Revitalizing Computer Science Education Through the Science of Digital Media: A Report of National Science Foundation Grant CCR 0722261
Detailed Annual Report, April 30, 2010

Jennifer Burg, Principal Investigator
Professor of Computer Science
Wake Forest University
Winston-Salem, NC 27109

Conrad Gleber, Co-Principal Investigator
Director of Digital Arts and Multimedia Design
La Salle University
Philadelphia, PA 19141

Rationale of the Grant Project
This report summarizes the talks, discussions, and findings from a series of seven National Science Foundation CPATH grant-supported workshops centered on the theme of “Revitalizing Computer Science Education Through the Science of Digital Media.” Each workshop focused on a subfield of digital media and its relationship to computer science (Table 1).

<table>
<thead>
<tr>
<th>University</th>
<th>Local Host</th>
<th>Date</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Salle University, Philadelphia PA</td>
<td>Conrad Gleber, Director, Digital Arts, Multimedia Design</td>
<td>May 29-30, 2008</td>
<td>Algorithms, scripting, and programming for visual art</td>
</tr>
<tr>
<td>Wake Forest University, Winston-Salem, NC</td>
<td>Jennifer Burg, Professor of Computer Science</td>
<td>July 28-29, 2008</td>
<td>Digital sound and music</td>
</tr>
<tr>
<td>Florida State University, Tallahassee, FL</td>
<td>Ken Baldauf, Director, Program in Interdisciplinary Computing</td>
<td>April 2-3, 2009</td>
<td>Visualization technologies and applications</td>
</tr>
<tr>
<td>Columbia College, Chicago, IL</td>
<td>Michael Niederman, Chair, Television Department</td>
<td>Sept. 11-12, 2009</td>
<td>Relationship between television and digital media curriculum</td>
</tr>
<tr>
<td>University of California, Santa Cruz, CA</td>
<td>Michael Mateas, Associate Professor of Computer Science</td>
<td>Nov. 19-20, 2009</td>
<td>Media computation and game programming</td>
</tr>
<tr>
<td>East Tennessee State, Johnson City, TN</td>
<td>Cher Cornett, Director, Digital Media Center</td>
<td>Sept. 16-17, 2010</td>
<td>Bringing computer science and art students together in computer game development</td>
</tr>
<tr>
<td>Miami University, Oxford, OH</td>
<td>Gerald Gannod, Associate Professor of Computer Science &amp; Systems Analysis</td>
<td>Oct. 14-15, 2010</td>
<td>Mobile media</td>
</tr>
</tbody>
</table>

Table 1 Workshops Funded by NSF CPATH Grant CCR 0722261
Strictly speaking, “digital media” is a plural term referring to means of communication or productions that are, at their base, represented in the language of computers – i.e., binary code. This includes digital images; sound and music; telephone, internet, and web communications; and the programming languages that tie these together (Burg 2008a). As an academic area, “digital media” can be taken as a singular term referring to the study and production of all things digital. The ubiquity of digital media places digital media studies in a central location, bridging an array of fields from art to science, as depicted in Figure 1. The contention of this grant is that the study of digitally-represented data, information, communication, and art is at the core of computer science. To keep our discipline relevant and to better serve today’s students, we need to recognize the centrality of digital media both in computer science education and in our students’ everyday lives, saturated as they are with digital sound, images, and communication. These ideas motivate the grant project.

The Workshop Series

The purpose the seven-workshop series was to rethink traditional computer science education with digital media more centrally located (Table 1). It was intended that each workshop would bring in the perspective of an academic discipline and a real-world business or industry related to computer science through the common thread of digital media – e.g., digital art, digital sound design and music production, scientific visualization, television production, and computer game development. Representatives from the related academic disciplines were invited to offer their suggestions of topics and approaches that could be introduced into computer science education to create stronger bridges between computer science and neighboring fields of study. Similarly, representatives from business and industry were invited to suggest curricular approaches...
in computer science education that would better prepare computer science graduates for employment in their fields.

The proceedings of the workshops are summarized below, with incrementally developed conclusions and suggestions for revisions of the traditional computer science curriculum.

**The Context for Curricular Reform**

Before recommending curricular revisions, we need to establish our point of departure. As a point of reference, the requirements for a Bachelor of Science computer science major at Wake Forest University as of 2007 are listed in Table 2. These requirements represent what we call a “traditional” computer science degree at a liberal arts university at that time.

<table>
<thead>
<tr>
<th>In the 2007-2008 catalog year, a major in computer science at Wake Forest University required 37 hours in computer science and three courses in mathematics. The required courses in computer science were</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Computer Science (using Java) aka CS1</td>
</tr>
<tr>
<td>Fundamentals of Computer Science (using C++) aka CS2</td>
</tr>
<tr>
<td>Computer Organization</td>
</tr>
<tr>
<td>Computer Systems (introduction to networks and operating systems)</td>
</tr>
<tr>
<td>Data Structures and Algorithms 1</td>
</tr>
<tr>
<td>Data Structures and Algorithms 2</td>
</tr>
<tr>
<td>Programming Languages</td>
</tr>
<tr>
<td>four electives chosen from Databases, Operating Systems, Architecture, Graphics, Artificial Intelligence, Internet Protocols, Digital Media, Software Engineering, Parallel Computation, Numerical Methods, Image Processing, and other special topics courses</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The required courses in mathematics were</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculus 2</td>
</tr>
<tr>
<td>Discrete Mathematics</td>
</tr>
<tr>
<td>Linear Algebra</td>
</tr>
</tbody>
</table>

Table 2 “Traditional” computer science curriculum at a liberal arts university circa 2007

The precipitous drop in computer science majors beginning around the year 2000 – coupled with an increasing sense of the irrelevance of traditional computer science education in a world of ubiquitous digital devices, communication, and entertainment in our everyday lives – has led to widespread reform in computer science programs.

Recent revisions in well-regarded computer science programs reveal a trend toward reducing the core requirements so that students have time to take courses in related disciplines. The areas of specialization are often referred to as “tracks” (or the Georgia Tech trademarked term “threads” (Furst, Isbell, and Guzdial 2007)). In our background research, we observed that the courses that are generally retained in the computer science core include:

- CS1
- CS2
- data structures and algorithms
• computer organization
• introduction to computer networks and operating systems
• discrete mathematics

This observation is illustrated by three examples of recently revised curricula, given in the figure and tables below. The first two are computer science degrees with tracks. The third and fourth examples represent new interdisciplinary digital-media-related programs that have strong connections with computer science departments.

Figure 2 Stanford’s computer science curriculum with tracks (Sahami, Aiken, and Zelenski 2010)

<table>
<thead>
<tr>
<th>Track</th>
<th>CS1</th>
<th>CS2 (OOP)</th>
<th>Data Struct. &amp; Algs.</th>
<th>Proofs (traditionally taught in discrete math)</th>
<th>Computer Org. and Programming</th>
<th>Computer Syst. &amp; Networks</th>
<th>Objects &amp; Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Devices</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Intelligence</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Modeling &amp; Simulation</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Platforms</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Information Internetworks</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Media</td>
<td>(Media Computing instead)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>People</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Theory</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 3 Core requirements in Georgia Tech’s eight threads for computer science majors (Furst, Isbell, and Guzdial 2007) and (Georgia Tech College of Computing)
Introduction to Unix  
Explorations in Computer Science  
CS1  
CS2  
Discrete Structures  
Software Practice I  
Software Practice II  
Computer Organization  
Algorithms  
Computer Systems  
Models of Computation or Scientific Computing  
Capstone project

Table 4 Required computer science courses in Entertainment Arts and Engineering program at University of Utah (Kessler, Van Langeveld, and Altizer 2009)

<table>
<thead>
<tr>
<th>CS1</th>
<th>CS2</th>
<th>Data and File Structures</th>
<th>Computer Organization</th>
<th>Computer Networks</th>
<th>Database Systems</th>
<th>Web Services Technologies</th>
<th>Interactive Digital Media Seminar</th>
<th>One elective from</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Visual Application Development</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Server Applications and Network Security</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Advanced Topics in Database Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Special Topics</td>
</tr>
</tbody>
</table>

Table 5 Required computer science courses in Northwest Missouri State University’s Interactive Digital Media major (Spradling, Strauch, and Warner 2008)

**Workshop 1: Algorithms, Scripting, and Programming for Visual Art**

**Overview**

Workshop 1 explored the relationship between digital art and digital media as computer science. The representatives from business and industry were Marc Fornes and Skylar Tibbets, architects from Theverymany experimental architecture research lab. The representative from a neighboring academic discipline was Ira Greenberg (Greenberg 2007), a computational artist from Miami University of Ohio (who has since moved to direct Southern Methodist University’s Center for Creative Computation).
Fornes and Tibbits were an unusual but not unfortunate choice as representatives from business and industry because of the extremely abstract and cutting-edge nature of their architectural designs. They displayed work which involves the mathematical and algorithmic design of complex 3-dimensional objects that are then fabricated as physical models (Figure 3). While the practical value of these models as physical spaces may not be immediately apparent, we can begin to appreciate the break with convention and unbridled spirit of innovation in Fornes’ description of one specific project, called “Pitch.” This project involves an interdisciplinary team with five components: scenario, design, production; math process; computation; robotics design; and physiological data scanning process and design. It is described as follows:

**Pitch** / "une architecture des humeurs", a research project / exhibition in which the R&Sie(n) architectural practice has worked with a group comprised of a mathematician, programmers, architects and a robotics designer to develop a computational approach based on biological and physiological data scanned from visitors who are put through situations inciting repulsion, stress and pleasure to conceive housing units and urban fragments based on relational protocols. From January 22 through the April 26, 2010, at Le Laboratoire, 4 rue du Bouloi, 75003 Paris (Fornes)

Fornes’ talk demonstrated the need for imagination, innovation, and out-of-the-box thinking of computer science students. Their work requires mathematical modeling of 3-dimensional spaces, programming skills, and the ability to work in an interdisciplinary team. Notably, this type of work challenges students to conceive of applications never before envisioned.

![Figure 3 Contemplating the Void by Theverymany](image)

The other speaker at Workshop 1, Ira Greenberg, described his personal transformation from traditional artist to computational artist, and how this journey helped him develop ways to teach programming to art students. Greenberg is the author of *Processing: Creative Coding and Computational Art*, a textbook that uses the programming language Processing as an introduction to computer programming, particularly for artistic applications. Processing is a simplification of the Java
programming language that allows beginning programmers – particularly art students – to start out at a higher level of abstraction, with built-in functionality for creating dynamic and visually interesting programs without a steep learning curve. Greenberg demonstrated numerous assignments that incrementally develop basic programming skills while offering gratifying visual feedback (Figure 4).

Among the characteristics identified as helpful to computer science students working side-by-side with art students were these: the ability to understand the specialized language and culture of another discipline (in this case, art or graphic design), a respect for another person’s discipline, knowledge, and skill set and what these can bring to a collaboration; the ability to communicate one’s own specialized language, without condescension, to someone from a different discipline; the ability to think creatively with regard to images and sound; and the ability to work with open-ended problems that have no definite right answer.

Greenberg’s talk generated discussion about how computer programming can be taught to students of mixed backgrounds and interests – some who may be immediately attracted to computer science, some who could succeed in computer science if not put off by a dry and unengaging introduction, and some who may be interested in digital art but are not naturally inclined toward computer programming. A number of languages were discussed and later investigated further (Burg 2008b), including Processing, Chuck, Alice, AgentSheets, Scratch, Flash/Actionscript, and Python. Research subsequent to this workshop has shown that many computer science departments are re-evaluating their preference of either C++ or Java as an introductory language. C++ and Java have been the languages commonly used in introductory courses in the past ten years, with hot debate about which should precede or supersede the other (Dewar and Astrachan 2009). With the trend toward reform, the alternative languages mentioned above have become more common in CS1 and interdisciplinary media programming courses over the past 10 years.

Processing is well suited for interdisciplinary courses that emphasize creativity and experimentation in an introduction to computer programming. The openprocessing.org collections website lists many courses that use Processing, including
Interactive Media at UCLA, Introduction to Programming at San Francisco’s Galileo Academy of Science and Technology, Creative Computing at Keele University in the UK, Computational Design at the Amsterdam University of Applied Sciences, and Introduction to Computation Media at the Tisch School of the Arts, NYU (openprocessing.org). Processing has been used in courses that allow experimentation not only with visual art but also with music as a path to learning basic computer skills and constructs (Meyers et al. 2009). An even more targeted environment for music experimentation is ChucK, an audio programming language for creation, composition, performance, and analysis of sound and music, free and supported on Mac, Windows, and Linux operating systems (The Chuck Team) (Misra, Blank, and Kumar 2009).

Alice, developed at Carnegie-Mellon University, is a language that engages students with interactive 3D objects and drag-and-drop program construction that prescribes program structures so that students are not waylaid by syntax errors. Alice’s playful environment appeals especially to young programmers at the elementary, middle school, and high school levels (Fincher et al. 2010). It has also been used in college-level CS0 and CS1 courses (Powers, Ecott, and Hirshfield 2007) (Cooper, Dann, and Pausch). Alice is free and works under Windows or Mac OS.

AgentSheets is another media-rich programming language accessible to young beginning programmers. It supports the creation of games and simulations using a drag-and-drop rule-based system called Conversational Programming (AgentsSheets, Inc. 2010). While the programmer works at a high level of abstraction, he or she can also look at the Java source code produced by the AgentSheets system (Repenning and Ioannidou 2008). AgentsSheets has evolved from a National Science Foundation research project to a commercial product.

Scratch was developed at MIT’s Media Lab as fun programming environment that could be used by beginning programmers to build interactive animations, games, music, and art. Scratch further engages the programmers by allowing them to communicate and share their creation in a social computing network. Programming is accomplished by dragging objects or programming constructs from a palette and stacking them as a way of defining steps. Like Alice and AgentSheets, Scratch takes the emphasis off syntax and focuses on the overall construction of the animation, encouraging playfulness and experimentation with immediate feedback in images and sound (Fincher et al. 2010). While Scratch – like Alice and AgentSheets – was designed for young programmers, it has been used at the high school level and college level (Wolz, Maloney, and Pulmood 2008) (Ruthmann et al. 2010) (Malan and Leitner 2007). Scratch is freely available for Windows, Mac, or Linux (MIT Media Lab).

Flash with ActionScript has the advantage of providing multimedia feedback to beginning programmers at the same time that it offers a full-powered object-oriented language for more advanced programmers. ActionScript’s syntax is very much like Java and C++, facilitating the transition to these languages. Flash with ActionScript is suitable for interdisciplinary programs that combine computer science with multimedia studies, game programming, or entertainment arts (Kessler, Van Langeveld, and Altizer 2009) (Spradling, Strauch, and Warner 2008) (Leutenegger and Edgington 2007). Flash is an Adobe commercial product.

Python is one of the most popular new languages for CS1 (Enbody, Punch, and McCullen 2009) and even CS0.5 (Sloan and Troy 2008) (Freudenthal et al. 2010).
Python is free and runs under Windows, Mac, and Linux. It is an interpreted language, which makes it good for experimentation. It is object-oriented, making a transition to C++ and Java easier. A large selling point of Python is that it strikes a balance between a high and low level of abstraction. Data structures such as strings, lists, associative arrays and sets are available early in the programmer’s experience. The syntax and constructs are readable, allowing the beginning programmer to focus on problem solving, but not overly masking the details. An active support community has created a large number of modules to extend Python, making the language increasingly useful and powerful.

Another line of thinking that arose in this workshop was a reconsideration of computer science’s role as a “service department” in the sense that it could offer courses that are targeted to the needs of other disciplines. To many faculty, the expression “service department” immediately has threatening or demeaning connotations. However, there were those in the workshop who argued that all educators are in the business of service, serving the educational needs of their students. Furthermore, playing a stronger service role can be to the benefit of computer science itself. It’s a fact that few programs outside of computer science require their majors to take a computer science course. Pre-med students typically must take chemistry, mathematics, and biology, but not computer science. The natural sciences in general require courses in other natural science departments and in mathematics, but not in computer science. Yet it’s clear that computer science is increasingly important for a complete education in these other disciplines – not only in the sciences and mathematics, but also in business, economics, art, music, and more. Introductory programming courses that are relevant to and draw assignments from these other disciplines would serve the needs of the students and, as a pleasant side benefit, might attract some of these students to further study in computer science (Radenski 2009).

**Conclusions from Workshop 1**

**Programming Languages for CS0**

Scratch, Alice, and AgentSheets are appropriate programming languages for CS0 courses, introducing students to logical problem-solving and programming constructs without undue emphasis on syntax and low-level details. They are engaging, media-rich, and interactive, encouraging imagination and experimentation with quick feedback.

**Programming Languages for CS1**

It is now widely accepted that the decline in computer science enrollments can be blamed, at least in part, on outdated CS1 courses. It was perhaps not so much the fault of the programming languages as the programming assignments in the course, which were often poorly motivated by applications that interested the students, offered little of the audio-visual feedback and interactivity that students expected from computers, and got the students bogged down in uninteresting details of syntax. From our experience, research, and discussions, we conclude that Java, Python, Flash/ActionScript, Processing, and C++ are all viable languages for CS1. Additionally, MATLAB is even a possibility to explore (Wirth and Kovesi 2006). The keys to success lie in the type of programming assignments given, the orderly presentation of concepts, and a good balance between
high and low levels of abstraction. Java, Python, Flash/ActionScript, and Processing have the advantage of lending themselves to audio/visual feedback. Processing may be a bit restrictive in its focus on bit manipulation, the event loop, and visual feedback, but it provides a palatable introduction to the Java environment. C++ has the advantage of being a more full-powered, real world language, and the potential for assigning interesting introductory-level programs that can be done in C++ should not be underestimated. (For examples of media-related programs that can be done in C++, see the learning supplements to The Science of Digital Media (Burg 2008a) (Pearson/Prentice Hall).)

**Programming Languages for Interdisciplinary Computer Science/Art Course**

Processing is a good choice for interdisciplinary courses that bring computer science and visual art students together. Similarly, ChucK would work well for interdisciplinary courses coupling computer science and music or sound design students. Processing. Flash with ActionScript is an obvious choice for interdisciplinary digital media programs and courses that focus on animation, video, or game development.

**Programming Assignments**

Computer science educators increasingly see the value of programming assignments that relate to an interesting application that the students recognize or care about. Among the obvious choices for application areas are graphics (Davis et al. 2007), image processing, sound and music (Meyers et al. 2009), and other physical systems (Freudenthal et al. 2010). Linking programming to application areas outside of computer science proper adds another dimension to both teaching and learning, as these assignments require application-specific knowledge in addition to the programming challenge itself. For example, for students to convolve an image, they need to know at least a little about what a convolution is. For them to apply a hand-programmed filter on a sound file, they may need to understand something of digital audio, time domain vs. frequency domain, and the Fourier transform. For them to do an LZW compression of a text file, they need to understand the algorithm. Yet we would argue that the added challenge is worth the effort, as it reaps benefits in motivation and interesting results.

Experience among the authors’ colleagues who have used high-level-abstraction programming languages and media-based computation has uncovered a pitfall. That is, if the level of abstraction remains too high and involvement in the applications or multimedia feedback takes priority, students in the end are not adequately prepared for CS2. It’s possible that in keeping them interested and entertained, we may shelter them from syntax, variables, and constructs for logical problem solving. Thus, it’s important to correlate the programming assignments with the programming constructs that are learned, as illustrated in (Meyers et al. 2009) and (Freudenthal et al. 2010).

Greenberg’s talk inspired the audience to try open-ended assignments that encourage exploration and creativity. Such assignments are particularly suitable for interdisciplinary courses, calling upon computer science students to think more creatively while the art students are being challenged to think more logically – creating a desirable synergy if handled well.
Suggested Changes to Computer Science Curriculum

1. As an introduction to programming and algorithm development, use media-rich languages in CS0 such as Scratch, Alice, AgentSheets, or Processing.

2. As feasible within a department, offer multiple sections of CS1, each with a different general application area and a base language appropriate to this. The learning objectives in all sections of CS1 must be the same so that the playing field is even for students entering CS2. Wake Forest University’s Computer Science Department is implementing this plan, offering alternate CS1 courses that vary by application area, as detailed in Appendix 1. Common topics and learning objectives for CS1, regardless of the base language, are also given in Appendix 1. These topics and learning objectives were adapted from those formulated in ACM Curriculum 2008.

3. Offer interdisciplinary elective courses (perhaps ½ semester) – for example, The Science and Art of Digital Images (base language: Processing; higher level development environment: Photoshop or GIMP and Illustrator, Xara, or Inkscape)

Workshop 2: Digital Sound for Computer Scientists, Musicians, Filmmakers, and Theatre Sound Designers

Overview

Workshop 2 brought together computer scientists, sound designers, musicians, and students. The students attending this workshop had just completed an eight week summer workshop on digital sound.

The introductory talk, given by the PI Jennifer Burg, attempted to show how the study of digital sound is relevant to the computer science curriculum, and how it relates to other disciplines such as mathematics, engineering (digital signal processing), physics, music, and theatre.

The second talk was given by Jason Romney, a digital sound designer from the University of North Carolina School of the Arts. This talk explained the practice of sound design, using as an example from the speaker’s recent work on The Phantom of the Opera. Romney showed how sound design progresses from artistic conception to practical details such as electrical power, wiring, speaker and microphone placement, and consideration of comb filtering and delays that affect audience perception.

The third talk, entitled “The REAL Business of Creating Sound for Television Programs and Commercials,” was give by Aaron Keane of Blazing Sound and Music in Raleigh, North Carolina. Keane does sound editing for commercials and television programs. He described the educational path he took to become a sound designer and showed his work process in designing sound for a power tools commercial and for the Jon and Kate Plus Eight television program.

Five students from the recently concluded digital sound workshop presented their research and creative results. This included songs that were written, performed, produced and arranged by the students, as well as sound processing programs written in C++ (e.g., vocoder and autotuner). The activities of the student summer workshop are described in (Burg and Romney 2009).
Videos, PowerPoint presentations, PDFs, and audio files related to the above talks can be found at [http://www.cs.wfu.edu/~burg/CPATH/Templates/workshops.php](http://www.cs.wfu.edu/~burg/CPATH/Templates/workshops.php) and [http://www.cs.wfu.edu/~burg/CCLI/Templates/workshops.php](http://www.cs.wfu.edu/~burg/CCLI/Templates/workshops.php). (Look under the appropriate workshop dates at these sites.)

It may be interesting to note that the PI and collaborator Jason Romney hosted a related workshop two weeks after the one under discussion. The workshop was entitled “Digital Sound for Computer Scientists, Musicians, and Sound Designers: What Can We Learn From Each Other?” and was sponsored by a National Science Foundation CCLI grant held by Burg and Romney (National Science Foundation CCLI grant DUE-0717743). This workshop continued to develop ideas about how digital sound can be made relevant and motivational within the study of computer science. The PI’s talk, “‘Ah-Ha! Moments’: Where Science Meets Art and Practice in Digital Sound,” ([http://www.cs.wfu.edu/~burg/CCLI/Workshops/AhHaMoments.ppt](http://www.cs.wfu.edu/~burg/CCLI/Workshops/AhHaMoments.ppt)) contains thoughts related to the discussion continued below.

**Conclusions from Workshop 2**

**Basic Computer Science Concepts Illustrated Through Digital Audio**

The primary conclusion drawn from Workshop 2 is that there are three intertwined concepts that should be more prominent in introductory computer science courses. These concepts are as follows:

1. Computers require a **digital encoding** of information. For information related to sight or sound, this could require a digitization process involving sampling and quantization. However, other encodings are possible, including those more symbolic in nature, like MIDI or vector graphics (in contrast to digital audio and bitmap images).

2. Working with information requires that a choice be made regarding the way in which the information is represented (**information representation**). This choice is important, as it has a great impact on how easily the information can be communicated and manipulated for a given purpose. Choosing to quantify a physical phenomenon (e.g., sound air pressure amplitude at a moment in time) as opposed to symbolizing the sound (e.g., symbolizing an instrument by some code, the playing of a note by another code, and so forth, as in MIDI) is a choice of information representation. Another important aspect is the **domain of representation** – the property according to which complex data is organized so that certain features can be seen and manipulated more clearly. Being able to transform data from one domain of representation to another – and recognizing when it is useful to do so – is a fundamental skill of computer scientists. Still another issue related to information representation is the scale of the representation, which once again affects how easily one can visualize and handle the properties under investigation.

3. Closely related to the process of encoding and information representation is the **processing of abstracting information** and **moving among levels of abstraction**. Encodings begin at the lowest level of abstraction – binary – but proceed up through levels of abstraction that get closer and closer to the way humans think and communicate. A fundamental activity for computer scientists is the process of abstraction, and a fundamental skill is the ability to move among levels of abstraction, always choosing one that is high enough to facilitate problem solving but low enough to allow sufficient access to details.
These three areas – digital encoding, information representation, and levels of abstraction – are basic to how computer scientists think and work, yet they are not given sufficient treatment in CS1 courses that focus largely on learning a particular programming language. Courses that take a broad view of computer science as a discipline often do deal with the topics above, as exemplified in class textbooks like Brookshear’s *Computer Science: An Overview*, now in its tenth edition (Brookshear 2009). However, many of these topics have been pushed back to CS0 courses or lost in the shuffle as CS0 courses moved toward web programming and media applications while CS1 focused on programming mastery. The ideal situation would be that computational thinking and basic digital fluency become fundamental subjects in K through 12 education. However, (Repenning and Ioannidou 2008) note that “the development of state-level curriculum standards for CS in the United States is nearly nonexistent,” citing the ACM K-12 Task Force Curriculum Committee (ACM K-12 Task Force Committee 2003). Until we have assurance that students learn fundamental computer science concepts in K through 12, we in higher education need to continue to find a place for these fundamental in courses required by computer science majors. If CS0 is not required, then we cannot omit putting these fundamental concepts into CS1. As (Repenning and Ioannidou 2008) note, the challenge is to “balance motivational and educational aspects by aligning interesting materials with principled educational frameworks.” We argue below that digital sound provides an application area in which these fundamental concepts and frameworks come to life.

The difference between analog and digital representations of data is an important concept for today’s student. The word “digital” is used to describe many of the devices we all interact with everyday, and the educated student ought to know precisely what that word means and what the difference is between, say, an analog and a digital telephone system. Learning about the digitization process – sampling and quantization – leads students to the subject of encoding, and how one encoding might be handier than another for a certain purpose. For example, music recorded as digital audio has its advantages – its fidelity to an actual recorded performance with all the nuances of the musicians and their instruments, stored in a format that lends itself to analysis and manipulation of frequencies. On the other hand, an entirely different type of encoding – MIDI – has different advantages, giving one the ability, through a computer interface, to change keys or instruments, erase errors, and alter timing with a simple menu selection.

The representation of sampled and quantified digital audio involves another choice, i.e., the choice between the time and frequency domains. In the time domain, digital audio is organized as a sequence of air pressure amplitude values that vary over time. In the frequency domain, digital audio is organized as a sequence of values that vary over the frequencies in the audible range – the value tell “how much” of each frequency is present in a clip of audio. The time domain may be more convenient for editing amplitudes at the sample level, while the frequency domain may be more convenient for equalization of frequencies.

Digital audio is a good context in which to explain the importance of scale as well. The human ear’s perception of loudness progresses in a nonlinear fashion as sound amplitudes increase. That is, if you turn up a not-very-loud single-frequency sound by \( x \), you’ll notice the difference a lot more than if you start at a significantly louder level and turn it up by \( x \). For this reason, it makes sense to represent loudness (sound pressure
amplitude) in logarithmic units – decibels – rather than linear units. Digital sound provides an excellent context in which to teach logarithms, a mathematical concept basic to computer science.

Digital sound design and music production are excellent application areas in which computer scientists can appreciate the importance of working at different levels of abstraction. Sound and music can be dealt with at many levels of abstraction, as listed in Table 6. Moving up a level hides details that don’t need to be touched for the moment. Moving down can lay bare these details and put them under programmer control. Computer science students need to learn how to move comfortably and appropriately from one level to another. The PIs three talks “Ah-Ha Moments” (Burg 2008e), “Digital Sound as Computer Science” (Burg 2008d), and “Points of View, Levels of Abstraction” (Burg 2008c) give examples of moving among levels of abstraction, taken from digital sound applications that can be used with computer science students.

<table>
<thead>
<tr>
<th>MIDI</th>
<th>Digital Audio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cakewalk Sonar, Logic, Pro Tools, Adobe Audition</td>
<td>Adobe Audition, Sony Sound Forge, Logic Pro Tools</td>
</tr>
<tr>
<td>Reason</td>
<td>MATLAB</td>
</tr>
<tr>
<td>MAX</td>
<td>MSP, ChucK</td>
</tr>
<tr>
<td>low-level language such as C</td>
<td>low-level language such as C</td>
</tr>
</tbody>
</table>

Table 6 Tools for working with MIDI and digital audio at various levels of abstraction

Programming and Computational Thinking Explored Through Sound and Music

The workshop discussions piqued the PIs’ interest in the possibility that programming or computational thinking might be taught in the context of music. Subsequent research uncovered some experiments in this direction. In one such experiment, an interdisciplinary computer science/music course Music Performatics (Heines, Greher, and Kuhn 2009) required that music students create their own music notation language, while computer science students wrote a notation programs that allows composition within the languages. In another computer/science interdisciplinary course, students were motivated to learn programming in Scratch through assignments that had them create small musical compositions that required loops, variables, modularization, and event processing (Ruthmann et al. 2010) (Meyers et al. 2009) describes an intensive three-week class called Musicomputation in which computer science and computer music were taught to 11-17 year old students. These students were chosen for their high potential and previous expertise in music and mathematics. The course inspired the learning of traditional topics such as variables, scope, type, and control structures with programs for creating music. Similarly but at the CS1 level, (Misra, Blank, and Kumar 2009) used a Python module and ChucK (a specialized audio programming language) to introduce imperative programming, objects, polymorphism, parallelism, and programming language syntax.

An area to explore is the relationship between algorithms and musical compositions and between data structures and musical entities. This is not a new subject. In 1980, Douglas Hofstadter won the Pulitzer Prize for Gödel, Escher, Bach: An Eternal Golden Braid, which explored the common use of symbols, structures, symmetries,
themes, and methods is mathematics, computation, and music (Douglas Hofstadter 1979). With the current interest in computational thinking and new interdisciplinary approaches to computer science education, teaching algorithms and structures with analogies to computer composition, theory, and symbols is gaining attention (Michael Edwards) (Eduardo Miranda, ed. and John Al Biles, ed. 2007).

In a more direct approach to teaching programming – by means of programming assignments that call for the use of certain variable types, control structures, data structures, or algorithm types – interesting problems related to sound and music can be used at the CS1 and CS2 level. The fact that these assignments create or do something interesting, with audible results, is motivating to the students. Ideas for assignments are listed in Table 7. Some of these assignments are described or implemented in (Burg 2008a) or (Burg and Romney).

<table>
<thead>
<tr>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>simulation of sampling and quantization of audio data</td>
</tr>
<tr>
<td>producing scales in different major and minor keys</td>
</tr>
<tr>
<td>comparison of just-tempered and even-tempered scales</td>
</tr>
<tr>
<td>noise dithering with the option of noise shaping</td>
</tr>
<tr>
<td>creating a vocoder</td>
</tr>
<tr>
<td>creating an autotuner</td>
</tr>
<tr>
<td>applying the Fourier transform (FFT, parallel FFT)</td>
</tr>
<tr>
<td>calculating optimal placement of loudspeakers</td>
</tr>
<tr>
<td>creating an impulse response filter to simulate an acoustical space</td>
</tr>
<tr>
<td>experiments the effects of comb filtering and the Haas effect</td>
</tr>
<tr>
<td>turning music into images (Evans 2005)</td>
</tr>
<tr>
<td>algorithmic composition</td>
</tr>
<tr>
<td>capturing and reinterpreting MIDI messages</td>
</tr>
<tr>
<td>creating or modifying a MIDI sample bank</td>
</tr>
</tbody>
</table>

Table 7 Programming topics related to sound and music

**Suggested Changes to Computer Science Curriculum**

1. Consider a music context for an introduction to programming and computational thinking in CS0.
2. Include concepts related to digital encoding, information representation, and levels of abstraction more prominently in CS1 (or CS0, if CS0 is required for computer science majors). Illustrate these with examples from digital audio (e.g., analog vs. digital representation, the digitization process, quantified vs. symbolic encodings, time vs. frequency domain, logarithmic vs. linear scale).
3. Include programming assignments related to sound and music topics throughout CS1 and CS2.
4. Offer interdisciplinary elective courses (perhaps ½ semester) – for example, Digital Sound and Music (base languages: ChucK and MAX/MSP or PD; higher level developments environments: Logic or Cakewalk Sonar, Reason)
Workshop 3: Drawing Students to Computing Through Visualization Technologies and Applications

Overview

The topic of Workshop 3 was “visualization” in a broad sense: basic visualization research; scientific applications of visualization (e.g., the visualization of molecules, cells, architectural spaces, or solar systems); information visualization; playful and experimental visualization of virtual spaces such as Second Life; and visualization as artistic expression. This workshop had four speakers, all of whom added interesting perspectives to the discussion.

The first talk, “Introduction to Visualization” (Victoria Interrante 2008), was given by Victoria Interrante from the University of Minnesota. Interrante described her basic research as the “design, implementation, and evaluation of methods for creating images of data, in order to enable that data to be more easily understood.” According to Interrante, the purpose of visualization is both knowledge discovery and knowledge communication. Visualization emerged as a field of research in computer science in the later 1980s. The first conference series began in 1990, and both the Information Visualization Symposium and Journal began in 1995. Interrante divided the field into scientific visualization and information visualization, noting that a big challenge in the latter is to use visual features to convey non-spatial data. Among the tools used in this field are the Visualization Toolkit (VTK), Amira, Tecplot, MATLAB, NLM Insight, ParaView, SciRus, and Processing. Key topics in the study of visualization are effective visualization design (e.g., mapping data variables to visual features and evaluating the capabilities and limitations of visual perception); classical visualization methods (e.g., volume visualization, flow visualization); and applications of visualization.

Interrante’s talk brought out the importance of algorithmic problem solving, data structures, modeling and simulation, geometry, trigonometry, and numerical analysis in preparation for work related to visualization.

The representative from business and industry was Mike Marsh, an applications engineer from the Visual Sciences Group at Mercury Visual Systems (Mike Marsh 2010). Marsh did a demonstration of one of their premier products, Avizo, a 3D visualization environment applicable to medical, industrial, climate, biomolecular, immersive, and information visualization, to name just some of its applications. He gave an overview of the educational background most valuable to a computer science student aiming to work in a company such as his. A background in mathematics is very important, including geometry, trigonometry, numerical analysis, and, more particularly, an understanding of multidimensional data structures, tensors, gradients, eigenvalues, convolutions, deconvolutions, and so forth. He also mentioned software engineering and programming skills such as unit testing, debugging, use of source repositories, use of libraries, and most importantly, logical problem solving skills. An interdisciplinary background relating biology, chemistry, physics, or mathematics to computer science would be of value as well. Marsh spoke enthusiastically about the potential for visualization applications to excite and motivate computer science students.

The third speaker, Michael Prushan, was from a neighboring discipline of computer science – chemistry – one of its primary links to computer science being
through visualization technology (Michael Prushan 2010). Prushan observed that chemistry is all about visualization, and has been from the beginning, citing Watson and Crick’s Nobel Prize for the structure of DNA as one of the most obvious examples of its importance.

Prushan passed out a paper written by Roald Hoffman entitled “Thoughts on Aesthetics and Visualization in Chemistry” (Hoffmann 2003). The importance of visualization in chemistry is expressed philosophically by Roald Hoffmann, who observes that visualization via models enters chemistry, as an inevitable corollary of the macro/micro motion that confusingly and productively mixes in the minds of modern chemists. Layers of iconic and symbolic representation of molecules mediate our struggle to propel ourselves into understanding why out of that blue solution comes a lime green precipitate. We understand, not just by cogitation, but through drawings of molecular models and orbitals, on blackboards and restaurant napkins.

Prushan gave examples of molecular visualizations which were improved by better design of perspective, depth cues, and attraction of viewer focus. To the previous list of topics and skills that pave the way for work in visualization, Prushan and the subsequent discussion added pattern matching, computational geometry, and group theory. Prushan also emphasized basic problem solving skills.

The fourth speaker, Meg Mitchell, spoke about “Data Visualization in the Processing Development Environment.” This was a totally different perspective from the previous talks. Like Greenberg in Workshop 1, Mitchell advocated “play” and experimentation in programming for creative expression, with no hypothesis or fixed goals. Rather than use well defined data structures for problem solving purposes, a programmer who aims at “creative visualizations” might combine data in unpredictable ways and observe the structures that emerge. Mitchell recommended Paul Graham’s Hackers and Painters (Paul Graham 2004) and Ben Fry’s Visualizing Data (Fry 2008) as good sources on artistic and creative visualization.

Conclusions from Workshop 3

Required Courses

Work in visualization – particularly for scientific applications – requires a background in mathematics. We should be able to assume that students have had geometry and trigonometry in K through 12 education. Linear algebra is an appropriate requirement for the computer science major (and is currently common in computer science programs). Numerical analysis is also appropriate for students concentrating on visualization. A computer science graphics course would also be valuable.

Programming Languages

MATLAB could be considered as a programming language for those interested in visualization. Its graphing ability and toolkits for things such as image and digital signal processing make it powerful for visualizing data. It is also possible to move from higher to lower levels of abstraction with the scripting language. Java could also work well, as it has graphics packages for displaying images.
Processing is a good language for artistic visualization, as was shown in Mitchell’s talk and Greenberg’s talk in Workshop 1.

Three-dimensional programming environments such as Maya would also be good for students working in visualization, although not at an introductory level.

**Basic Concepts: Modeling and Simulation**

Modeling and simulation are basic concepts for anyone wanting to work with visualizations, and thus these concepts should be incorporated into introductory computer science courses. Modeling can be defined as “a simplified mathematical description of a system or process, used to assist calculations and predictions” (AskOxford.com 2010). A simulation is the use of a model to imitate a system’s behavior, often as a way of predicting the behavior of the analogous real-world system. Although visualizations give a graphical representation, the first thing for students to master is how to create a model, turn this model into a computer program, and with it perform a simulation.

The above languages have built-in functionality for displaying images, which one would think is essential for students wanting to specialize in visualization. However, more traditional languages such as C and C++ could be used very effectively in introductory courses with assignments that help students understand the processes of modeling and simulation. (See, for example, Stanford University’s online course entitled “Object-Oriented Programming: A Modeling and Simulation Perspective” that uses C and C++ (Stanford University 2010). Georgia Tech has a computer science course called “Introduction to Modeling and Simulation” that uses C and Fortran (Georgia Tech 2010)). There are many areas from which modeling and simulation assignments can be taken – event-driven simulations like customers lining up for service; interactions in ecological systems; failure tolerance in computer networks; stock market behaviors; weather prediction; and on and on.

**Suggested Changes to Computer Science Curriculum**

1. Require numerical analysis and graphics for those specializing in visualization.
2. Incorporate modeling and simulation in programming assignments at the CS1 and CS2 level.
3. Consider adding a specialized elective course in modeling and simulation.

**Workshop 4: The Revolution Will be Televised: Revitalizing the Union of Computer Science and the Media**

**Overview**

Workshop 4 explored the role that computer scientists might play in the world of television production. The representatives from business and industry were Nathaniel Borenstein, Chief Open Standards Strategist and Distinguished Engineer at IBM, and David Halvorsen, Project Manager for Overdrive, a system for live television production control. The representatives from a neighboring academic discipline were Michael Niederman, Chair of the Television Production Department at Columbia College and Wojciech Lorenc, Digital Media Technologist for The Center for Instructional Technology at Columbia College.
The first talk was given by Nathaniel Borenstein, co-originator of the MIME standard and author of *Programming As If People Mattered: Friendly Programs, Software Engineering, and Other Noble Delusions* (Borenstein 1994). Borenstein spoke about the need for and difficulties of inter-departmental collaboration. Borenstein’s career has been focused on taking new innovations and driving them, collaboratively, to become successful products. His talk generated discussion about the tensions between “creative types” and “computer geeks” as they collaborate. Borenstein advocated the explicit teaching of collaboration and cooperation in computer science departments. As explained in his book, he also advocates a user-centered approach.

Software engineering has struggled so valiantly, and so single-mindedly, with the incredible problems of creating large software systems that it has for the most part failed to acknowledge the new problems that have been introduced by the demand for better user interface technology (Borenstein 1994).

Borenstein gave an overview of his professional experience. Among his “lessons learned” are these:

- Great technology cannot succeed unless it fills a customer need.
- There is no greater thrill than seeing other people use and enjoy the product of your work.
- Like it or not, patents are too valuable for a business to neglect.
- Be flexible in your business model.
- Avoid turf wars with big institutions.
- Technology is the easy part. Interpersonal relationships are what change the world.

These observations reflect the essence of Borenstein’s professional philosophy, which is simultaneously practical and human-centered. The main take-away from his talk was the idea that – even in the face of all the difficulties of collaboration – a computer scientist should aim to create products that are designed with the human user in mind and that are developed to the point where they are actually used.

Michael Niederman’s talk provided an historical context for the evolution of television production equipment. Niederman described the evolution of television equipment from analog devices to hardware and software systems developed by Aldus (Pagemaker and Quark xPress), Avid, Adobe (Hitchcock, one of the early nonlinear video editors), Grass Valley (a production switcher), Pixar, and EVS (“Instant Tape Technology” for television production). As the Chair of the Television Department Columbia College, Niederman oversees courses that have students doing real production, including “Newsbeat,” a daily newscast produced in collaboration with the Broadcast Journalism Department, and “The Rent” a new webseries. At the end of the workshop, Niederman gave the participants a tour of the department’s television production facilities and allowed them to watch part of the students’ daily newscast production.

The talk by David Halvorsen gave the participants a close look at a specific hardware/software system used in television production and broadcast, particularly news productions. A newscast is a complex, fast-paced event requiring the coordination of scripts, video, character generator data, teleprompter control, director notations, camera operator cues, and timing estimates for each section of the show. (A character generator, or CG for short, is a device or software that produces static, rolling, or crawling text that can be superimposed on video footage.) Ross Video’s Overdrive is a live production
control system that automates newsroom processes, with features for editing and transmitting text; controlling character generators, video servers, and robotic camera systems at multiple locations; and generally doing this without requiring human monitors at each workstation (Figure 5).

Figure 5  Ross Overdrive system layout (from the Ross Overdrive brochure)
An important part of this system is MOS (Media Object Server) technology, a protocol that integrates the handling of text-based material (i.e., news copy) with media objects (i.e., audio/visual material). MOS can be used to control and coordinate video servers, image management systems, audio devices, and character generators (Qureshi 2009). It was enlightening to see a system used extensively in the industry and to consider the kind of knowledge a computer scientist would need to participate in the development of such systems. As can be seen from Figure 5, Overdrive is a complex system whose development required collaboration among software engineers, interface designers, networking specialists, audio/video engineers, robotics specialists, news producers, workflow analysts, and more.

Wojciech Lorenc, a Digital Media Technologist for The Center for Instructional Technology at Columbia College, gave a talk contrasting traditional television with web video and video on mobile media. Lorenc gave an interesting enumeration all the things that traditional television is not.

- Traditional television is not mobile. On your cell phone, you can shoot video, edit, and upload the video online, anywhere on-the-go. Mobile video is the way of the future.
- Traditional television is not well targeted. Lorenc referred to web-based and mobile-media-based video as “citizen media,” existing in a forum that allows for a smaller, more precisely-targeted niche audience. He observed that with a smaller audience, we can take more risk and be more innovative. As an example of individual creativity with commercial benefits, he showed the participants an episode of the popular webseries “Val’s Art Diary,” which was nominated for a YouTube aware in 2006. In each episode, the charming artist, Val, describes how she created a particular work. Viewers can buy her artwork and interact through Twitter, Facebook, blogs, and special interest groups.
- Traditional television doesn’t facilitate direct commerce. Lorenc asked the participants to think about the process of buying something from a shopping channel: pick up the phone, dial, give the item number, give your credit card number, and so forth. Television does nothing to facilitate the process.
- Traditional television is not interactive. Users are becoming accustomed to the kind of interactivity available on the web. YouTube became interactive in June 2008. “Sexperts,” a webseries created by Lorenz and shown on its own channel on YouTube, illustrates some of the potential of interactive web video. The series is a sitcom, each show a five minute humorous story with an interactive quiz at the end. The series entertains and at the same time educates people about safe sex. The series exploits its commercial potential with ads and promos in the lower third of the screen.
- Traditional television is not social. Traditional television doesn’t tell you who else is watching a show and what other viewers might be thinking. It has none of the attractions of social networks. As an example of what can be done with mobile and web-based media, Lorenc described an application that allows you to pick one of Val’s artworks and have the names of friends put on the painting.
- Traditional television is difficult to measure.
It is difficult and time-consuming to analyze the number of viewers and type of audience for television programs. In web-based video, the fact gathering and analysis tools can be integrated directly into the technology, as exemplified by Google Analytics (Google).

- Traditional television is not personal.

Lorenc questioned why our televisions offer us content that we don’t like, have already watched, or isn’t relevant to our interests. He contended that if programming is better targeted, advertising can be better targeted as well. He cited a recent innovation at Google: the inclusion of video ads into text ads inserted into search-result pages (Krazit 2009).

- Traditional television is not equipped for aggregated media.

With tagging, keywords, and social bookmarking, web-based information and entertainment can be aggregated into a single location. This isn’t possible with traditional television.

- Television is not very adaptable.

Lorenc contended that too many interested parties have to agree in order for change to take place in the television industry, slowing the pace of innovation.

Lorenc’s talk outlined how the television industry is going to have to change as we go through this period of media convergence. Such observations are relevant to computer science students as they educate themselves to contribute to the future technology.

Conclusions from Workshop 4

Suggested Changes to the Computer Science Curriculum

In the discussions during the workshop, participants emphasized a number of subjects that they believed are important for computer science students wishing to play a part in the television industry, whether in the development or use of television production technology.

- Digital media – learn how audio and video are captured and processed digitally.
- Networks – learn how devices communicate (analog and digital), become familiar with transmission protocols, and industry standards.
- User-interface design – learn the importance of proper user interface design, issues of human-computer interactions; learn how to base design on metaphors that are appropriate for the habits, mind styles and work conditions of the target group.
- Database management – learn the importance of database management in a complex environment like television production. Learn particular languages and environments like SQL.
- Mobile and online media – learn to how produce, access, and distribute media through mobile devices and online media.
- Social and ethical issues – Be aware of the regulatory environment, business practices, unions, electronic property rights. Be aware of ethical issues like open access to information vs. privacy.
- Entrepreneurship – consider how products are developed to the point where they have commercial value.
• Collaborative work – gain experience in collaborating with others from different disciplines, including on large projects.

These topics could constitute required courses for students wishing to focus on areas related to television or other digital media, or they could be woven into existing courses.

Workshop 5: Game Computing, Story Telling, and AI

Overview

Workshop 5 explored an alternative view of computer science, recasting computer science as computational media, where logic and problem solving are taught through games and playable media. The representative from business and industry was Brian Moriarty from Worcester Polytech. Moriarty is a professional game developer with 30 years experience. The representative from a neighboring academic discipline was Michael Mateas, creator and Chair of the Computational Media Program at the University of California Santa Cruz.

Mateas gave the first talk, in which he proposed a different way of thinking about programming, describing computer science as a “representational practice most closely exemplified in artificial intelligence.” His view is that game programming provides a context in which students can learn the essential elements of programming and problem solving – content (i.e., meaning), rules (i.e., code) and assets (i.e., data). Following up on an approach conceived by his colleague Noah Wardrip-Fruin, Mateas divides the logic of game programming into two parts – operational logic and graphical logic. In his definition, “operational logics connect fundamental abstract operations, which determine the state evolution of a system, with how they are understood at a human level.” The intent is that operational logics connect technical implementation strategies with authorial and audience meaning. This abstraction is explained in more detail in a 2009 paper (Mateas and Wardrip-Fruin 2009).

The success of UC Santa Cruz’s Computational Media program – which has considerably more students than the traditional computer science program – attests to the attraction that game programming can have for students. However, Mateas attempts to go beyond the “fun” level of a game programming curriculum to develop a theory upon which it can be built. This adds a research component to the hands-on game development side. His talk led the group to discuss game programming as a way of exploring a computational space – for example, a baseball game as a way of exploring a Markov chain.

The practice of game development was presented in the second talk, given by Brian Moriarty, who has been in the business of designing, building, and deploying games and game engines for 30 years. Moriarty gave an overview of the evolution of game programming, beginning in the years when games were printed as text-code in game magazines and typed into computers by game lovers. He proceeded through games developed at Infocom and Lucas Films, including Beyond Zork, Maniac Mansion, and Loom, all of which Moriarty worked on.

When asked what he thought was most important to teach computer science students, Moriarty was very helpful with his answer. He listed the following:
• The ability to create software (game engines, for example) that have the qualities of stability and high performance, that are easy to maintain and extend, and that are well documented.
• The ability to do fast prototyping with rapid iteration of versions.
• The ability to communicate with a designer, understand what the designer needs in a project, and understand what levels of the implementation need to be accessible to the designer. (For example, the programmer of a game engine must understand what parameters should be available to the game designer when he or she uses the game engine to create a game.)
• Familiarity with issues regarding networks, latency, distributed systems, and scalability.
• The ability to work with 2-D and 3-D graphics.
• Understanding of motion and collision detection.
• Familiarity with aspects of artificial intelligence.
• Knowledge of scripting languages.
• Understanding of issues of human-computer interfaces, even at the hardware level.

The group discussed programming languages that are appropriate for a computational media program. Moriarty strongly advocated scripting languages that lend themselves to fast prototyping – in particular, Lua. Other languages discussed as appropriate for beginning students were Java, ActionScript, Flash, and Processing. Games engines that were cited included GameMaker, Scumm, Unity (for mobile device game programming), Blender, and UnrealScript.

Conclusions from Workshop 5

Suggested Changes to the Computer Science Curriculum

The last session of the workshop was an animated discussion about what should be taught to computer science students to enable them to branch to or interface with other disciplines. A spindle-shaped curriculum was proposed by Mateas, with a broad base at the bottom, a narrower core through the middle, and a wide cap at the top leading again to other disciplines. While the group never entirely agreed on what specific courses or topics should be included in each of these parts, some ideas stood out of the discussion. Most discussants seemed to advocate fairly traditional topics for the base – logical problem solving, algorithms, control structures, functional decomposition, variables, static vs. dynamic structures, object-oriented programming, levels of abstraction, and so forth. Most seemed to agree that these topics should be taught via specific applications – perhaps from other disciplines that might be of interest to students. In the core, the courses and topics that were put forward as essential included operating systems, data structures, databases, networks, architecture, and software engineering. Mateas strongly advocated a capstone project for all students.

Among the traditional courses that some discussants thought could be omitted were programming languages and a two-semester data structure/algorithms sequence. One participant argued that data structures should be taught at a high level of abstraction, with emphasis on use of existing libraries of code like C++’s stl, rather than having students do their own low-level implementation of dynamic data structures like stacks,
queues, and trees. This participant advocated only one data structures/algorithms course rather than a two course sequence.

Courses and topics that would be appropriate for a computational media program can be inferred from the topics that Moriarty outlined in his talk. These include software engineering, graphics, networks, operating systems, artificial intelligence, concurrency, HCI, and programming courses that include scripting languages. As was the case in previous workshops, the student’s ability to collaborate with others from other disciplines was emphasized. In particular, game programming students should have an appreciation of the importance of narrative, graphical design, and human computer interface issues.

**Workshop 6: Collaborating Across Silos to Create Interdisciplinary Game Design and Development Curriculum**

**Overview**

Workshop 1 was help on September 17 and 18, 2010 at East Tennessee State University in Johnson City, TN. The representatives from business and industry included Lewis Pulsipher (board game designer since 1980), Chris Wren (formerly of EA and Micropros), Steve Chelton (technical artist with Atomic Games), Roberto Garcia-Lago (game designer from n-space), and Rob Maxwell (animator, modeler, and lighting artist from Vicious Cycle). The representative from a neighboring academic discipline was Susan Gold, faculty member of the graduate gaming program at Full Scale in Orlando and (then) Education Chair at the International Game Developers Association.

Susan Gold was the first to speak, giving the benefit of her experience and extensive contacts in the education and gaming industry.

**Conclusions from Workshop 1**

**Suggested Changes to Computer Science Curriculum**

**Workshop 7: Setting a Curriculum for Mobile Media and Computing**

**Overview**

This workshop will be held at Miami University on October 15 and 16 in Oxford, Ohio.
References


---. 2008c. Points of View and Levels of Abstraction presented at the ThinkTank 2008 Symposium, April, Wake Forest University, Winston-Salem, NC.


Appendix 1

A Curriculum Revision at Wake Forest University
Offering Multiple Sections of CS1 That Vary by Application Area and Language
Spring 2010

The “CS1” course at Wake Forest University is CSC111 Introduction to Computer Science, described in the college bulletin as follows:

Introduction to the basic concepts of computer programming and algorithmic problem solving for students with little or no programming experience. Recommended as the first course for students considering a major or minor in computer science, also appropriate for students who want computing experience applicable to other disciplines. The programming language used and the focus will vary, as listed below. Lab-2 hours. (D)

a. general purpose computing  d. mobile computing
b. multimedia and game computing  e. business applications of computing
c. scientific and mathematical computing

Below are brief descriptions of the sections planned for Fall 2010:

111a (A): A general introduction to programming and algorithm development using Java. Similar to the current CSC 111.

111b (A): An introduction to programming and algorithm development using the Processing programming language, a language designed to provide instantaneous visual feedback and to simplify development of visually-driven programs. This section will focus on the use of imaging, animation, and interactivity to teach the fundamental abstractions employed in the field of computer science. This section will be particularly appropriate for those students who are interested in the visual arts or who are visual thinkers. Processing is a variation of the Java programming language.

111b (B): An introduction to programming and algorithm development using Python, a simple yet powerful programming language with extensive built-in libraries and user interface tools for easily creating multimedia and Web-based applications. Python is widely used in some of the largest installations on the Internet, including Google, YouTube, NASA, and Industrial Light & Magic. Python's simplified syntax aids learning the fundamentals of computation without too many details getting in the way.

It is anticipated that future offering will include, for example, a “mobile computing” section taught in Java or Objective C (depending on the mobile phone platform); a “multimedia and game programming” section taught in Flash; a “business applications of computing” taught in Visual Basic; and a “scientific and mathematical computing” section taught in MATLAB.
Learning Objectives for all Sections of CSC111 (aka CS1) at adapted from ACM Curriculum 2008 by Department of Computer Science, Wake Forest University, May 2010

**CS1: Programming Fundamentals**
Fluency in a programming language is prerequisite to the study of most of computer science. Undergraduate computer science programs must teach students how to use at least one programming language well; furthermore, computer science programs should teach students to become competent in languages that make use of the object oriented and event-driven programming paradigms.

This knowledge area includes those skills and concepts that are essential to programming practice independent of the underlying paradigm. As a result, this area includes units on fundamental programming concepts, basic data structures, algorithmic processes, and basic security. These units, however, by no means cover the full range of programming knowledge that a computer science undergraduate must know.

**Programming Languages and Environments**

*Topics:*
- Compilation compared to interpretation
- Integrated development environments (IDEs)
- Introduction to computer hardware and software organization

*Learning Objectives:*
1. Choose a compiled or interpreted language appropriately for a problem.
2. Choose an appropriate IDE for program development in a given language.
3. Use an IDE for program development.
4. Know the basic hardware and software components of a computer.
5. Know how a compiled or interpreted program is executed.

**Fundamental Constructs**

*Topics:*
- Basic syntax and semantics of a higher-level language
- Variables, types, expressions, and assignment
- Simple I/O • Conditional and iterative control structures
- Functions and parameter passing
- Structured decomposition

*Learning Objectives:*
1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs covered by this unit.
2. Modify and expand short programs that use standard conditional and iterative control structures and functions.
3. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, and the definition of functions.
4. Choose appropriate conditional and iteration constructs for a given programming task.
5. Apply the techniques of structured (functional) decomposition to break a program into smaller pieces.
6. Describe the mechanics of parameter passing.

**Algorithmic Problem Solving**

*Topics:*
- Problem-solving strategies, including simple patterns
- The role of algorithms in the problem-solving process
- Implementation strategies for algorithms
The concept and properties of algorithms
• Types of errors: syntax, logical, and run-time
• Debugging strategies, including some simple unit testing
• Introduction to exception handling

Learning Objectives:
1. Discuss the importance of algorithms in the problem-solving process.
2. Identify the necessary properties of good algorithms.
3. Create algorithms for solving simple problems.
4. Use pseudocode or a programming language to implement, test, and debug algorithms for solving simple problems.
5. Describe strategies that are useful in debugging.

Data Types and Data Structure

Topics:
• Representation of numeric data at an introductory level
• Range, precision, and rounding errors
• Arrays, lists, vectors, or records
• Representation of character data
• Strings and string processing
• Runtime storage management
• Pointers and references
• Strategies for choosing the right data type or data structure from those learned so far, introductory level

Learning Objectives:
1. Describe the representation of numeric and character data.
2. Understand how precision and round-off can affect numeric calculations, at an introductory level
3. Discuss the use of primitive data types and built-in data structures.
4. Describe common applications for each data structure covered.
5. Implement the user-defined data structures in a high-level language.
6. Write programs that use the following data structures: arrays, array lists, vectors, strings, or records or other simple data types, as appropriate.
7. Choose the appropriate data structure for modeling a given problem, at an introductory level.

Recursion

Topics:
• The concept of recursion
• Recursive mathematical functions
• Simple recursive functions
• Divide-and-conquer strategies

Learning Objectives:
1. Describe the concept of recursion and give examples of its use.
2. Identify the base case and the general case of a recursively defined problem.
3. Describe the divide-and-conquer approach.
4. Determine when a recursive solution is appropriate for a problem.

Event Driven Programming

Topics:
• Event-handling methods
• Event propagation
• Exception handling

Learning Objectives:
1. Explain the difference between event-driven programming and command-line programming.
2. Design, code, test, and debug simple event-driven programs that respond to user events.
3. Develop code that responds to exception conditions raised during execution.
**Object Oriented Programming**

*Topics:*
- Object-oriented design
- Encapsulation and information-hiding
- Separation of behavior and implementation
- Classes • Libraries and APIs and the use of their documentation

*Learning Objectives:*
1. Justify the philosophy of object-oriented design and the concepts of encapsulation, abstraction, inheritance, and polymorphism.
2. Design, implement, test, and debug simple programs in an object-oriented programming language.
3. Describe how the class mechanism supports encapsulation and information hiding.
4. Understand how libraries and APIs are used in program development.

**Secure Programming**

*Topics*
- Important of checking for and avoiding array and string overflows
- Programming language constructs to avoid and alternatives
- How attackers use overflows to smash the run-time stack

*Learning Objectives:*
1. Rewrite a simple program to remove a simple vulnerability
2. Explain why or why not a buffer overflow is possible in the programming language you know best
3. Explain why one or more language constructs may lead to security problems such as overflows.