τέχνη: Trial Phase for the New Curriculum

Timothy Davis, Robert Geist, Sarah Matzko, James Westall
Department of Computer Science
Clemson University
Clemson, SC 29634-0974
{tadavis|rmg|smatzko|westall}@cs.clemson.edu

ABSTRACT

The τέχνη project, which provides an unusual alternative to the standard design of the computing curriculum for the bachelor's degree in computer science, is now entering full-scale implementation at Clemson University. The approach relies extensively on problem-based instruction, computer graphics, and the notion of cognitive apprenticeship. The novelty arises from the magnitude and origin of the problems to be integrated into the curriculum and the breadth of impact across the curriculum. The first three courses in the new curriculum are now being taught. The design of each course is described, and preliminary assessments from earlier trial sessions of the first two courses are offered.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Curriculum*

General Terms

Design, Experimentation, Human Factors

Keywords

computer graphics, curriculum design, problem-based instruction, ray-tracing, τέχνη

1. INTRODUCTION

The keyword in the title, τέχνη, is the Greek word for *art*. It shares its root with τεχνολογία, the Greek word for *technology*. The goal of the τέχνη project [3] is a zero-based re-design of the undergraduate degree in computer science. A principal motivation for this re-design is to effect a reconnection of the artistic and scientific components of our students' intellectual development, components which are, unfortunately, separated early and decisively in the current educational system of the United States.

A foreshadowing of the potential benefits of such reconnection has been provided by the early successes of the Digital Production Arts program at Clemson. The M.F.A. in Digital Production Arts is a two-year, graduate degree program that is aimed at producing digital artists who carry a solid foundation in computer science and thus can expect to find employment in the rapidly expanding special effects industry for video, film, and interactive gaming. All students in the program are required to complete significant graduate level work in both the arts and computer science. Graduates of the program have been tremendously successful in finding employment.Equally important is the observed demographic effect. The DPA program has consistently maintained an enrollment that is more than 30% women and more than 12% African American, both well above the enrollment levels for those groups found in other graduate programs in computing, including those at Clemson.

Motivated by DPA, the τέχνη curriculum design relies heavily on a careful selection of large-scale problems from the visual domain. We take a *constructivist* view in this design, but, as noted by Duffy and Cunningham [2], this term has come to encompass such a wide span of directions in education that further specification is essential. Our view is certainly Piagetian, sometimes termed *cognitive constructivism*, and draws directly from Piaget [10], Dewey [4], and Rousseau [14] in that fundamental tenets are these:

- Learning is an active process of constructing individual knowledge.
- Learning occurs when observations differ from expectations, and new mental models must be constructed to accommodate the differences.
- Teaching is the process of invoking and supporting these constructions.

It is not surprising then that we use problem-based instruction as a vehicle, but our design differs from others in the depth and scope of the problems addressed. Fundamental tenets of our approach are that a visual problem domain will most quickly capture the attention and interest of students who have grown up in a society that is increasingly visually oriented, that a connection between scientific and artistic components will stimulate both deductive thought and creativity, and that toy problems would be of little value in effecting the principal desired accommodation, an ability to solve real problems. Thus the problems are large-scale, with one problem per semester, and they are selected from problems that have arisen naturally in the research investigations of the graphics faculty. These problems are CPU and storage intensive, yet simple to describe and address. A primary benefit of the class of problems is that none has a strictly "right" or "wrong" answer, and thus students must begin to develop non-trivial methods for evaluation of their own work.

Because the problems are large and often without bounds, we do provide a conceptual framework within which the students explore and build. Thus our method is probably best characterized as constructivism with an objectivist scaffolding [9], or, to those who

SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.

study epistemology, a naive retracing of first steps in the direction of Kant's transcendental logic [7].

In this context, with very large problems, we can also rely on the *cognitive apprenticeship*, a concept probably due to Resnick [13]. We contend that there is great value in observation of a master at work in problem solving. The process is the key. We thus encourage instructors to present solutions to problem components in a manner that is as close as possible to their own original solutions, including any and all missteps. Such solutions carry a vitality that is missing in solutions that have been cleaned and polished for presentation in more formal settings. The accomplished mathematician Wilhelm Stoll was once asked to explain how mathematicians translate their motivation into mathematical proof. He replied that a proof is motivation written backwards. We favor direct presentation of the motivation.

Of course any zero-based design must begin with an identification of goals, in this case, fundamental concepts and abilities to be developed in the undergraduate student of computer science. In this endeavor, we recognize the rapidly changing nature of the discipline and focus on the challenges the students will most likely face in the years after graduation. Ability and enthusiasm for computational problem-solving, rather than experience with popular software or hardware platforms, should then be of paramount importance. To this end, we have enumerated and categorized concepts in computer science that we deem absolutely essential to computational problem-solving. We have collected closely-related concepts to form the structural goals of individual courses and then identified semester-long problems from the visual domain that will invoke an exploration of these structures. The purpose of this paper is to describe the first three courses in this new design and provide some preliminary assessment. As of this writing (Fall, 2006), all three courses are being taught.

2. PROBLEM/LANGUAGE SELECTION

Our initial focus has been on defining the introductory sequence in the new curriculum. Hereafter we will refer to these courses as CS1, CS2, and CS3.

The programming languages of instruction for the courses were chosen carefully. We had to remain consistent with our constructivist design, with its roots in empiricism, but we also needed a tight integration with the semester-long problem in each course, so that the language provided a convenient vehicle for solution. We also sought a smooth, natural transition from each language to the next.

In all of our course design, we emphasize, even to a greater extent than does Resnick [13], the critical importance of appropriate mental models of complex systems. Early development of a mental model of computing, as described by the von Neumann architecture, is crucial to undergraduate student success. The attendant flexibility in response to unexpected situations cannot await post-graduate training. In this case, the use of a small, imperative programming language that is close to the machine itself is most conducive to this development, and so we use *C* in both CS1 and the first part of CS2. We are aware of arguments that advocate immersion of students in the object-oriented paradigm and use of an object-oriented language from the outset, but we reject those arguments on the same empiricist basis that society, decades ago, rejected the rationalists' "new math." Learners are simply unable to abstract details they have never seen and of which they are completely unaware. Learning naturally proceeds by accommodation from specific evidence and detailed observations to theoretical (operational) models.

A strong argument can be made that the extreme flexibility of *C* can allow bad programming habits to develop, and we do not dispute this, but the principal argument against the use of *C* in a first course appears to be centered on its complexity of expression. Dingle and Zander [5] point to the classic, fast string copy as evidence:

```
while(*s++ = *t++); // tricky for CS1 students
```

Yet, if an accurate mental model of the machine is in place, is this really tricky? It is somewhat ironic that, in his January, 2005, advice column to computer science students [15], Joel Spolsky refers to the very same example, with a very different take, "I don't care how much you know about continuations and closures and exception handling: if you can't explain why 'while(*s++=*t++);' copies a string, or if that isn't the most natural thing in the world to you, well, you're programming based on superstition, as far as I'm concerned: a medical doctor who doesn't know basic anatomy, ..."

The principal semester problem for CS1 is transferring color from one 2D image to another. It is a visual problem with a simple description that is accessible to a novice and yet induces an exploration of the roles of sequential execution, cache effects, dynamic allocation and access of internal (main memory) and external (disk) storage, information representation, bus contention, and the costs of resource access. Students learn to program in this problem context, which places particularly strong emphasis on file I/O, arrays and internal storage, looping constructs, and function calls.

The principal semester problem for CS2 is constructing a ray-tracing system for rendering synthetic images. It provides a natural introduction to recursion, pointers, linked lists, parsing, data structures, and performance issues. From a pedagogical perspective, the development of a ray-tracing system provides an ideal mechanism for exposing the student to the object-oriented paradigm in a way that decouples the paradigm from its implementation in a particular programming language. The system implementation is initiated in an imperative style, but it quickly becomes apparent that the most reasonable way to represent the interactions of photons or rays with different types of geometric objects is to associate specific functions for calculating ray-object intersection points and surface normal vectors with each type of object. The overall design is drawn, in a very natural way, into the object-oriented paradigm. The benefits of inheritance and polymorphism are clear from the onset of their introduction. Because the systems naturally grow large and complex very quickly, techniques for partitioning, testing, and large-scale development are well-received. When an object-oriented language (*C++*) is introduced in the latter half of the course, it can be regarded as a welcome tool for reducing workload and complexity, rather than as a burden of additional material to be mastered.

The principal semester problem for CS3 is constructing surfaces that best approximate extremely large point clouds in 3D space. The course is similar, in terms of topic coverage, to a classical Data Structures course. File structures, searching, sorting, hashing, kd-trees, modular design, and principal component analysis are all integral to the solution. Interest in performance and its relation to algorithm complexity are motivated early: The students are asked to read the list of data points and, for each point, find its three nearest neighbors. The students start with the obvious $O(N^2)$ algorithm. But, the list can contain 10 million points! When students tire of waiting for an answer, they are naturally drawn to investigate alternatives. The course language is *C++*. Java is introduced in CS4.

3. A CS1 PROBLEM SOLUTION

An algorithm used to solve the problem of color transfer is due to Reinhard, Ashikhmin, Gooch, and Shirley [12]. All image files are maintained in the portable pixmap (.ppm) format, which offers

easy parsing at some cost in representation efficiency. In this format, each pixel is represented as three successive bytes that specify its red, green, and blue (RGB) levels. The principal difficulty in using RGB colorspace for color modification is that natural images (photographs) exhibit a high degree of correlation among the channels. If a pixel has a large blue value, it is likely to have large red and green values as well. For this reason, Reinhard et al work in the so-called $l\alpha\beta$ colorspace, which offers a compact representation with little correlation among channels. The transformation from RGB to $l\alpha\beta$ (and back) relies on a simple 3x3 matrix multiplication. Once all the pixels are converted to $l\alpha\beta$, the three components can be treated independently. Means and standard deviations for each component are computed for both the original, source image, and the target, color-transfer image. Source means are subtracted from each source pixel, and then each is scaled by the ratio of target standard deviation to source standard deviation. Finally, the target means are added to each pixel, and a 3x3 multiplication is applied to return to RGB space.

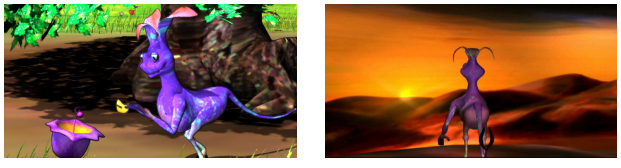
An example is shown in Figure 1, where frames from the digital animation, *Kneeknocker's Nose*, shown in Figure 2, are used to transfer color to a photograph of the Campanile of Trinity College, Dublin.

(a) (b) (c)

Figure 1: *Color Transfer:* (a) original photograph, (b) color transfered from 2a, (c) color transfered from 2b.

4. A CS2 PROBLEM SOLUTION

The scaffolding for our ray-tracing solution has been presented earlier [3]. Here we review only the principal features. Key to success is a carefully defined structure to represent the "objects" in the scene. A typical example is shown in Figure 3. Note the extensive use of function pointers.



(a) (b)

Figure 2: *Color Transfer:* (a) color palette transfered to 1b, (b) color palette transfered to 1c.

A *color* is specified as a triple of RGB intensities in the range $[0,255]$. The functions *ambient* and *diffuse* return coefficients representing the degree to which the surface of the object reflects red, green, and blue components of incident light. These functions typically return constant values but the functional representation supports procedural textures such as a checkerboard floor. An object's *hits* function is responsible for determining if and where a given ray intersects this object. The *get_normal* function returns a unit vector normal to the surface at any point.

```
struct object {
        struct color (*ambient)();
        struct color (*diffuse)();
        struct color mirror; /* weight on specular */
        void (*get_normal)();
        int (*hits)();
        union {
                struct ball ball;
                struct floor floor;
                } config;
        struct object *next;
        };
```

Figure 3: Fundamental Object Structure

A call to trace a ray from a virtual eyepoint through a pixel begins with an iteration over the object list to find (via *hits*) the object whose ray-object intersection is closest to the virtual eyepoint. The color of that pixel is set to the weighted sum of ambient, diffuse and specular illumination components. The ambient component is a constant unless a procedural texture is in use. The diffuse component is proportional to the cosine of the angle between the surface normal at the intersection point and a vector pointing toward the scene light source. The specular component is computed recursively. The incident ray is reflected about the surface normal, and a recursive call to ray-trace is made with the intersection point as the new virtual eye and the reflected ray as the new ray direction. The returned value from the recursion is the specular component. Pseudo-code for the ray-tracing algorithm may be found in [3]. A sample image, created using only concepts covered in CS2, is shown in Figure 4a.

5. A CS3 PROBLEM SOLUTION

An algorithm used to construct surfaces from point clouds is due to Hoppe, DeRose, Duchamp, McDonald, Stuetzle [6]. The first step in this algorithm is to construct a signed distance function that reports an approximate signed distance from any point in 3D space to the yet-to-be-determined surface. For each data point we locate a small neighborhood of nearby data points and compute their centroid (average). Using principal component analysis (least squares planar fit), we construct a surface normal at that centroid. In general, consistent assignment of normals is a difficult problem, but we avoid it by providing the location of the scanning device that produced the data set. To find the distance to the surface from any point in 3D space, we find the closest centroid and measure signed distance along the associated normal.

The second step in the algorithm is to use the signed distance function to construct a triangulated representation of the surface. Here we use the *marching cubes* algorithm [8]. We scale the data points to fit inside a rectangular, finite, 3D lattice, and then mark each lattice point with its signed (scaled) distance to the surface. The signs on the eight corners of any cube in the lattice completely determine the structure of the surface inside that cube. For example, if all signs are positive or all negative, the surface does not pass through the cube. Although 256 things "can happen", there are only 14 up to symmetry. A best triangulated approximation to the surface, based on the signs at the corners, is selected from among 14, and then that is scaled based on the magnitudes at the corners.

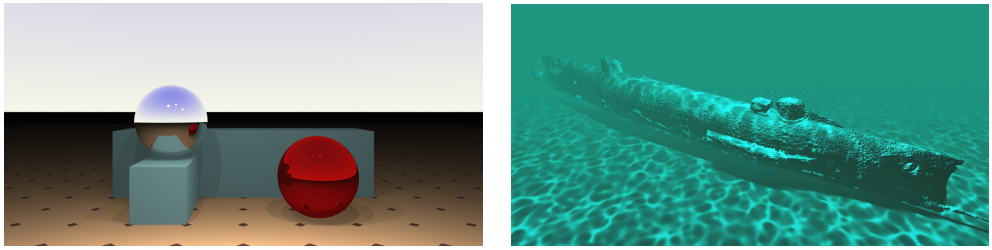


Figure 4: ***CS2/CS3 target images*** **(a) CS2 raytracing (b) CS3 surface reconstruction**

An example surface reconstruction of the recently salvaged Confederate submarine, the H. L. Hunley, is shown in Figure 4b. There are approximately 10 million data points in the scan from which this surface was constructed. This image, created by (former) Ph.D. student Chakrit Watcharopas as part of his dissertation, is considerably more sophisticated than the images of reconstructions that will be available to the CS3 students. Although it is conceivable that CS3 students could use their CS2 ray-tracing systems to achieve sophisticated renderings, it has been our experience that achieving a working solution to surface reconstruction requires interactive rates for viewing results of algorithm updates. Further, we are using computer graphics solely as a vehicle for instruction. It is not the object of that instruction, and so we want to avoid any discussion of hardware rendering and graphics languages, e.g. OpenGL and DirectX. Thus we provide the students with a (binary) interactive viewer of surfaces specified by files written to a simple format.

6. ASSESSMENT

We have previously presented an informal assessment, based on written student comments, of the CS2 course as described above [3]. The principal finding there was a heightened enthusiasm for computing among both students and instructors. Here we present a small, but more formal assessment of the relative merits of our approach to CS1 compared to a more conventional approach that is topic-driven and uses Java as the introductory language. In the Fall semester of 2005, new students were randomly assigned to one of two sections of CS1. One section was taught using the τέχνη approach, the other using the conventional, topic-driven approach. At the end of the course, in their laboratory sections, students answered a series of questions, some of which comprised an attitude survey and others of which comprised a skills test.

In Figure 5 we show results for six questions of interest to the τέχνη project designers. The attitudinal questions were:

- Do you feel the class made you more creative or better able to express you creativity?
- How would rate your knowledge of the {C|Java} programming language?
- Would you recommend CS1 to a friend or colleague?

The skills tests involved reading code, writing code, and understanding logical operators. The code reading question presented students with an implementation of insertion sort and asked for an explanation of what the code was doing. The code writing question asked students to provide code that would search for a specific value in an array. The logical operators question had five expressions composed of boolean variables and logical operators thereon. Each was to be labeled true or false, based on values assigned to the boolean variables.

The population sizes (16 and 20) were not quite large enough to justify t-tests for differences in means, so we used a Monte Carlo version of Fisher's permutation test [11]. We found significant (0.05 or better) differences on all three attitudinal questions and on the reading skills test. Although the mean values were also better for the τέχνη approach on the writing and logic skills tests, we could not declare a significant difference on either of those.

Planning for more careful assessment is underway. Assessment is, in the most general constructivist case, a difficult issue, since the validity of an individual accommodation is difficult to judge with a common measure. Because problem-solving in new contexts is the key development, we often see statements such as, "if learning is in the connections, in the activity itself, then learning is the test," [2] which we find perilously close to circular. Nevertheless, overlays of scaffolding and cognitive apprenticeship offer us some clear directions for the assessment task. We plan to use two approaches. Dynamic assessment [1] repeatedly presents, during the course of instruction, tasks that are just one step beyond existing competence and then provides individual assistance as needed to reach independent mastery. This is based on Vygotsky's zones of proximal development [16]. A zone is defined as the distance between the apprentice's actual development as measured by independent problem solving and his or her potential development as determined by problem solving under the guidance of a master. The measure is thus the amount of additional scaffolding required, beyond that supplied as starting foundation, to achieve an effective solution to a new problem. Consider an example from our CS2 course, where the semester-long problem is ray-tracing. Many instances of ray-object intersection algorithms are required for interesting scenes. The instructor may present a ray-sphere intersection algorithm and then open investigation into ray-box intersection, which is significantly more difficult (to do well), or leave the object to be determined, so that its selection is part of the problem.

An interesting alternative to this direct assessment is available from the study of educational environments. Walker and Fraser [17] observe that numerous studies report a strong correlation between traditional student outcomes (e.g. grades, test scores) and perceptions of classroom environments. The latter can be measured with unobtrusive and time-saving survey instruments. Walker and Fraser use factor analysis on field tests to develop a survey instrument of 34 ratings on six scales, instructor support, student interaction and collaboration, personal relevance, authentic learning, active learning, and student autonomy. Their instrument was ostensibly targeted at distance education, but if we simply omit the six items in the category of student interaction and collaboration, which is consistent with our cognitive constructivist view, we obtain a factorially valid instrument to measure classroom environments of any type. Thus we plan to use the resulting 28-item survey, in conjunction with dynamic assessment to measure the effectiveness of each of the three courses now underway.

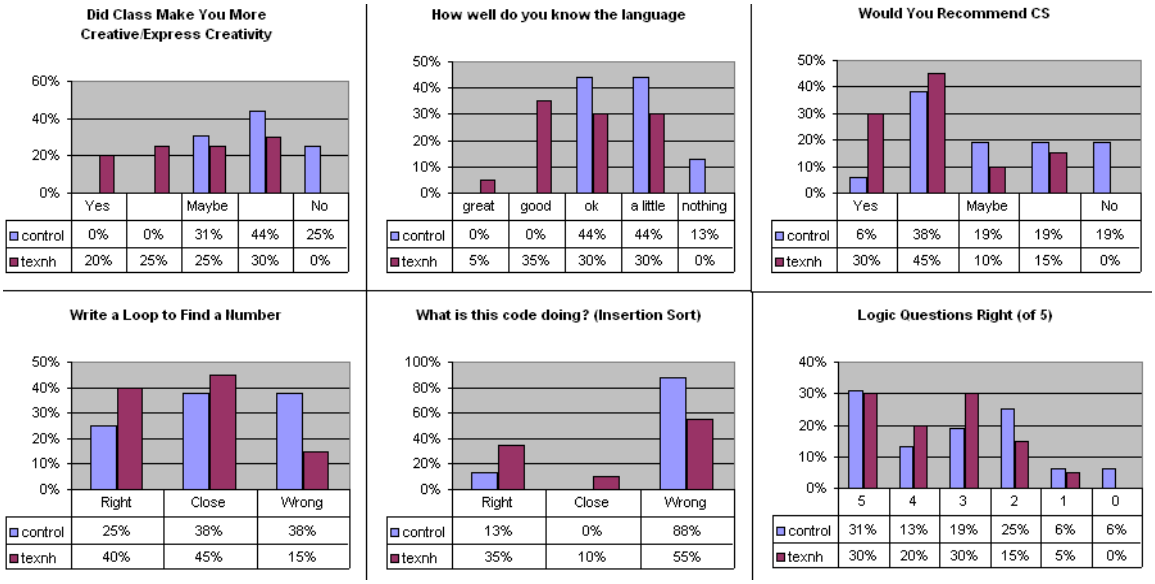


Figure 5: ***Comparative assessment.***

7. CONCLUSIONS

We have described the first three courses in the new, τέχνη curriculum for the undergraduate degree in computer science. All three courses are now being taught at Clemson University. The τέχνη approach relies extensively on problem-based instruction, where the problems are semester-length and selected from the domain of computer graphics. The approach is fundamentally constructivist but relies on an objectivist scaffolding provided by the instructor.

We have provided results from a small, formal assessment of the approach in a CS1 course. Although we find significant improvements in student attitudes, when compared with a conventional, topic-driven, Java-based course, significant improvements in student abilities appear to be limited to reading code for understanding. Plans are in place for more careful assessment of the three courses now underway.

8. ACKNOWLEDGMENTS

This work was supported in part by the CISE Directorate of the U.S. National Science Foundation under award EIA-0305318. The photograph of the Campanile of Trinity College and the frames from *Kneeknocker's Nose* are courtesy of John Kundert-Gibbs.

9. REFERENCES

[1] A. Brown, D. Ash, M. Rutherford, K. Nakagawa, A. Gordon, and J. Campione. Distributed expertise in the classroom. In G. Salomon, editor, *Distributed cognitions*, pages 7:188–228. Cambridge Univ. Press, Cambridge, England, 1993.

[2] D. Cunningham and T. Duffy. Constructivism: Implications for the design and delivery of instruction. In D. Jonassen, editor, *Handbook of Research for Educational Communications and Technology*, pages 170 – 198. Simon & Schuster/Macmillan, 1996.

[3] T. Davis, R. Geist, S. Matzko, and J. Westall. τέχνη: A first step. *ACM SIGCSE Bulletin*, 36(1):125 – 129, 2004.

[4] J. Dewey. *Experience and education*. The Macmillan Company, New York, 1938.

[5] A. Dingle and C. Zander. Assessing the ripple effect of cs1 language choice. *J. of Computing Sciences in Colleges*, 16(2):85–94, May 2001.

[6] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. In *Proc. SIGGRAPH '92*, pages 71–78, 1992.

[7] I. Kant. *Critique of Pure Reason*. St. Martin's Press, New York, 1965.

[8] W. Lorensen and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proc. SIGGRAPH '87*, pages 163–169, 1987.

[9] P. McKenna and B. Laycock. Constructivist or instructivist: Pedagogical concepts practically applied to a computer learning environment. In *Proc. ITICSE '04*, pages 166–170, Leeds, UK, June 2004.

[10] J. Piaget. *The development of thought: equilibration of cognitive structures (translated by A. Rosin)*. Viking Press, New York, 1977.

[11] R. A. Fisher. *The Design of Experiments*. Hafner Publishing Company, New York, 8th edition, 1966.

[12] E. Reinhard, M. Ashikhmin, B. Gooch, and P. Shirley. Color transfer between images. *IEEE CG & A*, September/October:34–41, 2001.

[13] L. B. Resnick. Learning in school and out. *Educational Researcher*, 16(9):13–20, 1987.

[14] J.-J. Rousseau. *Émile (translated by B. Foxley)*. Dutton, New York, 1955.

[15] J. Spolsky. Advice for computer science college students. http://www.joelonsoftware.com/, January 2005.

[16] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard Univ. Press, Cambridge, MA, 1978.

[17] S. Walker and B. Fraser. Development and validation of an instrument for assessing distance education learning environments in higher education. *Learning Environments Research*, 8:289–308, 2005.