Creativity Support for Computational Literature

By

Daniel C. Howe

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 2009

_____

Dr. Ken Perlin, Advisor

“If I have said anything to the contrary I was mistaken. If I say anything to the contrary again I shall be mistaken again. Unless I am mistaken now. Into the dossier with it in any case, in support of whatever thesis you fancy.”

- Samuel Beckett

ACKNOWLEDGEMENTS

I would like to thank my advisor, Ken Perlin, for his continual support and inspiration over the course of this research. I am also indebted to my committee members, Helen Nissenbaum, Katherine Isbister, Matthew Stone, and Alan Siegal for their ongoing support and generous contributions of time and input at every stage of the project. Additionally I wish to thank John Cayley, Bill Seaman, Braxton Soderman and Linnea Ogden, each of whom provided important contributions to the ideas presented here. While there are too many to mention individually here, I would like to thank my colleagues, collaborators, and friends who have been generous and tolerant enough to put up with me over the past five years. Finally, I would like to thank my parents and brother who have given me their unconditionally support, however misguided or unintelligible my direction, since the beginning.

ABSTRACT

The creativity support community has a long history of providing valuable tools to artists and designers. Similarly, creative digital media practice has proven a valuable pedagogical strategy for teaching core computational ideas. Neither strain of research has focused on the domain of literary art however, instead targeting visual, and aural media almost exclusively.

To address this situation, this thesis presents a software toolkit created specifically to support creativity in computational literature. Two primary hypotheses direct the bulk of the research presented: first, that it is possible to implement effective creativity support tools for literary art given current resource constraints; and second, that such tools, in addition to facilitating new forms of literary creativity, provide unique opportunities for computer science education.

Designed both for practicing artists and for pedagogy, the research presented directly addresses impediments to participation in the field for a diverse range of users and provides an end-to-end solution for courses attempting to engage the creative faculties of computer science students, and to introduce a wider demographic—from writers, to digital artists, to media and literary theorists—to procedural literacy and computational thinking.

The tools and strategies presented have been implemented, deployed, and iteratively refined in real-world contexts over the past three years. In addition to their use in large-scale projects by contemporary artists, they have provided effective support for multiple iterations of 'Programming for Digital Art & Literature', a successful inter-disciplinary computer science course taught by the author.

Taken together, this thesis provides a novel set of tools for a new domain, and demonstrates their real-world efficacy in providing both creativity and pedagogical support for a diverse and emerging population of users.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Literary artists who want to make use of programmable media... should not have to build their own tools. Until recently this has been the debilitating norm. – John Cayley [2009]

1.1 Supporting Literary Creativity

Computer science research in artistic creativity support has made significant progress in recent years, creating powerful new tools for photography, film and video, animation, drawing and music, that have transformed standard practices and inspired newly expressive forms. Similarly, creative arts practice in digital media has proven to be a valuable pedagogical strategy for teaching core computer science ideas, both within and beyond the boundaries of the department. Somewhat surprisingly however, given the historical association between the two fields[1], such research has rarely focused on the domain of literary art, instead almost exclusively targeting visual, and aural media[2]. In fact, the selection of computational tools available to contemporary literary artists remains largely unchanged over the past several decades. As one frustrated practitioner commented, "Basically, we are asking [the digital writer] to create sculptures using a hand cranked ice cream machine" [Larson 2005].

Metaphors aside, it would seem clear that literary artists have experienced relatively few benefits from the affordances of computational methods. Not only is this a missed

[1] Some examples of computer science research leveraging literature include Christopher Strachey and Alan Turing, Andre Markov and Claude Shannon, and, more recently, Selmer Bringsjord.

[2] Another important area of interest involves computer gaming, which often integrates several of these media types, e.g., audio, video, sound, and interaction.

opportunity for the creativity support community, but computer science educators have also been unable to leverage the synergies between these two areas as a means of advancing pedagogical goals. The synthesis of computer science, creativity support, and literary art, as argued below, holds unique potential, not only to facilitate new forms of literary creativity, but also to introduce a diverse population—from writers, to digital artists, to media and literary theorists, to creatively-motivated computer science students—to procedural literacy and computational thinking[3]. This dual-focus on creativity, both as and end-in-itself, and as a educational strategy, represents a potentially important means of broadening the appeal of computer science for a new generation of researchers, artists, and educators.

1.2 Current Creativity Support Tools

Over the past several decades, creativity support research has helped to generate an impressive range of computational tools for artists. The majority of these tools, however, have focused on visual and aural media, specifically film, animation, photography, music, etc.), as well as, to a lesser degree, architecture, industrial design, sculpture, and performance. Tools for language-based art, however, remain surprisingly rare. In fact, the selection of computational tools available to contemporary literary artists, especially those offering some degree of programmatic control, remains largely unchanged over the past 20 years; a situation that has generated significant frustration among practitioners in the literary community. As one renowned scholar and practitioner says , "I've often speculated, bitterly, as to why there is

[3] The terms 'procedural literacy' and 'computational thinking' are used somewhat interchangeably below though they have somewhat different connotations. The central ideas which both share are: a) an understanding of process (in contrast to a focus on programming or technical facility); and b) the application of core computational ideas (abstraction, decomposition, automation, recursion, etc.) to other areas of research and/or practice.

no word processor with the kind of filters and effects that are standard features in any of hundreds of graphic or audio manipulation programs" [Cayley 2009]. So why is it that, beyond the ubiquitous word processor, tools for language-based art-practice are so rare? Is it that they are more complex to realize? Or that they require inordinate processing and storage resources? Or is it simply the case, as Brion Gysin [1973] famously quipped, that "writing is still 50 years behind painting"? While no single answer appears to adequately answer this question—it is likely a combination of these and other factors—there are reasons to believe this situation is ripe for change.

Two early examples of authoring environments exploring the possibility for such change are the Dramatica (http://www.dramatica.com/) and StorySpace (http://www.eastgate.com/Storyspace.html) systems, both created in the early 1990s. While neither of these tools provide programmatic support—users do not have direct access to the code for their work—they do represent important early attempts to provide literary artists with creativity-enhancing tools. Unfortunately however, the fact that both were designed for quite narrow contexts—screen-writing and hypertext fiction respectively—has resulted in their adoption only in small niche markets. Further, the fact that both are proprietary, expensive, and closed-source has likely dissuaded a significant number of potential users, specifically artists and educators. Further, at least in the case of StorySpace, the fact that it does not produce web-executable content has resulted in its near obsolescence, as the CD-ROMs for which it was designed have fallen out of common usage. And while Dramatica retains a somewhat sizeable user-base among screen-writers, it generates only traditional (print-on-page) outputs and provides no support for alternative digital forms, e.g., interactive, generative, or multi-modal texts, the focus of the work presented here.

In light of this situation it has become almost common practice for literary artists interested in using computational methods to either create their own tools or to modify existing tools designed for their, often very different, purposes. In fact, artists have become somewhat proficient at creatively re-purposing 'traditional' software for their own agendas. Some examples (discussed further in chapter 4) include Jackson Mac Low's use of Microsoft Word, John Cayley's use of Apple's QuickTime components, and David Bryne's repurposing of PowerPoint[4]. to mention just a few well-known examples. Other artists, David Rokeby (author of OpenVNS), Charles Hartman (author of MacProse), Alex Galloway (author of Carnivore), and Casey Reas and Ben Fry (authors of Processing), have chosen to build their own software, rather than working with existing platforms. While these examples demonstrate that both high-quality software and artistic artifacts can be generated by practitioners in do-it-yourself fashion, this approach is only practical for a very small proportion of those who might benefit from such tools. For the majority of writers (and artists) interested in exploring computational literature, the situation is, to borrow Cayley's language, "debilitating" at best. This is not only an unfortunate situation for literary artists, but also for students and educators in both computer science and digital arts for whom creativity writing in digital media could prove to be a productive educational environment.

1.3 Computer Science and Creative Writing

While writing skills have received some attention in recent years from computer scientists [Pesante 1991; Van Wyk 1995; Ladd 2003; Hoffman 2006], the similarities (and differences) between *creative* writing and programming have been discussed primarily in other disciplines. One such example is the digital writing community, as perhaps best

[4] For more information on this work see http://www.davidbyrne.com/art/eeei/.

demonstrated by the 2008 NSF-sponsored *Codework* conference which brought together a number of researchers and practitioners in the field to discuss this very topic. Of the nineteen papers presented at the conference, however, only one was authored by a computer scientist[5], who happened also to be a practicing literary artist. Similarly, the 2009 ACM CHI *Workshop on Computational Creativity Support* featured sixteen papers on computationally-augmented creative practice, just one of which, by this author, addressed the context of writing and/or literature.

But recognizing the potential synergies between computer science and literature is only a first step. Because of the inherent complexity of natural languages, research in the literary context—perhaps more so than any of the other arts—requires tool support with a degree of sophistication that has thus far been unavailable. In fact, tool support for language-oriented computer science education has presented significant hurdles [Bird and Loper 2002]. Students often enter courses with vastly different backgrounds and skill sets, and their creative projects tend to integrate a variety of programming tasks. One approach to this problem has been to employ multiple programming environments, with each providing support for some specific task of interest. For example, an introductory computational (or 'digital') literature course might use “Perl” for text parsing and web-crawling, Apple's built-in “talk” facility for text-to-speech, “Flash” for text-display and animation, “Max/MSP” for audio support, and one of several research-oriented natural language packages for statistical analysis. By relying on the built-in features of these languages and platforms, instructors can avoid developing a software infrastructure on their own.

---

[5] See position papers from the *Codework* conference at http://www.clc.wvu.edu/projects/codework_workshop/codework_position_papers.

One of several unfortunate consequences of this strategy is that significant time must be devoted to teaching the specifics of each new environment. This increases the delay before students are able to engage substantive topics, whether in software engineering, natural language processing, or the creative practice of computational literature itself. Moreover, students cannot build on previously learned material in subsequent assignments. This lack of *scaffolding* is especially problematic when student projects tend to span a variety of 'core' tasks and thus require multiple environments to be bridged in a final project; an often formidable task. For example, a somewhat typical student project that involves extracting text from the web, altering it in some way, and visually displaying the results, accompanied by text-to-speech, might use most or even all of the environments mentioned above.

1.4 Motivations

Although the complexities of the literary context should not be underestimated, the benefits could be substantial were the considerations above to be effectively addressed. In addition to its importance in contemporary art, computational literature presents a unique context for conveying core computational ideas to students in an intuitive and organic fashion. As an example, consider the range of key computer science concepts that arise in conjunction with even the most introductory topics in a course on computational literature. In the two mini-projects[6] presented in chapter 3 for example, students are naturally exposed to a significant number of core ideas typically covered in an introductory computer science sequence; from finite-state automata to context-free grammars and the language hierarchy; from data structures to parse trees; from regular expressions to recursion. Rather than

[6] Discussed in the context of "Programming for Digital Art & Literature", a course taught by the author at Brown University.

appearing to students as arbitrary additions to the "real" topic at hand, the relevance of these core computational ideas, when presented with effective tool support, becomes readily apparent.

With the aim of addressing these concerns, this thesis introduces RiTa, a software toolkit designed and implemented specifically to support creativity in computational literature[7]. Two primary hypotheses have directed the research presented: first, that it is possible to implement effective creativity support tools for literary art given current resource constraints; and second, that such tools, in addition to facilitating new forms of literary creativity, provide unique opportunities for computer science education. Designed both for practicing artists and for pedagogy, RiTa directly addresses impediments to participation in the field for a diverse range of users and provides an end-to-end solution for courses attempting to engage the creative faculties of computer science students, and to introduce a wider demographic—from writers, to digital artists, to media and literary theorists—to procedural literacy and computational thinking.

RiTa covers a range of computational language tasks including text analysis, generation, display and animation, text-to-speech, text-mining, and access to external resources such as WordNet[8]. In addition to their use in large-scale projects by contemporary artists, they have provided effective support for multiple iterations of 'Programming for Digital Art & Literature', a successful inter-disciplinary computer science course taught by the author. Students from a wide range of backgrounds (creative writers, digital artists, media

---

[7] Other names for this sub-field of computational art focusing on language and/or literature include "Electronic Writing" or "E-Writing", "Digital Writing", and "Writing in Programmable Media".

[8] See http://wordnet.princeton.edu/ and [Fellbaum 1998].

theorists, linguists and programmers, etc.) have been able to rapidly achieve facility with the RiTa components, to gain an understanding of core language processing tasks, and to quickly progress on to their own creative projects in computational literature.

The RiTa toolkit, freely available via an open-source Creative Commons[9] license, is implemented in Java, optionally integrates with the Processing[10] language environment, and is compatible with all common operating systems and web browsers.

1.5 Contributions

This thesis will present RiTa from a number of perspectives: as tools and affordances for practicing writers; as a pedagogical strategy, both for teaching procedural literacy to humanities students, and for engaging computer science students with creative practice; and as a real-world testing ground for creativity support principles, providing a unique context for assessing the efficacy of design and evaluation strategies. As such, the contributions of this research fall into two related sub-fields: creativity support tools (CST) and computer science education[11].

In the context of creativity support research, RiTa represents the first end-to-end programmatic toolkit designed expressly for computational literature. In addition to those artists self-identifying as "digital" or "computational" writers, RiTa targets two distinct groups who might benefit from the synthesis of computational techniques and language-based

---

[9] See http://creativecommons.org/.

[10] See http://www.processing.org/ and Reas and Fry [2007].

[11] As mentioned above, we include in this category the notion of "computational thinking" as advanced by Wing [2006] and Guzdial [2008], and "procedural literacy", as advanced by a number of researchers since the early 1960s [Greenberger 1962; Sheil 1980; Bogost 2005; Mateas 2005].

practice. These are a) analog writers who have lacked simple tools with which to experiment computationally, and b) computational artists with a sound, image, or physical orientation, who are interested in extending their practice to include experiments in language.

As a pedagogical tool, RiTa represents the first toolkit intended specifically as an end-to-end teaching tool for courses in computational literature. It has been designed, from the start, to enable students with little or no programming experience to quickly experiment with generative computational methods. As such, it includes carefully written and updated documentation, numerous examples, and a wide range of publically available projects. Additionally, it includes a range of supporting applications that enable integration with popular teaching tools (e.g., Processing), and easy publication of sketches and source code from within popular development environments (e.g., Eclipse).

RiTa represents a unique approach to teaching computational thinking and procedural literacy for a diverse range of students beyond the borders of the computer science department. At the same time, the toolkit includes enough advanced functionality to engage creatively-oriented computer science students with a range of more advanced techniques in natural language processing and generation. Because it supports this broad range of experience, RiTa also provides a natural means with which to enable cross-disciplinary collaboration between students coming from diverse backgrounds. Lastly, because it is an end-to-end solution, including non-linguistic facilities for animation, text rendering, image, audio, etc., it frees the instructor from designing and implementing custom tools, and/or teaching multiple environments.

Finally, the design strategies employed in building a creativity-supporting toolkit have the potential to shed light on new strategies for software design, not only in the arts context, but in a wide range of creativity support research targeting both educational and

professional agendas. A clear benefit of tools that satisfy this joint goal is their ability to facilitate smooth transitions for students as they move between educational and professional environments[12]. In this way our approach, integrating value-sensitive design [Friedman et al. 2006; Flanagan et al. 2008], constructivist/constructionist educational philosophies, and design principles from creativity support research, is unique. Whether it is applicable to user-groups beyond the arts domain is a question that will require significant further research Toward this end we provide some unique experimental approaches for evaluating these tools, examining a range of metrics; from usability, to user attitudes, to programmatic skills, to more subjective measures of divergence; all implemented within the real-world framework of a typical semester-long university course.

1.6 Overview

Following this introduction chapter, chapter 2 presents a full description of the RiTa toolkit, from high-level goals, to design criteria, to a component-level description of functionality. Chapter 3 describes how RiTa has proven to be a useful teaching tool, enabling humanities students to easily engage with central concepts in computer science, and allowing computer science students to develop their creative faculties by engaging with 'real' problems in which they have a personal stake. Chapter 4 describes the range of prior work, both within and beyond computer science, that influenced the design, implementation, and development of the toolkit. In addition to supporting creativity for practicing artists, chapter 5 describes evaluation measures on a number of dimensions, from usability, to procedural literacy, to creativity support. Chapter 6 attempts to abstract a set of general principles for creativity

[12] This is especially true of artists who work with new technologies,. In such cases, it is not unusual for an single individual to alternate between teacher, student, and practitioner within a short span of time.

support and software design for the arts, and concludes with directions for future research. The appendix presents links to a range of related resources for RiTa, including examples, documentation, and a student project gallery.

CHAPTER 2: TECHNICAL DETAILS

"Nothing is more complex than the artifacts of the literary imagination, and any software that works to empower the literary as it traverses the landscape of computational technology will be equally complex... and enormously useful." - J. Carpenter

2.1 Introduction

The RiTa[13] toolkit is a suite of open-source components, plugins, tutorials, and examples that provide creativity support for a range of tasks related to the practice of writing in programmable media. Designed both as a toolkit for practicing writers and as an end-to-end solution for computational writing courses, RiTa includes components for text analysis and generation, animation, display, text-to-speech, web-based text-mining, and interfaces to external resources (e.g., WordNet). As RiTa optionally integrates with the Processing environment for arts-oriented programming, artists and students have immediate access to a large community of practicing digital artists, and can easily augment RiTa's functionality via the vast collection of libraries available.

The bulk of the RiTa toolkit is implemented as a Java library consisting of ten independent packages. The core object collection contains approximately 20 classes within the rita.* package, all of which follow similar naming and usage conventions. Additional packages provide support for these core objects, but are not directly accessed in typical usage. The philosophy behind the API is to be as simple and intuitive as possible, while still providing adequate flexibility for more advanced users. In addition to the core classes, RiTa provides statistical models for tagging, chunking, and parsing, and a plugin for the popular Eclipse development environment to facilitate publishing and sharing of projects (and source

[13] The name RiTa derives from Old Norse, meaning to scribble, scratch, or inscribe.

code). The following list provides a high-level view of the functionality provided by the RiTa tools, with each item discussed in detail below:

- Literary text-generation via Markov chains and context-free/mildly sensitive grammars.
- A customized user-modifiable Lexicon with letter-to-sound rules for unknown words.
- Feature extraction of Sentences, Words, Syllables, Phonemes, Stress, and Part-of-Speech.
- Support for analysis and generation of literary effects such as rhyme, alliteration, and meter.
- Efficient verb conjugation, noun pluralization, stemming and part-of-speech tagging.
- Support for statistical (maximum entropy) tagging, chunking, and (recursive) parsing.
- Web-accessible, cross-platform Text-To-Speech support (Windows, Mac, Linux.)
- Simple, browser-accessible integration with the WordNet lexical ontology.
- Concordances and Key-Word-In-Context (KWIC) model implementations.
- Support for transparent server mode for both local and remote data persistence.
- Real-time unigram, bigram and weighted-bigram measures via search-engines.
- Web/text-mining capabilities via both regular expressions and the Document-Object-Model.

- A customized behavior model for events and animation with a built-in behavior library with over a dozen varieties of 'easing' for custom typographical effects in 2D and 3D.
- Simple API for augmenting text with image and audio (aiff, wav, mp3) resources.

2.2 Design And Implementation

The following sections presents a range of design considerations that influenced the development and iteration of the RiTa tools. Beginning with the choice of coding environment, we present the design criteria (and *anti-criteria*) that emerged in and through the iterative process of teaching and creating with RiTa over the past three years. We present several concrete examples of so-called *design tensions* [Flanagan et al. 2008] in which specific usage scenarios cast two or more of these criteria into conflict, and discuss our resolutions to these. Following detailed functional descriptions of the core RiTa objects, a range of usage scenarios and high-level patterns are presented to demonstrate the interrelations between components.

2.2.1 Programming Environment

In addition to the specific functionality to be implemented in the toolkit, an important early decision concerned the choice of programming environment.[14] A number of unique

[14] "Environment" in this context refers to the combination of programming language and associated libraries, development environment (e.g., IDE), and associated tools for writing, compiling, debugging, running and publishing programs.

considerations influence this choice for artists and students working in computational literature.

First, it is important that the environment have a relatively shallow learning curve, so that novice programmers receive immediate rewards for their efforts. Second, it should support rapid prototyping and short develop-and-test cycles. Third, it should be widely used so that questions, examples, and projects can be easily located on the web. Fourth, it should facilitate both structured and unstructured programming (specifically, the language should provide both procedurally-oriented functions for rapid prototyping and full object-oriented support for larger, more complex programs.) Fifth, it should provide end-to-end support for tasks that, though not central to the practice of generative language, are often necessary for fully realizing a work (these include simple, yet robust access to sound, network, and graphics libraries). Sixth, all library functions should run in (perceptual) real-time (there should be no need for offline processing). Seventh, student programs should be easily publishable and include source code to facilitate knowledge-sharing. Finally, all programs should be executable in a web-browser environment to eliminate any dependencies on hardware, operating system, and configuration, which can waste valuable time in workshop-style settings. The following section on design requirements presents a more detailed discussion of these criteria.

In surveying the available environments, only two options presented themselves that appeared to satisfy a majority of these criteria, both of which used somewhat extensively in digital arts and introductory programming courses; specifically Adobe's *Flash* and associated *ActionScript* language, and *Processing* by Casey Reas and Ben Fry, an API and simplified programming environment using Java. The various tradeoffs between these two web publishing environments have been the subject of much debate, but for our purposes, the fact

that Flash programs could only be authored in the proprietary, relatively expensive, and somewhat opaque Flash environment simplified the decision.[15] Further we found Java to be a good fit as a general purpose language for teaching programming, not least of which due to the fact that it is free, open-source, and in wide use across a range of contexts. Further, it supports powerful (if less than perfect) object-oriented capabilities and has a relatively transparent syntax, especially for those familiar with other C-based languages. Lastly, in addition to a large and active user community, there exist a vast number of libraries and frameworks (both internal and external) available to extend the core language in many imaginable directions. Equally importantly, the Processing environment, even more so than Flash, provided students with immediate access to a large community of practicing digital artists, a majority of which were already committed to open-source practices and shared their code freely online.

This is not to say that this choice, and Java itself, does not present potentially significant drawbacks. A specific example of such a drawback is the separate compilation step required before one can run a Java program, a language "feature" in direct conflict with a number of the design criteria presented above. This problem, however, had already been addressed in two ways: first by the Processing environment itself, which combined "compile" and "run" into one action (with a simple "play" button widget), and second, by the Eclipse IDE which performs continual compilation, highlighting syntax and other errors as the programmer types. By the second iteration of the course, students were encouraged to use

---

[15] This situation has changed somewhat recently with Adobe's announcement in 2008 that the Flex 3 SDK would be made available under the open-source Mozilla Public License. Flash Player, the runtime on which Flex applications are viewed, and Flex Builder, the IDE used to build Flex applications, however, remain proprietary.

either or both of these tools, depending on their experience and comfort level with each. For those wishing to migrate from Processing to Eclipse, an online tutorial was created and an optional help-session, led by the teaching-assistant was scheduled.

A second drawback to using Java in the classroom was its lack of a rapid prototyping framework, as even the simplest Java programs require class definitions, typed functions, a main routine or html-page, and at least some understanding of "static" scoping for variables and methods, not to mention a host of potentially troublesome setup options including file and directory/package structures and classpath specification. Again however, most of these issues had already been addressed by the makers of Processing, who explicitly designed the environment to eliminate such considerations from the learning process. Running a simple Processing program—e.g., drawing a line to the screen—can be accomplished by writing a single line of code and pressing the "play" button (see Figure 1). Further, though Processing supports object-orientation (all of the Java language is in fact supported), its basic mode of operation is procedural, with simple function calls made via the Processing API. As new techniques were learned—e.g., user-defined functions, objects, classes, interfaces, polymorphism—students could integrate these into more complex programs while still using the Processing functions where desired (either within Processing or in a more advanced environment like Eclipse.) As one student commented[16] in an end-of semester reflection,

> [R]apid prototyping is definitely the most important truth I will take away from this course and this semester in general. Due to the freedom of our digital medium it takes no time to come up with a random idea and layer it with level upon level of abstraction, but to know it's a good idea you pretty much have to prototype it and the faster the prototyping the faster you can

[16] To protect student anonymity, names and dates have been omitted for student quotations. Unless otherwise noted, all student quotations are from the 'Programming for Digital Art & Literature' course.

drop that bad ideas. I just hope I can practice what I preach now that I have a bit more wisdom.

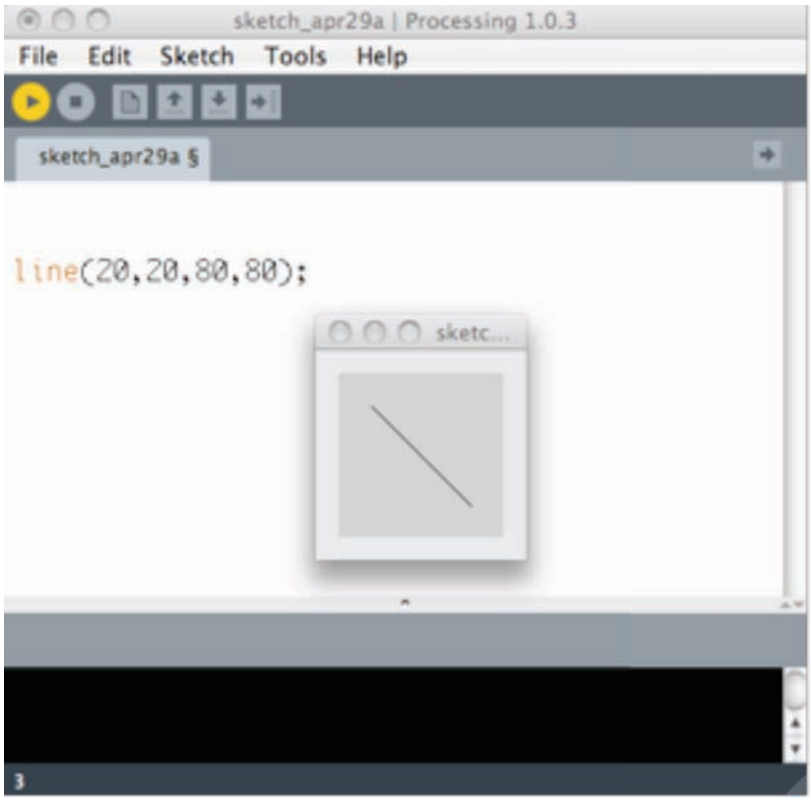

Figure 1: A one-line Processing sketch.

Of course there are still other aspects of Java, at least "out-of-the-box", that are less than ideal for arts-based software practice, e.g., the lack of dynamic typing, first-class functions, an interpreted execution mode, and "live-coding" support (topics discussed elsewhere,) all of which are present in languages like Ruby and Python. Yet for the requirements we had decided upon, most specifically web-based graphical execution, it was clearly the best selection from the existing choices.

2.2.2 Design Criteria

> *The constraints of technique, resources, and economics underdetermine design outcomes. To account fully for a technical design one must examine the technical culture, social values, aesthetic ethos, and political agendas of the designers. [Pfaffenberger 1992]*

Once a base environment in which to implement the toolkit had been selected, we identified several criteria which we felt would be important in design and implementation, generally following the methodologies laid out in earlier work on design methods [Friedman et al. 2002]. These criteria are listed below in their initial order of importance, though their relative priority changed throughout the design process. Likewise, we also attempted to decide what goals the toolkit would *not* attempt to accomplish and thus created a list of explicit *anti-criteria*, which the toolkit was not expected to satisfy [Bird and Loper 2002]. As discussed elsewhere[17], we were cognizant of the possibility that specific implementation decisions could (and likely would) bring design criteria into conflict with one another, specific cases of which are discussed below.

2.2.2.1 Requirements

- *Ease-of-Use.* The primary purpose of the toolkit was to allow students to effectively implement their own creative language works. The more time students spent learning the toolkit, the less useful it would be.
- *Consistency*. The toolkit should use consistent naming, syntax, functions, and design patterns. Therefore if one object was created via the traditional call to "new()", another should not be created via a static or factory creation pattern.

---

[17] See the discussion of "design conflicts" in Howe and Nissenbaum [2008].

- *Extensibility*. The toolkit should easily accommodate new component implementations, whether for replication of existing functionality (in exercises or assignments), or for the addition of new (for the needs of specific projects.)
- *Documentation*. The toolkit, its components, and its implementation should be carefully and thoroughly documented and updated. All naming conventions should be carefully chosen and consistently used.
- *Design-By-Interface.* Even if not exposed in basic usage, all core objects should be implementations of interfaces and typed internally as such, thus allowing students to build and easily test their own implementations in exercises or assignments. In the case of RiTa, this practice facilitated the development of multiple implementations of an object for different use-cases; e.g., a transformation-based tagger when speed and memory footprint were of primary concern, and a slower but more accurate maximum-entropy version for general use.
- *Small/Light.* To enable download and execution in web browsers, the library should be as efficient as possible with resources, most importantly browser memory. The most recent core version of RiTa (not including the WordNet database or the TTS voices] was ~2.5 MB.
- *Modularity*. The interaction between different components of the toolkit should be minimal, using simple, well-defined interfaces. In particular, it should be possible to complete individual projects using small parts of the toolkit without concern for how they interact with the rest. This allows students to learn the library incrementally over one or more semesters.
- *Share-ability*. Students programs should be easily exportable as web applets including generation of HTML pages and web-browsable source code. To facilitate

this, the toolkit should be efficient enough, in terms of memory, to ensure that student programs can run in common web browsers.

- *Performance*. Library functions should be fast enough that students can use the toolkit for interactive projects with functions returning in 'perceptual' real-time.
- *Transparency*. The library and its underlying support APIs (Java Processing, etc) should all be freely available with easily browsable and well-documented source code.
- *Platform Agnosticism*. The library should contain no operating system-specific code or behavior, allowing it be used in all major browsers and on all common platforms.

2.2.2.2 Anti-Requirements

- *Comprehensiveness*. The toolkit is not intended to provide a comprehensive set of tools for natural language processing or digital art. Instead, it should facilitate the addition of new functionality to either extend the toolkit itself or to integrate it with other libraries and tools.
- *High-Performance*. The toolkit need not be highly optimized for maximum runtime performance as long as the constraints mentioned above on memory and “perceptual” real-time are met.
- *Cleverness*. Clear well-documented designs and implementations are generally preferable to ingenious yet opaque ones, as advanced students (and practicing artists) should feel comfortable examining the source-code of the library itself.
- *Object-Orientation*. Consistency and conformity to intuitive use-patterns (both within the library and with the Processing environment) should come before best-practice rules and patterns for object-oriented design.

2.2.3 Design Tensions

At various times in the development and deployment of RiTa we identified tensions between two or more of our design criteria. When such tensions were recognized, following our prior work on activist gaming strategies [Flanagan et al. 2005], they were added to an evolving list, each of which was assessed in subsequent design iterations, much in the way that regression testing was employed to validate continuous technical progress.

2.2.3.1 Strings and Features

An example of a specific design tension that arose during implementation concerned RiTa's internal representation of character data. As discussed above, a flat String-based approach was eventually adopted, with each character string (potentially) linking to a map of key-value pairs (referred to as "features"). The potential augmentation of the Java String with such a map (the basic RiString object) allowed for data to be continually accumulated as an object was processed by different core components.

The notion of "features" originated in the Festival speech synthesis system[18], in which a similar mechanism was used to avoid the disadvantages of both "string-rewriting" and "multi-level data structures", common mechanisms in systems at the time [Taylor 1998]. Unlike Festival, RiTa does not maintain separate data-structures of "relations" between linguistic items, e.g., the words that constitute a phrase, or the phonemes that constitute a word. Rather, all such information is embedded in the feature data itself.

For example, one might begin with a RiString object containing the string "the white rabbit". This RiString might then be passed to a RiAnalyzer object where word-boundaries would be determined and each word tagged with part-of-speech, stress, syllables, and

[18] See http://www.cstr.ed.ac.uk/projects/festival/.

phoneme features, then to a RiChunker object which would add a "chunk-type" feature, tagging the phrase as a "noun-phrase". We might then tag the phrase with "custom" features specific to our application. As trivial examples we can use the URL (with key "source-URL"), where the phrase was found, perhaps during a text-mining operation, and the number of letters in each word (with key "letters-per-word"). Thus our feature data would appear as in Table 1, with custom feature displayed in italics.

keys	values
id	6
text	'the white rabbit'
tokens	the white rabbit
pos	dt jj nn
stresses	0 1 1 / 0
phonemes	dh-ax w-ay-t r-ae-b-ax-t
syllables	dh-ax w-ay-t r-ae-b / ax-t
chunk-type	noun-phrase
source-url	*http://alice.com/text*
letters-per-word	*3 5 6*

Table 1: Feature data for an example RiString phrase[19].

Once we have a RiString for a phrase containing multiple words, as above, we can use RiString.split(), to split the phrase into three new RiString objects[20], one for each word. Thus, assuming the RiString above was named "phrase1", the following lines of code:

[19] Note that the word and syllable boundaries are, in this case, the *tab* and *forward-slash* characters respectively, though these can be programmatically specified by the user.

[20] Note that each RiString is assigned a unique identifier on creation to enable easy cross-referencing via the feature-map. For example, a RiString containing a single word might have

```
RiString[] words = phrase1.split();
RiString lastWord = words[2];
```

would result in a reference to a new RiString representing the last word of the phrase, i.e., "rabbit", the features of which are shown in Table 2.

keys	values
id	9
text	'rabbit'
tokens	rabbit
pos	nn
stresses	1 / 0
phonemes	r-ae-b-ax-t
syllables	r-ae-b / ax-t
num-letters	*6*

Table 2: Feature data for an example RiString word.

Notice that the "split" operation maintains the default features that are still relevant (e.g., part-of-speech, stress, phonemes, syllables, etc.) to the whole, and deletes those that are not (e.g., "chunk-type", as the individual word is no longer part of a "noun-phrase"). Such "smart" decomposition works similarly for custom features (e.g., the trivial "letters-per-word" feature used in the example), assuming that the feature in question contains the same number of items, when split on the current word-delimiter, as the number of total words in the phrase. Once we have a reference to a RiString with a single word (which can be queried via the method RiString.getWordCount()), we can add a range of additional features via

---

a feature called "sentence-id", identifying it as being part of a sentence represented by another RiString.

various other RiTa objects. For example, we might use RiWordNet to add synonyms, meronyms, or hypernyms , or perhaps the RiLexicon object to add rhymes, "soundex" matches, and alliterations.

This approach provided several advantages: it was simple and easy to learn; it was easy to customize and extend; and it facilitated data sharing both between RiTa objects, and with external libraries, programs and web-services. Further, as Java strings were one of the few basic variable types (along with the primitives) that were used in the Processing API, it was intuitive to new users and required little additional documentation or explanation. The lack of support in Java for operator overloading prevented the use of this augmented String from being truly transparent; that is, instead of typing (in typical Java syntax) *String s = "hello"*, RiTa users had to type *RiString s = new RiString("hello").* The argument could be made, however, that this is more consistent with Java's distinction between primitives and Objects, however inconsistent it may be; all other Java objects (except for String) require the use of the "new" keyword, while all the Java primitives do not; a fact that often causes some confusion for new users.

As might be expected, the usual Java String methods (in fact the entire CharSequence interface) could be invoked on a RiString object just as one would on a String. As mentioned above, RiString.split(), for example, would split the instance described above into three new RiString objects, one for each word. This allowed RiString objects to be passed along a pipeline (similar to Unix command-line tools), and for lookups to be performed in "lazy" fashion, if and as needed. For example, if a core component (say the RiGrammar object) needed to know the part-of-speech for a RiString with which it was generating a phrase, it might ask whether the "pos" feature was present via the following line:

```
if (theRiString.hasFeature("pos")) {
      ...
}
```

If not, it might invoke the RiPosTagger object to first add the part-of-speech feature before proceeding with generation. Objects in different states of analysis could thus be exchanged with features added only as needed, enabling through a sort of ad-hoc introspection mechanism, a weak version of polymorphism (the lack of which being one critique of such a non-hierarchical approach). In fact, it was this feature that later enabled the use of RiTa objects in a real-time drag-and-drop environment for language processing (See chapter 6: Future Work).

As mentioned above, extending RiString objects with custom feature types was trivial, as one needed only to add a new key-value pair that could later be checked for and accessed (rather than creating a new subclass which required the understanding of inheritance, overloading, or overriding). Many students used this facility to add features that were specific to their projects, such as a feature denoting a "semantic-link" between word pairs, or a "source" tag during web-parsing to later identify the page on which a word or phrase was found. This structure also eliminated the need for many external data structures that would otherwise have been necessary. A potential downside was the aggregate expense for programs with many RiString objects. In some programs, for example, a large number of feature maps might be allocated (perhaps even one per word in a large text), each containing as few as one feature. Obviously in such cases a single map (or hashtable) with words as keys and desired feature as values would be more efficient. As one might expect, such optimizations were not encouraged for students in the early stages of their projects, but were easily implemented if and as necessary in later stages.

2.2.3.2 Statistical Crossover

A second design tension involved the constraint of (perceptual) real-time with the desires of students to work with very large amounts of text. For example, when working with *n*-gram models, students often wanted to encourage frequent "crossover" (defined below), among a large number of texts by using a relatively low *n*-value. While all methods on RiTa's *n*-gram object, RiMarkov, returned very quickly no matter the size of the model, there were two related problems. First, large models require large memory and this directly conflicted with our constraint of having programs run as applets in standard web browsers. Second, large models can take significant time to load, which a) may result in unacceptable delays for users, and b), more importantly, make development and debugging a tedious process as each run of the program might require up to 30 seconds to load, a situation that directly conflicted with the design constraint of facilitating rapid (or *micro*) iteration, as described below.

Thus we see that a single unexpected use-case threatened three related but distinct design constraints (perceptual real-time, micro-iteration, and web-based execution). We should note that this particular set of issues, the memory requirements of large textual models and corpora, is one reason why creativity support tools for this context have been so difficult to realize in the past [Howe and Soderman 2009]. Perhaps for the first time however, the computational power of consumer-level computers is near the point where this difficulty can be successfully mitigated.

Our attempt to resolve this design tension resulted in two (parallel) strategies, one involving the creation of a new mechanism (and package) in the toolset, presented to library users via the RiTaServer object described above, and one involving a change to the RiMarkov class itself. As computational writers often have very different goals than typical natural language researchers, it should come as no surprise that their perspective on a specific

process may be vastly different than a researcher interested in say, machine translation, even when they are using they very same algorithmic method. The use of *n*-grams (or Markov models) presented just such a case as, over the course of the semester, it became clear that the methods typically found in an *n*-gram package were not sufficient for students' needs. One example became evident through conversations with several students attempting to use large text models for *n*-gram based generation who, unlike in the translation case, were most interested in a property we came to call "crossover".

As an example, we can take two texts, A and B, with the number of sentences in each being *sA* and *sB* respectively. From the difference of the sets of sentences in these texts, there will be some number of unique sentences in each, *uA* and *uB* , and some number $d = (sA - uA) = (sB - uB)$ of sentences present in both. Depending on the uniqueness of the texts at the sentence level, as represented by

$$\epsilon = 1 / ((2d+1 / (sA + sB))),$$

some percentage of all *n*-grams in the joint model will contain words unique to each text. It was just these "crossover" sentences, present in neither text or A nor B that were of primary interest for the subset of users attempting to add more and/or longer texts. While the typical application of *n*-grams would be to find sentences that were *most* likely to occur, the goal of these users, rather interestingly, was the opposite, specifically to find *novel* sentences, those that could logically occur, according to the constraints of the model, but were less (or even least) likely to do so. This *inverted* use pattern, related to the artistic strategy of *misuse,* proved to be a recurring theme when algorithmic techniques were borrowed from existing areas of research for use in creative practice, a topic discussed further in the chapter 3.

For this group of users, however, a simple constraint on the generation process achieved the same goal (specifically, increased "crossover"), giving higher probabilities to

those sentences not existing in any of the input texts from which the model was built. Since texts were processed sequentially, when this option was specified (via a simple method call), it was easy to build a compact lookup table for these sentences (again assuming the number of input texts is not too vast) and then use the table to generate higher percentages of sentences not already existing in the model.[21] Sentences generated by the component could also be added to the lookup table, thus ensuring that no duplicate generations occurred.

2.2.3.3 RiGrammarView

A third design tension involved the RiGrammar object and a number of our design constraints, specifically transparency, open exchange, and rapid (or *micro*) iteration. To support the first three of these constraints, we designed the RiGrammar object to read grammar files from plain-text files stored in the user's resource (or data) folder, along with any required fonts, images, sound files, etc. When exported via the RiTa plugin, these files were linked, along with the project's source files, and displayed in the HTML tags for the page. Thus viewers of the piece interested in its inner workings could access at last two additional layers below its surface representation. In addition to facilitating transparency regarding the workings of piece, it also enabled open exchange in that students could download and experiment with each others grammar files. Further, as it provided a relatively clean separation of concerns, between process (the source) and data (the grammar), it satisfied the generic design principle of modularity.

[21] Note that this simple constraint does not guarantee "true" crossover, only generation of phrases or sentences not previously seen in any of the input texts.

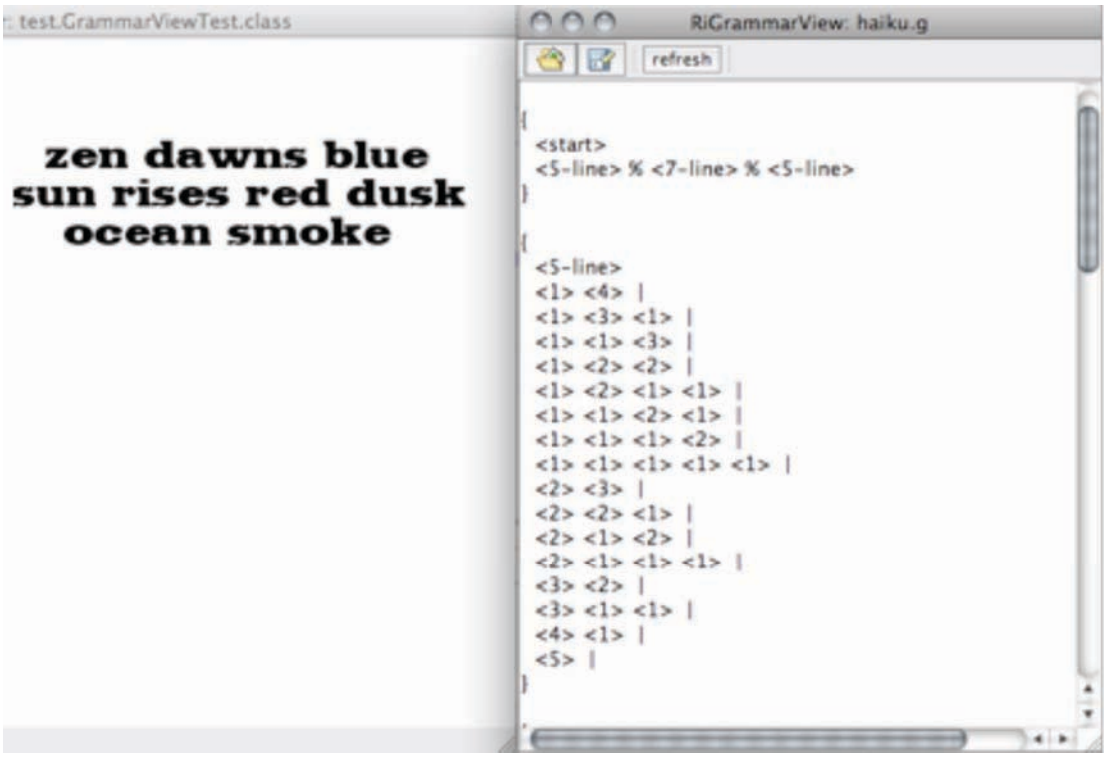

Figure 2: Screenshot of the RiGrammarView tool.

Unfortunately this setup significantly limited students ability to rapidly iterate during the potentially long phase of grammar development. With the original setup (depending slightly on the environment used), each time one wanted to modify the grammar, it required closing the program, modifying and re-saving the grammar file, recompiling the source, then re-launching the program. Over hundreds and thousand of small grammar changes, this time accumulated and caused significant frustration. Much like in the case of the RiTaServer above, our resolution to this tension involved the implementation of an auxiliary tool, namely the RiGrammarView component (as show in Figure 2). By adding a single line to their program (see Figure 3), users could invoke a custom editor that loaded (by default) the contents of the current grammar file. Then, by pushing a 'refresh' button the RiGrammarView would dynamically swap out the grammar rules in the associated RiGrammar object, replacing them with those found in the editor text.

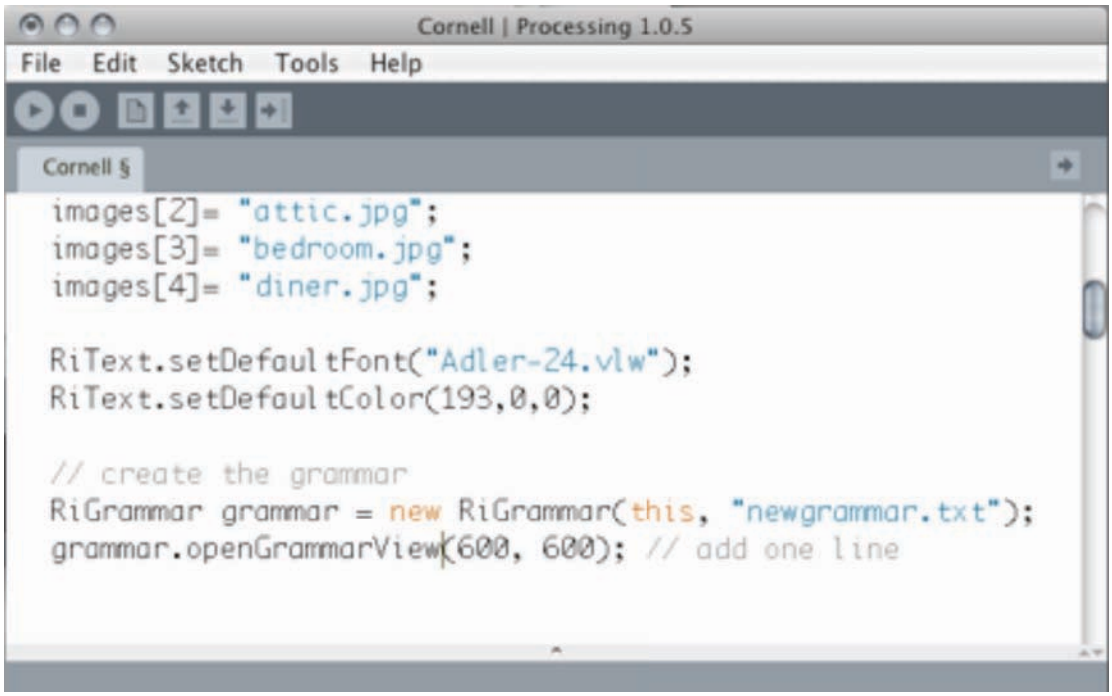

Figure 3: Code invoking the RiGrammarView editor.

With this setup users needed only to make changes and hit refresh to immediately see the results in their running program. If the changes were satisfactory the grammar file could be saved to disk. If one desired to test a program with multiple grammars, each could be loaded from separate files at runtime, and the results compared without restarting the program. When satisfied with the grammar as written, one could simply comment out the single line of code invoking the editor and publish their sketch as usual. In this way we were able to maintain a clean separation of concerns between code and grammar files, facilitate open exchange and transparency, and still enable rapid micro-iterations, here within a single execution cycle.

2.3 Component Descriptions

> *Researchers also need to distinguish between software features that are merely novel and those that are demonstrably effective in enabling users to produce creative outcomes. [Schneiderman et al. 2006]*

Since its creation, the RiTa toolkit has developed on a number of parallel tracks. New objects have been defined and new methods added to accommodate functionalities that arose during real-world use. Similarly, new mechanisms were implemented to better address elements of the project in which design tensions were identified. At the same time, the RiTa tools were continually re-factored to increase consistency, clarity, efficiency, and usability. This section describes the functionality provided by the objects in the "core" RiTa library. A brief description of each object follows, highlighting common usage patterns, and specific literary augmentations. Where applicable, explanations of design concerns leading to specific implementation decisions are presented.

The RiTa toolkit is implemented in Java, optionally integrates with the Processing language environment, and runs on all the major platforms including Windows, Mac OS X, and Linux/Unix. It is freely available under an open source Creative Commons license at http://www.rednoise.org/rita/. For further detail, see the complete API available at http://rednoise.org/rita/documentation/docs.html.

RiText

The basic object for displaying strings of text on screen in 2D and 3D. RiText contains a variety of utility methods for manipulating typography, fonts, and images, controlling animations, and handling audio and simple text-to-speech playback. In early versions of RiTa, each RiText needed to be explicitly drawn to the screen by the user. In a

simulation-based environment like Processing however, where the draw() loop is called automatically every frame, the cognitive load for new users is decreased and code reads more clearly when objects "draw themselves". Thus, in later versions all RiText objects that were explicitly hidden (via calls to riText.setVisible(false)) are automatically rendered on screen at each frame. This default behavior is not only consistent with the Processing paradigm, it also appears to be generally intuitive for users with backgrounds in Flash, Max/MSP, or Silverlight. Although some experienced Java users (and one QuickTime-user) expressed initial confusion over this behavior, such confusion didn't appear to persist beyond one explanation. For more complex situations, where complex layering or a specific ordering of affine transformations were required, users could disable this behavior by calling RiText.disableAutoDraw() and then drawing each object manually in the order required.

RiSpeech

This object provides basic cross-platform text-to-speech facilities with control over a range of parameters including voice-selection, pitch, speed, rate, etc. RiSpeech is based on FreeTTS's[22] implementation of the Java Speech API or JSAPI[23] specification. Due to a range of extensions to the FreeTTS components, RiSpeech works online in an applet/browser context without requiring Java Web Start[24] technology or even signed applets. Further, RiSpeech is tightly integrated with the RiTa lexicon for phonemic and syllabification data, generating this data in real-time (via procedural rules) when is not available via the lexicon. When needed, multiple RiSpeech objects can be created, each with their own parameters, for

[22] See http://freetts.sourceforge.net/.

[23] See http://java.sun.com/products/java-media/speech/.

[24] See http://java.sun.com/javase/technologies/desktop/javawebstart/.

concurrent speech. The RiSpeech object is also compatible with the MBROLA[25] voice set with installation of a platform-specific binary. These (optional) voices provide higher quality but sacrifice cross-platform compatibility due to their use of native libraries. Additionally, on the Mac platform RiSpeech provides programmatic access to Apple's built-in high-quality speech synthesis library with over 20 voices.

RiLexicon

A user-customizable lexicon equipped with implementations of a variety of matching algorithms—min-edit-distance, soundex, anagrams, alliteration, rhymes, looks-like, etc.—based on combinations of letters, syllables, stresses, and phonemes. For each word entry, RiLexicon provides syllabification, stress, and pronunciation information (for TTS) following the conventions of the CMU Pronunciation Dictionary. Additionally, a set of Part-Of-Speech tags are provided for use in the (default) transformation-based POS-tagger.

Users can also modify or customize the lexicon (e.g., add words, or change pronunciations) by editing the plain-text "rita_addenda.txt" file, an example of which comes as part of the core RiTa download. An example is presented in .

```
# an example rita-addenda file
seven s-eh1-v ax-n | jj nn
eight ey1-t | jj nn
nine  f-ay1-n | jj nn
ten   t-ay1-n | jj nn
jumps jh-ah-m-p-s | vbz nns
```

Figure 4: An example rita-addenda file.

[25] See http://tcts.fpms.ac.be/synthesis/mbrola.html.

RiTokenizer

A simple tokenizer for word and sentence boundaries with regular expression support for custom-tokenizing. As with all RiTa tools, RiTokenizer used the Penn corpus conventions and rules as default behavior.

RiTextBehavior

An extensible set of text-behaviors including a variety of interpolation algorithms for moves, fades, color-changes, scaling, rotating, and morphing text. By implementing the RiTextBehavior interface, students can add their own simple text behaviors whose lifecycles (creation, frame-by-frame updates, clean-up) are managed transparently by the library.

RiStemmer

A simple stemmer for extracting base roots from words by removing prefixes and suffixes. For example, the words “run”, “runs”, “ran”, and “running” all have “run” as the root. Based on Martin Porter's stemming algorithm [van Rijsbergen et al. 1980].

RiPluralizer

A simple rule-based pluralizer for nouns. When passed a stemmed noun (see RiStemmer,) it will return the plural form. Uses a combination of letter-based pluralization rules and a lookup table of exceptions for irregular nouns, e.g., appendix → appendices.

RiSearcher

A utility object for obtaining real-time unigram, bigram, and weighted-bigram counts for words and phrases via online search engines, e.g., Google (described in further detail below).

RiWordNet

Installed as a separate component or used in conjunction with the rest of the toolkit, RiWordNet provides straightforward access to the WordNet ontology, supporting all the common WordNet relation types, including synonyms, antonyms, hypernyms, & hyponyms, holonyms, meronyms, coordinates, similars, nominalizations, verb-groups, derived-terms, glosses, "see-alsos", examples, and descriptions, as well as distance metrics between terms in the ontology. RiWordNet supports WordNet version 2.x-3.0x across all platforms and, at the time of this writing, is the only public library that provides direct access to WordNet via browser-based web applets, with no need for special downloads, memory-configuration, "Java Web Start", or applet-signing.

Additionally RiWordNet provides each term in the ontology with a unique ID for the combination of sense and part-of-speech, facilitating direct reference to the sense in question. All methods take simple String or ints and return String arrays, greatly simplifying the complex pointer hierarchy found in the original implementation. In most cases, three methods are provided for each relation type (e.g., for hyponyms, getHyponyms(int uniqueId), getHyponyms(String word, String pos) and getAllHyponyms(String word, String pos) where the first returns hyponyms for a specific sense (as specified by its unique id), the second returns the most common sense for the specified part-of-speech, and the third returns all senses for the word/part-of-speech pair.

Additionally several literary-specific extensions are provided. These include part-of-speech and random iterators,[26] which allow users to query for random words, glosses, and descriptions that match a specific part-of-speech, facilitating simple substitutions on existing phrases. Similarly, each call to getX()—where X is one of the relations listed above—returns

[26] See *non-deterministic iteration* in Chapter 3: Pedagogy.

an array of Strings in random order (this default can be disabled for unit-testing,) thus enabling users to continually explore the parameter space of a generative piece during the development cycle. Lastly, RiWordNet adds a range of specific literary relations to the standard set including “soundex”, “letterex”, anagrams, and others.

RiTextField

A simple text field widget to handle user keyboard input. When user input is completed, a RiTaEvent callback is triggered as described in the ‘Events and Dynamic Callbacks’ section below.

RiSample

Provides intuitive library-agnostic audio support, handling playback of wav, aiff, and mp3 samples and server-based streaming of compressed mp3s.

RiPosTagger

Provides a standard interface for implementations of part-of-speech taggers. The current version of RiTa includes two such implementations, both using the Penn conventions (see Table 3 below), a faster and lighter-weight transformation-based tagger based on an optimized version of the Brill algorithm [Brill 1992] and the generally more accurate maximum-entropy tagger based on the OpenNLP[27] package. The transformation-based tagger is closely tied to the lexicon provided with RiTa that the set of part-of-speech tags for each word entry. This allows lookups to run in constant time, after which a set of context-specific rules are applied to select the appropriate set element for the specific context in which the word appears. Words not found in the lexicon default to the most likely part-of-speech, a singular noun, and are then run through a similar set of transformational rules which, based

[27] See http://opennlp.sourceforge.net/.

on spelling, phonemic data, and context (the surrounding words), create a "best-guess" for the part of speech, again in constant-time, once the lexicon has been loaded. Table 3 contains the full set of tags (following the Penn conventions) returned by the RiPosTagger (regardless of implementation):

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

Table 3: Alphabetical list of part-of-speech tags used in the Penn Treebank project.

RiHtmlParser

Provides various utility functions for fetching and parsing text data from web pages using either the Document-Object-Model (DOM) or regular expressions. Also provides a base implementation so that subclasses can override the handleText(), handleSimpleTag(), handleStartTag(), and handleEndTag() methods to define custom behavior (as in RiSearcher). Examples of basic functionality include the fetching of HTML pages as plain text, with or without the HTML tags stripped, and the ability to define custom parsing behavior, as in the fetchLinks() and fetchLinkText() methods which respectively fetch all the anchor links on a page and all the linked text (contained within an anchor) on a page.

RiTravesty

Represents a Markov chain (or *n*-gram) model that treats each character as a separate token (as in Kenner and O'Rourke's original Travesty program[28]). Provides a range of methods to query the model for probabilities and/or phrase completions. As RiTravesty's functionality was, to a large extent, overlapped in the RiMarkov object (also *n*-grams but at the word level), this component was used largely as a teaching tool. One of the students' assignments was to implement the Travesty interface on their own, to demonstrate their conceptual grasp of *n*-grams without needing to deal with the complexities and edge-cases found in word-based models.

RiAnalyzer

The RiAnalyzer object allows users to easily access micro-level features, e.g., syllables, part-of-speech, phonemes, and stress features for arbitrary strings of text. Analysis

---

[28] Originally published in Kenner, H. and O'Rourke, J. 1984. BYTE Magazine. Volume 9, Issue 12 (November, 1984): New chips.

involves a combination of lookup and algorithmic rules. First, RiAnalyzer performs a lookup in RiTa's custom lexicon (~35,000 words) for this data. If not found, this data at runtime is created via procedural rules, after which it is cached. Part-of-speech data is also provided (see RiPosTagger). Once a text has been analyzed, it is "annotated" with features for each of the elements mentioned (see RiString below for a full description), which can then be used in the generation of larger features. This range of micro-level features provides users with the necessary infrastructure with which to address a range of specifically literary technique from rhyme, to alliteration, line-break and enjambment, puns and visible plays-on-words, rhythm and musicality.

RiString

The RiString object augments the basic Java String with support for "features", key-value pairs that contain annotations about the String in question. Features, implemented via String-to-String (lazy-instantiated Hashtables), allow an arbitrary number of annotations to be attached to a given String, with a default set provided by the system itself. Support for new user-defined "features" is enabled (and encouraged) by the design.

RiString additionally provides implementations for all the usual Java String methods (in fact the entire CharSequence interface), allowing methods to be invoked on a RiString object just as they would be on a String. RiString.split(), for example, splits a RiString object above into some number of new RiStrings (just as String.split() does), one for each word, maintaining the features that remain relevant (e.g., part-of-speech, stress, phonemes, syllables, etc.) to the whole, and deleting those that are not (e.g., "chunk-type", as the individual words were no longer part of a "noun-phrase").

This allows RiString objects to be passed along a pipeline (similar to Unix command-line tools) and lookups to be performed in *lazy* fashion, if and as needed. For example, if a

core object (RiGrammar for example) needs to know the part-of-speech for a RiString from which it is generating a phrase, it might ask whether the "pos" feature is present via a call to RiString.hasFeature("pos"). If not, it might invoke the RiPosTagger object to first add the part-of-speech feature before proceeding with generation. Objects in different states of analysis can thus be exchanged with features added only when needed, thus enabling, through a sort of ad-hoc introspection mechanism, a weak version of polymorphism (the lack of which being one critique of such a non-hierarchical approach). In fact it was this feature that later enabled the use of RiTa processing objects in a real-time drag/drop environment for language processing (see the RiTa visual interface in the *Future Work* section of chapter 6).

RiKWICker

RiKWICker provides an efficient implementation of a KWIC-model generator. KWIC is an acronym for "Key-Words-in-Context", a common format for sentence-based concordances.[29] A KWIC model may also be referred to as a permuted index, referring to the fact that the model contains all cyclic permutations of each sentence in a text. The RiKWICker implements this model by sorting and aligning all the words within a text so that each word provides a hash key for a list of all sentences that contain that word. Thus, one can retrieve the sentences containing a word in constant time, O(1). The time to create the model is O(*n*) where *n* represents the number of words in the text. Since each word must be viewed at least once when creating the model, this performance is (asymptotically) optimal. Additionally, RiKWICker provides options to ignore *stop-words* and letter-case, each potentially decreasing the memory requirements of the model for longer texts.

[29] The term "Key Words In Context" (KWIC) was first coined by Hans Peter Luhn as described in Manning [1999].

RiMarkov

RiMarkov provides an implementation of a Markov chain (or *n*-gram) based generator with specific extensions for literary output. A language model is a statistical model used to analyze or generate elements of a text or texts via probability. *N*-gram models (or Markov chains) are statistical models in which the next item in a sequence is predicted based upon the frequency of that sequence in a set of input texts. The "*n*" in *n*-grams refers to the number of words in each sequence that is considered in our estimate. If $n = 1$ we have a *unigram* model in which the probabilities of a single letter, for example, are estimated based only on the frequency of that letter in the input. In a bigram model, where $n = 2$, we would estimate the likelihood of a two-letter sequence, "Qu" for example, based on its frequency in the input compared to all other two letter sequences in the input.[30] RiMarkov provides this functionality (by default) for words, though alternative tokenization strategies are supported via regular expressions.

Additionally RiMarkov provides a range of extensions specific to natural language and literature, including the recognition of abbreviations, elisions, and sentence boundaries to better facilitate sentence generation. Other such features included the weighting of inputs (e.g., for combining texts of different lengths), constraints on repetition, custom tokenization, feature compression (letter-case, synonyms, etc.), and the following methods: getCompletions(), getProbabilities(), and getProbabilityMap(), each of which allow for some degree of interactive control (or steering) of the model during generation. Additionally RiMarkov supports dual-mode operation in which each word is augmented by part-of-speech

[30] *N*-gram models were formalized by Claude Shannon in "A Mathematical Theory of Communication" [Shannon, 1949] and are a specific instance of the more general class of Markov chains.

information that can later be used to constrain the output to grammatical patterns based on Part-of-speech.

As noted by Wardrip-Fruin [2006], *n*-grams are a remarkably effective technique, given both the simplicity of the technique and its lack of specificity to the domain of language. One could imagine models substantially *more* specific to language, and in fact there has been a growing call for researchers to "put language back into language modeling" [Rosenfeld 2000]. This would require more complex models, like RiMarkov's dual-mode operation mentioned above, that work with features other than small groupings of undifferentiated word tokens. Researchers have experimented with approaches such as decision trees and link-grammars, but the majority of such attention has focused on "maximum-entropy" modeling [Berger et al. 1996]. Maximum-entropy is another statistical technique, like Markov chains, that has seen use in a range of contexts, but its flexibility allows for the selection of a wide range of features, including those specific to language and literature. The RiChunker and RiParser, described below, make use of maximum-entropy modeling techniques.[31]

RiChunker

Based closely on the OpenNLP implementation, this object provides a simple implementation of a maximum-entropy chunker for finding non-recursive syntactic "chunks" such as noun-phrases, using the Penn conventions. Table 4 present a list of phrase-tags used according to the Penn Treebank conventions.

---

[31] For more information, see "A Maximum Entropy Approach to Natural Language Processing", [Berger and Della Pietra 1996] which provides a good introduction to maximum entropy techniques.

Name	Description
adjp	adjective phrase
advp	adverb phrase
conjp	conjunction phrase
intj	interjection
lst	list marker
np	noun phrase
pp	prepositional phrase
prt	particle
sbar	clause introduced by a subordinating conjunction
ucp	unlike coordinated phrase
vp	verb phrase
o	independent phrase

Table 4: Alphabetical list of phrase-tags used in the Penn Treebank Project.

RiParser

Based closely on the OpenNLP implementation, this object provides an implementation of a maximum-entropy parser for finding recursive (or nested) syntactic "chunks" such as noun-phrases, using the Penn conventions as listed in Table 4.

RiGrammar

RiGrammar provides an implementation of a context-free grammar with specific extensions for generating literary texts. The implementation allows users to specify the rules and productions for a grammar in a local or remote plain-text file. A simple example grammar, following the RiTa conventions, is presented below.

```
# An Example CFG

#####################################
# s -> np vp
# np -> det n
# vp -> v | v np
# det -> 'a' | 'the'
# n -> 'woman' | 'man'
# v -> 'shoots'
#####################################

{
 <start>
 <np> <vp>
}

{
 <np>
 <det> <n>
}

{
 <vp>
 <v> | <v> <np>
}

{
 <det>
 a | the
}

{
 <n>
 woman | man
}

{
 <v>
 shoots
}
```

Figure 5: A simple RiTa grammar file.

While grammars are often used in natural language analysis, RiGrammar, like most RiTa objects, is implemented (and optimized) specifically for generation, and contains a range of features specific to the literary context. For one, all rules are dynamic; that is, they

can be interrogated and modified at runtime. This is particularly important when one wants to change elements of the generation process based on what has been generated thus far. Similarly, RiGrammar supports probabilistic rules in which probabilities can be modified dynamically at runtime.

In addition, specific extensions for generation have been implemented at the method level. A simple example of such a method is *expandFrom*. While the expand() method simply performs generation (or expansion) from the "<start>" symbol, *expandFrom(String from)* begins with the symbol contained in the "from" argument (which can consist of either terminals, non-terminals or both), and performs an expansion starting from there, so that partial expansions can be triggered without adjusting the grammar itself. Another example, perhaps the most specific to literature, is *expandWith* which takes two String arguments. During generation the first argument, a terminal, is guaranteed to be substituted for the second, a non-terminal. Once this substitution is made, the algorithm then works backwards (up the tree representing the grammar from the leaf) ensuring that the first argument, the terminal, will appear in the output string. For example, with the grammar fragment above, one might call:

```
grammar.expandWith("woman", "<n>");
```

assuring not only that the <n> rule will be used at least once in the expansion process, but that when it is, it will be replaced by the terminal "woman". Further, *expandWith* can be used with terminals that are not present in the grammar. So the following would also work:

```
grammar.expandWith("child", "<n>");
```

though the String "child" is not present in the grammar. This algorithm enables generation that can adjust to the current state of the applications. For example, if we want to generate a new sentence linking to one previously generated, we might pass one or more keywords from that sentence to *expandWith,* guaranteeing that the pair of generations will be linked, at least

minimally, by that keyword. Similarly, when accepting input from a user, a program can generate phrases based on that input. This is particularly useful in conversational applications or for those with interactive characters.

To further support these dynamic generation modes, callbacks, from the grammar into user code, are supported. To generate such a callback, users include method calls within their grammar, surrounded (by default) with back-ticks. Three examples of this type of callback are presented in the rule below.

```
<rule2>
{
      The cat ran after the `lookupNoun();`  |
      The cat ran after the `getRhyme("cat");` |
      The cat ran after the pack of `getPlural(<noun>);`
}
```

The first line provides a simple way of embedding external data sources within a grammar, while the second and third lines provide functionality that is dynamically applicable to the generation context (all three represent functionality not available in strictly context-free grammars). Any number of arguments may be passed in a callback, but for each call, there must be a corresponding method (with the same number and type of arguments) in the user's program, e.g.,

```
String getPlural (String noun) {
    return myRiPluralizer.pluralize(noun);
}
```

An additional tool provided with RiTa, and facilitated by RiGrammar's dynamic grammar rules (as described above), is an interactive application, which allows users to experiment with one or more grammars in real-time. One panel of the RiGrammarView

application provides the current grammar file definitions in an editable window. The other panel allows user to generate from that grammar and store the results. Thus the typical process of experimentation is greatly simplified. Rather than stop the program, open the grammar file in a text-editor, make changes, save, then re-open the program, all of this happens within the live environment provided by the tool.

2.4 Documentation

RiTa is accompanied by extensive documentation that explains the toolkit and describes how to use and extend it. This documentation is divided into four primary categories:

- *Examples* are provided for each of the core RiTa objects, clearly illustrating basic uses. Examples are posted as both downloadable and web-executable programs on the RiTa web-site and accompanied by links to carefully commented source code that explains the purpose of each line. Example assignments are also provided to assist teachers with course planning.
- *Reference Documentation* provides precise definitions for each interface, class, method, function, and variable in the toolkit. It is automatically generated in two formats for each version via custom comments embedded in the source code: a simplified single-page "procedurally-oriented" HTML reference[32] and a standard Java-style object reference[33].
- *Tutorials* teach students to use the toolkit incrementally by focusing on a single task, e.g., tagging, generation, or classification. The tutorials include a high-level

[32] See http://www.rednoise.org/rita/documentation/docs.htm.

[33] See http://www.rednoise.org/rita/javadocs/.

discussion that explains and motivates the domain, followed by a code-level walkthrough showing how RiTa would be used to perform the task in question.

- *The Project Gallery* provides students with a wide range of existing projects (implemented in RiTa) by other students and artists, all with linked source code. Students can access this archive either for inspiration on projects or for assistance in addressing particular issues. Most of the projects also include direct contact links for the authors that can be used if more specific questions are required. These projects also demonstrate proper documentation strategies, a particularly important element for those working with rapidly evolving technologies. Finally, students may, with instructor approval, add their own projects to the gallery, a goal that inspired some students and helped others to feel part of a community of practicing artists.

2.5 Using Rita

As consistency was one of our primary design criteria, all RiTa objects follow the same basic pattern for object instantiation, as follows:

```
RiObject objectName = new RiObject();
```

This syntax employs the conventional, if less than perfect, "new" operator[34] and represents the most generic mode of object creation in Java . Although restricting object creation to "new" significantly complicates the coding of some objects (see the section on the RiTaServer below), it was judged to be more intuitive for new users and does not require an

[34] See Jonathan Amsterdam's article on the topic at http://www.ddj.com/java/184405016;jsessionid=5GYK2EDVKVLMUQSNDLPSKHSCJUNN2JVN?_requestid=205339.

understanding of the class/object distinctions, nor of static methods (e.g., main() or createX()).

When used in Processing, object creation was performed like this:

```
RiObject objName = new RiObject(this);
```

To clarify this syntax, Processing "sketches" are, by default, Java applets that subclass the processing.core.PApplet class (which in turn subclasses java.applet.Applet) so that the "this" keyword passed to the constructor represents an instance of PApplet and provides the RiTa object with a reference back to the core Processing methods implemented within. This is the recommended syntax for Processing libraries and has been somewhat universally adopted by library developers. This back-reference to the PApplet is necessary in only a few cases within RiTa, the RiText object being a primary example, for which Processing is required, at least for now, to perform the supported 2D and 3D drawing functions. As all other core classes can be used with or without Processing, either of the syntaxes above is acceptable.

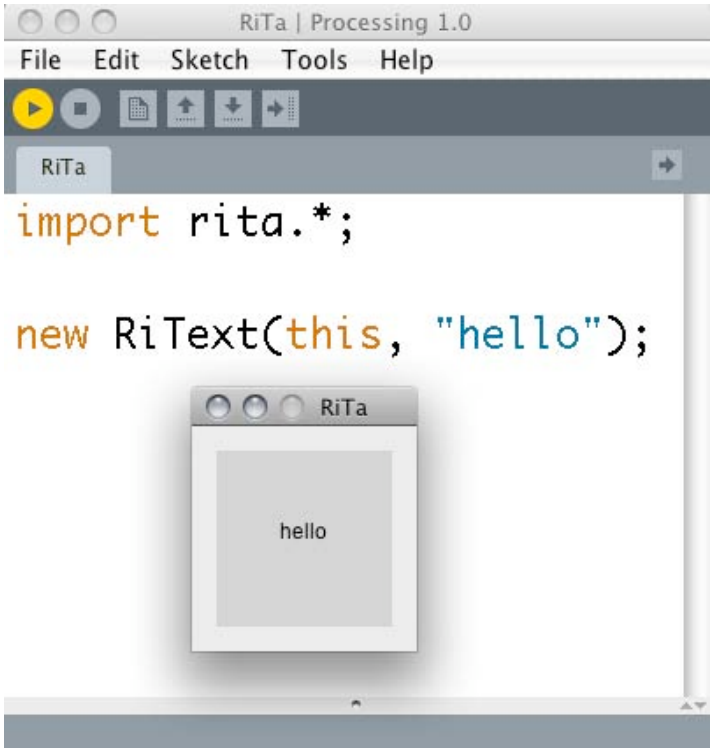

Figure 6: A simple RiTa sketch in Processing.

Typically however, the RiText object is generally the first object encountered by students as they often wish to immediately display some piece of text on which they are working. In fact, only one additional line of code is required to add visible text to an existing sketch. To illustrate, a very basic RiTa sketch is shown in Figure 6.

Figure 6 also illustrates the simplicity of the basic Processing environment. Creating this "sketch"—to use Processing terminology—required the user to a) select RiTa from the "tools" menu to generate the import line at top, and b) to type or paste in the single line shown that creates a single RiText object. To run the program, the yellow "play" button is pressed. For new or inexperienced students, this provides a significant usability gain over the complexity of a similar first program in raw Java (see Figure 7).

```
import java.awt.Graphics;
import java.applet.Applet;

public class Hello extends Applet {

  public void init() {
      resize(200, 200);
  }

  public void paint(Graphics g) {
      g.drawString("Hello", 80, 80);
  }
}
```

Figure 7: A minimal Java applet.

Notice the number of concepts present in this program, from classes, to functions with and without parameters, to inheritance via the “extends” keyword. Each of these concepts must either be learned by a new user, or ignored (as is unfortunately more often the case). Further, this program still requires several steps before any output can be seen, beginning with the separate compilation step that is required in Java (which generally requires knowledge of where the “javac” compiler is located, as well as the correct path for the Java runtime classes to be set in the CLASSPATH variable). Running the program is also not trivial, as Sun's “AppletViewer” tool requires an additional HTML document to be created and saved before one can even test the code. An example version of such a file is shown in Figure 8.

```
<HTML>
<HEAD>
<TITLE> A first program </TITLE>
```

```
</HEAD>
<BODY>
<APPLET CODE="Hello.class" WIDTH=200 HEIGHT=200>
</APPLET>
</BODY>
</HTML>
```

Figure 8: A minimal HTML page as required for Java's AppletViewer.

The URL for this file must then be passed to the AppletViewer (via the command-line) before the program can be launched. While this process has been somewhat simplified in some integrated development environments (or IDEs), e.g., Eclipse or NetBeans, installing and configuring these IDEs has been notoriously difficult for new users who are easily overwhelmed by the sheer number of menus, windows, and options present in the interface. As one student experienced with Java commented,

> RiTa is a great and really broad library that I used throughout the semester; coming from experience with Swing and the Java Applet library, I was happily impressed with the simplicity of RiTa, and of Processing libraries in general. To be able to spend time on the algorithmic, structural, and creative parts of projects rather than tedious GUI creation is a satisfying and liberating thing. [PDAL Student, 2009]

2.5.1 Animation and 'text behaviors'

Animation support in RiTa is provided by a combination of simple method calls (in the RiText object), and by the more flexible, but more complex, *TextBehavior* framework. The basic methods in RiText, with a variety of convenient overloads, include the following:

```
riText.moveTo(x, y, startOffset, duration);
riText.moveBy(xOff, yOff, startOffset, duration);
riText.fadeOut(startOffset, duration);
```

```
riText.fadeIn(startOffset, duration);
riText.fadeColor(newColor, startOffset, duration);
riText.fadeToText(newText, startOffset, duration);
riText.scaleTo(newTextScale, startOffset, duration);
```

These methods, all of which were implemented using the *TextBehavior* architecture, were included to allow users easy access to the basic functionality required for simple display and manipulation of text. Further, users who were familiar with working via timelines (e.g., in Director or Flash) could use the optional "startOffset" parameter to "script" an entire text sequence, starting and stopping various behaviors for each RiText object at specific times offset from the start of the program. For finer-grained and/or more interactive control, RiTa employs a callback mechanism (see Events/Dynamic Callbacks below) through which programmers can be notified at various stages of a TextBehavior's execution. To provide the types of motion and interpolation required for "natural-looking" motion, 13 types of interpolation (or easing) were implemented: LINEAR, EASE_IN, EASE_OUT, EASE_IN_OUT, EASE_IN_OUT_CUBIC, EASE_IN_CUBIC, EASE_OUT_CUBIC, EASE_IN_OUT_QUARTIC, EASE_IN_QUARTIC, EASE_OUT_QUARTIC, EASE_IN_OUT_SINE, EASE_IN_SINE, EASE_OUT_SINE, any of which can be enabled at the object-level by a call to riText.setMotionType(type).

Lastly, new behaviors could be created via the TextBehavior mechanism and then applied to instances of RiText. An example of this was the scaleTo() method, which was originally implemented in a student project as a TextBehavior (to gradually grow or shrink at text—via affine transforms—to a specific size), and was later refactored and included as part of the core library.

2.5.2 Events and Dynamic Callbacks

When creating an interactive program, it is often useful to know when certain events have started or finished. For example, one might want to move a RiText object to a particular location on screen, then, upon its arrival in the desired location, have its text contents spoken via a text-to-speech (or TTS) voice. To address this need, RiTa provided a "dynamic callback" mechanism that allowed users to handle such events in an intuitive fashion. The mechanism is "dynamic" because the programmer does not need to register interest in events, nor implement any interfaces (this is disallowed in the typical Processing sketch). Instead, because of the PApplet reference passed to the constructor of each RiObject, objects can attempt to dynamically invoke such methods back to their PApplet "parent". If the method is not found on the first attempt to locate it (via Java's reflection API) in the parent, then it is not called again. If however, it does exist, then all subsequent events are passed in calls to that method. In early versions, each event type had its own callback signature, so that animation would trigger a method like onBehaviorCompleted(), while a speech events might trigger onSpeechCompleted(), and user text input (see the RiTextField method) would trigger onTextInput().

In the current implementation, this mechanism has been made more generic. Each of these events are of now of the same type, represented by the RiTaEvent class, and trigger the same method, onRiTaEvent(), inside which the type of the event can be queried. From anecdotal feedback, this solution has been preferable, as a single conditional or switch statement can be used rather than implementing new callback methods whenever new functionality is added. A typical implementation of the generic callback (for several object/event types) might look like this:

```
void onRiTaEvent(RiTaEvent re) {
      switch (re.getType)
      {
            case RiTa.SPEECH_COMPLETED:
                  …
            case RiTa.MOVE_COMPLETED:
                  …
            case RiTa.TEXT_ENTERED:
                  // print the entered text
                  println(re.getData());
                  …
      }
}
```

In most cases, however, as users are interested in one, or at most two such callbacks, such awkwardly long conditional blocks are not necessary.

2.5.3 Parameter and Return Types

> *"Text needs to be a first-class object in these systems, and its representation is the most crucial part of the design." – Pablo Gervas*[35]

Early iterations of the RiTa toolkit implemented a hierarchical approach to text that restricted the types that could be passed to and returned by RiTa functions. Though such a hierarchical design, with Phrase objects containing Word objects, containing Syllable objects, containing Phoneme objects, had been used in several natural language libraries (Jet, FreeTTS, etc,), we eventually concluded that a simple string-based approach would offer a

[35] In conversation with Pablo Gervas at 'The Electronic Literature Organization 2008' conference held 29 May-1 June 2008, in Vancouver, Washington, USA

range of advantages for this context. In fact, all of RiTa's functionality was eventually made accessible via primitives, strings, and string-arrays, a feature that greatly reduced the learning curve for new users. In addition to presenting a clean and consistent interface, the choice of arrays, rather than Vectors or Lists (to use Java terminology), shielded users from potentially confusing elements of the Java collections package, e.g., repeated castings, differing behavior of primitives and objects, the (belated) support for generics, and the rather convoluted mechanisms for converting back and forth between collections and arrays. Additionally knowledge of data structures (beyond the simple array) was not necessary.

This did create, however, some additional design complexity in those cases where the processing of hierarchically-oriented data was necessary. RiWordNet, for example, which provides access to the Princeton's WordNet lexical ontology is a case in point. The WordNet dictionary[36] is comprised of a deeply nested tree-like structure of "pointers". Typically, libraries for WordNet have returned these pointers directly, allowing users to manually navigate the range of objects – from senses and lemmas, to synsets and glosses, to examples and descriptions. RiWordNet was able to avoid such a hierarchical approach by augmenting the WordNet database with unique identifiers (generated at runtime) for each accessed lemma, or lexical entry. In raw form, WordNet only supplies pointers to sets of synonyms, or synsets, so navigating to a specific word (a combination of its sense and part-of-speech) was not possible without repeated navigation of pointer hierarchies. With RiTa's id mechanism however, users could access a word of interest multiple times, by repeatable passing of its unique identifier to whatever methods were needed. Thus each method call could return a smaller set of data and users were not required to manipulate or understand object types beyond simple Strings and String arrays.

---

[36] See http://wordnet.princeton.edu/.

2.5.4 Lazy Instantiation and Caching

To meet the design constraints on memory, browser-based execution, and performance, all object and data-loading in RiTa is performed (by default) in lazy fashion, if and as needed. Thus, the lexicon is not loaded (unless specified to the contrary) until a query is performed on it. Similarly, stemming, pluralization, conjugation, and letter-to-sound rules are only loaded when required for a specific method to complete. This allows RiTa to support a relatively large range of functionality with only those users who needed specific resources having to pay the cost for loading and using them. Similarly, most RiTa objects instantiate the RiCacheable interface and can optionally cache the results for a range of functional calls (often referred to as *memoization*).

An example of this functionality is the RiSearcher object, which computes a range of statistical metrics via calls to the Google search engine. Not only are such calls time-intensive, but they can accumulate and repeat, especially when one is testing new code. One such call allows users to obtain the "weighted bigram coherence" for a given sentence. As an example, we can consider a sentence *S* with *n* words {W1,W2...Wn}. Bigram coherence refers to the frequency that a pair of words (Wk,Wj) will occur sequentially in a given set of input texts (in this case, the entire web as indexed by the Google search engine), and is defined in RiTa as the number of occurrences (or frequency) for the quoted query "Wk Wj" weighted against the sum of the queries for each individual word:

$$bigram\text{-}coherence(Wk, Wj) = count(Wk\ ,Wj) \,/\, ((count(Wk) + count(Wj))$$

To obtain the bigram coherence for all *n*-1 pairs of consecutive words, (W1 + W2, W2 + W3... Wn-1 + Wn), we measure each pair individually, then weight the average of these against the length of the sentence, represented by *n*:

avg-bigram-coherence(W1...Wn) =

(Sum(k), for k = 1...n-1: bigram-coherence(Wk, Wk+1)) / n

Obtaining the average bigram coherence for the entire sentence thus requires two "single" calls to the search engine, for (count(Wk) and count(Wk+1), and one paired call (the "and" of the two words), but—and here the problem shows itself—half of these single calls will be duplicates of previous calls. In such cases, using a cache significantly increases the efficiency of the algorithm (especially as Google has recently taken to blocking repeated calls from the same IP address when not made not through the proprietary Google API). To make matters worse, it is common during development to make repeated calls for the same sentence or paragraph, thereby increasing the number of duplicate calls. Here, allowing the cache to persist across multiple runs of the program (as when using the RiTaServer described in the following section) provides yet further gains in efficiency.

2.5.6 Mixed-Mode Operation

The RiTaServer mechanism provides an alternative client/server-based mode for RiTa objects that may have expensive initialization routines (e.g. building a large *n*-gram model from text files) or that may benefit from persistent caching (e.g., the RiSearcher object described above). Rather than incurring these costs each time the program is run (in development, for example, this may happen hundreds or even thousands of times), the server enables data stores to remain in memory while the program is stopped and started any number of times. This mechanism was added relatively late in the development of the toolkit, only

after it became clear (see the section on *Design Tensions*) that certain use cases would continue to warrant the added complexity. There was also significant experimentation and iteration done to minimize the impact for users switching between these operation modes. In the current implementation, once the RiTaServer has been started, the user is required only to add one additional line of code to their program to switch processing to the remote server's virtual machine. All the same methods are supported remotely as are locally, and all local and remote object creation, marshaling of parameters and return types, and network communications, are handled transparently. The example below shows how a program using the RiMarkov object can switch the execution environment to the remote server by adding this one line of code:

```
RiTa.useServer();  // one additional line
RiMarkov rm = new RiMarkov(this, 3);
rm.loadFile("war_peace.txt");
String[] sents = generateSentences(10);
for (int i = 0; i < sents.length; i++)
 display(sents[i]);
```

Beneath the surface, this additional line tells the RiMarkov object, which contains a reference to a MarkovModelIF interface, what concrete implementation (or delegate) to create and pass method calls to. In the remote case, instead of creating a concrete rita.support.MarkovModel object locally and delegating calls to it, a remote proxy object is created. On creation, the remote proxy sends a message to the RiTaServer to create the concrete MarkovModel in the remote virtual machine. An additional message may be sent containing mappings to the location of any external resources (e.g., “war_peace.txt” in the example above) that are required for the remote object's operation. Subsequently, all method

calls to the RiMarkov object are transparently routed to the remote proxy, converted into remote messages, with parameters marshalled appropriately, and executed in the server. The server then handles un-marshalling of parameters and dispatch of the method call to the local object. Similarly, when the method call on the server object asynchronously completes, the returned results are marshalled and sent back over the network to the local RiMarkov's remote proxy object, which then returns them to the user code.

Here we see another design tension, specifically between the remote proxy design pattern [Gamma et al. 1995], and the manner in which objects are created in RiTa (see object-creation above). In the typical implementation of this design pattern, the RiMarkov object would be an instance of an interface (or abstract class) and its concrete implementation would be created via a factory method as follows:

```
RiMarkovIF rmi = RiMarkovFactory.create
      (this, serverEnabled); // as an interface
```

or

```
RiMarkov rmi = RiMarkov.create
      (this, serverEnabled); // as an abstract class
```

Thus, the factory-style's static “create” method could then instantiate the appropriate delegate based on the “serverEnabled” flag. In the RiTa context however, the cognitive overhead of this pattern was judged, for a number of reasons, to be too high to justify. First, creation of RiTa objects via this pattern would break the consistency of the “new” paradigm for object creation, which prevails throughout Processing and its libraries. Secondly, users would need to understand (or ignore) the distinction between static and object methods, as well as why (in the interface case) multiple objects types would be required to create a single new object. The

abstract class case is perhaps more intuitive, but then rules out traditional inheritance for the core set of RiTa objects, each of which already extends RiObject. More importantly, creating RiTa objects would look significantly different than creating basic Java and/or Processing objects, adding an undesirable degree of complexity for new users. Thus the somewhat complex internal mechanism described above was implemented, breaking with typical design patterns, in order to preserve a simple and consistent interface for object creation while still facilitating the simplest mechanism for switching between "normal" and "client/server" modes.

While the RitaServer mechanism supports servers on remote computers (e.g. for access by "live" applets), this has not proven to be the typical use case. Perhaps unique to the context of computational art,[37] and in accord with our design constraint of rapid (or *micro*) iteration, more common is the use of this mechanism on a single local machine; e.g., both client and server running on a user's laptop. It is essential in development, at least in the arts context, to be able to rapidly iterate, trying and quickly abandoning new ideas, moving simultaneously in a number of development directions.

Let us imagine that it took thirty seconds to load and construct a very large *n*-gram model from a set of texts. This wait, occurring with each run of the program, would dramatically reduce the productivity of the artist-programmer in terms of iterations, and more significantly would diminish her ability to quickly develop new ideas. By running the RiTaServer locally however, this load time is paid for only once (unless new texts are swapped in) while the additional overhead for method calls, due to marshalling and local network latency, is generally minimal enough to be unnoticeable. Because the concrete server object (rita.support.MarkovModel in our example) implements the same interface as the local

[37] See the chapter 6 on creativity support engineering principles for the arts.

object, there is no need, and thus no performance penalty to paid, for method calls via reflection.

Of course, in our example, there is still the question of whether an end-user (someone viewing the applet as an online art piece) would be willing to wait the thirty seconds for the texts to load. Because this is a one-time wait and there is a perceived reward (the motivation to actually view the piece), the answer here is generally yes. In cases where such a wait is judged to be unacceptable, or in the scenario where the memory requirements for the model exceed those of the web browser itself, the RiTaServer can be run on the physically remote web server machine. In this scenario, model loading and memory constraints disappear for the user and are paid only once for the life of the application. Of course this scenario can require more permissions on the web server than are available to some RiTa users (especially inexperienced students), and is thus only recommended when fully necessary.

2.5.7 The EclipseP5Exporter

One of the most powerful features of the Processing environment, for experienced and inexperienced programmers alike, is its one-click "export" feature. After completing and testing a sketch within the Processing environment, a single button push will export the sketch either as a basic web-applet, a signed web-applet with native libraries (such as those necessary for OpenGL[38]) and/or a separate stand-alone application for each of the major platforms (Windows, Mac, and Linux). All required resources, including HTML pages, links to source code, libraries, and jars are generated and archived quickly and transparently. To publish a sketch as a publicly accessible applet, one need only copy the exported applet directory to a server via an FTP program (nearly all students nowadays have web server space

[38] See http://www.opengl.org/.

provided as part of their school accounts). As ease-of-use, share-ability, and transparency (e.g., visible source code) were among our primary design constraints, this was an essential feature.

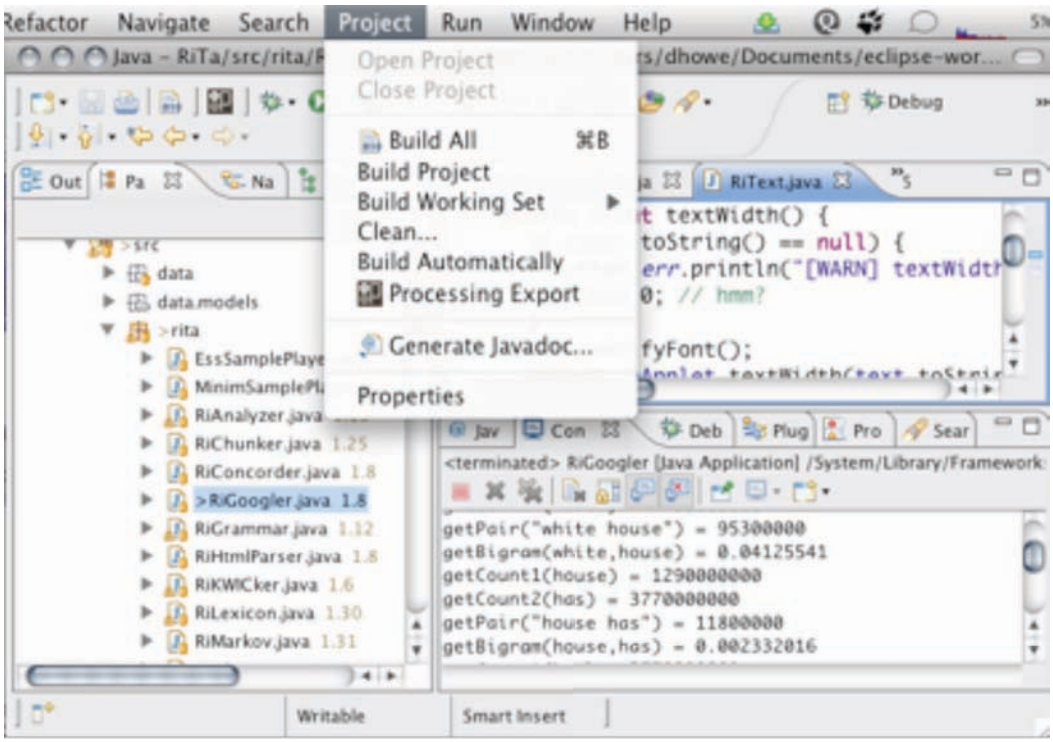

Figure 9: Screenshot of the RiTa-Eclipse plugin interface.

Yet the Processing environment, while ideal for beginners, was quickly outgrown by students as they advanced. In fact, by mid-semester there was often a vocal group of students asking for assistance in switching to the Eclipse IDE, a step-by-step tutorial for which was added to the course wiki. The problem, however, is that Eclipse provides no similar mechanism for export, not even for simple applications on one's own platform. This led to our development of a new RiTa-Eclipse plugin called the EclipseP5Exporter, which eventually became an important tool for the Processing community, whether or not it was used in conjunction with the other RiTa tools. The EclipseP5Exporter was designed and

implemented as a native Eclipse plugin that provides the same 'export' functionality as found in Processing. Built via the SWT toolkit, it integrates directly into the Eclipse IDE widget set and provides all the functionality listed above. Figure 9 (above) shows a screenshot of the exporter, visible in the Eclipse interface as both a toolbar button and as a menu item.

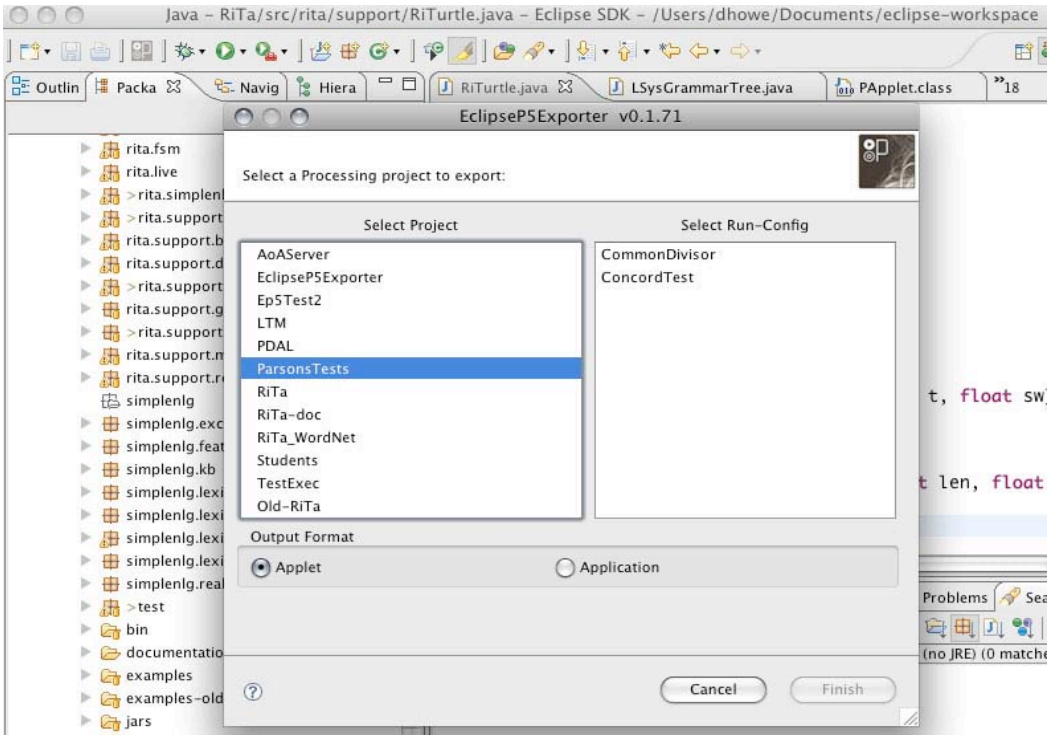

Figure 10: Screenshot of the RiTa-Eclipse plugin configuration widget.

Figure 10 shows a secondary screen in which users can choose the project and type of export to generate. When the export completes, a new directory is created containing the exported resources, which can then be launched via a simple double-click.

2.6 Conclusion

> *"What is really needed is social change: new play systems, new interaction models, expressive programming, and new role models in the field." [Perlin et al. 2003]*

In this chapter we have discussed the range of design considerations involved in making RiTa a functional and productive toolkit for practicing writers, and particularly as an end-to-end solution for computational writing courses. As a creativity support tool, RiTa includes a number of functions relating specifically to natural-language processing, carefully constructed to give students and artists at all levels a high degree of flexibility without sacrificing learnability or performance. Our goals dictated that the RiTa framework, wherever possible, be simple enough to be understood by entry-level students, and powerful enough to be adapted by those with more developed computer science skills. Though the RiTa toolkit is still in development, our initial experiences in the classroom indicate that, for the most part, these core goals have been met. In the next chapter, we will discuss the accompanying pedagogical framework with which we brought RiTa to the classroom, and then, in the final chapter, a range of evaluation metrics by which we have judged our success.

CHAPTER 3: PEDAGOGY

3.1 Introduction

> *[A]dmission and retention rates to computer science university courses are falling, enrollment is male dominated and although there is a thriving community of end-user programmers, there are serious concerns about the dependability of the software which they produce. Thus there is a need both to foster the development of computational thinking in young learners and to motivate them to study computing subjects by improving the perception of computing, especially for girls. [Romero et al. 2007]*

This chapter discusses educational applications of the RiTa tools. The bulk of the observations presented in this sections are based on our experiences in a series of courses taught over 4 semesters at Brown University[39]. In addition to providing a test environment for the tools in question, the goal of these courses was to enable students to create personally meaningful works of art and literature via computational methods while simultaneously developing the fundamentals of programming and computer science. To accomplish this dual goal we attempted to create a learning environment in which students' creative efforts to construct publicly viewable works in digital media doubled as programming exercises teaching basic procedural literacy. In close conjunction with the RiTa toolkit, a pedagogical approach was developed which we hoped would better leverage the affordances[i] of these new tools to accomplish our objectives.

In the sections that follow, we discuss a range of related issues; from previous work in the area, to guiding educational theories, to specific pedagogical decisions, to the

---

[39] See the "Programming for Digital Art and Literature" course website: http://www.rednoise.org/pdal/ [Howe 2009].

relationship between tools and teaching strategies. From these various perspectives we attempt to make the larger case that arts-focused tools provide a viable platform not only for teaching computationally-augmented art and writing practice, but also for teaching basic computer science concepts and advancing the goals of procedural literacy. In addition to the mechanics of the courses[40], we present some of the difficulties we experienced, as well as some recurring themes that arose as students engaged with these tools to implement their own creative works. By reflecting critically on the experience of teaching with RiTa it is hoped that we can further develop strategies for teaching core computer science skills, computer science education, and specifically, for computationally-augmented art and literary practice.

This chapter is divided into the following sections: Procedural Literacy and Computational Thinking; Constructivism, Constructionism and generative pedagogy; Educational software environments and related initiatives; Comparison of the PDAL course with other classroom tools/environments; and context-specific design considerations.

3.2 Procedural Literacy and Computational Thinking

> *Computer literacy for all citizens will be imperative for the United States to maintain a diverse, internationally competitive, and globally engaged workforce of scientists, engineers, and well-prepared citizens. This literacy must include computer programming and computer science fundamentals and involve both reading (using existing computer applications) and writing (making one's own applications).* *[Plass et al. 2007].*

[40] The course, taught between 2007-09, was originally entitled "Electronic Writing" before being renamed, for the subsequent three iterations, as "Programming for Digital Art and Literature".

Computer science researchers and educators have continually recognized the need to motivate a larger section of the population to understand and engage with core computational ideas. This research direction, pursued under various names, from Procedural Literacy[41], to Computational Thinking [Wing 2006], stresses the utility in procedural understanding for individuals in a range of professional, cultural, political, and educational contexts in digitally-mediated society. Similarly, a number of researchers [Greenberger 1962; Sheil 1980; Mateas 2005; Bogost 2005; Plass et al. 2007] have argued that engagement with programmatic concepts betters the lives of individuals in these contexts and furthers larger societal goals in a number of ways:

- By teaching general problem-solving techniques, applicable to a wide range of pursuits and disciplines;
- By enabling people to better communicate with others and express themselves in a digitally-mediated culture;
- By facilitating a better understanding of the ways in which technological choices made by a society influence its social, political, and cultural values;
- By including a larger, and more diverse segment of the population as 'decision-makers' on technologically-inflected issues.

[41] According to Guzdial [Guzdial and Soloway 2003], as reported in Mateas [2005], the earliest argument for universal procedural literacy is one given by A.J. Perlis in a symposium held in 1961 to celebrate the 100th anniversary of M.I.T., and published in the collection Management and the Computer of the Future by Greenberger [1962].

Forte and Guzdial [2004] characterize the problem as follows:

> As the skills that constitute literacy evolve to accommodate digital media, computer science education finds itself in a sorry state. While students are more in need of computational skills than ever, computer science suffers dramatically low retention rates and a declining percentage of women and minorities. Studies of the problem point to the overemphasis in computer science classes on abstraction over application, technical details instead of usability, and the stereotypical view of programmers as loners lacking creativity.

A number of reasons have been cited for the lack of progress in this respect: inappropriate tools and learning environments, lack of motivation for new learners (no support for the creativity-motivated), unsound pedagogy, lack of role models and poor evaluation of existing methods. While there seems to have been a recent upsurge in interest in the notion of widespread procedural literacy, it is by no means a new idea, but rather builds on a long tradition in the field. While this long lineage has been explored in detail elsewhere [Mateas et al. 2003], it is worth briefly noting what was likely its first mention within CS, from A.J. Perlis.

In his talk at a symposium held in 1961 to celebrate the 100th anniversary of M.I.T., and published in the collection *Management and the Computer of the Future* [Greenberger 1962], Perlis recognized and expressed a number of issues and concerns that are still central today. In his talk, entitled "The Computer in the University", he noted that most university computer use is characterized by "extensions of previously used methods to computers; and they are accomplished by people already well trained in their field who have received most of their training without computer contact." As Mateas points out, "this approach is similar to the way computing is currently taught in media arts programs, primarily as a black box tool (substitute Photoshop, Director and Flash for Algol 60) rather than as a process-based medium with its own unique conceptual possibilities" [Mateas 2005].

Perlis goes on to assert that the purpose of a university education, regardless of the particular field of study, is to help students develop an intuition for which problems and ideas are important or relevant ("sensitivity… a feeling for the meaning and relevance of facts"), to teach students how to think about and communicate models, structures and ideas ("…fluency in the definition, manipulation, and communication of convenient structures, experience and ability in choosing representations for the study of models, and self-assurance in the ability to work with large systems…") and to teach students how to educate themselves by tapping the huge cultural reserves of knowledge ("…gaining access to a catalog of facts and problems that give meaning and physical reference to each man's [sic] concept of, and role in, society" [Mateas 2005]. He argues that the computer plays a critical role in at least the last two areas, and, during the discussion period, agrees that computers play a critical role in the development of intuition and sensitivity as well.

As, for Perlis, procedural literacy lies at the heart of the fundamental aims of a university education, he consequently argues that all students should make contact with computers at the earliest time possible: the student's freshman year. For 1961, as Mateas acknowledges, this is a radical proposal: all students, engineering and liberal arts students alike, should have a two-semester computer science sequence in their freshman year, this at a time when computers were still rare and esoteric. Even today, with the relative ubiquity of computers, most universities do not have such a requirement in place. This may well be due to the fact that historically, it has been challenging to introduce students to the benefits of computer science, programming, and procedural tools [Mateas 2005]. As artist-programmer Golan Levin [2009] states,

> just as true literacy in English means being able to write as well as read, true literacy in software demands not only knowing how to use commercial software tools, but how to create new software for oneself and for others.

> Today, everyday people are still woefully limited in their ability to create their own software. Many would like to create their own programs and interactive artworks, but fear that programming is "too hard." The problem, it turns out, may not be programming itself so much as the ways in which it is conventionally taught.

3.2.1 The Difficulty of Teaching Programming

> *What hasn't changed is that programming is still a hard activity and a difficult skill to learn. Few students will understand programming well enough after completing their first programming courses to be able to write simple programs, let alone use programming as leverage for understanding other domains. - Guzdial [1994]*

Despite more than fifty years of attention from computer science educators, basic programming fundamentals remain a remarkably difficult subject for many students. As Jenkins [2002] argues, this situation is less than ideal:

> Few computing educators of any experience would argue that students find learning to program easy. Most teachers will be accustomed to the struggles of their first year students as they battle in vain to come to grips with this most basic of skills and many will have seen students in later years carefully choosing options so as to minimise the risk of being asked to undertake any programming. This is a sad and depressing state of affairs.

While there is a range of possible explanations for this fact, the most critical difficulties seem to stem from some or all of the following issues [Guzdial 1994]:

- *Assembling programs is hard*. Programming languages have only a few components, which are combined in many different ways, and learning to understand the semantic results of different combinations is complex [Schneiderman 1977]. Understanding how to combine programs to achieve particular goals is a challenge [Spohrer 1985, Spohrer 1989].

- *Syntax is complex*. When students try to combine elements, syntax gets confused, which leads to students battling syntax problems as they struggle to understand semantic ones [Perkins 1986, Johnson 1985]. When the syntax problems are alleviated, students can focus on the semantic ones [Hohmann 1992; Soloway 1993; Anderson 1989; Garlan 1984].
- *Students lack an understanding of computational process*. Many students do not understand how interpretation of traditional computer languages works, e.g., where does control flow and how do variables get updated [DuBoulay 1989]. If students are presented with a simplified or clearer description of the process, they can understand their programs more easily and perform more successfully [diSessa 1985; diSessa 1991].

There is no question that many students find the study of computer science and programming extremely difficult, especially at elementary levels. In fact, it would seem that often even the most basic concepts (e.g. variables) appear to be most difficult for students [Sleeman et al. 1988; Samurçay 1989; Paz 1996]. But deep misconceptions are not limited to elementary programming. Holland, Griffiths, and Woodman [1997] show the extent of the misconceptions held by more advanced students studying object-oriented programming. They report on a range of misunderstandings, e.g., the conflation of the concept of an object with other concepts like variable, class, and textual representation.

3.2 Constructivism, Constructionism and Generative Pedagogy

> *Constructivism is a theory of learning, which claims that students construct knowledge rather than merely receive and store knowledge transmitted by the teacher. Constructivism has been extremely influential in science and mathematics education...* *– Ben-Ari Mordechai [2001]*

To address the inherent difficulty of programming, educators have experimented with a range of pedagogical philosophies. Constructivism, based originally on Jean Piaget's cognitive theories on knowledge acquisition by children, is the theory that rather than passively receiving knowledge from teachers and textbooks, students learn best when they actively construct it, building recursively on their previous knowledge and experience of the world. After many years of debate with so-called "behaviorist" educational theories, constructivism has, arguably, emerged in recent years as a dominant philosophy [Mordechai 2001], and has been applied across disciplines, from the humanities to the sciences to the arts. Unfortunately, computer science education, where emphasis has traditionally been placed on abstraction, rather than application, has been slower than many other fields in this respect. As Mordechai [2001] states:

> Constructivism has been intensively studied by researchers of science education (Glynn, Yeany and Britton 1991) and mathematics education (Davis, Maher, and Noddings 1990; Ernest, 1994), to the extent that "radical constructivism represents the state of the art in epistemological theories for mathematics and science education" (Ernest, 1995, p. 475). However, there has been much less work on constructivism in computer science education (CSE)... While many computer science educators have been influenced by constructivism, only recently has this been explicitly discussed in published work…

The basic tenet of cognitive constructivism is the student's creation of meaning. While Piaget tended to emphasize learning through play, the basic theory supports a range of educational activities, as long as the student's construction of meaning is a primary concern. Constructivist educators emphasize having students take control of their own learning, and de-emphasize lectures and other transmissive forms of instruction [Guzdial 1997].

3.2.2 Constructionism

The most clearly articulated example of constructivist theory applied to computer science education (CSE) is the *constructionist* approach, developed by Papert in his 1980 book *Mindstorms: Children, Computers, and Powerful Ideas*, which has gained significant traction in some corners of the discipline. Constructionism focuses not on Piagetian stages of development and the constructivist nature of the mind, but rather on knowledge construction that occurs specifically during designing, building and making activities. As Papert says, "The constructionist approach to learning asserts that people learn particularly well when they are engaged in constructing a public artifact that is personally meaningful." [Guzdial et al. 1990]

What is critical in Papert's theory of constructionism is that students need to be engaged in the construction of artifacts in which they have some stake; "public" artifacts[42] as he calls them. When this condition can be met, powerful discussions about playing and making can occur as students are released from the abstracted busywork of traditional classroom activities. Instead they focus on learning by doing, learning not only *about* programming, but *through* programming; learning "to think, to tinker, to putter, to make mistakes and to learn from them" [1980]. To make clear the distinctions between these various terms (cognitive constructivism, philosophical constructivism, and constructionism), Guzdial [1997] says:

> Piaget was talking about how mental constructions get formed, philosophical constructivists talk about how these constructions are unique (noun construction), and Papert is simply saying that constructing is a good way to get mental constructions built. Levels here are shifting from the physical

[42] Note the emphasis here on the "public" nature of the artifact, often a problematic element for student work in computational media, as the current infrastructure (as well as academic norms) often frustrate or prohibit public sharing in this way. Enabling this at both the technical level (via the generation of browser-executable content) and the pedagogical level (via art-syle critiques) was key element of the RiTa/PDAL strategy.

> (constructionism) to the mental (constructivism), from theory to philosophy to method, from science to approach to practice.

The project-based workshop approach for the PDAL courses (and its integration with the RiTa tools) was directly influenced by the constructionist philosophy described above, both in its commitment to project-based, publicly viewable work, and in the notion of *reflexivity*. Papert describes how programming is *reflexive* with other domains, meaning that learning the combination of programming and another domain (art-making in this case) can be easier than learning each separately.[43] In such cases, synergies are created when concepts in another domain are mapped, or "reflected" back into the programming medium.

3.2.3 Generative Pedagogy

> *We want to improve learning by contextualizing concepts and problem solving inside structures which will give a base for making abstract problems "real." [Perlin et al. 2003]*

In the case of PDAL, learning to program means learning to construct representations for concepts. This in turn supports further learning of the concepts and the degree to which they can be procedurally manipulated in a creative fashion, thus providing a natural motivation for learning to program. This reflexivity is especially suited to the context of natural language, as we find direct mappings between the abstract nature of sign and signifier in both natural and computer languages. As one student commented,

> My work in this class has solidified my understanding of art and programming being very closely related. I have taken a number of courses... related to New Media art where programming and software art was often discussed though I never had the opportunity to create any myself. What this course really did was help me find a formal process for working within the

> intersection of my interests-- computer science and art. I intend to continue using the models we have used in this course to work on projects in the future. [PDAL 2008]

When students engage in manipulating mental models and creating symbolic representations, they find direct parallels between the domains of literature and computer languages. In literature, words, with parts-of-speech, map arbitrarily (and dynamically, based on the context) to concepts. In computer languages, variables (or objects), with types, map arbitrarily and dynamically to values in a running program, meaning there is no necessary or permanent semantic link between a variable and what it points to. In this way, the building blocks of language are similar to those of programming. Thus, in contrast with the recent trend toward visual media in such contexts (examples follow below), with language and literature we find direct corollaries between concepts in the application domain and in the programming domain, thus supporting constructionist reflexivity to a greater degree.

3.3 Related Pedagogical Initiatives

> *"What would happen if everyone in the US learned how to program computers at the same time they learned to read and write English?" [Perlin et al. 2003]*

While embedding computer science education in a socially meaningful context has generally not been a priority for mainstream computer science educators, there have been notable exceptions in recent years, specifically in the areas of gaming, storytelling, and creative expression in digital media.

3.3.1 Programmatic Game Environments

An increasing number of researchers have attempted to leverage gaming (especially multiplayer and/or collaborative games) to teach both STEM concepts[44] and core computer science ideas. As Plass et al. [2007] state:

> Increasingly, video games are being investigated for their instructional applications. Dickey (2005), Gee (2003), Shaffer, Squire, Halverson, and Gee (2005) and Shaffer (2006), among others, have all suggested that video games are environments that allow for "thickly authentic" (Shaffer & Resnick, 1999) learning, or learning that enables students to acquire knowledge that is personally meaningful, has real-world application, and that is associated with practice, rather than rote memorization... Games may mend what Brown, Collins, and Duguid (1989) refer to as the "breach between learning and use" and, through practice and reflection, enable learners to make the connection "knowing what" and "knowing how".

A subset of these projects focus on game *creation* which *(*implicitly or explicitly) leverages constructionist theories, e.g., Hands, ToonTalk, Klik'N Play, StarLogo TNG, and Scratch. Similarly, a number of gaming environments have included programming as a central element of the play experience. The Alice[45] project at CMU, for one, has shown notable results with this approach, especially regarding underrepresented populations, e.g., the poor, females, artistic and musically-oriented children, etc. [Kelleher and Pausch 2005]

CMU's Alice is a 3D programming environment that facilitates the creation of animations for storytelling, interactive games, and web videos. Alice is a freely available teaching tool intended as a student's first exposure to object-oriented programming. It attempts to allow students to learn fundamental programming concepts in the context of

---

[44] STEM refers to "Science, Technology, Engineering, and Mathematics". The STEM Education Coalition works to support STEM programs for teachers and students at the U. S. Department of Education, the National Science Foundation, and other agencies that offer STEM related programs. See http://www.stemedcoalition.org/.

[45] See http://www.alice.org/.

creating animated movies and simple video games. In Alice, 3-D objects (e.g., people, animals, and vehicles) populate a virtual world and students create a program to animate the objects. In Alice's interactive interface, students drag and drop graphic tiles to create a program, with instructions corresponding to standard statements in a production-oriented programming language like Java, C++, or C#.

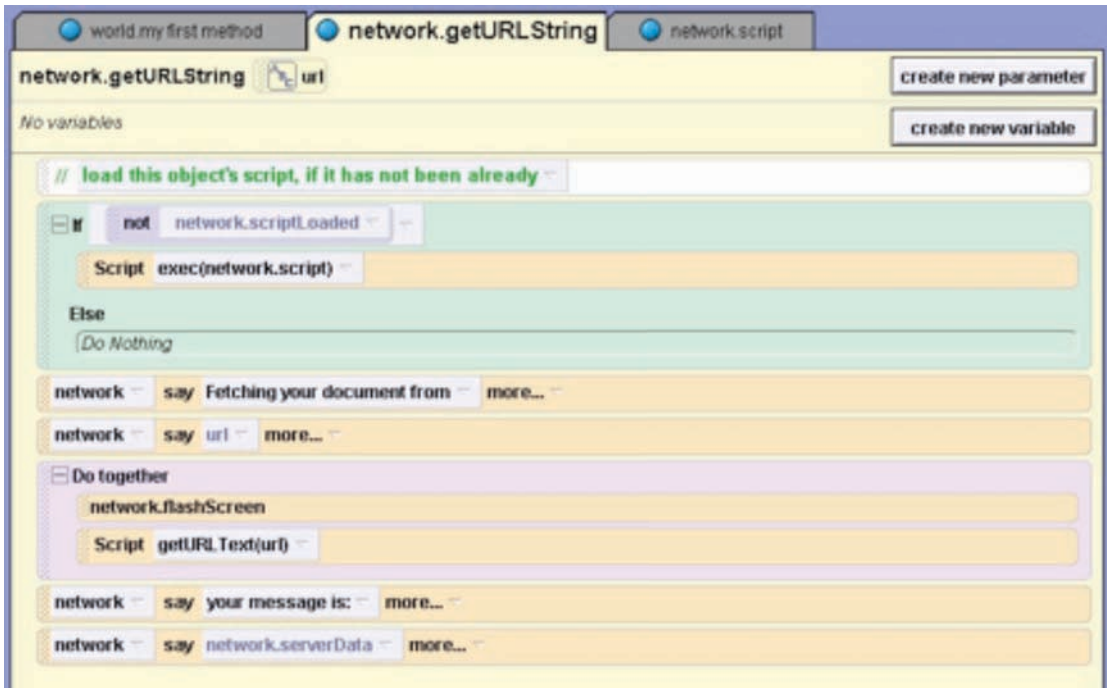

Figure 11: The Alice programming interface

Alice allows students to visualize how their animated programs run, enabling them to more easily understand the relationship between programming statements and the behavior of objects in their animation. By manipulating the objects in their virtual world, it is hoped that students gain experience with the programming constructs typically taught in an introductory programming course.

A related project is Kelleher's "Storytelling Alice",[46] which attempts specifically to attract girls to programming and computer science. Kelleher et al. [2007] describes Storytelling Alice as:

> [a] programming environment that gives middle school girls a positive first experience with computer programming. Rather than presenting programming as an end in itself, Storytelling Alice presents programming as a means to the end of storytelling, a motivating activity for a broad spectrum of middle school girls. The development of Storytelling Alice was informed by formative user testing with more than 250 middle school aged girls. To determine girls' storytelling needs Storytelling Alice includes high-level animations that enable social interaction between characters, a gallery of 3D objects designed to spark story ideas, and a story-based tutorial presented using Stencils, a new tutorial interaction technique.

The RAPUNSEL[47] project takes a somewhat similar approach, attempting to teach programming concepts to underprivileged populations via a social gaming environment. The project is described as:

> a large multi-disciplinary collaboration aimed at designing and implementing an experimental game prototype to promote interest and competence in computer programming among middle-school aged girls, including girls from disadvantaged home environments. This three-year project includes a variety of interlinked components: engineering, pedagogy, interface, graphics, networking and more… In the current iteration of the game, each player logs onto and is assigned a home environment which houses a central character. Players are supposed to 'teach' these characters to move, dance, and behave in a variety of ways by programming them in a simplified variant of the Java language. In one scenario, for example, tying mastery of Java with game performance, players must program characters to perform increasingly complex dance behaviors which, according to the game's narrative, increases the characters' degree of satisfaction across a range of metrics, such as being allowed by a fearsome bouncer into a club, or excelling in a dance competition. The motivation for this narrative is its potential to serve as an attractive pedagogical medium for the target audience. [Flanagan et al. 2005]

---

[46] http://www.alice.org/kelleher/storytelling/

[47] See http://www.RAPUNSEL.org/; also Perlin et al. [2003] and Plass et al. [2007].

Figure 12: The RAPUNSEL environment.

An important component of both of the projects is the implementation of real-time (always-running) custom coding environments designed to minimize errors that are common among student programmers. While Alice uses a visual programming language in which "instructions correspond to standard statements in a production oriented programming language, such as Java, C++, and C#",[48] the RAPUNSEL project attempts to teach a "dialogue" of Java, thus maximizing students' ease-of-transition to this commonly-used language.

[48] See http://www.alice.org/index.php?page=what_is_alice/what_is_alice/.

There are two other game environments worth briefly mentioning: Squeak Etoys and Scratch. Squeak Etoys[49] was inspired by LOGO, PARC-Smalltalk, Hypercard, and starLOGO. It is a media-rich authoring environment with a simple, powerful scripted object model for many kinds of objects created by end-users that runs on many platforms. It provides a unified user-interface and scripting environment for working with digital media. It includes 2D and 3D graphics, images, text, particles, presentations, web pages, videos, sound and MIDI, etc. It includes the ability to share desktops with other users in real-time, so many forms of immersive mentoring and play can be done over the Internet. It is multilingual, runs on more than 20 platforms bit-identically, and has been successfully used in the USA, Europe, South America (Brazil, Colombia, Argentina), Asia (Japan, Korea, India, Nepal), and elsewhere. It is free and open source, and can be downloaded for a number of platforms [Kay 2005].

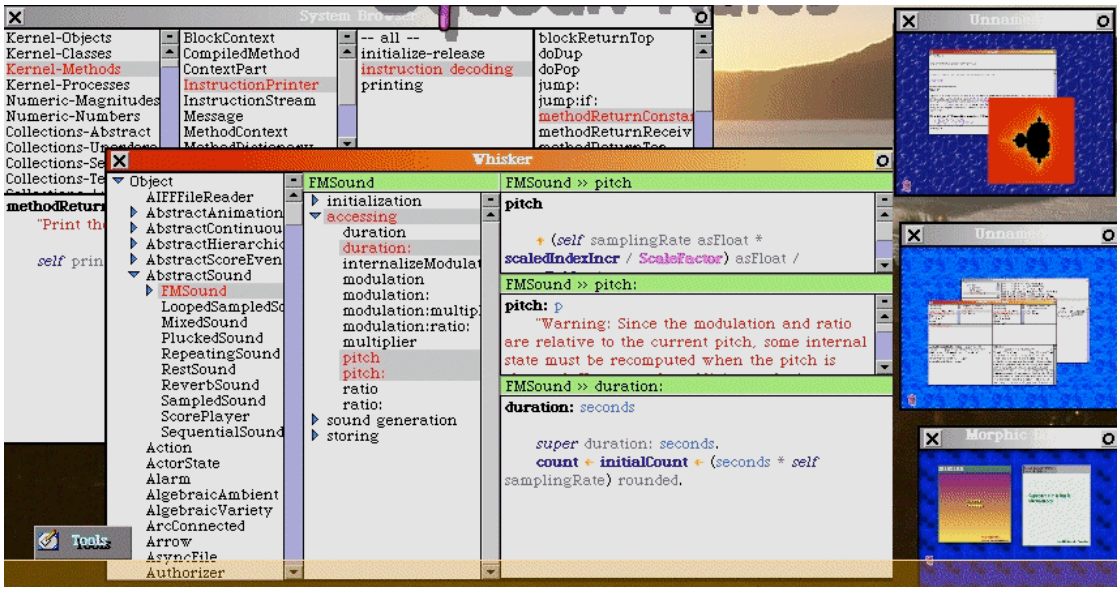

Figure 13: The Squeak-Etoys environment.

[49] See http://www.squeakland.org/.

Scratch[50] is a graphical-programming environment intended to enable young people (ages 8 and up) to create their own interactive stories, games, and animations, and to share their creations on the web. Scratch is designed to make programming more tinkerable, more meaningful, and more social. Since Scratch was launched in May 2007, more than 300,000 projects have been shared on the Scratch website, which has been called "the YouTube of interactive media." As young people create and share Scratch projects, they learn to think creatively, reason systematically, and work collaboratively. Scratch is a project of the Lifelong Kindergarten group at the MIT Media Lab, directed by Mitchel Resnick.

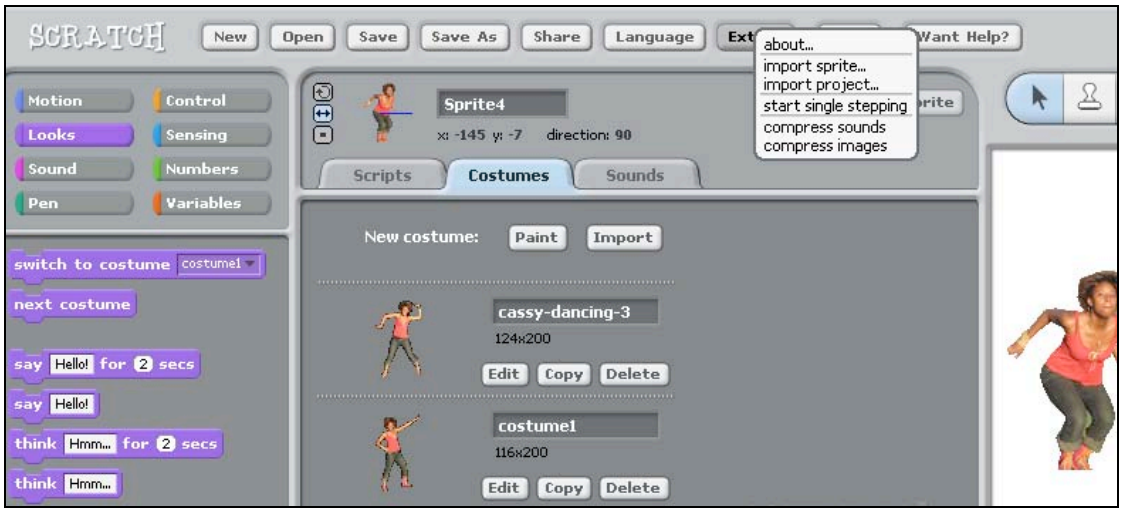

Figure 14: The Scratch environment.

3.3.2 Digital Media Manipulation and Max/MSP

> *Media and computation are complementary—digital media are computationally created and manipulated... students who are interested in working with digital media can look "under the hood" to see how their art, music and websites come to be. At the same time, a creative context for*

[50] See http://scratch.mit.edu/.

> *programming provides students with an interesting introductory computer science experience [Guzdial 2003]*.

Much like gaming, digital media manipulation has been proposed as a useful context for programming and computer science education [Guzdial 2003] as it is directly relevant to students' lives. According to a 2005 study conducted by the Pew Internet and American Life project, more than one-half of all American teens, and 57 percent of teens who use the Internet, could be considered media creators [Jenkins 06]. Mark Guzdial, for one, has been a consistent proponent of this approach, which he refers to as 'Media Computation':

> The use of domain-specific contexts for computer science learning is being explored by researchers with the aim of improving students' experiences in IT courses, we propose using this approach for introductory computer science. Because media computation focuses on data that is important to students—their own photographs, recordings, and creations—and allows them to use computation in a personally expressive way, we expect to better engage non-computer science majors than traditional introductory courses and, as a result, improve retention rates [Guzdial 2003].

He describes the goal of "Introduction to Media Computation", a class he started at Georgia Institute of Technology in 2003, as:

> learning about the fundamentals of digital media with basic programming skills and computer science concepts. Media and computation are complementary—digital media are computationally created and manipulated. In the media computation course, students who are interested in working with digital media can look "under the hood" to see how their art, music and websites come to be. At the same time, a creative context for programming provides students with an interesting introductory computer science experience [Guzdial 2003].

While this course (in contrast to PDAL) is not open to CS majors, its goals are very similar, that is, to teach programming and computation within the context of digital media. Thus far, the course seems to have been quite successful:

> The results have been dramatic. 120 students enrolled, 2/3 female, and only three students withdrew. By the end of the semester, the combined withdrawal, failure and D-grade rate had reached 11.5%--compared to 42.9% in the traditional introductory computer science course. 60% of the students who took media computation reported that they would be interested in taking an advanced version of the course; only 6% reported that they would otherwise be interested in taking more computer science. Results of the trial indicate that media computation motivates and engages an audience that is poorly served by traditional computer science courses [Guzdial 2003].

Success like that seen in Guzdial's class has bolstered claims by those who have argued that the difficulty students have with learning computer science is in part due to its presentation. These critics[51] have often concluded that a visually-oriented approach is superior, for a number of reasons:

- It provides a concrete (visual) metaphor for computational processes.
- It reflects the dominance of the visual in contemporary culture itself.
- It provides immediate "all-at-once" feedback, in contrast to more temporally-oriented media like language or sound, thus providing more "bang-for-buck" from programming techniques (compare an image blur to a sentence blur).
- The image is easily decomposed into atomic units (the pixel), which can then be modified and re-combined. In language it is difficult to make such distinctions. In various cases the atomic unit might be considered as the word, the letter, the phoneme, or syllable.
- "Non-figurative" or abstract images are generally easier to process than non-figurative language. Grammaticality and sense are less constrained in images. [Smith 1975]

---

[51] For an early instance of this argument, and an innovative system to address it, see the Pygmalion system [Smith 1975].

Often such critiques have led to new programming and/or teaching paradigms based on visual metaphors[52]. However, this trend has not proven to be a panacea. While there is evidence that visual environments can make programming easier, it is less clear that students who use them learn an equivalent amount about core CS concepts. A good example of this discrepancy is the Max/MSP environment originally designed by Miller Puckette in the mid-1980s.

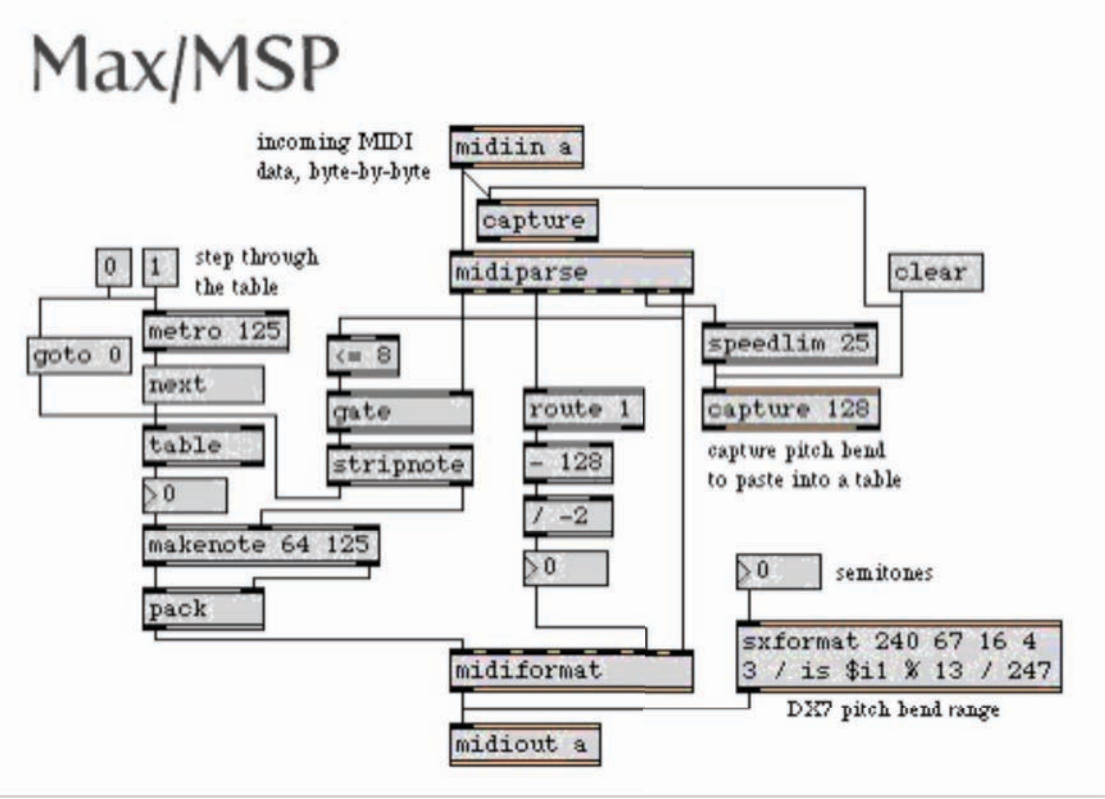

Figure 15: The Max/MSP environment.

[52] Examples include Pygmalion, Alice, MAX/MSP, Scratch; even the now ubiquitous WIMP (windows, icons, menus, pointing-device) interface.

Max is a graphical, real-time, drag/drop programming system initially designed for music processing. Rather than writing scripts or procedures, students connect objects in a visual space according to a "patchbay" metaphor, where the output of one "box" is connected to the inputs of other boxes.

While Max has proven to be a highly successful tool for practicing digital artists, it has had less success as a teaching tool. While students seem generally able to realize their ideas in the Max environment, prior exposure to Max seems to have little affect on a student's capabilities with more traditional textual languages like C, Java, or Python, especially as relates to program structure. As one student commented,

> My previous experience with programming in art was through the lens of Max/MSP, a program that is both very closely linked with sonic art as well as very unlike other programming languages (if it can even truly be called a programming language). The biggest adjustment I had to make while creating works in Processing was changing the way I thought about time...

Interestingly, "key" concepts found within most introductory computer science classes (for instance, variables and iteration) appear to be absent within Max (though they are actually just hidden). In fact, it is our experience that students with only a Max background seem to have more difficulty conceptualizing ideas like variables, iteration and operation order[53], as they try to map them onto their prior experience, than students with no prior programming experience whatsoever. Thus the question arises of just how important concepts like variables and iteration actually are if practicing digital sound artists can implement the procedural processes that they need to while these concepts remain hidden.

This question may be of particular importance to computer science educators as it suggests a potential reconsideration of what we consider "core" in the discipline. It would

[53] Some operations in Max/MSP are ordered according to their position in the interface.

appear that, in the light of new environments like Max, we would either have to delay topics like variables and iteration (usually among the first taught), or we would have to abandon Max-like environments as teaching tools (to the degree that they obscure these concepts), at least as first-languages for students. On the other hand, first learning a "procedurally-oriented language" does not appear to cause an analogous slowdown for students learning Max. In fact it seems, perhaps surprisingly, to have little affect whatsoever, at least in our experience with students in the PDAL class. While there may be minor frustrations for students with traditional backgrounds (having learned a language like C or Java) who try to learn Max, when trying to set or get a simple variable, for example, these appear to pass quickly in most cases.

3.4 Programming for Digital Art and Literature: the Teaching Environment.

> *If I were going to take one pedagogical myth to task in this thread, it would be the "adding pictures makes it easier for non-techies" myth – this isn't Fun with Dick and Jane we're trying to learn and teach. In fact, there are a number of expressive things with text that can be done with Scheme that don't involve higher math and still teach recursion, environments, scope, assignment, meta-circular evaluation and the like. -William Huber [in Mateas 2004]*

As a forum for engaging students with core programming concepts, Programming for Digital Art and Literature (PDAL) differs from the above approaches in several important respects. First is its use of the traditional text-based programming environment. In this respect it builds on the primary manner in which a vast majority of programming is currently done, both in academia and in industry. Although students using RiTa generally begin with "supportive" text editors, e.g., Processing or Eclipse, no such tools are required. RiTa, with or without Processing, can be written and compiled using a very basic text-editor and the

"javac" compiler. Further, as the primary representation of code is a plain text file, it is easy to share and adapt code to and from other textual languages, especially with other C-based languages, e.g., C/C++/C#/JavaScript, ActionScript, Ruby, and Perl. This fact goes a great distance toward both facilitating the adaptation of existing code and, more importantly, in maintaining transparency for the student. While there are always lower levels of code that may be temporarily obscured, the student can be confident that their tools are not performing some kind of *magic* to which they will never have access. Further, RiTa code *looks like* most of the other code students are likely to come across, which reinforces the idea that they are doing *real* programming. Finally, there is generally little difficulty for students as they migrate to other text-based languages in subsequent CS courses.

To further illustrate this point, we can place the environments discussed thus far on a continuum; from Guzdial's Media Computation class at Georgia Tech, which uses the Python language at one end, to the various classes at CMU and elsewhere that use the Alice environment, at the other. Python, as used in the Media Computation class, represents a general-purpose language with little specific support either for the arts in general, or for the specific exercises in the class. CMU's Alice environment, on the other hand, is highly tailored to the course content, but less generalizable to programming in general.

Although students using the Alice environment appear to explicitly learn core concepts (as opposed, for example, to those starting with Max/MSP), Alice does not provide a smooth transition path to "mainstream" languages, e.g., those used in business and research, or even in the arts. While it may be argued that this signals a problem with mainstream languages rather than a problem with visual drop interfaces like Alice or Max, the situation does not appear likely to change in the near future, and thus presents an added difficulty for students as they progress. Of course, the characterizations above are mostly anecdotal and

representative only of our experience with these environments in a classroom context. Further research is clearly needed to assess the affects of these different tools on the learning process, both within and beyond a student's first few exposures. Some initial steps in this direction has been taken by Tew et al. [2005] in their comparison of three courses at Georgia Tech, and also in the evaluation of the RiTa tools presented in Chapter 5. One potentially useful experiment would be to teach the same material with a range of different toolsets (e.g., a general-purpose language, a RiTa-like approach, and a custom environment, e.g., Max or Alice) and compare student confidence, efficacy and comprehension of core concepts.

3.4.1 Programming Languages in an Educational Context

To continue our comparison, we might also consider the level at which a teaching language operates on a similar continuum. On the one side of such a continuum we would place highly specific environments, well-suited for a particular purpose. An example might be Adobe's Photoshop, an image manipulation program with minimal programmatic capabilities (macros, actions, filters, etc.). It is highly optimized for its purpose and relatively intuitive for users, at least those familiar with the context. On the other hand, it is extremely difficult (if not impossible) to use in other contexts like sound or video manipulation, and as such, teaches little to users that can be generalized to other domains. Perhaps of more concern, at least in an arts context, is the way it narrows the range of possible outputs, so that artifacts produced tend to appear similar, and bear the "stamp" of the tool, as is often noted with various Photoshop filters[54].

[54] For an interesting take on this, see Adrian Ward's Auto-Illustrator project (at http://www.mediaartnet.org/works/autoillustrator/), described in detail in Weiss [2009].

On the other end of the spectrum we find general-purpose languages, like C, Python, or Java. When applied to a specific context, these languages tend to require a significant amount of scaffolding code before new users are able to accomplish even basic tasks (see the Java applet example in Technical chapter). While nearly all concepts are applicable across such general-purpose languages (learning one such language will generally decrease the time and difficulty to learn a second), these often prove frustrating for new students who wish to quickly translate their ideas into working programs. At the extreme end of this paradigm we might imagine Assembly language, or even a Turing machine. These formalizations are capable of computing just about anything one might imagine, but are so low-level as to be impractical for almost everything. As Mateas [2005] states:

> Any tools that reduces the friction for a certain class of programs, will dramatically increase the friction for other classes of programs. Thus, programming tools for artists, such as Flash, make a certain style of interactive animation easy to produce, while making other classes of programs difficult to impossible to produce. Every tool carries with it a specific worldview, opening one space of possibilities while closing off others.

Guy Steele demonstrates this situation quite palpably in his paper, "On Growing a Language" [Steele 1998], in which he allows himself only to use words which are either "primitives" or those he has defined previously in the talk, mirroring the difficulty of using a language without adequate library support. He says:

> [w]e have to do this a lot when we write real computer programs: a thought that seems like a primitive in our minds turns out not to be a primitive in a programming language, and in each new program we must define it once more… This is the sort of thing that makes a computer look like a person who is but four years old. Next to English, all computer programming languages are small; as we write code, we must stop now and then to define some new term that we will need to use in more than one place. Some persons find that their programs have a few large chunks of code that do the "real work" plus a large pile of small bits of code that define new words, so to speak, to be used as if they were primitives… I hope that this talk brings

> home to you… what it is like to have to program in that way... Each time I have tried this sort of thing, I have found that I can not say much at all till[sic] I take the time to define at least a few new terms. In other words, if you want to get far at all with a small language, you must first add to the small language to make a language that is more large [Steele 1998].

Steele, like others, advocates libraries as a solution to this problem, although he specifies that the libraries should be created by the community of users, rather than included as part of the original language [Steele 1998]. A language's ability to facilitate growth and development in this way thus becomes a primary criterion for its adoption. This maps closely to the RiTa approach, in which library features are added in an ad-hoc manner as suggested by students and practicing artists using the tools on a day-to-day basis. Thus the library is extended in an organic fashion, not by imagining potential uses, but instead by reacting to the actual needs of users. In this way, the functionality of the system does not grow to become overwhelming (due to sheer size and variety of uses) to new users. This approach is based directly on the model used in the Processing community (in contrast to Java itself, which includes hundreds of supplementary libraries as part of its distribution), which has proven to be highly successful in generating an active community of users and developers, contributing, and extending libraries in this incremental, as-needed fashion.

Another important way in which PDAL/RiTa differs from the other media-centric approaches discussed is its focus on language and literature. The use of natural language as a central element in digital media is potentially advantageous for a number of reasons.[55] First we can consider the ways in which language, and specifically literature, has been of unique historical importance in computer science. As described further in Chapter 4 (Prior Work), natural synergies between the two fields have led to a range of important results in both. Secondly, the reflexively informing relationship between natural and programmatic

[55] Though more research is required before this can be stated unequivocally.

languages can be quite productive. A focus on the former naturally and inevitably brings awareness of the unique (and sometimes arbitrary) properties of the latter, and vice versa [Howe and Soderman 2009]. As Stone [2002] describes, "[students] simultaneously get experience with central computer science ideas—data structures, unification, recursion and abstraction—and develop an effective starting point for their own subsequent projects."

For example, when first discussing ways of generating sentences with students, one of the first points that naturally arises is the distinction between context-free (programmatic) and context-sensitive (natural) languages. This key insight, which becomes quite tangible for students when presented with simple examples, opens immediately onto a host of important and central topics in computer science; from Turing machines, to Chomsky's language hierarchy, to state machines and regular expressions, to parsers and compilers and so on. No matter how immediate the response to a compelling visual image created programmatically[56], there is no such equivalence on a conceptual level when dealing with visual media. While one can imagine other ways of finessing this lack of conceptual relationship, what often happens is that media-based assignments and computer science concepts are addressed separately, and the motivation for learning the latter grows less clear.

3.5 Pedagogically-inspired Design Considerations

The above discussion of pedagogy in computer science (generally), and the pedagogical implications of using certain classroom tools and environments (specifically), highlights the importance of context in educational environments attempting to teach programming and computer science. This focus on context is an essential element of

[56] Via a convolution filter, for example, as assigned in Guzdial's "Media Computation" course.

constructivist-inspired pedagogy. Alison Tew [2008] describes context as "an application area for the content being learned that is familiar to and valued by the students in the course. A contextualized course applies its domain not only to projects, but to lectures, descriptions, examples, and assignments." She notes further that a range of studies have shown the value of such motivating contexts for teaching introductory computing courses, citing researchers who have observed that teaching within a context provides a means to attract and retain students—particularly those from underrepresented populations [Forte and Guzdial 2005]—and as a motivator for students to do work beyond that which is required [Forte and Guzdial 2004, [Kelleher 2006; Yarosh and Guzdial 2007], Furthermore, she describes how these effects appear to persist for both majors and non-majors, as well as for students at a range of institutions [Tew et al. 2005].

While we have thus far highlighted both tools and context, it should be noted that their relationship is as important as are either of their properties in isolation. In the case of PDAL/RiTa, the degree to which tools and learning are tightly coupled has proven to be beneficial and allowed both to develop in a mutually informing fashion. Such a coupling implies a degree of communication, if not close collaboration, between those creating the tools and those developing the accompanying intellectual program. In this regard we were in the fortunate (and perhaps rare) position of having a great deal of control over the ongoing development of both the tools in question and the accompanying pedagogical material (readings, assignments, discussions, critiques, etc.). In several cases, specific materials were chosen to reflect important aspects of the technology being used. Perhaps more unusually—and more interestingly—were the cases when the converse occurred; that is, where software tools were modified and/or extended in response to intellectual concerns raised specifically in

relation to the context.[57] Though the practicality of this situation may be questionable at larger scales, this should not prevent us from taking note of its benefits.[58]

Rather than simply providing the functionalities necessary for successful navigation of a course (as in the case of the NLTK[59]), we were forced to carefully consider the ways in which the design of RiTa might foster or frustrate practices important to artistic practice. The result of this iterative process was a set of arts-specific design considerations that we attempted to embed within each of the modules created. A subset of these considerations are presented in the Technical: Design Considerations subsection, with a focus on those that differ from "traditional" software engineering practices.

3.5.1 Supporting Serendipity

> *Creativity is allowing yourself to make mistakes. Art is knowing which ones to keep. (Scott Adams)*
>
> *A painting is a series of corrected mistakes. (Robert Bissett)*
>
> *It was when I found out I could make mistakes that I knew I was on to something. (Ornette Coleman)*
>
> *Do not be afraid of errors. There are no errors. (Miles Davis)*

---

[57] For an extended discussion of the notion of authorship within digital literary art, see [Howe and Soderman 2009].

[58] Custom tool integration in a specific pedagogical context also presents its own set of difficulties. The task of creating an instructional tool that specifically addresses any context (art-making in our case) complicates the design process by introducing a set of unique constraints which must be iteratively refined and tested.

[59] See [Bird and Loper 2002].

> *Most of my advances were by mistake. You uncover what is when you get rid of what isn't. (Buckminster Fuller)*

Often, as suggested by quotations above, an important element of the artistic process is unexpected or serendipitous outcomes that can take the programmer/artist in newly productive directions. Facilitating such outcomes is thus an important (and often overlooked) element of designing for the artistic context, whether in an academic or professional environments. Computer scientists tend to conceptualize the coding process as the (often difficult) process of correctly implementing a pre-existing idea in a concrete/formal language. For the traditional computer scientist, the targeted behavior of a program is generally known before coding begins. Thus, we see the emergence of methodologies like "test-driven-development", in which the tests of a program's correctness are written before the program itself. While such an approach is useful and often appropriate in many contexts, it is, at least in some cases, the opposite of what is required for the arts. Artists often learn what it is they are making as they make it. As one student commented,

> I need to first articulate what I want to say, through my program, plan it out and answer the entire how, what, when and where questions of my project. At this stage nothing can be left to chance. Then comes the execution and implementation. And at this point even though we are 'talking' to a computer that only does what we tell it to do, we find that in the gap between what we want it to do and what we actually tell it to do (also what the computer does), there lies the act of the unexpected outcome so valued in the intuitive process of art making.

Mistakes and accidents occur in artistic practice, as in more structured programming, but here they can often be highly productive, and thus warrant special attention from tool designers. In order to facilitate serendipity in its various manifestations (surprises, mistakes, chance), several specific features were included in the RiTa design: default non-determinism, soft-failure, and runtime exceptions, each discussed below.

In general practice, one wants a program to produce the same output each time it is run. While some algorithms may specifically use randomness (e.g., quicksort), this generally does not affect the output of the program. A correctly implemented randomized quicksort, for example, will return the same ordering each time it is run. Its randomness is intended only to optimize its performance. In fact, a whole class of algorithms, generally referred to as "Monte Carlo" methods[60], after the casino in Monaco, exploit random operations to achieve deterministic outcomes[61].

To the contrary, non-deterministic outcomes are often an important part of programs written by artists. This is especially true in interactive and/or generative work, in which it is the very fact that each piece starts off at a different "place" that is often part of the appeal. More important however is the fact that non-determinism allows the programmer herself to traverse the "probability space" of a program *during* development, which can provide key indicators as to whether the piece will prove interesting to an audience. To support this possibility, iterators in RiTa are non-deterministic by default. If one asks for the set of synonyms for a word, for example, unless specified to the contrary, they will be returned each time in a different order. Further, since a range of RiTa methods allow the programmer to specify an optimal number of results (after which the method returns), these methods actually

---

[60] Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results. Monte Carlo methods are often used when simulating physical and mathematical systems. Because of their reliance on repeated computation and random or pseudo-random numbers, Monte Carlo methods are most suited to calculation by a computer. Monte Carlo methods tend to be used when it is unfeasible or impossible to compute an exact result with a deterministic algorithm [Metropolis and Ulam, 1949].

[61] The name "Monte Carlo" was popularized by physics researchers Stanislaw Ulam, Enrico Fermi, John von Neumann, and Nicholas Metropolis, among others; the name is a reference to the Monte Carlo Casino in Monaco where Ulam's uncle would borrow money to gamble.

return different results each time. For example, if one is querying for rhymes in the lexicon, it is often useful to specify the minimum and/or maximum number of results desired; both to reduce processing time (when a great number of rhymes exist) and/or to allow the rhyme algorithm to relax its constraints (when too few are found)[62].

Additionally, to support discovery via unexpected results from programming errors, the RiTa toolkit has evolved toward what we call a "soft-failure" model This means that the library, wherever possible, attempts to present *some* result to the user, even in the case of "predictable" programming errors, such as when a resource does not exist at a specified location, or the pre/post constraints for a method are not met. Instead of quickly exiting, the library will attempt to generate some default behavior in such cases, while simultaneously printing an error report to the console to explain what is often unexpected behavior. Further, the library generates only "runtime" exceptions, which means that none of the user's code needs to be wrapped in try/catch methods. This aligned nicely with both the Processing and Eclipse environments, which eliminate the distinction between the compile and execution steps. Processing presents only a "play" button, which first compiles, then runs the current program, while Eclipse performs continuous compilation, highlighting errors as they are typed.

As one computer science student commented,

> For me, as a fairly experienced programmer... the coding I have previously done has had very strict rules about what I had to achieve, and the process was simply working on the code until it reached the constraints that had been set for me by someone else. The process of coding by happy accident, then, was something that I had never before experienced. The first or second week

[62] It is also essential of course that such functionality can be easily disabled during debugging so that problematic behavior can be reproduced as necessary until it can be repaired.

> of class, the piece we read about allowing yourself to make "good" mistakes really proved true in my experience. I found that trying something, and seeing what sort of effect it produces when I ran it, had a very strong influence on the ideas behind my projects. Each experiment communicated a different idea, and while I didn't always end up communicating what I had originally intended, I always found something interesting to say through the process of writing the code

3.5.2 Supporting Artistic 'Misuse'

> *Throughout the late twentieth century and into the twenty- first, it became almost common to see performances that used some element of the manipulation, breaking, or destruction of sound mediation technologies… [Kelly 2009]*

Somewhat related to the support of serendipity via coding errors is the consideration of intentional "misuse" of tools for artistic affect. Instances of this tendency emerged as early as the first semester of teaching with RiTa. Later we recognized these instances as part of a larger pattern, hinting at an artistic strategy that we might want to facilitate (though at the time it was less than clear how to do so). One student, in a warm-up exercise in the first weeks of the semester, discovered a bug in the text-rendering module which caused text objects (RiTexts) to leave traces on the screen when their contents were swapped at high rates. Rather than report the bug and ask for a fix, as students generally did, this student built a unique project around the behavior. By beginning with a high rate of text change, then placing control of this parameter in the hands of the user (via mouse movement), the user could build up "traces" of previous renderings. If performed "correctly" by the user, a "hidden" message was revealed in the textual sediment. Another student, exploiting a threading issue that occurred when large numbers of RiSpeech objects were created in the same applet, was able to create a unique aural experience as diphones from a single word or phrase were pronounced in sequence by different voices with different timbres. In both cases,

interesting interactive works were built around specific unintended behaviors in the toolkit, behaviors that generally occurred with very low frequency.

In his recent book on contemporary audio art, "Cracked Media", Caleb Kelly [2009] discusses this phenomenon in some depth:

> the inquisitive artist, on finding a technology that is new to him or her—be it a newly developed tool just released into the market or an outmoded technology found in a dusty corner of the studio—sets out to see how it works and discover the boundaries and limitations of the device. What can this tool do, and how can I use it in a way that may not have been originally intended? This might be achieved by simple manipulation or modification (taking the technology apart and trying to put it back together), or it might be through overloading it or otherwise stretching its operating parameters, until it starts to fall apart or break down... There is nothing new in this idea: the painter who uses the brush handle on the canvas and the guitarist who plucks the strings around the head of the electric guitar are both engaged in a similar area of practice. Experimentation with readily available tools and resources is central to contemporary artistic practice…

Of course such techniques are not limited to the realm of audio. Dating back to the mid-twentieth century and beyond, digital artists of all genres have manipulated, cracked, and broken media technologies to produce novel artistic experiences.

The question of how to support this artistic technique in the context of software, however, does not have a simple answer. One very basic (but rarely employed) technique is to provide continual public access to all versions of a software tool[63], even those with already-identified issues. This ensures that projects leveraging such "cracks," to use Kelly's terminology [2009], can be further developed, even after the relevant defect has been fixed (as was the case in the RiTa examples mentioned above). Another, perhaps more productive

[63] This has become a somewhat common practice in open-source projects, e.g., those hosted on Sourceforge. See http://sourceforge.net/.

strategy, is to attempt to code the software tool in such a way that it continues to function, even if in an unexpected way, in unexpected use-cases. If we model the possible execution paths of a program as a finite state machine, this entails special attention, and programming resources, to so-called *fail-states*. Rather than simply exiting with an error message, one must notify the user of the unexpected state, while still enabling the program to continue execution, often with some best guess as to reasonable default behavior. In the text-to-speech case above, it might appear acceptable to code the program in such a way that it simply exits (with a descriptive message) after some maximum number of speech objects are created. Unfortunately, this reasonable strategy would both have prevented the interesting outcome described above and prevented the discovery of the bug, which more than likely had other ramifications.

Similarly, considerations of misuse compelled us to permit a very wide range of parameter values for all functions in RiTa, enabling users to experiment with an object by "overloading it or otherwise stretching its operating parameters" [Kelly 2009]. This means that for most applications, the edges of the range (say for text-to-speech voice parameters), are not accessed in typical use-cases. But for those interested in putting the system to unexpected uses, the fact that the larger parameter range is accessible can be quite productive. John Ippolito [2001], one of the earliest theorists to have acknowledged this type of artistic strategy, says:

> One of those myths is that creativity lies in applying the right tool for the right task—i.e., managing technology. Magazine editors, advertising execs, and Web site producers regularly employ "creatives" to spice up their products. The assumption behind this ludicrous adjective-turned-noun is that a creative person is simply a painter of pictures or a teller of stories—especially one adept at Photoshop or AfterEffects. While managing technology is certainly a valuable skill—for artists and others—it's not the same as creativity. When you manage technology well, you are simply carrying out the agenda of the designers of that technology. A composer who

> uses a car to drive to the concert hall is managing technology. But when Laurie Anderson composed a drive-in concert of motorists beeping car horns, she was being creative. To misuse technology is not just to manage technology, but neither is it to mismanage technology. There's nothing creative about broken links or glacial downloads. Misuse is deliberate. When [Nam Jun] Paik made a work for violin, he did not merely play it badly. In 'One for Violin Solo' (1961), he raised the instrument slowly over his head and then brought it crashing down on a table, smashing it to smithereens. Mismanagement can sometimes be overlooked; misuse is unmistakable.

As Ippolito recognizes, it is often the *misuse* of technology that allows for the discovery of affordances either invisible to the designer or in direct contradiction to their aims. Such unexpected affordances often become the special ingredient that turns a programmatic work into an "art" piece, rather than just a re-inscription of the technology designer's intentions.

3.5.3 Supporting 'Inverted' Use

In the discussion of *Design Tensions* presented above (see section 2.2), we discussed the notion of *crossover* as manifested in students' *n*-gram projects. In addition to presenting a technical challenge, this design tension helped to illuminate another case in which the arts context suggested practices that contradict those of generally employed in research. While the typical application of *n*-grams would be to find sentences that are *most* likely to occur, individuals engaged in literary art practice often desired the opposite, specifically the generation of relatively *novel* sentences, those that could logically occur, according to the constraints of the model, but were *less* likely to do so. In fact, as the size of the statistical models grew, ideal sentences were assigned an increasingly small probability for generation. Although this result was unexpected it makes a good deal of sense when we consider, for example, Ezra Pound's exhortation for writers to always "make it new". Further, as novelty is a central element in nearly all definitions of creativity [Sternberg 1999, Hewett et al. 2005], then what literary artists looks for is often what could, but has not yet, been written. This "inverted" use pattern proved to be a recurring theme when algorithms were borrowed from

existing areas of research for use in creative practice, a topic discussed again in the chapter 6. The question of how to support such use patterns is one that perhaps can only be addressed adequately by evaluating tools in areal-world contexts in which such uses can be manifest, and by trying to minimize, to whatever extent possible, assumptions about the "normal" uses for processes borrowed from other contexts.

3.5.4 Supporting Micro-Iteration

> *Almost as soon as we start to play with (and talk about) one prototype, we start to think about building the next. This process requires both the right tools (to support rapid development of new prototypes) and the right mindset (to be willing to throw out a prototype soon after creating it). [Resnick et al. 2005]*

While rapid iteration is stressed in a range of software methodologies[64], it is essential in an arts context, as functional requirements are not defined, but created anew in each iteration. Further, project cycles are generally shorter, especially in an educational context, (from a few days to a few hours) due to the relatively small size of teams (often a single programmer). University programs add still further time constraints, as many students are taking five or more classes simultaneously and a typical 14-week semester provides little time for a leisurely pace, especially as the first and last weeks are absorbed with introductory materials and final projects respectively. With these factors in mind, enabling what we called *micro-iteration* (rapid iteration on the order of seconds as opposed to minutes or hours) became an important design-constraint for the RiTa tools, without which the project-based format of the course would likely have been impossible.

---

[64] "Extreme" and "Agile" methods (the latter uses the term "timebox") generally target iteration cycles as short as 1-3 weeks.

As one insightful student commented, “this is how the creative process works. Sometimes you have an idea that's not all that great, but you still spend a lot of time having to discover that. The experience makes me want to perfect my creative process in general towards getting ideas realized (or at least conceptually tested for technical plausibility and personal investment) sooner rather than later.”

To facilitate this goal, several additions were made to RiTa, most notably the addition of server-mode processing via the RiTaServer object[65]. The RiTaServer component allows students to debug, modify, and run their programs any number of times, without noticeable delay, no matter how many text files, databases, or models are required. Resource-loading and model-creation, in server-mode, are handled by the RiTaServer process, which runs in its own virtual machine (either locally or remotely). By adding a single line of code to an existing RiTa program, methods can be dispatched to the server process, via a custom remote invocation subsystem, and results returned with no perceptible overhead.

Lazy instantiation of nearly all RiTa objects also serves to facilitate micro-iteration, as resources are only loaded when used. For example, while the core RiTa library contains a full English lexicon, it is only loaded (by default) when actually used during the run of a program. The same is true for the default RiTa fonts, Text-to-Speech samples, WordNet data files, statistical models, etc. Thus, each program only “pays”, in memory and processing-time, for what it uses, while the library still supports a range of diverse use-cases, each of which is highly resource-efficient.

[65] See Chapter 2 for a detailed description of this component.

3.5.5 Scaffolding and Transparency

> *Constructivist approaches to education advocate situating learning in real-world problems- but these problems are complex. Providing scaffolding allows learners to deal with a problem's complexity and successfully solve and learn from these kinds of problems.* *[Hmelo and Guzdial, 1996]*

Abstraction (generally via encapsulation) is an important part of tool design—in fact, it is often the primary reason for programmatic support tools in the first place—the abstracting away of implementation details that are either too difficult, too time-consuming, or not relevant to a particular problem. Similar techniques are often applied in pedagogical contexts, where they are generally referred to as *scaffolding*. Guzdial [1995] describes scaffolding as educational support structures designed "to enable students to achieve a process or goal which would not be possible without the support and... to facilitate learning to achieve without the support. Yet there are many ways of implementing scaffolding, each with specific benefits and costs that relate directly to the learning environment at hand." He continues, "in an educational context it is important that we don't confuse the ability to ignore certain concerns with the inability to access them at all." Hmelo and Guzdial [1996] refer to this as the distinction between "black-box" and "glass-box" approaches to scaffolding:

> *Black-box scaffolding* is scaffolding that facilitates student performance. Black-box scaffolding performs a task in place of the student performing that performance goal, usually because learning to perform that goal is determined to be unimportant for the learning goals of the activity. *Glass-box scaffolding* is scaffolding that facilitates performance and learning. It is important for the student to understand what glass-box scaffolding is providing because we want the student to be able to take on the functions that the glass-box scaffolding is providing.

While facilitating "performance" (most often understood as "productivity") is an important goal in an educational environment where the difficulty of very basic tasks can otherwise lead to mounting frustration, its costs can outweigh its benefits. When the scaffolding that enables such performance is invisible or opaque, it can confuse students as they encounter new concepts and cause them to perceive the often inaccessible lower levels as "magical". Further, opaque tools and subsystems can serve to disempower, leaving students to perceive themselves only as increasingly advanced media-consumers, rather than creators. Yet at the same time, the mass of details that go into programming even a very simple sketch can be overwhelming to learners, especially so in some high-level languages, e.g., Java. Researchers like Hmelo and Guzdial [1996] have suggested various shades of "glass-box" scaffolding that provide support, but in a transparent fashion. With such an approach, students would to be able to ignore extraneous details whenever it is advantageous to do so, but still have relatively simple access to all of the layers below, should they progress to the point that they are curious as to the mechanisms at work in those layers.

The RiTa tools attempt to provide such a glass-box environment in several ways. First, through the initial use of the Processing environment, a range of difficulties inherent to Java are immediately avoided: from path and classpath configuration; to separate compile and run steps; to the complexities of the 'main()' method; to the scoping details for classes, methods, and variables; to the distinction between static and non-static elements; all of which can be baffling to new users. In contrast, to run a simple Processing program (the ubiquitous "hello world", for example) one simply types in 'print("hello world");', and pushes the 'play' button. Like Processing,[66] RiTa is open-source, but in contrast, to further facilitate

[66] At the time of this writing the Processing source code is available only via the Subversion (SVN) concurrent versioning system.

transparency, the source for RiTa is included with every download of the library. Additionally, multiple types of documentation are made available, including tutorials, a user's guide, a simplified index of the most-used methods in the library, and a full javadoc-style reference for more advanced users. Early on, as students are switching to the Eclipse environment, a tutorial is presented on linking to library source files, so that a single key press on a method name immediately takes one to its definition, down from RiTa, to Processing, to the core Java libraries themselves.

3.6 Supporting Creativity: 'Productivity' and Beyond

In its 2003 report, entitled 'Beyond Productivity: Information Technology, Innovation and Creativity', the National Academy of Sciences [Mitchell et al. 2003] states that: "software tools must not only be available, but they must be objects of critical reflection… Artists and designers can do more with IT if they become deeply conversant with its capabilities and limitations." While it may be useful in the short run to provide students and artists with tools that accelerate their productivity, it will be far more transformative to teach them how to "think procedurally" in their practice. This is what inextricably links *creativity support* with both *computational thinking* and *procedural literacy*. Rather than simply accelerating the rate at which practitioners accomplish their traditional day-to-day tasks, CST researchers are beginning to provide artists with direct access to the power of procedural methods and computational practices.

The analogy of the hungry man and the fishing instructor is apt in the case of classes designed both to teach core computer science concepts and engage the creativity of artists. Rather than giving away fish, we might instead teach people how to fish, or better yet, especially in the case of artists who have historically created a majority of their own tools,

help them learn how to build fishing rods of their own. In this way, they gain not only insight into the nature of fishing, but into the nature of mechanics, gravity, tension, fluid and aero-dynamics, and physics itself. Facilitating this iterative model of understanding requires two rather simple steps: a) we need tools that can be critically examined by their users; and b) we need users with enough basic understanding of computational processes that such access will be rewarding. For the first, we need to create and disseminate tools that not only are open-source, but that are transparent at multiple levels; with high-level descriptions and examples, careful documentation, and open access to all the layers of authoring that comprise the tool. For the second, we need to teach artists basic procedural literacy, without which no amount of transparency can be leveraged. RiTa attempts to be a tool that simultaneously addresses both of these objectives.

3.6.1 RiTa in the Classroom

Having designed and implemented the initial version of RiTa according to the above pedagogical constraints (supporting serendipity, inverted-use and misuse; scaffolding and transparency), our next task was to create a workshop-style context for a diverse group of students that supported our goals (to provide a test environment for RiTa, and to enable students to create art via computational methods while developing basic programming and computer science skills). Throughout this process, we attempted to maintain close links (see Tew [2005] above) between different elements of the course context: from overarching philosophy, to toolkit functionalities, to student assignments and projects, to supplementary reading materials, to reflective writing and thinking exercises. As concrete examples, we will look at two of the "mini-projects" assigned during the middle section of the course (these varied somewhat from section-to-section), following a range of introductory materials and preceding students' larger final projects.

3.6.1.1 Assignment I: Context-free Grammars

As mentioned briefly above, one of the first assignments each semester involved the use of context-free grammars, also called "recursive transition networks" in the literature on generation. To begin this mini-section, students were presented with a simple RiTa program that produced, in response to user input, a variety of haikus from a grammar. After some minimal explanation of the program code and basic grammar syntax, students were asked to modify the sketch in class to produce a work that was in some way personal and expressive. In addition to the textual material, students were asked to consider all material aspects of their piece (font, text-size, color, motion, etc.) as means for advancing the communicative potential of their work. The intent here was to facilitate students' experimentation *prior* to their having a full understanding of the mechanisms at work in the grammar framework.

As was the case throughout the workshop, this programmatic experiment was followed by four distinct but mutually-informing elements: readings, writings, discussion and critique. Readings for this assignment began with a brief introduction to the form of the haiku and several specific examples (both traditional and computer-generated). Additional readings were assigned (from Charles Hartman [1996], Selmer and Bringsjord [2000], and Funkhouser [2006], to the Dada Engine, by Andrew C. Bulhak [2009]) that discussed various grammar-based text-generation strategies. Works utilizing these techniques[67] were subsequently demonstrated and discussed in class. Additionally a number of related concepts were presented: Chomsky's [1965] theory of universal grammars (and some counter-arguments[68]), center-embedding (to demonstrate the infinite nature of English sentences), and the idea of

---

[67] Castigilia, Utterback and Wardrip Fruin's "Talking Cure" [2002] (http://www.hyperfiction.org/talkingcure/index.html) is one such example.

[68] See [Everett 2005].

recursive grammar rules. In a subsequent class, students were asked to present their programs as they would in a traditional art or writing workshop, picking and choosing versions of their output to present, and reflecting on the efficacy of the technique in their practice. As was the case throughout, each iteration of each project, with source code, was posted on the PDAL wiki for future review.

As critiques progressed, certain technical frustrations emerged, often involving perceived technical limitations—e.g., "how can I make sure no lines repeat?"—which were then discussed during class time. More complex strategies for using context-free grammars (CFGs) were presented (employing the probabilistic rules in RiGrammar), as well as specific effects that could *not* be produced with CFGs, but instead required additional expressivity (as provided by the callback mechanism in the RiGrammar library). This led into a discussion of Chomsky's language hierarchy and how (context-sensitive) human languages differed from (context-free) computer languages.

Some more complex example grammars were discussed (nested parentheses as one example), and presentations on students' completed mini-projects were scheduled for the following week. The example haiku sketch using RiGrammar was re-presented with both recursive and probabilistic rules to encourage experimentation and once again, students were advised to consider all aspects of their piece as means for achieving desired effects. Somewhat interestingly, a number of students seemed "hung-up" on the overloaded term "grammar", thinking that the so-called "production rules" in their grammar should map directly to English grammar rules. To break this misconception, a short (~20 line) grammar-based sketch was presented that recursively draws a tree-like structure (see screenshot in Figure 16), in a similar fashion to a Lindemayer System (or L-System.)

Figure 16: A simple L-System implemented via a (recursive) context-free grammar.

By the following week, all students were able to produce working prototypes that not only demonstrated a basic understanding of the concepts, but also expressed, in some way, a personal style or voice. Each student presented their work for critique and was asked, as per standard practice in traditional creative writing workshops, not to explain their intentions until all other comments had been heard. Finally, students wrote in class about their experiences with grammars, after which followed a lively discussion in which several key concepts were quickly raised, most notably *layering* and *authorial intent*.[69] Regarding the former, it seemed evident, following the presentations, that multiple layers of "text" existed in all of the presented works.

---

[69] A full discussion of these topics is beyond the scope of this work. For more information see [Howe and Soderman, 2009] .

One technical feature that enabled this realization was the fact that the source code and grammar files were published (by default) along with the finished piece. As such, students were able to easily read at two additional levels below the "surface" (the program's runtime output) during critiques. In fact it seemed that students naturally perceived the authored text to be some amalgam of the three layers they had written (surface, grammar, and code) rather than simply the "output" to the screen which varied from run to run of the programs. Further, when questioned, students seemed to feel that *all* the software tools employed in the process: RiTa, Processing, Java, the browser, and even the operating system, could be conceived as potential texts for analysis and artistic critique. According to student assessments of the experience, each tool contributed to the final output in varying degrees, generally contingent on the tool's distance from the surface, or conversely, their distance from the binary machine code:

> Theoretically the grammar exists independently as a piece of writing which, like most language on a page, does nothing, but I know when I look at a grammar file [I know[that it has a programmatic counterpart. It goes somewhere; it does something; it will change based on a set of rules it defines. Which is to say, I don't know how I feel when I look at a grammar. I consider it to be part of the text of a piece, but I also can't separate it from its use value, which is not to say it's an inferior piece of writing, just that I'm not sure how to categorize it or interpret it *as itself*. For me it opens a door into thinking about what poems I may write as a combination both of variables I can control and those I can't; that if they're planned-out in certain ways, and trusting to some larger structure I didn't contribute to in others, they may do work I never anticipated.

The other important issue that emerged in the context of this assignment was the tension between so-called "randomness" and authorial intent. Here the question of delegating the author's "choice" to the computer seemed of specific concern to writers, while those self-identifying as artists in the class seemed to accept this as a matter of course (perhaps due to

the fact that process-based work has been a central idea in visual and aural media for some time.

Interestingly however, writers cited a freeing aspect to this "loss" of control, specifically some version of plausible deniability that enabled them to take more risks in their writing than they might have otherwise:

> The poetics of randomness fascinate me. I felt that I really wanted to give away the power I have as an author to the computer. [PDAL student, 2008]

and

> I felt free when I was writing my grammar, freer than I've felt in writing anything for a while, because I knew it wasn't going to appear in the order I wrote it. [PDAL student, 2007]

The general argument here appears to be that because the computer has, in some sense, made the choices (e.g., the grammar probabilistically chooses between the various right-hand rules for a production,) the writer is somehow less responsible when a uninteresting word combination was generated, and the writer is thus able to blame the computer's "choice" rather than some inherent failing in themselves.

Further, students made the interesting point that they tended to write words or phrases rule-by-rule, rather than imagining and writing for a complete output sequence. So, because of their natural tendency to make associations with the surrounding text when writing, there was a specific writerly level present in the grammars, but often *absent* from the text itself. In this light, close readings of the grammars themselves proved interesting, as if the set of possible lexical choices for each rule constituted a self-contained poem. Below follows an excerpt from one student's grammar file (the '|' symbols signify OR,) so that for any run of the piece, only one of the lines below would appear, yet relationships *between* the

lines are clear, and at times, quite interesting, a fact often remarked upon by those students with a background in art, literary or media theory.

> and how he came to know the truth |
> is a dubious gesture, and one not to be trusted |
> as the boys' voices reached down through the floorboards |
> douglas fir we think, though we can't be sure |
> in concentric circles | with a passion typified by adolescent lust |
> and if it rang, did we pick it up? | they knew each other almost certainly |
> if the door had opened, it would have showed something completely...

As one student commented,

> The enjoyment and generativity is to be found in the unexpected matches between words and phrases, between things I would have never thought to put together, but which manage to make a kind of sense beyond themselves. I like the idea of a piece of writing creating its own meaning; it means I don't have to try to do it. I can write in a field as opposed to writing with a perceived trajectory.

3.6.1.2 Assignment II: Language Models

As an introduction to the statistical approaches that have become so popular in recent natural language research, PDAL students were asked to do a mini-project that employed some type of probabilistic language model. To provide some brief background, a language model can be described as a statistical model used to analyze or generate elements of a text via previously learned probabilities. The term originated from the probabilistic models of language generation developed for automatic speech recognition systems in the early 1980's [Jelinek 1997]. The history of language models, however, goes back to beginning of the twentieth century, when Andrei Markov used language models to model letter sequences in works of Russian literature (see Chapter 4, Prior Work). Another famous application of this technique, also discussed in the prior work chapter, was Claude Shannon's models of letter

sequences and word sequences, which he employed to create the theoretical foundations of information theory [Shannon 1949]. In the 1990's, language models were applied as a general tool for a range of natural language processing applications, including part-of-speech tagging, machine translation, and optical character recognition (OCR) and have since become increasingly popular in a range of information retrieval research [Hiemstra 2009].

N-grams (also referred to as Markov chains, after Andrei Markov mentioned above) are an example of a specific type of statistical model in which the next item in a sequence is predicted based upon the frequency of that sequence in a set of inputs (see Prior Work for examples). For example, we might estimate the probability of a given sequence of letters, words, or phrases given the probabilities of that sequence appearing in a specific input set, e.g., the New York Times articles for the year, or the poetry of Emily Dickinson. The '*n*' in *n*-grams refers to the number of words in each sequence that is considered as part of our estimate. If *n*=1 we have a *unigram* model in which the probabilities of a single letter, for example, are estimated based solely on the frequency of that letter in the input. In a bigram model, where *n*=2, we would estimate the likelihood of a specific two-letter sequence, "Qu" for example, based on its frequency in the input as compared to all other two-letter sequences in the input. In contrast to the grammars mentioned in the previous section, statistical approaches like *n*-grams tend to require less knowledge of the specific languages and texts involved. In the grammars mentioned above, all rules (for syntax, morphology, semantics, etc.) must be created by the author, but in statistical approaches, these rules are represented "implicitly" in the model.

This presents an interesting tradeoff between the two paradigms. With statistical models, relatively little prior knowledge of the genre is required in the creation of a model, while a "deep" understanding of the structure of the texts in question is generally needed to

construct a convincing grammar for it. Statistical models tend to work best with large amounts of textual input, as the statistical properties grow stronger (to a point) with a larger input set. Grammars, on the other hand can be represented compactly and require no external input set for operation, a fact that recommends them for web-based projects, at least prior to the addition of the RiTaServer module. Further, the two approaches differ in the fact that after creating a "successful" grammar, according to whatever definition one might use, one is often then in a position to generalize about the genre that is being examined. If the generated outputs are representative of the genre, then the grammar has generally captured some, if not all, of the relevant rules on which that genre operates. On the other hand, the success of a statistical model tends to tell us more about the particular statistical approach (and possibly the chosen input texts) than anything about the genre in question.

From a pedagogical perspective this is an important distinction. While grammars may require more knowledge and more effort to create, they tend to result in more post-process knowledge about the type of language under examination. Statistical models, on the other hand, require less overhead, tend to generate more "surprising" results, and teach students more about the "process" (statistical analysis) than the "product" text, which may or may not represent some existing genre. In the context of PDAL and RiTa, both approaches are important, and further, they shed light upon one another, especially when presented back-to-back, as was generally the case in the class. Lastly, they served as a conceptual link into a deeper understanding of the history of artificial intelligence, illustrating the historical distinction between "neat" and "scruffy" approaches [Wardrip-Fruin 2006] that has long been a subject of debate in the field. Of course the ideal approach is often, depending on the project, a mix of the neat and the scruffy. This avenue is made easily available through the

RiTa tools as when, for example, grammars make calls to statistical models to generate some text item that obeys both the grammatical specification and the statistical distributions.

While PDAL students were required to create a project employing some type of probabilistic language model, it was left to each student to decide what type of model to use. The RiTa toolkit provides a range of objects that leverage probabilistic techniques, including *n*-gram-based generators (RiMarkov), Keyword-In-Context models (RiKWICker), and "maximum entropy" parsers and taggers (RiPosTagger, RiChunker and RiParser.) However, as *n*-grams featured prominently in one of the course texts, Charles Hartman's *Virtual Muse*, they (with support via the RiMarkov object) were chosen by a majority of students for their projects. As was customary, the topic was presented with a range of reading, coding, writing and critiquing exercises. The primary readings for this section were Hartman's *Virtual Muse* and Eric Elshtain's writings on the Gnoetry engine [Elshtain 2006]. One of the unique contributions of the former text is its rigorous discussion of *n* -gram-based generation in a literary context, which Hartman used extensively in his work *Monologues of Soul and Body*. Elshtain's text presents an interesting set of extensions to the basic *n*-gram technique.

In addition to this reading, several artworks employing *n*-grams were presented and critiqued, including 'Talking Cure' [Castigilia et al. 2002], and several of the poetic texts generated using Elshtain's Gnoetry[70] engine. The introduction began with a very simple sketch (see Appendix: Examples) that generated new texts from a combined set of Wittgenstein and Kafka pieces, allowing users to interactively experiment with different *n* values and immediately see the affect on the output. In response to questions concerning the working of the program, a very basic introduction/review of probability was presented and

[70] For more information, visit the Gnoetry Engine at http://www.beardofbees.com/gnoetry.html.

students directed to create a similar "mash-up" of their own to be performed in the subsequent class. In the following class, students presented their programs to the class for critique. A discussion ensued about limitations of the approach and additional features of the RiTa tools were presented for those who had not discovered them on their own, either via the examples or documentation. These included weighting of inputs, constraints on repetition, custom tokenization, feature compression (case, synonyms, etc.), and the literary extension methods discussed in the technical section, (e.g., getCompletions(), getProbabilities(), and getProbabilityMap()), which allow for some degree of interactive control of the model during generation. In addition, several hybrid approaches were presented, including the use of RiMarkov on other features (provided by the RiAnalyzer) such as Part-of-Speech. Another approach was the combined use of a grammar for higher-level structures (e.g., section, paragraph or even sentence) with the use of the statistical models for lower-level tasks, such as word-selection, and semantic consistency. Several interesting (and publishable) projects resulted from this set of work, including a full-scale dramatic play generator complete with lighting and stage directions, etc. (see Appendix: Student Project Gallery).

Concurrently, students were given a coding assignment to build a letter-level concordance for an input text (presented alternatively as a *unigram* model, or a *n*-gram model with n=1). This assignment led naturally into a first lesson on data structures, as students quickly realized that to store the information required, some type of dictionary-like structure would be required. In this dictionary, given some "key" (often a single letter), one could obtain the number of times it appeared in the input, without scanning the input each time. The pros and cons of various approaches were discussed, from Lists, to Hashtables, to Arrays indexed on character code, in terms of both efficiency and storage space. By the end of the session, most students seemed comfortable with the assignment itself, and, more importantly,

with evaluating (at least at a very basic level) the different data representation alternatives, a central topic in many introductory computer science courses. Further, this discussion was motivated directly by the materials and problems of the given context (creative text generation) rather that by an abstract problem designed to match the topic to be taught.

Lastly, a maximum entropy approach, as represented in the tagging, chunking, and parsing components of RiTa was presented. Although not all students possessed the required mathematical experience for full comprehension, it was a clear "next step" after the previous assignment, especially in the case of those students intending to continue on to further computer science courses. As was generally the case with RiTa tools, there were (at least) two levels of understanding which enabled use of the tools: a base level concerning what the various methods could do, and a deeper understanding of the inner workings of the components (always available for inspection). While the latter allowed users to take full advantage of the functionalities and extensions in the RiTa objects, it was not required for those wishing to make only simple use of the tools. One of the part-of-speech taggers included with RiTa used a maximum entropy approach[71] and was presented as an example for this section. Not only was part-of-speech tagging an easily understood example, it led (as soon as substitutions were attempted) directly into chunking, where students wished to replace noun-phrases rather than simple nouns. This led into a discussion of parse-trees and strategies for parsing (bottom-up, top-down, chart-strategies, etc.) and additionally made clear the presence of recursive syntactic structures, e.g. noun-phrases containing other noun-phrases, and the need for (in some cases) a full-fledged parser (RiParser) rather than a simple chunker (RiChunker). The presence of such structures initiated a discussion of recursion

[71] The other was a faster, but less accurate, transformation-based tagger following Brill [1992].

itself, and a few simple recursive algorithms were presented in combination with a more general presentation of the kind of problem for which a recursive solution is recommended, e.g., one containing sub-problems with a similar structure to the initial problem. Rather than the typical Fibonacci or Factorization examples, recursively structured English sentences were presented as examples.

3.6.1.3 Integrating Computational Thinking

In the assignments above we can see how, in addition to the "artistic" elements required to make a compelling work of digital art, an impressive range of core computational ideas arise naturally as a result of the material at hand. Through just the two relatively simple examples presented above, grammars and *n*-grams, the student will have been introduced to an impressive number of key computer science concepts, many of which likely to be taught in an introductory CS sequence; from finite-state automata to context-free grammars and the language hierarchy; from elementary data structures to hashtables, to the construction of parse trees; from regular expressions to recursion. Rather than appearing to students as arbitrary additions to the "real" topic at hand, the relevance of these ideas is immediately apparent in a course that focuses on creative language-driven programming projects.

3.6.2 Final Student Projects in PDAL

Typically, final projects involved both novel combinations of existing RiTa components and the creation of custom code to extend or augment existing functionality. In several cases, such extensions have been added to the core RiTa library, with authors receiving credit on the RiTa website. Because RiTa provides a core subset of the potentially daunting infrastructure generally required for language-based artworks, students are comparatively free to explore a variety of topics through both individual and collaborative

projects and encouraged to focus on those aspects of their work they find most engaging.[72] Further, the open and community-oriented nature of the programming environment provided students with a sense that their projects were meaningful contributions, both to other RiTa users and to the larger digital art community, as opposed to just "exercises". Several students in the courses expressed interest in incorporating elements of their projects back into the library, while others exhibited and published their projects in well-respected galleries and journals for digital literature. Still others expressed interest in creating their own libraries to support creativity for specific domains. RiTa itself (the code for which was often discussed in class) provided a helpful example in these cases of what such a library might look like, with well thought-out interfaces, clean code structure, and thorough documentation.

In addition to source code and functioning programs, careful documentation of all aspects of students' process was stressed, both as a method to evaluate their development in a critical/reflective manner, and to provide examples and resources for others in the RiTa/Processing community. Finally, in the last meeting of each semester, students presented their work in a live setting to a larger audience of practicing artists, researchers and educators. Both the breadth and depth of these projects has been astonishing and is discussed further in Chapter 5 (Evaluation) as one measure of the tools ability to support a wide range of creative work. For those interested, several dozen of these projects are available in the project gallery located on the RiTa website at http://www.rednoise.org/rita/rita_gallery.htm.

[72] See the Chapter 5: Evaluation, for a further discussion of this claim.

CHAPTER 4: PRIOR WORK

4.1 Introduction

This chapter presents a summary of prior work that has influenced the theorization, design, implementation, and deployment of the RiTa toolkit. While the range of this work is broad, this is due to the fact that little, if any, existing research has targeted our specific goals With this in mind, we focus here on related research and practice whose goals overlap with at least one of the explicit goals of this project, as laid out in the introduction. While the brief discussion of prior work in the opening chapter present the current state of creativity support for the literary arts, this work in this section represents more direct influences on our research, and falls into the following primary categories:

- Programmatic Educational Environments
    - for Natural-Language Processing (NLTK, SimpleNLG, etc.)
    - for Procedural Literacy and Interactive Art (Processing, Max/MSP, etc.)
- Computer Science and Literary Art (Strachey, Shannon, Weizenbaum, Bringsjord)
- Computationally-augmented Literary Experiments: Tools and Practice

For reasons of economy, several areas of active research relating only tangentially to RiTa, specifically tools for interactive fiction (e.g. Inform or Curveship), games with narrative and/or conversational elements (e.g. Facade), non-programmatic support tools (e.g. script-writing aids like Dramatica, argumentative-writing aids like Euclid, and collaborative writing tools like EtherPad), are not addressed here, though pointers to resources on these topics have been included where applicable.

4.2 Programmatic Educational Environments

Computer Science (CS) researchers have created a wide range of programmatic libraries that attempt to aggregate the range of tasks required to perform high-level natural language research. Recent years have also seen some interest in adapting this approach to the classroom, by providing tools that specifically address pedagogical issues that arise as new computer science students attempt to work with natural language. Similarly there has been impressive growth in both the number and quality of libraries and environments designed specifically for computational artists. As the RiTa toolkit bridges these two research areas, this section presents a review of important works in each that have informed our approach.

4.2.1 For Natural Language Processing

As one might expect, there is significant overlap in the functionality required for computational literature and those designed for Natural Language Processing (NLP) and/or Natural Language Generation (NLG). Over the years, a number of general-purpose NLP toolkits have been created for research purposes including the CMU-Cambridge Statistical Language Modeling Toolkit [Clarkson and Rosenfeld 1997], the EMU Speech Database System [Harrington and Cassidy 1999], the General Architecture for Text Engineering (GATE) [Bontcheva et al., 2002], the Maxent Package for Maximum Entropy Models (Baldridge et al., 2002b), the Annotation Graph Toolkit (AGT) [Maeda et al. 2001], and MontyLingua [Liu 2004]. Although some of these resources have educational applications and have been used in teaching, their development has not been motivated primarily by pedagogical needs or requirements. On the other hand, there have been several toolkits, designed with less experienced users and/or students in mind, that directly address educational concerns. While not directed at either an art or literary context, or (with the exception of SimpleNLG) even generation specifically, these tools have at least indirectly

informed the development of RiTa, and as such warrant at least brief coverage here. Specifically we will look at two below: the Natural Language Tool Kit (or NLTK) by Loper and Bird, and the SimpleNLG package by Ehud Reiter.

4.2.1.1 NLTK

The Natural Language ToolKit (or NTLK) was designed by Edward Loper and Steven Bird as an end-to-end solution for new students in the field of Natural Language Processing. The first paper on the NTLK was published in 2002 and provides the following description:

> NLTK, the Natural Language Toolkit, is a suite of open source program modules, tutorials and problem sets, providing ready-to-use computational linguistics courseware. NLTK covers symbolic and statistical natural language processing, and is interfaced to annotated corpora. Students augment and replace existing components, learn structured programming by example, and manipulate sophisticated models from the outset. [Bird and Loper, 2002]

In its first iteration, the NLTK, implemented as a command line tool written in Python, provided modules for the following tasks: Parsing, Chunking, Tagging, Finite State Automata, Type Checking, Visualization, and Text Classification. While the functionality of the NLTK overlaps only slightly with that of RiTa, its emphasis on pedagogical concerns, specifically the difficulties inherent in teaching language processing to new students in a classroom context is directly relevant. See Figure 17 below for a detailed comparison of the various components and functionalities.

In fact, several of the design considerations listed in chapter two were first discovered and/or confirmed in the NLTK research. Although none of these can be claimed to have originated with the authors of the NLTK (or RiTa), the two do share the following explicit design requirements: Ease of Use, Consistency, Extensibility, Documentation, Simplicity,

Modularity, Comprehensiveness, and Efficiency. Additionally, the NLTK authors' concern with in-class demonstration, open-source deployment, exhaustive documentation, and online tutorials all provided inspiration for the development of RiTa.

4.2.1.2 SimpleNLG

The SimpleNLG package, written by Ehud Reiter, was developed contemporaneously with RiTa and focused on providing a simple interface for a range of Natural Language Generation (NLG) tasks. The website[73] provides the following information: "SimpleNLG is a simple Java class library which does basic NLG lexicalisation and realisation; it is primarily designed for data-to-text applications."

Reiter [2009] describes the system as:

> "[A] realisation engine which grew out of recent experiences in building large-scale data-to-text NL G systems, whose goal is to summarise large volumes of numeric and symbolic data (Reiter, 2007). Sublanguage requirements and efficiency are important considerations in such systems. Although meeting these requirements was the initial motivation behind SimpleNLG, it has since been developed into an engine with significant coverage of English syntax and morphology, while at the same time providing a simple AP I that offers users direct programmatic control over the realisation process" [Reiter, 09]

Here, as with the NLTK, there are a few areas of overlap between the realization process employed in SimpleNLG and the core RiTa functionality, specifically stemming, noun-pluralization, and verb conjugation. In fact, verb-conjugation and noun-pluralization in both packages use the often cited morphological rules specified in Minnen [2001]. Like RiTa, SimpleNLG places all functions in users' direct programmatic control, and appears to place high-importance on documentation and tutorials, as does the NLTK. Reiter says that

---

[73] http://www.csd.abdn.ac.uk/~ereiter/simplenlg/

> the simplicity of use of SimpleNLG is reflected in its community of users. The currently available public distribution, has been used by several groups for three main purposes: (a) as a front-end to NLG systems in projects where realisation is not the primary research focus; (b) as a simple natural language component in user interfaces for other kinds of systems, by researchers who do not work in NLG proper; (c) as a teaching tool in advanced undergraduate and postgraduate courses on Natural Language Processing. [Reiter, 09]

Further, as perhaps expected from its name, the syntax is both simple and consistent, with most method calls exposed as setters, e.g., setSubject() or setInterrogative() as below:

```
Phrase s1 = new SPhraseSpec('leave');
s1.setTense(PAST);
s1.setObject(new NPPhraseSpec('the', 'house'));

Phrase s2 = new StringPhraseSpec('the boys');
s1.setSubject(s2);
s1.setInterrogative(true);

// OUTPUTS -> 'Did the boys leave home?'

s1.setInterrogative(WHERE, OBJECT);

// OUTPUTS -> 'Where did the boys leave?'
```

While SimpleNLG represents a useful solution to the specific tasks for which it was designed, there are limitations one notices when it is applied to a literary context. First, employing a so-called "pipeline" architecture, it requires all parts of a sentence to be known before realization, as opposed to the varieties of 'incremental' generation allowed by other systems [Manurung 2003]. The standard pipeline architecture (as presented in Reiter and Dale) presents other problems for the realm of literature. Its limitations become apparent in

applications that include specific goals for the resulting surface text, from the inclusion of specific literary features, to the inclusion of idiomatic constructions, even to aiming toward a particular phrase, paragraph, or document length. Manurung [2003], summarizes the issue by describing how the elements of text generation are, in the case of poetry, not independent at all:

> Making choices in one aspect can preclude the possibility of choices in other aspects. When these decisions are made by the separate modules in the pipeline architecture ... the resulting texts may be suboptimal, and in the worst case, the system may fail to generate a text at all. This problem has been identified in Meteer (1991) as the 'generation gap', in Kantrowitz and Bates (1992) as 'talking oneself into a corner', and also in Eddy (2002), who notes that it is not easy to determine what effect a decision taken at an early stage in the pipeline will have at the surface, and decisions taken at one stage may preclude at a later stage a choice which results in a more desirable surface form.

Additionally, SimpleNLG provides no support for custom features or constraints—literary or otherwise—at the level of the phrase or sentence. Thus, the realization of a sentence in which formal and semantic elements are intended to receive equal weight in choosing a final surface realization can be problematic. In fairness however, we should note again that this is not a use for which SimpleNLG was intended.

4.2.2 For Computational Art

In recent years there has been impressive growth in both the number and quality of libraries and environments designed specifically for computational artists. While once artists and art students were forced to work either in medium-specific tools (e.g. Photoshop or ProTools) which are only very minimally programmatic, or so-called “general-purpose” languages (e.g., C/C++ or Basic), which provide little specific support for art practice, they are now faced with a wide range of languages, libraries, and environments designed specifically for the art context. While none of those listed below target the domain of

literature or even language processing—nearly all target visual media, and if not visual, then aural media—their approach to providing programmatic supports for the arts has directly influenced the development of the RiTa tools to varying degrees. Tools that have been peripherally important to RiTa have been included in the comparison table below for the purpose of presenting an overview of the domain. The most direct influence, however, has been from the Processing environment, with which RiTa optionally integrates, and which is discussed in detail below.

4.2.2.1 The Processing Environment

Processing is an open-source programming library, development environment, and online community that has promoted software literacy within the visual arts since 2001. Initially created to serve as a software sketchbook and to teach fundamentals of computer programming within a visual context, Processing quickly developed into a tool for creating finished professional work as well. It is used by students, artists, designers, researchers, and hobbyists for learning, prototyping, and production. It was created to teach fundamentals of computer programming within a visual context and to serve as a software sketchbook and professional production tool for programming images, animation, and interactions.

Processing is a free alternative to proprietary software tools with expensive licenses, making it accessible to schools and individual students. Its open-source status encourages the community participation and collaboration that is vital to its growth. Contributors share programs, contribute code, answer questions in the discussion forum, and build libraries to extend the possibilities of the software. The Processing community has written over seventy libraries to facilitate computer vision, data visualization, music, networking, and electronics. Processing was founded by Ben Fry and Casey Reas in 2001 while both were John Maeda's

students at the MIT Media Lab and was developed as a direct descendant of Maeda's "Design By Numbers" language [Maeda 2001].

Like Processing itself, which can be used directly as a Java library, RiTa is only loosely coupled with the Processing environment. It can be used with or without the Processing IDE and libraries.[74] Both the Processing libraries and development environment have been integral elements in the practical implementation of the PDAL class, and in the development of the RiTa tools. Having offered a discussion in this section of current tools available to students and programmers, the next section will present a chronological survey of procedural writing methods, experiments, and tools in the contexts of both computer science and Literary Arts.

[74] The one exception is the text display capabilities of RiTa which (as of v80) require Processing's core.jar archive.

	Processing	Scratch	Flash/Flex	VVVV	Silverlight	GA2	Open Frameworks	Max/MSP/ Jitter	Hackety Hack
Real-time Environment	No	Yes	No	Yes	No	No	No	Yes	No
Novel Interface / Environment	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes
Natural Language-Focus	No	No	No	No	No	No	No	No	No
Strength of Community (1-5)	4	4	5	2	4	1	3	5	2
Web / Browser Executable	Yes	No	Yes	No	?	No	No	No	No
Educational Focus	Yes	Yes	No*	No	No	Yes	No	Yes	Yes
Arts / Creativity Supporting	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Free / Open Source	Yes	No	No	No	No	No	Yes	No	Yes
Platforms (Win/Mac/*nix)	All	Win/Mac	Win/Mac	Win	Win	All	All	No	All
Core Language	Java	SmallTalk / Graphical	Action-Script	Custom / Graphical	.NET	Matlab	C++	Custom / Graphical	Ruby

	NLTK	RiTa	RAPUNSEL	Alice
Yes	Yes*	No*	Yes	Yes
Yes	No	No	Yes	Yes
No	Yes	Yes	No	No
4	2	4*	1	4
No	No	No	No	No
Yes	Yes	Yes	Yes	Yes
Yes	Yes	Yes	Yes	Yes
Yes	No	Yes	Yes	Yes
Win/Mac	All	All	Win/Mac	Win/Mac
Yes	All	All	Java*	Yes

Figure 17: Comparison of educational programming environments

4.3 Literary-focused Computer Science

This section presents a series of literary experiments in mainstream computer and information science that have led to important research results. Our goal here is to introduce the reader to early efforts that provide the underpinnings for subsequent work in the field, including, but not limited to, the RiTa project. In Christopher Strachey's "Love Letter Generator", we find a very early experiment with computer programs designed to produce creative literary outputs, "composed" without intervention by a human author. Claude Shannon later focuses on how literary language can be represented via probabilistic models. Joseph Weizenbaum uses simple transformational rules on natural language to create the first conversational agent, and Selmer Bringsjord argues convincingly for a new "Turing Test" based on literary creativity, and implements a state-of-the art story generation system designed to pass it. What all the researchers in this section have in common is that they use what have become core computer science methods to engage with the literary, a methodology that has significantly influenced the construction of the RiTa tools.

4.3.1 Christopher Strachey

The first known literary experiment with a modern computer was Christopher Strachey's "Love Letter Generator", written for the Manchester Mark I and completed in 1952. Much of the important research on Strachey, and his work with the seminal computer scientist Alan Turing, was done only recently by Noah Wardrip-Fruin in his 2006 dissertation at Brown University, in which he has argued rather convincingly that Strachey's piece is actually the first work of digital art of any kind, a not insignificant fact when considering the importance of literary experiments to the burgeoning fields of both digital art and computer science. In this light, Strachey is a figure of some importance, not least of which due to the insight his story provides into Turing's early career. And of course, without Turing, Strachey would have not had access to the computer on which the "Love Letter Generator" was programmed. Their association story begins in 1951 when Strachey, still only a teacher at the Harrow School, asked Turing for a copy of the manual for the Mark I computer he had recently written. Turing's somewhat surprising acceptance of the request facilitated Strachey's sudden appearance in the world of modern computing [Wardrip-Fruin 2006].

Strachey visited Manchester for the first time in July of 1951 and discussed his ideas for a checkers-playing program with Turing. These ideas impressed Turing, who suggested that the problem of making the machine simulate itself using interpretive trace routines would also be interesting Strachey, taken with this suggestion, wrote such a program. As Strachey's biographer Martin Campbell-Kelly writes:

> The final trace program was some 1000 instructions long, by far the longest program that had yet been written for the machine, although Strachey was unaware of this. Some weeks later he visited Manchester for a second time to try out the program. He arrived in the evening, and after a "typical high-speed high-pitched" introduction from Turing, he was left to it. By the

> morning, the program was mostly working, and it finished with a characteristic flourish by playing the national anthem on the "hooter".[75]

This was a considerable achievement for an unknown amateur. He had written, within a single session, the longest program for the Mark I thus far. As Martin Campbell-Kelly asserts, his reputation was virtually established overnight. By June 1952 Strachey had completed his responsibilities at the school and officially began full-time computing work as an employee of the NRDC. That summer he developed, probably with some input from others including Turing, the Mark I program that y, it is unlikely that Strachey had digital art of the sort we see today in mind. For one thing, there would have been little thought of an audience. As with his checkers-playing program, the love letter generator could be reported to a wider public, but only experienced directly by a small audience of his fellow computing researchers. At the same time, it certainly was not an official assignment from the NRDC, but rather, like many creative computing projects, undertaken for enjoyment and to see what could be learned. Not everyone in Strachey's small audience enjoyed it equally [Wardrip-Fruin 2006].

Turing biographer Andrew Hodges [2000] reports that "Those doing real men's jobs on the computer, concerned with optics or aerodynamics, thought this silly, but ... it greatly amused Alan and Christopher". Looking at the program's output today, we can understand why Turing and Strachey's colleagues thought the project silly. In 1954, Strachey published the following article in the art journal *Encounter* (immediately following texts by William Faulkner and P. G. Wodehouse):

---

[75] For further information, see http://grandtextauto.org/2005/08/01/christopher-strachey-first-digital-artist/.

Darling Sweetheart

You are my avid fellow feeling. My affection curiously clings to your passionate wish. My liking yearns for your heart. You are my wistful sympathy: my tender liking.

Yours beautifully

M. U. C.

Clearly there are a number of shortcomings apparent in the "letter" above, but, like many creative computing experiments, such outputs are not the most interesting part of the project, but rather the vast combinatory potential that such programs afford. It is likely that this unpredictability and procedural "expansion" is part of what amused Strachey and Turing, though, as has often been the case, the process itself is now lost to us and only sample outputs remain. As Wardrip-Fruin argues [2006], this is a problem for work in digital literature and art generally. We tend to focus on surface output, and as a result our understandings do not include the "hidden" procedural elements that work to create such outputs. He goes on to say:

> [Process and data] are integral parts of computational works, and to fail to consider them means we only see digital literature from the audience's quite partial perspective. The fundamental fact about digital works is that they operate, that they are in process, and only once our interpretations begin to grapple with the specifics of these operations will we be practicing a method commensurate with our objects of study.

This is a problem that RiTa, with its focus on interpretation at multiple levels (from outputs, to source code, to intermediate 'texts' like templates and grammar files) addresses directly.

Here is another example from Strachey's Encounter article:

Honey Dear

My sympathetic affection beautifully attracts your affectionate enthusiasm.

You are my loving adoration: my breathless adoration. My fellow

feeling breathlessly hopes for your dear eagerness. My lovesick adoration

cherishes your avid ardour.

Yours wistfully

M. U. C.

"M. U. C." is of course a reference to the Manchester University Computer, or Mark I, who "plays" the part of a love letter author by carrying out the process outlined in the article:

> Apart from the beginning and the ending of the letters, there are only two basic types of sentence. The first is "My, (adj.), (noun), (adv.), (verb) your, (adj.), (noun)." There are lists of appropriate adjectives, nouns, adverbs, and verbs from which the blanks are filled in at random. There is also a further random choice as to whether or not the adjectives and adverb are included at all. The second type is simply "You are my, (adj.), (noun)," and in this case the adjective is always present. There is a random choice of which type of sentence is to be used, but if there are two consecutive sentences of the second type, the first ends with a colon (unfortunately the teleprinter of the computer had no comma) and the initial "You are" of the second is omitted. The letter starts with two words chosen from the special lists; there are then five sentences of one of the two basic types, and the letter ends "Yours, (adv.) M. U. C."

Words in parenthesis are randomly substituted according to the following word-lists:

Adjectives:
anxious, wistful, curious, craving, covetous, ...

Nouns:
desire, wish, fancy, liking, love, fondness, ...

Adverbs:
anxiously, wistfully, curiously, covetously, ...

Verbs:
desires, wishes, longs for, hopes for, likes, ...

Letter-Start:
dear, darling, honey, jewel, love, duck, moppet, sweetheart

Table 5: Examples from the Love Letter Generator's Input Data.

As we can see in the data presented above, Strachey's generator involves a high degree of combinatorial choice, with a choice among many options provided for nearly every word. It is at once a literary work and a work of computer science exploiting non-determinism over a clearly-defined search space to achieve a specific (creative) effect. These days, process-oriented works of digital literature tend to use algorithms of a complexity that dwarfs that of those described in Strachey's generator, but the importance of the context—computers that can emulate human creative processes—has in no way diminished.

4.3.2 Claude Shannon

Claude Shannon was a seminal thinker in both computer science and information theory—he arguably invented the latter[76]—whose work laid the foundations for the statistical methods we find in such widespread use today. Though the experiment described below does not target literary outputs as directly as Strachey's "Love Letter Generator", it presents another example of fruitful synthesis between the literary context and computer science research. Working from already-constructed literary texts Shannon created probabilistic models that could approximate various properties of the text being examined. Like the other researchers presented in this section, the context for Shannon's experiments was based not only in natural language, but specifically in literature. To quote Golumb [2002], "it is no exaggeration to refer to Claude Shannon as the 'father of the information age', and his intellectual achievement as one of the greatest of the twentieth century".

One of Shannon's important early contributions was his work with *n*-grams, based on the notion of Markov models as invented by Andre Markov in 1906. Like Shannon, Markov himself used literary language, specifically the novels of Pushkin, as a means of analyzing the general statistical properties of natural language. The basic question the two researchers considered was, given any sequence of English letters or words, what is the likelihood of the occurrence of the next letter or word? Shannon published the answer to his question in a paper entitled "A Mathematical Theory of Communication" [Shannon 1949] where he formalized, among other things, the notion of *n*-grams. To illustrate the concept, he provided six sample "messages" from the English alphabet. In the first message, each of the alphabet's 26 letters and the space appear with equal probability:

XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGHYD...

[76] See [Golomb 2002].

In the second, the symbols appear with frequencies weighted by how commonly they appear in English text (i.e., “E” is more likely than “W”):

OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA TH EEI ALHENHTTPA ...

Shannon’s remaining four sample messages were produced with a somewhat different process. In the third, symbols appear based on the frequencies with which *sets of two* of the symbols appear in English. That is to say, after one letter is recorded, the next is chosen in a manner weighted by how commonly different letters follow the just-recorded letter. So, for example, in generating the previous message it is only that “E” is a more common letter than “U”. However, in creating the third message, it is also important that if a pair of letters begins with “Q” it is *more* likely that the complete pair will be “QU” than “QE”. Taking the frequencies of pairs into account in this manner means paying attention to the frequencies of “bigrams” [Wardrip-Fruin 2006]. The sample message created in this way begins:

ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D...

In the fourth, symbols appear based on the frequencies with which sets of three of the symbols appear in English. This is called a “trigram”, with the choice of the next letter weighted by the frequencies with which various letters follow the set of two just recorded. Shannon’s sample message begins:

IN NO IST LAT WHEY CRATICT FROURE BIRS GROCID PONDENOME...

In the fifth, the unit is moved from letters to words. In this message, words appear in a manner weighted by their frequency in English:

REPRESENTING AND SPEEDILY IS AN GOOD APT OR COME...

Finally, in the sixth sample message, words are chosen based on the frequency with which pairs of words appear in English. This, again, like the technique of choosing based on pairs of

letters, is called a “bigram” technique, but here applied to words. The complete final message Shannon used was:

> THE HEAD AND IN FRONTAL ATTACK ON AN ENGLISH WRITER
>
> THAT THE CHARACTER OF THIS POINT IS THEREFORE ANOTHER
>
> METHOD FOR THE LETTERS THAT THE TIME OF WHO
>
> EVER TOLD THE PROBLEM FOR AN UNEXPECTED.

Shannon describes the progression as follows:

> The resemblance to ordinary English text increases quite noticeably at each of the above steps. Note that these samples have reasonably good structure out to about twice the range that is taken into account in their construction. Thus in (3) the statistical process insures reasonable text for two-letter sequences, but four-letter sequences from the sample can usually be fitted [sic] into good sentences. In (6) sequences of four or more words can easily be placed in sentences without unusual or strained constructions. The particular sequence of ten words “attack on an English writer that the character of this” is not at all unreasonable. It appears then that a sufficiently complex stochastic process will give a satisfactory representation of a discrete source. [Shannon 1949]

To create the last four messages, Shannon [1949] using ordinary books, which he explains:

> To construct (3) for example, one opens a book at random and selects a letter at random on the page. This letter is recorded. The book is then opened to another page and one reads until this letter is encountered. The succeeding letter is then recorded. Turning to another page this second letter is searched for and the succeeding letter recorded, etc. A similar process was used for (4), (5) and (6). It would be interesting if further approximations could be constructed, but the labor involved becomes enormous at the next stage.

That is to say that the last sample message (which begins with a sequence that sounds surprisingly coherent) was generated by opening a book to a random page, writing down a random word, opening the book again, reading until the just-recorded word was found, writing down the following word, opening the book again, reading until that second word is found, writing down the following word, and so on.

So why does the 6th message sound so coherent, if all Shannon did was repeatedly open a book at random? As Wardrip-Fruin points out [2006], the answer is that Shannon is operating on the assumption that ordinary books reflect (more or less) the frequencies of letters and words in English. And this, in turn, is why we find a passage of unexpected coherence in the last sample message. Because choosing an ordinary book is actually choosing a piece of highly-shaped textual data (shaped, for example, by the frequencies of words and sequences of words in the book's language, the topic of the book, the author's particular style.[77] When these "statistical" measures are aggregated for one or more input texts, we have what is called a language model.[78] Further, any new text that we choose to generate from this model will produce results that reflect the data we used to create it.

Shannon was not working in a literary or artistic context, but his ideas were applied by practitioners for years to come, even though it is only in relatively recent time that

---

[77] This type of analysis is often referred to as "computational stylistics".

[78] The term language models originates from probabilistic models of language generation developed for automatic speech recognition systems in the early 1980's [9]. Speech recognition systems use a language model to complement the results of the acoustic model which models the relation between words (or parts of words called phonemes) and the acoustic signal. The history of language models, however, goes back to beginning of the 20th century when Andrei Markov used language models (Markov models) to model letter sequences in works of Russian literature [19133]. Another famous application of language models are Claude Shannon's models of letter sequences and word sequences, which he used to illustrate the implications of coding and information theory [17]. In the 1990's language models were applied as a general tool for several natural language processing applications, such as part-of-speech tagging, machine translation, and optical character recognition. Language models were applied to information retrieval by a number of research groups in the late 1990's [4, 7, 14, 15]. They since became quite popular in information retrieval research.

common computers could handle word-level *n*-gram models for non-trivial inputs. Here we see how an implementation's efficiency (e.g., with memory, with disk access, with processing power) and the need, or lack thereof, for certain kinds of correctness (e.g., in statistical distribution) can be determining factors in whether a technique is applicable to an artistic context. In fact, the modern availability of computing power has made practical, and shown the power of, a whole area of research, specifically that of "statistical" natural language, that previously was out of reach. Shannon's initial experiments manually generating sentences that approximated those found in literature were the first step in this trajectory. Parenthetically, it is interesting to note that the first application of Markov models was also linguistic and literary, modeling letter sequences in Pushkin's poem "Eugene Onegin", though this was presented from a mathematical, rather than communication-oriented, perspective [Markov 1913].

For almost three decades after the publication of Shannon's paper, the techniques that he outlined for text generation were barely explored. While their application to analysis was somewhat further investigated, there seems to have been little interest in generating text with it, and as Wardrip-Fruin states [2006], it appears to have had no actual literary use until much later. In part this may have been due to the effort involved in building texts by hand, as Shannon did, combined with the fact that even for severely restricted versions of the method, computers that could handle the amount of data generated in accurate statistical models (at least when employing the most obvious approaches to the problem) were unavailable until the

1970s. How *n*-grams were later employed in actual literary practice[79] is explored in below in our discussion of Charles O. Hartman's "Monologues of Body and Soul".

4.3.3 Joseph Weizenbaum

Using Markov models in the language experiments above gave researchers access to new tools for the analysis and generation of natural language and laid the groundwork for the range of more sophisticated probabilistic methods to come. Joseph Weizenbaum approached natural language processing from a more elementary perspective, but with no less interesting results. Weizenbaum was a professor of computer science at MIT when, in 1966, he published a comparatively simple program called ELIZA, which used pattern-matching and transformation to simulate human conversation (programs like this are now generally called "chatterbots", or "chatbots").

Driven by a script named DOCTOR, it engaged a human user in a conversation with a simulated psychologist. Weizenbaum modeled the program's conversational style after Rogerian therapy, which uses open-ended questions to encourage patients to communicate more effectively with their therapists. The results were surprisingly engaging, as Eliza/Doctor used simple rules to turn the audience member's typewritten statements back to the user in the form of open-ended questions and prompts to talk further. Weizenbaum was shocked that his program was taken seriously by many users, who would open their hearts to it. In his article, "From Computer Power and Human Reason", Weizenbaum [1976] describes how quickly and deeply people became emotionally involved with the computer program, e.g., taking offence when he asked to view the transcripts, saying it was an invasion of their

[79] In addition to widespread literary use, n-Grams have also been used extensively in musical composition. One example is the contemporary Austrian composer Karlheinz Essl, who reassembled a Bach violin sonata via n-grams, calling it "Bach sausage."

privacy, even asking him to leave the room while they were working with the program, a phenomenon Weizenbaum found quite disconcerting.[80]

An example of Weizenbaum's man-machine conversation included a chat between a simulated therapist and a patient. A segment follows:

"Men are all alike," the patient states.

"In what way," asks the computer-therapist?

"They're always bugging us about something."

"Can you think of a specific example?"

"Well, my boyfriend made me come here."

"Your boyfriend made you come here?"

"He says I'm depressed much of the time."

"I'm sorry to hear you are depressed."

"It's true."

At this point in the conversation when he tested it in his office, Weizenbaum's secretary asked him to leave, because the conversation was getting too personal [Wallace 2009].

Eliza/DOCTOR was considered by many to be a forerunner of the "thinking machines" trumpeted by the press and members of the AI community, computers able to simulate human cognitive processes. Interestingly it was Weizenbaum himself who most strongly argued, in his book *Computer Power and Human Reason* [1976], against this interpretation, explaining the limits of computers, and arguing that any anthropomorphic view of computers represents a reduction of the human being. David Gardner refers to

---

[80] When Weizenbaum informed his secretary that he, of course, had access to the logs of all the conversations, she reacted with outrage at this invasion of her privacy" [Wallace 2009].

Weizenbaum as "the brilliant MIT researcher who threw water on some of the wildest predictions about computers as 'thinking machines'".[81]

The system had a tremendous impact on a number of subfields in computer science: from natural language processing to artificial intelligence; from interactive narratives to conversational agents, the relationship of computing and psychotherapy, the ethical uses of computers, and computer gaming, to name just a few. Janet Murray identifies Eliza/Doctor as the "moment in the history of the computer that demonstrated its representational and narrative power with the same startling immediacy as the Lumieres' train did for the motion picture camera." She calls Weizenbaum "the earliest, and still perhaps the premier, literary artist in the computer medium" [Murray 1997]. While not all writers would be prepared to recognize Eliza/Doctor as literature, most can accept (as Wardrip-Fruin [2006] argues) the idea that presenting a character through conversation with the audience guided by previously-authored texts and rules, rather than through recitation of unvarying text, has the potential to be literary.

4.3.4 Selmer Bringsjord

Perhaps less well know than the previous researchers, Selmer Bringsjord (1958-) is the current director of the Rensselaer Artificial Intelligence and Reasoning (RAIR) Laboratory and a professor of computer science, as well as a Professor of Philosophy, Logic, and Cognitive Science. Bringsjord is perhaps best-known for his meta-level proofs of contentious issues in computer science, e.g., his modal argument using analog computation to show that P=NP [Bringsjord and Taylor 2005]. Of particular interest here is his refutation of

---

[81] See: http://www.informationweek.com/news/global-cio/showArticle.jhtml?articleID=206903443. Accessed 7/01/09.

the Church-Turing thesis via what he refers to as "literary creativity". While the specifics of the proof, and the arguments of his many critics, detailed in his book, *Artificial Intelligence and Literary Creativity (*AILI), with David Ferrucci [2000] are beyond the scope of this research, his reasoning for selecting this specific domain is relevant. He argues, citing research by a range of scholars that "literary creativity" may represent the best measure of intelligence at our disposal. In fact, he argues for a literary alternative to the Turing test. He writes:

> Though the Turing test is currently out of reach of the smartest of our machines, there may be a simpler way of deciding between the strong and weak forms of AI – one that highlights creativity…. The test I propose is simple: Can a machine tell a story?

He goes on to describe his test in more detail.

> But what would the story game look like? In the story game, we would give both the computer and a master human storytellers relatively simple human sentence, say "Gregor woke to find his abdomen was as hard as a shell, and that where his right arm had been, there now wiggled a tentacle." Both players must then fashion a story designed to be truly interesting, the more literary in nature – in terms of rich characterization, lack of predictability, and interesting language, the better. We could then have a human judge the stories so that, as in the Turing Test, when such a judge cannot tell which response is coming from the mechanical muse and which is from the human, we say that the machine has won the game. [2000]

As he suggests that the creation of a novel is so far beyond the capabilities of today's AI techniques that its existence "simply can't be conceived", he restricts the test described above to 500 words. Then, in an attempt to answer the question, he sets out, with the help of Ferrucci and senior scientists at IBM's Watson Research Center, to build a system, called "Brutus", that generates short fiction within that constraint. Their efforts make up the majority of the chapters of AILI, by the end of which they have created Brutus.1, an instantiation of the more generic Brutus architecture specializing in the literary theme of

'Betrayal', for which he provides a definition in formal logic, and operating over a narrow range of character types; an ontology that includes Professor, Students, Dissertations, Classes, etc. After 7 years of the 10-year project, he writes, "though I expect to make headway… first-rate story-telling will always be the sole province of human masters" [1998]. Whether or not he has successfully refuted the Church-Turing thesis in the process, or even provided a more accurate version of the Turing Test via his 'story-telling game', his choice of the literary context for his experiments again demonstrates its unique characteristics and continued utility in computer science research.

4.4 Procedural Writing: Tools And Practice

Having presented (in section 4.2) an overview of educational tools for computer science students and writers wishing to explore computational methods, and (in section 4.3) a survey of research by computer scientists addressing literary language, we presents here, in rough chronological order, a range of procedural writing experiments undertaken by practicing artists. Unlike Strachey and Shannon, for instance, who did not use creative writing as their starting point, the work that follows is integrally tied to literary production. Theo Lutz, for example, used stochastic methods to generate poetry, while Brion Gysin leveraged combinatory techniques to create new works, and Nanni Balestrini constructed poems by procedurally 'mashing-up' a number of different texts. In addition to these examples of procedural techniques, we also mention a number of individual works (e.g. *House of Dust*), exhibitions (e.g. Cybernetic Serendipity), and programs (e.g. Auto-Beatnik) which further explore computational writing in the context of artistic practice. The section concludes with an investigation of the Dada and Oulipo movements who, though not always

utilizing computational tools, were committed to procedural methods that link them closely to the other work discussed[82].

Since long before the invention of computers, artists and creative writers have experimented with the use of *procedural techniques* in their art practice. As Florian Cramer [2005] puts it:

> Executable code existed centuries before the invention of the computer, in magic, Kabbalah, musical composition and experimental poetry. These practices are often neglected as a historical pretext of contemporary software culture and electronic arts. Above all, they link computations to a vast speculative imagination that encompasses art, language, technology, philosophy and religion.

As Cramer notes, it is unlikely a coincidence that the Gospel of John was one of the first texts manipulated in the early computational poetry experiments of Brion Gysin and William S. Burroughs [1978], discussed below [Funkhouser, 2007].

IN THE BEGINNING WAS THE WORD

IN THE BEGINNING WAS WORD THE

IN THE BEGINNING WORD THE WAS

IN THE BEGINNING WORD WAS THE

IN THE THE BEGINNING WAS WORD

IN THE THE BEGINNING WORD WAS

---

[82] Due to vast scope of this work, however, we are only able to touch on a handful of examples in the categories below. For a more detailed history of the topic, we recommend the full-length monographs, "Prehistoric Digital Poetry: An Archaeology of Forms" by C. Funkhouser [2007], and "Words Made Flesh: Code, Culture, Imagination" by F. Cramer [2005], both of which present a wealth of in-depth information on the topic

IN THE THE WAS BEGINNING WORD

IN THE THE WAS WORD BEGINNING

IN THE THE WORD BEGINNING WAS

IN THE THE WORD WAS BEGINNING

IN THE WAS BEGINNING THE WORD

IN THE WAS BEGINNING WORD THE

[. . .] [Burroughs [1978]

As one might expect, as soon as computer technologies became accessible to artists working with procedural methods, they were put to immediate use, sometimes in quite surprising and productive ways. This section presents a brief history of artistic experiments by those working in, or at the borders of, the "literary", with particular focus on those whose work led, directly or indirectly, to techniques employed in the RiTa toolkit.

4.4.1 Theo Lutz

An important early contributor to procedural literature was the programmer Theo Lutz, a student of Max Bense[83], who created a series of "stochastic poems" on a Zuse Z22 computer in 1959. Examples of this work, which applied the tools of mathematics and

[83] Max Bense (1910-90) was a professor of the philosophy of technology, scientific theory, and mathematical logic at the Technical University of Stuttgart and an important figure in early concrete and computer-aided poetry. In his research, he was devoted to creating an information theoretical foundation for aesthetics and to text produced with machines. The philosophy of visual poetry was to a considerable extent indebted to Bense. For more information, see http://www.medienkunstnetz.de/artist/bense/biography/.

calculation (i.e., logical structures) to produce language, were first published, with descriptions of the processes employed, in Bense's journal *AugenBlick* in an article entitled "Stochastic Texts" [Funkhouser 2007].

Working from Kafka's famous text, *The Castle*, Lutz created a database of sixteen subjects and sixteen titles. He then used the computer's random number generator to create random sequences from this database. Logical constants (gender, conjunction, etc.) were used to connect the lines in a readable syntax:

> Not every look is near. No village is late.
>
> A Castle is free. and every farmer is distant.
>
> Every stranger is distant. A day is late.
>
> Every house is dark. An eye is deep.
>
> Not every castle is old. Every day is old.
>
> Not every guest is furious. A church is narrow.
>
> No house is open and not every church is quiet.
>
> Not every eye is furious. No look is new.

Similar to Strachey's work, which we can consider to be *recombinant*, this piece combines fragments from a database to create a vast number of "original" compositions.

4.4.2 Brion Gysin

As noted above, Brion Gysin pursued combinatory techniques with the aid of computers as early as 1960 in a series of permutational pieces created in collaboration with the mathematician Ian Somerville. In the piece above, from The Gospel of John, the poem shuffles its words according to a formal algorithm that traverses a total of 720 permutations on an early Honeywell computer [Cramer 2005]. The critical anthology *Brion Gysin: Tuning*

in to the Multimedia Age presents several other examples of computer-generated permutation poems, programmed to appear in block formation. One of these poems, also from an easily recognizable source, is presented in Funkhouser [2007]:

I AM THAT I AM

I THAT AM I AM

I AM I THAT AM

I I AM THAT AM

I THAT I AM AM

I I THAT AM AM

I AM THAT AM I

I THAT AM AM I

I AM AM THAT I

I AM AM THAT I

I THAT AM AM I

I AM THAT AM I

I AM I AM THAT

I I AM AM THAT.[84]

---

[84] Although the programming details are not available; alternate versions of the poem, in which the words appear with a different sort of arrangement, are included in Williams's An Anthology of Concrete Poetry (1967) and in Kostelanetz's Text-Sound Texts (1980). As Funkhouser writes, "Gysin's permutation poetry imposes a pre-established pattern on the

4.4.3 Nanni Balestrini

Continuing chronologically, we come to Nanni Balestrini's computer-generated "Tape Mark" poems of 1961. In these works, he begins with a database containing texts by Lao Tzu's (*Tao Te Ching*), Paul Goldwin (*The Mystery of the Elevator*), and Michihito Hachiya (*Hiroshima Diary*) [Funkhouser 2007]. The program[85] then combines and constructs chains of words from these passages, ultimately portraying a scenario of nuclear disaster as present in Hachiya's text.

A range of these works were later presented at the 'Cybernetic Serendipity' Reinhardt, 1968] exhibition in 1968 (discussed below). The exhibition catalog includes this passage, which demonstrates a similar flavor of permutation as in Gysin:

> Hair between lips, they all return
>
> to their roots, in the blinding fireball
>
> I envision their return, until he moves his fingers

---

words in a phrase, so they appear in different orders until all possibilities have been exhausted. Thus, a poem made with a three-word phrase will be six lines long (3x2x1); a poem that begins with a five-line phrase, such as "I am that I am" will be one hundred twenty lines long (5x4x3x2x1). The availability of computer technology automated the process of randomizing these permutations." In "Cut-Ups Self-Explained," Gysin [Burroughs 1972] declares, "The permuted poems set the words spinning off on their own; echoing out as the words of a potent phrase are permuted into an expanding ripple of meanings which they did not seems to be capable of when they were struck and then stuck into that phrase" (Brion Gysin,154). [Funkhouser 2007].

[85] No specific information on the program used is available; it may have been Autocoder, which was the program used most commonly on the IBM 7070, or Fortran or RPG (Report Program Generator), which also ran on that machine [Funkhouser 2007].

slowly, and although things flourish

takes on the well known mushroom shape endeavoring

to grasp while the multitude of things comes into being.

In the blinding fireball I envisage

their return when it reaches the stratosphere while the multitude

of things comes into being, head pressed

on shoulder, thirty times brighter than the sun

they all return to their roots, hair

between lips takes on the well known mushroom shape.

Funkhouser [2007] writes:

> Though the shapes of each stanza are similar, Balestrini's programming method can generate a variety of poems (within finite parameters) from words composed for other purposes; the program, like Lutz's, devours multiple texts in order to produce combinatoric, permutation poems. The brief phrases in Balestrini's dictionary collect and intricately reconfigure excerpts from previously-written texts to generate hybridized, contemplative, and haunting expression.

4.4.4 Auto-Beatnik

In 1962, we find computerized literature reaching a much wider audience, as pieces generated by Auto-Beatnik were published by *Time Magazine.* Auto-Beatnik was created when R.M. Worthy and the engineers at Librascope,[86] concerned with the problem of

[86] The article notes the Librascope Division of General Precision Inc. in Glendale, California as the site of the computer. Hartman lists R.M. Worthy as author of the program, and reports that examples of Auto-Beatnik poems were published in a magazine called Horizons in 1962 (2). Only one "Auto-Beatnik" poem can be found on the WWW at present, "Poem No. 41: Insects;" see http://hem.fyristorg.com/stettin/hemsida/poem.html. Accessed 8/05/2004).

effective communication with machines in simple English, first "fed" an LGP 30 computer with thirty-two grammatical patterns and an 850-word vocabulary, allowing it to select at random from the words and patterns to form sentences. The results included "Roses" (shown below). Later Worthy shifted to the project to a more advanced RPC 4000, fed with a store of about 3,500 words and 128 sentence structures.

As Funkhouser describes: "the November issue of *Time Magazine*] featured a brief notice in the books section titled 'The Pocketa, Pocketa School,' introducing 'Auto-Beatnik' as a computer programmed to create poetry." This unattributed exposé prints and informally discusses two examples of "Auto-Beatnik" poems, and offers an interpretation of one of them. The syntax and thematic material in the poems published are a result of the narrow vocabulary (3,500 words and 128 simple sentence patterns) included in the program. An example:

Roses

Few fingers go like narrow laughs.
An ear won't keep few fishes,
Who is that rose in that blind house?
And all slim, gracious, blind planes are coming,
They cry badly along a rose,
To leap is stuffy, to crawl was tender.

While one notices several unconventional connections and phrases, none seem beyond the boundaries of poetic license. Traditional poetic properties such as action, description,

question, projection, and judgment are all present [Funkhouser 2007]. A second excerpt is similar to the first only in that it uses the simile "like" in the first line and that it contains unusual inflections:

> All girls sob like slow snows.
>
> Near a conch, that girl won't weep.
>
> Stumble, moan, go, this girl might sail
> on the desk.
>
> This girl is dumb and soft.

The program can emulate free verse as well, and aesthetically resembles, according to Funkhouser [2007], strains of so-called "Beatnik" poetry.

4.4.5 A House of Dust

In 1967, Alison Knowles and James Tenney created the famous *House of Dust*, a computer-aided poem/sculpture, which consisted of randomly generated quatrains in the following form: "a house of (list material), (list location) (list light source) (list inhabitants), leading once again to a vast set of possible compositions" [Funkhouser 2007]. The piece grew out of an informal "course" in FORTRAN that Tenney gave to several of his friends, including Philip Corner, Dick Higgins, Alison Knowles, Jackson Mac Low, Max Neuhaus, Nam June Paik, and Steve Reich. The poem was published by Verlag Gebruder Konig in Cologne in 1968 and later appeared at the Cybernetic Serendipity exhibition discussed below.

As Funkhouser [2007] describes, *House of Dust* is among the first poems featuring collocation via a programmed slot-system, and appears in several publications (each time

with a different title).[87] The poet-programmers in each instance establish four categories (materials, situations, lighting, and inhabitants) that determine the content of each line within a stanza. "Random meetings" of one element from each of the four categories generate a serial poem. Hundreds of "houses" can be created if all of the possibilities of this program are exhausted. The cumulative effect of the disparities in each of the poems, with their lightly absurdist expressions, begins to create a mental architecture for readers, though the output syntax is fixed and this work is repetitive. An example from *House of Dust* follows:

A HOUSE OF STEEL

IN A COLD, WINDY CLIMATE

USING ELECTRICITY

INHABITED BY NEGROES WEARING ALL COLORS

A HOUSE OF SAND

IN SOUTHERN FRANCE

USING ELECTRICITY

INHABITED BY VEGETARIANS

---

[87] The poem first appeared Cybernetic Serendipity as "The House," then in Dick Higgins's Computers for the Arts (1970) under the title "Proposition No. 2 for Emmett Williams," and later in FANTASTIC ARCHITECTURE as "A house of dust, computer poem." Computers for the Arts is a short and technical memoir in which Higgins introduces two works ("Hank and Mary" and "Proposition No. 2") to discus the "artificial language" Fortran as a vehicle for poetry. FANTASTIC ARCHITECTURE (an anthology edited by Higgins and Wolf Vostell, 1971) stems from Fluxus; the book mainly focuses on visual arts or architecture and contains commentary on art and society by Joseph Beuys, Raoul Hausmann, Franz Mon, Carolee Schneeman, and others.

A HOUSE OF PLASTIC

IN A PLACE WITH BOTH HEAVY RAIN AND BRIGHT SUN

USING CANDLES

INHABITED BY COLLECTORS OF ALL TYPES.

4.4.6 Cybernetic Serendipity

The now-famous Cybernetic Serendipity show, held at the ICA London in 1968, included several instances of computer-generated work. Curated by Jasia Reichardt, it was the first exhibition to attempt to demonstrate all aspects of computer-aided creative activity: art, music, poetry, dance, sculpture, and animation. The exhibition included robots, poetry, music and painting machines [Reichardt 1968].

In addition to new versions of Balestrini's Tape Mark poems [Balestrini 1996] and Knowles and Tenney's *House of Dust*, the exhibition featured "Computerized Japanese Haiku"[1968], by Margaret Masterman[88] and Robin McKinnon Wood. These pieces were written in the TRAC language, and featured nine slots, each of which could be filled with words from nine different databases. The show featured several poems created by this program, the following two of which are found in Funkhouser [2007]:

[88] Masterman was a member of the Cambridge Language Research Unit. She was not a poet, but rather a scholar who wrote profoundly on the growth of scientific knowledge (including a widely cited the essay "The Nature of a Paradigm"), and who became extremely interested in machine translation.]

1. Poem

eons deep in the ice

I paint all time in a whorl

bang the sludge has cracked

...

3. Poem

all green in the leaves

I smell dark pools in the trees

crash the moon has fled.

The program clearly outputs syntactically and mathematically correct poems that follow the haiku format. While the structure is fixed, the words selected by the database are variable, similar in style, notes Hartman, to the popular 'Mad Libs' game. Because of its length, strict formal constraints, and abstract nature, haikus have been a favorite testing-ground for computer-aided literary experiments since the creation of "Computerized Japanese Haiku". A Haiku generator using context-free grammars is included in the RiTa example programs included in each download.

4.4.7 John Cage

John Cage is another artist who experimented extensively with procedural methods (and later computer programming), and whose musical work was featured in the Cybernetic Serendipity exhibition. From the time of his 1953 *Music of Changes*, Cage employed the concept of "nonintentionality" in art, in which the artist no longer makes all decisions in her or his compositions, but instead lets chance control certain elements of the creative process.

Initially, Cage used the *I Ching*, "…the ancient Chinese oracle that uses chance operations to obtain the answer to a question," to accomplish tasks for him [Rettalack 1996]. Our discussion on Cage will be followed directly by an investigation of the work of Jackson Mac Low, an early and important procedural poet who also made use of nonintentionality in his work.

Cage experimented extensively with the aleatoric *I Ching* process, a "discipline" that involved formulating a question and then using coins to divine numbers that provided the answers. As Perloff [1991] writes in *Radical Artifice*, the process allowed Cage to "break with ego, with habit, with self-indulgence". He employed these methods as a writer as well, using the *I Ching* to structure poetic lectures and compose poems as early as in the late 1960s. Computers provided a natural vehicle for Cage's non-intentional work and he would become known a few years later for a unique form of poetry (often computerized) known as "mesostics" [Funkhouser 2007]. Cage describes the Mesostic form in *I-VI*: "Like acrostics[89], mesostics are written in the conventional way horizontally, but at the same time they follow a vertical rule, down the middle not down the edge as in an acrostic, a string which spells a word or name, not necessarily connected with what is being written, though it may be" [Cage 1973]. Beginning in 1984, Cage made use of the program Mesolist, written by Jim Rosenberg (who later emerged as a pioneering digital poet in his own right). Mesolist mechanically performed Cage's methodical "mesostic" treatment of texts. Until then, the tedious task of reading through a book, identifying words to be used, transcribing them, and restructuring them for the page had to be done manually. Cage also used a program called *IC*, which

[89] Acrostic poetry is a form in which the first letter of each line contributes to a word or phrase spelled vertically down the left-hand margin of the page.

emulates the calculations of the *I Ching*. His first computer-assisted works were presented as a series of 1988-89 lectures he made at Harvard University that are collected in the volume *I-VI*.

Funkhouser writes [2007] that "in *I-VI*, Cage employs elaborate processes and contributes significant input in generating his non-intentional work. He composes or identifies a source text that he uses as an 'oracle', and asks it what words to use for each letter of the (vertical) poem, a process that, he writes, "frees me from memory, taste, likes and dislikes" [Cage 1990]. Mesolist lists all words in the source that satisfy the mesostic rule, then IC selects words from the lists. The forty-five characters to the right and left of the chosen words in the original text ("wing words") are included, and Cage removes those he does not like [Cage 1990]. To prepare these lectures, Cage writes, "four hundred and eighty-seven disparate quotations have been put into fifteen files corresponding to the fifteen parts of [his text] *Composition in Retrospect*: method, structure, intention, discipline, notation, indeterminacy, interpenetration, imitation, devotion, circumstances, variable structure, non-understanding, contingency, inconsistency, and performance" [Cage 1990]. The source texts for the lectures included *Composition in Retrospect*, and a range of other sources, including writings by Henry David Thoreau, Ralph Waldo Emerson, L.C. Beckett, Fred Hoyle, Marshall McLuhan, Buckminster Fuller, Gene Youngblood, and from articles that had appeared in daily newspapers. After using other formulas to determine the number of mesostic strings per file and then to reduce the volume of source material, Cage produced lectures of roughly twenty-five hundred lines each. After giving the initial lectures, he realized the need to establish a simple notational system that would instruct him to take a breath when reading the work aloud (i.e., " '" [space apostrophe]), and where to stress

syllables that, "would not normally be stressed but should be" (i.e., bold typeface)." [Cage 1990] As a short example of one of Cage's mesostics, this passage from *Finnegan's Wake*:

> Just a whisk brisk sly spry spink spank sprint of a thing theresomere, saultering, Saltarella come to her own. I pity your oldself I was used to. Now a younger's there. Try not to be part! Be happy, dear ones! May I be wrong! For she'll be sweet for you as I was sweet when I came down out of me mother. My great blue bedroom, the air so quiet, scarce a cloud. In peace in silence.

becomes the following mesostic:

Just a whisk
Of
pitY
a Cloud
in pEace and silence

As Cage writes in the preface to an essay titled "Anarchy", these works "… do not make ordinary sense. They make nonsense…. If nonsense is found intolerable, think of my work as music, which is…a question of repetition and variation, variation itself being a form of repetition in which some things are changed and others are not" [Funkhouser 2007]. A similar sense of the language as musical is found in works by Mac Low and some by Hartman (with and without Kenner). Kenner himself describes the look of Cage's *Sentences* on the page as "Chant, therefore Voice" [Kenner 1995].

In her discussion of this work in *Radical Artifice*, Perloff observes that Cage prefers "to let us participate in the process whereby unfinished news items and bits of information…can be absorbed into the rhythms of individual consciousness; they remain discrete entities that we restructure according to our own predilections" [1994] . This focus on process is a trait that procedural and computer literature often manifests. The procedural

(or machinic) process is naturally highlighted (as discussed in the *lessons section)* and is cast against both our 'traditional' processes of writing and inevitably, as noted by Perloff in the fragment above, our thought processes, and even, the processes of consciousness itself. Cage's manipulation of chance elements mirrors the work of Jackson Mac Low (discussed below).

4.4.8 Jackson Mac Low

Mac Low, already an established print poet when he began experimenting with computer processes, was one of the first American poets to use chance methods in his pre-computer work[90]. These methods were not, however, the only means that Mac Low deployed in his larger attempt to solve the problem of producing so-called "egoless" poetry and music, a project particularly suited to the use of computational processes. Like Cage, Mac Low made use of the concept of *nonintentionality*, in which the artist no longer makes all decisions in her compositions, but instead lets chance control certain elements.

Mac Low created his first computer poems at the Los Angeles County Museum in the summer of 1969, using a PFR-3 programmable film reader designed for graphics applications connected to a DEC PDP-9 computer. Mac Low's program, he explains in *Representative Works: 1938-1985*, selected and combined words from a list of short messages he had composed. The program's database (the "message lists") and selection process allowed Mac Low to create "an indeterminate poem, of which each run of the printout is one of an

---

[90] In the late 1950s Mac Low started experimenting with chance methods using a copy of *One Million Random Digits, and 100,000 Normal Deviates*, created by the RAND Corporation for use in Monte Carlo algorithms.. He continued using this book in his work throughout the sixties and beyond [Mac Low(b) 1997].

indeterminable number of possible realizations" [Funkhouser 2007]. The following excerpt demonstrates the program's style:

...
THE BUSHES GROW.

THE INSECTS GATHER FOOD.
THE BIRDS GATHER FOOD.

THE PLANETS SHINE.
THE MOON SHINES.
THE SUN SHINES.

THE TREES DRINK. THE FUNGUSES DRINK.

THE MOSSES TURN TOWARD THE LIGHT.
THE FLOWERS TURN TOWARD THE LIGHT.
THE TREES TURN TOWARD THE LIGHT. (p214-15)

Mac Low also used deterministic methods to generate poetry whose form and content were not known in advance, but could be reproduced given identical initial conditions [Mac Low(b) 1997]. His approach to writing poetry was literally "experimental". The primary question he would ask was, "how can I achieve a certain effect?" as opposed to "what will happen if I implement this particular algorithm?" As his son, Mordecai-Mark notes [Mac Low(b) 1997], his approach was similar in many respects to that of an applied mathematician or computer scientist who studies both the general properties of algorithms and their adaptation to specific applications. He was not, of course, attempting to prove theorems or support particular theories with his experiments, but rather to empirically invent techniques of artistic production meeting certain criteria.

Mac Low began working with computers sooner even than many scientists. While he was an instructor at New York University in the late sixties, he took advantage of a free course in FORTRAN (one result was a short poem exploring output of a flawed program that showed gravity increasing as the height of the simulated fall increased). This was during the period when programs were written by hand on coding forms, which were then run through a computer in batch mode. In 1969 he was offered a fellowship by the Los Angeles County Museum of Art, to make so-called 'verbal' artworks at an IBM facility in Los Angeles, and he subsequently ended up working with Information International, Inc., "Triple-I", a company that went on to become one of the dominant computer graphics and printing companies. There he worked with John Hanson, the VP of programming, and his assistant Dean Anschultz, to write assembler language code for displaying poetry on a Tektronix vector graphics screen, and eventually for printing it out as well, using a programmable film reader driven by an early minicomputer [Mac Low(b) 1997].

Throughout the seventies, before the widespread availability of microcomputers, his work made more use of audio electronics than computers. He finally got his own machine in 1987 which “let loose the usual flood of manuals over his workspace” [Mac Low(b) 1997], but gradually gave him additional tools as well. He made at least one further attempt to learn to program (this time in the C language), but eventually came to rely on already written software.[91] In fact, much of Mac Low's computer-based work was realized in collaboration with Charles O. Hartman (see following section), author of *Virtual Muse: Experiments in Computer Poetry*. In the 1980s, Hartman automated many of Mac Low’s procedures for

[91] His use of software often involved creative ‘misuse’, as discussed in Chapter 3: Pedagogy, as when he completely filled the available space in his word processor's glossary with phrases drawn from work of and about Kurt Schwitters [Mac Low(b) 1997].

computer, including Mac Low's 'diastic' procedure, originally developed in 1963[92] and automated by Hartman in the late 1980's as part of the DIASTEXT program, which Mac Low began using in earnest in 1989.[93] These programs were used extensively in his *42 Merzgedichte In Memoriam Kurt Schwitters* (1994), a series of poems constructed by a variety of procedural methods, all employing found texts relating to the artist Kurt Schwitters. Mordecai-Mark describes the project:

> The first poem in *42 Merzgedichte in Memoriam Kurt Schwitter]* [Mac Low, 1994], was written by impulse-chance-selection from sources about and by Schwitters, the 2nd through the 30th by computer-aided chance operations, and the 31st through the 42nd by computer-automated diastic methods, which in some of the last *Merzgedichte* were supplemented by use of Hugh Kenner and Joseph O'Rourke's program TRAVESTY, which produces what Kenner calls "pseudo-texts," determined by letter-group frequencies in English [Mac Low 1998].

As Mac Low himself notes, the writing of several sections in this piece involved Kenner and O'Rourke's TRAVESTY program[94], based on Shannon's *n*-gram procedure.

---

[92] "Diastic" is a word coined by Jackson from the Greek words "dia" (through) and "stichos"(a line of writing, a verse) and is contrasted to "acrostic." (from "akros" (an extreme, such as the letter at the beginning or end of a verse line). "Acrostic" reading-through procedures draw words and other linguistic units from source texts by "spelling out" their titles with linguistic units that have the letters of the words in the titles as their initial letters. One reads through a source text and finds successively linguistic units spelling out the title as follows: the units spelling out individual words comprise single lines (often long ones) and the series of lines spelling out the whole title comprises a stanza. (The "asymmetries" are nonstanzaic but still partially acrostic.)

[93] He also experimented extensively with DIASTEX4, an improved version of the program which allows the user to choose and employ a separate index instead of using the whole source text as the index.

[94] TRAVESTY was written by literary critic and James Joyce expert Hugh Kenner wrote, in collaboration with the programmer Joseph O'Rourke, and its , a text recombination program

Kenner, a well-known literary theorist, was co-author (with Hartman), of *Sentences*, a volume of poems generated also with the help of TRAVESTY (For a full-description of TRAVESTY and its many incarnations, see Noah Wardrip-Fruin's dissertation "Expressive Processing" [2006]). Just as John Cage used the computer to facilitate work that he had previously performed manually, Hartman's program mechanically accomplished—with some variation and advancement—the procedural work that Mac Low had practiced for many years. According to Funkhouser [2007],

> once his attention became focused on Schwitters, Mac Low devised a computer program that would randomly select linguistic units that his initial poem for Schwitters ("Pieces O' Six: XXXII") stored in a "glossary" in Microsoft Word and process these fragments into what Mac Low describes as, "entirely new constellations". Over the course of two years, Mac Low implemented modifications to the program, its glossary, and made other adjustments to create a substantial body of poems.

With *31st Merzgedichte in Memoriam Kurt Schwitters* he began to incorporate DIASTEXT and TRAVESTY into his process; his uses of the programs were quite precise:

> I utilized these programs in different ways, employing earlier *Merzgedichte* as source texts: (1) For the *31st Merzgedichte*, I ran the *25th Merzgedichte* through DIASTEXT alone. (2) For the *32nd*, I ran the *4th* through DIASTEXT alone. (3) For the *33rd*, I ran the *2nd* through DIASTEX4 alone. (4) For the *34th*, I ran the *8th* through DIASTEX4 alone. (5) For the *35th*, I ran the *9th* through DIASTEX4 alone. (6) And for the *36th* through the *42nd*, I ran the *29th* first through TRAVESTY, asking for "low-order" output--i.e., scanning for sequences of very few characters, to insure the outputting predominantly of letter strings that aren't real words (pseudo-words), along

---

based on the Markov model. Dubbed Travesty, its source code was published in a 1984 issue of the popular computer magazine BYTE [Kenner & O'Rourke 1984]. For the algorithm, Kenner credited the "long-ago idea from the Father of Information Theory, Claude Shannon." The code was adapted in 1990 by Larry Wall, creator of the Perl programming language, and published as a programming example in the 1st edition of the book Programming Perl. The second edition of the same book featured examples of "Perl poetry"[Wall and Schwartz 1988]". 25 (see p. 94) .

> with a few real words, most of them embedded in pseudo-words--and then through DIASTEX4.

An excerpt from 29th Merzgedichte in Memoriam Kurt Schwitters:

> Tyll Eulenspiegel
>
> BucTurAPsor
>
> LIs lovewood revonTTed.
>
> Ander,
>
> IcTes.
>
> Ang Iners con
>
> LysAff brine
>
> Alsend brub HAgmes mencess kInces AumeIng

As has been noted repeatedly, Mac Low and Cage share at least some goals as artists, and their work is often discussed in similar ways, though Mac Low himself suggests that too strong a parallel has been drawn between the two:

> The thing is that there is too much pairing of John's and my work, despite our strong mutual regard. We're both concerned with intentionality and nonintentionality and started doing this sort of work from understandings of Buddhism, especially Zen & Kegon as taught by Daisetz Suzuki at Columbia University in the 40s and 50s. But I seldom used "pure" chance operations after 1960. My algorithmic work is often mistakenly thought to be chance-generated, as they say, and I too used to think it was "chance-generated" work, but I realized sometime in the 80s that the only chance involved is in the making of mistakes (and after a certain point--especially book publication--the mistakes must be accepted as integral to the works). Otherwise, whatever gets into the poetry is determined by the generative method & lies there waiting in the source texts. [Mac Low 1998]

4.4.9 Charles O. Hartman

Like many others of his generation, poet and digital literary theorist Charles O. Hartman was very much influenced by the work of Cage and Mac Low. Writing in *Virtual Muse: Experiments in Computer Poetry* [1996], he devotes almost an entire chapter to the relationship of chance, randomness and digital literature. He begins the third chapter by reminding us that "one of the Greek oracles, the sibyl at Cumae, used to write the separate words of her prophecies on leaves and then fling them out of the mouth of her cave. It was up to the supplicants to gather the leaves and make what order they could." He compares this with his early poetic experiment for the Sinclair ZX81, a BASIC program called RanLines that stored 20 lines in an internal array and then retrieved one randomly each time the user pressed a key [Hartman 1996].

One of the unique contributions of *Virtual Muse*, a required text in 'Programming for Digital Art and Literature' is the in-depth discussion of *n*-gram-based generation, which Hartman used extensively in his work *Monologues of Soul and Body*. The project was conceived during Hartman's experiments with the code for Travesty, the *n*-gram based generator originally published by Kenner and O'Rourke in Byte magazine [1984]. He says, "Here is language creating itself out of nothing, out of mere statistical noise. As we raise *n*, we can watch sense evolve and meaning stagger up onto its own miraculous feet." [Hartman 1996]

Interestingly, at the time of Hartman's experiments with Travesty, he was also working on a poem that took Alan Turing as a subject, specifically the famous 'Turing test' for machine intelligence. Hartman took the poem he had written and ran it through his version of TRAVESTY at eight different chain lengths: n=2 through n=9. The results proved to be evocative of his themes, as the computerized *n*-gram process appeared to build a sort of

sense that wove through his input texts which addressed human and computerized sense-making. But rather than simply use the n-gram outputs, he created a dialogue between his traditionally authored text and the TRAVESTY-generated text, the *soul* and the *body* of the title. As Hartman puts it, “In the computer output I saw the body constructing itself out of the material of soul, working step by step back to articulation and coherence. It's a very Idealist poem, and at the same time very Cartesian, and perhaps monstrous.” [Hartman 1996] Or, as Funkhouser [2007] puts it, “As the poem progresses, and the ‘body’ text is less abstract, the author succeeds in creating parallel monologues in which one (‘body’) borrows from the other.”

Again we find a multi-layered text emerging from the writers’ engagement with machine processes in Hartman’s monologues. There is Markov’s original formulation of the *n*-gram process; Shannon’s use (three decades later) of this abstract process for text generation; Bennett, Hayes, Kenner, and O'Rourke demonstrating (later still) that Shannon’s linguistic operations could create intriguing results with literary texts; Hartman's decision use this literary operation in his own work; and the innovations in implementation of Bennett, Hayes, Kenner and O'Rourke, and Hartman himself. Further, we have Hartman’s decisions concerning the input texts, not to mention the lines he wrote himself, and the compositional attention paid to how these were combined into the final piece. More recently, we see a range of practicing artists (John Cayley, for instance, who uses word-level bi-grams that he calls “collocations”) exerting influence on a new generation the series of student and artist projects using the word and sentence level *n*-gram facilities built into the RiTa toolkit (see the RiTa gallery for a number of examples of such work).

4.4.10 Dada

The definition of a famous Dada poetry generation process follows:

> To make a dadaist poem
> Take a newspaper.
> Take a pair of scissors.
> Choose an article as long as you are planning to make your poem.
> Cut out the article.
> Then cut out each of the words that make up this article and put them in a bag.
> Shake it gently.
> Then take out the scraps one after the other in the order in which they left the bag.
> Copy conscientiously.
> The poem will be like you.
> And here are you a writer, infinitely original and endowed with a sensibility that is charming though beyond the understanding of the vulgar.

This quotation, from Tristan Tzara's 1920 "Manifesto on feeble love and bitter love" [Motherwell 1981] is one of the most commonly reprinted texts from the Dada movement. While often associated with nihilism and portrayed as an 'anti-art' movement, Dada's most significant area of artistic innovation may have been in the creation of procedures like the one above [Wardrip-Fruin 2006]. In fact, as is the case with many examples of generative art, rather than to read individual works, an understanding of Dada involves, as Wardrip-Fruin argues, a reading of the processes employed, perhaps even independently of any examples of work at all.

An example of contemporary algorithmic implementations of Dada processes is Florian Cramer's 1998 reimplementation of Tzara's newspaper poem process as a web-based CGI script. The resulting web page performs a computationally-implemented version of Tzara's process (without a sack, a hand, or paper scraps of differing sizes) and allows the page's visitor to choose to use the text of a newspaper (from a pull down menu), the text of a

particular web page (there is a form element for entering page addresses), or any body of text the visitor may write or paste in (there is a form element for entering text) [Wardrip-Fruin 2006].

4.4.11 The Oulipo

Aphorismes by Marcel Bénabou [1996], appeared in ”Syntexts” in 1977. This generator, written in the APL language by Kenneth Iverson [1983], produces twenty-five aphorisms at a time in French, intended to reflect some version of profound insight. The program features a number of different slotted configurations such as, “X is in Y, not Z,” “A delivers B but C will deliver us from D,” “Q is the continuation of R by other means,” etc. A sample activation of the program in Funkhouser generates the following outputs:

> Beauty is the continuation of patience by other means.
>
> Hatred of ignorance is no other than the love of the rhythm.
>
> Science delivers evil, but what will deliver us from the present?
>
> Happiness is in horror, not in hatred.

He writes:

> The programming reflects tendencies that have existed since the outset of text-generation, though the output, due to the aptitude and choices of the programmer, also reflects a more complex effort in programming than found in many works. Beyond formulating the equations, the author must select appropriate materials to fill the slots. In a case such as “9” above, the closely connected variables call for setting up a range of language that will juxtapose effectively; the same principle is true, but less direct, in equations with more variables. The phrases are clear, grammatical aphorisms made with poetic language… Bénabou’s construction uses a finite amount of programming code to write endless aphorisms Funkhouser 2007] .

Bebabou was an original member of the important Oulipo group, and his precise formulation of the permutations to be performed are characteristic of their work. The Oulipo

group (Ouvroir de Litterature Potentielle, or Workshop for Potential Literature) represents perhaps the most direct precursor to the work presented here. Although much of the group's work did not directly involve computers, (this was later taken up exclusively by a splinter group called the Alamo), their focus on procedures, constraints, and transformation in the service of experimental literature directly informed much of the computational work to come. Founded in 1960 by Raymond Queneau and Francois Le Lionnais, the group came eventually to include an international roster of well-known writers and mathematicians, including Jean Lescure, Marcel Benabou, Harry Matthews, and Italo Calvino, each of which are introduced briefly below. For a full treatment of the group and their continuing work, there are a number of full-length books devoted to the subject [Lescure 1986; Mott 1986; Mathews and Brotchi 1998].

Unlike traditional writers (note the word potentielle or 'potential' in the group's title), Oulipans have been as concerned with abstract forms (often taking the shape of constraints on a text) as with instantiated instances of those forms. The Oulipan position is that all writing is constrained writing, but most writers work within constraints that are either a) traditional (e.g., the sonnet), or b) so ingrained as to be almost invisible (e.g., the 'realist' novel), or c) and perhaps worst, unknown to the writer (e.g., automatic writing of the type performed by the Surrealists) [Wardrip-Fruin 2006]. A primary aim of the Oulipo is to supplement or replace these potentially 'hidden' constraints with two types of new constraints: on one hand, the traditional constraints (e.g., poetic forms), once largely forgotten, which they hope to help revive; and on the other, constraints new to literature, many adopted from mathematics (and mathematical games) [Wardrip-Fruin 2006]. Georges Perec was an Oulipan who excelled at the use of both sorts of constraints; for example, writing a novel without the letter "e" (the "lipogram" is a constraint with a long tradition) or one structured by innovative application of

”the Graeco-Latin bi-square, the Knight's Tour, and a permuting schedule of obligations” [Mathews and Brotchie 1998].

Further, the ‘*potentielle’* of the Oulipo highlights its relation to computational literature in general and more specifically to the work presented here. The group has insisted, since its beginning, on the distinction between 'created creations' (creations crees) and 'creations that create' (creations creantes), focusing their attention on the latter, as does RiTa. Oulipan artists have been concerned not with literary works themselves, but with the procedures and structures capable of producing them. As Wardrip-Fruin writes:

> The Oulipo clearly has [procedures and structures] at the heart of it efforts. But this has not always been understood. After all, literary groups, of which the Oulipo is certainly one, generally produce texts, rather than structures and processes for others to use in creating texts. And certainly members of the Oulipo have produced remarkable texts using Oulipan procedures, as the novels of Perec, Calvino, and Matthews attest. But these procedures have also been used for literary works by non-Oulipans. Just as Perec made masterful use of the lipogram, so Gilbert Sorrentino, Christopher Middleton, and others have made remarkable use of 'N + 7.' But we must not, when impressed by these examples of procedures in use, allow this to cloud our vision of Oulipan potential. As Oulipo scholar Mark Wolff puts it, 'Writing is a derivative activity: the Oulipo pursue what we might call speculative or theoretical literature and leave the application of the constraints to practitioners who may (or may not) find their procedures useful' [2005].

In addition to their work on constraints for writing, there are also two other types of Oulipan proposals that Wardrip-Fruin notes. One type are formal procedures for the transformation of text via substitution, reduction, and permutation (either of one's own text or of found text[95]. The most famous Oulipan procedure of this sort, 'N + 7,' involves substituting all the nouns in text with the noun found seven dictionary entries later. Another is called the

---

[95] This technique is employed in a number of RiTa components, including the RiLiPo object which provides implementations of a range of Oulipian procedures like N+7, which uses the RiTa Lexicon as a “dictionary”.

"chimera"[96], wherein one produces a new text from four source texts. From the first text the nouns, verbs, and adjectives are removed. These are then replaced with the nouns taken in order from the second text, the verbs from the third text, and the adjectives from the fourth text. Similarly, "definitional literature" replaces a text's verbs, nouns, adjectives, and adverbs with their dictionary definitions, and can be applied recursively (replacing those words with their definitions, and so on). Interestingly, unlike Dada, Surrealism, or most other art movements of the 20th century, the Oulipo is one of the few that continues to practice to this day, nearly 50 years after its founding.

Raymond Queneau, one of the two Oulipan founders, contributed a number of important works to the Oulipan canon. He is also known for his characterization of the Oulipo as a group of "rats who build the labyrinth from which they plan to escape." [Mathews and Brotchie 1998]. One of his earliest works, a quintessential recombinant text, is his *Cent mille milliards de poems (CMMP), or One Hundred Thousand Billion Poems (*1961*),* which consists of ten 14-line sonnets. Due to the unique construction of the text—each poem is set on a page cut into fourteen strips that can be turned individually—the reader can construct alternate poems by reading the first line of any of the original sonnets, followed by the second line of any other sonnet, followed by the third line of any another, and so on. The work is constructed so that any reading of this sort produces a sonnet that functions syntactically, semantically, metrically, and in its rhyme scheme. The process of creating unique poems according to this procedure exposes a vast number of possibilities to the reader. When choosing which of the first line to read, there are ten possibilities. Next, one has ten choices for the second lines, giving one hundred (10 * 10) possibilities for each of the first

[96] The 'chimera' and several other related constraints are presented and described further in "the Oulipo Compendium" [Mathews and Brotchie, 1998].

two lines. After reading a second line, one chooses from the ten third lines, giving a thousand (100 * 10) total possibilities for the first three lines, and so on.

This type of work has been called “combinatorial literature” and Oulipo member Harry Mathews writes [Mathews and Brotchie 1998] of combinatorics whose: [object is] the domain of configurations, a configuration being the preset arrangement of a finite number of objects, whether it concerns “finite geometries, the placement of packages of various sizes in a drawer of limited space, or the order of predetermined words or sentences.” Arrangement, placement, order: because these are the materials of Oulipan combinatorial research. What results can generally be called rearrangement, replacement, or reordering, all subsumed by the generic term permutation

Italo Calvino**,** perhaps the most internationally renowned Oulipan, was invited in 1973 by IBM to write a story using one of their computers. In response to this challenge, Calvino had the protagonist of the story, entitled "The Burning of the Abominable House," use punchcards to feed data into a computer. But according to Calvino's wife, the limited computer access in Paris, where they were living at the time, meant that Calvino worked by “carrying out all the operations the computer was supposed to do himself”[Booth 1965]. Paper seems to have been the mechanism, not just the eventual interface, of Calvino's initial foray into computing [Montfort, 2004]

From the beginning however, Calvino was interested in science, and particularly so in its “cybernetic” developments. In “The Literary Machine”, for instance, he demonstrates a clear appreciation of the possibilities offered by the affordances of computer applications. But perhaps his most well-known work is *If on a Winter’s Night a Traveller*, a novel written in the second person. In this work, “you” are the main character, a bookseller journeying from bookstore to bookstore, attempting to locate a complete and correct version of the same book

you are reading, which is never found. Though the work does not follow such mechanistic procedures as “The Burning of the Abominable House”, it nonetheless represents and enacts a constant investigation of the author’s positionality in the constructed literary work.

Like Calvino, the Oulipo as a group maintained an evolving relationship toward the computer. Mark Wolff [2005] describes early Oulipan experiments with Computers this way:

> When the Oulipo formed in 1960, one of the first things they discussed was using computers to read and write literature. They communicated regularly with Dmitri Starynkevitch, a computer programmer who helped develop the IBM SEA CAB 500 computer. The relatively small size and low cost of the SEA CAB 500 along with its high-level programming language PAF (Programmation Automatique des Formules) provided the Oulipo with a precursor to the personal computer. Starynkevitch presented the Oulipo with an 'imaginary' telephone directory composed of realistic names and numbers generated by his computer. He also programmed the machine to compose sonnets from Queneau's Cent mille milliards de po`emes.

Yet few other works by the Oulipo directly employed computers, though it was discussed quite regularly at meetings. In fact, quite soon after the publications of Queneau's book CCMP, specialists had created computer versions of it. This led Paul Braffort and Jacques Roubaud in July 1981 to propose the creation of a new group named ALAMO (Atelier de Littèrature AssistÈe par la Mathèmatique et les Ordinateurs). For an in-depth discussion of the ideas associated with the ALAMO movement, see “Reading Potential: The Oulipo and the Meaning of Algorithms” [Wolff 2005].

CHAPTER 5: EVALUATION

"The strength of [contextual] methodologies lies in the fact that they place the study of creativity in a personal, social, societal, cultural and even an evolutionary context. The projects studied are defined by the practitioner and the research studies creativity using research based in actual practice."
[Mayer 1999]

5.1 Introduction

In this chapter we present a pilot evaluation designed to evaluate the efficacy of the RiTa tools in conjunction with the Programming for Digital Art & Literature (PDAL) course. From a combination of student feedback, assignments, and projects, in conjunction with our own perception of students' experiences with the tools, it was our hypothesis that RiTa and PDAL achieved several pedagogical goals: affecting student attitudes toward programming, enhancing their programming ability, and supporting their creativity both within the classroom and in the larger context of digital art. To assess this we administered a survey and programming quiz before and after one iteration of the semester-long course, and performed a descriptive analysis of students' final projects in the course. We also looked at a range of solicited and unsolicited feedback to further support the trends we have observed.

Before describing the specific methods employed, it is important to note that this research employs a qualitative methodology based upon constructivist[97] learning theories. As such, it includes the cultural and personal conditions of the researchers as well as that of the research population The process is that of a participating observer, who is not only an investigator in the field of research, but also one who is a part of that field According to this approach, the researcher is conscious of the field, conscious of herself, and not only affects

[97] See pedagogy section for further discussion of this term.

reality, but also helps to create it. The researcher does not come to the research *tabula rasa*, but possesses a range of previous attitudes regarding the field and domain of study. The experience of the researcher affects her views, attitudes, and actions, and her voice is present throughout the research work [Hammersley, Hazan, 2001; 1995; Phillips, 1990; Ragonis & Ben-Ari, 2005; Sabar Ben-Yehoshua 2001].

5.2 Goals

> *The first objective is to enhance the personal experience of the person who wants to be creative. The second is to look for ways to improve the outcomes and artifacts. The third objective is to support the improvement of process by providing tools that are designed with certain functional requirements in mind. [Hewett et al. 2005]*

Our goals in this pilot evaluation were three-fold. First, we wanted to measure changes in students' attitudes about programming before and after a semester-long exposure to the RiTa tools in the context of the Programming for Digital Art & Literature (PDAL) class. Second, we hoped to measure changes in their programming ability and assess whether these shifts correlated with any perceived attitudinal changes. Third, we hoped take an initial measure of the degree to which these tools served to support students' creativity in digital media. Our initial expectations were:

- that a semester-long exposure to the RiTa tools and PDAL would lead to a positive change in students' attitudes about programming;
- that a semester-long exposure to the RiTa tools and PDAL would lead to a positive change in both students' assessment of their programming ability and in their measureable programming ability;

- and that the RiTa tools would help students to be more creative programmers in the context of digital arts.

5.3 Methods

Given the exploratory nature of this pilot study, we collected data using a combination of pre-post surveys, a programming quiz, coding/evaluation of student projects, informal student feedback, and qualitative observations. Each of these methods attempted to compare the learning, behavior, and attitudes of students before and after using the RiTa tools as part of the semester-long Programming for Digital Art and Literature (PDAL) class.

5.4 Participants

A total of 15 students from Brown University and the Rhode Island School of Design (RISD) participated in the pilot evaluation. Students were a mix of undergraduates (from RISD and Brown) and graduates (RISD) from a variety of departments including Computer Science, Literary Arts, Visual Arts, and Modern Cultures & Media at Brown, and Digital+Media, Graphic Design, and Glass at RISD. From this group of students, 10 (0.67) were male and 5 (.33) female. 6 (or 0.40) were from RISD 's Digital+Media graduate program, 4 (0.27) were computer science majors, 2 (0.13) were from Literary Arts, while the remaining were either undecided or from one of the other departments above (<= 1 per department). In addition to the computer science (CS) students, who had each at least 1 prior CS course, 3 (0.27) of the non-CS students had taken a previous CS course. The average number of years programming for the class was 21.75/15=2.13 years. For CS students, this average was slightly higher at 3 years, while for non-CS students it was slightly lower at 1.79

years (19.75/11 or 10.75/9 without most/least=1.01)[98]. The course included several auditors who were not included in the survey results.

5.5 Observation Set 1: Surveys

5.5.1 Pre-Survey

Participants were given 20-25 minutes to complete a 32-question pre-survey (including basic demographic info and the pre-programming quiz discussed below). Pre-survey participation was optional and participants were instructed that their answers would in no affect their evaluation in the course. Students provided a unique identifier (generally, but not always, their student ID) to correlate pre and post results. Surveys were administered by a third party that was unaffiliated with the course and participants were instructed not to discuss the questions or to consult any references (e.g., laptops) during the allotted time.

5.5.2 Post-Survey

Participants were again given 20-25 minutes to complete a 28-question post-survey (including basic demographic info and the pre-programming quiz discussed below). Post-survey participation was optional and participants were instructed that their answers would in no affect their evaluation in the course. Students again provided the unique identifier that they used in the pre-tests to correlate their data. Surveys were administered by a third party that was unaffiliated with the course and participants were instructed not to discuss the questions or to consult any references (e.g., books, laptops, etc.) during the allotted time.

---

[98] This average is somewhat misleading due to a single student who entered a very high number. Omitting this student from the calculation, the average was 1.01 years of programming experience.

5.5.3 Attitudes Toward Programming

In order to measure participants' general attitudes toward programming we developed four questions, each of which were asked in both the pre and post surveys. In each of the three questions, a paired-samples *t*-test indicated differences in attitude toward programming between the pre and post surveys.

The first question addressed students' general sense of confidence in their programming and showed significant increases between pre- and post-test scores according to a paired-samples *t*-test, $t(11) = 3.677$, $p < .001$, $d = 1.109$. The second addressed students' assessment of their programming ability and also showed significant increases, $t(14) = 2.687$, $p < .01$, $d = 0.718$. The third question addressed the frequency with which students could "express their creative ideas" via programming and also showed a significant increase, $t(12) = 2.560$, $p < .001$, $d = .739$**.** The fourth question addressed students' feelings about programming and while not statistically significant did show a small increase between pre- and post-test scores While it would be premature to assert any causal relationship based on these findings, they do suggest a positive impact on students' attitudes toward programming, results which were similarly significant across gender, department, school, and previous technical background.

5.5.4 Knowledge and Self-efficacy

To measure students' confidence, knowledge and self-efficacy in a more granular fashion, eight additional questions were asked regarding knowledge and confidence on specific programming constructs and tools. On all of these dimensions, a paired-samples *t*-test indicated significant differences between the pre and post surveys (see Table XX below). While it is unclear that such differences would not have resulted under other circumstances, it does suggest that confidence and knowledge were significantly increased through the

semester long exposure to the tools and techniques. To compare students' self-assessments with their actual skills, programming ability is tested below (via Parsons' problems) for analogous gains.

	t-Score (n=14)	Mean difference (post-pre)	Standard Deviation	Cohen's D	Significance
Variables	2.9245	0.7143	0.9139	0.7816	p< .01
Conditionals	3.5393	0.8214	0.8684	0.9459	p<.001
Iteration/Loops	4.2041	0.7857	0.6993	1.1236	p<.001
Object-Orientation	2.3786	0.5357	0.8427	0.6357	p< .01
Data-Structures	2.2234	0.3214	0.5409	0.5942	p< .05
The Java Language	3.1225	0.4286	0.5136	0.8345	p<.001
Natural Language Processing	6.8165	1.3571	0.7449	1.8218	p<.001
Probability/Statistics	3.2289	0.6429	0.7449	0.8630	p<.001

Table 6: Pre/post change in students' knowledge and self-efficacy with programming concepts.

5.6 Observation Set 2: Code Reading and Writing

> *Parsons problems are easier and more reliable to mark than code writing, provide an opportunity to test student misconceptions more specifically than code writing, yet they appear to require the same set of skills... This makes them an excellent alternative to traditional code writing questions. [Denny 2008]*

5.6.1 Parsons Problems

The work reported here stems from a desire to better understand the degree to which PDAL students were actually learning to write and (equally importantly) read programming code. To this end, pre- and post-programming tests were administered to the students in

PDAL before and after exposure to the tools and course. The pre-test was given during the first class meeting, before any material had been presented and the post-test was given during the last meeting, just before final project presentations. Again, surveys were administered by a third party unaffiliated with the course and participants were instructed not to discuss the questions or to consult any references (e.g., books, laptops, etc.) during the allotted time.

```
[ return result;
[ return word;

[ String result = "";
[ String result;

[ if (word.charAt(i) == 'x')
[ if (word.charAt(i) != 'x')

[ for (int i = 0; i < word.length(); i++)
[ for (int i = 0; i < word.length; i++)

[ result = result + word.charAt(i);
[ result = word.charAt(i);

[ String removeAllXs(String word)
[ String removeAllXs(word)
```

Figure 18: A Parsons problem example (pre-test).

To measure students' understanding of programming we presented each with a question designed to measure both code writing and reading skills. The style of the problem, generally referred to as a 'Parsons problem'[99] presents students with a paired superset of the

[99] The term "Parsons' programming puzzle" was first introduced by Parsons and Haden [2006].

lines of code required to solve a programming problem. The student's task is to select the correct line of code from each pair, and then place the selected lines in the correct order to define a (Java) method that accomplishes the specified task. The Parsons problem from the pre-test for example, shown in Figure 18, asked students to define a method that would remove all 'x' characters from a String.

Our choice of Parsons problems was motivated primarily by two factors: available time and student diversity. Due to the limited time available for this component of our evaluation, separate code reading and writing problems, the latter being a potentially time-consuming activity, would have reduced the number of additional measures we could test. Rather than asking students to write code from scratch in one problem then to trace provided code in another, Parsons problems embed both skills into a single problem in which no code need be written from scratch. Further, the task presented is immediately comprehensible to students, even those who have no idea as to an answer. As regards the diversity of students taking the quiz, allowing students with little or no programming experience to simply guess at answers to the Parsons problems reduced, in our opinion, the chance that they would become intimidated by the material in the pre-test, possibly even to the extent that they might drop the class.

While both code reading and writing are embedded in a single Parsons problem, they can be coded and evaluated separately[100]. Further, as discussed in Denny [2008], this type of question simplifies distinguishing syntax errors from logic errors. For example, if a student

[100] We should note that, in contrast to code-writing tests, for which scores on Parsons problems show a high degree of correlation, scores on code-tracing problems seem to vary substantially from those on Parsons problems, suggesting that this somewhat novel approach needs further refinement in order to more adequately capture both skills [Denny 2008].

were to select the incorrect line from the second, fourth, or sixth pairs, this would prevent the code from compiling and hence be classified as a syntax error. Selecting the incorrect line from the first, third, or fifth pairs would generate incorrect programs, and would therefore classify as logic errors. While various strategies have been employed for evaluating such questions, in this study we chose to use a negative marking scheme, following Denny [2008], by defining the types of errors for which marks would be deducted.

Each Parsons problem consisted of six pairs of statements and was evaluated for both syntax and ordering on a scale of 0-9. For syntax, as in Denny [2008], we deducted one mark for each incorrectly chosen line from the pairs. For ordering, however, we used a minimum-edit, or Levenshtein distance[101] measurement to evaluate student variation from the correct result. This has several advantageous properties including: a) it can be reliably (and automatically) calculated by treating student answers as simple strings, so that inter-coder reliability is not an issue; and b) misplacing a single line of code does not destroy a student's score as could happen when using a simple binary (correct/incorrect) rubric for ordering. Consider the two example answers in the following table:

	Response	*Binary Score*	*MED Score*
Correct Order	A B C D E F	0	0
Student A	A B C D **F E**	-2	-2
Student B	**F** A B C D **E**	-6	-2

[101] Levenshtein distance is a measure of the similarity between two strings, generally referred to as the source string and target strings. The Levenshtein, (or 'min-edit') distance, is the minimal number of 'edits' needed to transform the source string into the target string, where an 'edit' consists of the deletion, insertion, or substitution of one character. [Levenshtein 1966; Marzal and Vidal 1993]

Table 7: Comparison of ordering metrics (incorrect lines are bolded).

In both of these cases, the ordering is correct with the exception of two lines that have been swapped. According to a naive binary metric, Student A would be marked as having four lines in the correct locations and two misplaced, while Student B would be marked as having all six lines incorrectly placed. According to the Min-Edit metric however, both students would be assessed an edit-distance of two points. Since both represent optimal non-correct answers as regards ordering (it is impossible in this format to have only one line out of place), these were assessed a one-point penalty by halving the Min-Edit-Distance. While Denny deducts a maximum of two out of nine total points for ordering, our metric, in the worst case, deducts three points (with an MED of six). Thus, by comparison, the ratio of syntax to ordering in our evaluation was 2-1, while in Denny [2008], still more weight was given to syntax (~78%) in relation to ordering (~22%).

5.6.2 Results

> *The multimedia terrain, with its strata of meanings, its combination of media, its compilation of data, and its branching, tangential connections would seem the ideal tool for this 'postmodern' age. But its chameleon character – a tool for writing, reading, talking and listening, a tool for drawing and looking, a tool for animating and viewing and a tool for gaming, interacting and consuming – makes it less easy to gauge in evaluative terms. [Sinker 2000]*

The Parsons question, measuring students' code reading and writing skills, showed a significant improvement between students' pre- and post-test scores: $t(7) = 7.5863$, $p<.00001$, $d= 2.1702$. The mean improvement was 3.3571 with a standard deviation of 1.5468, strongly

suggesting that students' programming skills had increased through exposure to RiTa and PDAL.

Statement pairs	*Proportion of students selecting the correct statement in Pre-Test*	*Proportion of students selecting the correct statement in Post-Test*
Return	0.43	1.00
Header	0.71	0.86
Initialization	0.71	1.00
Conditional	0.43	0.86
Accumulation	0.29	1.00
Loop	0.43	1.00

Table 8. Breakdown of the Parsons problem results (syntax).

Table 8 shows the breakdown of results obtained for the Parsons problem. The proportion of students that selected the correct statement from each pair is listed. In addition, we looked at the correctness of the ordering of the statements according to the MED algorithm described above. The results for the ordering of the statements in the Parsons problem are shown in Table 9.

Marks (0-3 deductions)	*Proportion of students in Pre-Test*	*Proportion of students in Post-Test*
Mostly Incorrect (-1.5 > x ≥ -3)	0.29	0
Partially Correct (0 > x ≥ -1.5)	0.57	0.57
Fully Correct (x = 0)	0.14	0.43

Table 9. Breakdown of the Parsons problem results (ordering).

Since Parsons problems consist of lines of code arranged in pairs, and students are asked to select the correct line of each pair to use in the solution, it was our hope to more easily measure the kinds of errors that students make. Since the correct option is always visible to students, when they choose the incorrect option we know that it is not a typo, but rather that a deliberate choice of the wrong option, potentially providing insight into the kinds of mistakes and misconceptions students have and allowing us to identify elements of the course that students are struggling with. The use of Parsons problems thus allows us to test knowledge needed for code writing and reading in a manner in which we can isolate specific misconceptions.

For example, Table 8 shows that only 29% of students chose the correct line for accumulating values on the pre-test. Thus, when presented with both alternatives, 81% chose the simple assignment statement rather than the correct statement. In a different context this might lead one to consider spending more time discussing the different roles that a variable can play in a program, and explicitly distinguishing between assignment operation and accumulation of a value in a variable. Although neither survey was scored until after the end

of the class, we see that in the post-survey 100% of students correctly selected the right accumulation statement.

Score (scale=0 - 9)	*Proportion of students in Pre-Test*	*Proportion of student in Post-Test*
Mostly Incorrect ($0 \leq x < 5$)	0.71	0
Partially Correct ($5 \leq x < 9$)	0.29	0.57
Fully Correct ($x = 9$)	0	0.43

Table 10. Breakdown of the Parsons problem results (total score).

Table 10 presents students' total scores on the programming quiz in both pre- and post-tests. Clearly, according to this metric, significant gains were made in code reading and writing skills over the course of the semester.

5.7 Observation Set 3: Creativity Support

> *One important implication of this inescapable trade-off between control and generalizability is that laboratory definitions of "creativity" are often so tightly constrained that they do not capture more than a piece of a person, product or process. [Hewett et al. 2005]*

As one of the project's hypotheses was that the RiTa tools provided a significant degree of creativity support for students and artists working in digital media, we attempted to further address this (beyond the survey data) by evaluating several aspects of students' final projects. Final project topics were proposed by students and developed over the last month of the course and while each proposal required the instructor's approval, the only requirements were that the project utilized computational methods and be of appropriate scope. Students

were not required to use any of the RiTa modules, the Eclipse Plugin, the Processing environment, or even Java itself, nor to focus specifically on language-based art. For those interested further, a number of these projects have been included in the RiTa gallery[102].

5.7.1. Evaluating Creativity

> *Basically, creativity can be considered to be the development of a novel product that has some value to the individual and to a social group. However, it seems that the research conducted by psychologists on creativity does not allow us to clarify or simplify this definition any further. Different authors may provide a slightly different emphasis in their definition but most (if not all) include such notions as novelty and value. [Hewett et al. 2005]*

To evaluate the degree of creativity support provided by tools like RiTa, it would be ideal to first decide unequivocally on a definition of creativity to employ. At the same time, such a definition has been highly contested in the literature [Turner, 2007] and is beyond the scope of this research. In the various definitions proposed in recent research however, there appear to be at least two components common to a majority [Sternberg 1999], specifically *novelty*, and *value*[103]. With this fact in mind we have chosen adopt the rather generic definition used in the 2005 Creativity Support Tools conference [Hewett et al, 2005], which is in turn based on Sternberg [1999] and focuses on the notion of *creative outputs*. Creative

---

[102] See http://www.rednoise.org/rita/rita_gallery.htm.

[103] For example, Gardner (1989) emphasizes that creativity is a human capacity but includes novelty and social value in his definition. Thus, our decision to adopt Sternberg's [1999] definition is not arbitrary as it represents somewhat of a consensus in the field. As Hewett et al. note [2005], the authors in Sternberg's collection provide a high level view of the state-of-the-art… "the work in this Handbook is highly consistent with the work of several other authors who have also surveyed major aspects of the research findings, e.g., Csikszentmihalyi [1997] and Gardner [1989]."

outputs can be conceptualized as artifacts generated in a specific context that score highly on both of these metrics, not only differing significantly from those artifacts already in existence (novelty), but doing so in a way that demonstrates value to the individual and/or social group (value) [Sternberg 1999]. To evaluate the creative outputs of our participant group we analyzed and coded students' final projects on a number of dimensions, believing this to be the most representative of the semester's outputs.

5.7.1.1 Evaluating "Value"

> *The National Advisory Committee on Creative and Cultural Education draws upon a range of conceptualisations of creativity and presents a definition which is a useful framework for educators - 'imaginative activity fashioned so as to produce outcomes that are both original and of value'. [Loveless 2002]*

As discussed above, the tools and techniques employed in the context of the PDAL course appear to have facilitated significant increases in students' programming efficacy and ability. Students' self-assessed ability to creatively express themselves through programming,, as noted above, showed a significant improvement over the course of the semester $t(12) = 2.560$, $p<.001$, $d = .739$. This, in combination with the fact that a broad range of students with highly variable prior experience[104] were able to complete works of significant depth and breadth in digital media, suggests that some non-trivial degree of creative utility (or value) was achieved. But what *kinds* of creative outputs were these?

As noted in our previous discussion of support software (see Chapter 3: Pedagogy), various tools will support diversity of output to varying degrees (often in inverse proportion

[104] See demographic data on majors and prior computing experience above.

to the specificity of the context for which they were designed,) and this is a key property to consider when assessing their efficacy. A common critique of 'user-level' tools like Photoshop or PowerPoint is that they tend to generate outputs that converge toward a distinct 'style' or 'signature'. General-purpose languages like C++ or Python, on the other hand, tend to support a wide variety of outputs, but exhibit steeper learning curves and often require significant scaffolding, especially for those users with diverse, or non-typical, backgrounds [Kelleher 2007].

In contrast to these approaches, we have positioned RiTa in a productive middle-ground position, attempting to satisfying all three of Resnick's primary design criteria for creativity support software: "low steps, wide walls, and high ceilings". In his discussion of these design principles, he uses l*ow steps* to refer to the incline of the learning curve, which should be as shallow as possible, while *wide walls* refers to the degree to which different learning paths can be followed and diverse outcomes achieved; and *high-ceilings* refers to the degree to which the tools grow with users through various levels of mastery – tools with low ceilings are easily mastered and thus do not continue to challenge and inspire learning [Resnick 2005].

The RiTa tools were designed with these principles in mind, targeting the joint goal of providing a) adequate scaffolding for new users, b) adequate flexibility and expressive power for advancing users, and c) support for a diverse range of creative outputs reflecting the diversity of users themselves. As all of the participants in the study were able to complete multiple computational literary projects, at least one of which was a mid to large-scale artwork, it would appear this goal was, at least partially, achieved. While it is beyond our scope to argue for the societal *utility* of such artwork in the abstract, we can however note the *perceived* utility to students in the class, who felt their work (and that of their peers,) to

represent an important means of expression. The following comments, from four different PDAL students, demonstrate the point:

> Prior to reading virtual muse, playing with different digital art projects of my own, and observing the different art galleries and products of digital artists, I had not thought much of using programming to express the artistic creative side of me. I had built up a mental barrier between these two worlds that this class has broken down. I think that this class is just a first step in my personal exploration of art and technology as a tool of expression.
>
> I just want to feel that what I'm working on, and the field I'm working in, are vibrant, changing, full of possibilities. That to sit down and write can mean and result in a lot of different things. That feeling is one I've had in this class (from the Markov models, from Hartman's pragmatism, from the grammars) and that experience is one I'm grateful for.
>
> If i [sic] have learned one thing, it is that code, like any other medium, can be coaxed, teased, and reworked into all kinds of forms, creating all kinds of subtle and expressive changes in a work.
>
> the fact that this class focused on language-based projects more than visual art was also a somewhat surprising but ultimately really fascinating turn. For one, I am personally profoundly enraptured by the kinds of texts that generative grammars in particular make; but the way n-grams and other statistic- and probability- based algorithms play with language I found unexpectedly interesting. I would love to do some work with these kinds of algorithms that would combine creative production and textual analysis; in even our mini-projects I was enchanted by the idea that recombining poetic texts might not only produces a new creative work but also allow for the gleaning of new meaning and making new connections...

Additionally, the reception of this work by the larger digital art community suggests a degree of utility in these outputs which appears to have been facilitated, at least in part, by the tools employed. As renowned digital media theorist and practitioner John Cayley [2009] writes about the student project gallery,

> Here's a gallery of proof that if you give expressive programming artists the right tools - Rita's programmable writing tools - then language will also drive their art, and make a literal art. As for literary artists, Rita gives them an articulate introduction to software art, and does so in a way that respects the language-making at the root of writerly practices. So many new things with novel aesthetics...

To further support the claim of value, we can note that projects generated in the class achieved significant success in external venues, including publication in prestigious digital literary journals, presentations at conferences, international awards, and grants for further work[105]. However, if we accept the general definition of creativity as outlined above, we must evaluate whether so-called "novel" outputs are generated by users of these tools as well. As a first step toward answering this question, we analyzed students' final projects from the semester, looking specifically at the diversity of their output as an experimental indicator of novelty.

5.7.1.2 Evaluating "Novelty"

> *"When evaluating the use of creativity support tools, we consider diversity of outcomes as an indicator of success. If the creations are all similar to one another, we feel that something has gone wrong." [Resnick et al. 2005]*

In attempting to analyze the novelty of final project results, we looked at the diversity of student project outcomes, first considering the categories of media that the projects engaged. Our intuition here, supported by a number of researchers [Resnick 2000; Resnick et al. 2005] was that a outcomes created via support tools that do not adequately support novelty would group together into one or a few categories of output. The following categories were proposed for the analysis: Audio, Video, Text, Image, Animation, Text, Text-To-Speech, Performance, Installation, Interactive, Locative or Psycho-Geographic, Physical

[105] Details of these are withheld in order to preserve student privacy in accordance with UCAIS (human subjects) guidelines.

Computation, Networked, and Web-Art[106], from which multiple codings were allowed. As shown in Table 11, projects spanned all of the aforementioned categories, with only three—Text (60%), Interactive (73%), and Web-Art (67%)—containing a majority of students. The popularity of these three categories was somewhat expected as nearly all examples and assignments during the semester fit into *all* of these categories. Audio (60%) was also a highly populated category, though an explanation of this result is less apparent. The topic of audio was not a central part of the course and only a few of the RiTa examples featured audio. Hypotheses for this finding include a) the fact that audio can generally be added to an existing project without much additional difficulty, b) several students had prior backgrounds in audio processing, and c) several examples works discussed in class contained audio components. Further study would be required to determine if this result is either significant or indicative of students' affinity for aural media.

Media Type	*% of Students (n=15)*
Audio	60%
Video	13%
Text	67%
Text-To-Speech	13%
Image	40%

[106] While a 'Web-Art' coding refers only to the project's being accessible (and executable) on the web, projects coded as 'Networked' employ custom client-server networking as an integral component of the work.

Performance	20%
Physical Computation	13%
Animation	40%
Networked	20%
Web-Art	73%
Interactive	67%
Installation	13%
Locative	13%

Table 11: Final project attributes.

A second element of our analysis of diversity projects looked at the RiTa modules employed in each final project. Our intuition here was that a outcomes created via a support tool that was too narrowly focused (i.e., lacking "wide walls") to adequately support novelty would leverage just one or a few modes of use. Here we coded each project with the specific RiTa modules that it employed (again multiple codings were allowed as many projects employed multiple parts of the toolkit.) Rather than proceeding by hand, codings here were produced automatically via source code examination (parsing both import statements and variable declarations) to minimize concerns of inter-coder reliability. Such analysis was enabled by the fact that students' source-code was required among the final project deliverables. Several students (20%) did not use RiTa at all, and were thus coded in the single category 'Did not use RiTa'.

In Table 11 we see that with the exception of Text-Display (53%), each of the modules were used by (at most) one-third of students. Several modules were either

(unexpectedly) unused or used by less than 10% of students. These include the Web-Mining (0%), RiTa-Server (0%) , WordNet (0%) and Audio (7%) components. Our hypothesis here is that each of these modules, with the possible exception of WordNet, are designed primarily to facilitate rapid development, a central design constraint for the toolkit (*Design Constraints* in Chapter 2). As such, they can often be substituted out in a final project by embedding their functionality elsewhere. The RiTa Server (also presented in Chapter 2) presents a case in point, as the module was designed primarily to enable micro-iteration in cases where loading times for large data models could be prohibitive, rather than to be used in 'finished' pieces. Similarly, the RiSearcher object provides real-time *n*-grams (for small *n)* via the Google Search engine and can often be replaced by a more-efficient embedded *n*-gram model (that does not require network access) in cases where the lexicon can be pre-determined.

RiTa Module	*% of Students (n=15)*
Grammars	33%
N-Grams	20%
Text Display	53%
Text Animation	13%
Text-To-Speech	20%
Feature Extraction (Tokenizers, Stemmers, Taggers, etc.)	20%
Audio Support	7%
Did not use RiTa	20%

Table 12. RiTa modules used in final projects.

The findings in Table 12 suggest that significant variation was in fact present in student projects. And while a number of factors may have contributed to this finding, it seems likely that the affordances of the RiTa tools played some not insignificant part. Further evidence in support of this finding was the fact that the two mini-projects previously assigned focused heavily on the grammar and n-gram modules. Thus one might reasonably expect these modules would be used more far more frequently in student projects, but this was not the case (average of all modules=23%, n-grams=20%, grammars=33%). This finding also

suggests that, as students gain facility with the tools and, concurrently, with digital arts and literary practice, they make progress toward the more general goal of procedural literacy. As literacy improves, their reliance on a single module, library, or even programming language, diminishes so that by the end of the semester students are capable of navigating APIs, documentation, and source-code, using tools that were not specifically discussed in class, and ignoring elements of tools designed to provide scaffolding for specific tasks which are no longer perceived as problematic. As discussed in Chapter 3 (Pedagogy), well-designed scaffolding provides support only for as long as the skill for which it is designed remains difficult. When this is no longer the case, the scaffolding should become not only unnecessary, but more importantly, transparent—incrementally illuminating the conceptual details it once allowed the learner to avoid in lieu of more pressing concerns.

5.8 Limitations

> *The further one moves away from the controlled laboratory situation the more difficult it becomes to establish a clear [and] unambiguous set of relationships that support valid conclusions. Research based in actual practice often supports many alternative explanations of what happens and how it happens. [Hewett et al. 2005]*

There are number of limitations to the pilot evaluation presented, specifically the lack of a control group, the nature of our participant pool, and the presence of a highly motivated experimenter. Further, as discussed in Denny [2008], Parsons' problems exhibit a number of limitations unrelated to this study.

5.8.1 Lack of Controls

As this pilot study is contextual, it exhibits both the strengths and drawbacks of this methodology. One such drawback in this case is the lack of any formal control group; that is, one or more sections of the course in which some other toolset and pedagogical approach was used. Unfortunately this type of experiment has thus far been infeasible in the context of PDAL. As such, two important questions remain unanswered as regards the efficacy of our tools and teaching strategies. First it is unclear whether one factor or multiple factors in combination are responsible for the results presented above. As noted in Hewett et al [2005], a weakness of conceptual methodologies is the fact that,

> the further one moves away from the controlled laboratory situation the more difficult it becomes to establish a clear unambiguous set of relationships that support valid conclusions.

Thus, in our study, students' gains in self-efficacy and performance may well be due to a number of factors: the selected tools, the pedagogical philosophy, the assignments and exercises, the public critiques—all of which are rather unique in computer science education—or some complex combination of one or more of these. Additionally, it is unclear to what degree students' measured progress was due to our approach or to other (external) factors; e.g., outside exposure to programming, or learning accomplished in other classes. As Turner [2007] states:

> It is difficult to precisely tell the effect of the presence of a technology and methodology, since there is no way to observe a creative process then 'rewind' it, insert the new technology and methodology, press 'play' and compare the results. The difference must be subjective, at least at small and medium scales of deployment. By using the term 'prefer', I have chosen one factor that could be said to represent an improvement in the creative process—that the creative worker prefers it. Other questions I could have chosen are, for example, whether the tools or methodologies made the process 'quicker', 'cheaper', 'more prize-winning', and so on, but these are difficult to evaluate in the context of interactive art at small or medium scales.

5.8.2 Participants

A further potential source of bias in the study was the fact that participants were from a narrow and non-representative demographic (specifically university and graduate students at two top-tier universities). Thus, it is unclear whether the effects noted would generalize to other demographic groups, still less so for under-represented groups [Plass et al. 2007]. Further, all participants in the study self-selected by enrolling in the course and consenting to participate in the study. As the course was not required, it is likely that participants had at least some prior motivation for learning the material in question, casting further doubt on the generalizability of our results.

5.8.2 Experimenter

As is often the case in contextual studies of this kind, the course instructor and software creator was also the author of the study and clearly influenced participants' experience of both the tools and teaching. As mentioned previously (See Chapter 3: Generative Pedagogy), this helped to facilitate the adaptive character of the course, as pace and direction were adjusted based on feedback about both the tools and the materials. It is unclear whether a course taught by a different instructor, specifically one less familiar with these tools, would yield similar results.

5.8.2 Software

Similarly, as is often the case in contextual studies of this kind, the software being evaluated was modified throughout the course of the study. Several bug fixes, and even a number of features requests were accommodated over the course of the semester, thus introducing another potentially confounding factor to the study.

5.8.2 Parsons problems

While Parsons problems present a number of advantages over traditional code reading and writing test, there are also weaknesses to this approach. First, it is quite possible to solve a Parsons problem (even correctly) and still not understand at a deep level how the code works. Similarly, for weak students, the Parsons problem format provides an opportunity to guess an answer that is not available in code-writing questions, though it is less likely, as compared to multiple-choice style questions, that many of these guesses will be correct. Further, Parsons problem do not adequately address process, as they present only a single method, while nearly all real programs skip from method to method. Finally, in contrast to code-writing tests which show a high degree of correlation to Parsons scores, scores on code-tracing problems seem largely independent [Denny 2008].

5.9 Summary

> *We have no delusions that evaluating tools is an easy task, but we also believe that the potential impact of improved tools would be enormous in amplifying and inspiring creativity. [Resnick et al. 2005]*

As noted above, there are significant issues with the largely descriptive evaluation presented here, perhaps most notably the lack of any true control, which is often cited as a weakness of contextual methods. Yet as an initial investigation into the real-world effects of these tools (and others like them) it is encouraging on a number of different dimensions. Students' attitudes, self-efficacy and ability all showed significant positive change measured between the pre- and post-semester surveys and programming quiz results. Additionally, though difficult to measure in a classroom environment, we see some evidence that the tools support student creativity. Students noted in increase in the frequency of their creative expression via computational media over the course of the semester and perceived their

outputs to be of importance beyond the class. Further, their work was well-received by the larger digital arts community, with projects later receiving publication in journals of digital literature, arts grants, and international awards. Further, students' projects exhibited surprising diversity both in terms of the types of media represented, and in the elements of the toolkit utilized. Of course, to verify any of the trends discussed here, additional research is needed, a topic discussed further in the concluding chapter.

CHAPTER 6: CONCLUSIONS

6.1 Contributions

This thesis has presented RiTa from a number of perspectives: as tools and affordances for practicing writers; as a pedagogical strategy, both for teaching procedural literacy to humanities students, and for engaging Computer Science students with creative practice; and as a real-world testing ground for creativity support principles, providing a unique context for assessing the efficacy of design and evaluation strategies. As such, the contributions of this research fall primarily into two related sub-fields: Creativity Support Tools (CST) research and Computer Science (CS) education, according to which they have been grouped below. In addition to contributions and future work in these areas we present several possible directions for additional evaluation of tools like those presented. Finally we conclude with a set of more speculative arguments based on our experience with artists and students as they engaged with these tools over the last several years.

6.1.1 Contributions: Creativity Support

In the context of creativity support tools research, RiTa represents the first production-quality toolkit designed specifically for practicing computational literary artists, complete with thorough documentation, examples, and an extensive catalog of sample projects, each with source code, screenshots, and descriptive text. In addition to the implementation details of the toolkit, we have presented a series of extensions to traditional natural language algorithms developed specifically for the needs of computational writers. Additionally we have created a number of auxiliary tools (e.g., the RiTaServer, the RiTa-Eclipse plugin, and the RiGrammarView application) to augment the library's functionalities in several widely-used environments. Further, we have detailed the set of *design criteria* (and

anti-criteria) that guided our implementation decisions and have enumerated the set of *design tensions* that arose in the course of our iterative development process. In detailing our resolutions to these conflicts, we have abstracted a set of principles which appear unique to creativity support in an arts context; specifically the need to support serendipity, inverted use, artistic misuse, and micro-iteration. Additionally, we have detailed the specific technical implementations with which we were able to realize the above principles, without sacrificing usability or performance goals. Most importantly we have distributed (and evaluated) these tools in real-world creative contexts, over the past three years, with a diverse population of users including students, practicing artists, and educators.

6.1.2 Contributions: Education

In the context of digital arts and computer science education, RiTa represents the first end-to-end toolkit designed specifically to support courses in computational literature. Additionally, the RiTa tools appear to accomplish several related but distinct pedagogical objectives:

- to effectively engage computer science students with creative practice and introduce new ways of thinking about the discipline;
- to broaden interest in computer science for a diverse range of students, furthering procedural literacy and computational thinking beyond the boundaries of the computer science department;
- to positively affect students attitudes and beliefs about programming, as well as their quantitatively-measured programming skills.

Finally, the RiTa tools, in combination with the pedagogical approach presented, and the Programming for Digital Arts and Literature (PDAL) course represent one of the first

successful integrations of tools and pedagogy in a digital media context, for both undergrads and graduate students, focusing on language and literature. As such it represents a new, and potentially important context for educators and researchers with which to explore the viability of procedural literacy, expressive programming, and computational thinking for a broader demographic.

6.1.3 Artistic Strategies

> *My most notable breakthrough, really, was when I realized my standard practice for CS classes doesn't work here. [PDAL student, 2009]*

It is evident from our experience, as described in the chapters above, that while there may be significant overlap between the strategies taught in a typical software engineering context, and thus designed into creativity support tools, some of these may be unproductive in an artistic and/or educational context. An obvious example is black-box style encapsulation, which, while generally a useful technique for API and/or library design, can often frustrate students' efforts to understand the larger picture of how their software operates, and contradicts pedagogical theories that advocate transparency and/or self-directed, exploratory learning.

Perhaps more interesting still, there appear to exist artistic programming strategies which extend beyond, and may even contradict, those generally taught in introductory computer science courses. Examples presented in chapter three include strategies that leverage micro-iteration, artistic misuse, inverted use, and serendipity. The reverse also appears to be the case as techniques generally considered to be standard practice in many software development contexts can be perceived as counter-productive in an arts context. One

such example is test-driven development which, at least during the early stages of a project, can significantly restrict the exploratory process so important to successful artistic outcomes.

These observations highlight two important conclusion for creativity support, both within and beyond the education environments we have been discussing. First, that context must be taken seriously at all levels of design; from workflow, to interface, to the design principles guiding even the most minute implementation details. This perspective corresponds to some recent research which demonstrates how social, political, and cultural assumptions and values can become embedded in technical artifacts at even the very lowest levels [Friedman et al. 1996, 2006; Flanagan et al. 2005b, 2007, 2008]. While the added effort required to analyze just what is at stake in such decisions may seem large at first, it pales in comparison to the time spent on software projects which fail, either partially or totally, due to their lack of attention to details such as these[107].

Second, these observations highlight the need for educators to pay careful attention to the mindset of students and its relation to the context at hand. It has not been the case, at least in our experience, that skills learned in say... an introductory programming class, can be easily applied in another pedagogical context, or at least this should not be assumed to be the case by educators. As one student faced with this difficulty put it, “I feel like I need to ... really fit myself into the mindset of an artist right now. Given that this is the only computer science course I'm taking at the moment, one would assume that would be easy, but that's not true at all. I'm having a lot of difficulty shifting my mindset into a more appropriate one.” Or, in the words of another student, “My most notable breakthrough, really, was when I realized my standard practice for CS classes doesn't work here. I approached this class as a CS

[107] For an overview of Value-Sensitive Design and its applications in a number of different research and industry projects, see http://depts.washington.edu/vsdesign/projects.shtml.

programming class, which meant staying up late at the last minute and doing my work then. This tactic doesn't lend itself to art in any shape or form, and though I am an artist, I haven't thought of computers in the same fashion." Here again we see how, beyond principles and implementation strategies, careful attention to the specificities of the context in which computation occurs appears to be an important factor in the relative success of tools like these; a factor that appears still more relevant when the context is an educational one.

6.2 Future Research

The research presented here on the RiTa toolkit and its use as a pedagogical tool suggests a number of different directions for further research in both the creativity support and educational communities. This section describes several additional projects in these disciplines that could be achieved with adequate institutional support and the right team of collaborators. They range in size from small, semester-long projects that could be accomplished by a single researcher (or even graduate student), to larger multi-year projects that might well require inter-departmental collaboration. The projects are organized according to the aspect of the research they would extend, whether the core toolkit itself, auxiliary support tools, one or more pedagogical applications, and evaluation measures, although several are applicable to more than one such area.

6.2.1 The RiTa Toolkit

There are a number of potentially interesting ways in which the core RiTa toolkit could be enhanced via further research and development. Some ideas along these lines include more sophisticated generation strategies, speech recognition capabilities, support for additional graphics contexts, and support for mobile platforms.

Over the past few month we have performed some initial experiments toward the goal of adding more sophisticated generation support, specifically for the lexicalized Tree Adjoining Grammar (TAG) formalism [Joshi et al. 1975]. While the mildly-context sensitive properties of TAG grammars would provide a degree of expressive power beyond that of the basic context-free grammars implemented in RiTa, this functionality would need to be balanced with an API that minimizes the complexity associated with such techniques, perhaps along the lines of Stone's [2002] *Taglet* system. This would provide a further level to explore with students in discussions of regular expressions, state-machines and the language hierarchy. On the other hand, the callback functionality currently provided in the RiGrammar object surpasses the functional capabilities of a basic TAG grammar (that is, a TAG object could be simulated by the current RiGrammar implementation, but not vice versa.) With this in mind, any gains in pedagogical clarity, most relevant perhaps to those students interested in natural language research, would have to be weighed against the additional complexity and potential confusion that might result from the inclusion of multiple grammar objects.

Additional implementations of RiTa's graphics capabilities are also a worthwhile direction to explore. At present, RiTa uses the rendering capabilities found in Processing[108], but it would be reasonably straightforward to implement multiple subclasses of the RiText object, each of which handles drawing in a different environment. An initially obvious choice for new graphics implementation would be a 'straight' Java version of RiText, which would bypass all Processing methods calls and instead utilize Java's drawing primitives. Another option, though more difficult to implement, would be to use the browsers built-in JavaScript

[108] A tutorial on the RiTa website details for students how these functions can be accessed outside of the Processing environment, e.g., in Eclipse.

rendering capabilities, similar to the way in which the Processing.js (http://processingjs.org/) library functions.

The inclusion of speech recognition capabilities in the toolkit is a feature requested by several students in the context of their PDAL projects. Toward this end, we have experimented with wrappers for the CMU Sphinx recognition framework which could enable users to define (and modify), in a plain text file similar to a RiTa grammar or addenda file, a context for speech recognition. Unfortunately, the data models required, at least in the Sphinx framework, are prohibitively large and would preclude execution in a web browser context. This is not to say that such functionality is not worth exploring further, only that it would likely need to be added as an optional or supplementary module, rather than as part of the core RiTa tools.

Lastly, an implementation of RiTa for one or more mobile platforms could be of significant utility as the importance of these devices for personal and/or artistic expression continues to grows. Further, the resource constraints of today's mobile platforms are analogous to those of web browsers, both placing strict limits on memory allocation, CPU utilization, and resource file sizes. Thus much of the work accomplished in optimizing RiTa for the web could be easily repurposed for mobile contexts. The open-source Android environment represents an initially promising candidate for such an effort, as it already contains a nearly complete J2SE stack and built-in speech-recognition capabilities. Such a port would again require a new graphics implementation as described above, though the fact that Android supports the OpenGL ES standard could significantly simplify this endeavor. While potentially a longer term project, mobile platforms represent, in our opinion, a particularly promising realm of exploration for the further development of artistic support tools.

6.2.2 Auxiliary Tools

> *I've often speculated, bitterly, as to why there is no word processor with the kind of filters and effects that are standard features in any of hundreds of graphic or audio manipulation programs; why page layout programs don't just 'know' how to typeset poetry with either traditional or post-Mallarmé sensitivities. [Cayley 2009]*

In addition to augmentations of the toolkit as discussed above, there are a number of ways in which the development of related tools might be pursued. One such direction might involve integration with one or more open-source word processing package to provide access (at the interface level) to a variety of literary filters and extensions, similar to the spell-check or thesaurus options found in current packages. Such an integration has been requested in the past by less technical users, outside of the educational context, who have expressed trepidation concerning RiTa's programmatic interface. While supporting these users is clearly important, the fact that a majority of existing creativity support research has targeted graphical user interfaces has caused us, at least thus far, to hesitate before embarking down this path.

Another augmentation, or alternative, to RiTa's text-based programmatic interface would be a visual programming interface similar to the Max/MSP environment discussed in chapter 3. An initial prototype of this functionality has been implemented and is currently being tested (see Figure 19 below).

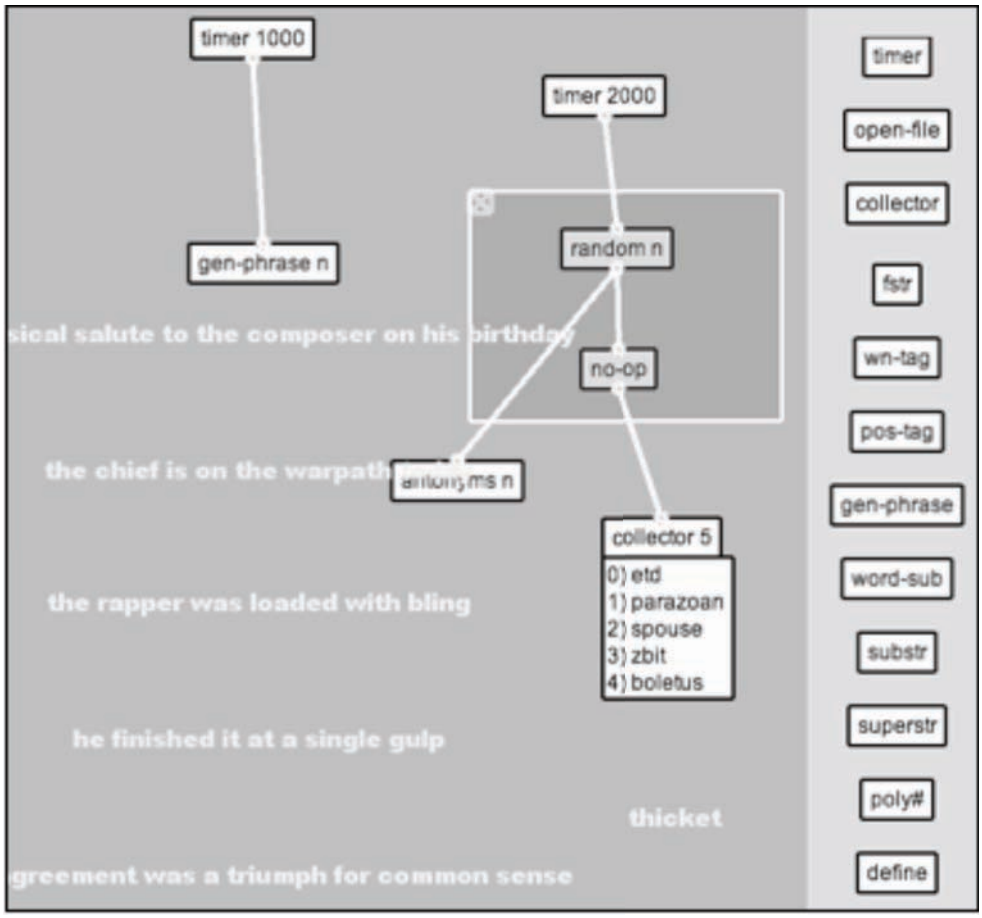

Figure 19: The RiTa live-coding environment (prototype).

6.2.3 Evaluation

> *It is no longer sufficient to evaluate whether people can use a given design to achieve a task easily and efficiently. We also need—sometimes primarily—to understand how the design resonates aesthetically, emotionally, socially and culturally, both with particular users and with a larger audience. And this implies that we need new sources of assessment on the one hand, and that assessments need to be multi-layered on the other. [Gaver 2007]*

6.2.3.1 Further Study

Further study is clearly needed to assess the effects of different tools on the degree of creativity exhibited in project outcomes, both for students and artists. Unfortunately such evaluation has generally proven to be both difficult and resource intensive. For an in-depth discussion of the issues involved and a variety of proposed approaches, we recommend 'Creativity Support Tool Evaluation Methods and Metrics' [Hewitt 2005], a report produced as part of the 2005 NSF-sponsored Creativity Support Tools conference. Similarly, further evaluation of the learning that occurs in courses like 'Programming for Digital Art and Literature' (or Mark Guzdial's 'Media Computation') would be extremely beneficial, especially were it to include a longitudinal component (following students beyond a single semester or year). One potentially useful experiment would be to teach similar material in parallel courses using a range of different toolsets (e.g., a general-purpose language like Python or Java, a RiTa-like approach, and a environment customized for education, e.g., Scratch or Alice) and compare students' relative confidence, self-efficacy and comprehension of core concepts after each. Of course the practical difficulties of designing and implementing such an experiment are significant; any proposal along these lines would likely require significant investment from one or more institutions. For those interested, Tew et al.'s [2005] "Impact of Alternative Introductory Courses on Programming Concept Understanding" describes some initial experiments along these lines at Georgia Tech.

6.2.3.2 Metrics for *programmatic* support

As mentioned in previous sections, evaluation metrics for creativity support tools to this point have primarily focused on usability as manifest through graphical user interfaces

[Nickerson and Landauer, 1997]. We can see this bias toward the visual and the surface even in the introductory paragraph of the NSF-sponsored Creativity Support Tools conference report which reads (in its entirety):

> Paradigm shifting breakthroughs make for great stories, but normal science is equally important in the evolutionary development of science, engineering, and medicine. Large and small breakthroughs are often made by scientists, engineers, designers, and other professionals who have access to advanced tools. *The telescopes and microscopes of previous generations are giving way to advanced user interfaces on computer tools that enable exploratory search, visualization, collaboration, and composition*. [Hewitt 2005, italics ours]

Here we can appreciate what has been referred to "the tyranny of the visual" [Arlen 1979]. Instead of focusing on the power of computational methods to help generate new creative techniques and outcomes, the focus is on the power of the user interface, and on tools for augmenting vision and visualization. While such approaches are clearly worth pursuing, they are not the only avenue for new research. In fact, this overwhelming focus on the visual interface suggests that new researchers in the field might perhaps do well to focus their attention elsewhere; on the aural, the haptic, the linguistic, or the conceptual.

In addition to this overwhelming focus on the visual, usability has generally been evaluated in the context of *productivity* in traditional tasks, even when performed by creative professionals. As Hewett [2005] comments, "researchers often focus on serving professionals such as business decision makers, biologists exploring genomic databases, designers developing novel consumer products..." In contrast, we argue, further attention to *exploratory* creativity is also warranted, focusing on contexts (e.g., the arts) in which creative goals are less-well-defined, and users work at a range of proximities to computational mechanisms. As Jennings notes, "new media arts practitioners and researchers should be regarded as valuable contributors not only as users needing better creativity support tools (CST) to enhance their

own creative process, but also as the designers of experimental and innovative creativity support tools" [Jennings et al 2005].

Here is where the notion of a *programmatic* usability might prove useful, shifting our focus away from the interface and toward the authoring of creative *processes.* Such a focus might not only increase the utility and novelty of creativity outputs, but would facilitate a transition away from surface effects and instead toward a deeper understandings and engagement with process, such as has been argued for by proponents of procedural literacy. As a trivial example, instead of realizing a new way to use a provided Photoshop filter, a procedurally literate user might design and implement their own filter, which not only better expresses their creative vision, but could then be shared with other filter users and designers. As Noah Wardfrip-Fruin says in his introduction to Expressive Processing [2009], "It is common to think of the work of authoring, the work of creating media, as the work of writing text, composing images, arranging sound, and so on. But now one must think of authoring new processes as an important element of media creation."

6.2.3.3 Longitudinal Studies

> *It is still an open question how to measure the extent to which a tool fosters creative thinking. While the rigor of controlled studies makes them the traditional method of scientific research, longitudinal studies with active users for weeks or months seem a valid method to gain deep insights about what is helpful (and why) to creative individuals.* *[Seo 2006]*

Lastly, creativity support tools research such as presented above could benefit greatly from the application of longitudinal studies, with students and, especially, practicing artists. While short term evaluations have clear advantages, their efficacy is often limited as the success of creativity support software is determined over years, not weeks or months. Studies

that examine the all the various phases in a tools adoption, from initial uses, to increasing familiarity, to eventual mastery, to use in teaching and/or mentoring, will be particularly useful in eliciting a more complete set of properties that either facilitate or discourage creativity in these different scenarios.

6.3 Final Thoughts

In conclusion there are several points from above that we wish to reiterate. First, that perhaps for the first time, in part due to increases in the computing resources available to the average user, tool support for computational literature appears to be feasible, even when considering constrained-execution domains such as the web browser. Second, given such support, computational literature appears to be a productive new context for a wide range of users, not only for practicing computational artists and others with extensive programming knowledge, but also for students with variable backgrounds and skill levels. Third, this fact is particular encouraging as the context of computational literature also appears to quickly and naturally raises core ideas in both art and computer science, facilitating students' creative expression while simultaneously advancing procedural literary, and exposing students to the core concepts that constitute 'computational thinking'. Although not practical in all cases, it is also worth noting that the benefits of this approach appear to multiply when tools and accompanying pedagogy can be mutually informing, as was the case with the RiTa tools in context of the 'Programming for Digital Arts & Literature' course.

Additionally we wish to raise a few more speculative ideas for future researchers which, although not adequately tested, appear evident from our experience. First, that creativity support is not something easily simulated in a laboratory environment, and thus the field desperately needs more real-world applications, with careful and well-planned

evaluation. Second, whenever possible, creativity support researchers should leverage synergies with other research programs stressing creativity, specifically procedural literacy, expressive programming, and the computational arts. Much of the creativity support research performed thus far has been limited by the lack of such cooperation, and by an over-emphasis on the visual elements of creative artifacts, whether the visual properties of creative outputs, or the interface elements implemented in new creativity support tools. It is important for researchers to appreciate the fact that artists have a long history of building and refining their own tools to better match their needs, and low-level control over the medium has generally been an important property of such tools. Further, such tools, even when designed and developed by a single artist for a specific project, have often proven to be of significant general utility for others working in related areas.

Lastly, we conclude by making explicit what the reader has likely sensed beneath the surface of this work; specifically that our interest in these topics is not motivated only by an impassive scientific curiosity, but instead by a deep and abiding interest in the topics at hand. It is not by chance that this research focuses on the intersection of creative writing and exploratory computational practice, but rather because these have been central concerns in our lives, in our own artistic practice and in the work of the authors and artists that have moved us. As such, we can only hope that this writing inspires in the reader some small fraction of the aesthetic and intellectual excitement we have experienced in its creation.

APPENDIX A: RESOURCES

Links to further resources for RiTa / PDAL	
The RiTa library	*http://www.rednoise.org/rita/*
RiTa for WordNet	*http://www.rednoise.org/rita/wordnet*
The RiTaServer	*http://www.rednoise.org/rita/documentation/ritaserver_class_ritaserver.htm*
The RiTa-Eclipse plugin	*http://www.rednoise.org/ep5/*
RiTa example programs	*http://www.rednoise.org/rita/examples/*
The RiTa Project Gallery	*http://www.rednoise.org/rita/rita_gallery.htm*
Programming for Digital Art & Literature course web	*http://www.rednoise.org/pdal/*
RiTa documentation (reference)	*http://www.rednoise.org/rita/documentation/docs.htm*
RiTa documentation (javadocs)	*http://rednoise.org/rita/javadocs/index.html?rita/package-summary.html*

ENDNOTES

[i] The concept of an affordance was coined by the perceptual psychologist James J. Gibson in his seminal book The Ecological Approach to Visual Perception. The concept was introduced to the HCI community by Donald Norman in his book The Psychology of Everyday Things from 1988. According to Norman (1988) an affordance is the design aspect of an object which suggest how the object should be used; a visual clue to its function and use. Norman writes:

"...the term affordance refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used. [...] Affordances provide strong clues to the operations of things. Plates are for pushing. Knobs are for turning. Slots are for inserting things into. Balls are for throwing or bouncing. When affordances are taken advantage of, the user knows what to do just by looking: no picture, label, or instruction needed." (Norman 1988, p.9)

Norman thus defines an affordance as something of both actual and perceived properties. The affordance of a ball is both its round shape, physical material, bouncability, etc. (its actual properties) as well as the perceived suggestion as to how the ball should be used (its perceived properties). When actual and perceived properties are combined, an affordance emerges as a relationship that holds between the object and the individual that is acting on the object (Norman 1999). As Norman makes clear in an endnote in Norman (1988), this view is in conflict with Gibson's idea of an affordance (explained next).

As opposed to Norman's use of his term, Gibson intended an affordance to mean "an action possibility available in the environment to an individual, independent of the individual's ability to perceive this possibility" (McGrenere and Ho, 2000). Unlike Norman's

inclusion of an object's perceived properties, or rather, the information that specifies how the object can be used, a Gibsonian affordance is independent of the actor's ability to perceive it.

Gibson's Affordances	*Norman's Affordances*
* Action possibilities in the environment in relation to the action capabilities of an actor * Independent of the actor's experience, knowledge, cultre, or ability to perceieve * Existence is binary - an affordance exists or it does not exist.	* Perceived properties that may not actually exist * Suggestions or clues as to how to use the properties * Can be dependent on the experience, knowledge, or culture of the actor * Can make an action difficult or easy

Table E1: Affordances as defined by Gibson and Norman [McGrenere and Ho, 2000].

BIBLIOGRAPHY

Ades, D. 1974. *Dada and Surrealism*. Thames and Hudson, London, UK.

Anderson, J. R., Conrad, F. G. and Corbett, A. T. 1989. Skill Acquisition and the LISP Tutor. *Cognitive Science* 13, 467-505.

Anewalt, K. 2002. Experiences teaching writing in a computer science course for the first time, Journal of Computing Sciences in Colleges, v.18 n.2, p.346-355, December 2002

Arlen, M. J. 2000. The Tyranny of the Visual. In *The Norton Reader: AANP*. Ed. Eastman, A. M. et al. New York: Norton 2000. 1067-1074

Attridge, D. 1995. Poetic Rhythm: an Introduction. Cambridge University Press. Back, T., Fogel, D., and Michalewicz, Z., editors (1997). Handbook of Evolutionary Computation. Oxford University Press and Institute of Physics Publishing.

Bachleitner. N. 2005. The Virtual Muse. Forms and Theory of Digital Poetry. In *Theory into Poetry: New Approaches to the Lyric*. Rodopi. Amsterdam/New York, 2005.

Bailey, R. W. 1974. Computer-assisted poetry: the writing machine is for everybody. In Mitchell, J. L., editor, *Computers in the Humanities*, pages 283–295. Edinburgh University Press, Edinburgh, UK.

Baldridge, J. Morton, T. and Bierner, G. 2002 The MaxEnt project. Available at http://maxent.sourceforge.net/. Accessed 04/01/2006.

Balestrini, N. 1996. Tape Mark I. In *Cybernetic Serendipity: The Computer and the Arts*. E. Morgan, Tr., 55-56.

Balpe, J.-P, 2005. Principles and Processes of Generative Literature, In *Dichtung Digital*. Available at http://www.dichtungdigital.com/2005/1/Balpe/. Accessed 08/09/09.

Baldwin, S. 2008. Codework: Starting Points. Wiki for *Codework: Exploring relations between creative writing practices and software engineering.* A workshop sponsored by the National Science Foundation. West Virginia University, April 3-6, 2008. Available at http://clc.as.wvu.edu:8080/clc/CodeworkWorkshopWiki. Accessed 08/08/09.

Basharin, G., Langville, A., and Naumov, V. 2004. The Life and Work of A. A. Markov. *Linear Algebra and its Applications* 386, 3–26.

Begel, A. 1996. *LogoBlocks: A Graphical Programming Language for Interacting with the World*. EECS, MIT Press, Boston, MA.

Bénabou, M. 1996. *Aphorismes*. C. Funkhouser and N. Peyrafitte, Tr. Syntext. Diskette. Porto: Edições Afrontamento.

Berger, A. L., Della Pietra, S., and Della Pietra, V. 1996. A Maximum Entropy Approach to Natural Language Processing. *Computational Linguistics*, 22(1), 39–71.

Bird, S and Loper, E. 2002. NLTK: The Natural Language Toolkit. In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Association for Computational Linguistics, Somerset, NJ, 62-69.

Black, A. W., and Taylor, P. A. 1997. The Festival Speech Synthesis System: System documentation. *Technical Report HCRC/TR-83*, Human Communication Research Centre, University of Edinburgh, Scotland, UK, 1997.

Block, N. 1981. Psychologism and Behaviorism. *The Philosophical Review*, 90(1), 5-43.

Boden, M. 1990. *The Creative Mind: Myths and Mechanisms*. Basic Books, New York, NY. (Precis, with peer reviews, in *Behavioral and Brain Sciences*, 17(3), 1994.)

Bogost, I. 2005. Procedural Literacy Problem Solving with Programming, Systems, and Play. *The Journal of Media Literacy*, 52(1-2).

Bontcheva, K., Cunningham, H., Tablan, V., Maynard, D., and Hamza, O. 2002. Using GATE as an environment for teaching NLP. In Proceedings of the Acl-02 Workshop on Effective Tools and Methodologies For Teaching Natural Language Processing and Computational Linguistics - Volume 1 (Philadelphia, Pennsylvania, July 07 - 07, 2002). Annual Meeting of the ACL. Association for Computational Linguistics, Morristown, NJ, 54-62.

Booth, A. D. 1965. *Digital Computers in Action*. Pergamon Press, Oxford, UK.

Bootz, P. 2008. What is the relationship between CREATIVE programming and creative writing? Paper presented at *Codework: Exploring relations between creative writing practices and software engineering.* A workshop sponsored by the National Science Foundation. West Virginia University, April 3-6, 2008.

Brill, E. 1992. A Simple Rule-based Part of Speech Tagger. In Proceedings of the Third Conference on Applied Natural Language Processing, 1992, 152-155.

Bringsjord, S. 1998. Chess is too Easy. *Technology Review*, 101(2), 23-28.

Bringsjord, S. and Ferrucci, D. 1999. *Artificial Intelligence and Literary Creativity: Inside the Mind of Brutus, A Storytelling Machine*. Lawrence Erlbaum Associates, Mahwah, NJ.

Bringsjord S. and Taylor, J. 2005. An Argument for P = NP. http://kryten.mm.rpi.edu/scb_pnp_solved22.pdf. May 31, 2005. Accessed November 11, 2006.

Bringsjord, S. 2008. The Logicist Manifesto: At long last let logic-based artificial intelligence become a field unto itself. *Journal of Applied Logic*, 6(4), 502-525.

Brooks, F. P. Jr. 1996. The computer scientist as toolsmith II, Communications of the ACM, v.39 n.3, p.61-68, March 1996 [doi>10.1145/227234.227243]

Brown, J. S., Collins, A., and Duguid, P. 1989. Situated Cognition and the Culture of Learning. *Educational Researcher*, 18(1), 32–42.

Bruckman, A. 1997. *MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids*. MIT Media Lab, Boston, MA.

Bruckman, A., and Resnick, M. 1995. The MediaMOO Project: Constructionism and Professional Community. Convergence, vol. 1, no. 1, pp. 94-109.

Burg, J. 2003. Creative Explorations in Digital Media. In Lassner D. & C. McNaught (Eds.), *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2003* (pp. 2285-2288). Chesapeake, VA: AACE

Bulhak, A. 2009. The Dada Engine. Available at http://dev.null.org/dadaengine/. Accessed 7/01/2009.

Bulhak, A. 2009. Postmodernism Generator. Available at http://www.elsewhere.org/pomo/. Accessed 7/01/2009.

Burroughs, W. S. 1978. *The Cut-Up Method of Brion Gysin*. Viking, New York, NY, 29–33.

Cage, J. 1969. *A Year From Monday*. Wesleyan University Press, Middletown, CT.

Cage, J. 1973. *Silence: Lectures and Writings by John Cage*. Wesleyan University Press, Middletown, CT.

Cage, J. 1980. Music and Particularly Silence in the Work of Jackson Mac Low. *Paper Air*, 2(1), 36-39.

Cage, J. 1990. *I-VI.* Wesleyan University Press, Hanover, NH.

Calvino, I. 1974. *Invisible cities*, W. Weaver, Tr. Harcourt, New York, NY.

Calvino, I. 1995. *Numbers in the Dark and Other Stories*, T. Parks, Tr., Preface by E. Calvino. Pantheon Books, New York, NY.

Calvino, I., 1986. Prose and Anticombinatorics. In *Oulipo: A Primer of Potential Literature*, W. F. Motte, Ed. University of Nebraska Press, Lincoln, NE.

Candy, L. and Edmonds, E. A. 1997. Supporting the Creative User: A Criteria-Based Approach to Interaction Design. *Design Studies* 18, 185-194.

Candy, L. and Edmonds, E. 2002. Explorations in art and technology, Springer-Verlag, London.

Cayley, J. 1996. Beyond Codexspace: Potentialities of Literary Cybertext. Visible Language, 30(2), 164–83.

Cayley, J. 2003. Hypertext/Cybertext/Poetext.' Assembling Alternatives: Reading Postmodern Poetries Transnationally. Ed. Romana Huk. Middletown, CT: Wesleyan University Press, 2003. 310-26.

Cayley, J. 2003(b). 'Inner Workings: Code and Representations of Interiority in New Media Poetics.' dichtung-digital 29 (2003). From a presentation at the Language and Encoding Conference, Buffalo, Nov. 02, Proceedings edited by Loss Pequeño Glazier. Available at http://www.dichtungdigital.com/03/3-cayley.htm. Accessed 04/01/08.

Cayley, J. 2004. 'Literal Art: Neither Lines nor Pixels but Letters.' First Person: New Media as Story, Performance, and Game. Eds. Noah Wardrip-Fruin and Pat Harrigan. Cambridge: MIT Press, 2004. 208-17.

Cayley, J. 2004(b). 'The Code Is Not the Text (Unless It Is the Text) = Der Code Is Nicht Der Text (Es Sei Denn, Er Ist Der Text).' p0es1s: Ästhetik Digitaler Poesie = the Aestheticsof Digital Poetry. Eds. Friedrich W. Block, Christiane Heibach and Karin Wenz. Ostfildern-Ruit: Hatje Cantz Verlag, 2004. 287-306.

Cayley, J. 2006. 'Time Code Language: New Media Poetics and Programmed Signification.' New Media Poetics: Contexts, Technotexts, and Theories. Eds. Adalaide Morris and Thomas Swiss. Cambridge: MIT Press, 2006. 307-33.

Cayley, J. 2009. Electronic mail correspondence. 08/15/09.

Castigilia C., Utterback C., and Wardrip-Fruin, N. 2002. Talking Cure. Available at http://www.hyperfiction.org/talkingcure/index.html. Accessed 7/05/09.

Chomsky, N. 1965. *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA.

Clarkson, P. and Rosenfeld, R. 1997. Statistical language modeling using the CMU-cambridge toolkit. In *EUROSPEECH-1997*, 2707-2710.

Conway, M. 2000. Alice: Lessons Learned from Building a 3D System For Novices. In *Proceedings CHI'2000: Human Factors in Computing Systems*. The Hague, The Netherlands, Apr 1-6, 2000. pp. 486-493.

Cramer, F. 2005. *Words Made Flesh: Code, Culture, Imagination*. Piet Zwart Institute, Rotterdam. http://pzwart.wdka.hro.nl/mdr/research/fcramer/wordsmadeflesh/.

Cramer, F. 2005. Combinatory Poetry and Literature in the Internet. Available at http://userpage.fuberlin.de/~cantsin/homepage/writings/net_literature/permutations/kassel_2000/combinatory_poetry.html. Accessed 08/01/09

Csikszentmihalyi, M. 1997. *Creativity: Flow and the Psychology of Discovery and Invention.* Harper Perennial, New York, NY.

Davis, R.B., Maher, C.A., and Noddings, N., Eds. 1990. *Constructivist Views of the Teaching and Learning of Mathematics*. National Council for the Teaching of Mathematics, Reston, VA.

De Smedt, K., Horacek, H., and Zock, M. 1996. Architectures for natural language generation: Problems and perspectives. In Adorni, G. and Zock, M., editors, *Trends in Natural Language Generation: An Artificial Intelligence Perspective*, number 1036 in Springer Lecture Notes in Artificial Intelligence, pages 17–46. Springer-Verlag, Berlin, Germany.

Denny, P., Luxton-Reilly, A., and Simon, B. 2008. Evaluating a New Exam Question: Parsons problems. In Proceeding of the Fourth international Workshop on Computing Education Research, Sydney, Australia, September 2008. ICER '08. ACM, New York, NY, 113-124.

Denning, P. J. 2003. Great principles of computing, Communications of the ACM, v.46 n.11, November 2003.

Diaz-Agudo, B., Gervás, P., and Gonz´alez-Calero, P. 2002. Poetry generation in COLIBRI. In Proceedings of the 6th European Conference on Case Based Reasoning (ECCBR 2002), Aberdeen, UK.

diSessa A. 1985. A Principled Design for an Integrated Computational Environment. In *Human-Computer Interaction*, 1(1), 1-47.

diSessa A. 1991. Local Sciences: Viewing the Design of Human-computer Systems as Cognitive Science. In: *Designing Interaction: Psychology at the Human-Computer Interface*, J. Carroll, Ed. Cambridge University Press, New York, NY.

diSessa, A. 2000. *Changing Minds: Computers, Learning, and Literacy*. MIT Press, Cambridge.

Dougherty, M. 2004. What Has Literature to Offer computer science? In *HUMAN IT* 7.1(2004): 74-91. Available at http://www.hb.se/bhs/ith/1-7/md.pdf

Douglass, J. 2000. Machine Writing and the Turing Test (presentation in Alan Liu's Hyperliterature seminar, University of California, Santa Barbara). http://www.english.ucsb.edu/grad/studentpages/jdouglass/coursework/hyperliterature/turing/.

DuBoulay, B., O'Shea, T. and Monk, J. 1989. The black box inside the glass box: Presenting computing concepts to novices. In *Studying the Novice Programmer*, E. Soloway, J.C. Spohrer, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ.

Durndell, A., & Haag, Z. 2002. Computer self-efficacy, computer anxiety, attitudes towards the Internet and reported experience with the Internet, by gender, in an East European sample. *Computers in Human Behavior*, 18(5), 521-535.

Elshtain, E. 2006. Gnoetry: an interview. Published by C. Dena, April 2nd, 2006 in *HCTI, generators, Poetics, bots, CCS, Text Art, Criticism, Software, HAI*. Available at http://writerresponsetheory.org/wordpress/2006/04/02/gnoetry-interview-with-eric-elshtain/. Accessed 7/04/09.

Ernest, P., Ed. 1994. *Constructing mathematical knowledge: Epistemology and mathematics education*. The Falmer Press, London, UK.

Everett, D. 2005. Cultural Constraints on Grammar and Cognition in Pirahã: Another Look at the Design Features of Human Language. *Current Anthropology*, 46(4).

Fellbaum, C., ed. 1998. WordNet: An Electronic Lexical Database. MIT Press.

Fishwick, P. 2006. An introduction to aesthetic computing. Aesthetic Computing. The MIT Press, Cambridge, MA, 2006, p3-27.

Flanagan, M., Howe, D. C. and Nissenbaum, H. 2005. Values at Play: Design Tradeoffs in Socially-Oriented Game Design. In Proceedings of CHI 2005: Portland Oregon. ACM Press, New York, NY.

Flanagan, M., Howe, D. C. and Nissenbaum, H. 2005b. New Design Methods for Activist Gaming. In Proceedings of DiGRA 2005, Vancouver, BC, Canada, June 2005.

Flanagan, M., Howe, D. C. and Nissenbaum, H. 2007. Design Method Outline for Activist Gaming. In *Worlds in Play*, S. de Castell and J. Jensen, Eds. Peter Lang Publishing, New York, NY.

Flanagan, M., Howe, D. C., and Nissenbaum, H. 2008. Embodying Values in Design: Theory and Practice. In *Information Technology and Moral Philosophy*, J. van den Hoven and J. Weckert, Eds. Cambridge University Press, Cambridge, UK.

Florida, R. 2002. The Rise of the Creative Class and How It's Transforming Work, Leisure, Community and Everyday Life. Basic Books, New York, NY.

Forte, A. and Guzdial, M. 2004. Computers for Communication, Not Calculation: Media as a Motivation and Context for Learning. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*, Vol. 4, 40096a.

Friedman, B. 1996. Value-sensitive design. *interactions* 3, 6 (Dec. 1996), 16-23.

Friedman, B., Khan, P. H., and Howe, D. C. 2000. Trust online. *Communications of the ACM* 43, 12 (Dec. 2000), 34-40.

Friedman, B., Howe, D. C. and Felten, E. W. 2006. Informed Consent in the Mozilla Browser: Implementing Value-Sensitive Design. In *Internet Security: Hacking,*

CounterHacking, and Society, K.E. Himma, Ed. Jones and Bartlett Publishers, Boston, MA, 2006.

Funkhouser, C.T. 2007. *Prehistoric Digital Poetry: An Archaeology of Forms (Modern & Contemporary Poetics)*. University of Alabama Press, Tuscaloosa, AL

Funkhouser, C. 2008. expansion within Restriction: conviction and poesis in computer poems. Paper presented at *Codework: Exploring relations between creative writing practices and software engineering.* A workshop sponsored by the National Science Foundation. West Virginia University, April 3-6, 2008.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Gardiner, H. 1993. *Creating Minds*. Basic Books, New York, NY.

Garlan, D. and Miller, P. 1984. Gnome: An introductory programming environment based on a family of structure editors. In Proceedings of the Software Engineering Symposium on Practical Software, Pittsburgh, PA, April 1984.

Gatt, A., and Reiter, E. 2009. SimpleNLG: A realisation engine for practical applications. In *Proceedings of ENLG-2009.* Forthcoming 2009.

Gaver, W. W. 2007. Cultural commentators: Non-native interpretations as resources for polyphonic assessment. In *International Journal of Human-Computer Studies*, 65 (4) pp. 292-305

Gervás, P. 2002. Exploring quantitative evaluations of the creativity of automatic poets. In Proceedings of the 2nd. Workshop on Creative Systems, Approaches to Creativity in Artificial Intelligence and Cognitive Science, 15th European Conference on Artificial Intelligence(ECAI 2002),

Gibson, J. J. 1977. The theory of affordances. In *Perceiving, Acting, and Knowing*, R. E. Shaw and J. Bransford, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ.

Gibson, J. J. 1979. *The Ecological Approach to Visual Perception*. Houghton Mifflin, Boston, MA.

Glynn, S.M., Yeany, R.H., and Britton, B.K., Eds. 1991. *The psychology of learning science*. Lawrence Erlbaum, Hillsdale, NJ.

Golomb, S., Berlekamp, E. R., Cover, T. M., Gallager, R. G., Massey, J. L., Viterbi, A. J. 2002. Claude Elwood Shannon (1916-2001). In *Notices of the American Mathematical Society*, January 2002, 8-16.

Greenberger, M., Ed. 1962. *Management and the Computer of the Future*. The MIT Press, Cambridge, MA.

Gruber, H. and Davis, S. 1988. Inching our way up mount olympus: The evolving systems approach to creative thinking. In Sternberg, R. J., editor, The Nature of Creativity, pages 243–269. Cambridge University Press, New York, USA.

Gutierrez, J. B. and Marino, M. C. 2008. Literatronica: adaptive digital narrative. In *Proceedings of the Hypertext 2008 Workshop on Creating Out of the Machine: Hypertext, Hypermedia, and Web Artists Explore the Craft* (Pittsburgh, PA, USA, June 19 - 21, 2008). Creating '08. ACM, New York, NY, 5-8.

Guzdial M. 1991. The need for education and technology: Examples from the GPCeditor. In *Proceedings of the National Educational Computing Conference*. Phoenix, AZ, 16-23.

Guzdial M., Soloway E., Blumenfeld P., et al. 1992. The future of CAD: Technological support for kids building artifacts. In *Learning to Design, Designing to Learn: Using*

Technology to Transform the Curriculum, D. Balestri, S. Ehrmann and D. L. Ferguson, Eds. Ablex Publishing Company, Norwood, NJ.

Guzdial M. 1993. Emile: Software-realized scaffolding for science learners programming in mixed media. Unpublished Ph.D. dissertation, University of Michigan, Ann Arbor, MI.

Guzdial, M. 1994. Software-Realized Scaffolding to Facilitate Programming for Science Learning. *Interactive Learning Environments*. 4(1), 1-44.

Guzdial, M. 1997. Constructivism vs. Constructionism. Available at http://guzdial.cc.gatech.edu/Commentary/construct.html. Accessed 04/23/09.

Guzdial, M. 2003. A media computation course for non-majors, In *ITiCSE Proceedings*. ACM, New York, NY, 104-108.

Guzdial, M. and Soloway, E. 2003. Computer science is more important than calculus: The challenge of living up to our potential, In *Inroads-The ACM SIGCSE Bulletin*, 35(2), 5-8.

Guzdial, Mark. (2008). Education: Paving the Way for Computational Thinking. Communications of the ACM, 51(8), 25-27

Hammersley, M. 1995. The Politics of Social Research. Sage Publications, Thousand Oaks, CA.

Harel, I. 1991. *Children Designers: Interdisciplinary Constructions for Learning and Knowing Mathematics in a Computer-rich School*. Ablex Publishing, Norwood, NJ.

Harrington, J. & Cassidy, S. 1999. *Techniques in Speech Acoustics*. Kluwer Academic Publishers: Foris, Dordrecht.

Harrington, D. 1990. The Ecology of Human Creativity: A Psychological Perspective. In M. Runco, & R. Albert (Eds.), *Theories of Creativity*, Sage, Newbury Park, CA, pp. 143-169.

Hartman, C. O. 1988. Essay Ending with Six Reasons to Read Jackson Mac Low: *Representative Works: 1938-1985* and *The Virginia Woolf Poems*, [by] Jackson Mac Low. *Prairie Schooner* 62.

Hartman, C. O. 1996. *Virtual Muse: Experiments in Computer Poetry*. Wesleyan University Press, Hanover, NH.

Hayes, B. 1983. Computer recreations: A progress report on the fine art of turning literature into drivel. *Scientific American*, 249(5), 18–28.

Hazan, H. 2001. The other voice: On the qualitative research sound. In *Qualitative Research: Genres and Traditions in Qualitative Research*, N. Sabar, Ed. Zmora Bitan, Tel-Aviv, Israel, 9 – 12.

Herman, J., Ed. 1973. *Brion Gysin Let The Mice In* (with William S. Burroughs & Ian Sommerville). Something Else Press, VT.

Hewett, T., Czerwinski, M., Terry, M. et al., 2005. Creativity Support Tool Evaluation Methods and Metrics. in *Creativity Support Tools*, Washington, D.C.

Hiemstra, D. 2009. Language Models. In *Encyclopedia of Database Systems*. Springer Verlag, Berlin, Germany.

Hmelo, C. E. and Guzdial, M. 1996. Of black and glass boxes: scaffolding for doing and learning. In Proceedings of the 1996 international Conference on Learning Sciences, Evanston, Illinois, July 1996. D. C. Edelson and E. A. Domeshek, Eds. International Conference on Learning Sciences. International Society of the Learning Sciences, 128-134.

Hodges, A. 2000. *Alan Turing: The Enigma*. Walker & Company, New York, NY.

Hoffman, M. E., Dansdill, T., and Herscovici, D. S. 2006. Bridging writing to learn and writing in the discipline in computer science education. SIGCSE Bull. 38, 1 (Mar. 2006), 117-121.

Hofstadter, D. 1996. Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought, Basic Books, Inc., New York, NY. Holcombe, John C. (2007).

Hohmann, L., Guzdial, M. and Soloway, E. 1992. SODA: A computer-aided design environment for the doing and learning of software design. In Proceedings of Computer assisted learning: 4th international conference, Berlin, Germany, 1992. Springer-Verlag, Berlin, Germany, 307-319.

Holland, S., Griffiths, R. and Woodman, M. 1997. Avoiding Object Misconceptions. In ACM SIGCSE Technical Symposium, San Jose, CA, 1997. *ACM SIGCSE Bulletin*, 131-134.

Howe, D. C. and Nissenbaum, H. 2008. TrackMeNot: Resisting Surveillance in Web Search. In *On the Identity Trail: Privacy, Anonymity and Identity in a Networked Society*, I. Kerr, C. Lucock and V. Steeves, Eds. Oxford University Press, Oxford, forthcoming.

Howe, D. C. and Soderman, A. B. 2009. The Aesthetics of Generative Literature: Lessons from a Digital Writing Workshop. *Hyperrhiz Journal of New Media Cultures*, forthcoming.

Howe, D.C. 2009. Programming for Digital Art and Literature. Available at http://www.rednoise.org/pdal/. Accessed 06/09/09.

Iverson, K. E. 1983. APL syntax and semantics. SIGAPL APL Quote Quad 13, 3 (Mar. 1983), http://doi.acm.org/10.1145/390005.801221.

Jelinek, F. 1997. *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, MA.

Jenkins, T. 2002. On the Difficulty of Learning to Program. University of Leeds, Leeds, UK. Available at http://www.psy.gla.ac.uk/~steve/localed/jenkins.html. Accessed 08/01/09.

Joshi, A.K. Levy, L. and Takahashi, M. 1975. Tree adjunct grammars. *Journal of the Computer and System Sciences*, 10:136–163.

Joubert, M. M. 2001. The Art of Creative Teaching: NACCE and Beyond in A. Craft, B. Jeffrey and M. Leibling (eds) *Creativity in Education*. London: Continuum.

Jurafsky, D. S. and Martin, J. H. 2000. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice-Hall.

Kac, E. 2007. From ASCII to Cyberspace: a Trajectory in Digital Poetry. In *Media Poetry: an International Anthology* (Second Edition), Bristol: Intellect, 2007, pp. 45-65.

Kafai, Y. 1995. *Minds in Play: Computer Game Design as a Context for Children's Learning*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Kay, A. and Goldberg, A. 1977. Personal Dynamic Media. *Computer* 10, 31-41.

Kay, D. G. 1998. Computer scientists can teach writing: an upper division course for computer science majors, *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, p.117-120, February 26-March 01, 1998, Atlanta, Georgia.

Kelleher, C., Pausch, R. and Kiesler, S. 2007. Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. CHI 2007 Proceedings. Programming By & With End-Users, San Jose, CA, May 2007.

Kelleher, C. and R. Pausch. 2006. Lessons Learned from Designing a Programming System to Support Middle School Girls Creating Animated Stories. IEEE Symposium on Visual Languages and Human-Centric Computing.

Kelly, C. 2009. Cracked Media: The Sound of Malfunction. The MIT Press. Cambridge, MA

Kenner, H., and O'Rourke, J. 1984. A travesty generator for micros: Nonsense imitation can be disconcertingly recognizable. Byte 9(12):129–131, 449–469.

Kidder, T. 1981. *The Soul of a New Machine*. Little, Brown, and Company, Boston, MA, 86.

Kenner, H. and Hartman, C. O. 1995. *Sentences*. Sun & Moon Press, Los Angeles, CA.

Knowles, Alison. (n.d.) Online biography. Available at http://www.lefthandbooks.com/knowles_bio.html. Accessed 16 Jan 2006

Knowles, A. and Tenney, J. A Sheet from "The house", a computer poem. In *Cybernetic Serendipity: The Computer and the Arts*. Ed. Jasia Reichardt. London: Studio International (special issue), 1968. 56.

Kussmaul, C. 2005. Using agile development methods to improve student writing. In the *Journal of Computing Sciences in Colleges*, v.20 n.3, p.148-156, February 2005.

Ladd, Brian C. 2003. It's all writing: experience using rewriting to learn in introductory computer science, *Journal of Computing Sciences in Colleges*, v.18 n.5, p.57-64, May 2003

Larsen, D. The Electronic Literature Studio: A List of Tools for Creating. In Fundamentals : Rhetorical Devices for Electronic Literature. Available at http://www.deenalarsen.net/fundamentals/tools.html. Accessed 08/12/09.

Lave J. 1993. *Tailored Learning: Education and Everyday Practice among Craftsmen in West Africa.* Unpublished manuscript, University of California, Berkeley, CA.

Le Lionnais, F. 1986. Lipo: First manifesto. In *Oulipo: A Primer of Potential Literature*. W. Motte, Ed. University of Nebraska Press, Lincoln, NE, 26–28.

Lescure, J. 1986. A brief history of the oulipo. In *Oulipo: A Primer of Potential Literature*, W. F. Motte, Ed. University of Nebraska Press, Lincoln, NE.

Levenshtein, V. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8), 707-710.

Levy, R. P. 2001. A computational model of poetic creativity with neural network as measure of adaptive fitness. In Bento, C. and Cardoso, A., editors, *Proceedings of the Fourth International Conference on Case Based Reasoning* (ICCBR'01) Workshop on Creative Systems: Approaches to Creativity in AI and Cognitive Science, Vancouver, Canada.

Liu, H. 2004. MontyLingua: An end-to-end natural language processor with common sense. Available at: web.media.mit.edu/~hugo/montylingua. Accessed 08/01/09.

Loveless, A. (2002) Literature review in creativity, new technologies and learning. Report 4: A report for NESTA Futurelab. Available at http://www.nestafuturelab.org/research/reviews/cr01.htm. Accessed 19 May 2008.

Lutz, T. 2003. Stochastic Texts. M. Freitag, Tr. 18 July 2003. http://www.reinhard-doehl.de/poetscorner/lutz1.htm.

Mac Low(b), M., 1997. "Science, Technology, and Poetry: Some Thoughts on Jackson Mac Low" in Festschrift for Jackson Mac Low's 75th Birthday, Crayon ([Fall] 1997).

Mac Low, J. 1994. *42 Merzgedichte In Memoriam Kurt Schwitters*. Station Hill Publishers, Barrytown, NY.

Mac Low, J. 1998. Response to Piombino. Available at http://epc.buffalo.edu/authors/maclow/piombino.html. Accessed 6/1/2009.

Maeda, K., Bird, S., Ma, X., and Lee, H. 2001. The annotation graph toolkit: software components for building linguistic annotation tools. In Proceedings of the First international Conference on Human Language Technology Research (San Diego, March 18 - 21, 2001). Human Language Technology Conference. Association for Computational Linguistics, Morristown, NJ, 1-6.

Maeda, J. 1999. *Design by Numbers*. MIT Press, Cambridge, MA.

Mak, B. 2001. Learning art with computers - a LISREL model. *Journal of Computer Assisted Learning* 17: 94 - 103.

Makela, T., Tarkka, M., Czegledy, N., Diamond, S. Donovan, A., Ferran, B., Jennings, P., Moller M., Samola, J., Schwarz, M., Sengupta, S., Le Sourd M. 2004. *The Helsinki Agenda: Strategy Document on International Development of New Media Culture Policy*. Available at http://www.ifacca.org/files/040916Helsinki_agenda_final.pdf. Accessed 8/20/09.

Manning, C. D., and Schütze, H. 1999. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, MA, 1999.

Manovich, L. 2001. *The Language of New Media*. Leonardo Books, The MIT Press, Cambridge, MA.

Manurung, H. M. 2003. An evolutionary algorithm approach to poetry generation. Ph.D. thesis, University of Edinburgh, Scotland.

Manurung, H.M., Ritchie, G. and Thompson, H. A 2000. Flexible Integrated Architecture for Generating Poetic Texts. *Informatics Research Report*, EDI-INF-RR-0016, Division of Informatics, U. of Edinburgh, May 2000.

Markov, A.A. 1913. Primer statisticheskogo issledovanija nad tekstom 'evgenija onegina' illjustrirujuschij svjaz' ispytanij v tsep / An example of statistical study on the text of 'Eugene Onegin' illustrating the linking of events to a chain. Izvestija Imp. Akademii nauk, serija VI(3), 153–162.

Marzal, A. and Vidal, E. 1993. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):926–932.

Masterman, M. and Wood, R. M. 1968. Computerized Japanese Haiku. *Cybernetic Serendipity: The Computer and the Arts*, J. Reichardt, Ed. Studio International (Special Issue), London, UK, 54.

Mateas, M. 2001. Expressive AI: A Hybrid Art and Science Practice. *Leonardo* 34.2. 2001 p147-153.

Mateas, M. 2002. Interactive drama, art and artificial intelligence. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Mateas, M. 2003. Expressive AI: Games and Artificial Intelligence. In Proceedings of Level Up - Digital Games Research Conference. Utrecht, Netherlands, Nov. 2003.

Mateas, M. and Stern, A. 2004. Natural Language Processing In Facade: Surface-text Processing In Technologies for Interactive Digital Storytelling and Entertainment (TIDSE), Darmstadt, Germany, June 2004.

Mateas, M. 2004. Procedural Literacy: An Idea Whose Time has Come (43 years ago). Post on *Grand Text Auto.* Available at: http://grandtextauto.org/2004/06/02/procedural-literacy-an-idea-whose-time-has-come-43-years-ago/. Accessed 08/12/09.

Mateas, M. 2005. Procedural Literacy: Educating the new media practitioner. In *On the Horizon: Special Issue on Future Strategies for Simulations, Games and Interactive Media in Educational and Learning Contexts*, 3(2).

Matthews, H. and Brotchie, A., Eds. 1998. *Oulipo Compendium*. Atlas Press, London, UK.

Mayer, R. E. 1999. Fifty Years of Creativity Research. In *Handbook of Creativity*, R. J. Sternberg, Ed. Cambridge University Press, Cambridge, UK.

McGrenere, J. and Wayne, H. 2000. Affordances: Clarifying and Evolving a Concept. In Proceedings of Graphics Interface, May 2000, Montreal, Quebec, Canada, 179-186.

Mellish, C. and Dale, R. 1998. Evaluation in the context of natural language generation. *Computer Speech and Language*, 12(4):349–373.

Meehan, J. R. 1976. The Metanovel: Writing stories by computer. Ph.D. thesis, Yale University, New Haven, CT.

Minnen, G., Carroll, J., and Pearce, D. 2001. Applied Morphological Processing of English. *Natural Language Engineering*, 7(3), 207—22.

Mitchell, W. J., Inouye, A S., and Blumenthal, M. S. Editors, Committee on Information Technology and Creativity, 2003. Beyond Productivity: Information, Technology, Innovation, and Creativity. National Academy Press, Washington, DC.

Milic, L. T. 1970. The Possible Usefulness of Poetry Generation. Non-Journal Publication. ERIC# ED055471.

Mitchell, W. J., Inouye, A. S., & Blumenthal, M. S. 2003. *Beyond Productivity: Information Technology, Innovation, and Creativity*, The National Academies Press, Washington, DC.

Montfort, N. 2004. Continuous Paper: The Early Materiality and Workings of Electronic Literature. Talk presented at the MLA Convention, Philadelphia, PA, December 2004. Available at http://nickm.com/writing/essays/continuous_paper_mla.html. Accessed 6/1/2009.

Mordechai, B. 2001. Constructivism in computer science Education. *Journal of Computers in Mathematics and Science Teaching*, 20(1), 45-73.

Motte, W. F. 1986. *Oulipo: A primer of potential literature*. University of Nebraska Press, Lincoln, NE.

Murray, J. H. 1997. *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*. The MIT Press, Cambridge, MA.

National Science Foundation. Investing in America's Future: Strategic Plan 2006-2011. Arlington, VA, 2006.

Nickerson, R. S. 1999. Enhancing Creativity. In *Handbook of Creativity*, R. J. Sternberg, Ed. Cambridge University Press, Cambridge, UK.

Nickerson, R., Landauer, T., 1997. Human–computer interaction: background and issues. In: Helander, M., Landauer, T., Prabhu, P. (Eds.), *Handbook of Human–Computer Interaction*, second ed. Elsevier, Amsterdam.

Norman, D. A. 1988. *The Psychology of Everyday Things*. Basic Books, New York, NY.

Norman, D. A. 1990. *The Design of Everyday Things*. Doubleday, New York, NY.

Papert, S. 1980. *Mindstorms: Children, Computer, and Powerful Ideas*. Basic Books, New York, NY.

Papert, S. 1991. Situating Constructionism. In *Constructionism: Research Reports and Essays, 1985-1990*. I. Harel and S. Papert, Eds. Ablex Publishing Corporation, Norwood, NJ, 1-11.

Parsons, D. and Haden, P. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In Proceedings of ACE 2006, Hobart, Australia, January 2006. 157-163

Paz, T. 1996. Computer science for vocational high-school studuents: Processes of learning and teaching. Masters thesis, Technion—Israel Institute of Technology (in Hebrew).

Perez y Perez, R., and Sharples, M. 2004. Three computer-based models of storytelling: BRUTUS, MINSTREL and MEXICA. *Knowledge-Based Systems*, 17(1), 15–29.

Perlin, K, Flanagan M., and Hollingshead, A. 2003. RAPUNSEL MANIFESTO. Available at http://www.maryflanagan.com/rapunsel/manifesto.htm. Accessed 06/03/2009.

Perloff, Marjorie. 1994. *Radical Artifice: Writing Poetry in the Age of Media*. University of Chicago Press, Chicago, IL.

Pesante, L. H. 1991. Integrating writing into computer science courses. In *Proceedings of the* Twenty-Second SIGCSE Technical Symposium on computer science Education (San Antonio, Texas, United States, March 07 - 08, 1991). SIGCSE '91. ACM, New York, NY, 205-209.

Pfaffenberger, B. 1992. Technological Dramas. In *Science, Technology & Human Values* 17, no. 3 (1992): 282–312.

Phillips, D.C. 1990. Subjectivity and objectivity: An objective inquiry. In *Qualitative Inquiry in Education: The Continuing Debate*, E.W. Eisner & A. Peshkin, Eds. Teachers College Press, New York, NY, 19-37.

Plass, J.L., Goldman, R., Flanagan, M., Diamond, J., Song, H., Rosalia, C., and Perlin, K. 2007. RAPUNSEL: How a computer game designed based on educational theory can improve girls' self-efficacy and self-esteem. Paper presented at the 2007 Annual Meeting for the American Educational Research Association (AERA), Division C-5: Learning Environments, Chicago, IL, 2007.

Prensky, M. 2008. Programming Is the New Literacy. *Edutopia magazine*. February 2008 online issue. Available at http://www.edutopia.org/literacy-computer-programming. Accessed 08/07/09.

Queneau, R. 1961. *Cent mille milliards de poemes*. Paris: Gallimard, Paris, France.

Queneau, R. 1984. *The Policeman's Beard is Half Constructed: Computer Prose and Poetry.* Warner Books Inc., New York, NY.

Ragonis, N. and Ben-Ari, M. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *computer science Education*, 15(3), 203-221.

Reas, C. and Fry, B. 2007. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, Cambridge, MA.

Reichardt, J. 1968. Cybernetic Serendipity: The Computer and the Arts. *Studio International* (Special Issue), London, UK, 56.

Reichardt, J. 1971. *The Computer in Art*. Studio Vista, London, UK; Van Nostrand Reinhold Company, New York, NY.

Reichardt, J. (n.d.) Cybernetic Serendipity. Media Art Net, London. Available at http://www.medienkunstnetz.de/exhibitions/serendipity/. Accessed 07/04/09.

Reiter, E. and Dale, R. 2000. *Building Natural Language Generation Systems*. Cambridge University Press, Cambridge, UK.

Resnick, M., Bruckman, A., and Martin, F. 1996. Pianos Not Stereos: Creating Computational Construction Kits. *Interactions*, vol. 3, no. 6 (September/October 1996).

Resnick, M., Berg, R. and Eisenberg, M. 2000. "Beyond Black Boxes: Bringing Transparency and Aesthetics Back to Scientific Investigation," *Journal of the Learning Sciences.* 2000. 9(1). pp. 7-30.

Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B., Pausch, R., Selker, T. and Eisenberg, M. 2005. Design Principles for Tools to Support Creative Thinking. *NSF Workshop Report on Creativity Support Tools*, Washington, DC, June 2005, 37-52.

Resnick, M. 2007. Sowing the Seeds for a More Creative Society. Learning and Leading with Technology, December 2007.

Resnick, M. 2007(b). All I Really Need to Know (About Creative Thinking) I Learned (By Studying How Children Learn) in Kindergarten. ACM Creativity & Cognition conference, Washington DC, June 2007.

Resnick, M. 2009. Kindergarten is the Model for Lifelong Learning. *Edutopia*, June 2009.

Rettalack, J., Ed. 1996. *MUSICAGE: Cage Muses on Words, Art, Music (John Cage in Conversation with Joan Retallack).* Wesleyan University Press, Hanover, NH.

Réty, J.-H., Bouchardon, S., Clément, J., and Szilas, N. 2008. An experimental tool for digital literature. Paper presented at *Elit in Europe*. Bergen, Norway, September, 2008.

Ritchie, G. 2001. Assessing creativity. In *Proceedings of the AISB'01 Symposium on Artificial Intelligence and Creativity in Arts and Science*, pages 3–11, York, UK. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour.

Ritchie, G. 2006. The Transformational Creativity Hypothesis, *New Generation Computing*, special issue on Creative Systems, 2006.

Romero, P., Good, J., Robertson, J., du Boulay, B., Reid, H. and Howland, K. 2007. Embodied interaction in authoring environments, *Second International Workshop on Physicality*, Lancaster, September 2007.

Rose, C. 2006. *Five Essays On Design*. Aardvark Publishing. Sandy, Utah.

Rosenberg, J. 2008. Codework: Starting Points. Wiki for *Codework: Exploring relations between creative writing practices and software engineering.* A workshop sponsored by the National Science Foundation. West Virginia University, April 3-6, 2008. Available at http://clc.as.wvu.edu:8080/clc/CodeworkWorkshopWiki. Accessed 08/08/09.

Rosenfeld R. 2000. Two decades of statistical language modeling: Where do we go from here? In *Proceedings of IEEE*:88(8).

Roubaud, J. 1986. Mathematics in the method of Raymond Queneau. In *Oulipo: A primer of potential literature*, W. F. Motte, Ed. University of Nebraska Press, Lincoln, NE.

Roubaud, J. 1998. The oulipo and combinatorial art. In *Oulipo compendium*, H. Mathews and A. Brotchie, Eds. Atlas Press, London, UK, 37–44.

Russell, S. and Norvig, P. 1994. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Saddle River, NJ.

Sabar Ben-Yehoshua, N. 2001. The history of qualitative research – influences and directions. In *Qualitative Research: Genres and Traditions in Qualitative Research*, N. Sabar, Ed. Zmora Bitan, Tel-Aviv, Israel, 13-16.

Samurçay, R. 1989. The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In *Studying the novice programmer*, E. Soloway & J.C. Spohrer, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 161-178.

Schank, R. C., and Cleary, C. 1995. *Engines for education*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Schank, R.C., and Riesbeck, C.K. 1981. The theory behind the programs: A theory of context. In *Inside computer understanding: Five programs plus miniatures* (Artificial Intelligence Series), R. C. Schank and C. K. Riesbeck, Eds., Lawrence Erlbaum Associates, Hillsdale, NJ, 27–40.

Schoen, D.A. 1983. *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York, 1983.

Schunk, D. and Zimmerman, B. 1998. *Self-Regulated Learning: From Teaching to Self-Reflective Practice*. The Guilford Press, New York, NY.

Selker, T. 2005. Fostering Motivation and Creativity for Computer Users, *Journal of Human-Computer Studies* 63, 4-5, *Special Issue on Computer Support for Creativity*, Edmonds, E. Candy, L. (Eds.), 2005. pp. 410-421.

Seo, J., and Shneiderman, B. 2006. Knowledge Discovery in High-Dimensional Data: Case Studies and a User Survey for the Rank-by-Feature Framework. *IEEE Transactions on Visualization and Computer Graphics* 12, 3 (May. 2006), 311-322.

Shaffer, D.W. and Resnick, M. 1999. Thick authenticity: new media and authentic learning. *Journal of Interactive Learning Research*, 10(2), 195–215.

Shannon, C. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal* 27. 379–423.

Shannon, C. and Weaver, W. 1949. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, IL.

Sharples, M. 1996. An account of writing as creative design. In Levy, C. M. and Ransdell, S. E., editors, *The Science of Writing: Theories, Methods, Individual Differences, and Applications*. Lawrence Erlbaum Associates, Mahwah, USA.

Sheil, B. A. 1980. Teaching procedural literacy (Presentation Abstract*). In Proceedings of the ACM 1980 Annual Conference* ACM '80. ACM, New York, NY, 125-126.

Shneiderman B. 1977. Teaching programming: A spiral approach to syntax and semantics. *Computers and Education* 1, 193-197.

Shneiderman, B. *Leonardo's Laptop: Human Needs and the New Computing Technologies*, MIT Press, Cambridge, MA, 2002

Shneiderman, B., Fischer, G., Czerwinski, M., Resnick, M., Myers, B. 2006. Creativity Support Tools: Report from a U.S. National Science Foundation Sponsored Workshop. *International Journal of Human-Computer Interaction*, 20(2), 2006, p. 61-77.

Shneiderman, B. 2007. Creativity support tools: accelerating discovery and innovation. *Communications of the ACM* 50, 12 (Dec. 2007), p20-32.

Sinker, R. 2000. Making Multimedia: Evaluating young people's creative multimedia production. In J.Sefton-Green and R. Sinker (eds) *Evaluating Creativity*. London: Routledge.

Sleeman, D., Putnam, R. T., Baxter, J. A. and Kuspa, L. 1988. An introductory Pascal class: A case study of student errors. In *Teaching and Learning Computer Programming*, R.E. Mayer, Ed. Lawrence Erlbaum Associates, Hillsdale, NJ, 237-257.

Soloway E., Ehrlich K., Bonar J. and Greenspan J. 1982. What do novices know about programming? In *Directions in Human-Computer Interaction*, A. Badre, B. Shneiderman, Eds. Ablex Publishing, Norwood, NJ.

Soloway E. 1986. Learning to program – learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9), 850-858.

Soloway, E., and Spohrer, J.C., Eds. 1989. *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Soloway E. 1993. Should we teach students to program? *Communications of the ACM*, 36(10), 21-24.

Spohrer, J.C. 1989. MARCEL: A Generate-Test-and-Debug (GTD) Impasse/Repair Model of Student Programmers. Unpublished Ph.D. dissertation. Yale University, New Haven, CT.

Spohrer, J.C. and Soloway, E. 1985. Putting it all together is hard for novice programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Hancock, PA, 1985.

Sternberg, R. J. 1999. *Handbook of Creativity*. Cambridge University Press, Cambridge, UK.

Stone, M., Doran, C. 1997. Sentence planning as description using tree-adjoining grammar. In *Proceedings of ACL*, pages 198–205.

Stone. M. 2002. Lexicalized Grammar 101. ACL Workshop on Tools and Methodologies for Teaching Natural Language Processing, pages 76-83.

Stone, M., Doran, C. Webber, B., Bleam, T., and Palmer, M. 2003. Microplanning with communicative intentions: the SPUD system. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.20.8123. Accessed 08/01/09.

Strachey, C. 1952. Logical or non-mathematical programmes. In Proceedings of the 1952 ACM National Meeting, Toronto, CA, 1952. ACM Press, New York, NY, 46–49.

Strachey, C. 1954. The "thinking" machine. *Encounter*, 3(4), 25–31.

Szilas N., Marty O., Réty J.-H. 2003. Authoring Highly Generative Interactive Drama. In *Proceedings of International Conference on Virtual Storytelling*, Toulouse, 2003.

Taylor, P. A., Black, A. and Caley, R. 1998. The architecture of the festival speech synthesis system. In The Third ESCA Workshop in Speech Synthesis, pages 147-151, Jenolan Caves, Australia, 1998.

Turner, G. 2007. Supportive Methodology and Technology for Creating Interactive Art. Unpublished Ph.D. Dissertation. M. Comp. (Hons) Loughborough University, Leicestershire, UK.

Turing, A. M. 1936. On computable numbers with an application to the entscheidungs problem. *Proceedings of the London Mathematical Society* 2(42).

Turner, S. R. 1994. *The Creative Process: A Computer Model of Storytelling and Creativity*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Van Wyk, C. J. 1995. Programming as writing: using portfolios. *SIGCSE Bull*. 27, 4 (Dec. 1995), 39-42.

van Rijsbergen, C. J., Robertson, S. E. and Porter, M. F. 1980. *New models in probabilistic information retrieval*. British Library Research and Development Report no. 5587. British Library, London, UK.

Vos, E. 1996. New Media Poetry - Theories and Strategies, In *Visible Language*, Vol. 30, N. 2, 1996, 215-233.

Wall, L. and Schwartz, R. 1988. *Programming Perl*. O'Reilly and Associates Inc.

Wallace, R. S., 2009. From Eliza to A.L.I.C.E. Available at http://www.alicebot.org/articles/wallace/eliza.html. Accessed 6/11/2009.

Ward, N. 1994. A Connectionist Language Generator. *Ablex Series in Artificial Intelligence*. Ablex Publishing, Norwood, USA.

Wardrip-Fruin, N., and Montfort, N., Eds. 2003. *The New Media Reader*. MIT Press, Cambridge, MA.

Wardrip-Fruin, N. 2006. Expressive Processing: On Process-Intensive Literature and Digital Media. Unpublished Dissertation, Brown University, Providence, RI.

Wardrip-Fruin, N. 2009. Introduction to *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*., MIT Press, Cambridge, MA. pp 1-22.

Noah Wardrip-Fruin

Watten, B. 1992. Poetic Vocabularies: A Conversation between Barrett Watten and Jackson Mac Low, KPFA, Berkeley, CA 24 October 1985 (Edited and revised by Watten and Mac Low in 1992), *Aerial*, 8.

Weide, R. L. (1996). Carnegie Mellon University pronouncing dictionary. Available at http://www.speech.cs.cmu.edu/cgi-bin/cmudict. Accessed on 04/01/2006.

Weizenbaum, Joseph. 1966. ELIZA — A Computer Program for the Study of Natural Language Communication Between Man and Machine. *Communications of the ACM*, 9(1), 36-45.

Weizenbaum, J. 1976. *Computer Power and Human Reason: From Judgment to Calculation.* W. H. Freeman, New York, NY.

Wing, J. M. 2006. Computational thinking. *Communications of the ACM*, 49(3), 33-35.

Wilson, S. 2002. *Information Arts: Intersections of Art, Science and Technology*. MIT Press, Cambridge, MA.

Wolff, M. 2007. Reading Potential: The Oulipo and the Meaning of Algorithms. *Digital Humanities Quarterly*, 1(1).

Yeomans, M. 1996. Creativity in Art and Science: A Personal View. *Journal of Art and Design Education*: 241 250.

Zeki, S. 2001. Artistic Creativity and the Brain. *Science*. 293: 51.