Generative Design in the Context of Process

The design world is increasingly integrating generative approaches to their workflows. 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).

The Programmer and the Designer

A Split, With Origins

Alan Kay's 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*. 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)[1].

To this end, Kay and other members of the PARC team developed this language Smalltalk to facilitate a "qualitative shift in belief structures—a new Kuhnian paradigm in the same spirit as the invention of the printing press" (Kay 1993). The technical details of Smalltalk are indeed impressive to those who can read and understand them: Smalltalk is *object-oriented*, *dynamically typed*, and *reflective* (*Smalltalk* 2010). All three of these approaches have found increasing adoption in computer programming languges. For instance, Yukhiro Matzumoto's popular language Ruby offers all three of these characteristics and the founder readily admits that his language is greatly influenced by Smalltalk (!CITE!). For the sake of reference, the idea of dynamic typing remains controversial among programming language designers from the implementation of Smalltalk in the 1970s until this very day.

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.']

[1] http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

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[2]: 400).

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 shipped without the programming tools required to even make a program on one's own. 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—might be should either of those

[2] The page numbers refer to the article's position within *The New Media Reader*, published in (!CITE!). The original date of publication has been maintained in order to respect the publication's true chronological position in computer history.

seminally-positioned products have shipped with the kind of development environment to be found in Kay's largely-forgotten prototype.

The Contemporary Situation

This line of inquiry into lost historical possibilities serves only to highlight the 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 the indeed the process with which Apple builds its invisible, proprietary codebase into the commodity known as Mac OS X.[3]

[3] 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 intersections between capitalism and FLoSS.