Graphics-Based Learning in First-Year Computer Science

T. A. Davis

Department of Computer Science, Clemson University, Clemson, SC, U.S.A.

Abstract

This paper proposes a method for teaching a first-year course in computer science using graphics-based problems as the teaching medium. Specifically, we present a method for instruction in programming using a semester-long project of developing a ray tracer. This effort is part of a larger project, known as τέχνη, in which a broad range of undergraduate courses are taught using computer graphics as the motivating application. An overview of this project is provided, along with description and results from the first trial CS2 course instructed using this technique.

Categories and Subject Descriptors (according to ACM CCS): K.3.2 [Computers and Education]: Curriculum.

1. Introduction

Engaging students in the study of computer science is becoming increasingly important as enrollments in CS degree programs continue to decline. One method of meeting this growing need involves the offering of practical and interesting problems that ignite student attention and encourage continued study. Selecting the right type of problem must therefore hold some sort of relevance and intrinsic interest to students. We submit that computer graphics fulfills these criteria and provides an ideal problem-based learning platform upon which a variety of topics can be taught effectively.

The proliferation of visual media is a worldwide phenomenon, fueled by the demands of an ever expanding visually-based population. The visual entertainment industry, which includes film, television, and computer gaming, can be found in all parts of the world and generates tens of billions of dollars per year. As a result, college students typically have experienced broad exposure to computer-generated images and often have a strong interest in their creation and use. Leveraging this interest, we have developed a new approach, termed τέχνη (or TEXNH), to teach general computer science concepts through computer graphics problem-based instruction in the undergraduate curriculum.

Computer graphics provides a natural application area for teaching concepts in computer science due to the complexity of the problems, which often require a wide variety of programming constructs, as well as a substantial level of mathematic calculations. Further, through graphics programming projects, students can evaluate the correctness of solutions and discover problem areas quickly through visual inspection. Finally, such projects allow for a certain degree of artistic freedom in terms of algorithm design as well as scene content, which is often not found in other projects.

While the goal of τέχνη is to span all courses in the undergraduate curriculum, our focus is CS2, a first-year second semester course. Specifically, we describe how the course is organized to incorporate graphics projects for continuing freshmen. We begin by further describing the τέχνη project and our experiences to this point in Section 2. We continue in Section 3 by characterizing related work describing similar approaches. Section 4 details the structure of the course, including class projects, while Section 5 concludes and presents student results.

2. τέχνη Project Overview

The name of our approach, τέχνη, is the Greek word for art, and shares its root with τέχνgoloία, the Greek word for technology. The similarity of these two words reveals the close academic relationship the two fields have held historically. Unfortunately, this relationship

has been weakened over the years; one of the primary goals of the τέχνη project is to reunite these two areas to create a better and more effective environment for learning.

Inspiration for the τέχνη project arose from the creation of a separate, but somewhat related, program at Clemson: Digital Production Arts (DPA). This master's level program, begun in 1999 at Clemson University, combines elements of computer science, art, and theater, among other fields, in its curriculum. Graduates who complete the DPA program pursue careers in the ever-expanding special effects industry, which focuses on film, television, and gaming. Our graduates now work in a variety of studios, including Rhythm & Hues, Industrial Light & Magic, Electronic Arts, Pixar, Sony Imageworks, and Blue Sky. While DPA is currently a graduate-only program, undergraduates at Clemson have shown interest in the program as early as freshman year. Some have made adjustments in their undergraduate majors/minors, as well as courses, to better prepare them for applying to the program. Overall, the number and quality of applicants from other universities are also on an upward trend.

Form our experiences with the DPA program, we sought opportunities for duplicating our successes in the undergraduate computer science curriculum. The result was the τέχνη project, in which the overriding goal is to incorporate projects and research in the undergraduate computer science curriculum to teach basic computer science topics. Semester-long graphics projects in required courses form the basis of a problem-based learning curriculum that promotes the development of programming skills as well as creativity. We believe this approach could be quite effective since it encompasses several important education-theoretical techniques, including: visual feedback, problem-based learning [DGA01] [Cun02], intentional learning [Mar97], and constructivism [Mor98]. By incorporating these key components, we seek to create a focused, problem-based learning environment that educates, motivates, and broadens students at all levels.

Our first trial course under the τέχνη approach was a sophomore-level course in C/C++/Unix (CPSC 215 – Tools and Techniques for Software Development). Students entering this course had taken CS1, CS2, and data structures with Java. The projects for the course included image processing and the implementation of a ray tracer. The course has been offered several times with excellent results. Students evaluate the course highly, with visual feedback and the ability to show off impressive results to family and friends as strong positives. Additional details may be found in [DGMW04].

After the success of τέχνη in this sophomore-level course, we began to broaden the application of the τέχνη approach to additional courses, beginning with CS1. In this course, students programmed solutions for various image processing problems, including blurring and sharpening images, converting color images to grayscale images, and applying the color scheme of one image to the color palette of another. This final project was based on a paper by Reinhard, et al [RAG01] wherein first-semester(!) students were required to implement a non-trivial research result. Further explanation can be found in [MD06-1] and [MD06-2]. We are now applying this approach to the second-semester course, with plans to extend its usage to all freshman and sophomore courses beginning Fall 2006.

3. Related Work

Freshmen entering college typically have been exposed to computer-generated imaging in a variety of formats. Most work thus far, however, has focused on teaching students using image processing. Allowing students to explore image processing topics is becoming popular, due primarily to the interest generated by such projects and the principles reinforced. Working with such projects is interesting to a wide range of students, from elementary school [MV05] to college [WN05] [AR98] [Bur03] [FP97] [Hun03] since students react positively to visual results and working on real-world problems.

In teaching students about programming, image processing and rendering projects naturally require students to deal with dynamic memory allocation and two-dimensional arrays [Bur03]. Additionally, due to the sheer volume of pixel data (a standard image contains hundreds of thousands of pixels), students are required to produce generalized algorithms for pixel calculations; "hard coding" solutions is not tractable [MV05]. In addition to these important concepts, using a problem-based learning model further highlights the necessity of complex programming techniques. And computer graphics has been shown to be an effective tool in teaching computer science in a problem-based approach [Cun02].

Previous work involving rendering techniques or image processing in CS1 has been performed on a limited scale. Some approaches have provided ready-made GUI environments [AR98], code for function prototypes [WN05], and pre-written functions [AR98] [Bur03] [FP97] [Hun03] to handle various types of image manipulation. Also, image processing is but just one of several projects used in courses under these approaches. Our approach for CS2 under τέχνη is unique in at least two ways. First, instead of using image processing, we require students to build a rendering system (ray tracer) as a semester-long project. Second, students are required to develop all rendering, parsing, manipulation, and output code without any specialized help, such as a GUI environment or starter code.

4. Course Structure

As mentioned previously, students entering this course had programmed various image manipulation projects in CS1 the previous semester. As a result, they had been introduced to the C language and had experience with introductory programming concepts. Specifically, they understood how to output image data in a simple graphics file format, *ppm*, using redirected I/O from stdout. Since some of the later projects required image processing, students also learned how to store two-dimensional image data in one-dimensional arrays using index arithmetic. Students therefore were not faced with a steep learning curve to implement the semester-long project in CS2, a ray tracer.

The implementation of a ray tracer system is an ideal pedagogical mechanism for problem-based learning in our course for several reasons. First, the implementation of a ray tracer covers a broad range of topics, all of which must be mastered in order to develop a full-featured system. Second, the organization of the underlying design of the code naturally leads to the object-oriented paradigm. Finally, and perhaps most importantly, the system provides visual feedback in the form of images throughout its development, allowing students to determine quickly the proper functioning of their programs, and to express their creativity through scene content.

While we have offered a sophomore-level course using ray tracing as the semester-long project for students who knew how to program but were just learning C, this course represents our first experience in bringing the ray tracer to the freshman level. Indeed, the literature contains few, if any, teaching experiences that have brought such advanced-level graphics to a first-year course. Here, our focus is to teach programming concepts to students for the first time using a graphics-based approach.

4.1 Project 1 – Basic Ray Tracer

The basic ray tracer required students to construct a program with basic capabilities that provided a basis for future enhancements. Specifically, the program produced a single red sphere with ambient, diffuse, and specular lighting contributions. The target image is shown in Figure 1.

This project afforded several opportunities for teaching programming concepts. First, continuing from their experiences in the preceding semester, students gained further experience in using non-trivial operators, including trigonometric functions, to compute ray/sphere intersections and lighting contributions. Additionally, reviewing mathematical concepts involving vectors and their associated operations (e.g., dot product), plunged students into mathematics from the start, which was useful not only for this course, but most probably will be for future computer science courses as well.

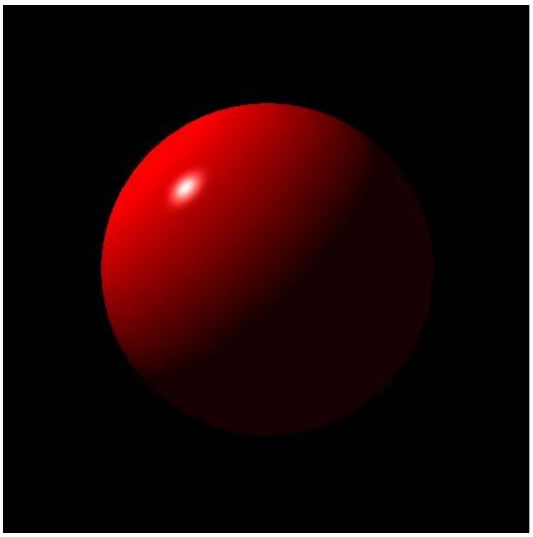

Figure 1: *Project 1 result*

Students also learned more about programming with functions, as they found vector operations, such as normalize, add, subtract, and dot product, used so widely that they warranted functions simply to reduce typing. These kinds of discoveries often produce longer-term retention since students must invent their own solutions to self-imposed problems. Classroom discussion of functions and their parameters allowed additional review of pointers for variable parameters. As pointers are one of the most difficult concepts for students to grasp, any opportunity for additional discussion is welcome.

Another requirement of this project, and projects following, was the use of header files. These files naturally led to a discussion of preprocessor directives, such as **`#define`** and **`#ifndef`**, to prohibit multiple header file inclusion. More important, however, was the instruction on user-defined types.

Although CS1 introduced students to the **`typedef`** concept, they had no experience in constructing complex data types, e.g., structures within structures. Geometric primitives, such as spheres, within the ray tracer provided an ideal platform upon which to build complex data. Figure 2 shows an object type. Note that we have basic data types, including **`POINT_T`**, **`COLOR_T`**, and **`SPHERE_T`**, and that **`SPHERE_T`** includes fields of the other types. At this point, students were just beginning to understand how to construct more complex data, which was further developed in project 2.

```
typedef struct {
  double red, green, blue;
} COLOR_T;

typedef struct {
  double x, y, z;
} POINT_T;

typedef struct {
  POINT_T center;
  double radius;
  COLOR_T color;
} SPHERE_T;
```

Figure 2: *Project 1 data types*

4.2 Project 2 – Intermediate Ray Tracer

With the introduction of multiple objects, including a new plane primitive, project 2 required students to extend their data types from project 1. First, to handle multiple primitives and/or multiple lights, the students were required to become familiar with arrays of structures. As each object needed only the fields associated with its own geometry type, unions were introduced, along with a more in-depth discussion of machine memory. The new object data structure is shown in Figure 3.

Further, the concept of function pointers was introduced to ease the coding of primitive-specific implementation, such as intersection testing. That is, when a ray is tested for intersection with all objects in the scene, it must undergo an intersection test with each primitive type (e.g., sphere or plane). Since each of these intersection test functions are differently coded according to geometry, the program would normally be required to know the primitive type (often implemented with a switch statement) in order to call the appropriate function. Function pointers allow the correct function to be registered at the start of the program, such that individual intersection tests could be called in the same way for all primitives.

```
typedef struct object {
  COLOR_T color;

  union {
    SPHERE_T sphere;
    PLANE_T plane;
  } geometry;

  int (*intersect)(struct object *obj,
       RAY_T ray);
} OBJECT_T;
```

Figure 3: *Project 2 data types*

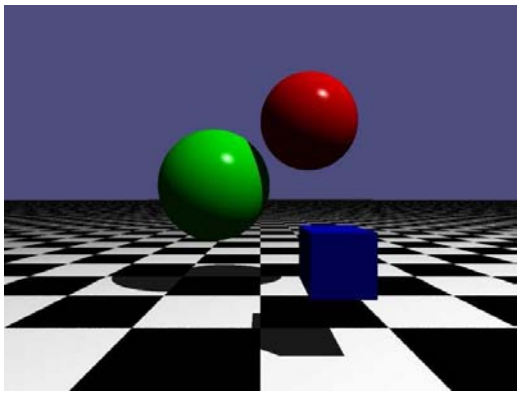

Figure 4: *Project 2 result*

The target scene for project 2 is shown in Figure 4. Note that the plane is procedurally textured with a checkerboard pattern. While not only adding interest to the project, the checkerboard texture provided an opportunity to introduce binary operations. The checkerboard pattern is a three-dimensional procedural texture that colors alternating unit cubes with the same color. Accordingly, the checkerboard color of the point *(x, y, z)* on the plane is given by the expression in Figure 5.

Two additional topics rounded out project 2: two-dimensional arrays and file output. The former required that students explore multidimensional arrays to store image data and perform antialiasing in the form of supersampling the image. As each element represented an RGB color, the array promoted additional practice with complex data. The latter requirement involved transitioning the ray tracer from redirecting `stdout` to the proper use of I/O (**`FILE`**) data operations to save the *ppm* image. This topic also included file input, which was part of project 3.

4.3 Project 3 – Advanced Ray Tracer

At this point, student ray tracers were ready to handle a variety of scenes, not only a single hard-coded environment. The obvious way to allow this flexibility was to create a simple scene description language that the ray tracer could read, and then render the corresponding image. This feature allowed students not only additional experience with file operations, but also with string

```
if ((floor(x) + floor(y) + floor(z)) && 1)
  color = white

else
  color = black
```

Figure 5: *Checkerboard expression*

processing, as tokens in the file were represented as strings. Another possible avenue for file input included reading in *ppm* files for texture mapping; however, we did not pursue this option due to time constraints.

As the number of objects read from a file was arbitrary, students were naturally confronted with the problem of dynamic storage. In dealing with this issue, dynamic memory allocation and linked lists were covered in class. Students implemented solutions using *malloc* and pointers with the solid understanding of exactly why they were doing so!

Finally, recursion was introduced to handle reflection in ray tracing. Since reflected rays are subjected to the same set of intersection tests as a first-level ray, the algorithm is naturally recursive. Using the ray depth as a stopping condition is also intuitive as students understand the nature of ray tracing at this stage.

4.4 Project 4 – C++ Ray Tracer

At this point in the course, we transitioned from C to C++, discussing both the procedural and object-oriented paradigms. Although we had extremely limited time left in the semester, we were able to proceed quickly due to the foundations laid in earlier parts of the course.

For instance, students were able to pick up the concept of a class quickly since our graphics primitives correspond directly to C++ objects (e.g., sphere). Since we had required that such objects be place in their own header/source files, the "objects" were already in place. Further, the notion of function pointers, used earlier to call the primitive's intersection test, provided a graceful transition into member functions.

Finally, the user-defined **`vector`** type, provided a good opportunity for demonstrating a tight class with private data *(x, y, z)* and a variety of public operations (e.g., **`add`**, **`subtract`**, **`normalize`**, etc.). These operations further led naturally to operator overloading (e.g., **`v1 + v2`**), which was viewed as a greatly simplified way to express required operations.

Students were only required to convert project 1 (Basic Ray Tracer) to C++; however, some students chose to go further.

5. Conclusion and Future Work

Given the relatively small number of graphics primitives available in the ray tracer, students produced a surprisingly creative set of images (see example images in Figure 6). As evidenced by course evaluations, quality of work, and enthusiasm, students appeared to be much more engaged in the course than in previous semesters. One advantage typically cited was the visual feedback provided by the ray-traced images. Also, students tended to prefer working on a project they could use or show off after the course ended, over more traditional projects. Students also routinely went above and beyond the requirements of the projects to produce more advanced effects (such as 3D red/blue images).

Although overall the course ran smoothly, we do recognize some areas for improvement in future semesters. First of all, although part of our goal was for students to learn to program without a great deal of supplied starter code, some students fell behind and were not able to complete the final version of the ray tracer. Perhaps intermediate baseline code could be made available in future offerings of the course. Also, we would like to improve the introduction of the ray tracer to the students, such that the learning curve can be relaxed.

During the trial period, we offered two sections of CS2 (approximately 20 students in each section) per year: one using the τέχνη approach and one without. Students were originally placed in the τέχνη program randomly, but once they began with the approach, they had to continue with it through their second years. In terms of instructors, most τέχνη courses thus far have been taught by professors and graduate students with backgrounds in computer graphics. We have already begun to indoctrinate non-graphics faculty. To ease this process, we work closely with them on their first preparation and provide extensive web-based resources as well. Our goal is that any faculty member would be able to teach a τέχνη course without significant problems.

Acknowledgments

This work was supported by the CISE Directorate of the U.S. National Science Foundation under award EIA-035318.

References

[AR98] ASTRACHAN, O., AND RODGER, S. H. Animation, visualization, and interaction in CS1 assignments. *SIGCSE Bulletin*, 30(1), 1998, 317-321.

[Bur03] BURGER, K. R. Teaching two-dimensional array concepts in Java with image processing examples. *SIGCSE Bulletin*, 35(1), 2003, 205-209.

[Cun02] CUNNINGHAM S.: Graphical problem solving and visual communication in the beginning computer graphics course, *ACM SIGCSE Bulletin*, 34(1), 2002, pp. 181-185.

[DGMW04] DAVIS T., GEIST R., MATZKO S., WESTALL J.: τέχνη: A first step, *ACM SIGCSE Bulletin*, 36(1), 2004, pp. 125-129.

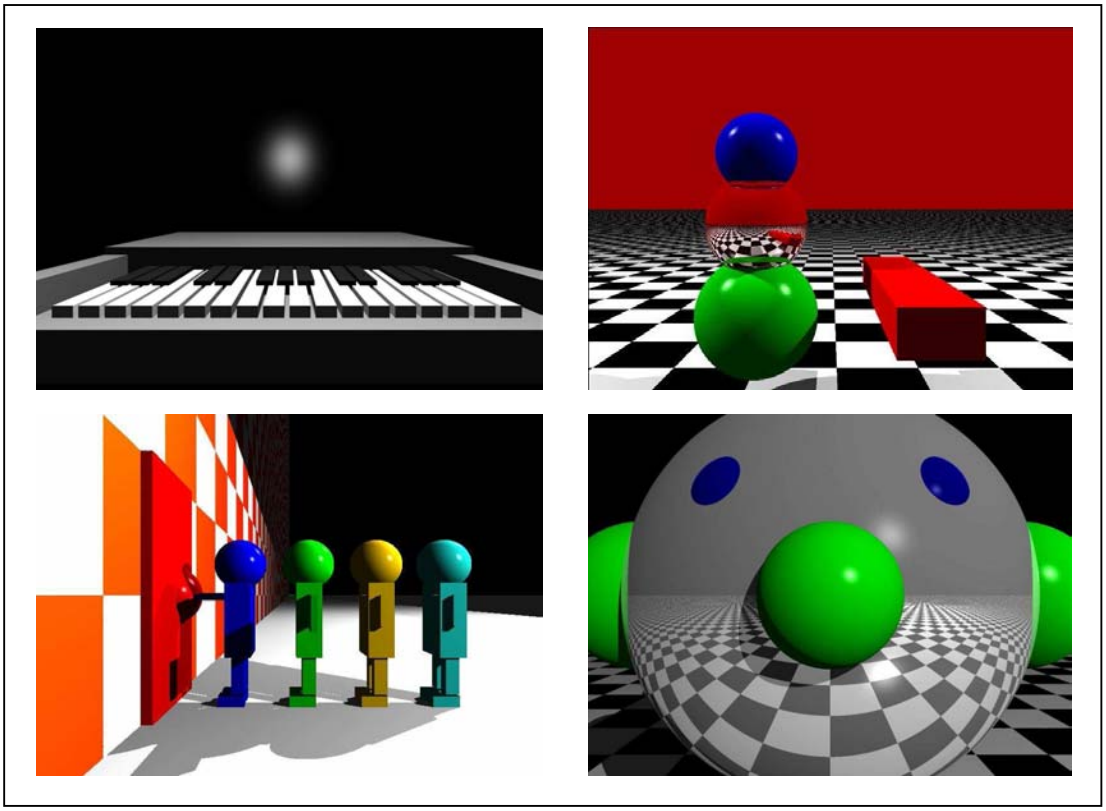

Figure 6: *Student images*

[DGA01] DUCH B., GRON S., ALLEN D.: *The power of problem-based learning*, Stylus Publishing, LLC, Sterling, VA, 2001.

[FP97] FELL, H. J., AND PROULX, V. K. Exploring Martian planetary images: C++ exercises for CS1. *SIGCSE Bulletin*, 29(1), 1997, 30-34.

[Hun03] HUNT, K. Using image processing to teach CS1 and CS2. *SIGCSE Bulletin*, 35(4), 2003, 86-89.

[Mar97] MARTINEZ M.: Designing intentional learning environments, *Proceedings of the 15th Annual International Conference on Computer Documentation*, ACM Press, 1997, pp. 177-178.

[MD06-1] MATZKO, S. AND DAVIS T.: Teaching CS1 with graphics and C, *ACM SIGCSE Bulletin*, 38(3), 2006 [to appear].

[MD06-2] MATZKO, S. AND DAVIS, T.: Using graphics research to teach freshman computer science, *SIGGRAPH 2006 (Educators Program)*, 2006 [to appear].

[MV05] MCANDREW, A., AND VENABLES, A.: A "secondary" look at digital image processing. *SIGCSE Bulletin*, 37(1), 2005, 337-341.

[Mor98] MORDECHAI B-A.: Constructivism in computer science education, *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, 1998, pp. 257-261.

[RAG01] REINHARD, E., ASHIKHMIN, M., GOOCH, B., AND SHIRLEY, P. Color Transfer between Images. *IEEE Computer Graphic and Applications*, 21(5), 2001, 34-41.

[WN05] WICENTOWSKI, R., AND NEWHALL, T.: Using image processing projects to teach CS1 topics. *SIGCSE Bulletin*, 37(1), 2005, 287-291.