Programming and the Fold

Aden Evens
Dartmouth College
HB 6032, Department of English
Hanover, NH 03755 USA
603 646 9115
aden@who.net

ABSTRACT
Programming offers arguably the greatest opportunity for creative investment in the computer. But, given the mechanistic relationship between source code and executable and the highly constrained formalisms of programming, it is hard to see where creativity would find a place within the rigor and determinism of code. This paper places this question of creativity in the context of a broader problem of creativity in the digital generally, then identifies an ontological structure, called a *fold* or *edge*, that marks the creative moment of digital interaction. In programming, the edge appears in the object, recognizable in object-oriented programming but common to every creative innovation in coding technique.

Categories and Subject Descriptors
F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs – *Object-oriented constructs*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *Object-oriented programming, structured programming*; K.2 [**Computing Milieux**]: History of Computing – *software*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic – *Lambda calculus and related systems*.

General Terms
Languages, Theory.

Keywords
fold, edge, creativity, object, binary, digital, ontology, Deleuze, abstraction

1. NOTE
First, a note about the title, "Programming and the Fold." I have never been entirely comfortable with my appropriation of the term *fold* in the context of the digital. My choice of this term was initially motivated by a certain topological image of the digital: the digital, I propose, is a flat surface and my research aims to discover how we fold this flat surface to bring it into contact with its outside, the actual, material world. While this motive remains valid for me, the term *fold* itself is overdetermined, favored by Deleuzians and other philosophers. Though I believe that my usage points roughly in the same direction as Gilles Deleuze's, I don't really mean to recall his *fold*, in part because I don't understand it well. Rather than educating myself, I have chosen to adopt a different term, one possibly freer of philosophical baggage. While retaining your associations to the original title, "Programming and the Fold," I invite you to think of this talk with an alternate title, "Programming at the Edge." The edge, also topological, has the advantage for my purposes of pushing more firmly against the binarity that the fold suggests: | versus ∧. Edge and fold are both asymmetrical, but the fold also admits a symmetry and so invites the binary. Programming at the edge…

2. PREFACE
My argument is motivated by what must be a Deleuzian premise, a claim about fundamental ontology: that the actual, the material, the human, the real all are creative. Reality, the world *happens*, it generates itself, its things, its meanings, its values all the time, inventing, always again offering up the new. To me this suggests something problematic about the relationship between digital and actual. For it is hard to see where we might discover a similarly creative principle at work in the arithmetic of 0s and 1s. Once digitized, information operates deterministically, and even allowing for something like chance does not introduce much contingency into the mix. More 0s and 1s, a different order of 0s and 1s. It isn't the most colorful palette for a painting.

I am relying implicitly on another premise, really a definition. The definitive characteristic of the digital, what makes a digital technology *digital*, is the prominence of these 0s and 1s, the use of a discrete code. Whatever is digital is digital because at some point it passes through this binary code, captured by 0s and 1s, evaluated using 0s and 1s, output in 0s and 1s. I mean, digital technologies encode not only their objects using 0s and 1s but also the behaviors of these objects, the actions they suffer, the possibilities of their interactions. Digital technologies are many things besides 0s and 1s; they are material, cultural, historical, and otherwise human. But what distinguishes them from other technologies is the essential role of the binary code. Restating my Deleuzian intuition: there is something difficult, perplexing about the relationship between this code and the world. How do 0s and 1s, at the heart of the digital, make sense of the world and in the world? What part of the human, the material meets this code, what can it encode? If the digital is, as is evidently the case, the site of a great deal of creative investment and creative production, how does the digital encounter this creativity? Or one might ask even more basically, What does the digital do?

These questions will no doubt seem somewhat forced or overly abstract. The binary may well be a key component of digital technologies, but after all it's only one part of the complex and varied devices that we call *digital*. However, the binary is more than a necessary element of the digital, not just an essential

Digital Arts and Culture, December 12–15, 2009, Irvine, California, USA.

resource for the operation of digital technologies. Rather, the binary is the essence of the digital, driving digital technology, determining to a large extent digital materiality, appearance, behavior, history, culture, the digital's distinctive way of being a part of our world. Not just an arcane abstraction known only to those engineers who design microchips or compilers, the binary logic reaches well outside of the microchip, encoding much of the material in and around the computer. Computing materials are chosen for their most efficient, least resistant capacity to store and/or logically manipulate tokens for 0 and 1. 0 and 1 direct the form of the computer, keyboard, mouse, monitor, hard drive. We shape our computers, their peripherals, their interface components to afford ready data entry, click or no click, keypress or no keypress, pixel red or pixel black or pixel green. The mouse and monitor show their complementary design around 0 and 1, mouse allowing linear motion in a bounded Cartesian plane, monitor mirroring that plane on its surface and assigning a Cartesian coordinate system that identifies each pixel by a series of 0s and 1s.

Evidence of the reach of the binary could be no balder than the mouse button, which offers the most direct material analog of 0 and 1. The mouse button is thus the simplest digital interface, up or down, click or not. Each key on the keyboard is similarly a binary device, either engaged or disengaged. And these materials-made-digital of the human-computer interface point toward the many ways in which users also are reshaped according to the binary logic. The interface itself positions our bodies, hands in front, eyes forward, minimal motion except for the crucial deployment of our own digits. The interface allows each finger to express 0 and 1 with the least effort.

The mouse button lets you say *this*, *here*, and *now*. On one side it is the element of the digital, the bit extruded into the material. On the other, it reaches for the general form of deictic specification, the human potential to interrupt the flow of time and the underlying continuity of space to assert a unique point or place or moment, an edge.

This is to say nothing of the way that the binary code imposes itself on the user's habits of thought in relation to the computer. Each step at the computer is a choice from preestablished options, one menu item or another, this filter or that one, click here or here or here. To express oneself or to satisfy one's desires at a computer is to represent that desire or expression as already digitized, as already made of 0s and 1s. Each desire is discrete, each goal specifiable in advance, each key pressed for a particular purpose. This habit of digital thinking and action is both challenged and confirmed by programmers. Programmers are uniquely positioned to expand the possibilities of expression at the computer, to find new uses for those 0s and 1s and new expressions that take advantage of the computer's powers. But programmers know more acutely than anyone the rigid constraint of a binary logic, as they confront this binarity while working with code. From the most sophisticated actions to the most rudimentary, to program a computer is to represent one's desire as a discrete sequence of definitive steps.

To insist on 0 and 1 as the essence of the digital is not to deny its irreducible materiality. Undoubtedly, the digital is material. It enjoys histories and cultures. Artists invent digital materiality, theorists plot it, programmers leverage it. But materiality, history, and culture do not make the digital digital. Rather the digital encodes history and culture, rendering material difference as 0 and 1. The digital's technique is abstraction. Abs-tract, from Latin *to draw away*, draws off difference, captures specificities of space and time in the generality of a code.

Though I am hinting at a worry I harbor about the reduction of the world to a string of 0s and 1s, I shouldn't overlook the extraordinary power of this technique of abstraction. Abstracting from the specificity of content, from the weight of time and place that anchors the ordinary events and artifacts that we encounter in the world, the digital reduces material resistance to a vanishing point. The same abstraction that threatens its creative capacities also enables the digital's unprecedented capability. Only because it lacks contingency and admits no accident can the digital function with nearly total accuracy and consistency. Only due to the rarefied purity of its logical foundation does it reduce material resistance to a vanishing point, so that it can be stored in any medium, executed across various machines, transmitted over any carrier. Only the radical agnosticism of the binary code allows the universal capture, the treatment of any information whatsoever using the same hardware and software. Miniaturization, rapidity, standardization, random access, precision, simulation, selection and manipulation of incomprehensibly large data sets, these are the unparalleled powers of the digital, advantages won by a zealous application of the power of abstraction.

0 and 1, the lingua franca of the digital domain, are thus best thought not as the source of the digital's vast reach but as the ultimate symptom, the telos of abstraction. If the digital renders time, space, and much else besides using an abstract logic of 0s and 1s guiding other 0s and 1s, this is because the binary approaches degree zero of abstraction. 0 and 1, operative in the digital, are not the tokens that you might type or write, *0* and *1*, nor the tokens that you might utter, "zero" and "one." Nor are they the numbers those tokens typically represent (the first two numbers we count with, perhaps). They aren't *on* and *off*, or *plus* and *minus*, or *true* and *false*, or *yes* and *no*, or anything so specific as to retain the semantic richness of these conjoined pairs. 0 and 1 are definitely not *nothing* versus *something*.

Rather, the bit, 0 and 1, is the difference whose posit is difference itself, an apotheosis of abstraction. The bit is an indifferent marker of difference. The meaning of the bit 0 is only that it is not (but might be or might have been) 1. A bit means *that it is not its other*. The number 0 is filled with significances, historical, lexical, symbolic, arithmetic, nominal, etc. But a bit, 0 or 1, is only the fact of not being its other, its value is nothing more than the negation of something which is similarly without positive value. Its specific meaning reduced to a minimum, the claim to be its other's other, the bit is ready to accept any interpretation, any structure made of positive difference, which is to say, any structure at all.

3. FOR PROGRAMMING

To summarize my project… The binary code is the essential technology that makes the digital what it is and gives it its particular way of operating, producing digital aesthetics, digital culture, digital ontology. Everything digital is digital because it passes through this binary code; the rich input of human expression and the rich output of sense phenomena are joined or jointed at this wasp waist of the binary, the narrow passage that admits only a stream of 0s and 1s, the universal solvent of the digital.

But 0 and 1 can't account, I say, for the creativity that the digital so patently manifests. If every difference in the digital were between 0 and 1, the digital would never make any real difference, would never reach the actual, the human world. If the digital operates by abstraction, how does this operation grasp the concrete, and what does it return to the concrete to make a difference there?

Thus we must search for that mechanism, that *edge* where the digital meets the actual, where the binary code encounters the creative in order to matter in the world. My contention is that this edge, along which the digital adjoins the creative, has its own recognizable character. Creativity takes a particular form in the digital domain, and that is the primary object of my research.

You may note that it's something of a straw man argument. Only because I *define* the digital as effectively barren does its evident productivity become unlikely or paradoxical, motivating my research. This objection would carry more weight were the binary code operating only behind the scenes as an invisible engine of digital technologies. But the sterility of the binary threatens to reach out into the human world as well, for the binary code materially determines the sorts of expressions that the computer can accept. It thus makes all the difference that the abstraction of the binary code structures even the material of the machine, the body of the user, and the habits of the culture that surrounds digital technology. Which is to say, my worry about the binary is not solely a matter of metaphysics or engineering. The digital sucks the world into its code, demands conformity. Only through an adequate understanding of how to safeguard the creative potential of this technology, only by ensuring that the sterility of the code does not sterilize the imaginations of those who use it can we leverage the extraordinary power of the digital without succumbing finally to its universal and totalizing encoding. Programming therefore enters the picture at a crucial juncture: programmers are uniquely equipped to advance the leading edge of the digital, holding open its potential. Where do we find the edge of programming? Or maybe the question is, Along what edge does the programmer insert her desire, her creative investment into the computer?

First I should note that I employ a very broad conception of programming, likely far too broad. Programming, I propose, is the act of directing the computer according to one's will, the act of expressing desire in or through a digital device. The breadth of this definition has the advantage of incorporating all of those many actions that constitute programming, which is not a uniform behavior but a motley: not just entering lines of code but testing, debugging, compiling, sketching, learning, designing, commenting, decoding, and more are all part of the normal activity of the programmer. (When I worked as a programmer, I spent about half my time in meetings.) Moreover my expansive definition includes as programming an action as unsophisticated as setting preferences in your browser, for this too directs the computer according to one's will. In fact, the most mundane end user activities all count as programming: entering text into a word processor, sending someone an e-mail, first-person shooting. I am open to restricting this loose definition, but I have yet to hear a convincing criterion that would distinguish fundamentally between the guild of programmers, with their pizza boxes and their acronyms and their nerdy jokes, and ordinary folks who use computers to see pictures of their grandkids and check the stock reports.

Programming as the act of imposing one's will upon the digital. This image suggests a particular aim, a horizon that impels the progress of programming. The programmer wishes to be able to impose her will, express her will with the least effort, the greatest immediacy. The ultimate programming system (the "silver bullet") would be one in which the will of the programmer were immediately concretized into a program. We could feed the computer the spec and the computer would create the software whose perfection was determined by the precision of the spec. But note that this fantasy implies the elimination of programming per se, the equivalence, at the limit, of programming and any other use of the computer. One always wishes to impose one's desire on the machine, so if a programming language existed that made this easy or natural, every user would be a programmer and programmers would be users like anyone else.

Part of the intuition that drives this fantasy of programming's telos is that the labor of coding feels always peculiarly superfluous, prompting noted computer scientist Fred Brooks [1] to quash the fantasy, declaring that there is "no silver bullet," that programming will always be laborious. Nevertheless, the development of computer languages, archives of code libraries, handbooks for design patterns, syntactically intelligent editing environments, these are all designed to reduce the gap between the spec and the software, to ease the programmer's clerical work and allow her instead to focus on the creative dimension of her task. In other words, the ultimate desire of programming, forever out of reach, is to erase itself, to make programming akin to any other activity at the computer.

I wish to restate my general concern about the digital now with respect to the activity of programming. Programming marks a crucial transition point between actual and digital, an expression of human desire packaged into a set of symbolic forms. The programmer more than other users must consciously submit his thoughts, his methods to the digital, represent his aims in terms of a formal logic. To program is to effect a translation of desire into a language of logic, and in this sense even high level programming languages are only a step or two shy of the 0s and 1s at the core of the computer.

4. THE OBJECT

Given that programming presents perspicuously the fundamental ontological dilemma of the digital, it is instructive to inquire after its edge. The question is surely naïve, for programming is rife with edges. If the edge marks every meeting between the binary code and creativity, then we should expect to find edges throughout the activities of programming, from the proliferation of paradigms and languages, to the development of sophisticated editing and planning environments, to the complex interventions of compilers, linkers, assemblers and other programs that mediate between source code and object code. Nevertheless, I designate one particular structure as the archetype of the edge in programming, a structure so fundamental to programming that they are almost coextensive. I have in mind *the object*, as made explicit in object-oriented programming (OOP).

My point is not to say that code reaches its pinnacle in object-oriented programming. On the contrary, the object has inhabited the activity of programming since the origins of software, rising to a point of particular visibility in OOP. Analyzing the characteristics of the object that determine it as an edge of code, we can also see how the object represents the key innovation of

programming, like the monolith in *2001* that shows up alongside evolutionary advance.

Proposition: An edge has four characteristics, (1) an increase in the number of dimensions, (2) hierarchical distinction, (3) inside-outside distinction, and (4) an enfolding of disparate spaces, times, and logics. These four symptoms operate more and less clearly at every edge, but the object presents them with a forceful intensity. I'll deal with them in sequence.

4.1 Increase in Dimensions

The most basic operation of an object is to aggregate diverse data and subroutines under a single name. (I am somewhat sloppy with my terminology. I often say *object* when I mean to be talking about an *object class*.) Conceptually, an object is like a set, not another element but the possibility of a group of individuals, which introduces a new dimension. Data and subroutines remain, now organized by the object at a new scale. The object does not exist on the same plane as its parts; its invention is a matter of seeing its parts from outside of them, rising above the plane of data and algorithms to generate a supervenient organization. (Formally, some object-oriented languages or environments do place the object on the same level as its parts, making both objects and their parts just types that can be imposed on any data. In such a case, the very notion of typing evinces the additional dimension; a type is a pattern that informs data from without, a named or measured structure.)

This increase in dimensions may seem abstract when described this way, more of a way of looking at the program than a fact about it. But the augmented dimension has concrete, material ramifications. Object-oriented programming tools allow the programmer to switch dimensions easily, providing a typical coding environment for the linear generation and editing of data structures and subroutines but also providing the opportunity to work directly at the level of program structure. The programmer works in the higher and lower dimensions, outlining the structure of object classes in the higher dimension and filling in the details of algorithmic content in the lower. Declarations define this structure, and are both placeholders for future content but also already the substance of the program, employed by the compiler to establish variable names and scope. Even the compiler recognizes the distinction between dimensions, dedicating different passes or phases of the compile process to the different dimensions, an initial pass that builds an image of the higher-level structure of the object classes and another pass to draw in the details of actual sequential operation.

4.2 Hierarchy

As for hierarchy, to some extent it is implicit in the dimensional augmentation: the object stands over the data and subroutines it comprises. The programmer frequently works from the object as a basis, considering the appropriate behavior of the object and programming subroutines accordingly. That is, the object is not only a formal grouping of code text, it is also a guiding principle of the code and of the coding. Objects must be organized to make a program, but in turn they organize much of the programming, determining what gets coded and where and how. (Popular philosophies of object-oriented coding promote the idea that the choice and definition of object classes are the substantive acts of OOP, while the implementation of particular methods is, relatively speaking, a mere formality.)

But hierarchy appears even more explicitly in the overarching organization of objects along filial or genetic lines. Inheritance makes one object class the parent of another, bequeathing by a kind of administrative shortcut all of the data and behaviors of the parent to the child, and allowing the selective alteration of these data and behaviors plus the addition of more data and behaviors to accommodate the child's special role in the program. This organization is biunivocal, as most OOP systems provide tools to allow either parent or child to take precedence in a given instance. That is, it is possible for the parent to defer to the child (by declaring a given function as *virtual* for instance), and it is possible for the child to defer to the parent, by not overriding a procedure or even by explicitly invoking the parent procedure when appropriate. Hierarchy is thus asymmetrical but not one-sided, allowing for considerable flexibility.

4.3 Inside-Outside

The firmest criterion marking the inside of an object is its scope. Using scoping, the object hides its inside, controls access, presenting an interface, a surface of exchange between inside and outside. According to rules enforced at compile time, only an object may directly alter its own data. Other objects may request information or propose an alteration, but a response is screened by the object itself.

Scoping is not confined to object-oriented programming but circumscribes plenty of other insides, the inside of a procedure, of a data structure, a `DO WHILE` loop, etc. Scoping is often the foundation of encapsulation, which demarcates an inside and tends to institute hierarchy and dimensional increase as well. The foundational role of scoping hints at the universality of the object, whose apotheosis in OOP recapitulates a long history of objects in code. The object, the finest edge of programming, haunts code's every advance, driving the progress of programming.

Given its centrality to code, scoping also holds an essential position in modeling calculi for code, including the λ-calculus and π-calculus. As the λ-calculus models linear algorithmic processing, it uses primarily order of operations and other orthography (such as parentheses) to indicate the inside of a λ operation, and scoping is a secondary if still basic feature of the calculus. But the π-calculus models communicating processes, where each statement in the calculus is implicitly surrounded by a context of other statements, which makes the demarcation of an inside fundamental to the operation of the calculus. The π-calculus thus provides two different means to impose a scope on a variable, including one symbol (v) whose sole purpose is to limit a variable's scope. The manipulation of scope in the π-calculus gives it its significant advantage over other calculi for modeling mobile processes, such as the Calculus of Communicating Systems. By providing the opportunity to extrude the scope of a private variable into a broader context, the π-calculus makes it possible for the inside of a process to expand its territory, to bring its outside in.

4.4 Enfolding of Spaces, Times, and Logics

A commonplace in the study of programming is the notion that programming involves a deferral of expression. The programmer exercises her will in the computer, but her will operates from a temporal distance, influencing an unspecified future, willing something in relation to the eventual user. The programmer's will is deferred, captured by the executable, and reexpressed,

represented by the running program. Compile time and runtime are not only arbitrarily separated from each other but are largely independent. Notably, the programmer's relationship to the machine is radically different from the user's relationship. A painter stands in largely the same relationship to her painting as does the (eventual) viewer of the painting, and even a composer is also a virtual audience of his work in progress. Certainly a programmer needs to act as user in order to test and evaluate a program she is working on, and her judgments rely on an ability to stand in the user's shoes, anticipate her level of understanding and her needs. But the programmer when coding does not relate to the program as a running executable. She does not sculpt an application out of its visible parts; she does not write first and foremost for an audience but for the computer, offering it instructions to be carried out in a different context and for a user radically unlike herself. This is the strongest sense in which programming adjoins or abuts disparate times, spaces, and logics, the heterogeneity of programmer and user.

This fourth characteristic of the edge is not yet specific to objects, for my comments apply to programming in general. But the object amplifies the enfolding of space, time, and logic. In a sense, the object extends the programming environment into the user's environment, coinciding runtime and compile time. While the programmer creates object classes, the user's actions cause the instantiation of actual objects, as though she is working in the code in retrospect. Her actions as the program is running are directed at entities encapsulated in objects and therefore referable to structures recognizable as such by the programmer during the coding process. Elsewhere I discuss how the object, through the `self` or `this` keyword, allows a deictic gesture from the user to point back into the code, narrowing the gap between the programmer and her deferred expression in the program. More accurately, these self-referential keywords allow the programmer to leverage an ambiguity in the code, so that the object referred to by `self` or `this` is not determined until runtime, the programmer's will directed at the unknown.

5. AT THE EDGE

I have insisted on but not supported the claim that edges are everywhere, that the object can be glimpsed at each moment of code's progress throughout the history of computing. I would offer briefly one further example intended to suggest the pervasive role of the object in programming. The foundational moment of software, the implicit separation of digital instructions from the hardware in which they are inscribed, already outlines an object. Software itself constitutes an additional dimension, a dimension of abstraction that has a reality independent of the bug-ridden or otherwise unpredictable materiality of the hardware. The asymmetry of hardware and software generates numerous hierarchies, as the hardware maintains a kind of absolute or final say over any actual operation, while the software comes to direct the hardware that serves it. Software invites a dangerous but tempting image of an inside, which persists today in the common conception of the computer as a "giant brain" with software playing the role of the mind. Finally, the invention of software moves the scene of programming outside of the hardware of the machine and divorces programming time from running time.

This model of the edge of the digital, appearing as the object in programming, allows theorists of software and codework to distinguish mundane from extraordinary programming activities, and to locate the moment of creative investment in the digital more generally. Programming presents a vexing milieu, seemingly hemmed in by its mechanism, it is nevertheless experienced by practitioners as an art. The edge and the analysis of which it forms a part demonstrate the extent to which programming is indeed mechanistic but also the places where it exceeds mechanism, prompting a creative intervention from the programmer. Careful study of the edge should lead not only to a better understanding of digital culture and artifacts but should help generate richer programming habits and better models for learning to program.

6. REFERENCES

[1] Brooks, F. 1986. No silver bullet: essence and accidents of software engineering. In Information Processing, H.J. Kugler, Ed. Elsevier Science, Amsterdam, 1069–1076.