τέχνη: A First Step

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

A new approach to the design of the computing curriculum for a Bachelor of Arts degree is described. The approach relies extensively on problem-based instruction and computer graphics. The novelty arises from the magnitude and origin of the problems to be integrated into the curriculum and the breadth of the impact across the curriculum. Results from a trial course, the first experiment with the new approach, are described. The course, Tools and Techniques for Software Development, is a sophomore-level course in programming methodology. Construction of a ray-tracing system (for generating synthetic images) was the vehicle chosen for the instruction.

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*. It is unfortunate that this strong connection between the two, established in the language of those who took a foundational role in both, has been largely dismissed by the U.S. educational system. Today undergraduate curricula in computing are laden with computing-specific requirements that allow little time for exploration of disciplines outside science, let alone the connections between each of those disciplines and computing.

In 1999, Clemson University established a (graduate) degree program that bridges the arts and the sciences. The Master of Fine Arts in Digital Production Arts is a two-year 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 graduate level work in both the arts and computer science. Although the DPA program was not begun until 1999, graduates have already found employment in many of the top studios, e.g., Industrial Light & Magic (Lucasfilms), Rhythm & Hues, BlueSky Studios, Tippett Studios, and Pixar. The program has also effected a significant change in the Clemson undergraduate degree program enrollment. The faculty has witnessed a substantial shift of undergraduate majors from the B.S. degree in Computer Science to the B.A. degree in Computer Science with an elected minor in Art. Whether these undergraduate students ultimately pursue the DPA Program or not, it is our position that this shift to a more balanced educational experience is of substantial benefit to the students and to society.

The goal of the τέχνη project is a zero-based re-design of the B.A. degree program in computer science. The overriding directive of the new design is the direct incorporation of DPA and computer graphics research results into all required computing courses in the curriculum. Instruction in these (new) courses will be strictly oriented toward large-scale problem-solving, where the large-scale problems are those that have arisen naturally in the research investigations of the computer graphics faculty. Problem-based instruction is not new (See [5] for a comprehensive treatment.), nor is the suggestion that computer graphics may be an ideal vehicle for such. Cunningham [3] recounts tools of thought for problem-solving identified by Root-Bernstein [13] and then effectively argues that these aspects of problem-solving, as well as associated communication skills, are remarkably well supported by computer graphics. What is new here is the magnitude and origin of the problems to be integrated and the broad impact of the approach across an entire curriculum.

Any zero-based design must begin with an identification of goals, in this case, fundamental concepts and abilities to be imparted to and developed in the undergraduate student of computer science. In this endeavor, we must 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.

We recognize the extensive efforts in curriculum design and evaluation that are conducted annually by the Computing Accreditation Commission (CAC). The B.S. program in Computer Science at Clemson was among the first 35 programs to receive accreditation from CAC (then CSAC) in June, 1986. The Curriculum Standards

SIGCSE 2004 Norfolk, Virginia USA

document of CAC will be kept in hand during this design process. Nevertheless, we must also note that the B.A. degree in Computer Science at Clemson was born from a general faculty sentiment that the Curriculum Standards may be overly prescriptive for many of our students. Recent discussion of intentional learning [11] would support this view. Although the newer Curriculum Standards are decidedly less prescriptive than the original [8], we do not want to commit, a priori, to meeting standards such as, "IV-1. The curriculum must include at least 40 semester hours of up-to-date study in computer science topics." The "up-to-date" would not trouble us, but the "40 semester hours" might. Our approach is certainly constructivist in nature [2], and one might argue, as do Kirkley and Duffy [9], that such an approach could significantly impact intention. If so, it could subvert our goal of flexibility in design. Nevertheless, the interaction between constructivism and intention is not well-established, whereas the overall benefits of cooperative learning [7] and a view of learners as constructors of knowledge [1] are well-accepted and hence direct our approach.

Further, the collection of key concepts we have tentatively identified shows considerable overlap with the Curriculum Standards item IV-6, but it also show considerable differences. Our tentative collection comprises:

- a machine model (imperative programming, machine capabilities, machine limits)
- a connection model (networks for communication and distributed processing)
- software design (the object-oriented paradigm, large-scale development, testing)
- windowing and operating systems (resource management, protection levels, security)
- data structures and performance (performance measurement, bottleneck identification, work-flow management)
- cross-platform computing (PDAs, embedded systems, cross-compilation, external device control)

We stress that this is a starting point, and we recognize that it may undergo substantial revision during the formative evaluation process.

A trial course in this new program has now been completed, and it is our purpose here to discuss the results from that offering. The trial course, with key concept *software design*, is CPSC 215, Tools and Techniques for Software Development. The intent is instruction in programming methodology using C and C++. Construction of a ray-tracing system for rendering synthetic images is the large-scale problem upon which the course is based.

2. COURSE DESIGN

Ray-tracing is a technique for synthesizing images by following hypothetical photon paths [4, 6, 12]. A ray-tracing system models a virtual viewer looking upon a collection of geometrically specified objects in Euclidean three-space. A virtual rectangular viewing screen is interposed between the viewer and the objects. The screen is oriented so that a vector normal to the screen and based at the center of the screen passes through the eyepoint of the viewer. The screen is considered a two-dimensional lattice of equally spaced points representing pixels or sub-pixels of the final projected image. The ray-tracing algorithm creates the image by firing a virtual photon from the viewpoint through each lattice point. If the photon hits an object it bounces, and it may proceed to hit additional objects. The color assigned to the lattice point is a weighted sum of the colors of all objects hit by the photon. Many commercial rendering systems for special effects are based upon ray-tracing [10].

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 can be 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 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.

2.1 Phase I - Fundamentals

Students entering CPSC 215 typically have completed the CS I and II courses but have had little or no exposure to the C language or basic concepts of computer graphics. Three weeks of the fifteen-week course were devoted to fundamentals of the C language, the standard library, and their use in the representation, storage, and retrieval of image data. These topics were introduced in the context of several assigned 2D image transformation problems including: converting color images to grayscale; digital halftoning (converting grayscale to black-white); "colorizing" grayscale movie frames; and reformatting standard television images to display on High Definition monitors.

2.2 Phase II - Ray-tracing Structure

In the next five weeks of the class the ray-tracing problem was described and the key elements required in its solution were introduced. These elements included the use of structures, unions, pointers, and recursion. A breadth-first approach with repeated refinements was employed. In this way the students were quickly able to render simple images and then refine them using more sophisticated treatments. Key to success was a carefully defined structure to represent the "objects" in the scene. A typical example is shown in Figure 1. Note the extensive use of function pointers. A *color* is

```
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 1: Fundamental Object Structure

specified as a triple of red, green and blue (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 ob-

ject'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. In Phase II, only two types of objects, a sphere, here named *ball*, and an infinite horizontal plane, here named *floor*, were defined, and colors were limited to grayscale (R = G = B).

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 is shown in figure 2.

```
color_t raytrace (ray_t ray, float ray_depth)
{
if (ray_depth > max_depth) return(black);
best_distance = +∞;
for (each object in the scene){
   compute ray-object intersection point, pt;
   if (distance(pt,viewpoint)< best_distance){
      record object and pt;
      update best_distance;
      }
   }
if (no object intersected) return(background color);
add best_distance to ray_depth;
set color to ambient color for this object;
get_normal for this object at pt;
for (each light in the scene){
   if (pt not in shadow){
      compute diffuse component for this pt/light;
      add diffuse color to color;
      }
   }
if (object has a specular component){
   compute reflected ray;
   reflected_color = raytrace (reflected ray, ray_depth);
   add reflected_color to color;
   }
return(color);
}
```

Figure 2: Pseudo-code for ray-tracing

Requirements for the first ray-tracing project were flexible:

1. Scene geometry must include at least one light source, two spheres, a checkerboard planar surface, and a sky.
2. The image should illustrate shadows, diffuse and specular illumination, and anti-aliasing through sub-pixel sampling.
3. The image should have aspect ratio 4:3, with no ratio-induced distortions, and it should be at least 1024x768 pixels.

In spite of the limited tools available to them, the students showed an impressive ability in code design and an impressive creativity in scene design. In Figure 3 (a) and (b) we show images from two of the students.

2.3 Phase III - Ray-tracing Refinements

In the next four weeks of the course, the design of the ray-tracing system was extended to include refraction, stereographic projection, and new object types. New types included boxes, quadrics, and surfaces of revolution. Algorithms for the defining functions, *hits()* and *get_norm()*, were derived in class, but the implementation was left to the students. These additions to the task naturally generated discussions of new tools and techniques including dynamic memory allocation, non-trivial linking, and modular program design. Rather than a burdensome extension, students found "makefiles" to offer a welcome relief. Requirements for the second project were again brief and flexible:

1. Scene geometry must include at least 3 light sources, 2 boxes, 2 spheres, a planar surface, and a sky (unless the scene is completely enclosed by a sphere or box).
2. The image should illustrate shadows, diffuse and specular illumination, and anti-aliasing through sub-pixel sampling.
3. The image should be in color and should have an aspect ratio of 16:9 (HDTV ratio), with no ratio-induced distortions, and it should be at least 1024x576 pixels.

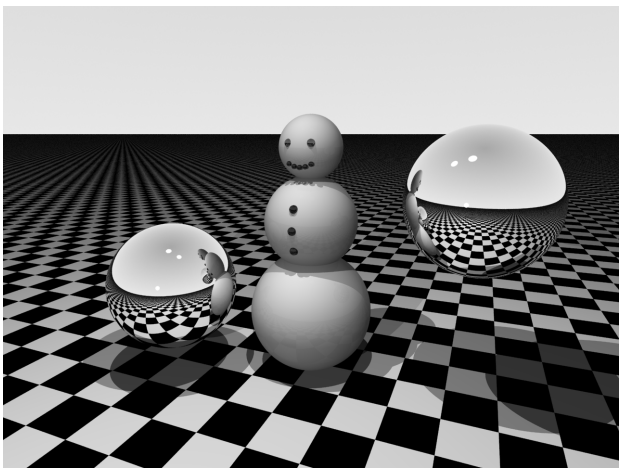
2.4 Phase IV - Scene Specification Language

To permit complex scenes and animations to be defined using external, file-based specifications, a week was spent on additional I/O techniques and the design of an integrated parser. The Scene Specification Language suggested was a highly simplified synthesis of several current formats [14]. Students were free to extend the language, and several did. In the final two weeks, source code *simplifications* available through use of C++ were discussed. Because OO is a design paradigm, not a language, and because the students had a large OO design in front of them, the transition was not viewed as a major one. In particular, simplifications available in vector operations (spatial and color) through operator overloading and the advantages of derived classes were easily described. The use of C++ for the final ray-tracing project was optional, and several took that option. In figure 3 (c) - (f) we show final images from two classes.

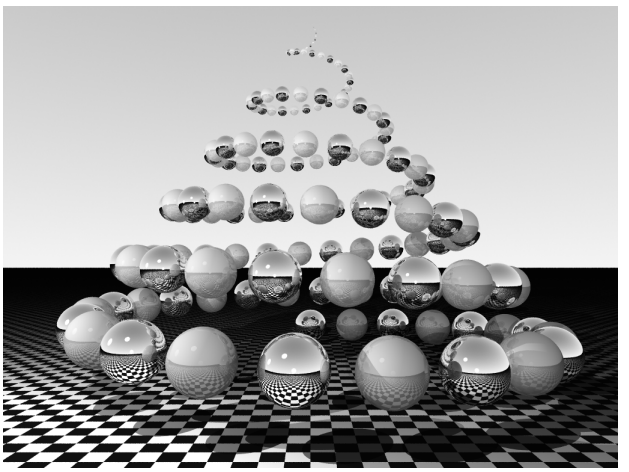
3. DISCUSSION

Throughout all phases of the ray-tracing project, but especially in the final phase, students were encouraged to be creative in scene design. As noted earlier, the images in 3 (a) and (b) represent work from the first phase of the ray tracer. Considering that the only geometries known to the students at this stage were the sphere and the infinite plane, these images show an impressive capability, which can probably be attributed to the students' heightened level of interest. The remaining images show mastery of additional features, such as reflection and anti-aliasing, as well as optional features, such as quadrics and textures.

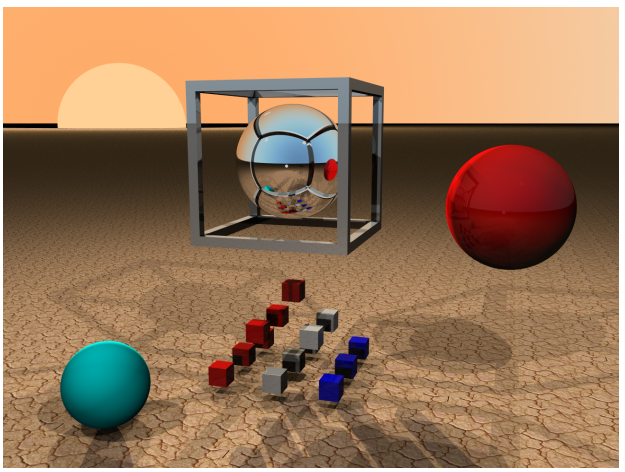
As evidenced by anonymous semester-end evaluations, students responded positively to learning C/C++ through graphics. Many students felt that the semester-long project was educational and interesting to implement. They especially seemed to appreciate the visual feedback from their projects, both for aesthetic and problem determination purposes. Corroborating evidence is supplied by the near absence of student decisions to drop the course, which is unusual for these classes. Many students brought their laptops (Clemson requirement) to class to discuss (or show off) the previous night's rendering successes and failures.



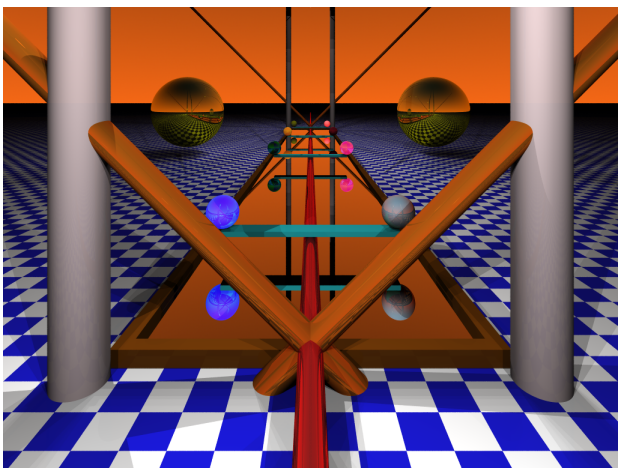
(a) Image by undergraduate student S. Duckworth.



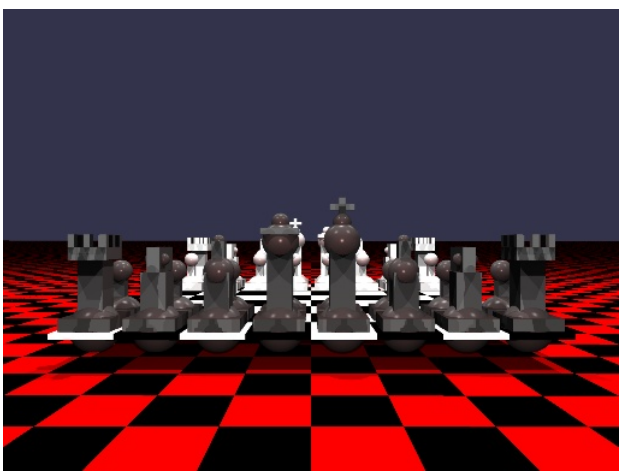
(b) Image by undergraduate student T. Nguyen.



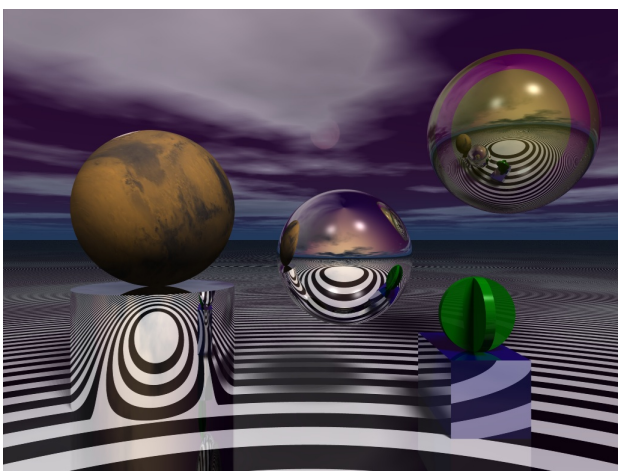
(c) Image by undergraduate student S. Duckworth.



(d) Image by undergraduate student T. Nguyen.



(e) Image by undergraduate student J. Holcombe.



(f) Image by undergraduate student S. Haroz.

Figure 3: CPSC 215 Example Student Renderings.

The following are sample excerpts from anonymous student evaluations of the course:

> The ray tracer project was good because it gave visual feedback of your accomplishments and impressive results. I liked that we continued with several versions of the project leading to a large and useful program in the end.
>
> The ray tracer is a much better assignment than the usual use-once throw away programs we develop.
>
> The ray tracing project was great. It provided practical usage to learning C rather than just making a useless program that 'implements a linked list or binary tree.' The ray tracer also gave me a much stronger knowledge of C than [other courses] did with Java.
>
> It is the first class where I wrote a program that I will not throw away at the end of the semester.
>
> The class wasn't just like some ordinary class. We got to do something fun and different.
>
> Making a ray tracer is so much cooler than making a card game.

Other areas of comment, but on the negative side, involved the amount of work required toward the end of the semester. We are attempting to address this problem, and accrue some additional learning advantages, by changing the third phase of the ray tracing project to a cooperative venture. The τέχνη plan calls for specific instructional procedure as well as content. Cooperative learning in attacking the large-scale problems will be essential. Of course, even the strongest advocates of cooperative learning [7, 15] will admit to drawbacks, and we acknowledge that certain hooks to capture the positive aspects of competitive learning will be necessary. In team programming projects, weaker students will occasionally "hide" in large, strong teams and fail to engage in the projects. Offsetting competitive hooks, such as a bonuses on test scores if all team members exceed given levels, can often ameliorate such difficulties, and we plan to pursue this approach.

While increased motivation was an expected result from the new course, a surprising aspect was the extent of extra work the students actually performed, in many cases more than our graduate students who also write a ray-tracing program for an advanced graphics course. Several of the undergraduates investigated advanced techniques, such as new object geometries, texturing, and 3D stereograms, on their own. These features were far beyond the requirements for the assigned projects.

4. CONCLUSIONS

Overall, the results from our experimental course have been encouraging. Based on the work performed and student evaluations, we feel the projects are more effectively engaging the students in learning.

A benefit of the τέχνη project is the opportunity for faculty to engage undergraduates with discussions of the research that carries their enthusiasm. We conjecture that the benefits to the students will arise not only from the problem-solving orientation of the instruction and the exposure to the vitality of real research problems but also from a newly induced vitality in the instructors.

Finally, an interesting and initially unexpected result of the (graduate) DPA program has been the demographic of the students enrolled. The problem of under-representation of women and minorities in computing programs is well known and, nation-wide, shows little sign of amelioration. The DPA program has a current enrollment that is 32% women and 16% African American, both well above the averages for more conventional graduate programs in computing, including those at Clemson. A natural conjecture is that a DPA-based re-design of the B.A. program will effect similar enrollment shifts, and the new curriculum may merit widespread adoption on this basis as well as on the bases of enhanced problem-solving skills of the students and enhanced enthusiasm of all participants.

5. ACKNOWLEDGMENTS

This work was supported in part by the CISE Directorate of the U.S. National Science Foundation under award EIA-0305318 and the ITR Program of the National Science Foundation under award ACI-0113139.

6. REFERENCES

[1] D. Cunningham. Assessing constructions and constructing assessments: A dialogue. *Educational Technology*, 31(5):13 – 17, 1991.

[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] S. Cunningham. Graphical problem solving and visual communication in the beginning computer graphics course. *ACM SIGCSE Bulletin*, 34(1):181 – 185, 2002.

[4] T. A. Davis and E. W. Davis. Exploiting frame coherence with the temporal depth buffer in a distributed computing environment. In *Proc. IEEE Parallel Visualization and Graphics Symposium*, San Francisco, CA, October 1999.

[5] B. Duch, S. Gron, and D. Allen. *The Power of Problem-Based Learning*. Stylus Publishing, LLC, Sterling, VA, 2001.

[6] A. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.

[7] D. Johnson and R. Johnson. *Learning Together and Alone*. Allyn and Bacon, Needham Heights, MA, 5 edition, 1999.

[8] L. G. Jones and A. L. Price. Changes in computer science accreditation. *Communications of the ACM*, 45(8):99 – 103, 2002.

[9] J. Kirkley and T. Duffy. Expanding beyond a cognitivist framework: A commentary on martinez's 'intentional learning in an intentional world'. *ACM J. on Computer Documentation*, 24(1):21 – 24, 2000.

[10] J. Kundert-Gibbs. *Maya: Secrets of the Pros*. SYBEX, Almeda, CA, 2002.

[11] M. Martinez. Intentional learning in an intentional world. *ACM J. on Computer Documentation*, 24(1):3 – 20, 2000.

[12] F. Musgrave. Grid tracing: Fast ray tracing for height fields. Technical Report RR-639, Yale University, Dept. of Comp. Sci., July 1988.

[13] R. S. Root-Bernstein. Tools for thought: Designing an integrated curriculum for lifelong learners. *Roeper Review*, 10:17 – 21, 1987.

[14] K. Rule. *3D Graphics File Formats: A Programmer's Reference*. Addison-Wesley, Reading, MA, October 1996.

[15] L. Williams and R. Upchurch. In support of student pair-programming. *ACM SIGCSE Bulletin*, 33(1):327 – 331, 2001.