τέχνη Photons: Evolution of a Course in Data Structures

A. T. Duchowski[†1]

[1]Clemson University, USA

Abstract

This paper presents the evolution of a data structures and algorithms course based on a specific computer graphics problem, namely photon mapping, as the teaching medium. The paper reports development of the course through several iterations and evaluations, dating back five years. The course originated as a problem-based graphics course requiring sophomore students to implement Hoppe et al.'s algorithm for surface reconstruction from unorganized points found in their SIGGRAPH '92 paper of the same title. Although the solution to this problem lends itself well to an exploration of data structures and code modularization, both of which are traditionally taught in early computer science courses, the algorithm's complexity was reflected in students' overwhelmingly negative evaluations. Subsequently, because implementation of the kd-tree was seen as the linchpin data structure, it was again featured in the problem of ray tracing trees consisting of more than 250,000,000 triangles. Eventually, because the tree rendering was thought too specific a problem, the photon mapper was chosen as the semester-long problem considered to be a suitable replacement. This paper details the resultant course description and outline, from its now two semesters of teaching.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics Data Structures and Data Types

1. Introduction

This paper is the latest in the succession generated by the τέχνη project, our problem-based undergraduate curriculum which originated from our experiences in the establishment of a cross-disciplinary Digital Production Arts (DPA) program. The DPA master's level degree combines elements of computer science, art, theater, and psychology, among others. Graduates who have completed the program pursue careers in the special effects industry for film, television, and gaming. Studios that have hired our students include Rhythm & Hues, Industrial Light & Magic, Pixar, Blue Sky, Electronic Arts, and Sony Imageworks.

The τέχνη project advocates problem-based learning in teaching undergraduate courses, emphasizing development of a semester-long project that incorporates concepts taught in the course. Although any engaging, cogent problem is thought to be better than popular but disparate "toy" problems, computer graphics problems are particularly suitable. Computer graphics problems lend themselves to teaching general computer science concepts for three primary reasons [DD07]. First, the complexity of graphics problems draws on sophisticated solutions. Second, visualization of solutions provides visual feedback to students that provides evaluation of correctness as well personal satisfaction. Third, graphics problems also provide a level of artistic freedom not as readily afforded by other problems (e.g., list sorting).

The τέχνη project specifies four pillars upon which courses should be designed and taught [DGSW11]:

1. *re-combining art and science*: the word τέχνη is the Greek word for *art* and shares its root with τεχνολογία, the Greek word for *technology*. Essentially, we believe that technical computing problems combined with a visual component engages students more than either technically challenging or visually demanding problems alone.
2. *problem-based learning*: this approach is well-known and its use widespread [DGA01], and in our τέχνη variation is extended to last the duration of the course, i.e., a semester in our case.
3. *visual domain*: problems are drawn from computer graphics and visualization. We do this partially because

† duchowski@clemon.edu

DOI: 10.2312/conf/EG2012/education/049-056

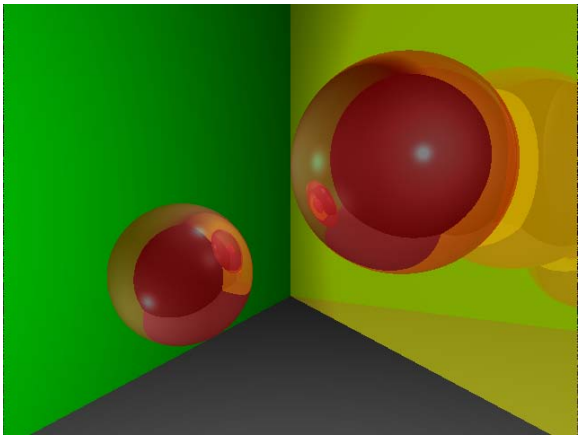


Figure 1: *Result of the final programming assignment in CS2 where a ray tracer is developed in C.*

we believe that graphics are interesting to students, and partially because of Cunningham's [Cun02] observation of computer graphics providing support for development of cognitive tools for effective problem solving.

4. *cognitive apprenticeship*: we believe that the master-apprentice relationship transfers to development of cognitive skill in the classroom, particularly when the content is challenging for both instructor as well as student.

The τέχνη project's philosophical basis, as well as results of instructional experiments with introductory courses, have been presented in several papers [DGMW04, MD06, DD07, DGSW11], and in the dissertation by Matzko [Mat08].

In this paper the latest incarnation of a data structures course is discussed which relies on the development of a photon mapper, following both a traditional data structures textbook [Wei06] and Jensen's [Jen01] monograph.

2. τέχνη Project Overview

The first course in the τέχνη curriculum is CS1 (first-year course introducing computer science), in which a semester-long image processing project is used to teach the required basics (e.g., file input/output, looping, etc.). The course culminates in the implementation of an image re-coloring algorithm described by Matzko and Davis [MD06].

In the next course, CS2, students implement a ray tracer, a topic previously exclusively taught in our graduate-level advanced graphics course. The ray tracer is pedagogically ideal for several reasons: it covers a broad range of concepts, provides visual feedback, and naturally leads to object-oriented program design. The course has been offered in various forms several times with excellent results, detailed by Davis et al. [DGMW04]. An example of a fairly simple image readily produced by students in the course is shown in Figure 1.

Duchowski and Davis [DD07] first published experiences from the initial problem-based version of the next course, CS3, the data structures and algorithms course that is the subject of this paper. Our initial choice of a semester-long graphics problem was Hoppe et al.'s [HDD*92] surface reconstruction from unorganized points. This problem was originally chosen partially for its complexity as well as for its necessity for efficient algorithmic design, without which a large data set would require significantly long periods of computation. Unfortunately, while the latter aspect of the problem is justifiable from the perspective of teaching algorithm analysis, the former complexities were too great to easily teach within one semester at this level of instruction.

What the CS3 course required was the motivation for efficient algorithmic design, e.g., strategies for fast search or query, without increasing the complexity too much beyond the ray tracer. The next evolution of the course was described by Duchowski et al. [DGSW11], wherein a ray tracer relied on a spatial subdivision data structure to efficiently search through more than 250,000,000 triangles evaluated in the ray-object intersection. At this point we taught the course in two variants, where one section of the course employed the graphical τέχνη problem-based approach while the other employed the traditional textbook approach. Assessment of pre-test and final exam responses showed that the τέχνη approach led to improved exam scores, providing compelling evidence for the effectiveness of the τέχνη method.

However, the second version of the CS3 course was also subjectively evaluated by students as lacking in variety, possibly suffering from a singular focus. In the version of the course discussed in this paper, a simpler version of the problem was adopted, namely the photon mapper, which still relies on efficient query operations (namely as facilitated by the *k*d-tree), but its implementation is somewhat simpler as it largely relies on manipulation of points instead of triangles. The course content is detailed in subsequent sections.

2.1. Pre-requisite Course Description

The CS3 data structures course builds on completion of the CS2 pre-requisite course covering the C programming language and providing a brief introduction to C++. The pre-requisite C/C++ course covers advanced programming and culminates in the development of a basic ray tracer written in C. The course also includes basic programming concepts, including file input and output (writing `.ppm` image files) and programming tools such as the use of the `Makefile`.

The course teaches development of a basic vector library to support ray-object intersection, a linked list to support storage of scene objects and basic iteration, and a hierarchical data structure to represent the objects. Design and implementation of these data structures follows object-oriented principles. The vector library implements vector initialization and the idea of simple vector operations, e.g., vector addition, subtraction, etc. The linked list resembles a generic

container in that it stores `(void *)` pointers. Data structures representing objects implement inheritance by deriving specific C structures from a generic object type.

Although the data structures supporting the ray tracer are developed using object-oriented ideas, the implementation falls short of a complete object-oriented system, resulting in a number of important limitations that are addressed in the follow-on CS3 data structures course.

3. Course Content and Project Description

The CS3 data structures course content and organization generally follows Weiss' *Data Structures and Algorithm Analysis in C++* textbook [Wei06] listed below and split into three main parts:

1. Objects and C++
   - objects and classes
   - templates
   - inheritance
2. Algorithms and Analysis
   - algorithm analysis
   - data structures
   - recurrence relations
   - sorting algorithms
3. Implementations
   - priority queues (heaps)
   - linked lists
   - stacks and queues
   - binary search trees
   - AVL trees
   - graphs
   - hash tables
   - (2D and 3D) *k*d-trees

Jensen's *Realistic Image Synthesis Using Photon Mapping* [Jen01] is listed as a supplemental text, but is not required.

The course progresses with seven assignments whose development builds toward the final implementation of the course project, the photon mapper. The middle assignment implementing AVL trees is the only one that does not directly contribute to the final project:

1. basic C++ ray tracer
2. parallelized ray tracer using `OpenMP`
3. ray traced transmission
4. AVL trees
5. photon emission
6. 2D *k*d-tree visualization
7. photon mapper

The AVL tree serves as something of a break in the development of the (approximately 16-week) semester-long project.

Project development is supported to a limited extent by twelve laboratory exercises. Some labs are directly related to the project, others concentrate on algorithm analysis:

1. array class
2. timer class, bubblesort
3. quicksort
4. `omp` parallelization
5. binary heap
6. linked list class
7. stack
8. binary search trees I
9. binary search trees II
10. `Qt` photon visualizer
11. *k*d-tree
12. Dijkstra's shortest path

Abstract data type C++ class development gradually incorporates the C++ Standard Template Library (STL). For example, students first develop their own doubly-linked list along with their own implementation of iterators, then eventually are introduced to STL's templated `vector` class.

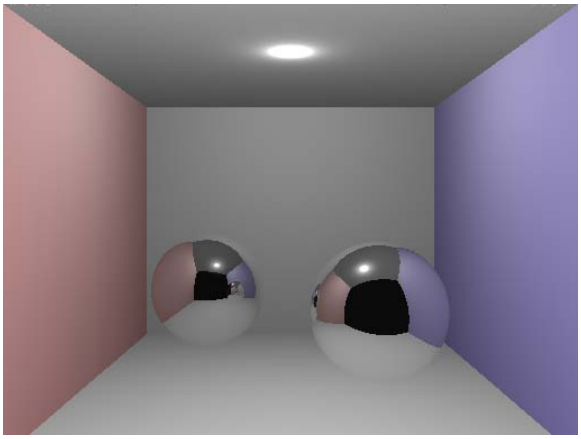


Figure 2: *Result of the first two programming assignments showing distance-attenuated reflection with the scene patterned after the well-known Cornell box scene used by Jensen [Jen01].*

Eventually, lab 12 (Dijkstra's shortest path) is implemented using STL `map` (associative array) containers.

Select source code snippets are dissected in class and source code solutions to each assignment are provided following the due date.

3.1. From Ray Tracer to Photon Mapper

The photon mapper was chosen as the semester-long problem because it puts to use the theory that is taught in a data structures and algorithms analysis course: good data structures and efficient algorithms are required to quickly render photon mapped scenes. In particular, the photon mapper relies on spatial localization of photons at the ray-surface intersection, retrieved through search queries to a *k*d-tree data structure which stores the photons. It is made clear to students that, without the *k*d-tree, this search, repeated several times by each ray, requires $O(n)$ comparisons of a large number of photons (e.g., 20,000), whereas reduction of the number of comparisons to $O(\log_2 n) \cong 14$ per ray speeds up the program considerably. Natural questions of what is "big-oh" notation and how is this speed up achieved are met with a response indicating that that is what the course is about.

Note that the focus of development is more on data structures than on computer graphics effects, hence inter-reflections are not considered during photon bounces. Effects such as color bleeding may be introduced in future iterations of the course.

3.1.1. Assignment 0

The ray tracer developed for the first assignment is a basic C++ implementation. An initial version is made available

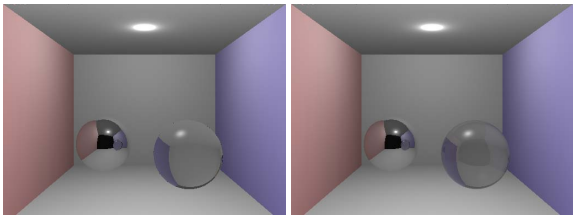


Figure 3: *Result of the third programming assignment showing refraction when the right sphere is made transparent, without (left) and with (right) Schlick's approximation.*

to students, and it should match what students would have developed in the pre-requisite C programming course. The point of the assignment is to level the playing field making sure all students start with the same basic ray tracer implementation. One small requirement is that the previous standalone `ray_trace` C function is now converted to a public member function of a newly introduced C++ ray class. A code snippet of what is meant by "ray spawning" is given.

```
ray = new ray_t(pos,dir); // spawn ray
ray->trace(model,color);  // trace ray
delete ray;               // delete ray
```

3.1.2. Assignment 1

The basic ray tracer exposes two design flaws that appear when, during development of the second assignment (see Figure 2), the code is parallelized for execution on multicore machines via the `OpenMP` library (supported by recent versions of the GNU `g++` compiler). The first of the two design flaws involves per-object storage of ray-object intersection tests, i.e., an object stores a pointer to the object last hit by a ray. This works sufficiently well when there is only one ray shot into the scene. When there are multiple rays shot into the scene simultaneously, race conditions ensue when the ray queries the object it hits testing for self-collisions.

The second of the basic ray tracer's design flaws concerns the linked list used in the pre-requisite C course to store scene objects. The linked list maintains a pointer to the current object during iteration through the scene objects testing for ray-object intersection. Once again this functions well enough when there is only one ray, but race conditions once again ensue when multiple rays concurrently query the list for the next object when the list (scene model) is shared among rays. Copying the model to each of the concurrent `OpenMP` threads is clearly inefficient; the more appropriate solution is to make the current object pointer private to each thread. This naturally suggests the implementation of a list iterator which each thread maintains during ray-object intersection tests. A code snippet of the `omp` call is given but private and shared arguments are left for students to fill in.

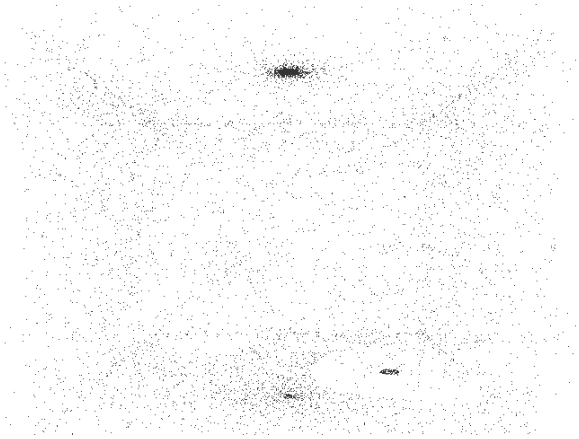


Figure 4: *Result of the fifth programming assignment showing a visualization of emitted photons that have "stuck" to surfaces in the scene. The important aspect of this visualization is the concentration of caustic photons below the right sphere following emission in random directions from the source point light source.*

```
#pragma omp parallel for \
          shared(...fill this in...) \
          private(...fill this in...) \
          schedule(static,chunk)
```

3.1.3. Assignment 2

The ray class developed in the first assignment facilitates implementation of the third assignment, requiring handling of transparent objects. Transparent rays implement refraction through transparent objects as governed by Snell's Law (see Figure 3). Calculation of the transmission ray's direction,

$$t = \frac{n}{n_t}(\mathbf{d} - (\mathbf{d}\cdot\mathbf{n})\mathbf{n}) - \mathbf{n}\sqrt{1 - \left(\frac{n}{n_t}\right)^2 (1 - (\mathbf{d}\cdot\mathbf{n})^2)}$$

with incoming ray direction $\mathbf{d}$, surface normal $\mathbf{n}$, and indices of refraction n and n_t, as derived by Shirley [Shi02], is implemented as a vector class `refract` member function. If the term in the radical is negative, total internal reflection is returned by calling the vector's `reflect` member function. Schlick's [Sch94] approximation is mentioned in passing and left to students as an optional implementation. Prototypes for the vector class member functions are suggested.

```
vec_t vec_t::refract(const vec_t&, float);
vec_t vec_t::reflect(const vec_t&);
```

3.1.4. Assignment 3

Following the third assignment, the fourth assignment requires implementation of an AVL tree. The AVL tree imple-

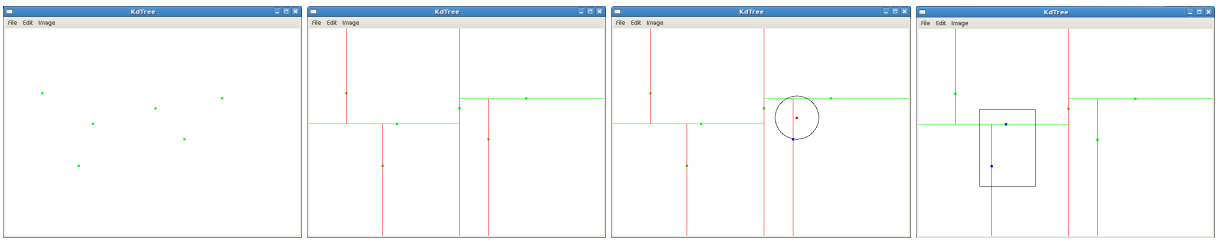


Figure 5: *Result of the sixth programming assignment showing a visualization of a 2D kd-tree storing a number of 2D points manually input by the user. The two rightmost panels depict the nearest-neighbor query and the range query on the data set.*

mentation prepares students for the *k*d-tree implementation by getting them accustomed to tree insertion, traversal, and public and private access to the tree's member functions.

3.1.5. Assignment 4

The fifth assignment (see Figure 4) is closely linked to the tenth lab in that it relies on the development of a simple `OpenGL` visualizer of emitted photons that have "stuck" to surfaces in the scene. Photon emission code is provided.

```
vec_t photon_t::emit(const vec_t& n)
{
    double azmt=genrand(0.0,2.0*M_PI);
    double elev=genrand(0.0,2.0*M_PI);
    double sinA=sin(azmt), cosA=cos(azmt);
    double sinE=sin(elev), cosE=cos(elev);

  vec_t dir=vec_t(-sinA*cosE,sinE,cosA*cosE);
  vec_t vup=vec_t( sinA*sinE,cosE,cosA*sinE);
  vec_t out=(dir + vup).norm();

  return (out.dot(n) >= 0) ? out : -out;
}
```

Because the `OpenGL` library leaves open rendering window management, the programmer is free to choose the windowing toolkit within which the `OpenGL` scene is rendered. The data structures course introduces students to event-driven programming through the use of the `Qt` object-oriented C++ toolkit. `Qt` provides several conveniences that simplify implementation, including a file browser dialog box along with easy backbuffer copying and image class output for saving screenshots of the scene.

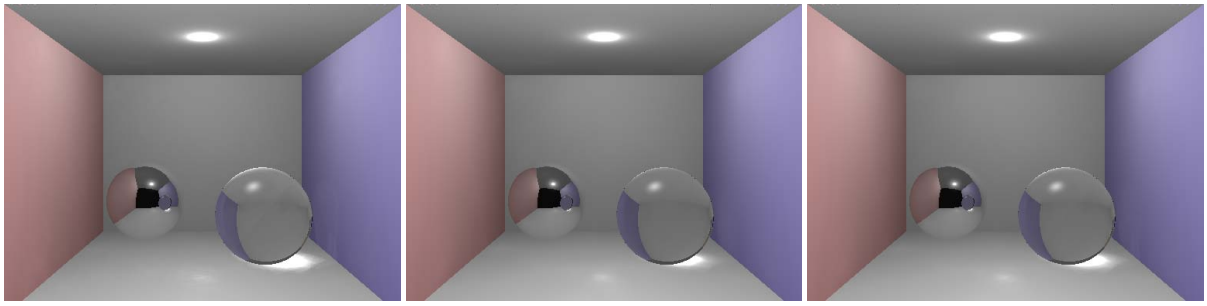
3.1.6. Assignment 5

The sixth assignment also relies on code developed in the lab as both focus on development of the *k*d-tree. Besides its construction, three nearest-neighbor query variants are implemented (with increasing difficulty): nearest-neighbor and *k*-nearest-neighbor searches, as well as the range query. Eventually, the *k*-nearest-neighbor query is the one that is used by the photon mapper, however, the nearest-neighbor query is easier to implement first. The only differences between the two queries is that the former requires a list of nearest neighbors that must be maintained in sorted order, with respect to each point's distance to the query point, and the distance to the *k*th nearest point must not be set until *k* candidate points have been stored in the list. An excerpt from Andrew Moore's PhD thesis is provided to students in understanding how the *k*d-tree query tree traversals work [Moo91].

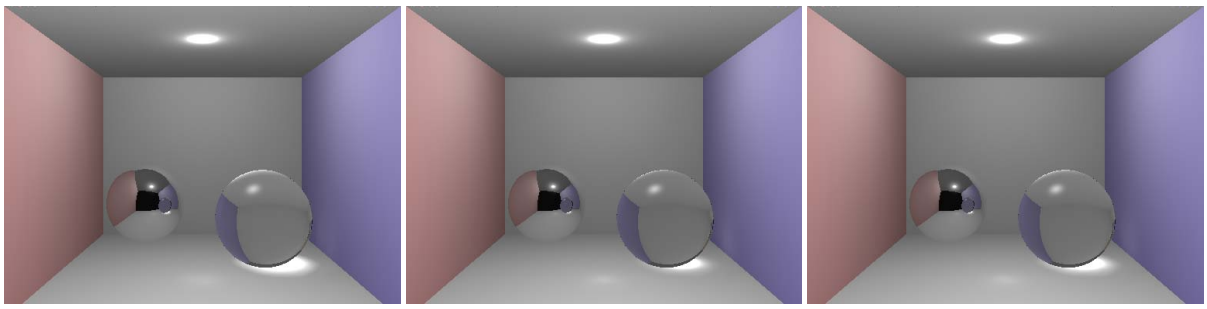
Once again, `Qt` is used to interactively test the *k*d-tree implementation (see Figure 5). The previous `Qt` assignment acted mainly as a simple, static photon viewer and did not exploit user interactivity beyond the use of a simple file dialog. In this assignment, more event-driven programming concepts are introduced. The *k*d-tree visualizer requires that the program respond to the user events listed in Table 1.

Table 1: *Overriding `Qt QGLWidget` events.*

`initializeGL():`	simply sets `glClearColor`.
`resizeGL():`	the resize event sets up the orthographic projection via `gluOrtho2D`.
`paintGL():`	the main redraw event handler must be made to function in a continuous loop even when there is nothing to draw, e.g., no input points exist, the *k*d-tree is empty, etc.
`mousePressEvent():`	unmodified button presses add points to the input point list, `SHIFT`-modified presses set up the first point of the range query rectangle.
`mouseReleaseEvent():`	button-modified mouse releases initiate queries: `ALT`-modified button releases query for the nearest neighbor point, `ALT-SHIFT`-modified releases for the *k* nearest neighbors, `SHIFT`-modified releases obtain the second point of the range query rectangle.



(a) 2,000 global and 500 caustic photons, 10 photon samples, 5 ray and 10 photon bounces executing in about 5 seconds.



(b) 20,000 global and 5,000 caustic photons, 100 photon samples, 5 ray and 10 photon bounces executing in about 112 seconds.

Figure 6: *Results from the seventh and final assignment, executed in parallel on an 8-core Mac Pro, with images shown, left-to-right with no filtering, cone filter, and Gaussian filter.*

3.1.7. Assignment 6

The seventh and final assignment requires implementation of the photon mapper. At this point in the semester, this is largely a matter of assembly of previously developed code, hence an element of code re-use is implicitly invoked. In particular, the *k*d-tree is now made to work with 3D photons instead of 2D points. However, because photons are essentially 3D points in space, the C++ *k*d-tree class only changes in its specialization, i.e., because it was designed as a templated container, no code changes are necessary provided the photon class implements the same required member functions that the point class did (e.g., calculation of distance between points). Deriving the photon class from the ray class, all code for photon reflection and transmission is already in place.

Because the final assignment is mainly a matter of code re-use, optional considerations can be explored, such as the use of a filter to smooth the appearance of the radiance estimate. Two filters, one from Jensen's text [Jen01] and one from his papers [Jen96] are used: the cone (w_{pc}) or Gaussian (w_{pg}) filter, each of which is used to weight the power of each of the photons used in the flux computation:

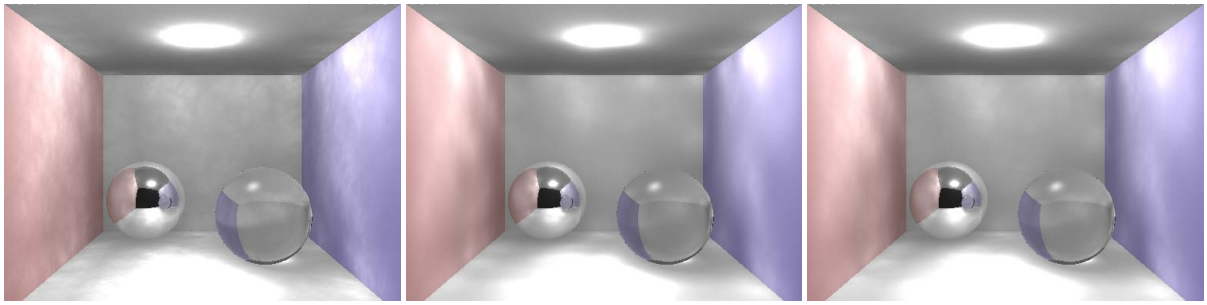
$$w_{pc} = \frac{1 - d_p/(kr)}{1 - 2/(3k)}, \qquad w_{pg} = \alpha \left(1 - \frac{1 - e^{-\beta d_p^2/(2r^2)}}{1 - e^{-\beta}} \right)$$

where $\alpha = 1.818$, $\beta = 1.953$, d_p is the distance from the ray-surface intersection point to each photon, and $k = 1.1$ (just a constant, not to be confused with the *k*-nearest neighbor photons). Expected results are shown in Figure 6, achievable by most students in the class.

4. Lessons Learned from the Course Implementation

As with previous τέχνη courses [DGSW11], on the one hand, this course also requires more effort than a traditional, textbook-directed approach. On the other hand, the effort is motivated by a clear end goal, namely that of matching the scenes rendered by Jensen [Jen01], whereas the textbook approach, motivated by a series of disjointed "toy problems", lacks this type of coherence. Indeed, a fair amount of "sweat equity" is invested by the course instructor, but the return on investment is substantial, particularly toward the end of the semester when students actively participate in critical evaluation of the instructor's work. Two examples of this type of iterative improvement are given.

1. During the first iteration of the course, students pointed out the lack of caustics missing beyond the transparent sphere in the scene. Indeed, on inspection, it was found that the ray object's overloaded `trace` routine failed to call the appropriate version of itself when spawning refracted rays. That is, two overloaded `trace` functions are developed, one that includes photon gathering, effecting the photon mapped synthetic scene, and one that does



(a) 20,000 photons, 50 photon samples, 5 ray and 10 photon bounces executing in about 56-93 seconds.

Figure 7: *Results from the first iteration of the photon mapper with spherical instead of hemispherical photon emission, executed in parallel on a 4-core MacBook Pro, with images shown, left-to-right with no filtering, cone filter, and Gaussian filter.*

not, producing the normal ray traced scene. The function prototypes differ in that one takes the *k*d-tree as an argument, the other one does not. The simple mistake involved the new `trace` calling the old one.

2. During the second iteration of the course, a student pointed out that the suggested code for photon emission sent photons in random directions within a sphere. Images from this iteration are shown in Figure 7 (c.f. Figure 6). The correct solution emits photons in random directions within a hemisphere about the ray-surface intersection point. The solution is straightforward requiring the testing of the dot product of the emission vector with the surface normal—if negative, the emission vector is negated. This fix was particularly rewarding because it led to scenes matching those of Jensen's, which in turn was hopefully perceived as instructor enthusiasm.

Various other small bugs were revisited by the instructor throughout both semesters in an effort to improve the code so that rendered images matched what was found in Jensen's book and web pages, and to facilitate teaching. Solution code is provided following completion of each programming assignment. One particular source of frustration was correct ray transmission calculation, which had eluded the instructor during the first iteration of the course. This nagging problem was finally resolved through consultation of Shirley's book [Shi00] along with his on-line notes, contrasted against what is found in Glassner's text [Gla89].

Finally, with an increased understanding of the photon mapper came the opportunity for divergence of the solution, namely, we used one photon map that was populated by two different types of photons, caustic and global. Jensen [Jen96] suggests the use of two separate photon maps, but it appears that one is sufficient to generate comparable results.

5. Student Feedback and Performance

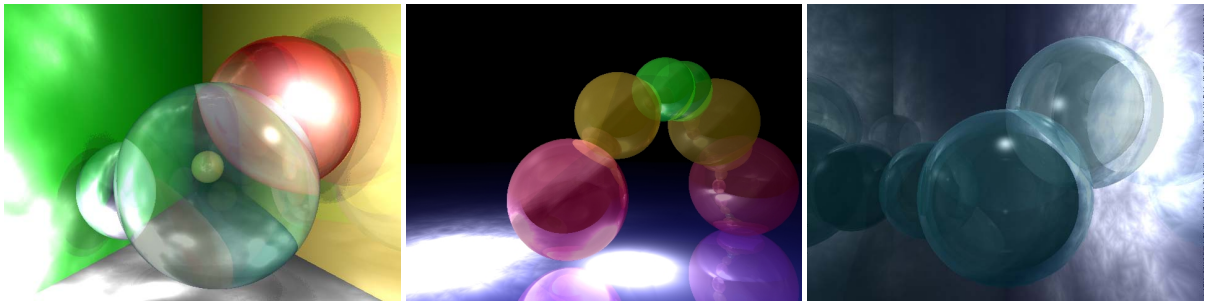
Student responses (12 of 36) from the first semester's class indicated that the course was perceived as more difficult than other classes at the same level (mean response 4.167 vs. 3.736 on a 5-point response scale) while requiring a comparable amount of work (mean response 4.083 vs. 4.125 at the same level). The course does indeed require a good deal of programming, with the final assignment consisting of about 3,000 lines of code (counting C++ interface and implementation files). Unprepared students are likely to complain.

It is not clear which cross-section of the class submitted student evaluations, but only one third did so. Similarly, only one third of the class submitted an optional, "creative" image. It is likely that these were not the same students who provided student evaluations. However, of the 12 who submitted creative images (three are shown in Figure 8(a)), comments they provided to the instructor suggested that the final assignment was easier than initially perceived, and that they derived satisfaction from its completion.

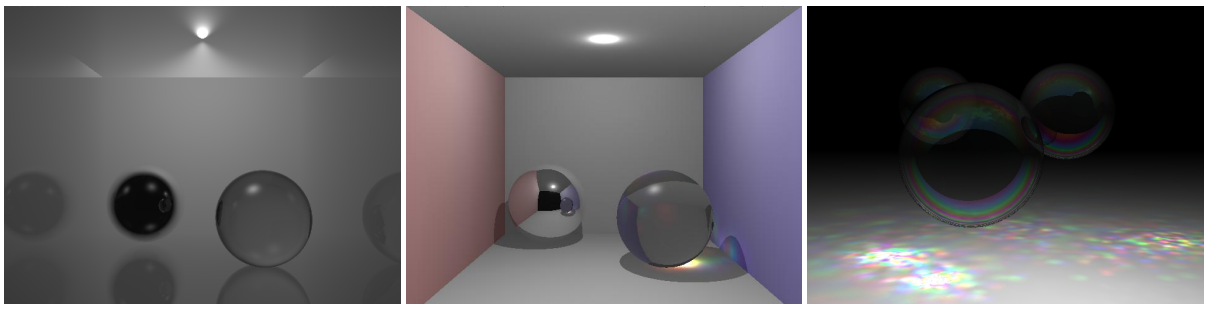
Student responses from the second semester have not yet been distilled, but are likely to be similar. Of the cross-section of the class that is expected to submit creative images, it appears that these students relish the challenge given to them and enjoy not only learning data structures and algorithms, but they also derive a measure of satisfaction from the visual nature of the task. Several of them pursued implementing Schlick's approximation to the Fresnel equations that was only discussed briefly, and at least one student has expressed interest in pursuing a computer science honors project involving further exploration of the photon mapper. This particular student independently implemented ray-cast shadows as well as a model of wavelength-based refraction following Cauchy's equation, using it to simulate thin-film interference in soap bubbles (see Figure 8(b)).

6. Acknowledgments

This work was supported in part by the US National Science Foundation under Award #0722313.



(a) Example images from Spring 2011 students: left-to-right, Brenden Roberts, Carrie Eisengrein, and Jacob Adelberg.



(b) Example images from Fall 2011 students: left-to-right, Ryan Geary, and Jason Anderson's Cauchy-based refraction and simulation of thin-film interference in soap bubbles.

Figure 8: *Results from two semesters of teaching the course.*

References

[Cun02] CUNNINGHAM S.: Graphical problem solving and visual communication in the beginning computer graphics course. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer Science education* (New York, NY, 2002), ACM Press, pp. 181–185.

[DD07] DUCHOWSKI A. T., DAVIS T. A.: Teaching Algorithms and Data Structures through Graphics. In *Proceedings of EuroGraphics 2007 (Education Papers)* (New York, NY, USA, 2007), ACM Press.

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

[DGMW04] DAVIS T., GEIST R., MATZKO S., WESTALL J.: τέχνη: a first step. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer Science education* (New York, NY, 2004), ACM Press, pp. 125–129.

[DGSW11] DUCHOWSKI A. T., GEIST R., SCHALKOFF R., WESTALL J.: τέχνη Trees: A New Course in Data Structures. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (New York, NY, 2011), SIGCSE '11, ACM, pp. 341–346.

[Gla89] GLASSNER A. S. (Ed.): *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surfrace Reconstruction from Unorganized Points. In *Computer Graphics (SIGGRAPH '92)* (New York, NY, 1992), ACM, pp. 71–78.

[Jen96] JENSEN H. W.: Global Illumination using Photon Maps. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)* (1996), Pueyo X., Schröder, (Eds.), Springer-Verlag, pp. 21–30.

[Jen01] JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. A K Peters, Ltd., Natick, MA, 2001.

[Mat08] MATZKO S.: τέχνη *and Quest-Oriented Learning*. PhD thesis, Clemson University, Clemson, SC, 2008.

[MD06] MATZKO S., DAVIS T.: Using graphics research to teach freshman computer science. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Educators program* (New York, NY, 2006), ACM Press, p. 9.

[Moo91] MOORE A. W.: *Efficient Memory-based Learning for Robot Control*. PhD thesis, University of Cambridge, Cambridge, UK, October 1991. Technical Report No. 209, URL: http://www.autonlab.org/autonweb/14712.html.

[Sch94] SCHLICK C.: An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum 13*, 3 (1994), 233–246.

[Shi00] SHIRLEY P.: *Realistic Ray Tracing*. A K Peters, Ltd., Natick, MA, 2000.

[Shi02] SHIRLEY P.: *Fundamentals of Computer Graphics*. A K Peters, Ltd., Natick, MA, 2002.

[Wei06] WEISS M. A.: *Data Structures and Algorithm Analysis in C++*, 3rd ed. Pearson Education (Addison-Wesley), Boston, MA, 2006.