photoshop, Andy gave GIMP a try. Frustrated by the difficulty of transitioning, he complained to his roommate that GIMP simply was not powerful enough. It didn't have all the features of Photoshop. Nothing was in the same place.

"How long did it take you to learn Photoshop?" his roommate replied.

"Five years," replied Andy.

"Learn GIMP for five years. Then you can complain."

Five years later and counting, Andy Fitz is no longer complaining.

### 1 Software Time

Five years is both a long and short time for FLoSS. It is long because a piece of software that hasn't been updated for five years not only seems horribly outdated, the chances are high that it will not even compile against the versions of the software it depends on that exist today. It is also a short time because there are many projects for which five years is just a small slice of the time it has been in development. For instance, GNUStep (a free software implementation of the same NeXTSTEP technologies behind Cocoa in Mac OS X) has been in development since the mid–1990s, which means it predates the development of Cocoa by five or more years itself. The typesetting system TeX began development 32 years ago—while the main program is no longer extended, new variations that fix long-standing issues with the original implementation continue to evolve. Five years can be a long time in terms of new features and code growth, yet it is not necessarily a long time in the lifespan of a FLoSS project.

Scott Rosenberg provides an important investigation into software time in his book *Dreaming in Code* (2007). While previous books have detailed the dynamics of software management in important ways—Frederick Brooks' *The Mythical Man-Month* being the prototypical example—they are all engaged in documenting proprietary development processes (Brooks 1975). *Dreaming in Code* instead focuses on the development of Chandler, a personal-information manager carrying with it ambitions of silo-less information management (calender events can be "stamped" as emails, a process that grafts the grammar of email (To:, Subject:, etc) onto the event). While it is an open-source project, it also has the advantage of a

Five years ago the Inkscape project began. Inkscape is a vector graphics editor similar in goals to that of Adobe Illustrator. Five years ago it did not exist. Today it is known for enabling workflows that incorporate both generativity and WYSIWYG design approaches. After five years of development, it has effectively implemented the features necessary for graphic design on the web.

# 10 Screenic Text

# 11 Text as Interface/Text as Process

The command-line interface (CLI), once a culturally universal site of intersection between human and digital process, has found itself virtually superceded by the visually metaphoric instrumentation of the graphical user interface (GUI). The mechanism of this transition from CLI to GUI within mainstream computing was the introduction of Microsoft Windows into the ecosystem of IBM-compatible PCs. The result was the injection of an additional semiotic layer, charged with a new modality of visual signification, between the user and the hardware (Stephenson 1999). For almost two decades consumer versions of Windows, however, were "DOS front-ends" that could not function without real, historical dependencies fulfilled by the presence of DOS deep within the guts of the operating system. Windows 1.0, for instance, used DOS's file operation functions (*Windows 1.0* 2010). This dependency on DOS recedes over time, eventually disappearing entirely in Windows XP, in which the DOS interface and functionality still exists but has migrated out of the substrate and into a virtual machine (*Windows XP* 2010).

The roots of the command line lie in a very physical process: the teletype. A teletype is resembles a typewriter in that it presented the users with a standard typewriter keyboard as a control. Pressing a key would result in an inked stamp of that keys respective character smacking onto the paper and retract, leaving its mark. Simultaneously the triggering of the key might be punched into a tape as a binary sequence representing the character. If so, the control was thus separated intrinsically between human and digital—it was not, as in todays keyboards, simply electrical signals converted into numbers transparently beneath our fingertips but rather also a physical instantiation of the sequence on a paper strip. The screen of this human-digital intersection was instantiated on the same paper as the recording of the input, using the same ink and stamps but now powered by the response of the machine to its human input.

Stephenson identifies an extremely formal dynamic of interacting through teletypic screens he encountered when learning to program in high school:

Anyway, it will have been obvious that my interaction with the computer was of an extremely formal nature, being sharply divided up into different phases, viz.: (1) sitting at home with paper and pencil, miles and miles from any computer, I would think very, very hard about what I wanted the computer to do, and translate my intentions into a computer language—a series of alphanumeric symbols on a page. (2) I would carry this across a sort of informational cordon sanitaire (three miles of snowdrifts) to school and type those letters into a machine—not a computer—which would convert the symbols into binary numbers and record them visibly on a tape. (3) Then, through the rubber-cup modem, I would cause those numbers to be sent to the university mainframe, which would (4) do arithmetic on them and send different numbers back to the teletype. (5) The teletype would convert these numbers back into letters and hammer them out on a page and (6) I, watching, would construe the letters as meaningful symbols. (Stephenson 1999)

In this text, titled ¨In the beginning was the command line,¨ Neal Stephenson proceeds to identify the underlying mechanisms of human-digital processual intersections: ¨computers do arithmetic on bits of information. Humans construe the bits as meaningful symbols.¨ He notes, however, that this act of translation is increasingly obscured by ever-increasing metaphoric abstraction, starting with the GUI and carrying on over the course of the evolution of graphical interfaces. Command-line interfaces are close to the bottom of the ¨stack¨ of the cross-translation between symbols and bits, whereas ¨[w]hen we use most modern operating systems, though, our interaction with the machine is heavily mediated. Everything we do is interpreted and translated time and again as it works its way down through all of the metaphors and abstractions¨ (Stephenson 1999).

Text, as the least abstracted of the available sites of symbol translation between digital binary forms (which can be considered the text of a different alphabet) and human process, is the formal level of computing. As such, the general non-consideration of the command-line interface in new media discourse is a disservice to the metamedium with which much of that discourse concerns itself.

One of the angles by which the CLI approached—tangentially—are discussions of code poetry. Florian Cramer provides a step by step engagement with computation both within digital contexts and written language that leads to an important insight:

The cultural history of computation shows that it is as rich and contradictory as that of any other symbolic form. It encompasses opposites, algorithms as a tool versus algorithms as a material of aesthetic play and speculation, computation as inner workings of nature (as in Pythagorean thought) or God (as in Kabbalah and magic) versus computation as culture and a medium of cultural reflection (starting with Oulip and hacker cultures in the 1960s), computation as a means of abolishing semantics (Bense) versus computation as a means to structure and generate semantics (as in Lullism and Artificial Intelligence),

computation as a means of generating totality (Quirinus Kuhlmann) versus computation as a means of taking things apart (Tzara, cut-ups), software as ontological freedom (GNU) versus software as ontological enslavement (Netochka Nezvanova), ecstatic computation (Kuhlmann, Kabbala, Burroughs) versus rationalist computation (from Liebniz to Turing) versus pataphysical computation as the parody of both rationalist and irrationalist computation (Oulipo and generative psychogeography), algorithm as expansion (Lullism, generative art) versus algorithm as constraint (Oulipo, net.art), code as chaotic imagination (Jodi, codeworks) versus code as structured description of chaos (Tzara, John Cage).

This contradictory nature envelops the command line as well. A tool at once more powerful and more flexible yet equally more opaque and unyielding. To begin to understand the command-line is to begin shooting lit arrows in the dark, lighting fires of process that will burn the results of their functioning onto your harddrive, your graphics card, your BIOS, or your network as onto your screen. The Unix command

```
rm -rf /*
```

will erase the entire contents of that Unix´s filesystem from the hard drive. The code for `rm` loaded into memory survives to delete itself from a core component of its materiality, that is, the raw 1s and 0s on the magnetic platters that constitute the persistent body of the command. It will not, however, survive the reboot inevitably awaiting such a fubar´d system.

## 11.1 Remediation and the Command Line

While the modern command line may be a remediated teletype machine, it is crucial to note that the commands available at the prompt *re-mediate nothing*. The processes embodied in file operation commands, for instance, instantiate into material effects on hard drives. They are abstractions of processual hybridization that results in the same command in the same operating system having the same effect, in this instance, on the file system. The modules loaded into the assemblage offering this abstraction depend on the format of the file system (NTFS, HFS+, ext*, etc.), the motherboard-to-disk controller protocol (IDE, SCSI, SATA, etc.) and the driver specificities of that disk controller. All of these elements are unique, digital assemblages. The *embodied processes* that are typed commands cannot be accurately held to the standard of a theory that is based on a conception of mediums as containing and extending previous mediums.

These commands, these arrows flaming into the dark, have only the output of text in order to satisfy the needs of *immediacy*. There are interactive commands, to be sure. Text editors and email clients are two commonly abstracted interfaces. However, do to the natural reliance of the keyboard as the control of the command line, these interfaces are as

likely to rely on combinations (or "chords") of key presses for purposes of navigation and process instantiation. This is a new type of immediacy only available within the electronic writing space, an immediacy that comes with a steep learning curve but which—once mastered—rewards the user with productive potentials beyond what is accomplishable through the semiotic abstractions of the GUI. Some hackers joke that their favorite operating system is their text editor `emacs`, which is noted for accomplishing everything from coding to typesetting (through the powerful AUCTeX extension) to emails and calenders to web browsing. All interaction is accomplished through these chords of key presses. The learning curve requirement means that "so much needs to be filled in" by the user: commands are clearly 'cool,' in the language of Marshall McLuhan (1964).

Neatly obscured and packaged as a 'command,' processes such as the aforementioned `rm` are *nothing* if not process. Commands are either latent or instantiated process.

# 12 Language is Programming

As N. Katherine Hayles puts it, "screenic text and programming are logically, conceptually, and instrumentally entwined" (Hayles 2004: 80).

In their book *The Alphabetization of the Popular Mind*, Ivan Illich and Barry Sanders write of the dimensionality that phonetic alphabets introduce to words. For them, "language" does not proceed our capacity to store representations of words as sounds:

> The historian misreads prehistory when he assumes that "language" can be spoken in that word-less world. In the oral beyond, there is no "content" distinct from the winged word that always rushes by before it has been fully grasped, no "subject matter" that can be conceived of, entrusted to teachers, and acquired by pupils (hence no "education,"learning," and "school"). For it is the record in phonetic writing that first carries what is heard across a chasm separating two heterogeneous eras of speech. The alphabetic scribe carries what is spoken from the ever-passing moment and sets down what he has heard in th permanent space of language. Only with this act can knowledge, separate from speech, be born. (Illich and Sanders 1988: 7–8)

Though their phrasing is inflammatory in its assured rejection of the possibilities of "knowledge" and "learning" amongst oral societies, they do however illustrate the new dimensions of potential arising from the intersection of words and the phonetic alphabet. The encoding of the *sounds themselves* into writing transforms "the page into a mirror of speech," freezing "the flow of speech itself onto the page" (11, 13). For Illich and Sanders, the resulting new dimensions delivered to speech through this synthesis of speech and alphabet into language include nothing less than "knowledge," "education," and even

"logic" itself. Though Illich and Sanders do not use the language of Simondon's ontogenesis, we can map how, as these dimensions begin to fill and expand over time, the metastability of the system continuously fluctuates as new potentials are described and, through that description, affect the landscape upon which further potentials unfold. For instance, in 1492 language becomes recognized as an avenue of control as the Spanish royalty begin the project of implementing a standard language to consolidate their subjects into a more rule-able assemblage.

Illich and Sanders note the lineage of Orwell's Newspeak, a product of the utopian writerly tradition of positioning language as the means of subjugation (in dystopias) (110). Through Newspeak the power of the State supplants the exercising of power by elites—the power of the State is exercised *on them*, rather than by them—as "the State has turned into a book that is constantly rewritten" (111). Illich and Sanders, however, use the term Newspeak to refer to "an approach or an attitude that treats language as a system and a code" (112). Fitting directly into this phrase, then, is cybernetics theory.

While digital processes necessarily change the understandings and uses of any word used to describe them, it is important to note that for the most part the words of computer science are appropriations from language that existed before computers. For instance, the word 'program' was first applied to computers in 1945 to refer to the "act of expressing an operation in the terms appropriate for the performance of a computer" (Illich and Sanders: 113). Thus, the word was appropriated from physical schedules of performances by human beings in an event and mapped onto the logospace of computational performance.

If the ¨so-called 'language' of physics is a code, a system of signs, a formal theory, an analytic tool that derives part of its value from its near-independence from ordinary speech" (Illich and Sanders: 116), then the `language´of the command-line fits this description as well, with one further caveat: *text on the command-line is kinetic*.

Florian Cramer provides an important perspective when he notes ¨the technical principle of controlling matter through the manipulation of symbols, is the technical principle of computer software as well¨ (Cramer 2005: 16). This manipulation of symbols underpins the entirety of the digital assemblage, and in that way the digital reflects humanity´s relationship with language. In an earlier essay called ¨Digital Code and Literary Text,¨ Cramer identifies a privileged relationship between language and binary code (Cramer 2001).

The connection to magic is instantiated culturally in the language of the hacker class. Firstly, in the formulation of the individual command-line entry as an `invocation.´ This recognition is important, for text has never been so kinetic as it is on the command line. From the literally typed input of the teletype machines to the virtual terminals with transparent backgrounds and multi-aphabet encoding running in a GUI, the textual input of the

command-line represents a site of language that promotes words from their status as *signifiers of meaning* to become *signifiers of action*. Not merely the description (evocation) of action, but the literal *invocation* of action through words.

# 13 Language adds dimensions

Language can be conceived as a 'program' for decoding the strings of symbols we call an alphabet into the meaning those symbols were arranged in order to convey. Language is thus adding the dimension of meaning to the digital code of the alphabet. Contrary to the popular misconstruing of 'digital' with 'binary', the term 'digital' just means a system with discrete units capable of formally representing some *thing*. For instance, alphabets are utilized for calculations in some fields of informatics. Likewise, characters of the alphabet are used to stand in for entire mathematical algorithms in the case of physics. Due to the discrete jumps between characters, they are utilizable for computation as well as representation (one could even argue that there is a computation occuring in the decoding of the representation from signifier to signified).

A good example to demonstrate this aspect of the alphabet is what is known as hexadecimal notation. Rather than our familiar base–10 (decimal) system of representing numbers, hexadecimal is a base–16 system. Counting from 1–9 is the same as in base–10, but starting at 10 we run into the difference. Being base–16 means that any given column in a numerical sequence must "count" 16 times (including 0). So 10–15 are represented in hexadecimal by the alphabetic characters A-F. Hexadecimal is a common notation in computation due to its extreme translatability back and forth between binary while at the same time maintaining a more compact system of representing numbers than either binary or decimal notation. Whether one would consider such utilization of the alphabet for representing numbers constitutes a remediation is a question that points to the fragility of the remediation concept: by focusing on 'media', the term becomes useless to discuss appropriation between anything that is not considered a medium. That is, it invites increasingly expansive definitions of 'medium' or risks blocking itself from application in a diverse range of instances. Processual hybridity, however, is a means for denoting this intersection of digits and alphabet, thus providing a handle by which to grasp hexadecimal notation in a critical context without resorting to materialist theories.

In the words of Florian Cramer, the "alphabet of both machine and human language is interchangeable, so that 'text'—if defined as a countable mass of alphabetical signifiers—remains a valid descriptor for both machine code sequences and human writing" (2001). The difference lies in syntax and semantics—that is, "computer algorithms are, like logical statements, a formal language and thus only a restrained subset of language

as a whole" (Cramer 2001). In recognizing this fact, note that syntax and semantics also influence the dimensional modulation of language in its application to systems. Whereas "language" as a whole can be conceived as a program for decoding 'alphabetical signifiers' into meaning, the subsets of language known as programming languages undergo a process of *transcoding*. Formulated as action from its first instance, words in computer code are *signifiers of digital process* rather than meaning. (Or, in the case of "codeworks" and code poetry, the meaning of the words is a meaning constructed of references to both the signifieds of both human language and digital process).

The relationship of code to language is that of a subset constrained by the specificities of syntax (Cramer 2001). The digital computer is ruled by syntax, which could be considered the defining means of mediation between digital computers and human processes. These processes include the actions of the users, the objects created/stored/distributed/displayed on digital computers, *and the processes by which these digital operations are instantiated*.

¨Computation and its imaginary are rich with contradictions, and loaded with metaphysical and ontological speculation. Underneath those contradictions and speculations lies an obsession with code that executes, the phantasm that words become flesh. It remains a phantasm because again and again, the execution fails to match the boundless speculative expectations invested in it.¨ (Cramer 2005: 125).

# 14 Top Down, Bottom Up

## 14.1 WYSIWYG

WYSIWYG, meaning "What You See Is What You Get," is a mode of interface design in which operations are performed in an extremely top-down manner. In terms of typesetting, the definitive example of WYSIWYG is Microsoft Word. Word can be deemed a remediation of the typewriter. However, it may be useful instead to consider it as an *appropriation of the grammar* of the typewriter. That is to say, rather than speak of it in terms of 'remediation,' which makes less and less sense the more digitally-unique features are added to the interface of Word. What does our knowledge that Word remediates the typewriter add to a discussion of WYSIWYG?

WYSWIWYG is clearly an instance of an attempted *immediacy*—by remediating the then-familiar modality of the typewriter into the context of the computer screen. But the entire concept of media seems complicated by this remediation: if the typewriter is

a medium, something which few media theories would likely argue against, then is Microsoft Word then a medium as well? This begins a slippery slope: would not all applications become, or have some claim to be considered as, media?

To avoid this I would argue that what is called remediation is, rather than media "consuming and extending" previous media, a dynamic in which the *grammars* of one processual hybridity (the keys-stamps-ink-paper assemblage of the typewriter) are appropriated and *re-conditioned* by another process (MS Word).

The WYSIWYG interface is clearly a top down approach, as all manipulations come from invoking processes onto what has already been placed into the interface.

## 14.2 Processed Text

Processed text comes in two flavors: *semantic* and *formal*. Semantic formats such as HTML and XML are far more widely used than formal formats such as TeX (and LaTeX/ConTeXt). While both are bottom up in contrast to WYSIWYG, there is a distinction even here between top down and bottom up. (They are both bottom up in that the typesetting goals of any block of text are specified at the beginning of that block, i.e. `<em>` and ``‘’` are both typed before the block of text begins.

HTML is top down, however, because that is its rendering model. By imbuing blocks of text with semantic qualities, one abstracts away the process of displaying those semantic blocks. Order is imposed from above, both through Cascading Style Sheets (CSS) and through the rendering algorithm of a given browser's implementation.

# 15 Constraints: Generative Design in FLoSS

# 16 Enter the Conditional

The design world is increasingly integrating generative approaches to their workflows. The most explicit manifestation, in writing, of this impulse is the "Conditional Design Manifesto" (Maurer, Paulus, Puckey, Wouters 2008). Written by Amsterdam-based designers Luna Maurer, Edo Paulus, Jonathan Puckey, and Roel Wouters, this manifesto outlines the approach that they have named conditional design and presents its three central elements: *process*, *logic*, and *input*. The statements of their manifesto on these topics are reproduced in full below:

**Process** The process is the product.

The most important aspects of a process are time, relationship and change.

The process produces formations rather than forms.

We search for unexpected but correlative, emergent patterns.

Even though a process has the appearance of objectivity, we realize the fact that it stems from subjective intentions.

**Logic** Logic is our tool.

Logic is our method for accentuating the ungraspable.

A clear and logical setting emphasizes that which does not seem to fit within it.

```
We use logic to design the conditions through which the process can take
place.

Design conditions using intelligible rules.

Avoid arbitrary randomness.

Difference should have a reason.

Use rules as constraints.

Constraints sharpen the perspective on the process and stimulate play within
the limitations.
```

**Input** The input is our material.

Input engages logic and activates and influences the process.

Input should come from our external and complex environment: nature, society and its human interactions.

"The process is the product," the manifesto declares. Employing the "methods of philosophers, engineers, inventors and mystics," the four authors of the manifesto seek to abandon the idea of a product in favor of "things that adapt to their environment, emphasize change and show difference."

These principles have been applied both within and outside of the domain of software. For instance there is the application of cellular automata rules (rules which algorithmically

govern the placement of points on an iterative graph) to seating in an auditorium. By invoking rules based on gender, the application of generative design into the physical world of seating arrangements provides angles of critical engagement with gender politicals and social organization.

Dutch designer Pietr van Blokland, faced with the task of producing brochures in 32 different languages as the house designer of Rabobank, has "thrown away" Adobe and WYSIWYG design in favor of a generative workflow. In fact, the firms constituting the cutting edge of design in the Netherlands work within the context of generative processes (Cramer, Mansoux, and Murtaugh 2010). Florian Cramer identifies this transition to the generative as an opportunity for free software to become assert a dominant position in the field.

# 17 Generative Design is driven by text

As Lev Manovich notes, the program Processing is gaining an increasing degree of marketshare among designers—in fact, Manovich singles out Processing as "coming close to [Alan] Kay's vision" of an easy-to-use programming environment that allows users to "develop complex media programs and also to quickly test out ideas" (Manovich 2008: 79–80).

Processing requires hand-coding of generative algorithms, but it provides extensive support in both a simplified syntax (in relation to Java) and a large set of library functions that enable easy integration of common processes.

# 18 Cases

## 18.1 The Piet Zwart Institute

Working with students from a diverse range of backgrounds, the Networked Media design program at Piet Zwart is guided by a "simple formula"—"it's not about designing *with* media, but it's design *of* media" (Cramer, Mansoux, and Murtaugh 2010). In their joint presentation of three of the faculty of this program at the Libre Graphics Meeting 2010 in Brussels, Florian Cramer explains that "free software provides the building blocks—the Lego bricks—for this kind of self-created media practice."[11]

[11] All quotes in this section, unless otherwise denoted, are from the presentation.

Not all media designed at the Piet Zwart Institute is powered by software: for example, the Open Streetlamp is nothing more than an ornamented wooden box .

Cramer mentions the program's stance against the division of programmer and designer, citing a "classical failure of new media projects"—the lack of shared language between designer and coder leads to lackluster results.

Proprietary program interfaces are a product of the 1980s. The transition from "manipulation to more symbolic thinking" is a positive aspect. Mansoux notes that graphical interfaces for design can become an "annoyance" that "locks artists into constrained workflows."

"We don't simply want free versions of existing tools. . . What we are more interested in is to see FLoSS as an entry point into a different media practice based on the comprehensive critical rethinking of communication in its relation to technology"

They push their students to use git.

## 18.2 Open Source Publishers

Femke Snelting and Pierre Huyghebaert are two members of Open Source Publishers, a collection of designers who collaborate on projects using only free or open source software. In a personal interview, they shared their reasons for going to open source. It was an apparent mixture of dissatisfaction with the interfaces of the proprietary offerings, which they found to be holding them back, and the ideology of free software itself (Snelting and Huyghebaert 2010).

In FLoSS, OSP finds "liberating constraints," a sensibility that finds form in a text prepared for *Libre Graphics #0!*, a magazine released at the Libre Graphics Meeting 2010: "With FLOSS, the resistance of the tool is now for us such a daily meal, that it has become a work field, an investigation space, and a playground" (OSP 2010).

Whereas the constraints of Adobe products were enough to inspire these designers to consider dropping them as tools, the constraints of open source seem to inspire OSP. Adobe products are shaped by the demands of industry, whereas FLoSS products are shaped by the demands—and self-initiative—of the community.

Snelting again and again invokes constraints as the most engaging aspect of working with FLoSS. The tools, often unfinished or limited in some way, necessarily inform the shape of the output that she is attempting to produce. This echoes one of the applications of logic found within the Conditional Design Manifesto: "Constraints sharpen the perspective on the process and stimulate play within the limitations."

# 19 Crystalized Process: Text That Typesets Itself

The time has come to for the self-reflective approach of this thesis to come into play. For a book called *Writing Space*, this work by Jay David Bolter provides scant discussion of actual writing environments on the computer. Originally written before the expansion of the World Wide Web into the sphere of popular culture, *Writing Space* is concerns itself with "the space of electronic writing."[12] In defining this space as "both the computer screen, where text is displayed, and the electronic memory, in which it is stored," Bolter belies the relative absence of process in materialist forms of media analysis (2001: 13).

Bolter's over-simple definition of electronic writing space does not incorporate the act of writing, only the display of it. This surface-level analysis fits well the application of his remediation theory—the surface of a medium (it's "screen") is the host site of remediation. Bolter delves below the surface in his explanation of a shift to topographical writing. In the electronic writing space, "any relationships that can be defined as the interplay of pointers and elements" are representable (32). The writing space "itself has become a hierarchy of topical elements" (32). This is the effect of the computer's affinity for symbol-processing: "Any symbol in the space can refer to another symbol using its numerical address" (30). To highlight the dimensional shift in the writing space, Bolter describes the operation of outline processors. These programs abstract a text to the level of sections. These sections can be moved around and manipulated.

# 20 Environment of Operation

This text is not typed in the manner that you see it. The above header is instead written like this:

```
# Environment of Operation #
```

Through the wrapper program `pandoc`, this input (written in Markdown) is converted into HTML and ConTeXt outputs.

**HTML** { Environment of Operation }

**ConTeXt** { \section{Environment of Operation} }

[12] The second edition of the book, published in 2001, was used for this thesis. While it is updated to include the Web, its roots in a significantly older text are worth noting.

The syntax of HTML represents a semantic operation: “Dear Mr. Browser, treat this as a header of level 1.” The syntax of ConTeXt, however, represents a macro command within a programming language. What it says is “call the sections of code that translate the text within the brackets to the parameters specified for the `\section{}` command.”

The literal ‘writing space’ of this thesis is a program called Textroom. Textroom is a minimalist text editor in which there are no buttons, taskbars, or other clutter. Only you, your words, and (optionally) informational text reporting the time, word count, percentage to accomplishing your writing goal, etc. By writing in plain-text, I open myself to the opportunities afforded me by version control systems. Developed to enable collaboration of programmers on a code base, version control systems can track changes in text across time (useful for this project) and allow for massively distributed workflows involved tens of thousands of individuals (useful for the Linux kernel).