Programming Perspectives in Texts and Technology: Teaching Computer Programming to Graduate Students in the Humanities

Rudy McDaniel
University of Central Florida

Abstract. This article discusses how one doctoral program implemented a course designed to teach computer programming to graduate students in the humanities. The article first discusses recent literature that makes connections between programming and the humanities and argues that program administrators may wish to support the learning of programming for a number of reasons, not the least of which is additional scholarly and creative design opportunities for their students. The latter half of the article then discusses the interdisciplinary Texts and Technology PhD program at the University of Central Florida. It broadly describes the program's history and governance structure and then details how a programming course was recently integrated into the core curriculum. Course goals, assignments, and programming tutorials are discussed and a few examples of major student projects, such as a Dada-inspired sound poem and an experimental web site that converts short stories into abstract art, are presented and made available through hyperlinks. The paper concludes by discussing implications for the field and providing a list of online tutorials and resources that are available for administrators, faculty members, and students who wish to learn more about programming using any number of popular computer languages.

Keywords. curriculum reform, computer programming, coding, project-based assessment, experimental pedagogy, digital literacy, praxis

Given the centrality of programmed tools and technologies in the everyday lives of students and faculty, a number of pertinent questions are raised within the domain of program administration. For example, we may contemplate how our programs position the role of software in relation to students' research and creative activities. At what point do students in academic programs become active producers, rather than just consumers, of software? How can software move beyond organizing one's professional life and take a more central role in data collection, research, and writing? Further, what role might students play in designing and developing software tools to build more capable systems?

This paper discusses one course from a humanities doctoral program in Texts and Technology (T&T) at the University of Central Florida that considers these types of questions. The T&T program has developed a required programming course within its core curriculum. The course challenges students to learn enough about programming to develop their own scholarly projects using Internet scripting languages. This essay first discusses the idea of programming as a form of digital crafting, creating interesting possibilities for digital writing practices. It then provides an overview of the T&T program, focusing specifically on the design- and development-oriented course devoted to teaching computer programming as an aspect of digital literacy. The paper concludes with an annotated bibliography offering several online resources to students, faculty, and program administrators who may be interested in learning more about programming.

Programming as Digital Crafting

Under various descriptions such as "critical code studies," (Marino, 2006), "procedural literacy" (Bogost, 2007), "digital literacy" (Spilka, 2010), "source literacy" (Stolley, 2012), and "computational literacy" (Vee, 2013), a number of scholars make thoughtful connections between computer programming and the intellectual territory of the humanities. These researchers urge students and faculty to become more familiar with the capabilities and limitations of programming. They note knowledge about programming provides opportunities for theory building as well as the construction of unique tools and digital projects. Tools and digital projects developed by humanities students can then be used for specialized research and creative activities within their fields.

These applied activities have theoretical implications. For example, Karl Stolley (2012) champions what he terms "source literacy" as a core philosophy for the field of computers and writing. Source literacy employs digital craftsmanship as a central idea. Source literacy, in Stolley's view, means understanding the craft of digital writing in a more comprehensive fashion, through both technical knowledge of programming and a rhetorical understanding of the practices and products generated by programming. Source literacy encourages not only critiquing programmed texts, but also understanding their computational underpinnings in an informed way.

For those new to programming, we are at a moment of unprecedented opportunity for better informing ourselves about software design and development. An abundance of open-source and popular web-based scripting or programming languages—such as PHP, Python, and Ruby—are widely available. These languages are thoroughly documented with tutorials, videos, and wikis. Because major commercial sites run using these technologies, it is easy to find examples to show to students. Additionally, the distribution of students' finished projects is easy to manage. Anyone with a web browser can easily publish materials to the Web and promote work through social media, blogs, or listserv posts. The number of tutorials and reference materials continue to increase and become more accessible for non-technical audiences.

Why Graduate Program Administrators Should Care about Programming

Despite the accessibility of training materials and the enthusiasm students often bring to the idea of learning new technical skills, a note of caution is in order. For example, as program administrators, we often need to temper excitement for new curricular reform initiatives with the realities of our resources and the political climate in which our programs reside. After all, it is unusual to be able to affect sweeping changes with the modest financial support and personnel resources provided to many graduate programs in the humanities. So, with these caveats in mind, why is integrating a programming course into a graduate curriculum something a humanities program administrator might want to consider?

First, technical knowledge is becoming more important for the large number of humanities students with advanced degress who enter alt-academic careers. These careers are represented in job titles such as technical communicators, content specialists, editors, multimedia designers, educational content designers, web developers, or information architects. Students entering industry often interact and communicate with diverse professionals using project management methodologies derived from the world of software engineering or product design. Knowledge of coding and production strategies such as iterative design, agile development, extreme programming, scrum, and the software development life cycle (SDLC) (Dicks, 2010) are important for students to understand. Even students who do not enter alt-academic careers in industry or research find value in "speaking the language" of coding and understanding the ways in which problem solving can be approached from technical perspectives.

This intellectual common ground is handy when communicating with industry partners for collaborative purposes. For graduate students who will be working in research and development (R&D) environments or taking on positions of academic leadership, these problem-solving abilities in new textual and technological environments are even more critical.

Second, in the case of students' publication and creative activities, there will always be a point in which a preexisting software application leaves something to be desired for a particular scholarly purpose. For example, a graduate student investigating whether or not a program such as Twitter can be used to broadcast emergency messages to employees within an organization might find it to be suitable at first, but would later be frustrated by the privacy settings, character limits, distribution model, or other aspects of the software. Similarly, a newly hired MA graduate working in industry might attempt to install and deploy a commercial content management system such as Drupal or WordPress only to discover certain features do not operate as anticipated. Or perhaps, even worse, the features are absent altogether. The problem in these examples is that the software being used for these purposes was designed by individuals who all exist outside the students' spheres of influence.

Third, it is important to introduce new ideas about design and usability into the development of computer software. Bringing humanities students into design and development roles catalyzes new ways of thinking about user interface design. It creates opportunities to consider new possibilities for data architecture and organization and afford new opportunities for creative experimentation. Technical and professional communication students, in particular, have much value to add in creative experimentation. Technical communication is a profession in which technical communicators need to understand audiences, the tools they use, and the capabilities and limitations of those tools (Swarts, 2013). Further, many technical communicators already self-identify as designers (Pringle & Williams, 2005). In addition to documenting processes and procedures of existing software tools, technical communicators can also use their insights regarding audiences and their informational needs to play a more active role in design and development.

In sum, a humanistic programmer is a programmer who is also an expert in practices such as audience analysis, critical analysis, historical evaluation, communicating, editing, and writing. As program administrators, we create the environments in which this interdisciplinary scholarship becomes possible. Humanities students who know how to program can serve as user advocates throughout the design and development process, not just at the end. And while user-centered design practices, hybrid teams, and participatory design techniques can all act as surrogates for the direct participation in the coding process by a graduate student in the humanities, it is the code that determines the interactions and affordances of the software technologies manipulated by end users. Although it is true that students can create new projects using existing tools rather than programming them from scratch, it is also true that these extant tools will shape the direction of the new projects and limit the possibilities of creative expression.

In the second half of this essay, I discuss efforts to integrate a programming course into the core curriculum of a doctoral program in the humanities. After a discussion of the program's history, governance structure, and administration, I delve more deeply into the specific course curriculum and assignments and provide examples of student programming projects. The article concludes with a postscript containing ten online resources for program administrators, students, and faculty who might wish to pursue programming-related activities within their own academic programs.

The Texts and Technology (T&T) Program

The Texts and Technology (T&T) PhD program is an interdisciplinary humanities doctoral program located in Orlando, Florida. The program was proposed to the Board of Trustees as a new doctoral program for the UCF English Department in 2000 and pitched as a unique PhD program that melded the latest digital technologies with the field of textual studies. The program accepted its first cohort of graduate students in the fall semester of 2001. Students enter the program with master's degrees obtained from a variety of different academic fields including technical communication, professional writing, rhetoric and composition, creative writing, literary studies, history, anthropology, criminal justice, and education.

In 2011, as a result of a program evaluation, the T&T program relocated from the English Department to the College of Arts and Humanities and broadened its core faculty to include academic expertise and participation from additional disciplines outside of English and the newly formed Department of Writing and Rhetoric. The years from 2011 to 2015 further broadened disciplinary representation by including new faculty members from departments such as Digital Media, Philosophy, and History. This diversifying of expertise is supported by recommendations in the literature for more broadly interdisciplinary communication programs (Ecker & Staples, 1997). To maintain curricular focus as areas of faculty expertise broadened, the curriculum subcommittee worked with the program director in 2012 to develop areas of specialization within the program. In the short term, these areas of specialization guide students toward the selection of appropriate elective courses and suggest potential committee members, who are working in those fields, for their dissertation and exam work. On a long-term basis, the areas help students professionally identify with existing fields and target and market themselves to particular job descriptions when entering the academic or industrial job markets.

The T&T program is managed by a program director who reports directly to the dean of the College of Arts and Humanities. An assistant director reports to the director and handles a majority of the day-to-day student issues such as course registration, admissions processing, and routine correspondence with the Graduate College. The director is in charge of strategic initiatives for the program, handles the negotiation of teaching and assistantship placements with departmental chairs, and schedules monthly meetings of the T&T faculty. These faculty meetings occur three to four times each academic term.

In terms of program governance, T&T operates according to a set of program bylaws[1] developed by a faculty subcommittee in cooperation with the program director. These bylaws specify the mechanisms for faculty membership and dictate the types and responsibilities of faculty subcommittees. For example, when proposing curricular revisions, the T&T curriculum subcommittee develops the necessary revisions or course materials and then brings a proposal to the full T&T faculty for a vote.

Building a Programming Course into the Curriculum

Design and Development of T&T[2] is a course in the core curriculum of the T&T program. It is a hybrid, mixed-mode course, in which face-to-face lectures are accompanied by online content and exercises. Students take the course in their second year, following other core courses such as research methods, theory, and history. The idea behind the course is simple: provide doctoral students with enough programming familiarity to complete their own digital scholarly projects. Although the students can accomplish such a goal using existing tools, this further allows them to build projects without *requiring them* to rely solely on these existing tools and data sets. In regards to those students who do have project ideas best served by existing tools, the course provides them with a deeper level of procedural knowledge. This knowledge enables them to interact with existing tools through application programming interfaces (APIs), custom data queries,

[1] See ‹http://tandt.cah.ucf.edu/files/BYLAWSFinal10_30_13.pdf›.
[2] See ‹ http://rudymcdaniel.com/pubs/pp/files/aa_syllabus.pdf ›.

and other advanced mechanisms for interfacing with both commercial and open source software and databases.

The Design and Development course requires students to read six core texts as well as a small number of supplemental online readings from various sources to augment specific topics in the class. For example, roughly a third of the course is spent talking about scholarship in the digital humanities. Readings include Stephan Ramsay (2011), the *Switching Codes* edited collection (Bartsherer & Coover, 2011), an article from Wendy Chun and Lisa Rhody (2014) recounting a roundtable MLA discussion about the "Dark Side of the Digital Humanities," *Alien Phenomenology* from Ian Bogost (2012), *How We Think* by N. Katherine Hayles (2012), and *The Design of Future Things* by Donald Norman (2007). These texts examine aspects of design and development in various theoretical and practical ways.

Each of the chosen texts relates to programming in a different way, and sometimes those relationships are not immediately obvious. For example, Bogost's *Alien Phenomenology* was not included due to any particular excellence in explaining the process of object-oriented programming, but rather because it takes on the difficult task of considering objects outside the usual paradigm of human-centered analysis, the correlationalist model. Such an unusual, or alien, perspective is helpful in directing students toward a mode of carefully considering the forms and functions of everyday objects and, in turn, helps them to conceptualize their own object design and development during that portion of the course. This text serves as a bridge-building discussion opportunity for the students to discuss similarities and differences between object-oriented theories and object-oriented programming practices. It also allows them to consider objects in more precise detail than their everyday use might normally require. For example, one discussion posting exercise, "Objects 15 Ways," required students to write fifteen different definitions for everyday objects. They accomplished this using various definitional approaches (e.g., instrumental, operational, socio-cultural, technical, physical, and functional). This followed a practice in the text in which Bogost defined everyday objects in a similar fashion. The course exercise helped students recognize the difference between communication practices in natural language, in which context and jargon can be loose and ambiguous, versus programming language, in which details must be exhaustively and precisely listed.

Course Goals

A primary course goal is for students to better develop their understandings of computational media. Students in the Design and Development course are taught to understand the procedural affordances of digital software. They are taught how to solve problems using available functionality and resources. This problem solving might allow students to identify the appropriate data structures to hold different types of information or understand the appropriate methods for packaging similar units of code into functions or objects for re-use and efficiency. Or, it might mean using divide-and-conquer techniques to decompose a larger problem into smaller and more manageable units. Combined with their knowledge of rhetoric, history, writing, critical thinking, participatory design, user-centered design, and communication, this knowledge makes them versatile for a number of academic and industrial tasks. This type of functional digital literacy, which some scholars have termed *procedural literacy* (Bogost, 2007), involves students' recognition of the unique capabilities of texts (broadly considered) as they exist in digital form. Students begin the course with a digital pre-test[3] that gauges their current knowledge about programming.

Another course goal is to change the way in which students think about programming. For instance, despite the rigidity with which programming syntax is often crafted, programming is very much a creative activity. Even seemingly straightforward programming problems can usually be solved in a variety of ways. Selecting a particular approach is a creative act in and of itself. Creativity also requires developers to work around the limitations imposed by language and resources, a situation familiar to many students through their studies about writing for different audiences and different rhetorical contexts. Matthew G. Kirschenbaum (2009) describes programming as creative in the sense that it teaches us how to solve constrained problems. This process often requires us to change our perspectives of the world. He notes the importance of modeling creativity and reminds us models can be constructed in a variety of ways, so even building something as simple as a retail inventory can be pursued in a number of different ways. It is also true, though, that modeling becomes more useful when it combines with one's knowledge of the world and how things happen inside that world. Building an accurate and functional model means one must understand not only the particular components to be modeled, but also the relationship of this model with the environment in which it is to be integrated.

From Theory to Practice: Project Assignments

As this course was conceptualized as a means of delivering technical skills while also situating the tools critically within relevant theoretical models,

[3] See ‹http://rudymcdaniel.com/pubs/pp/files/ab_pretest.pdf›.

the projects require students to synthesize theory and practice. The first digital project assignment,[4] for example, incorporates theoretical ideas from Stephen Ramsey (2011), Marcel O'Gorman (2006), and other scholars exploring the intersections between computation and the humanities. Ramsey's (2011) work tackles the complex task of exploring the differences and similarities between humanistic scholarship and the quantitative work done by computing machines. Central to his writing is an analysis of computing from the perspective of literary studies and the humanities; he notes the difficulty in relating humanities discourse with computer-based processes, pointing out that humanistic methods and computational methods are frustratingly different.

A significant challenge, then, in moving from theory to practice is explaining to students how to temporarily suspend those important intellectual strategies or behaviors that we have worked so hard to teach them during their first year of study. This shift from theory to practice complicates prior tactics such as challenging binary dichotomies, questioning the social and cultural constructions of language and identity, and approaching the understanding of textuality from multiple perspectives. When learning how to communicate with a computer, students must learn to be binary communicators: direct, unequivocally precise, and straightforward. Semantically, they must remember to write down every single step in a series of rules; omitting even an obvious step is an insurmountable obstacle for most computer programs. In terms of syntax, even the omission of a single character, such as a semicolon, can lead to hours of frustrating troubleshooting. After working through only three grueling weeks of learning how to program, however, student programmers are able to produce an array of interesting projects[5] including text adventure games, love sonnet generators, interactive fiction experiments, and specialized tools for research and information retrieval.

Learning to Program, One Week at a Time

The primary instructional mechanisms for teaching the students how to program are found in a series of required Codecademy tutorials linked to each week in the course. Lectures and face-to-face discussions focus on the readings for each week and the students use their additional online work time to complete Codecademy tutorials. Student questions are answered using our course content management system's (CMS's) discussion

[4] See ‹http://rudymcdaniel.com/pubs/pp/files/ac_project1.pdf›.
[5] See ‹http://www.rudymcdaniel.com/pubs/pp/files/ad_p1_examples.pdf›.

boards. Additional instructional videos are also created and uploaded to the CMS by the instructor.

Codecademy was launched in 2011 and rose to prominence with its "Hour of Code" iPhone application. The application contained bite size lessons teaching users how to program in popular languages such as JavaScript, Python, and Ruby (Dredge, 2013). A key strategy employed by both the Hour of Code initiative and Codecademy is to release code in small chunks delivered using a gentle pace, so as not to overwhelm novice users. The accompanying web site[6] uses an interactive prompt that shows the results of coding in real time. This strategy allows learners to immediately see feedback and receive coaching when they make a mistake. The Codecademy interface uses three primary panels. The leftmost pane is an instructional panel with background information about the topic. It provides instructions for appropriate syntax and usage for whatever coding feature or concept is being presented. The middle panel displays a code file that a student can edit. As the student edits the file, the results are displayed in a panel to the right. When errors appear, the system provides additional feedback to help learners figure out what went wrong.

By combining instructional information, editing capabilities, and output displays, the Codecademy interface makes the once highly distributed task of learning how to program a little bit easier. Without this type of system, a student would need to edit and save files on a local computer, upload these files to a remote server, then access the remote files using a browser. When something did not work as anticipated, the student would need to consult various help files to determine why the files were not working as imagined. Combining everything into a single instructional application makes this process less onerous for beginners.

In terms of choosing a specific programming technology to learn, beginning programmers are encouraged to choose PHP. This scripting language is heavily documented with tutorials, books, and enthusiast web sites. It has additionally been used to develop tutorials specifically for technical communication tasks, such as building single-sourcing systems for documentation (Applen & McDaniel, 2009). PHP is also fairly forgiving to new programmers. For example, rather than requiring a developer to specify an initial "type" for every variable used in the program, the language only requires "strong typing" for certain types of complex variables, such as objects and arrays. More advanced programmers can choose to use a language they have not yet learned, such as Python or Ruby.

[6] See ‹www.Codecademy.com›.

Aside from its helpful chunking of programming lessons into digestible modules, Codecademy also excels in providing feedback and encouragement to students as they make incremental progress in the course. A bar chart shows progress in specific skills and topics as students progress through the course modules. In addition, course badges, similar to those used in modern console video games, are "unlocked" as students complete modules within the site. Students can also return to the web site at any time to view their progress, as represented through their skills interface and badge collection, on the Codecademy web site.

Major Projects

Procedural literacy encompasses knowledge of digital rules, models, and algorithms, elements that may eventually become more central to a modern humanities education (Kirschenbaum, 2009). Creating a fuller understanding of procedural information processing necessitates a type of digital writing that relies upon the programmatic affordances enabled by digital technologies. In other words, students need to create projects that do more than just display printed text on the screen. However, it is also important that familiar conceptual anchors are provided to topics with which the students are already comfortable. This makes the creation of projects focused on familiar material an important component of the course. As such, the following projects are assigned:

- A rough project[7] based on the design and development of a "potential literature machine."
- A polished and iteratively designed improvement of the first project.
- A final project[8] to build a "T&T memex machine," or a database combining information archiving and retrieval with theoretical ideas from course readings or outside sources.

Expecting students to build fully polished and perfectly functioning projects in this small amount of time, especially as beginners, is unreasonable. Students are instead encouraged to craft with a goal in mind. They are counseled to not become too discouraged if everything does not come together in the way they had initially hoped and imagined. By the end of the most recent semester, however, students learned the skills to build a rough yet functioning prototype of a database-driven web application using MySQL. They were able to design and develop their own preliminary

[7] See ‹http://www.rudymcdaniel.com/pubs/pp/files/ac_project1.pdf›.

[8] See ‹http://www.rudymcdaniel.com/pubs/pp/files/ae_final_project.pdf›.

data sets using database tables and then perform basic database operations (e.g., INSERT, UPDATE, and SELECT) with those data using PHP scripts. Better yet, they created database-driven web applications that did not yet exist prior to their efforts, and that suggested new ways of conceptualizing relationships between audiences, ideas, and technologies.

For example, one student designed a final project that enables visitors to create, upload, and peruse Dada-inspired sound poems.[9] Another developed a utility and algorithm for converting short stories into abstract expressionist artworks.[10] The overarching purpose of teaching these students how to program was to provide them with the skills with which to complete these hybrid projects. They were advised to use these skills in a creative fashion to meet their own research goals. Such an effort required knowledge about their own research interests in the arts and humanities as well as knowledge about coding and database construction.

Conclusion

Experiences in designing and teaching this type of course suggest that learning programming is challenging for students, but also very rewarding. The act of programming catalyzes new forms of digital writing and affords diverse, interactive, and creative forms of scholarship. Even so, program administrators and faculty members who wish to integrate a focus in programming into their own classes and curriculum will need to be patient. It is not reasonable to expect that humanities students who have never had any exposure to programming will be proficient programmers after only a single course. It is recognized within the computer science literature, for example, that it takes ten years to transform a novice programmer into an expert (Winslow, 1996). This learning curve is likely even steeper in disciplines outside of computer science.

Even within a landscape constrained by these challenges, this course within the T&T program provides a study of how programs might tackle the issue of programming web-based digital crafting within a humanities curriculum. Educators in other programs who wish to follow suit might adopt other methods entirely, perhaps by focusing on programmable tools or teaching students how to program statistical analysis software. Whatever the method employed, program administrators and educators in the humanities should never adopt the position that programming is necessary to address some deficit within our students that needs to be remedied

[9] See ‹http://www.rudymcdaniel.com/pubs/pp/files/af_final_project_examples.pdf›.
[10] See ‹http://www.rudymcdaniel.com/pubs/pp/files/af_final_project_examples.pdf›.

by teaching them more applied technical knowledge. The idea is in fact the reverse: teaching humanities students how to program adds value to the existing body of interactive and electronic media by introducing more diverse ideas from the humanities into computer software.

Programs wishing to explore computer programming within their own curricula should first survey the landscape to determine if programming is an activity that students wish to pursue and faculty members find worthwhile. Next, educators should determine the resources necessary to integrate programming competencies into their courses. This decision raises questions about faculty training, resources, lab space, and a myriad of other procedural and technical issues. Having these conversations within and between program administration communities is a valuable step in training graduate students in the humanities and making them more versatile in both skill and knowledge. A promising side effect is that these experiments in programming are also adding significant value to existing sources of electronic software and scholarship.

These types of intellectually risky digital crafting exercises deserve further encouragement through curricular strategies in humanities programs. As program administrators, we can help create the academic climates in which such experimentation is rewarded and supported. Included in the Postscript are references and resources that may be useful to program administrators and educators interested in learning more about programming.

Postscript: Online Resources for Learning Programming

Following is an annotated list of ten online resources, helpful to students and faculty members for learning to program or for brushing up on the newer features of programming languages. Some resources are free and some require modest subscription fees.

1. Code.org (‹www.code.org›)

   Code.org is the online portal for a non-profit organization that was launched in 2013 to expand participation in computer programming and increase the availability of easy-to-use tools and tutorials. The organization specifically caters to women and underrepresented students of color. Each year, an "Hour of Code" event is held collaboratively at schools across the world and there are a number of tutorials and curricular ideas organized on the web site.

2. Codecademy (‹www.codecademy.com/›)

Codecademy is an interactive site that lets users work through a series of online lessons to learn the basics of programming. The site incorporates a wide range of programming languages including HTML, CSS, JavaScript, PHP, Python, and Ruby. All courses are free to use.

3. Code School (‹www.codeschool.com›)

Code School is another site that allows users to work through online lessons in languages including Ruby, JavaScript, HTML/CSS, iOS, and Git. Visitors to the site can browse through specific courses in the paths and there is an "elective" series to allow users to venture off the beaten path. Many courses are free, but monthly subscription rates are also available.

4. Lynda.com (‹www.lynda.com›)

In addition to video tutorials for popular graphical software such as Adobe Photoshop, this site offers video tutorials in a number of different areas that include programming. The videos are created by industry professionals and experts, and the videos range in difficulty from beginning to advanced-level content. A membership is required to access the videos.

5. Safari (‹www.safaribooksonline.com›)

For those who prefer to learn from books, Safari's online library contains both videos and technical texts in electronic book format from major publishers of technical tutorials and programming references. Individual and team subscriptions can be purchased on a monthly or yearly basis.

6. SQLZOO (‹sqlzoo.net›)

This site offers free, interactive tutorials in SQL, or the structured query language, designed for communicating with databases. The site offers a comprehensive approach to learning the language and ranges from tutorials for the beginner to those who already have SQL coding experience.

7. Treehouse (‹www.teamtreehouse.com›)

This site provides a combination of video tutorials and interactive workspaces to help novice programmers learn to code. The tutorials often result in projects, meaning the user learns the code along the way. The site has a free trial period and a "basic" and "pro" option for monthly subscription services.

8. Udacity (‹www.udacity.com›)

This site provides video lectures and interactive activities to help students learn tech skills needed to stand out in the workplace. The site provides what they call "Nanodegrees" ™ that provide industry credentials. Udacity is free to use.

9. Udemy (‹www.udemy.com›)

This site offers a wide variety of video courses in development. While the majority of the courses cost money to use, some of the content is free. The coursework spans numerous coding languages and software. Both beginners and advanced learners find compatible content.

10. W3schools.com (‹www.w3schools.com›)

This site offers tutorials and chapters for several programing languages including HTML, CSS, JavaScript, SQL, PHP, and JQuery. This site is not as fully interactive as some of the ones described previously, but the examples are clear and efficient. The site is free to use.

References

Applen, J. D., & McDaniel, Rudy. (2009). *The rhetorical nature of XML*. New York: Routledge.

Bartscherer, Thomas, & Coover, Roderick. (Eds.). (2011). *Switching codes: Thinking through digital technology in the humanities and the arts*. Chicago: The University of Chicago Press.

Bogost, Ian. (2007). *Persuasive games: The expressive power of videogames*. Cambridge: The MIT Press.

Bogost, Ian. (2012). *Alien phenomenology, or, what it's like to be a thing*. Minneapolis: University of Minnesota Press.

Chun, Wendy H. K., & Rhody, Lisa. M. (2014). Working the digital humanities: Uncovering shadows between the dark and the light. *Differences, 25*(1), 1-25.

Dicks, R. Stanley. (2010). The effects of digital literacy on the nature of technical communication work. In Spilka, Rachel (Ed.), *Digital literacy for technical communication: 21st century theory and practice* (pp. 51-81). New York: Routledge.

Dredge, Stuart. (2013). Codecademy: Hour of Code app teaches programming skills to iPhone owners. *The Guardian*. Retrieved from ‹http://www.theguardian.com/technology/2013/dec/09/codecademy-hour-of-code-iphone-app›

Ecker, Pamela. S., & Staples, Katherine. (1997). Collaborative conflict and the future: Academic-industrial alliances and adaptations. In Selber, Stuart A. (Ed.), *Computers and technical communication: Pedagogical and programmatic perspectives* (Vol. 3, pp. 375-387). Santa Barbara, CA: Greenwood Publishing Group.

Hayles, N. Katherine. (2012). *How we think: Digital media and contemporary technogenesis*. Chicago: University of Chicago Press.

Jenkins, Henry. (2006). *Convergence culture: Where old and new media collide*. New York: New York University Press.

Kirschenbaum, Matthew G. (2009). Hello worlds: Why humanities students should learn to program. *Chronicle of Higher Education, 50*, B10-B12. Retrieved from ‹http://chronicle.com/article/Hello-Worlds/5476›

Marino, Mark. (2006, December 4). Critical code studies. Electronic Book Review. Retrieved from ‹http://www.electronicbookreview.com/thread/electropoetics/codology›

Norman, Don A. (2007). *The design of future things*. Philadelphia: Basic Books.

O'Gorman, Marcel. (2006). *E-crit: Digital media, critical theory and the humanities*. Toronto: University of Toronto Press.

Pringle, Kathy, & Williams, Sean. (2005). The future is the past: has technical communication arrived as a profession? *Technical Communication, 52*(3), 361-370.

Ramsay, Stephen. (2011). *Reading machines: Toward an algorithmic criticism*. Urbana: University of Illinois Press.

Spilka, Rachel. (2010). *Digital literacy for technical communication: 21st century theory and practice*. New York: Routledge.

Stolley, Karl. (2012, October 10). Source Literacy: A Vision of Craft. *Enculturation*. Retrieved from ‹http://www.enculturation.net/node/5271›

Swarts, Jason. (2012). How can work tools shape and organize technical communication? In Johnson-Eilola, Johndan, & Selber, Stuart A. (Eds.), *Solving problems in technical communication* (pp. 146-164). Chicago: University of Chicago Press.

Vee, Annette. (2013). Understanding computer programming as a literacy. *Literacy in Composition Studies, 1*(2), 42-64.

Winslow, Leon E. (1996). Programming pedagogy—a psychological overview. ACM SIGCSE Bulletin, 28(3), 17-22.

Author information

Rudy McDaniel is an associate professor of digital media for the School of Visual Arts and Design at the University of Central Florida. He is a graduate of the Texts and Technology doctoral program and is currently its director. In addition, he serves as assistant dean of research and technology for the College of Arts and Humanities. Rudy's research currently focuses on game-based design, narrative, and organizational knowledge manage-

ment. He is co-author of *The Rhetorical Nature of XML* (Routledge, 2009) and has published in journals including *Technical Communication*, *Communication Design Quarterly*, the *British Journal of Educational Technology*, *Presence*, *Educational Technology & Society*, and *Information Systems Management*. He is also technical editor and Co-PI for the Charles Brockden Brown Electronic Archive and Scholarly Edition, funded by the National Endowment for the Humanities. Although his primary academic field is in digital media, he considers technical communication a second home and regularly attends conferences within the technical and professional communication community.