Procedural Literacy: Educating the New Media Practitioner

Michael Mateas

Abstract

For humanities scholars, artists and designers, computer programming can seem a narrow technical skill, one that has no bearing on the theoretical and aesthetic concerns of new media. New media scholars and practitioners, including game designers and game studies scholars, may assume that the "mere" technical details of code can be safely bracketed out of the consideration of the artifact. Contrary to this view, I argue that procedural literacy, of which programming is a part, is critically important for new media scholars and practitioners, that its opposite, procedurally illiteracy, leaves one fundamentally unable to grapple with the essence of computational media. This paper looks at one of the earliest historical calls for universal procedural literacy, explores how games can serve as an ideal object around which to organize a procedural literacy curriculum, and describes a graduate course developed at Georgia Tech, Computation as an Expressive Medium, designed to be a first course in procedural literacy for new media practitioners.

Introduction

For humanities scholars, artists and designers, computer programming can seem a narrow technical skill, a mere familiarity with the latest fads and facility with the latest jargon of the computer industry. In this view, programming has almost no connection with theoretical and philosophical considerations, with concept and aesthetics, with a design focus on human action and interpretation. This attitude is often adopted by new media scholars and practitioners, including game designers and game studies scholars, who may assume that the "mere" technical details of code can be safely bracketed out of the consideration of the artifact. In this paper I argue that, contrary to this view, procedural literacy, of which programming is a part, is critically important for new media scholars and practitioners, that its opposite, procedurally illiteracy, leaves one fundamentally unable to grapple with the essence of computational media. In fact, one can argue that procedural literacy is a fundamental competence for everyone, required full participation in contemporary society, that believing only programmers (people who make a living at it) should be procedurally literate is like believing only published authors need to learn how to read and write; here I will restrict myself to the case of new media scholars and practitioners.

By procedural literacy I mean the ability to read and write processes, to engage procedural representation and aesthetics, to understand the interplay between the culturally-embedded practices of human meaning-making and technically-mediated processes. With appropriate programming, a computer can embody any conceivable process; code is the most versatile, general process language ever created. Hence, the craft skill of programming is a fundamental component of procedural literacy, though it is not the details of any particular programming language that matters, but rather the more general tropes and structures that cut across all languages.

Without an understanding of how code operates as an expressive medium, new media scholars are forced to treat the operation of the media artifacts they study as a black box,

losing the crucial relationship between authorship, code, and audience reception. Code is a kind of writing; just as literary scholars wouldn't dream of reading translated glosses of work instead of reading the full work in its original language, so new media scholars must read code, not just at the simple level of primitive operations and control flow, but at the level of the procedural rhetoric, aesthetics and poetics encoded in a work.

Procedurally illiterate new media practitioners are confined to producing those interactive systems that happen to be easy to produce within existing authoring tools. Even those practitioners who don't themselves write much code will find themselves on interdisciplinary collaborative teams of artists, designers and programmers. Such collaborations are often doomed to failure because of the inability to communicate across the cultural divide between the artists and programmers. Only practitioners who combine procedural literacy with a conceptual and historical grounding in art and design can bridge this gap and enable true collaboration.

A number of current intellectual movements highlight the need for humanistic procedural literacy. In game studies there is a growing understanding that much of the meaning of a digital game, including the gameplay, and the rhetoric and ideology of the game, are encoded in the procedural rules. The nascent field of software studies has begun to explicitly explore code, and how code functions within culture. And of course for interactive art and design, procedurality lies at the heart of the meaning of interactive artifacts. New media artists, game designers and theorists, media and software studies theorists, and generally anyone involved in cultural production on, in or around a computer, should know how to read, write and think about programs.

Games, and more generally interactive art and entertainment, can play a crucial role in the education of the procedurally literate theorist and practitioner. Bridging the worlds of game education and educational games, students can construct procedural, interactive experiences, informed by theoretical readings and the study of successful procedural artifacts, as a means of gaining procedural literacy. Whether the student will go on to become a theorist in new media or software studies, a game designer, or a new media artist, the procedural literacy gained through the study and construction of games and other procedurally expressive artifacts will enable her to successfully grapple with the expressive power of computation. The rest of this paper looks at one of the earliest historical calls for universal procedural literacy, explores how games can serve as an ideal object around which to organize a procedural literacy curriculum, and describes a graduate course I've developed, Computation as an Expressive Medium, designed to be a first course in procedural literacy for new media practitioners.

Procedural Literacy: A (Very) Brief History
There are a number of contemporary educational projects organized around the notion of procedural literacy. Flanagan and Perlin have begun the Rapunzel project, an agent-based programming environment intended to appeal to middle school girls, precisely at an age when many girls, for a variety of social and cultural reasons, start loosing interest in math and science (Flanagan & Perlin). Guzdial has developed a university freshmen media computation course that introduces computer science from the perspective of

manipulating media objects such as still images, sound and video. His course is designed to address the high withdraw-or-failure rates in introductory computer science courses, particularly among women (Guzdial 2003). Maeda's Computational Aesthetics group at The Media Lab has developed a number of programming environments intended to facilitate visually oriented designers and artists in learning to program, including *Design By Numbers* (Maeda 1999) and *Processing* (Fry & Reas). In my own educational work, I'm developing courses on "programming for artists" for the Georgia Institute of Technology's undergraduate and graduate courses in computational media (I describe the graduate course, Computation as an Expressive Medium, in more detail below).

This recent work builds on a long tradition of work on universal procedural literacy: Papert's work teaching children to program in Logo, described in the 1980 book *Mindstorms* (Papert 1980), Kay and Goldberg's work on procedural environments in which everyone, including children, can build their own simulations, described in the 1977 paper *Personal Dynamic Media* (Kay & Goldberg 1977), and Ted Nelson crying in the wilderness that "you can and must understand computers NOW" in his 1974 *Computer Lib/Dream Machines* (Nelson 1987). However, the earliest argument I've seen for universal procedural literacy, pointed out to me by Guzdial (Guzdial & Soloway 2003), is one given by A.J. Perlis in a 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). The symposium consisted of 8 talks, with two discussants responding to each talk, and was attended by such luminaries as C. P. Snow, J. W. Forrester, Herb Simon, J. McCarthy, and A. J. Perlis. Perlis' talk, The Computer in the University, focused on the role the computer should play in a university education. Perlis' argument is worth reviewing in some depth, as he accurately identified a number of issues and concerns that are still with us today.

Perlis begins by describing current common uses of the computer in university settings, primarily for numerical analysis. He notes 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" (p. 186). He notes that most students learn to use computers in relatively haphazard ways, either driven by the need of some particular application, or in the context of a numerical analysis course that is primarily focused on teaching numerical methods, or on their own, or in a course that teaches some particular programming language. None of these approaches focus on the teaching of computation per se. In describing the programming language approach, for example, he writes:

> "A credit course involving the use of some automatic programming language is provided. Fluency in 'conversation' with this language and clear understanding of the language's grammar are the intent of such a course. Here, too, the approach suffers from limited and even misguided pedagogic objectives; and the result is a student well conversant in, say, Algol 60, but still very likely uneducated as to the scope of computation." (p. 187)

Note that 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. The procedural literacy revolution has really been more of a slow evolution. Certainly more people are able to control their computers in more ways now than they could 43 years ago. Scripting environments such as Flash or Director, VB, and back in the 1980's, Hypercard, enable more people to engage in some degree of procedural authorship. But all of these special purpose environments come at the price of obscuring the full expressive potential of computation. The next stage of procedural literacy is learning to navigate the huge tower of abstraction that exists in any computer system, with each layer defining its own little process universe, and with all layers, including the programming languages themselves, contingent human-authored artifacts, each carrying the meanings, assumptions, and biases of their authors, each offering a particular set of affordances.

Perlis goes on to describe 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 (a cultural grounding for knowledge, "sensitivity… a feeling for the meaning and relevance of facts" (p. 188)), 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…" (p. 188)) 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." (p. 188)). During the talk 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. Thus, for Perlis, procedural literacy lies at the heart of the fundamental aims of a university education. Consequently, he argues that all students should make contact with computers at the earliest time possible: the student's freshman year. For 1961 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, esoteric monsters. Even today, with the relative ubiquity of computers, most universities do not have such a requirement in place.

Admitting that the optimum content for such a course is not yet known (and, 43 years later, is still not known), he goes on to describe a two term course then being developed at Carnegie Tech (now Carnegie Mellon).

> "During the first term the students wrote programs in a symbolic machine
> code, the Carnegie TASS system; and during the current term they are
> writing their codes in GATE, the Carnegie Algebraic Language system.
> Coding in machine language, they are taught mechanical algorithms of
> code analysis that enable them to do manually what the GATE translator
> does automatically. In particular, they are becoming adept in decoding

complex logical relations to produce branching codes and in manual decoding of complex formula evaluations by mechanical processes. The intent is to reveal, through these examples, how analysis of some intuitively performed human tasks leads to mechanical algorithms accomplishable by a machine." (p. 189)

Notice that in this early course there's a focus not on particular languages or tools, but on how the computer can be transformed into any language or tool; computation is treated as a universal representational medium for describing structure and process. Achieving this level of procedural literacy for new media practitioners is a huge challenge; we don't want to simply teach specific tools or programming environments, but a general competence in computation as *the* medium for representing structure and process. Ideally, as in this early Carnegie Tech course, students would understand that even programming languages are just tools, that the space of computation is bigger than the particular view of it embodied (enforced) by any particular programming model (e.g. the sequential model of languages like C++ or Java, the eval-apply loop of languages like Lisp and ML, the search and unification process of languages like Prolog, etc.). While teaching machine code and having students write their own assembler in machine code is probably not the way to teach these concepts to new media practitioners, we need to find a way to get them across somehow.

Several respondents commented on Perlis' talk, including Peter Elias and J. C. R. Licklider. Elias disagrees with the fundamental importance of programming.

> "Perhaps our most serious difference is in predicting the ultimate state of affairs when time-shared computers are available on every campus and good symbolic processing languages are in use. By that stage it sounds to me as though Perlis would have programming assume a large role in the curriculum, while I should hope that it would have disappeared from the curricula of all but a moderate group of specialists."

> "I have a feeling that if over the next ten years we train a third of our undergraduates at M.I.T. in programming, this will generate enough worthwhile languages for us to be able to stop, and that succeeding undergraduates will face the console with such a natural keyboard and such a natural language that there will be very little left, if anything, to the teaching of programming…

> "I think that if we stop short of that, if it continues to demand as much effort to learn how to speak to machines as it costs us to teach students a course for a couple of semesters, then we have failed. I do not see anything built into the situation which requires as much as that." (p. 203)

Elias desires the development of frictionless tools that, like the computers on Star Trek, allow us to make the computer do our bidding wit h little work (e.g. "Computer, gather data on the anomaly, correlate it with all known space phenomena, and suggest a course

of action." Computer: "Done"). The problem with this vision is that programming is really about describing processes, describing complex flows of cause and effect, and given that it takes work to describe processes, programming will always involve work, never achieving this frictionless ideal. Any tools that reduce 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. A procedurally literate practitioner will still make use of specific tools for specific projects, but will be aware of the constraints of specific tools, will be capable of considering a space of computational possibility larger than any specific tool.

Licklider responds:

> "Pete [Elias], I think the first apes who tried to talk with one another decided that learning language was a dreadful bore. They hoped that a few apes would work the thing out so the rest could avoid the bother. But some people write poetry in the language we speak. Perhaps better poetry will be written in the language of digital computers of the future than has ever been written in English." (p. 204)

What I like about this is the recognition that computer languages are expressive languages; programming is a medium. Asking that programming should become so "natural" as to require no special training is like asking that reading and writing should become so natural that they require no special training. Expressing ideas takes work; regardless of the programming language used (and the model of computation implicit in that programming language), leaning how to express oneself in code will always take work.

In Perlis' response he clarifies his argument as to the central importance of procedural literacy.

> "Perhaps I may have been misunderstood as to the purpose of my proposed first course in programming. It is not to teach people how to program a specific computer, nor is it to teach some new languages. The purpose of a course in programming is to teach people how to construct and analyze processes. I know of no course that the student gets in his first year in a university that has this as its sole purpose."

> "This, to me, is the whole importance of a course in programming. It is a simulation. The point is not to teach the students how to use Algol, or how to program the 704. These are of little direct value. The point is to make the students construct complex processes out of simple ones (and this is always present in programming), in the hope that the basic concepts and

abilities will rub off. A properly designed programming course will develop these abilities better than any other course." (p. 206)

Here Perlis makes it clear that programming is a medium, in fact *the medium* peculiarly suited for describing processes, and as such, a fundamental component of cultural literacy, and a fundamental skill required of new media practitioners and theorists.

Procedural Literacy and Games
OK, assuming at this point that we agree that new media folk should be procedurally literate, how should we achieve this literacy? Just throwing new media students into introductory CS courses is inappropriate. Such courses tend to focus on abstract features of computation, such as recursion, environments, scope, and so forth, without relating them to the design and analysis of digital media. The examples used in such courses tend to focus on engineering, mathematical and business applications (e.g. teaching recursion using the Fibonacci sequence, teaching functional abstraction using examples from physics, teaching object-oriented design using simple database-like models of people with attributes such as name and age). And the culture of such courses, the implicit background against which the material is taught, tends to be the technophilic culture of the adolescent male geek, emphasizing narrow technical mastery disconnected from broader social and cultural issues. In addition to not emphasizing computation as a medium, the culture and assumed student background of such courses tend to alienate the new media student, further emphasizing the "two-cultures" divide, the gap between engineering/science and art/humanities that is precisely the gap we're trying to close.

It is important not to view computation for new media students as a dumbed-down version of the traditional computer science courses. Teaching programming for artists and humanists shouldn't merely be simplified computer science with lots of visually engaging examples, but rather an alternative CS curriculum. Traditional CS courses tend to emphasize programming as a kind of reified mathematics, emphasizing mathematical abstractions and formal systems. For new media students we need to emphasize that, while programming does have its abstract aspects, it also has the properties of a concrete craft practice. In a practice that feels like a combination of writing and mechanical tinkering, programmers build elaborate Rube Goldberg machines. In fact, the expressive power of computation lies precisely in the fact that, for any crazy contraption you can describe in detail, you can turn the computer into that contraption. What makes programming hard is the extreme attention to detail required to realize the contraption. A "loose idea" is not enough - it must be fully described in great detail before it will run on a computer. A New Media introduction to CS *should* be a difficult course, with the challenge lying not in programming conceived of as applied mathematics, but in connecting new media theory and history with the concrete craft practice of learning to read and write complex mechanical processes.

Games can serve as an ideal object around which to organize a new media introduction to CS. Games immediately force a focus on procedurality; a game defines a procedural world responsive to player interaction. Additionally, unlike other procedurally intensive programs such as image manipulation tools or CAD systems, games force a simultaneous

focus on simulation *and* audience reception. A game author must build a dynamic, real-time simulation world such that, as the player interacts in the world, they have the experience desired by the author. Unlike the design of other software artifacts that minimize the authorial voice, maintaining an illusion of neutrality, games foreground the procedural expression of authorial intentionality in an algorithmic potential space. Of course other kinds of software, such as image manipulation tools and network protocols, are not truly neutral, but rather can be unpacked in order to read the mark of the author, her implicit world view and ideology. But students may best understand computation as a procedural medium by starting with a software form, such as games, which makes this explicit.

Rather than using the computer as a playback device for more traditional media assets such as sound and still and moving imagery, games are a native computational form; code defines the game's response to player interaction. To describe the relationship between computation and media assets, Chris Crawford introduced the term *process intensity* (Crawford 1987). Process intensity is the "crunch per bit", the ratio of computation to the size of the media assets being manipulated by the system. If a game (or any interactive software) primarily triggers media playback in response to interaction, it has low process intensity. The code is doing very little work - it's just shoveling bits from the hard drive or CD-ROM to the screen and speakers. As a game (or any interactive software) manipulates and combines media assets, its process intensity increases. Algorithmically generated images and sound that make no use of assets produced offline have maximum process intensity. Process intensity is directly related to richness of interactivity. As process intensity decreases, the author must produce a greater number of offline assets (e.g. pre-rendered animations or video) to respond to the different possible interactions. As the number of offline assets required to maintain the same level of interactivity tends to increase exponentially as process intensity decreases, in general decreases in process intensity result in decreases in the richness of interactivity. Games such as *Dragon's Lair* that structure interaction primarily through media playback rather than procedurality are the exceptions that prove the rule. After a brief popularity driven by their graphic richness relative to contemporaneous games, such video playback games disappeared from the gaming landscape.

As described at the beginning of this article, procedural literacy is not just the craft skill of programming, but includes knowing how to read and analyze computational artifacts. Because the procedural structure of games is the essence of the game medium (not mere "technical detail"), teaching procedural literacy through the creation of games is not intended merely as training for future game programmers, but as a process intensive training ground for anyone interested in computation as a medium. The fundamentally procedural nature of games can be seen by looking at the two sources of activity within a game: game AI and game physics (Mateas 2003). Game AI is a concerned with "intelligent" behavior, that is, behavior that the player can read as being produced by an intelligence with its own desires, behavior that seems to respond to the player's actions at a level connected to the *meaning* of the player's actions. Game AI produces the part of a game's behavior that players can best understand by "reading" the behavior as if it results from the pursuit of goals given some knowledge. Game physics deals with the "dead"

part of the game, the purely mechanical, causal processes that don't operate at the level of intentionality and thus don't respond to player activity at the level of meaning. A complete analysis of a game requires unpacking the procedural rules behind the AI and physics. Squire's analysis in this issue of the boss Hulk Davidson in *Viewtiful Joe* is an example of a procedural analysis focused on the game AI.

To explore the distinction between game AI and game physics, consider a game that implements a nice water simulation. If the player throws objects in the water, they float or sink depending on their density. If a flow of water is obstructed, the water backs up or flows around the obstruction. When the player jumps into the water, they produce a satisfying splash. The water thus responds to player action – the water behaves differently depending on what the player does. The water simulation is part of the game physics, not the game AI, despite the fact that the water's response is beautiful and/or realistic and the simulation code is complex. In order to understand the water, the player doesn't have to read psychological motivations into the water. The water always behaves the same, doesn't act like it has its own goals or desires, and doesn't respond to the player's actions as if these actions had any particular meaning for it. Contrast this with the ghosts in *Pacman*. In order to make sense of the ghosts' behavior, the player projects goals onto the ghosts (e.g. "they want to get me", "they are running away from me") and interprets the ghost behavior in terms of these goals. The ghosts support a psychological, intentional interpretation of their behavior, which the water simulation does not, even though the code for the water simulation may be much more complex than the ghost code. Game AI lies at the intersection of player perception (the player is able to read part of the game behavior as alive) and the game code that supports this perception.

But in the case of both game AI and game physics, the game's response to player interaction is process intensive, depending on algorithmic response rather than playback of media assets. Thus reading and writing games and game-like artifacts requires procedural literacy, making games an ideal artifact around which to organize a procedural literacy curriculum.

Teaching Computation as an Expressive Medium
For the last two years at Georgia Tech I've taught the graduate course *Computation as an Expressive Medium*, a graduate-level practical and theoretical introduction to programming organized around the creation of game-like artifacts (the Fall 2004 syllabus can be seen at www.lcc.gatech.edu/~mateas/courses/LCC6317Fall2004/Syllabus.html). While one of the goals of the course can be described as "programming for artists", the course doesn't focus only on craft skills, but combines theoretical readings in New Media theory with a consideration of the affordances and possibilities of computational media; that is, it seeks to teach procedural literacy. For the readings I use *The New Media Reader* (Wardrip-Fruin & Montfort 2003), a nice collection of historically significant writings in both New Media and Computer Science. I consider this course a prototype of the game-centric approach to teaching procedural literacy that I describe above. While the course projects include games and game-like artifacts, they also include projects such as procedural manipulation of web pages. In all cases the projects require students to think about concept (What's interesting about their project? Why are they doing it?), audience

reception (What experience and/or idea are you trying to convey? Can interactors figure out how to read and interact with the piece?) and programming.

Students come to the course from the master's program in Information Design and Technology, the Human-Computer Interaction master's program, and the Ph.D. program in Digital Media. One of the challenges in teaching the course is the diversity of backgrounds and programming experience students bring to the class; students come to the course with diverse backgrounds including psychology, fine arts, literary studies, graphic and industrial design, film studies, mathematics, computer science, physics and various engineering disciplines. Some students have never programmed before, even in scripting environments, while others have extensive programming experience. A simple way to reduce the diversity would be to excuse students with programming backgrounds from taking the course. However, I'm uncomfortable with doing this because the students with programming backgrounds are often unfamiliar with New Media theory and history and have not thought about programming in this context. Perhaps surprisingly, students with extensive programming backgrounds who have taken the course, including students with bachelor's degrees in CS, have all described the course as rewarding. A number of them have said that in previous programming classes they only got to work on "boring" programs, and never had the chance to think about programming as a medium, nor to relate programming to New Media theory and history. This reaction supports the idea that procedural literacy is broader than programming competence, and requires a new curriculum.

There are six projects during the semester, each of which is designed to exercise new programming concepts, explore different issues in code-based art, and coordinate with readings from the *New Media Reader*. The six projects are:

- *Display the progress of time in a non-traditional way.* The goal of this project is to start students thinking about procedural generation of imagery as well as responsiveness to input, in this case both the system clock, and potentially, mouse input.
- *Create your own drawing tool, emphasizing algorithmic generation/modification/manipulation.* The students in this course have all had experience with tools such as Photoshop, Premier or Director. The goal of this project is to explore the notion of tool, how tools are not neutral, but rather bear the marks of the historical process of its creation, literally encoding the biases, dreams, and political realities of its creators, offering affordances for some interactions while making other interactions difficult or impossible to perform or even conceive. While the ability to program does not bring absolute freedom (you can never step outside of culture, and of course programming languages are themselves tools embedded in culture), it does open up a region of free play, allowing the artist to climb up and down the dizzying tower of abstraction and encode her own biases, dreams and political realities.
- *Create a literary machine*. Literary machines are potential literature, procedurally producing textual traces in response to interaction. Examples of literary machines include interactive fiction, nodal hypertexts, interactive poetry (often with

animated typography), and chatterbots. For this project, the literary machine must include algorithmic elements, such as animated typography, generated text, or conditional responses as a function of the previous interaction trace. It must respond to external inputs (e.g. user interaction). With this project I want students to think about language and computation, including strategies for language generation, manipulation, and display (typographic manipulation).

- *Create an applet that dynamically does something to one or more web pages (e.g. collage, systematic distortion, re-layout, ironic superposition, etc.).* Hypertext was conceived as a computer-aided form of reading and writing whose structure matches that of the human mind (a tangled web of association), thus enabling humans to make sense of the exponential growth of knowledge experienced in the 20th century. The World-Wide Web, while a rather anemic implementation of hypertext, makes up for these deficiencies by providing us a sneak preview of what it might be like to have a truly global repository of knowledge. But making sense of the world is not just a matter of structure, but of process, of the *dynamic* construction of meaning. With this project I want students to move away from a static, structure-based view of the web, to a process-based view. This project continues a concern with language (and juxtaposition of language and image) from the literary machine, but moves it into the web, to include link structure and dynamic parsing of web pages.

- *Build a collection of Braitenberg vehicles that respond to each other, to objects in the environment, and to player interaction.* Braitenberg vehicles (Braitenberg 1986) are a simple autonomous agent model in which sensors are directly connected to wheel-like actuators. Vehicles with different "personalities" can be built simply by manipulating the wiring of the vehicle, for example crossing sensor outputs and wheels (e.g. left sensor output goes to the right wheel), inverting sensor output and so forth. This project allows students to explore an artificial intelligence model of behavior, and how the complex, generative responses of AI systems can be harnessed for expressive purposes. As has already been discussed above, AI approaches are used extensively in games to build, for example, tactical and strategic opponents, non-player characters, and player modeling systems.

- *Create a simple game.* As a capstone project, students are encouraged to bring elements from previous projects into this one. Since students only have two weeks per project, the game should be a simple or "casual" game that is asset light (this also forces a focus on procedurality), easy to learn, but with gameplay depth that is revealed as you spend more time with the game.

Readings in the *New Media Reader* are coordinated with each of the projects. For example, while working on Project 2, the "create your own drawing tool" project, we read:

- *Man-Computer Symbiosis*, J.C.R. Licklider
- *Sketchpad: A Man-Machine Graphical Communication Systems*, Ivan Sutherland
- *Direct Manipulation: A Step Beyond Programming Languages,* Ben Schneiderman

- *A Cyborg Manifesto*, Donna Haraway
- *The GNU Manifesto*, Richard Stallman
- *Happenings in the New York Scene*, Allan Kaprow
- *The Cut-Up Method of Brion Gysin*, William S. Burroughs
- *Six Selections by the Oulipo*, Raymond Queneau, Jean Lescure, Claude Berge, Paul Forunel, Italo Calvino

Project 2 explores the idea of tool, how tools create new ways of relating to machines, how tools contain the dreams and biases of the designer and thus constrain as well as enable, and what it means to make your own custom tools. Project 2 also explores the tension between tools that support human creativity and tools that have their own autonomy. Thus, in *Man-Computer Symbiosis*, we look at the vision of the computer as an "AI research buddy" that collaborates with the user. In *Sketchpad* and *Direct Manipulation* we look at the vision of the graphical user interface as a transparent tool that leverages our ability to manipulate objects in the physical world. In the *Cyborg Manifesto* we look at how any tool is really composed of both technology and social practices surrounding technology, and how our subjectivities are defined by our tools. With *The GNU Manifesto* we explore what it means to truly own your tools, to be able to modify them in any way you want, and how procedural literacy is necessary to have this kind of control over your tools. Finally, in *Happenings*, *The Cut-Up Method* and *Six Selections*, we look at algorithmic generation via processes of recombination and constraint, preparing the way for both projects 2 and 3, which both explore the concept of autonomous generation.

One of the challenges with helping students to engage the readings is balancing the conceptual material with programming details. The urgency and anxiety some students feel with learning to program can make it difficult for them to focus on the readings. If significant mental energy is going into understanding class inheritance and event-based processing, it can be difficult for students to think about the historical origins of the graphical user interface and its relationship to cybernetic discourse while connecting this back to the nitty-gritty details of writing code.

Another goal of the readings is to introduce students to the styles of writing found in technical, critical theoretic and art discourse. Since being procedurally literate includes being able to unpack social and cultural assumptions of code (deep readings of code), to understand the relationship between creative expression and code, as well as being able to program, students must comfortable participating in a variety of discourses. For most students, one and sometimes two of the three genres of writing is new to them. Thus, just as there is variability in programming background, there is variability in people's abilities to read and discuss various genres of writing. Facilitating fruitful class discussions requires being able to situate each of the readings, providing the background necessary to allow the whole class to engage the readings and relate the readings to programming practice.

I used raw Java the first time I taught the course. One of the goals of the course is to introduce artists and designers to computation itself as a medium. Thus I don't want to

teach the course within a scripting tool or programming environment that has been designed specifically for artists; such tools inevitably make a certain class of projects easy (e.g. web animation) at the expense of making other projects hard or impossible. Java, as a widely-used general purpose programming language for both stand-alone and web-based applications, with huge libraries of pre-written components freely available, allows students to experience the generality of programming and provides them with concrete skills they can use after the class. However, the overhead of using the standard Java classes for input/output, particularly the overhead of using the Swing library for graphical windows, turned out to be problematic. Before a student can open their first window and display something in it, they must become fairly fluent in object-oriented programming, as well as learn a rather complex class library (Swing) that even more experienced programmers sometimes have difficulty using. While students do learn object-oriented programming in this class, the complexities of Swing forced sophisticated object-oriented concepts too early in the course, and resulted in students only being able to complete four out of the six projects and a reduced number of readings.

The next time I taught the course I used Processing (www.processing.org), a programming environment and API built on top of Java. Processing provides a graphical window and drawing commands as built-in primitives, as well as a scripting-like programming environment that allows new programmers to quickly create straight-line code without classes. There is an active art community of Processing enthusiasts who share code, providing students with a community of practice within which to learn art-centric programming. And, since Processing is built on top of Java, it's possible to import classes from the standard Java API and to write arbitrary Java programs that make as much or as little use of the Processing-provided primitives as desired. With Processing providing scaffolding, particularly early in the course, students were able to successfully complete all six projects, and to engage the full set of readings I wanted to cover.

A down-side of using Processing is that it was definitely developed from a visual arts background (it was developed by grad students in the Aesthetics and Computation group at the MIT Media Lab); it is designed to support procedural graphics, but not other forms of procedurality such as text manipulation/generation, web parsing and recombination, and AI and Artificial Life models of behavior. For projects that required such capabilities, I gave students starter code to work from. Since Processing is built on Java, in future iterations of this course it would be possible to provide such capabilities as library extensions to Processing, though it's still useful to have students look at the source code so as to understand how such capabilities can be added.

Conclusion
Procedural literacy is necessary for new media theorists and practitioners. Without a deep understanding of the relationship between what lies on and beneath the screen, scholars are unable to deeply read new media work, while practitioners, living in the prison-house of "art friendly" tools, are unable to tap the true representational power of computation as a medium. The ideal of procedural literacy as necessary, not just for new media practitioners, but as requirement and right of an educated populace, has been with us for over 40 years. Yet the two culture divide persists, with artists and humanists often

believing that programming is a narrow technical specialty removed from culture, and computer scientists and engineers often happy to pursue just such an unexamined and narrow focus. Computer games are a prefect vehicle around which to build a procedural literacy curriculum that spans the two-culture divide. Poised to become the primary interactive art form of the 21st century, games appeal across engineering, art and the humanities, uniting technical and expressive concerns. Games define a procedural world, foregrounding the relationship between simulation and audience reception. My graduate level course, Computation as An Expressive Medium, is organized around the construction of game-like artifacts, combining technical skills with sophisticated historical and theoretical understandings of computational artifacts. Students from technical, arts, and humanities backgrounds have successfully engaged the course, adding empirical support to the idea that games (and game-like artifacts) can serve as a successful organizing framework for procedural literacy courses, at least for students in new media programs.

To achieve a broader and more profound procedural literacy will require developing an extended curriculum that starts in elementary school and continues through college. Encountering procedurality for the first time in a graduate level course is like a first language course in which students are asked to learn the grammar and vocabulary, read and comment on literature, and write short stories, all in one semester; as my students I'm sure would agree, this is a challenging proposition. In the same way that people engage language throughout their entire educational trajectory, so to should students engage procedurality. Only then will computation truly become an expression of culture on par with language, image, sound, and physical objects, adding process-based representation to the human conversation.

Bibliography

Braitenberg, V. 1986. *Vehicles: Experiments in Synthetic Psychology*, MIT Press, Cambridge, Mass.

Crawford, C. 1987. Process Intensity, *The Journal of Computer Game Development*, Vol. 1, No. 5. Online at: http://www.erasmatazz.com/library/JCGD_Volume_1/Process_Intensity.html.

Flanagan, M. and Perlin, K. *Rapunzel*, http://www.maryflanagan.com/rapunsel/about.htm.

Fry, B., and Reas, C. *Processing*, http://processing.org/.

Greenberger, M. (Ed.) 1962. *Management and the Computer of the Future*, The MIT Press, Cambridge, Massachusetts.

Guzdial, M. 2003. A media computation course for non-majors, In *ITiCSE Proceedings* p. 104-108. ACM: New York.

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*, Vol. 35(2), pp. 5-8.

Kay, A. and Goldberg, A. 1977. Personal dynamic media, *Computer*, 10: 31-41.

Maeda, J. 1999. *Design by Numbers,* MIT Press, Cambridge, Mass.

Mateas, M. 2003. Expressive AI: Games and Artificial Intelligence. In *Proceedings of Level Up - Digital Games Research Conference*.

Nelson, T. 1987. *Computer Lib/Dream Machines.* Redmond, WA: Tempus Books, a division of Microsoft Press. Original editions 1974.

Papert, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, Inc., New York, New York.

Wardrip-Fruin, N., and Montfort, N. (Eds.) 2003. *The New Media Reader*, MIT Press, Cambridge, MA.