Teaching Introductory Computer Graphics Via Ray Tracing

Helen H. Hu
Westminster College, Salt Lake City, UT
hhu@westminstercollege.edu

Figure 1. Examples of student work. For fun, enthusiastic students created these scenes, which showcase mirror reflections, shadows, texture maps, transparency, and Mandelbrot fractals.

Abstract

Ray tracing is a computer graphics technique capable of creating visual effects such as realistic soft shadows, reflections, refractions, motion blur, and depth of field. Historically considered computationally expensive, ray tracing is gaining in popularity as computing power (primarily the recent increase in the number of processor cores) continues to increase. This paper details an introductory computer graphics course taught around a Java ray tracer. For most students this was their first exposure to the field of graphics. Having programmed primarily in Java since CS 1, the students preferred Java to C++. Like any other Java program, the ray tracer ran on all platforms, minimizing students’ frustrations and avoiding investment in a new IDE. For most homework assignments, students added features to a bare-bones ray tracer, converted from the C++ code in Kevin Suffern’s textbook Ray Tracing from the Ground Up [13]. Additional non-ray tracing homework was also assigned to expose students to other fundamental areas of computer graphics, such as color models, transformations, viewing and event-driven interactions.

1. Introduction

The traditional introductory computer graphics course emphasizes geometric and raster algorithms, such as the Bresenham line drawing algorithm, scan conversion, BSP trees, and clipping. Cunningham describes this course as “Computer Graphics Algorithms and Techniques” [2]. With the innovation of graphics APIs such as OpenGL and Direct3D in the 1990’s, the priorities of graphics programmers shifted. In response, many instructors transformed their graphics classes to use OpenGL and emphasized interactive camera control over raster line algorithms [12]). Today, most graphics textbooks follow this newer course model and are written for students programming in C++ using OpenGL. Other classic computer graphics textbooks still in print follow the traditional course model (“Computer Graphics Algorithms and Techniques”). What is missing from these textbook options is a computer graphics textbook written for students programming in other languages that emphasizes advanced graphics effects rather than the classic algorithms.

The OpenGL course model has many advantages over the traditional computer graphics course. By using an API that provides the fundamentals (such as polygon drawing and occlusion), students quickly progress to writing programs with advanced graphics effects, such as shading and fog. Students are motivated by their results, but OpenGL does have the drawback of being difficult to debug. OpenGL was created with graphics hardware in mind, not for programmer convenience, making it less appropriate for teaching graphics concepts. Larger departments, especially those with graduate programs, often compromise by offering two or more graphics classes, to be taken
in series. The first is typically the more traditional course, emphasizing classic graphics algorithms. The follow-up course then introduces students to OpenGL programming.

At smaller colleges where only one computer graphics course is offered, an OpenGL course may not be the best choice. Westminster College is a small liberal arts college graduating approximately 10 computer science majors each year. For most students, our single graphics course is their only exposure to computer graphics. Other courses are taught primarily in Java, with little exposure to C++. Because the department does not have its own computer lab, most students program on their own laptops. When OpenGL was used in a previous offering of the graphics course, several students had difficulty installing it. They also were unfamiliar with C++ and struggled debugging their programs.

In response to these difficulties, some Java-based departments have had success using Java2D [15], Java3D and/or the JOGL libraries in their graphics courses [17]. Java2D and Java3D are graphics APIs written specifically for the Java language. JOGL allows Java programmers to access the OpenGL API. However few general purpose graphics textbooks exist for either Java3D or JOGL; most Java3D and JOGL books are programming tutorials instead. Furthermore, JOGL has OpenGL’s disadvantages as a difficult state machine to debug.

To give students the motivation of implementing advanced graphical effects, while avoiding the challenges of OpenGL, I redesigned my introductory graphics class in Spring 2010. For my new course I taught computer graphics within the context of a Java ray tracer, assigning readings from the textbook Ray Tracing from the Ground Up [13]. Although the code accompanying the textbook is in C++, I gave my students a bare bones ray tracer written in Java. For homework assignments, students added features to the Java ray tracer, learning about fundamentals such as perspective projection, shading models, transformations, anti-aliasing, texture mapping and instancing (Figure 1).

Ray tracing is an elegant rendering algorithm first introduced in 1980 [14]. Instead of using transformations to create an image of a 3D scene (as with OpenGL and Direct3D), ray tracing reverses the physics of light transport. In the real world, light photons leave the light source and scatter throughout the environment until an extremely small percentage of the light reaches a human eye or a camera lens. To save time, a ray tracer “traces” rays starting at the eye and scattering throughout the scene. The ray tracer generates at least one ray for each pixel in the image. The color of each pixel is based on the objects the ray hits and the light visible from these hit points. Originally too computationally expensive to be used widely, ray tracing has become a viable rendering method in recent years. It is often used to create visual effects in movies, and some experts speculate that video games will soon be rendered with ray tracing rather than OpenGL and Direct3D [12].

Other instructors have published their experiences using a ray tracer in their computer graphics courses [4, 9, 16], although these ray tracers are typically written in C++ rather than Java. There are also many ray tracing books besides the one assigned in this course, however most are too advanced for an introductory graphics course [5, 8, 11]. Other texts are inappropriate as textbooks because they are too focused on ray tracer programming and not enough on the fundamental graphics algorithms.

The next section of this paper provides a detailed description of the course content. The paper then discusses how to cover all essential graphics topics in this type of graphics course. Finally, improvements to the course are suggested.

2. Course Content

The computer graphics course at Westminster College is an upper-division class taken by junior and senior computer science majors. While most of the course was structured around the textbook, Ray Tracing from the Ground Up [13], handouts on more general graphics topics were used to supplement the required textbook (Table 1).

Providing the students with a bare bones ray tracer was essential to the course. Kevin Suffern’s web site http://www.raytracegroundup.com/ provides two versions of a C++ ray tracer. With the author’s permission, I converted most of the online C++ code to Java and released the code incrementally to my students. Of the seven programming assignments, four involved extending the ray tracer. The remaining three assignments taught basic graphics concepts that were not covered by the ray tracer.
<table>
<thead>
<tr>
<th>Week</th>
<th>Topic</th>
<th>Assigned Reading</th>
<th>Week</th>
<th>Topic</th>
<th>Assigned Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Colors</td>
<td>*</td>
<td>9</td>
<td>Shadows</td>
<td>Ch. 16</td>
</tr>
<tr>
<td></td>
<td>Pixels</td>
<td>Ch. 1</td>
<td></td>
<td>Reflections</td>
<td>Ch. 24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Transparency</td>
<td>Ch. 27</td>
</tr>
<tr>
<td>2</td>
<td>Gamma</td>
<td>*</td>
<td>10</td>
<td>Viewing Transformations</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Modeling</td>
<td>Ch. 23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Math Background</td>
<td>Ch. 2</td>
<td>11</td>
<td>Texture Mapping</td>
<td>Ch. 29, 30, 31</td>
</tr>
<tr>
<td>4</td>
<td>Bare Bones Ray Tracing</td>
<td>Ch. 3</td>
<td>12</td>
<td>Anti-aliasing</td>
<td>Ch. 4, 5</td>
</tr>
<tr>
<td></td>
<td>Raster Algorithms</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Viewing System</td>
<td>Ch. 9</td>
<td>13</td>
<td>Ambient Occlusion</td>
<td>Ch. 17</td>
</tr>
<tr>
<td></td>
<td>Ray Object Intersection</td>
<td>Ch. 19, 23</td>
<td></td>
<td>Area Light Sources</td>
<td>Ch. 18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Grid Acceleration</td>
<td>Ch. 22</td>
</tr>
<tr>
<td>6</td>
<td>Lights and Materials</td>
<td>Ch. 13, 14, 15</td>
<td>14</td>
<td>Thin Lens Camera</td>
<td>Ch. 10</td>
</tr>
<tr>
<td>7</td>
<td>Theory</td>
<td>Ch. 13</td>
<td></td>
<td>Global Illumination</td>
<td>Ch. 26</td>
</tr>
<tr>
<td></td>
<td>Smooth Shading</td>
<td>Ch. 23</td>
<td></td>
<td>Radiosity</td>
<td>*</td>
</tr>
<tr>
<td>8</td>
<td>Transformations</td>
<td>Ch.s 20, 21</td>
<td>15</td>
<td>Graphics Pipeline</td>
<td>*</td>
</tr>
</tbody>
</table>

* refers to supplemental handouts from other sources

Table 1. Topics Covered in CMPT 360, Spring 2010. Chapter numbers are from Suffern's textbook.

2.1 Homework Assignments

The seven homework assignments were:

1. Color Models (2 weeks)

   Students learned about color models and matrix multiplication. They wrote a utility class that converted between color spaces and unit tests to test their methods. Students then converted a Java Swing GUI into a simple testing program that displayed two colors (the original color and the same color run through two consecutive conversions).

2. Building Orthonormal Basis and Modeling (1 week)

   Students were given the code for the bare bones ray tracer, and were expected to get it compiling and running on their own systems. They also answered some simple questions about how a ray tracer worked. (Questions from the textbook were assigned for all ray tracing homework assignments.)

   Separately, they wrote a utility class that built a 3D orthonormal basis. This class relied on two of the ray tracer’s classes, which required students to start familiarizing themselves with the ray tracer code.

   Finally, they used their utility class to complete a 3D model of a barn. They were given the barn’s vertices, but were expected to return the polygon faces and their normals.

3. Pinhole Camera and Mesh Geometry (2 weeks)

   Students converted the bare bones ray tracer (orthographic viewing) to a pinhole camera. They then wrote a method to read in SMF (“Simple Mesh Format”) meshes into their ray tracer. (The online code included C code to read in PLY files.)

4. Phong Materials and Smooth Shading (2 weeks)

   Java code implementing directional light sources and matte materials was released. Students added point light sources and Phong materials (both the original Phong model and Blinn’s modifications). Students implemented smooth shading for their SMF meshes and generated images contrasting flat to smooth shading.

5. Reflections, Shadows and Instancing (2 weeks)

   Again, more Java code was released. Students added instancing to their ray tracer, allowing them to convert a sphere to an ellipsoid without adding a new geometric primitive to their ray tracer. Students also added reflections and shadows to their ray tracer. These two visual effects required students to make minor modifications to several classes.

6. 3D Viewing (2 weeks)
This non-ray tracing assignment required the students to complete a utility class that built matrices for orthographic and perspective projection. They were given a Java GUI program that read in 3D geometry. They were allowed to use their choice of either the ray tracer’s Matrix3D class or the JAMA Matrix class [6]. Students displayed 3D wireframe objects in their programs through matrix multiplications and homogenous coordinates. They also added translation, rotation and zooming interactions of the 3D objects.

7. Transparency, Texture Mapping, Grid Acceleration (2 weeks)

Most of the code for transparency was provided, although students wrote a transparency material class. Students also implemented simple texture mapping. Finally, students added a bounding box method to each geometric object class, to make grid acceleration work in their ray tracers.

2.2 In-class Laboratory Exercises

In addition to the homework assignments, students participated in a few in-class laboratory exercises. As opposed to the graded homework assignments, students earned credit for simply participating in these exercises. The first exercise walked the students through building a GUI for a Java Swing program. A second exercise required them to complete Bresenham’s line algorithm, again in a Java Swing program. Later, students manipulated 2D transformations with Brown University’s excellent online exploratories, specifically the Transformation game [1]. Finally, students experimented and displayed noise-based textures in a simple Java program.

3. Conformance to 2004 SIGGRAPH Recommendation

In 2004, a working group on computer graphics education [3] proposed that an introductory computer graphics course should teach:

1. Transformations
2. Modeling: Primitives, Surfaces, and Scene Graphs
3. Viewing and Projection
4. Perception and Color Models
5. Lighting and Shading
6. Interaction: Event-Driven and Using Selection
7. Animation and Time-Dependent Behavior
8. Texture Mapping

Many but not all of these eight topics can be covered in the context of a ray tracer. By starting with a working bare bones ray tracer, the students were able to quickly implement a pinhole camera (perspective projection), the Phong reflection model, shadows, reflections, transparency and texture mapping. This list of advanced visual effects is similar to the visual effects programmed in many OpenGL computer graphics classes.

The three additional non-ray-tracing assignments were used to expand on most of the remaining topics that were not well covered by the ray tracer. Both color models and GUI interactions were explored in the first homework assignment. A second homework introduced modeling and orthonormal bases, to prepare students for camera transformations. Perspective projection was first introduced with the ray tracer’s pinhole camera, but was later addressed more thoroughly in the 3D Viewing assignment as part of the graphics pipeline. Transformations were coded as part of instancing in the ray tracer, but were also part of the same 3D Viewing assignment.

Two topics from the SIGGRAPH recommendation, animation and scene graphs, were not covered well in the Spring 2010 course. In the future, I would consider replacing a few of the advanced ray tracing topics (e.g., ambient occlusion, area lighting, and the thin lens camera) with a week on scene graphs and animations. Scene graphs could be explored using Brown University’s Scenegraph Builder [1], while animation might be explored using a modeling and animation program, as described by [18].

In my next offering of this course, I plan to add an in-class OpenGL exercise to the “graphics pipeline” week, to allow students to experiment with OpenGL. While the 3D Viewing homework assignment covers the perspective projection aspects of the pipeline sufficiently, students could benefit from seeing everything else that the graphics hardware can now handle.
4. Reflection on Course

While the textbook, *Ray Tracing from the Ground Up*, does not assume any prior computer graphics background, it is not a general purpose graphics textbook. It is intended for anyone who wants to learn about ray tracing. Students will need additional references to supplement the textbook in areas such as perception, color models, 2D algorithms, curves and surfaces, 3D viewing transformations, the graphics pipeline, and radiosity (Table 1). Many of these topics are covered well online (e.g., Siggraph’s HyperGraph at http://www.siggraph.org/education/materials/HyperGraph/toc.htm), but instructors could also assign a more general graphics textbook, possibly as an optional textbook. *Fundamentals of Graphics*[10] is an excellent graphics textbook that is independent of API, while *Computer Graphics: Theory Into Practice*[7] presents OpenGL material separately from the other topics, allowing instructors to decide how much OpenGL to cover.

To reserve time for other more general graphics topics, I limited the discussion of several topics relevant only to ray tracing that were covered in the textbook. While we discussed sampling in the second half of the semester, my students’ ray tracers only generated one ray per pixel. They did not code up any visual effects that required more samples, such as area light sources, glossy reflections, depth of field, or path tracing.

For students used to working in Java, the C++ code throughout the textbook can be a struggle. While I would prefer the same textbook written in Java, the exposure to C++ code is generally educational for students.

As a large collection of over 60 classes, the ray tracer does bring some challenges with it. For some students, the ray tracer was the largest software project they had worked on individually. Many students found the first ray tracing assignment (Homework 3) the most challenging because they had difficulty familiarizing themselves with the packages and classes. In the future, I might assign a UML diagram or some other design project as part of Homework 2 or 3, to help students to learn the ray tracer’s classes better.

The amount of code provided on the textbook web site was a boon, because it allowed students to create advanced visual effects without too much struggling. The majority of the class did not have to debug the “black screen” that is usually faced by most programmers writing their first ray tracers. At the same time, the abundance of code provided had some pedagogical disadvantages. Even some students felt too much code was provided, making some parts of the assignments trivial (e.g., reflections and transparency). The textbook was written for a broad audience; the author had to provide enough code for the independent reader.

In the Spring 2010 class, I gave two students permission to program in C++. In the future, I will be requiring everyone to write their ray tracers in Java. I can be more selective of the released Java code and require students to do more of the programming for reflections and transparency.

The Java code released to my students should be cleaned up and improved. Most importantly, I would like to add the option of saving rendered images to a file. I would be happy to share the Java code with other instructors. Instructors would then be able to customize the difficulty of their assignments by selecting which of the Java classes to release to their students. The bare-bones Java ray tracer can be found on my website http://people.westminstercollege.edu/faculty/hhu. Please contact me by email for the Java code for the full ray tracer.

Students generally gave positive feedback for the course. Every student strongly agreed with the statement “This course challenged me” (a 5 on a scale of 1-5). A typical comment from a student was “Very intense class. Learned a lot of material in the given time.” A few students did not like the 3D Viewing assignment, which produced only wireframes, after working on their ray tracers for three consecutive assignments. In the future, I might assign the 3D Viewing assignment closer to the start of the semester (before the Pinhole Camera homework). This scheduling change has the advantage that students would first learn about perspective transformations by matrix multiplications, followed immediately by the perspective projection that happens automatically with a ray tracer’s pinhole camera. To reduce the amount of background required for the 3D Viewing assignment, I would assign only orthographic and perspective projection. Students could add rotation and translation interactions to the the 3D Viewing program as an in-class exercise later in the semester.

Comparing student performance on the Spring 2010 final exam to those from previous offerings of introductory computer graphics, I believe Spring 2010 students may have had less comprehension of reflection model calcula-
tions. This difference might occur because the calculations were spread out between many different classes in the ray tracer, or perhaps because we programmed only the more complex Phong reflection model (the simpler Gouraud model makes little sense for ray tracing). In the future, I might add an in-class pencil-and-paper exercise on reflection models to help cement student understanding of this topic.

This paper has described an introductory graphics course taught within the context of a Java ray tracer. Because the course emphasizes advanced rendering techniques rather than basic graphics algorithms now handled by hardware, many of my students found this graphics course more engaging and relevant. In addition, while a ray tracer can be difficult to debug, many students found it less frustrating programming in Java, a language they were already proficient in. For faculty in Java-based departments, I strongly recommend this ray tracing based approach for teaching introductory computer graphics.

5. Acknowledgements

I would like to thank Kevin Suffern for his valuable assistance and my CMPT 360 students for providing constructive feedback on this course redesign. Thanks also to David Weinstein and the reviewers for their helpful comments and edits improving this paper.

References