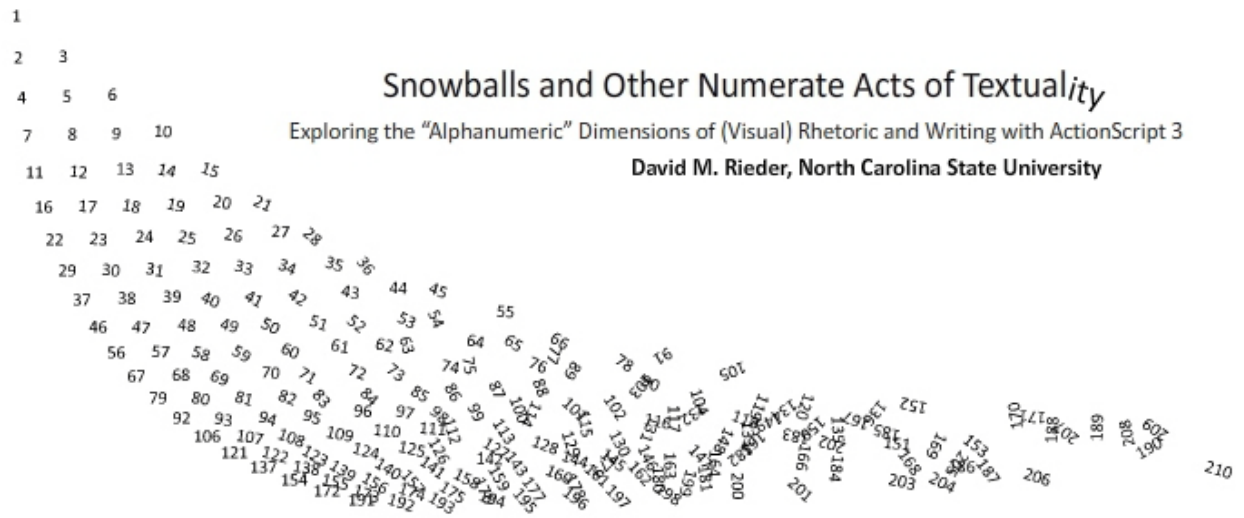

Abstract

The emerging field of "software studies" in the humanities provides an opportunity for scholars affiliated with computers and writing to delve more deeply into the computational dimensions of digital rhetoric and writing. To demonstrate how to develop an approach related to software, I introduce the work of a mid-twentieth-century research group known as the Oulipo, whose members used mathematics to explore the hidden potentials in rhetoric and writing. By associating the Oulipo's use of mathematics with the emergent field of "software studies," I demonstrate how writers can combine numerate and literate methods of writing in order to discover new, hybrid forms of textuality as well as new methods of writerly invention. The Oulipo's mathematical approaches to writing offer scholars in computers and writing a method of engaging with the study of code in the humanities.

This project is divided in three parts. In the first, I introduce the Oulipo and the call for the study of software introduced by Lev Manovich and Matthew Fuller. In doing so, I point out a key obstacle to the study of software in the humanities—the lack of engagement with numerate thought and expression. In a field such as *computers* and writing, which, considering the numerical basis of computers, could be called numbers or (mathematics) and writing, the lack of scholarly engagement with numerate thought is a blind spot. Especially due to the growing interest in "software studies," which includes an explicit engagement with computational methods, numerate and literate forms of writing must be combined. In the second part, I introduce an experimental form of text popularized by the Oulipo called a *boule de neige* or snowball. In addition to an overview of the textual form, I explain both how the Oulipian snowball contributes to visual rhetoric and how to study it as a numerate object. In the third part, I offer a detailed explanation of the computational dimensions of three snowballs. The goal of these studies is to show how a study of computational forms of writing leads to new approaches to both text and invention in computers and writing. All of the code for the three software studies is available as "freeware" (under the GNU General Public License) in the Downloads section.

Part I: The Oulipo, Software, and Mathematics

i. The Oulipo and Computers and Writing
ii. "Software Studies" on the Rise
iii. (In)numeracy and the "Two Cultures" Split: Applying for Dual Citizenship

Part II: Rhetorical and Numerical Properties of the Oulipian Snowball

i. The *Boule de Neige*
ii. Snowballs and the Canon of Style
iii. Snowballs *de Longueur* and Pythagoras' Triangular Numbers

Part III: Computational Studies of Three Snowballs *de Longueur*

i. Introducing AS3
ii. Snowball #1
    Output of the program
    Explanation of the code
iii. Snowball #2
    Output of the program
    Explanation of the code
iv. Snowball #3
    Output of the program
    Explanation of the code

Conclusion(s) {
Implications for Visual Rhetoric
and Writing Studies

Works Cited

Downloads
Article in PDF
.as & .fla files

Part I: The Oulipo, Software, and Mathematics

i. The Oulipo and Computers and Writing

In the fall of 1960 in France, a small, diverse group of fiction writers, poets, mathematicians, university professors, and "pataphysicians" formed a collective that would soon after be known as the *Ouvroir de Littérature Potentielle* (the Workshop for Potential Literature), the Oulipo. Their stated goals were to identify new forms of writing unrecognized by tradition and convention, and to develop new methods of invention. They viewed themselves as a working group involved in inductive or experimental research for the benefit of all writers: "[the Oulipo] merely seeks to formulate problems and eventually to offer solutions that allow any and everybody to construct, letter by letter, word by word, a text" (Motte 46).

The Oulipo's approach to writing was based in part on their opposition to the conventional notion that creativity emerges from unfettered imagination and inspiration. For the members of this group, the assumption that a discursive rule serves to *limit* creativity is naïve—in fact, it is the limit, the rule, which makes creativity possible. As Oulipian Marcel Bènabou argues, "Even the most rabid critics of formalism are forced to admit that there are formal demands which a work [of writing] cannot elude" (Motte 41).

Members of the Oulipo were not aware of the bureoning field of composition studies in the 1970s, but I assume they would have been wary of "moderate expressivist" arguments that reinforce the conventional notion of creativity, such as Peter Elbow's argument against the inhibiting forces of audience in "Closing My Eyes as I Speak," (51), Donald Murray's demand that students use their own language "in order to embark on a serious search for their own truth" (5) and Ken Macrorie's calls for students to rely on their "natural language" as an antidote to writing *Engfish* (9). In all three cases, constraints are viewed as a hindrance in that they threaten to limit the individual's freedom of expression.

The Oulipo endeavored to turn this equation on its head. For example, in the following passage, Queneau argues that unfettered "inspiration" is a kind of slavery, and that the observance of rules can be, in fact, liberating:

> Inspiration which consists in blind obedience to every impulse is in reality a sort of slavery. The classical playwright who writes his tragedy observing a certain number of familiar rules is freer than the poet who writes that which comes into his head and who is the slave of other rules of which he is ignorant. (41)

For the members of the Oulipo, writerly freedom is, paradoxically, based on an explicit understanding of and engagement with structure and constraint: "thus, it is not only the virtualities of language that are revealed by constraint but also the virtualities of him who accepts to submit himself to constraint" (Motte 43).

For the Oulipo, constraints are found at every level of the writer's process. In the group's first manifesto titled "Lipo," Le Lionnais argues that any process of writing must "accommodate itself as well as possible to a series of constraints and procedures that fit inside each other like Chinese boxes" (Motte 26). He goes on to list several kinds that include constraints of grammar,

diction, style, narrative, and genre. A writer is most free when s/he strategically uses the rules and constraints that define h/er methods and goals.

Based on their formalist assumptions about writing, the Oulipians argued that the process of writing could be innovated in two ways: 1) by pushing the conventional limits of ancient constraints in order to generate new forms of textuality, and 2) by developing new writing processes with the help of rules and procedures found in other disciplines. An example of the first method is the group's experiments with the Ancient Greek *lipagrammatos* or lipogram—a text in which words that contain one or more pre-selected letters are omitted from the writer's vocabulary. In his 300-page novel titled *La Disparation* (*A Void*), George Perec did not include a single word with the letter e. An example of the second method is the group's application of formulas from various branches of mathematics to generate texts. Claude Berge writes about the group's experiments with poetic forms including Boolean, Fibonaccian, and exponential poems like Raymond Queneau's often-cited sonnet, *Cent Mille Milliards de Poèmes* (*One Hundred Thousand Million Poems*). Skewing, stretching, and otherwise transforming the structure by which a text is produced leads to new forms of textuality and inventional methods.

Mathematics became the basis for many of the Oulipo's experimental studies with the written word. Le Lionnais claims, "Mathematics—particularly the abstract structures of contemporary mathematics—proposes thousands of possibilities for exploration" (Motte 27). In fact, founding members Jacques Roubaud, Claude Berge, and Paul Braffort were trained mathematicians, and several other members were self-described amateurs or enthusiasts. According to Warren F. Motte, the Oulipo aspired to contribute to a lesser tradition in the West in which the languages of word and number are intertwined to form hybrid approaches to both math and writing. In the introduction to his collection of the Oulipo's essays titled *Oulipo: A Primer of Potential Literature*, Motte traces this tradition from Pythagoras through to Ezra Pound. While there were competing methods of integrating the two fields, they all endeavored to think in a way that Le Lionnais esteemed as a form of "double nationality":

> Visited by a mathematical grace, a small minority of writers and artists (small, but weighty) have written intelligently and enthusiastically about the "queen of sciences." Infinitely rarer are those who—like Pascal and d'Alembert—possessed double nationality and distinguished themselves both as writers (or artists) and as mathematicians" (Motte 14-15).

Despite the novelty of this approach to writing, the Oulipo's cross-disciplinary experiments with textuality and invention have not been cited in either composition studies or computers and writing. Nevertheless, I argue that there are compelling reasons to study their work from within these disciplines—especially computers and writing. I focus specifically on two. The first is the way in which the Oulipo introduces an approach to writing with code, which is described in Part 1.ii of this essay. The second is the Oulipo's promotion of a "double nationality" comprised of numerate and literate approaches to wriitng and language, which is described in Part 1.iii.

ii. "Software Studies" on the Rise

First and foremost, the Oulipo's approach to constraint as an inventional method—and especially their use of mathematics—can be valued as a way for the field of computers and writing to delve more deeply into the *programmable* dimensions of writing in new media. If the programming concepts, rules, and procedures in programming languages are viewed as a generative medium for exploration, scholars in computers and writing could contribute to the emergent field of "software studies" by identifying and experimenting with the capacities of a programming language to generate new forms of textuality and methods of invention.

In his book titled *Software Takes Command*, Lev Manovich argues that the study of programming languages has been ignored for the most part in the humanities and social sciences. While scholars associated with these disciplines have established approaches to the studies of cyberculture, new media, Internet studies, and other related fields, the impact of software has been of marginal interest: "the underlying engine which drives most of these subjects—software—has received little or no direct attention" (Manovich 4). Adding to the force of his claim is Matthew Fuller's description of software as a "blind spot [in] the wider, broadly cultural theorization and study of computational and networked digital media" (3).

Both of these scholars advocate the establishment of a field called "software studies," and they offer compelling reasons for doing so. Manovich offers one based on his analysis of global economic trends. He argues that commercial software development, which includes applications for everything from blogging and mapping to data analysis and writing, is at the forefront of the global economy. Companies like Google, Oracle, Yahoo!, Microsoft, and Amazon are at the top of financial indexes and other types of market lists. These companies represent the vanguard of today's post-industrial economy: "If electricity and the combustion engine made industrial society possible, software similarly enables global information society" (4). Based on his economic analysis, Manovich argues that failing to recognize the role of software renders our scholarly work incomplete:

> if we want to understand contemporary techniques of control, communication, representation, simulation, analysis, decision-making, memory, vision, writing, and interaction, our analysis can't be complete until we consider this software layer. (7)

Likewise, Fuller argues that software is an inescapable part of the lives of citizens in the "global north." Software has become a "putatively mature part of societal formations," and its youngest generations are growing up under its many influences. For this reason, scholars in the humanities and social sciences "need to gather and make palpable a range of associations and interpretations of software to be understood and experimented with" (3). Fuller advocates that scholars extend their interests in digital media and culture by learning key concepts and methods in modern programming. Based on the essays in his edited collection, *Software Studies: A Lexicon*, which is touted as the first book with the moniker, "software studies," in its title, scholars associated with this emergent field will be able to work both theoretically and practically with key concepts and methods in modern languages. Scholars should know what are variables, data types, functions, loops, classes, and objects. They should know concepts such as encapsulation, elegance, and ethnocomputing, and the basic differences between procedural and object-oriented languages.

Inspired by Manovich's and Fuller's arguments, I can appreciate how they might read most studies of digital media as a viewer might experience the cinematic special effect known as a Hitchcock- or Dolly Zoom. Popularized by Alfred Hitchcock in his 1958 film, *Vertigo*, this effect is based on two opposing movements. In the following quote, the mechanics of the effect are described:

> The camera starts at a distance from the subject with the lens completely zoomed in. This is the start point. From here, when the shot begins, the camera tracks into the subject and at the same time the lens is zoomed out relatively *so that the subject in the foreground remains the same size*. This causes the backdrop to increase in depth of field as the lens is zoomed out causing a sense of gaining distance between the subject and the backdrop. (Valluri, par 5; *my emphasis*)

As the cinematographer tracks the camera in toward the scene, s/he zooms out the lens. The effect has been used in numerous films since *Vertigo* including *Jaws*, *Pulp Fiction*, and *Lord of the Rings*. My point in associating this effect with a research method that does not reconcile the *machinic* dimensions of new media (i.e., software) with the on-screen phenomena on which it is usually focused is the following: as our methods of engaging with digital media grow increasingly sophisticated, they remain paradoxically "screen deep." As we achieve greater levels of sophistication, drawing closer to the media that we study, we simultaneously marginalize the software underlying our studies. The closer we get to the screen, the farther away we push the machinic.

Manovich's and Fuller's concerns extend to the field of computers and writing, because few published books or articles affiliated with this field focus explicitly on key concepts in software programming, or specific methods in programming languages. Nor do the vast majority of books and articles include block quotes of executable code for discussion. While over the past decade, there have been a handful of articles and essays that focus on the importance of programming languages and executable code, these contributions represent a small, discontinuous set. In 1999, Ron Fortune and Jim Kalmbach guest edited an issue of *Computers and Composition* about programming and writing. In my estimation, their issue represents the most compelling, in-depth focus on topics related to programming in the field. In addition to their introduction titled "From Codex to Code," contribution in their issue included the following three articles that quote lines of code explicitly in the body of their texts: Cecilia Hartley's "Writing (online) Spaces: Composing Webware in Perl," Alan Rea and Doug White's "The Changing Nature of Code: Prose or Code in the Classroom," and Mark Haas and Clinton Gardner's "MOO in Your Face: Researching, Designing, and Programming a User-Friendly Interface." The importance of these articles is that they provide scholars and instructors in computers and writing with a solid basis for studying the rhetoricity of code as well as some of the connections between programming and writing. Unfortunately, the momentum that their issue represents was never picked up.

Since then, in 2003, N. Katherine Hayles published "Deeper into the Machine," an article in which she challenges computers and writing scholars to incorporate the "machinic" dimensions of writing (i.e., code) in to their work. Also in *Computers and Composition* in 2006, an article about Flash was published in which a section about ActionScript was included. In 2008, an

article in *Technical Communication* was published that includes a focus on code. Most recently, in 2009, two essays published in *Computers and Composition Online* include examples of executable code and some commentary related to them. The first is Anthony Ellerston's essay, "New Media Rhetoric in the Attention Economy," which includes a few snapshots of ActionScript programming in the Adobe Flash environment. The second is Brian Ballentine's "Hacker Ethics and Firefox Extensions." Under the header titled "Sample Hacks," Ballentine includes excerpts of the JavaScript code used to generate the hacks about which he writes, which he justifies including for the following reason:

> Although mastering Web Developer, Greasemonkey, or computer languages like JavaScript are not required to introduce students to white, black, and grey hat ethics, we should not shy away from engaging these new technologies.

Based on Manovich's and Fuller's calls for a turn to software, I argue that these scripts should be studied explicitly. They should not, as Ballentine implies, be supplied in a roundabout way to address the current limits of writing with and about programming languages and code in computers and writing. The opportunity to develop connections between the machinic and screenic dimensions of writing and rhetoric in computational new media is an opportunity worth raising to a necessity.

From a technical standpoint, the central focus of the field is the study of application environments in which new forms of writing are identified and practiced. Beginning with early studies of word processors, the field has developed an impressive body of technical knowledge related to specific applications. Nevertheless, in my investigations on the subject, few if any contributions in the field delve into the deeper programmable dimensions of those environments that provide access to a software language. Few essays and articles explore the "software layer" about which Manovich writes.

If Manovich's and Fuller's arguments are compelling, and the study of software provides a way in which the field can extend its areas of expertise, then the Oulipo's method offer a productive approach. The Oulipo's use of mathematical concepts to generate new forms of textuality and inventional methods can be transposed to the exploration of programming concepts and procedures in writing and rhetoric. In particular, the algorithmic dimensions of computational new media can be explored.

iii. (In)numeracy and the "Two Cultures" Split: Applying for Dual Citizenship

The second reason for studying the Oulipo's contributions to writing is related to Le Lionnais' esteem for scholars with "double nationality." Members of the Oulipo, who combined the languages of number and word in their research methodologies, demonstrated how writers can cross the "two cultures" split between the humanities and sciences. In 1959, C.P. Snow popularized the trope of the "two cultures" in his Rede Lecture at the University of Cambridge. Snow argued that intellectual work in the West had split in two. On one side of the dividing line are the humanists, and on the other are the scientists. Between the two exists "a gulf of mutual incomprehension—sometimes (particularly among the young) hostility and dislike, but most of all lack of understanding" (4). Snow's concern with the split is that it threatens intellectual progress at all levels of society: "This polarization is a sheer loss to us all. To us as people, to our society. It is at the same time practical and intellectual and creative loss" (11). Toward the end of his lecture, he calls for educators to help bridge the gap between the two.

Compared to other fields in the humanities, computers and writing has developed numerous bridges across the "two cultures" divide. Nevertheless, from the standpoint of "software studies," the lack of engagement with numerate modes of thinking and research is a topic that needs to be addressed. Considering the extent to which computers and computational media are based on numerate concepts and methodologies, an explicit engagement with the language of size, number, and quantity (i.e., mathematics) would be a productive basis for the study of software.

In his book titled *Innumeracy: Mathematical Illiteracy and Its Consequences*, John Allen Paulos defines innumeracy as "an inability to deal comfortably with the fundamental notions of number and chance" (3). His book opens with an epigraph that reads, "Math was always my worse subject" (3). In the introduction to his book, Paulos presents an anecdote that demonstrates the way in which innumeracy contributes to the "two cultures" split about which Snow spoke. Moreover, it is an anecdote that most English majors and their professors will relate.

> I remember once listening to someone at a party drone on about the difference between "continually" and "continuously." Later that evening we were watching the news, and the TV weathercaster announced that there was a 50 percent chance of rain for Saturday and a 50 percent chance of rain for Sunday, and concluded that there was therefore a 100 percent chance of rain that weekend. The remark went right by the self-styled grammarian, and even after I explained the mistake to him, he wasn't nearly as indignant as he would have been had the weathercaster left a dangling participle. (3-4)

This anecdote resonates with my past experiences. On several occasions, I have heard graduate students and professors in the humanities express their disdain for mathematics and their inability to think with numbers. An implied distrust supports their innumeracy. Based on my own dislike of mathematics after a dismal high school experience, I sympathize. I wish I'd had someone like Ian Stewart to write letters in response to my questions about math in high school. In the following excerpt from his book titled *Letters to a Young Mathematician*, Stewart offers a simple analogy for "school math":

> The school math you are learning is mainly some basic tricks of the trade, and how to use them in very simple contexts. If we were talking woodwork, it's like learning to use a hammer to drive a nail, or a saw to cut wood to size. (19)

A few pages later, he adds, "Schools—not just yours, Meg, but around the world—are so preoccupied with teaching sums that they do a poor job of preparing students to answer (or even ask) the far more interesting question of what mathematics *is*" (22). I did not begin entertaining answers to the definitional question that Stewart claims is missing until graduate school, which is when I took a series of courses in software programming. As I learned how to think *with* software in order to conceptualize solutions to "problems" I was programming, I developed an enthusiasm for math and numerate thinking.

One of the ways in which that enthusiasm manifested itself was during moments of brainstorming. Since a good portion of the programming with which I was preoccupied was visual, I was particularly interested in the ways in which a programming language could be used to represent perspective, movement, and the physical phenomena related to them. For example, after watching door chimes swaying in the wind, I wondered how the property, swaying, could be programmed. It's not as if there is wind in a programming environment. Rather, there are numbers that can be generated within the programming environment and passed to an object that uses those numbers in an equation that simulates swaying. I would pass my time thinking through issues such as these.

In his book titled *Mathematics for the Million*, Lancelot Hogben explains that linguistic thought and communication is concerned largely with qualitative issues. By contrast, the language of mathematics is concerned with quantitative issues. Hogben explains, "In contradistinction to common speech which deals largely with the qualities of things, mathematics deals only with matters of size, order, and shape" (75). As I think about the my enthusiastic brainstorming sessions as an amateur programmer, Hogben's distinction helps me understand the different ways of thinking and that developed during my training in software writing. For scholars and instructors in computers and writing, an ability to think in quantitative terms is an important part of thinking with software, which means that innumeracy is an obstacle

In my estimation as a software programmer, becoming numerate does not entail learning anything beyond the branches of mathematics offered in high school, but rather how to recognize the basic arithmetical operations and numerate properties with which an on-screen event is structured. In *The Language of New Media*, Manovich's first principle of new media can be interpreted as a call for a numerate turn in our studies of computers and writing. Manovich's first principle is "Numerical Representation," which he defines as follows: "All new media objects, whether created from scratch on computers or converted from analog media sources, are composed of digital code; they are numerical representations" (27). He lists two "consequences" of this principle of new media.

1) A new media object can be described mathematically. This means that the properties by which a programmer will define and subsequently access an object, requires one or more numerate acts.

2) A new media object is “subject to algorithmic manipulation” (27). The same object, whether it is a word or an image, can be transformed with the help of arithmetical operations. I can use basic algebra to evaluate the position of a letter in a string of words, or trigonometry to move those words in an arc across the screen.

The ability to expand literate modes of reasoning and expression numerically is essential for scholars and instructors in computers and writing who want to expand their interests in new media to software programming. In addition to these two consequences is a third: the opportunity to delve more deeply into the computational layers of writing and rhetoric in new media.

Part II: Rhetorical, Numerical, and Computational Properties of the Oulipian Snowball

i. The *Boule de Neige*

In order to demonstrate how an exploratory approach to "software studies" might be developed, the remainder of this essay is devoted to a constraint popularized by the Oulipo known as a *boule de neige* or snowball. There are several types of snowballs.

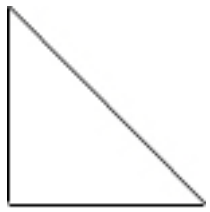

Figure 1. *Boule de neige de longueur*

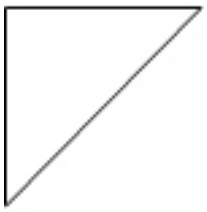

Figure 2. *Boule de neige fondante*

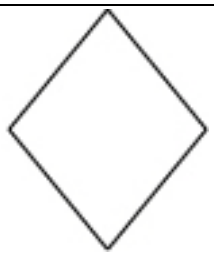

Figure 3. *Boule de neige losange*

Figure 4. *Boule de neige sabient*

Since they are essentially visual or "concrete" texts, Figures 1-4 above are visualizations of the geometric shapes that each of the snowballs represent. There are snowballs *de longueur* or growing snowballs (Fig. 1). There are others that are *fondante* or melting (Fig. 2). And there are snowballs that include both of these attributes. If they grow and then melt, they are *losange* (Fig. 3). If they melt and then grow, they are *sabient* (Fig. 4). Harry Mathews' "Liminal Poem" is an example of a snowball *losange* (see Fig. 5 below).

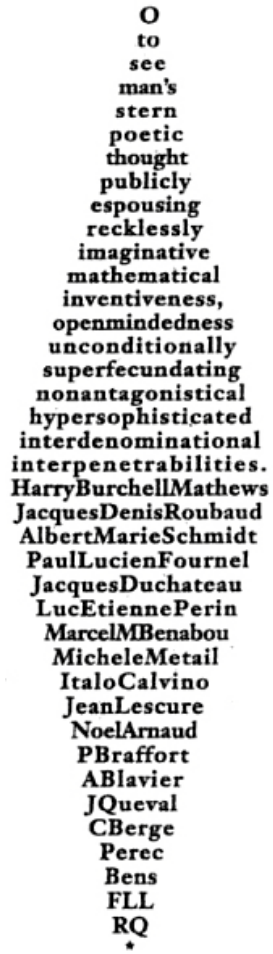

Figure 5. Harry Mathews' "Liminal Poem"

In the glossary of his book about the Oulipo, Warren F. Motte, Jr, provides the following definition of a snowball *de longueur*: "A form in which each segment of a text is one letter longer than the segment preceding it" (213). The following untitled poem by John Newman represents Motte's definition:

I
am
now
post
haste
(sort of)
posting
new topic
to discuss.

do you enjoy
constraints?
does word play

give headeaches?
are you confused?

This is a snowball,
A poetic form which
was created by those
who group themselves
with the name of Oulipo.
Every line contains one
Additional letter. U like?

Due to the successive growth of each segment, the text is characterized as snowballing.

Motte's definition limits the segments of a text to letters, but the segments of a snowball can be syllabic or word-based. For example, Dominique Fitzpatrick-O'Dinn's poem titled "But" is a word-based snowball *fondante*. Each successive segment of Fitzpatrick-O'Dinn's poem is one word fewer in length.

And I wanted to tell her my dream that she would be satisfied with me
that the importance she ascribed to having an actively nonmonogamous sexual life would fade
that in my mind she would find a community as diverse as any
in my hands would be revealed a language of new gestures
in my eyes were swimming visions sufficient to orient a superlative future
in my face were unfamiliar expressions waiting to emerge
in my voice were the murmurs of many
in my ears were a thousand songs
in my feet a world map
that I had been building
I had been learning
had been storing
was ready
But

For a creative writer or artist, a snowball can be used as an inventional technique related to the Oulipo's interests in constraint. By limiting or qualifying the ways in which s/he "naturally" writes, new associations among both words and concepts can be developed. Language becomes new again—or, as Russian formalist Viktor Shklovsky would phrase it, language is *defamiliarized*. In the following excerpt from his essay, "Art as Device," Shklovsky defines both the goal of art and the importance of its defamiliarizing capacity:

> The technique of art is to make objects "unfamiliar," to make forms difficult, to increase the difficulty and length of perception because the process of perception is an aesthetic end in itself and must be prolonged. *Art is a way of experiencing the artfulness of an object; the object is not important.*

A snowball, then, can lead to new experiences of both language and textuality.

Related to the way in which a constrained form of writing can lead to new perceptions of both language and text, a writer or artist can use a snowball to explore "concrete" dimensions of writerly expression. There are a number of competing definitions of concreteness in writing, but the one thing they all have in common is an interest in exceeding the conventional aspects of writerly meaning—grammar, syntactical structure, rhythm, and rhyme—with explorations of material considerations, which are usually visual, kinetic, or phonetic. In the introduction to her book, *Concrete Poetry: A World View*, Mary Ellen Solt explains this materialist focus as follows:

> Emotions and ideas are not the physical materials of poetry. If the artist were not a poet he might be moved by the same emotions and ideas to make a painting (if he were a painter), a piece of sculpture (if he were a sculptor), a musical composition (if he were a composer). Generally speaking the material of the concrete poem is language: words reduced to their elements of letters (to see) syllables (to hear). Some concrete poets stay with whole words. Others find fragments of letters or individual speech sounds more suited to their needs.

There are some parallels between the Russian Formalist idea of defamiliarization and the concrete poet's focus on the materialist. Specifically, the concrete poet is reacting against the conventional way of exploring and constructing linguistic meaning. S/he does not want to rely on the age-old modes of expressing emotional and conceptual meaning. Echoing this, Solt writes,

> the old grammatical-syntactical structures are no longer adequate to advanced processes of thought and communication in our time. In other words the concrete poet seeks to relieve the poem of its centuries-old burden of ideas, symbolic reference, allusion and repetitious emotional content; of its servitude to disciplines outside itself as an object in its own right for its own sake.

If this unconventional, materialist focus on visual and aural meaning coincides with *defamiliarization*, so be it. It most assuredly echoes the Oulipo's approach as when Le Lionnais writes, "Must we adhere to the old tricks of the trade and obstinately refuse to imagine new possibilities? . . . Should humanity lie back and be satisfied to watch new thoughts make ancient verses?" (27). As you should expect, Le Lionnais' answer is no.

For scholars in computers and writing, the *boule de neige* can be valued in two different ways. The first is by associating it with several different forms of visual style, which is the subject of the next section of this essay. The second way is to recognize its potential as a form of invention—specifically as a form of *heuretics*, which is Gregory Ulmer's neologistic twist on the tern, *heuristics*. In his book, *Heuretics: The Logic of Invention*, Ulmer introduces his approach to invention as an alternative to the hermeneutic approach in the humanities: "Theory is assimilated into the humanities in two principle ways--by critical interpretation or by artistic experimentation" (3). A *heuretic* is the latter approach. It can be characterized as an off-shoot of inductive or experimental science. The *boule de neige*, like all of the Oulipo's constraints, is a *heuretic* because it offers writers an open-ended method of inquiry. The value of this approach is to introduce new ways of viewing an otherwise well-known object (*defamiliarization*) and new forms of knowledge.

ii. Snowballs and the Canon of Style

Interspersed among the chapters in Richard A. Lanham's book, *Analyzing Prose*, is an argument about the visual dimensions of rhetorical style in writing that has a bearing on the ways in which we can value the Oulipian snowball as a form of visual rhetoric. His argument develops out of his explanation that writing is supposed to be transparent. We are supposed to look *through* it, not *at* it. We are not supposed to notice the words on the page, since the goal of prose is to communicate facts and conceptual reasoning in as efficient, clear, and sincere a manner as is possible. Lanham writes,

> [O]nly ideas matter, not the words that convey them. Words linger in the air only as a temporary contrivance for transferring ideas from mind to mind. To look *at* them, rather than *through* them to the ideas beneath, is to indulge ourselves in harmless antiquarian diddling or, still worse, treat ordinary language like poetry" (1, *his emphasis*).

But this attitude toward prose has always been countered by a tendency to "build back into literate culture the powers of oral expression which literacy by its very nature abjures" (xi).

The counter-tendency, which Lanham calls an *alphabetic counterculture*, is an attempt to recapture the power of oral expression related to voice, rhythm, and other “behaviors” to which speakers have access in oral communication. It is this rich, inviting dynamic that writers endeavor to recreate with the help of stylistic features, which, Lanham strongly suggests, are visual in nature.

Lanham's argument is divided between two levels. The first level is typographical and includes conspicuous examples of this "counterculture" that are representative of every major period in western culture, from Simias of Rhodes' *technopaegnia* in the fourth century BCE to Kenneth Burke's twentieth-century "flowerishes"; from the tradition of the illuminated manuscripts in the medieval period to Laurence Sterne's eighteenth-century novel, *The Life and Opinions of Tristram Shandy*. In the twentieth and early twenty-first centuries, it is found in a wide range of avant-garde writing from works by Dada to the electronic *technotexts* about which N. Katherine Hayles writes in her book, *Writing Machines*.

The second level is discursive and includes prose patterns like the *period*, and a wide range of classical schemes related to parallelism, balance, and antithesis. This is the level at which Lanham's argument is most relevant, because he argues that these stylistic patterns and schemes represent a “vertical visual” dimension of meaning.

Lanham’s claim is based on his method of charting the stylistic patterns in prose writing. Throughout his book, Lanham has developed numerous images, diagrams, and other forms of line art to help illustrate the implicit stylistic pattern in prose passages. As Lanham describes his process, “We have been, in each case, converting the linear text into an image” (97). Toward the end of his chapter titled “Styles Seen,” Lanham speculates that his images may represent more than a useful analytical technique for studying style: “doesn’t the technique work just because there is imagistic information already in the prose, information which is suppressed by customary prose presentation?” (97). Lanham is not willing to offer a definitive answer to his

question, but his preoccupation with the "vertical visual" dimension in prose implies that he's merely waiting for right form of proof. Meanwhile, in the following excerpt, Lanham presents his strongest call for associating stylistic patterns with a visual dimension of meaning in both speech and writing:

> The rhetorical tradition has always recognized patterns—chiasmus, for example—which have been called figures of shape, just as figures of sound like alliteration have always been acknowledged. Might we argue that, in wider and more frequent ways, prose styles call upon our powers of visual understanding" without seeming to?

Even if Lanham doesn't have the proof for which he's waiting, the answer to his question appears to be an emphatic yes.

Based on Lanham's argument, Oulipian snowballs can be studied as a form of visual rhetoric. In my estimation, snowballs *de longueur* visually represent schemes of amplification, such as climax. *Silva Rhetorica* defines a climax as "the arrangement of words, phrases, or clauses in an order of increasing importance, often in parallel structure." If this definition is extended to a visual form of rhetoric, the increasing length of each successive line can be interpreted as a visual depiction of "increasing importance."

Related to climax, Fitzpatrick-O'Dinn's snowball *fondante* is a visual representation of a *period*. In his book, *Analyzing Prose*, Richard A. Lanham explains that the periodic style is one in which the meaning of the discourse is postponed or suspended until the end. The *period* is traditionally associated with the sentence, but Lanham expands its application to much longer texts. Lanham explains, "it doesn't matter if the period stretches over one sentence or several. The main thing is the suspension, both of syntax and sense, until the end" (50). In Fitzpatrick-O'Dinn's poem, the funneling down of the length of the lines related to the narrator's characteristics contributes to a climactic end, and the single word ends the suspense that was building. The poem appears to conform to the characteristics of the period, albeit visually.

In addition to these two examples are several more, which I will present in the conclusion.

iii. Snowballs *de Longueur* and Pythagoras' Triangular Numbers

Snowballs can also be studied numerically or mathematically. The following explanations are limited to snowballs *de longueur*. Figure 6 is a numerical representation of 55-word snowball. Incidentally, it is output of the first software program that I will present shortly, a T_{10} snowball *de longueur*.

1									
2	3								
4	5	6							
7	8	9	10						
11	12	13	14	15					
16	17	18	19	20	21				
22	23	24	25	26	27	28			
29	30	31	32	33	34	35	36		
37	38	39	40	41	42	43	44	45	
46	47	48	49	50	51	52	53	54	55

Figure 6. A numerical representation of a 55-element, T_{10} snowball *de longueur*

Exploring the numerical properties of this snowball leads to the general topic of *triangular numbers*, which are numbers that can be represented geometrically in the shape of a triangle. Pythagoras is credited with identifying the triangular numbers, which bolstered his and his followers' beliefs that Nature is numerical and that the source of all meanings can be derived mathematically. In the following excerpt from her book, *The Mystery of Numbers*, Annemarie Schimmel offers the following context for their mathematical view of the cosmos:

> everything seemed expressible in numbers. Observation of the regular movements of the sky led to the concept of a beautifully ordered harmony of the spheres. The evolution of the world was paralleled by that of numbers: unity can into existence from the void and the limit; out of the One the number appears, and out of the number comes the whole heaven, the entire universe.

Out of the one were the Pythagorean triangles, which grew equilaterally from a simple set of calculations.

The triangular numbers were identified by Pythagoras as a way to represent numerically geometrical shapes. Figure 7 is a sample of the first six triangles from Pythagoras' study.

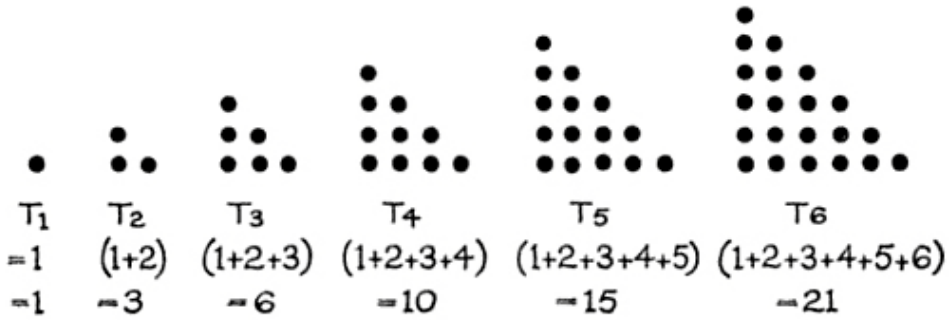

Figure 7. Illustration of the first six triangular numbers from Lancelot Hogben's *Mathematics for the Million*

The images are comprised of dots that are supposed to resemble the pebbles or stones that a follower of Pythagoras would have used to construct the figures. The method of arrangement was to add one more pebble to each subsequent row in order to construct the triangles. Figure 8 demonstrates how the rows are formed.

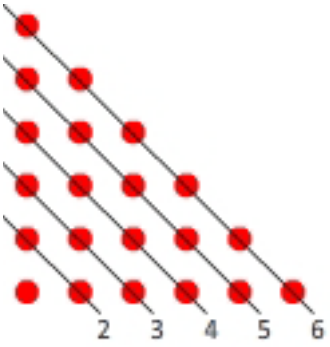

Figure 8. $T_2 – T_6$ with the rows outlined

For scholars in computers and writing, the numerical basis of this textual form is valuable for two reasons. First, it contributes to the cross-disciplinary approach to language and writing that Le Lionnais characterized as "dual citizenship" (see Part 1.iii). Second, it leads to new methods of writing in computational new media. Related to Manovich's first principle of new media (see Part 1.iii), the ability to recognize the numerical character of a text is an essential step in the direction of new, experimental forms of writing, rhetoric, and textuality.

One intriguing characteristic of triangular numbers—and snowballs *de longueur*—is that they can be derived by adding up the consecutive integers that constitute their elements, which is a characteristic that will serve us well as we use AS3 to program our first snowball, which I'll explain shortly when I discuss the importance of looping. This characteristic is represented in the parenthetical explanations in Figure 6. With Figure 6 in mind, Pythagoras' fourth triangle, T_4, which has a value of 10, is based on the addition of the numbers 1 – 4. Likewise, the value of T_6, which is 21, is derived by adding together the integers 1 – 6. The numerical representation of T_{10} is the sum of 1 – 55, which is 1035.

Let's dwell on this characteristic for a moment: *the total number of elements in a triangular figure can be derived from the value of one of its sides*. Let's imagine that a creative writer has asked you to help h/er determine the number of words s/he will need to develop a 16-row

snowball *de longueur*. What will be her total word count? Based on this characteristic of triangular numbers, you should be able to present h/er with an answer.

The answer is 136, and the solution can be reached in several different ways. Figure 9 depicts the three solutions presented in Brian Hayes essay titled “Gauss’s Day of Reckoning.”

$$\frac{n}{2}(n+1) \quad \frac{n(n+1)}{2} \quad n\frac{(n+1)}{2}$$

Figure 9. Three solutions to the sum of an arithmetic progression presented by Brian Hayes

Hayes’ essay is an investigation of the historical accuracy of an anecdote about a young math prodigy named Carl Freidrich Gauss. As an adult, Gauss distinguished himself as one of the greatest mathematicians; as a child, his solution to a question posed by his school teacher echoes into the present. In the following excerpt from his essay, Hayes relays the story as it’s usually recounted:

> In the 1780s a provincial German schoolmaster gave his class the tedious assignment of summing the first 100 integers. The teacher's aim was to keep the kids quiet for half an hour, but one young pupil almost immediately produced an answer: 1 + 2 + 3 + ... + 98 + 99 + 100 = 5,050. The smart aleck was Carl Friedrich Gauss, who would go on to join the short list of candidates for greatest mathematician ever. Gauss was not a calculating prodigy who added up all those numbers in his head. He had a deeper insight: If you "fold" the series of numbers in the middle and add them in pairs—1 + 100, 2 + 99, 3 + 98, and so on—all the pairs sum to 101. There are 50 such pairs, and so the grand total is simply 50×101. The more general formula, for a list of consecutive numbers from 1 through *n*, is $n(n + 1)/2$.

This description of Gauss’ solution troubles Hayes because his research into the historical accuracy of the account is unfounded. Ultimately, no one knows which method the young Gauss used to solve the problem. In response to the three solutions that he presented in his essay (Fig. 9), Hayes writes, “Mathematically, its’ obvious they are equivalent. For the same value of n, they produce the same answer. But the computational details are different and, more important, so are the reasoning processes that lead to these formulas” (204).

The Oulipo were not Pythagoreans, but considering the fact that the snowball *de longueur* is a triangular figure, members of the research group might have been interested in its cross-disciplinary uses for invention. They may have been interested in the way that texts "constrained" by this mathematical structure develop into two-dimensional objects that have symmetrical or equilateral sides. They may have also been interested in the historical relevance of the triangle as it relates to Gauss’ solution to the question posed by his schoolmaster.

In sum, the triangular figure or snowball is is an object that represents a provocative blend of numerate, literate, and rhetorical methods of writing. It is a mathematically quantifiable object that can be generated computationally, it is an experimental form of writing that can be valued

by poets and other creative writers, and it is a heuretic that can generate novel forms of textuality that represent visually schemes of amplification such as climax, hyperbole, and even the period.

Part III: Computational Studies of Three Snowballs *de Longueur*

i. Introducing AS3

There were two languages that would have suited the explorations developed in the following three studies. One is *Processing*, which is a relatively new, open-source language developed for a cross-disciplinary audience of computer programmers and visual artists. The language was introduced to help these two groups learn how to explore the other field in a relatively easy-to-learn programming medium. The other language is AS3.

I chose AS3 for a number of reasons. First, Adobe Flash is a popular application environment among scholars and instructors affiliated with computers and writing. Second, there is, in fact, an open source Flash development community. Although Adobe Flash is a commercial application, programmers working in AS3 distribute source code widely under open-source licenses. (If you are interested in open-source Flash, I recommend Chris Allen's book titled *The Essential Guide to Open Source Flash Development*). Third, AS3 is a relatively easy language to learn. Pedagogically, this makes the "learning-curve" much less steep. The code is English-based, which means that many of the commands and concepts are intuitive, and a student can write the traditional first program, the "Hello, World" program, in a single line of code. For a non-programmer, AS3 will make the transition to writing code a little less intimidating. Finally, AS3 is based on the internet language standard known as ECMAScript 4. In the 1990s, the ECMA (the European Computer Manufacturer Association) proposed an internet language standard based on JavaScript and JScript. The proposed standard has gone through several versions, the most recent of which is ECMAScript 4. The fact that AS3 conforms to this standard means that it may very well stand the test of time. In fact, due to its standardization, ActionScript programmer and writer Bill Sanders has called AS3 "the language of the Internet."

The following three studies delve fairly deeply into explanations of the code, but I don't go through every single line. Rather, I focus on the "algorithmic" lines of code. In *The Language of New Medi*a, Manovich explains that software is comprised of databases and the algorithms that manipulate them to generate results. For all intents and purposes, the numbers and text displayed on the screen are database values, and the lines of code related to their creation are not that interesting; they are mostly set-up. The provocative aspects of these three projects—especially from an Oulipian perspective—are the lines that manipulate the numbers and text to generate the on-screen texts.

All of the code cited in the three snowball projects is available for download. It is also distributed as "freeware" under the GNU General Public License, which is included in the zipped file that contains all of the .as and .fla files.

ii. Snowball #1

Output of the program:

1

2 3

4 5 6

7 8 9 10

11 12 13 14 15

16 17 18 19 20 21

22 23 24 25 26 27 28

29 30 31 32 33 34 35 36

37 38 39 40 41 42 43 44 45

46 47 48 49 50 51 52 53 54 55

Figure 10. A numerical representation of a T_{10} snowball *de longueur*

Software program:

Snowball_1.as

```
1  package {
2
3   import flash.display.Sprite;
4   import flash.text.TextField;
5   import flash.text.TextFormat;
6
7   public class Snowball_1 extends Sprite {
8
9    public function Snowball_1() {
10    var odo:int = 1;
11    var mag:int = 10;
12    var lf:Calibri = new Calibri();
13    var tf:TextFormat = new TextFormat();
14        tf.font = lf.fontName;
15        tf.align = "center";
16        tf.size = 10;
17        tf.color = 0x000000;
18
19    for(var r:int = 0; r <= mag; r++) {
20      for(var c:int = 0; c < r; c++) {
21        var t:TextField = new TextField();
22            t.defaultTextFormat = tf;
23            t.width = t.height = 19;
```

```
24              t.embedFonts = true;
25              t.text = odo.toString();
26
27              t.x = 1 + ((t.width + 5) * c);
28              t.y = 1 + ((t.height + 5) * r);
29
30              addChild(t);
31              odo++;
32            }
33          }
34        }
35      }
36  }
```

Explanation of the Code:

```
10      var odo:int = 1;
11      var mag:int = 10;
```

. . .

```
19      for(var r:int = 0; r <= mag; r++) {
20        for(var c:int = 0; c < r; c++) {
21          var t:TextField = new TextField();
```

. . .

```
25              t.text = odo.toString();
```

. . .

```
27              t.x = 1 + ((t.width + 5) * c);
28              t.y = 1 + ((t.height + 5) * r);
```

. . .

```
30            addChild(t);
31            odo++;
32           }
33          }
```

Our exploration of the T_{10} snowball *de longueur* begins with these twelve lines of code because they constitute the algorithmic center of the program. Beginning with lines 19 and 20, the

snowball that is "outputted" to the screen is based largely on these two *for* loops. These loops are the engine or our program.

The First *for* Loop

A *for* loop is one of several types of loops in AS3. The *for* loop is well-known and often-used loop most every modern programming language, from C to PHP. What the *for* loop does is allow you to execute one or more lines of instructions a specified number of times. For example, if you write the following lines of code in the Actions Panel in the first keyframe of a .FLA file, save it, and then test the movie, the numbers 0 – 10 will be published in the Output Panel.

The code	The output
1 for(var k:int = 0; k <= 10; k++) { 2 trace(k); 3 }	0 1 2 3 4 5 6 7 8 9 10

In order to read for loop, the following explanation breaks it down into its constituent parts.

1. for — All *for* loops begin with this keyword. You can think of it as a formal method of announcing a set of conditions as in a legal document.

2. (var k:int = 0; k <= 10; k++) — Within these parentheses are three statements separated by two semi-colons.

    i) The first statement, var k:int = 0, creates a *var*iable called *k*, "types" it as an *int*eger, and assigns it a value of 0. Variables are, in essence, containers. The "var" *k* is a container that has been assigned (i.e., contains) the value of 0. As the loop iterates, the value of *k* changes. In *Mathematics for the Million*, Hogben explains that a mathematical variable of which *k* is paradigmatic, is the equivalent of a pronoun. H explains that a pronoun can be "assigned" a wide range of antecedents. In AS3, variables must be "typed, "which means that you must define the kind of value that will be contained in the variable. This may seem unduly complicated, but it is an incredibly useful way to safeguard against errors in long, complicated software programs. If the usefulness of typing isn't apparent, imagine the way typing might help a speaker who says, "John and *myself* went to the store last night." If the speaker's words were expressed in a "typed"

environment, s/he would have been warned that s/he'd tried to assign the wrong antecedent to the pronoun/variable typed by that grammatical context.

The difference Between = and ==

In programming languages such as AS3, a single = sign does not signify equality. Rather, it is an "assignment operator," which means that the value to the right of the = is assigned to the variable to the left. In the for loop, `var k:int = 0` means that *k* has been assigned the value of 0. In order to signify equality, AS3 uses two equal signs, ==. `k == 0` signifies that *k* is equivalent to 0.

ii) The second statement, `k <= 10`, is the "condition" that must be met in order for the *for* loop to execute the instructions between the { curly braces }. He condition must be true in order for the loop to execute the code between its braces; so, as long as *k* is less-than-or-equal to 10, the `trace()` command will be executed.

iii) The third statement is a condensed method of writing the statement, `k = k + 1`. In a *for* loop, this increment operator is known as the *iterator*. After the `trace()` command is executed, 1 is added to the value of *k*. The iterator can be defined in any interval. I could have written i += 5, which would mean the `trace()` command would be executed twice, since the loop will not execute if *k* has a value over 10.

3. { . . . } The curly braces or brackets are used to contain a series of instructions that are supposed to be executed by the loop.

4. `trace()` In AS3, the *trace()* command is a built-in method. It is used to display the value of variables or expressions while testing and a software program. You can think of it like a thermometer for checking the temperature of a program at various moments during its execution. It is very useful when you are trying to track down a problem. For our purposes, it is a simple way to output the values of k to a screen without having to define text fields, x and y properties, etc.

```
11    var mag:int = 10;

. . .

19    for(var r:int = 0; r <= mag; r++) {
```

With an understanding of the *for* loop, lines 11 and 19 should make more sense. In line 11, I declared (i.e., created) a variable which I named *mag*. I typed it as an integer, which means that it

can only contain (negative and positive) counting numbers such as – 2, - 1, 0, 1, and 2. I assigned it a value of 10.

In line 19, *mag* is in the second set of instructions in the *for* loop, which is the condition that must be met in order for the loop to execute the instructions in between the curly braces; so, we now know that the value of mag defines the number of times our loop can iterate, which is 10. If you already made the association between mag and the number of rows in our T10 snowball, you've realized for what the first *for* loop is meant. The *for* loop in line 19 is generates the 10 rows in our snowball.

The Second *for* Loop

20	for(var c:int = 0; c < r; c++) {

As we transition to the second *for* loop, our program gets a little more complicated. You may have noticed that the second *for* loop in line 20 is inside the curly braces for the first *for* loop. What we have is a loop inside a loop. The first *for* loop executes the second *for* loop, which is an efficient way to generate a two-dimensional grid in which the rows and columns are symmetrical. The first loop is for the rows, and the second loop is for the columns. Due to the symmetry of the rows and columns in snowballs *de longueur*, the value of the *r* defines the number of times the second loop will iterate. Figure 11 illustrates the way the two loops are used to generate our two-dimensional figure.

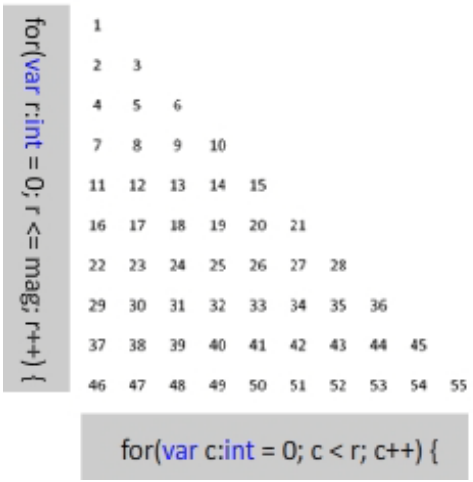

Figure 11. An illustration of the two for loops. The first loop generates the rows, and the second loop, which is executed each time the first one iterates, generates the columns

Like the first *for* loop, the second one begins with the declaration of a variable called *c*, which I typed as an integer and assigned an initial value of 0. Moreover, like the first *for* loop, the value of *c* will rise by one with each iteration (*c*++); so, *c* will begin at 0 and iterate by one each time it executes the code in its curly braces.

"Moment of Zen":

The second statement in the second *for* loop is the most challenging moment in this entire program, because the *r* in the conditional statement, `c < r`, is the variable from the first *for* loop. The meaning of `c < r` is the following: the second *for* loop will execute the instructions in its curly braces as many times as `c < r` is true. If the first *for* loop has iterated three times, which means that r has a value of 4, then the second *for* loop will execute three times, because 4 is not less than 3.

In sum, these two interrelated *for* loops generate our triangle. Since the number of the row and the number of columns in the row are equivalent, this method of iteration is both logical and efficient. If `r == 3`, then the second *for* loop will execute the code in its curly braces 3 times. It won't execute its code a fourth time, because the value of *c* would be 4 which is greater than 3, which means the condition in the second *for* loop is false.

Odo, TextField, addChild(), and x/y

There are three more lines of code that need to be explained. The first is a variable that I've called *odo*. The name, *odo*, is supposed to be short for odometer. In Line 10, *odo* is assigned an initial value of 1. In Line 31, which is inside the second *for* loop, a value of 1 is added to it each time the second for loop executes the code in its curly braces, which is a total of (10 / 2)(10 + 1) or 55 times.

```
10      var odo:int = 1;

. . .

31          odo++;
```

Simply put, *odo* is the variable that contains or holds the value of the number that will published on the screen. Since it represents something like a numerical distance, I thought that of it like the odometer in a car.

Lines 21, 25, and 30 are also inside the curly braces for the second *for* loop. These three lines accomplish three relatively simple actions. In Line 21, another variable is declared. I called it t, and it is typed as a TextField. As the name implies, a text field displays string values (i.e., text) on the screen. The property of a text field that is assigned the display value is .text, which means that the value to the right of the = sign in Line 25 is what will be displayed on the screen.

```
21          var t:TextField = new TextField();

. . .

25              t.text = odo.toString();

. . .

30              addChild(t);
```

Since the value of odo is typed as an integer, and t is typed as a text field, the programming environment would throw an error if we tried to put a number in a text field. For this reason, AS3 has a method that JavaScript'ers may recognize. It is a method that transforms numbers into text, the *toString()* method.

One the value of odo is transformed into a string (i.e., text) and assigned to the text field, t, it has to be published to the screen, which is what the addChild method in Line 30 accomplishes. In Line 30, each time the second *for* loop executes, the value of *t*, which is the value of *odo*, is published to the screen.

The final two lines of code, Lines 27-28, which are executed by the second *for* loop, determine where each of the 55 *t*s (i.e., each of the numbers in our snowball) will be published on the screen. The * is the sign for multiplication.

```
27  t.x = 1 + ((t.width + 5) * c);
28  t.y = 1 + ((t.height + 5) * r);
```

TextField have x and y properties. In order to determine where the text fields are published on the screen, we have to determine the values for the variables in these lines of code. The variables are the two properties, width and height, and the variables c and r, which are from the two for loops.

The values assigned to the width and height properties determine where on the screen the text field will be displayed. They were assigned in Line 23. They both have a value of 19 pixels.

```
23  t.width = t.height = 19;
```

This means that Lines 27 and 28 can be rewritten as follows:

```
27  t.x = 1 + ((19 + 5) * c);
28  t.y = 1 + ((19 + 5) * r);
```

And these equations can be further reduced as follows:

```
27  t.x = 1 + (20 * c);
28  t.y = 1 + (20 * r);
```

If we focus on Line 28, which is the line that determines the rows (the y axis is the vertical axis), we can determine exactly where each row will be published on the screen. The first for loop will iterate 10 times, which means r will have a value of 0 – 10; therefore, the rows will be published at the following vertical positions on the screen:

r = 0	t.y = 1 + (20 * 0);	t.y = 1
r = 1	t.y = 1 + (20 * 1);	t.y = 21
r = 2	t.y = 1 + (20 * 2);	t.y = 41
r = 3	t.y = 1 + (20 * 3);	t.y = 61

r = 4	t.y = 1 + (20 * 4);	t.y = 81
r = 5	t.y = 1 + (20 * 5);	t.y = 101
r = 6	t.y = 1 + (20 * 6);	t.y = 121
r = 7	t.y = 1 + (20 * 7);	t.y = 141
r = 8	t.y = 1 + (20 * 8);	t.y = 161
r = 9	t.y = 1 + (20 * 9);	t.y = 181
r = 10	t.y = 1 + (20 * 10);	t.y = 201

The position of t.x can be determined in a similar fashion, so I will let you, the reader, calculate their values.

iii. Snowball #2

Output of the program:

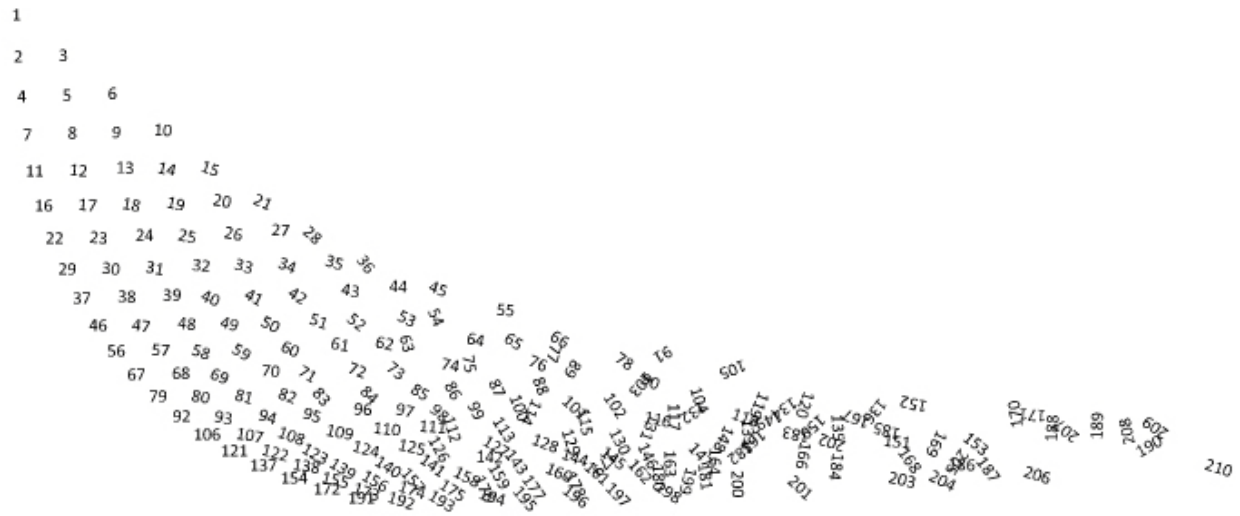

Figure 12. A numerical representation of a T_{20} snowball *de longueur* with dynamic changes to x, y, and rotation properties of the text fields

Software program:

Snowball_2.as

```
1   package {
2
3     import flash.display.Sprite;
4     import flash.text.TextField;
5     import flash.text.TextFormat;
6
7     public class Snowball_2 extends Sprite {
8
9       public function Snowball_2() {
10        var odo:int = 1;
11        var mag:int = 20;
12        var lf:Calibri = new Calibri();
13        var tf:TextFormat = new TextFormat();
14            tf.font = lf.fontName;
15            tf.align = "center";
16            tf.size = 10;
17            tf.color = 0x000000;
18
19        for(var r:int = 0; r <= mag; r++) {
20          for(var c:int = 0; c < r; c++) {
21            var t:TextField = new TextField();
```

```
22              t.defaultTextFormat = tf;
23              t.width = t.height = 19;
24              t.embedFonts = true;
25              t.text = odo.toString();
26
27              t.x = (1 + odo) + ((t.width + 5) * c);
28              t.y = (1 - odo) + ((t.height + 5) * r);
29              t.rotation = r * c * Math.random();
30
31          addChild(t);
32          odo++;
33           }
34         }
35       }
36     }
37  }
```

Explanation of the Code:

Although an on-screen comparison of Snowballs 1 and 2 seems dramatically different, the differences in the code are minimal. In the second snowball, changes have been made to three lines of code, and one additional line code has been added (Line 29). The first change is to Line 11. The value of mag has been raised to 20, which means that the snowball will be double in size; the magnitude of the snowball has been doubled.

```
11      var mag:int = 20;
```

Instead of 10 rows, there are 20, and instead of 55 numbers/text fields, now there are 210. The second and third changes are to Lines 27-28, which define the location of each text field on the screen.

Snowball #2

```
27  t.x = (1 + odo) + ((t.width + 5) * c);
28  t.y = (1 - odo) + ((t.height + 5) * r);
```

Snowball #1

```
27  t.x = 1 + ((t.width + 5) * c);
28  t.y = 1 + ((t.height + 5) * r);
```

In Snowball 2, *odo* was added to the x or horizontal value in Line 27, which means that each text field displayed on the screen will have "drifted" over one additional pixel to the right with each iteration of the second *for* loop. In Line 28, the vertical height of the snowball is reduced by a similar degree. In other words, as the rows are placed on the screen, the distance between them is reduced as the value of odo rises. If you study Figure 12, you'll notice that some of the last rows (i.e., rows 15 – 20) begin to rise above the rows preceding them. A curling effect is generated by this simple change to Line 28.

The last change is the addition of Line 29, which adds rotation to each of the text fields displayed on the screen. One again, the numerical values used to determine the value of the rotation of each text field is generated from variables already in use in the program. The *Math.random()* method adds an aleatoric dimension to the text fields on the screen.

```
29    t.rotation = r * c * Math.random();
```

iv. Snowball #3

Output of the program:

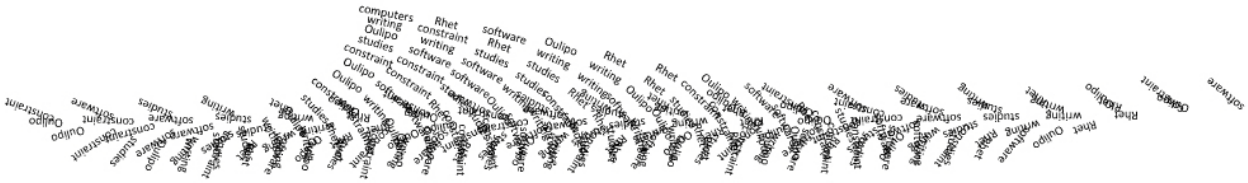

Figure 12. A T_{20} snowball *de longueur* with textual elements *and* changes to the x, y, and rotation properties

Software program:

Snowball_3.as

```
1   package {
2
3     import flash.display.Sprite;
4     import flash.text.TextField;
5     import flash.text.TextFormat;
6
7     public class Snowball_3 extends Sprite {
8
9       public function Snowball_3() {
10
11        var mydb:Array = new Array();
12              mydb[0] = "computers";
13              mydb[1] = "writing";
14              mydb[2] = "Rhet";
15              mydb[3] = "Oulipo";
16              mydb[4] = "constraint";
17              mydb[5] = "software";
18              mydb[6] = "studies";
19
20        var counter:int = 0;
21        var odo:int = 1;
22        var mag:int = 20;
23        var lf:Calibri = new Calibri();
24        var tf:TextFormat = new TextFormat();
25            tf.font = lf.fontName;
26            tf.align = "center";
27            tf.size = 10;
28            tf.color = 0x000000;
29
30        for(var r:int = 0; r <= mag; r++) {
31          for(var c:int = 0; c < r; c++) {
32            var t:TextField = new TextField();
```

```
33              t.defaultTextFormat = tf;
34              t.height = 19;
35              t.embedFonts = true;
36              t.text = mydb[counter];
37
38              t.x = (1 - odo) + ((t.width / 2) * c);
39              t.y = 1 + ((t.height / 3) * r) + 5;
40              t.rotation = odo + 10;
41
42              addChild(t);
43              odo++;
44
45              if(counter == mydb.length - 1) {
46                  counter = 0;
47              }
48              else {
49                  counter++;
50              }
51          }
52      }
53  }
54 }
55 }
```

Explanation of the Code:

Our third and final example of a snowball looks markedly different from the first one, but, again, the difference in the code is not that significant. The most significant difference in this program is the use of text, which I'll go over shortly. The differences in the on-screen appearance of the original snowball can be traced to three lines, 38-40, which are the three lines that define the location and rotation of the text fields on the screen.

Snowball #1

```
27      t.x = 1 + ((t.width + 5) * c);
28      t.y = 1 + ((t.height + 5) * r);
```

Snowball #2

```
27      t.x = (1 + odo) + ((t.width + 5) * c);
28      t.y = (1 - odo) + ((t.height + 5) * r);
```

Snowball #3

```
38      t.x = (1 - odo) + ((t.width / 2) * c);
```

```
39      t.y = 1 + ((t.height / 3) * r) + 5;
40      t.rotation = odo + 10
```

In Line 38 of the third snowball, it is the x or horizontal value that is reduced with each addition to the value of odo. The statement, t.width + 5, has been replaced with t.width / 2, which is considerably shorter, too. Figure 13 below, which is a numerical version of the snowball in Figure 12, illustrates the way in which the manipulation of the x value has changed the look of the snowball on the screen.

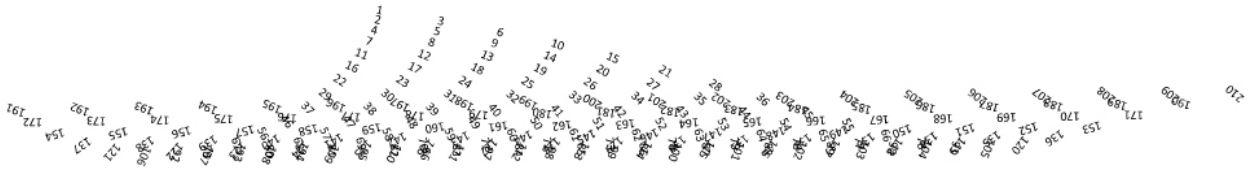

Figure 13. A numerical version of the T_{20} snowball *de longueur* in Figure 12

With a few keystrokes, the lines appear to be curving back as well as folding over each other. A third dimension is simulated by the rotation and curvature of the text fields.

Line 39 has also been changed, although not as dramatically. Essentially, the space between the rows has been reduced.

In addition to the dramatic on-screen changes introduced by Line 38, the change in the rotational values added to Line 40 has transformed the look of the on-screen text. Especially when we look at the role of rotation in Figure 12, the lines appear to be forming a twisting, horizontal column in the middle of the textual iteration.

Finally, the other significant difference in our third snowball is the introduction of text. We have finally moved beyond numerical representations. Lines 11-18 represent an Array.

```
11      var mydb:Array = new Array();
12          mydb[0] = "computers";
13          mydb[1] = "writing";
14          mydb[2] = "Rhet";
15          mydb[3] = "Oulipo";
16          mydb[4] = "constraint";
17          mydb[5] = "software";
18          mydb[6] = "studies";
```

In their reference guide to AS3 titled *ActionScript 3.0 Bible*, Roger Braunstein, et al introduce the array as a complex data type. They explain that it is "a lot like a numbered list of items" (153). In fact, they encourage their readers to visualize an array as a "set of numbered cubbyholes, each containing a single item" (154). Variables like *mag*, *odo*, *r*, and *c* are "primitive," because they can only hold one value at a time, an array contains numerous values. An array is complex because it can contain hundreds of values in an ordered list-like structure.

Lines 12-18 constitute a 7-item list of strings or words. They ordered numerical beginning with 0. In order to access one of the items in an array, I place the number at which the value is listed in square brackets to the right of the name of the array.

```
36          t.text = mydb[counter];
```

Line 36 does not have a number in the square brackets to the right of *mydb*, but a quick review of Line 20 shows us the reason: *counter* is a variable that I created and typed as an integer. It's initial value is 0, which means that the very first word published to the screen will be "computers."

```
20      var counter:int = 0;
```

The last series of lines that need to be introduced are 45-50

```
45          if(counter == mydb.length - 1) {
46              counter = 0;
47              }
48          else {
49              counter++;
50              }
```

These lines demonstrate the use of an if-else statement. I use it to reset counter to 0 when it reaches a value of 6. The reason is that there are only 7 elements in mydb; so, if the counter has a value beyond 6, it would try to access a slot in the array that doesn't exist. mydb[6] is the last element in that array. mydb.length has a value of 7, but the first slot is a 0, which means that the highest value counter can contain is 6, which is why I evaluate for the value of mydb.length – 1.

The basic structure of an if-else statement can be summarized in the abstract as follows:

if(true) {
execute these instructions;
}
else {
execute these instructions, instead;
}

Braunstein, et al explain that the if-else conditional statement is one of the most basic ways to introduce decision-making in a software program (19).

Based on this abstraction, let's assume that the value of *counter* is 6.

if(6 == (mydb.length – 1) {
change the value of counter to 0;
}

Since the value of mydb.length – 1 is 6, counter will be reassigned a value of 0. By contrast, if the value of *counter* is 2, which means that the parenthetical statement to the right of if is false, an additional value of 1 will be added to counter (i.e., counter++).

Conclusion(s) {

After the computational studies in Part III, there are three interrelated topics with which I'll conclude. The first is about new methods of invention based on the generative power of software loops, the second is about the elevated role of style in studies of computational texts such as the snowball, and the third is the importance of recognizing the way the software programs developed in Part III demonstrate one way in which alphabetic and numeric thought are interrelated in a programming project.

Invention and the Loop

In *The Language of New Media*, Manovich writes about the importance of the software loop. In fact, he characterizes it as the concept that "gave birth . . . to computer programming" (317). His explanation is that loops are what generate the basic flow of software programs. He writes, "A computer program progresses from start to finish by executing a series of loops" (317). In conjunction with conditional statements such as the if-else statement that was used in Lines 45-50 in Snowball_3.as, looping structures are essential. Based on the role that the two *for* loops played in all three software programs, Manovich's claims seem well-founded. The two-dimensional texts in all three programs were generated by running two *for* loops.

Directly related to the topic of invention, Manovich asks, "Can the loop be a new narrative form appropriate for the computer age?" (317). It's a provocative question, but I'm more interested in the role of the loop in visual rhetoric and writing. Based on the role of the *for* loops in these and numerous other programs that I've written, I argue that software loops can be the basis for new forms of invention. This is because the loop is not just an iterating structure. In the context of software programs such as the three presented here, the loop is a quantitative force. Like the wind that makes a pinwheel turn, it is a numerical force that can be used to transform the way in which objects on the screen appear as well as interact with each other. In the second and third snowball projects, values from the two for loops were instrumental in generating the novel textual forms displayed on the screen. The numerical values of *c* and *r* (the variables in the two *for* loops) were channeled to the x and y properties of the text fields, which helped generate the novel textual forms in both the second and third snowballs.

Software Studies and Style

There are two stylistic approaches to computational forms of text such as the snowball that have a bearing on the theory (and practice) of visual rhetoric. I introduced the first one at the end of the section titled "Snowballs and the Canon of Style." In that section, I argued that snowballs *de longueur* are visually related to rhetorical schemes like climax and the period. Additionally, the second snowball could characterized as a visual representation of the running style. Lanham describes the running style as one that was supposed to simulate the rambling, stream-of-consciousness of thought: "It tried to reflect the mind in the process of thinking by using connectives that did not subordinate but simply added on" (65). The structure of the second snowball begins in a manner that seems periodic, but the changes to the placement and rotation of the elements on the screen demonstrate a devolution toward an unstructured process. Incidentally, if you change the value of mag in Snowball_2.as to 30 or dare I say 75, the text will

have changed dramatically from the well-structured triangle with which it started. It will have turned into a provocative text that devolves, twists, and eventually explodes into a running style.

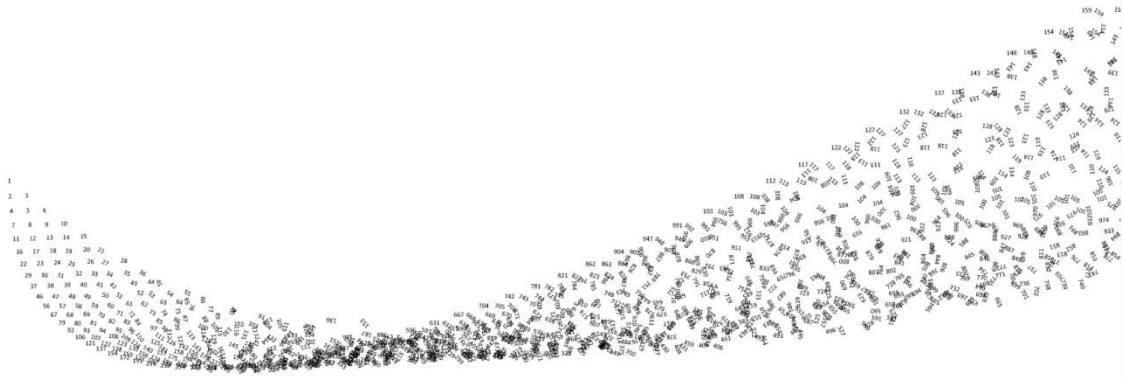

Figure 14. The output of Snowball_2.as when *mag* is assigned the value of 75

Finally, the third snowball's folded appearance could be characterized as visual forms of schemes like antithesis, chiasmus, and other schemes in which oppositions are exploited. Moreover, as additional texts are added to that program, the juxtaposition of form and content could lead to "thick" stylistic studies of visual rhetorical texts.

> In fact, if the following changes are made to Snowball_3.as, you can use any text that you like. All you have to do is replace lines 11 – 18 with the following:
>
> ```
> 11  var mytextstring:String = "ADD YOUR TEXT HERE";
> 12  var mydb:Array = textstring.split(" ");
> ```
>
> The text that you add between the quotes in Line 11 can be hundreds of words long, if you like. In Line twelve, there should be a single white space between the set of quotes inside the parentheses.

The second stylistic approach is to develop associations between software concepts and methods and rhetorical style. For example, looping structures could be equated with the role of repetition in the canon of style. If studies of digital rhetoric incorporate explicit associations between numerate and literate forms of writing, the canon of style may be a productive body of knowledge on which to draw. It may be possible for scholars in visual rhetoric to reconcile the programming used to generate a text with the visual forms displayed on the screen. That kind of cross-disciplinary reading of new media textuality would be a fascinating contribution to "software studies."

Literacy++;

Finally, the three software offer both scholars and practitioners of digital rhetoric and writing an opportunity to begin thinking "extra-curricularly." First, there is the use of variables, and the use of operators to combine or transfer values from one "container" to another. Perhaps the most interesting aspect of the three software programs is the way in which the variable r is used in both for loops as well as in the lines that define the y value of the text fields. Compared to a literate approach to writing, the way in which the variable r is linked to several areas within the

software code demonstrates a difference in the ways in which a numerate writer will approach a problem.

Works Cited

Ballentine, Brian. "Hacker Ethics and Firefox Extensions: Writing and Teaching the 'Grey' Areas of Web 2.0." *Computer and Composition Online*. Fall, 2009. Web. 30 May 2008.

Berge, Claude. "For a Potential Analysis of Combinatory Literature." *The New Media Reader*. Ed. Noah Wardrip-Fruin and Nick Montfort. Cambridge, MA: MIT Press, 2003. Print.

Braunstein, Roger, et al. *ActionScript 3.0 Bible*. Indianapolis: Wiley Publishers, 2008. Print.

Christian, Peter. "N+7 Machine." 10 November 2008. Web. 29 May 2009.

Elbow, Peter. "Closing My Eyes as I Speak: An Argument for Ignoring Audience." *College English* 49:1 (1987): 50-69. Print.

Ellerston, Anthony. "New Media Rhetorics in the Attention Economy." *Computer and Composition Online*. Spring, 2009. Web. 30 May 2008.

Fuller, Matthew. *Software Studies: A Lexicon*. Cambridge, MA: The MIT Press, 2008. Print.

Hayes, Brian. "Gauss's Day of Reckoning." *American Scientist* 94:3 (May-June 2006): 200-205. Web. 2 June 2009.

Hogben, Lancelot. *Mathematics for the Million: How to Master the Magic of Numbers*. New York: W.W. Norton & Co, 1993. Print.

Lanham, Richard A. *Analyzing Prose*. 2nd edition. New York: Continuum, 2003. Print.

Macrorie, Ken. *Telling Writing*. 4th edition. Portsmouth, NH: Boynton/Cook Publishers, 1085. Print.

Manovich, Lev. *The Language of New Media*. Cambridge, MA: The MIT Press, 2001. Print.

--------. *Software Takes Command*. 23 November 2008. Web. 14 March 2009.

Motte, Warren F. Jr, ed. *Oulipo: A Primer for Potential Literature*. Trans. Warren F Motte Jr. Normal, IL: Dalkey Archives Press, 1998. Print.

Murray, Donald. "Teaching Writing as Process Not Product" *Cross-Talk in Comp Theory*. Ed. Victor Villanueva. Urbana, IL: National Council of Teachers of English, 2003. Print.

Newman, John. Untitled. Web. 15 Mar 2008.

Paulos, John Allen. *Innumeracy: Mathematical Illiteracy and Its Consequences*. New York: Hill and Wang, 1988. Print.

Schimmel, Annemarie. *The Mystery of Numbers*. New York: Oxford University Press, 1993. Print.

Snow, C.P. *The Two Cultures: And a Second Look*. Cambridge, England: Cambridge University Press, 1986. Print.

Stewart, Ian. *Letters to a Young Mathematician*. New York: Basic Books, 2006. Print.

Ulmer, Gregory L. *Heuretics: The Logic of Invention*. Baltimore: The Johns Hopkins University Press, 1994. Print.

Valluri, Gautam. "The Vertigo Zoom: How I Learned to Fall into an Infinite Abyss While Standing Still." *Broken Projector*. 27 August 2007. Web. 28 May 2009.