Towards a Singleton Undergraduate Computer Graphics Course in Small and Medium-sized Colleges

AMIT SHESH, Illinois State University

This paper discusses the evolution of a single undergraduate computer graphics course over five semesters, driven by a primary question: if one could offer only one undergraduate course in graphics, what would it include? This constraint is relevant to many small and medium-sized colleges that lack resources, adequate expertise and enrollment to sustain multiple courses in graphics that spread out its vast and evolving content. We strive to include material that would provide (a) a basic but solid theoretical foundation (b) topics, data structures and algorithms that are most practically used (c) ample experience in actual graphics programming and (d) a basic awareness of advanced topics. We have a secondary objective of relating and complementing computer graphics knowledge and programming with topics in other computer science courses to provide a more cohesive understanding to our students. We achieve both objectives by using an "early-scenegraphs" approach to progressively create graphics applications that use XML-based modeling and both pipeline-based and ray traced rendering. We report and analyze results that show how students were able to achieve more complex results within similar time periods while largely retaining prior average student performance in the course. Students also report higher rates of satisfaction with the course when it follows our proposed approach. Pedagogically our main contribution is an evolving blueprint for a single undergraduate CG course that offers flexibility to emphasize different aspects like modeling, rendering, etc. according to the instructor's and students' interests, while aligning the course better within the computer science curriculum especially when resources are limited.

Categories and Subject Descriptors: I.3.6 [**Picture/Image Generation**]: Graphics data structures; K.3.2 [**Comp. & Info. Sci. Education**]

General Terms: Computer graphics, scene graphs

Additional Key Words and Phrases: Computer graphics, scene graphs, graphics in small colleges, graphics education, ray tracing

ACM Reference Format:
Shesh, A. 2012. Towards a Singleton Undergraduate Computer Graphics Course in Small and Medium-sized Colleges *ACM Trans. Comput. Educ.* 9, 4, Article 39 (March 2012), 18 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

As is typically found in undergraduate computer science degree programs, our computer graphics course is one of the most popular upper-level courses. One of the main reasons for this popularity is the prior exposure of most students to the glamorous manifestations of computer graphics, such as computer games, special effects in movies, etc. It is not uncommon for instructors to find students taking a computer graphics course hoping to write a game at the end of the semester. Such high expectations, along with the inherent mathematical nature of the subject and its ever-evolving vast content, make designing and teaching computer graphics courses challenging.

Many degree programs spread out the content of graphics over several courses, forming a cohesive sequence. This allows them to offer courses specializing in animation, real-time graphics, non-photo-realistic rendering, etc. that require a basic foundation provided by an earlier course. However offering multiple courses related to computer graphics alone is not feasible at many universities, due to insufficient resources and enrollment to sustain them or inadequate research focus and expertise in graphics. Given the vast array of topics, if one could offer only one graphics course, what would it include? This paper attempts to answer this question by reporting our experience in incorporating various topics and techniques, looking at students' achievements and arriving at what we consider a set of "optimized" content and assignments. This content to us represents material and programming experience that will best prepare students to either pursue graphics in advanced, graduate studies or dabble in writing computer games and other applications themselves using available frameworks.

A secondary objective is to make the graphics course more "inclusive" and integrated with the overall computer science curriculum. While a typical introductory CG course builds upon prior knowledge of linear algebra, coordinate geometry and programming, it introduces mostly unique and specialized content to its students. This is certainly not because computer graphics as a subject is not inclusive, but because given the vast content such courses focus exclusively on concepts in graphics. Can a graphics course incorporate material that students can relate to other courses? In small but significant ways we were able to introduce topics that are not only related to graphics, but also either taught or used in other courses.

Pedagogically the main contribution of this paper is a customizable blueprint of content and assignments for an introductory undergraduate computer graphics course that has evolved over 5 semesters. This paper *does not* propose a new graphics curriculum or a novel teaching technique. This paper reports our experience with including a collection of traditional topics using specific, progressive and challenging assignments and evaluates how successful it was in fulfilling course objectives and students' expectations. We introduce three themes in our assignments.

First we use an "early-scenegraphs" approach in our assignments, i.e. have students implement and use a scene graph. In addition to being the central data structure in many graphics applications and supported in most game engines, scene graphs can also be valuable pedagogical tools to teach new and reinforce previously taught concepts from other courses. Basing our assignments on scene graphs also allows us to customize the course from one semester to another by emphasizing some topics more than others: modeling, rendering, animation, etc. Scene graphs also work well with XML, a representational format that is widely used but is seldom taught in detail in traditional programming courses.

Secondly we have observed many benefits of building a single graphics system progressively during the semester rather than piece-wise assignments that concentrate on specific aspects of graphics and OpenGL. While this is not a unique idea, we find that when students progressively build a single application by adding new or replacing existing functionality, their knowledge of relevant concepts is reinforced. Pedagogically this presents an additional advantage: such a project reinforces relevant software design concepts while concentrating on their implementation.

Thirdly we include ray tracing in our course. Although uncommon in undergraduate courses [Wolfe 2000] and usually part of a graduate-level course, we have found that our undergraduate students respond enthusiastically to ray-tracing assignments. It also unifies and demystifies all the math and graphics discussed throughout the course, and in our opinion reinforces student understanding. Lastly it fits very well with the overall project architecture, and thus over 15 weeks students end up with an

Table I. Overview of our course

Week	Topics
$1, 2$	Introduction to graphics and OpenGL
3	2D graphics and transformations
4	Introduction to 3D modeling, transformations
$5, 6$	Transformations, coordinate systems, hierarchical modeling
$8-9$	Lighting: basics and OpenGL lighting
10	Texture Mapping
11	Shadow techniques and OpenGL buffers
$12-14$	Ray tracing
$14-15$	Optional topics (Intro to GLSL, advanced rendering techniques, etc.)

interactive OpenGL-based application capable of rendering lit, textured objects, with an in-built ray tracer capable of producing shadows, reflections and transparency.

We find that students in such a course are able to achieve more complex implementations within the same time, respond enthusiastically and maintain or exceed prior average student performance. Such a course also offers a good mix of the foundations of computer graphics and the latest techniques. Instructors can also customize this content, concentrating on specific techniques and algorithms for modeling, rendering, animation, shader programming or even other advanced topics.

The rest of the paper is organized as follows: Section 2 places our course in context of previous relevant literature. Section 3 summarizes the five different *trials* of our course with respect to assignments and increasingly complex student work, culminating in the proposed set of assignments (listed in the appendix). Section 4 discusses the main aspects of our course and correlates them with recommendations in the ACM Computer Science curriculum [2008]. Section 5 briefly discusses some material that integrates with other CS courses. Section 6 assesses the effectiveness of our approach qualitatively (illustrated by student work) and its quantitative impact on student learning, performance and perception. Section 7 discusses some possible variations of our approach to suit similar courses and Section 8 concludes the paper.

2. RELATED WORK

The ACM computer science curriculum [2008] is a widely-used benchmark for assessing the quality of offerings in a computer science degree program. Our course fulfills all the core requirements of this curriculum and includes topics from most other elective offerings. Our course also includes the most popular concepts taught in such introductory courses as well as topics that are relatively uncommon at this level.

Over the years there has been consistent interest in evolving graphics courses in response to changes in graphics hardware, approaches, and supporting frameworks [Cunningham et al. 1988; Grissom et al. 1995; Cunningham 1999; Hitchner et al. 1999; Cunningham 2000; Angel et al. 2006]: Paquette [2005] provides a survey. Various themes to teach computer graphics have been proposed over the years. Bresenham *et al.* [1994] discuss alternatives: systems-based (i.e. using a specific system/API), engineering (i.e. creating libraries for basic graphics algorithms) and application-oriented (i.e. focused on a specific application of graphics). Two distinct approaches to teaching computer graphics have emerged: top-down (using a particular API to *implement* specific applications and techniques) and bottom-up (focusing on fundamental algorithms while giving API usage secondary importance) [Sung and Shirley 2003; Angel et al. 2006; Tori et al. 2006]. More complete tools like Renderman [Owen 1992] or API-agnostic approaches [Schweitzer et al. 2010] have been used in the top-down approach instead of programming APIs. Our course has el-

ements of both strategies. While we discuss most concepts related to modeling and rendering in a generic way, our assignments adopt a specific API (OpenGL). However implementation of a ray tracer in the end forces students to implement operations that had been abstracted or not supported by OpenGL, thus incorporating the bottom-up approach. Due to the ubiquity of inexpensive graphics processors (GPU) shader programming has been integrated in traditional content of graphics courses [Angel and Shreiner 2011]. Section 7.3 provides a brief discussion on how this approach compares with our proposed blueprint. Fink *et al.* [Fink et al. 2012] suggest a shader-focussed approach to building progressive assignments in one course. However their assignments involve completing a significant implementation provided to students, whereas most of our assignments are built by students from scratch. Gearing graphics teaching and programming to specific audiences has also been investigated [Cunningham 2000; McDonald and Luecking 2002; Linares-Pellicer et al. 2010]. Another interesting approach has been to teach graphics in a completely *applied* way, using it as a means to complete activities [Anderson and Peters 2009]. Techniques focusing on classroom teaching rather than curriculum in graphics have also been proposed (Web-based [Klein et al. 1998], interactive tools [Schweitzer et al. 2011] and rapid prototyping [Gómez-Martín and Gómez-Martín 2006]).

Much work on computer graphics education recommends that hierarchical representations in general and scene graphs in particular be a part of traditional CG courses [Cunningham 1999; Wolfe 2000; Bouvier 2002b; Cunningham and Bailey 2001]. Bouvier [2002a] suggests using scene graphs to understand the composition of a 3D scene, using them to compose specified 3D environments, familiarizing the use of scene graph APIs and implementing them (recommended for graduate courses). Our course incorporates all these aspects in our latest set of assignments.

Implementing ray tracers in an introductory undergraduate graphics course is relatively uncommon [Wolfe 2000], although there are some examples [Gribble 2008]. However some work has been done to use ray tracing as a *teaching tool* in such courses [Hu 2010]. Shirley *et al.* [2007] discuss the how CG courses in the future may adapt to advances in interactive ray tracing, a topic we briefly allude to when talking in class about advances in and other applications of ray casting.

Some results shown in this paper were presented as a student project [Jones and Shesh 2013]. This paper discusses the details and the course itself.

3. THE COURSE: OVERVIEW AND EVOLUTION

This course is the first and only undergraduate course in computer graphics offered at our mid-sized university in the United States. It is offered once a year and requires relevant courses in data structures and algorithms (core courses) and basic knowledge of linear algebra. Students taking this course are typically in the third or fourth year of their 4-year computer science degree program. Most students taking this course have no prior experience in graphics. Table I summarizes the typical topics and time line of this course. The course uses C++ with OpenGL for all its programming projects.

We taught the same course for five semesters using different assignments each time, culminating in the proposed set of assignments that we are currently using. In all semesters we encouraged students to create their own models for the ray tracer. Also, all ray tracers worked with implicit shapes, not triangle meshes. Each semester began with one assignment in 2D graphics to ease students into 3D graphics.

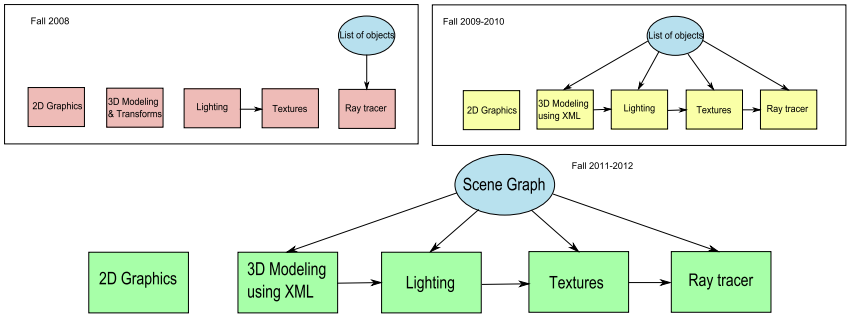


Fig. 1. The three varieties of assignments that we tried over five semesters. Arrows show that the next assignment built upon the previous one. Version 1 (red) used largely independent assignments, culminating in the ray tracer using a list of objects representation. Version 2 (yellow), used for 2 semesters included more progressive assignments. The program used for modeling and viewing was extended to support lighting, textures and finally ray tracing, all using a common list of objects read from XML files. Version 3 (green) is our proposed pipeline using scene graphs. Similar to version 2 it progressively builds a complete graphics application. However because of the scene graph representation, more complicated models and their animations were produced by students in the same time as corresponding assignments in Version 2.

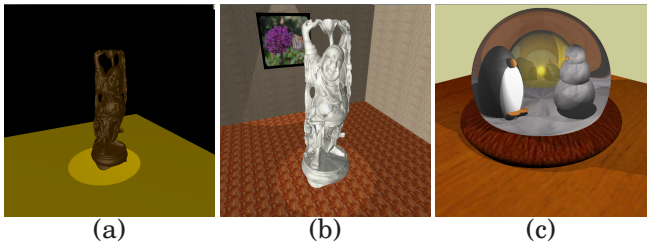


Fig. 2. Example outputs of various assignments for the 1^{st} semester (Fall 2008) with mostly unrelated assignments with no standard input file format (except the ray tracer). (a) Students had to create a "museum" room model with a triangle mesh. (b) Textured scene from (a) by determining texture coordinates for the Happy Buddha model. (c) Example output from student's ray tracer.

3.1. First Semester

In the first semester (red in Figure 1), we emphasized on individual operations such as modeling, lighting and textures. The first assignment on 3D graphics asked students to create and render an animated solar system consisting of spheres and orbits, and transform it using a track-ball interface. The next two assignments wrote programs to load planes and mesh models and enable lighting (Figure 2(a)) and textures (Figure 2(b)). Finally the last assignment asked students to write a ray tracer that used a list of implicit shapes (planes and spheres) as the primary data structure (Figure 2(c) shows a student's work). This course presented two difficulties. First, since the ray tracer worked with a list of objects, building complex 3D models using only a linear arrangement of simple objects was cumbersome for students (scene graphs were discussed, but students did not work with them in assignments). Secondly, since the ray tracer project was not connected to any earlier assignments, the instructor had to provide a program that rendered a scene using OpenGL which students were asked to ray trace. This created some difficulties as students had to understand the given program before attempting to augment it.

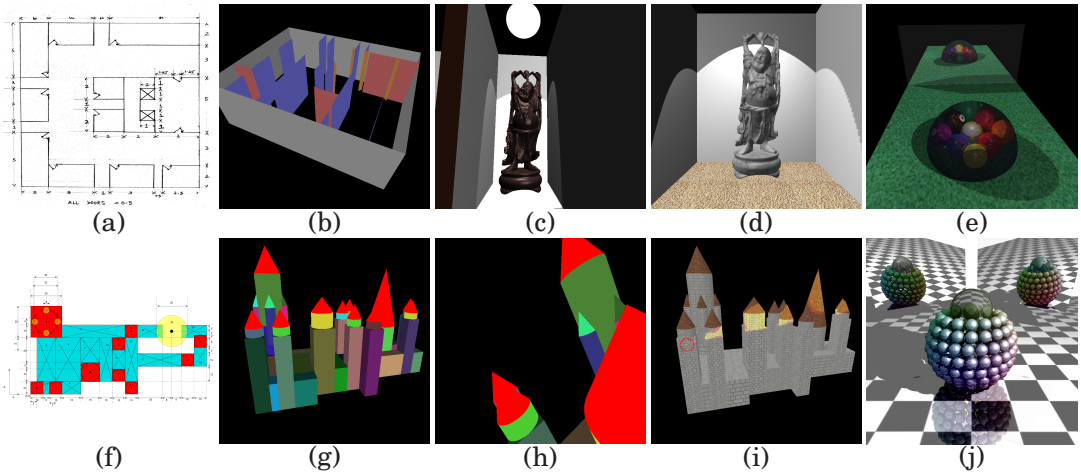


Fig. 3. Example outputs of various assignments for the 2^{nd} and 3^{rd} semesters, which had progressive assignments with XML-based input. Row 1 (2^{nd} semester, Fall 2009) (a) A floor plan specification provided to students (please zoom into the PDF document). (b) Corresponding model created by a student. (c) The model navigated using a keyboard based interface along with lighting. Students were asked to give the Happy Buddha model a "chocolate"-like appearance by choosing appropriate material properties. (d) Same model with textures. (e) Example output from a student's ray tracer using the same XML specification. Row 2 (3^{rd} semester, Fall 2010) (f) Specification to crudely replicate *Hogwart's School* from the *Harry Potter* movies (please zoom into the PDF document). (g) Example model created using the specification. (h) Same model rendered from the cockpit of an airplane model flying through the scene (i) Lit and textured model, with headlights on a moving airplane (encircled in red)(please see video at the link provided in Section 10). (j) Example output of a more complicated scene from a student's ray tracer.

3.2. Second and Third Semesters

In the second semester (yellow in Figure 1) we used XML to specify 3D models and designed assignments that progressively built upon each other. We also switched from GLUT [glu 2011] to Qt [qt 2011] as our windowing toolkit. Students were provided with the skeleton of an XML SAX[1] parser (written using Qt classes [qtx 2011]), and they were asked to extend it to parse a given XML file that specified various implicit shapes (i.e. planes, spheres, cylinders, etc.) with their colors and transformations. Using a provided floor plan (Figure 3(a)) students created a 3D scene (Figure 3(b)) using XML and implemented a keyboard-controlled camera. Subsequent assignments extended the XML specification to add support for lighting (Figure 3(c)), texturing (Figure 3(d)) and ray tracing capabilities (Figure 3(e) shows a student's work) to the same program. Thus students created a single application that parsed an XML file, rendered a 3D lit and textured navigable environment using OpenGL, and also switched to a ray tracing mode that rendered shadows, reflections and optionally, transparency.

The third semester followed a very similar pattern of assignments (a floor plan (Figure 3(f)) used to create a 3D scene (Figure 3(g))), but added the ability to load objects from different XML files simultaneously in a single scene. This somewhat eased the task of creating a 3D scene as it could be "assembled" from multiple XML files of small sizes. Students used this to add cockpit views to a plane flying along pre-defined paths (Figure 3(h), accompanying video at the link provided in Section 10) and object-relative lights (Figure 3(i)). Finally students added a ray tracer to this program (Figure 3(j)) that rendered shadows, reflections and optionally, transparency.

[1]Simple API for XML

In both semesters, students found it easier to specify more complicated models (see table in Figure 5(a)) using XML. The main limitation was the difficulty in animating parts of objects. This was because XML models were still stored as a list of objects in their programs, thereby destroying any semantic relationships between parts.

3.3. Fourth and Fifth Semesters: Proposed Blueprint

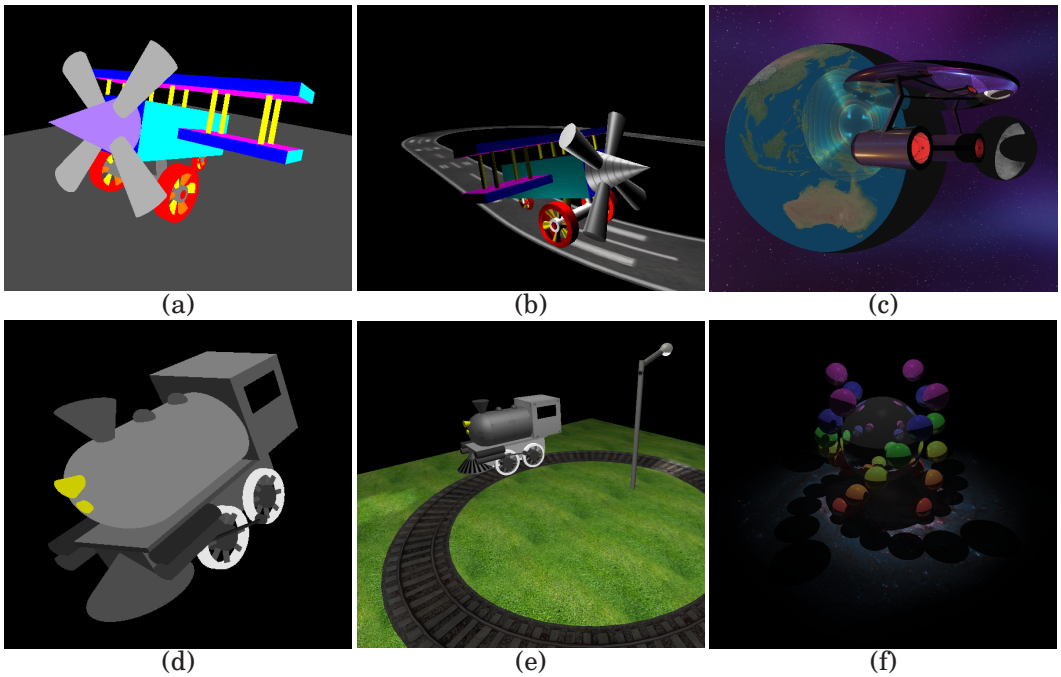


Fig. 4. Example outputs of various assignments for 4^{th} and 5^{th} semesters, with scene graph based modeling using XML input. Row 1: Fall 2011. (a) A hierarchical model of a wheeled object created using spheres, boxes, cylinders and cones. Students animated the model by moving wheels and turning rotors corresponding to the speed of the vehicle. (b) Same model lit and textured on a circular track. (c) Example output from a student's ray tracer using the same XML specification and scene graphs, including shadows, reflections and transparency. Row 2: Fall 2012. (d) Hierarchical model (e) Same model lit and textured with a headlight on a circular track (see accompanying video at the link provided in Section 10). (f) Example output from a student's ray tracer using the same XML specification and scene graphs.

The pipeline shown in green in Figure 1 shows our proposed course work, and the appendix provides a set of assignments. The major change from the earlier version was that students used scene graphs (as trees) to represent the virtual world, while largely retaining the same XML representation. Students extended an XML parser that created a list of objects so that it instead compiled a scene graph and then rendered it. They implemented the scene graph data structure by themselves from scratch. Students then used the program to create an animated wheeled locomotive (Figure 4(a,d), accompanying video at the link provided in Section 10). Along with lighting and texturing the scene (Figure 4(b,e)), subsequent assignments implemented a ray tracer based on the same scene graph with similar capabilities as before (Figure 4(c,f)).

The specific benefits of teaching scene graphs [Bouvier 2002b; Bouvier 2002a] and ray tracing [Hu 2010; Gribble 2008] are well-known in the graphics research and education community. We now correlate many objectives of a typical undergraduate graph-

ics course specified by the ACM curriculum [2008] with our specific material and highlight implementation details, benefits and opportunities for customization.

4. COURSE CONCEPTS ⇔ MATERIAL

4.1. Concept: Creation of 3D Models

ACM category: Geometric Modeling

3D models in computer graphics are typically represented using triangle meshes, points or volumes. Irrespective of their representation most practical 3D models are created progressively out of simpler primitives using 3D modeling software like 3D Studio Max [3ds 2012], Sketchup [ske 2012], etc. In order to reduce modeling complexity and represent not only structure but motion capabilities, such models are hierarchical, with every part composed internally of other parts. Learning how to specify and use models in a hierarchical fashion is useful for any student of graphics.

Scene graphs and XML both fit the above requirement perfectly. Being naturally hierarchical, it is convenient to specify hierarchical models within an XML specification. Structural (e.g. normal vectors, positions, transformations, etc.), motion (e.g. animating transformations, velocity, acceleration, etc.) and material (e.g. color, interaction with light, textures, etc.) properties can all be embedded within XML. Hierarchical XML specifications map into hierarchical scene graphs within the program. In our course students could create more complicated models by manually typing an XML file rather than a linear arrangement of the same primitives without any hierarchy. For example, the moving locomotive (Figure 4(a-b)) would be very cumbersome to create without a hierarchy, and whereas the Hogwarts model (Figure 3(b-c))) could be created with a simple collection of basic shapes, using hierarchies for each building could have made the task more convenient.

4.2. Concept: Transformations and Coordinate Systems

ACM category: Fundamental Techniques

Understanding the nature of transformations, their use to produce a desired output and the connection between the visual and the underlying math is arguably the most critical component of a computer graphics course. In our experience students take significant time in decomposing a desired animation or camera movement into transformations. Therefore we strive to give students a lot of practice using transformations by asking them to implement specific visual results throughout the course.

We start by asking students to create 3D models in XML, first linear (3D model from floor plan (Figure 3(a,f)) and then hierarchical (Figure 4(a-b))), thereby reinforcing the advantages of scene graphs over lists. These transformations are stored at various nodes in the resulting scene graph. We ask students to either animate models realistically or attach cameras to moving parts of the scene. Implementing these effects may seem complicated to the untrained mind, but representation of the model as a scene graph is very helpful in such situations.

The scene graph hierarchy is an n-way tree (i.e. each non-leaf node has up to n children) that implicitly represents each part (node) at the center of its own coordinate system. This coordinate system acts as a reference for all its sub-parts (children nodes). The part itself is placed relative to its "parent part" (parent node). Drawing any part correctly requires transforming it to the camera coordinate system which is the system of the scene graph root. Thus rendering a scene graph requires a simple depth-first traversal. More importantly, *any* relative motion of parts requires determining transformations from the coordinate system of the moving part to that in which it moves. The duality of transformations and coordinate system changes is a very helpful

concept, and practically is also one of the most confusing ones for students. Using a scene graph greatly aids in revealing this duality.

Although earlier semesters taught scene graphs, we believe the illustration and the learning power was much greater when students actually implemented them. We discovered that in many cases students were able to determine appropriate transformations, arrange them in the correct order more quickly when the models were already organized in a scene graph. A preliminary proof of their effectiveness is given by the fact that students were able to implement a scene graph data structure, create the 3D model of the moving locomotive in XML (without textures), and animate it with rotating wheels and fan blades along a circular path in a single assignment in only $2-3$ weeks while working individually.

Implementation of the ray tracer using the same scene graph further reinforces this understanding. Tracing rays through a scene requires transforming a ray up and down the scene hierarchy. More than the practical utility of implementing a ray tracer, we believe this project further solidifies students' understanding of 3D transformations and coordinate systems.

4.3. Concept: Lighting

ACM category: Basic Rendering

Besides transformations, lighting is the most mathematical concept in an introductory course. Most APIs like OpenGL provide support for basic lighting. However one cannot directly implement many effects using most available APIs (e.g. specify a moving light).

We believe using scene graphs earlier in the course helps students understand the concept of moving lights, because scene graphs help thinking of lights as geometry. Thus creating moving lights (e.g. a miner's hat or headlights on a moving object) becomes as simple as determining the appropriate scene graph node to attach them to (i.e. the appropriate coordinate system to specify the light).

While using an API like OpenGL hides students from the implementation of the lighting model, building a ray tracer forces them to implement it from scratch. Understanding how lighting works in a graphics environment is one of the main benefits of writing the ray tracer. Students are typically asked to implement "per-pixel lighting", an effect that cannot be produced in OpenGL without writing GPU shaders.

4.4. Concept: Texture mapping

ACM category: Basic Rendering

Texture mapping is the task of pasting images onto 3D models to give them a realistic appearance. Most existing APIs support this operation and basic related operations. However the ray tracer project asks students to implement texture mapping from first principles. Thus students are exposed to some basic but important math that APIs successfully abstract from programmers. Knowledge of how such basic operations work is critical when customizing them using GPU shader programs.

4.5. Concept: Ray Tracing

ACM category: Advanced Rendering

OpenGL implements a "pipeline"-based rendering model. A ray tracer application exposes students to a radically different way of producing realistic images. Progressively adding both forms of rendering to the same application helps further reinforce and appreciate the similarities and differences between the two approaches.

Implementing a ray tracer gives students a chance to implement the actual math that they learn in the course and create pictures without the help of an existing API. Also it allows students to achieve effects that are not supported directly by traditional

APIs (e.g. shadows). Students implement realistic shadows, reflections and (optionally) transparency, which are conceptually much simpler in a ray tracer than in pipeline-based APIs. Contrasting the OpenGL lighting model with that of the ray tracer often leads to a discussion on other rendering methods such as photon mapping.

4.6. Concept: Hidden Surface Removal

ACM category: Advanced Techniques

Hidden-surface removal is directly supported by most APIs, mostly in the form of Z-buffers. However students effectively implement this technique in the ray tracer project, by sorting all points of intersection along a ray from the camera to pick the nearest object. Although other techniques for hidden surface removal like BSP trees are no longer practically used for this operation, they find other applications in the ray tracer. Including the ray tracer in our course provides a context to discuss the possible role of BSP trees and other data structures to make it more efficient[2]. We feel going through the hardships of implementing a ray tracer that inherently works slowly compared to earlier programs underscores the need to think about other data structures and algorithms.

5. INTEGRATION WITH CONCEPTS FROM OTHER COURSES

Our choice of the above topics and implementation details helped us to expose students to small, but new concepts and tools.

XML is widely used in many applications, but is seldom part of a traditional programming course. XML parsers (SAX and DOM) are common components in many applications, and students in our course benefited from using XML in the context of graphics. We observed a few instances of students, having taken the graphics course, using similar XML parsers with Qt in other courses that we taught. We regard this as small but preliminary evidence that the knowledge gained from the graphics course proved useful elsewhere.

Constructing a scene graph using XML proves to be a worthwhile exercise in algorithms. Students were asked to use provided hints and their understanding of trees to implement this bottom-up construction themselves. The actual algorithm is very similar to construction of expression trees [Weiss 2006], a common example used to illustrate binary trees. Scene graphs also exemplify "n-way" trees, a topic that students learn about but often do not actually implement in algorithms courses. Rendering and destroying a scene graph amounting to a pre-order traversal was a revelation to many students who had encountered this operation only in the context of search trees.

An indirect benefit of implementing a single application incrementally is that students get a chance to observe many design issues specifically concerning graphics applications. In general it is a challenge to provide students with the experience of analyzing, designing, implementing and testing the same application with practical utility. Most curricula start with programming followed by courses in software analysis and design that involve partial or no implementation. A progressively built system in a CG course represents a reverse approach: implementing and testing a system with a provided design. We found the component-wise design to be especially beneficial when augmenting the program with a ray tracer, making it possible to implement the ray tracer over a period of $3-4$ weeks.

During every semester, we found that several students were simultaneously enrolled in or had recently taken physics courses that included optics. These physics courses and our graphics course seemed to nicely but unexpectedly reinforce each other with

[2]Students did not implement any data structures to make the ray tracer efficient; they were just discussed.

Table II. Qualitative assessment of required tasks assigned to students, categorized into *modeling*(Rows 1-4), *lighting and texturing*(Rows 5-8) and *ray tracing*(9-12). Grey columns highlight semesters with proposed assignments. Table outlines various common graphical effects and details of whether they were assigned in one or more assignments in the five semesters. Later semesters included more sophisticated effects (hierarchical modeling, moving lights, ray tracing scene graphs).

Effect/Task	Fall 2008	Fall 2009	Fall 2010	Fall 2011	Fall 2012
Creating scenes from scratch	Simple	Simple	Simple	Hierarchical	Hierarchical
Load models from files	Code provided	Code provided	Partly write XML parser	Partly write XML parser	Partly write XML parser
Object animation/movement	Whole	No	Whole	Whole and part	Whole and part
Camera movement	No	Keyboard-controlled	Pre-defined path	No (equivalent scene movement)	Pre-defined path
Placing lights	Stationary	Stationary	Stationary and/or moving	Stationary and/or moving	Stationary and/or moving
Render pre-textured objects	Yes	Yes	Yes	Yes	Yes
Determine texture coordinates	Yes	Yes	Yes	Yes	Yes
Implement given lighting model	Yes	Yes	Yes	Yes	Yes
Ray casting in 3D	List of objects	List of objects	List of objects	Scene graph	Scene graph
Shadows	Yes (ray tracer)	Yes (ray tracer)	Yes (ray tracer)	Yes (ray tracer)	Yes (ray tracer)
Reflections	Yes (ray tracer)	Yes (ray tracer)	Yes (ray tracer)	Yes (ray tracer)	Yes (ray tracer)
Refraction	Optional (ray tracer)	Optional (ray tracer)	Optional (ray tracer)	Optional (ray tracer)	Optional (ray tracer)

students learning about the theory in the former and actually implementing it in the latter (usually in the ray tracer).

6. IMPACT ON STUDENT LEARNING AND PERFORMANCE

We believe that our proposed set of assignments and their objectives improve the overall design of a singleton graphics course. Specifically we claim the following: (1) students who completed our proposed set of assignments achieved more sophisticated results in the same time frame (2) students achieved this without negatively affecting their performance in the course, and (3) students report a comparable or higher level of satisfaction when our course offers the proposed assignments. We support these claims by presenting an elementary analysis of student evaluations, student performance and a qualitative assessment of student achievements. This analysis uses data that is routinely available to instructors.

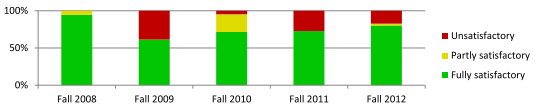
6.1. Qualitative Observation of Student Achievement

The primary way in which we assess the impact of our projects is to observe the complexity of results achieved by students. In all semesters students were encouraged to create their own models and render them in at least some of the assignments.

Our primary objective for qualitative observation was to observe which "practical" graphical data structures, operations and effects our students able to implement (reasonably) from scratch. Table II summarizes how assignments became more sophisticated in later semesters, by enumerating a categorized list of required tasks/effects in

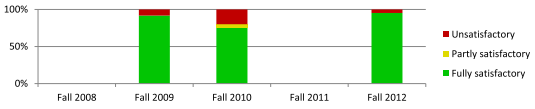
various assignments and how (or whether) they were implemented by students. With respect to *modeling*, Table II shows how Fall 2011-2012 included hierarchical modeling using scene graphs, leading to more complicated models, and more advanced animation effects through either transformations or equivalent camera movements. This is further supported by the table accompanying Figure 5(a) that compares the complexity of models across semesters, and visually illustrated in Figures 2- 4. With respect to *lighting*, Fall 2011-2012 included lights that were stationary or moved with objects). Finally Fall 2011-2012 included ray casting with scene graphs, enabling students to ray trace potentially more complicated 3D scenes.

Since students individually completed corresponding assignments each semester in largely the same time($\sim 2-3$ weeks), it implies that students could achieve better graphical results due to being able to create more complex models and/or implement more complicated effects like animation and moving lights in the same amount of time.

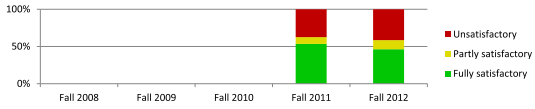


Average no. of primitives objects used in a scene created by students				
Fall 2008	Fall 2009	Fall 2010	Fall 2011	Fall 2012
6 objects	57.4 objects	51.4 objects	32.3 objects, 7.7 groups	38.3 objects, 8.8 groups

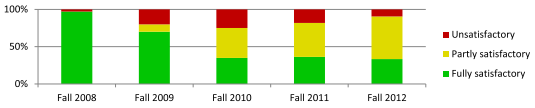
(a) Create scenes from scratch as specified



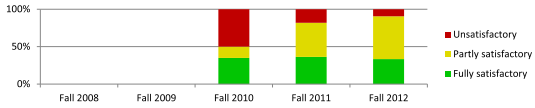
(b) Move camera as specified



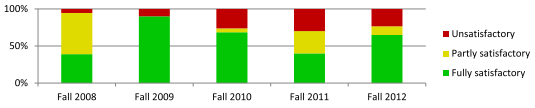
(c) Move parts of objects as specified



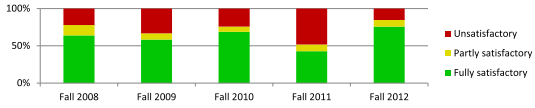
(d) Place stationary lights as specified



(e) Place moving lights as specified

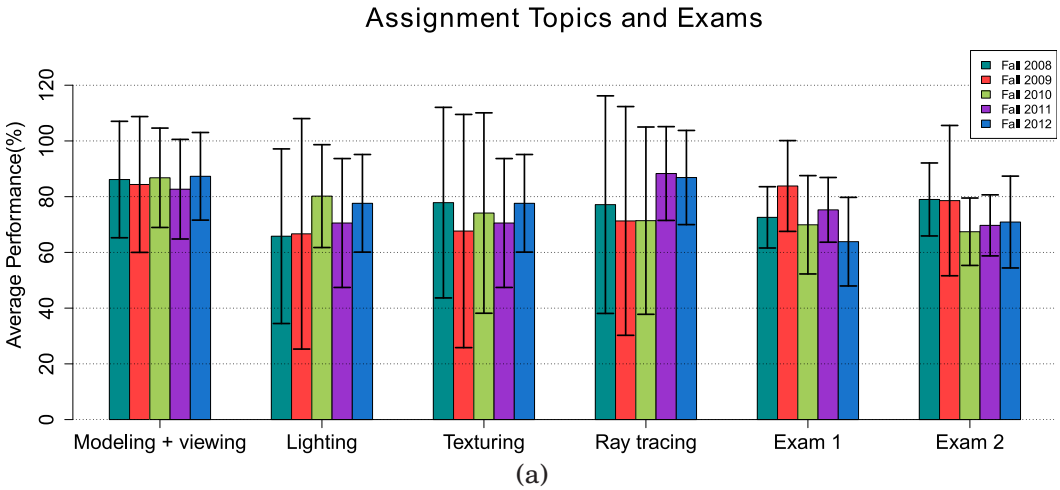


(f) Implement OpenGL shading model



(g) Texture-map objects with images

Fig. 5. Assessment of student work according to various required criteria. Please see Section 6.2 for details on how these charts were produced. Charts show how student performance was either similar or improved as the tasks become more complex with every semester. Blanks in each chart indicate that the task was not assigned for that semester (either unnecessary or infeasible). (a) Task: create scenes from scratch as specified. Table shows the average number of primitive objects (plane, sphere, cone, cylinder) per student scene and the average number of groups (only for hierarchical models). (b) Task: move camera as specified. Fall 2008 did not include this task, while Fall 2011 completed an equivalent task involving moving the scene. (c) Task: move parts of objects as specified. Such animation was not feasible in the first three semesters as models were not hierarchical. (d) Task: Place stationary lights as specified. The last two semesters did not *explicitly* specify this task as stationary and moving lights can be handled in the same way. (e) Task: Place moving lights as specified. This task entailed attaching lights to moving objects (e.g. headlights). (f) Task: Implement the existing OpenGL shading model manually (in the context of the ray tracer). This task forces students to understand the math behind OpenGL lighting. (g) Task: render objects that are texture-mapped. This task also involved determining texture coordinates for some primitive objects in the ray tracer.



(a)

Pairwise t-tests

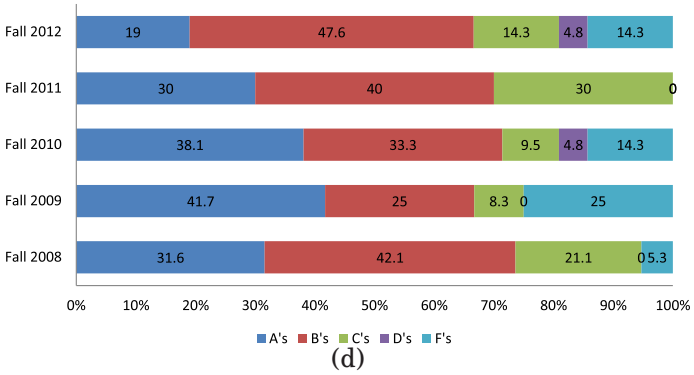
Comparison	Topic	t	Pr(T≤t) (2-tailed)
Fall 2008 vs. Fall 2011	Modeling	0.4862	0.6313
	Lighting	-0.4849	0.6317
	Texturing	0.7159	0.48
	Raytracing	-1.1106	0.2759
Fall 2008 vs. Fall 2012	Modeling	-0.2	0.8426
	Lighting	-1.5075	0.1417
	Texturing	0.0284	0.9775
	Raytracing	-1.039	0.3078

(b)

Analysis of Variance: 2-sided ANOVA

Criterion	Df	F value	Pr (>F)
Topic	3	3.7969	0.0106
Semester	4	0.9997	0.407
Topic:Semester	12	0.6010	0.841

(c)



(d)

Pearson's χ^2 test

Comparison	Df	χ^2	p-value
Fall 2008 vs Fall 2011	3	0.7559	0.86
Fall 2008 vs Fall 2012	4	2.6718	0.6142

(e)

Fig. 6. Performance comparison across five semesters. (a) Average performance as a trend per deliverable (topic/exam), with error (standard deviation). (b-c) Analysis of significance of average performance. (b) Pairwise t-tests on Fall 2008 and Fall 2011-2012 shows that the performance before and after the proposed assignments did not vary significantly. (c) Results of the 2-sided ANOVA test on actual scores, showing that the performance across the 5 semesters (row 2) did not vary significantly for any topic, nor was there any significant correlation between the semester and deliverable on specific topics. (d) Percentage of letter grades per semester. (e) Results of the χ^2 test showing that the grade distributions are independent (p-value $>$ 0.05) of the types of assignments offered (Fall 2008 vs. Fall 2011 and Fall 2012). Overall this shows that student performance was not affected significantly as a result of adopting the proposed assignments.

6.2. Student Performance

We assessed the possible impact of our assignments on student learning by observing average student performance. In the five semesters in question (Fall 2008-2012) had 19, 12, 21, 12 and 23 students respectively, which is fairly representative of our class sizes.

Figure 5 shows student performance using various expected outcomes of a graphics course as criteria. For each criterion, we mapped relevant parts of (possibly multiple) assignments for each of the 5 semesters and calculated the average student performance in those parts (using points pre-assigned by the instructor in the grading rubrics) as a percentage. We then mapped this average to categories as follows: *Fully satisfactory* ($\geq$ 80%), *Partly satisfactory* (between 60% and 80%) and *Unsatisfactory*($<$ 60%). Blank regions indicate that the criterion in question either could not be implemented (e.g. animation of parts in earlier semesters) or were not explicitly required (e.g. moving the camera in Fall 2011 was indirectly required by moving the scene in a specific way).

Figure 6(a) shows the average performance of the class in exams and assignments (actual units of graded course work) that broadly span the topics of *modeling and viewing*, *lighting*, *texturing* and *ray tracing*. Table 6(b) shows, using pairwise t-tests [Kutner et al. 2004], that there was no significant change in average performance before and after the proposed assignments. Table 6(c) reports the results of the 2-sided ANOVA test [Kutner et al. 2004] on the actual scores, showing that the average performance did not vary significantly across semesters (row 2), and there was no significant interaction between topics and semesters. Figure 6(c) shows the percentage of letter grades awarded in each semester. Table 6(d) shows that according to the Pearson's χ^2 test the grade distributions before and after adopting the assignments are independent of each other (p-value greater than 0.05). Thus we conclude that assigning increasingly complex tasks in later semesters varied slightly or retained the overall average performance.

6.3. Student Perception

All courses in our school are evaluated by students using IDEA evaluations [ide 2012], that report statistics based on questions that aim to assess progress towards course objectives, performance of the instructor and student perception of the course. Table III summarizes student responses to six questions related to (a) completion of stated course objectives, (b) course work expected from students and (c) an overall rating of the course. This table compares student responses from Fall 2008 (separate assignments, no scene graphs), Fall 2011 and Fall 2012 (both semesters with proposed set of assignments and scene graphs). Although student perception is traditionally not considered as the most reliable tool to assess curriculum and course effectiveness, we observe that students rated the latest forms of this course higher than the first one. The lower table in Table III presents the results of the Welch t-test [Kutner et al. 2004] on the response data before (Fall 2008) and after (combined data from Fall 2011 and Fall 2012) adopting our proposed assignments to determine if the difference in student evaluations is significant. These results show that the differences are significant for 4 out of the 6 above questions. These student perceptions combined with inferences from earlier qualitative assessments provide some evidence that our latest set of assignments achieved more positive outcomes.

We have found it difficult to reliably assess how our assignments have affected students' holistic understanding of computer science concepts. This is primarily because of two reasons. First, it is difficult to map these benefits to actual material in courses students take subsequently, thereby hindering a cause-effect assessment. Secondly as stated earlier many students graduate soon after taking the CG course. Nevertheless

Table III. Student perceptions of our course from IDEA evaluations when taught in Fall 2008 (no progressive assignments, no scene graphs) and combined data from Fall 2011-2012 (early-scenegraphs, progressive assignments). Top table: raw scores given by students. All scores are on a 5-point scale, 5 being the highest rating. Bottom table: statistical significance of the difference between the two sets of student evaluations (Fall 2008 and combination of Fall 2011-2012) using Welch t-test. These results show that the differences are significant ($P(T <= t) <= 0.05$ and $\|t_{critical}\| <= \|t^*\|$, shown in bold) for Questions 1, 2, 3 and 6.

Question	Fall 2008 (before changes)						Fall 2011 (after all changes)						Fall 2012 (after all changes)					
	Number of responses					Avg.	Number of responses					Avg.	Number of responses					Avg.
	1	2	3	4	5		1	2	3	4	5		1	2	3	4	5	
1. (Progress on) Gaining factual knowledge (terminology, classifications, methods, trends)	0	1	3	6	6	**4.1**	0	0	0	4	7	**4.6**	1	0	0	4	13	**4.6**
2. (Progress on) Learning fundamental principles, generalizations or theories	0	1	4	5	6	**4.0**	0	0	0	5	6	**4.6**	1	0	0	5	12	**4.5**
3. (Progress on) Learning to apply course material (to improve thinking, problem solving and decisions)	1	0	4	7	4	**3.8**	0	0	0	4	7	**4.6**	1	0	3	5	9	**4.2**
4. (The instructor) gave tests, projects, etc. that covered the most important points of the course	0	2	0	7	7	**4.2**	0	1	1	2	7	**4.4**	1	0	2	6	9	**4.2**
5. (The instructor) gave projects, tests or assignments that required original or creative thinking	0	1	4	3	8	**4.1**	0	0	0	3	8	**4.7**	1	1	0	5	11	**4.3**
6. Overall I rate this course as excellent	1	0	3	9	3	**3.8**	0	0	0	4	7	**4.6**	1	0	1	4	12	**4.4**

Question	1	2	3	4	5	6
t^*	-1.8831	-1.806	-1.6942	-0.2839	-1.1504	-2.3987
$P(T <= t)$ (one-tail)	**0.0351**	**0.041**	**0.0506**	0.3891	0.1297	**0.012**

we have observed some anecdotal evidence of its impact. From our discussions, it is apparent that many of our students who have dabbled in writing games found knowledge of XML processing and scene graphs very helpful. We have found some instances of students using the XML processing learned here in other courses.

7. SUGGESTIONS FOR CUSTOMIZATION

Our evolving blueprint that includes scene graphs, progressive assignments and ray tracing can be customized in several ways to offer courses with a different or specific focus and targeting different audiences.

7.1. Scene graphs: Implement or Use?

Many modern game engines (e.g. Unreal [unr 2011]) and libraries (e.g. Open Scene Graph [ope 2011]) include scene graphs with support for multiple programming languages. This course can be designed to concentrate on the use of scene graphs rather than their construction. While we feel asking students to implement a scene graph data structure has significant learning benefits, adopting an existing implementation will allow more room to concentrate on other relevant topics, such as view-frustum culling and collision detection using bounding volume hierarchies for use in games [Eberly 2001].

7.2. Focus on specific applications

Many small and medium-sized colleges offer specific courses that use computer graphics for applications, such as art and design or computer games. Scene graphs support a wide variety of operations concerning modeling, animation or rendering. A course may be customized to focus on any technique while retaining our overall blueprint. For example a course may emphasize on creating an interactive modeling package where the scene graph representation can be used to facilitate user interactions for various operations. Alternatively an animation-based package may be progressively constructed that uses scene graphs to allow interactive user input for animating scenes. The ray-tracing component may either be used to create a "demo picture" or used for object-picking within the application rather than actual rendering.

7.3. Shader Programming

Shader programming using GLSL or Cg is an increasingly popular topic in undergraduate graphics courses. Availability of inexpensive GPUs and the ubiquitous use of shaders in graphics programming are compelling reasons to integrate shader programming in a *singleton* graphics course. Including shader programming simply as a separate module (like ray tracing) especially in a *singleton* course such as ours has several difficulties. First, it is difficult to include *both* shader programming and ray tracing in a semester without compromising on other content due to lack of time. Secondly, shader programming uses a different programming model (Single Instruction Multiple Data or SIMD) with a different set of development and debugging tools and techniques than what students are normally exposed to, making it difficult to do justice to this topic in a short span of time. Finally, as Angel *et al.* [Angel and Shreiner 2011] report, the modern OpenGL standard has deprecated much of the traditional pipeline and many standard mathematical functionality. Although they suggest how instructors can use this to their advantage, it may add to the "startup" time of a *singleton* course (i.e. the amount of new concepts and programming that students must be familiarized with before they implement something meaningful and substantial).

Our primary reason for choosing ray tracing was that it better reinforces all the basic mathematical and graphical concepts by essentially re-implementing the graphics pipeline. A minor secondary reason was that although our department offers laboratories equipped with modern GPUs, many students did not have access to them off-campus where they chose to complete most course work.

There have been some recent advances in introducing shaders early in the graphics course, such as the new edition of the popular textbook by Angel [Angel 2011]. We believe their approach of *integrating* shaders with traditional graphics programming throughout the course rather than as a *separate* module can be combined with our proposed blueprint. As such we plan to adopt this approach as a "shaders-first,early-scenegraphs" introductory graphics course that would retain ray-tracing. In such a version, we envision that shaders would be considered as an integral and mandatory part of any initial source code given to students. Throughout the course, students can then be asked to modify/add shaders that produce certain effects, similar to how they currently add parts to existing programs. For example, students can be asked to implement the same effect two ways (traditional vs. shader-based) to highlight the power and potential of shaders. Since most students taking our course have no prior experience in graphics programming in particular, we believe there is a viable way of integrating shader programming without sacrificing much content, in the way illustrated by Angel [Angel 2011].

Table IV. Image credits

Images	Name of student
Figure 2(c)	Christopher Olson
Figure 3(b,c,d)	Benjamin Ritter
Figure 3(e)	Clint Riley
Figure 3(g,h,i)	Sarah Steffen
Figure 3(j)	Christopher Murphy
Figure 4(a,b)	Geoffrey Godwin
Figure 4(c)	Thomas Mazzotti
Figure 4(d,e,f)	Matthew Jones

8. CONCLUSION

We have discussed the evolution of an introductory CG course over five semesters, culminating in a progressively built scene-graph-based application that uses XML for input and includes a ray tracer. We believe this content and related projects effectively reinforce graphics concepts and provide students with ample experience in graphics implementation in a single course, preparing them for further study of computer graphics or pursuing projects such as games.

9. ACKNOWLEDGMENTS

Credits for images produced by students are enumerated in Table IV. We would like to thank James Wolf for his help in the statistical analysis of our course and student grade data.

10. SUPPLEMENTARY MATERIAL

The videos showing some student results for this paper can be viewed at https://drive.google.com/folderview?id=0B-7vVRZV2KC0WWdpakFfS21KQkk&usp=sharing

REFERENCES

2008. ACM CS Curriculum. (2008). http://www.acm.org/education/curricula/ComputerScience2008.pdf.

2011. The OpenGL Utility Toolkit. (2011). http://www.opengl.org/resources/libraries/glut/.

2011. OpenSceneGraph. (2011). http://www.openscenegraph.org/.

2011. QT: A Cross-Platform UI Framework. (2011). http://qt.nokia.com/products/.

2011. The QT XML Module. (2011). http://doc.qt.nokia.com/5.0-snapshot/qtxml.html.

2011. UnReal Technology. (2011). http://www.unrealengine.com/.

2012. 3D Studio Max. (2012). http://usa.autodesk.com/3ds-max/.

2012. IDEA Diagnostic Form Report. (2012). http://www.theideacenter.org/sites/default/files/InterpretativeGuideDiagForm.pdf.

2012. Trimble Sketchup. (2012). http://sketchup.google.com/.

Eike Falk Anderson and Christopher E. Peters. 2009. On the Provision of a Comprehensive Computer Graphics Education in the Context of Computer Games: An Activity-Led Instruction Approach. In *Proc. Eurographics (Education Papers)*. 7–14.

E. Angel. 2011. *Interactive Computer Graphics: A Top-down Approach with Shader-Based OpenGL* (sixth ed.).

Edward Angel, Steve Cunningham, Peter Shirley, and Kelvin Sung. 2006. Teaching computer graphics without raster-level algorithms. In *Proc. SIGCSE*. 266–267.

Ed Angel and Dave Shreiner. 2011. Teaching a Shader-Based Introduction to Computer Graphics. *IEEE Computer Graphics and Applications* 31, 2 (2011), 9–13.

Dennis J. Bouvier. 2002a. Assignment: scene graphs in computer graphics courses. In *ACM SIGGRAPH 2002 conference abstracts and applications (SIGGRAPH '02)*. 42–45.

Dennis J. Bouvier. 2002b. From pixels to scene graphs in introductory computer graphics courses. *Computers & Graphics* 26, 4 (2002), 603–608.

Jack Bresenham, Cary Laxer, John Lansdown, and G. Scott Owen. 1994. Approaches to teaching introductory computer graphics. In *Proc. SIGGRAPH*. 479–480. Chairman-Larrondo-Petrie, Maria M.

Steve Cunningham. 1999. Re-inventing the introductory computer graphics course: Providing tools for a wider audience. In *Proc. Graphics and Vis. Ed. Workshop (GVE)*. 50.

Steve Cunningham. 2000. Powers of 10: the case for changing the first course in computer graphics. In *Proc. SIGCSE*. 46–49.

Steve Cunningham and Michael J. Bailey. 2001. Lessons from scene graphs: using scene graphs to teach hierarchical modeling. *Computers & Graphics* 25, 4 (2001), 703–711.

Steve Cunningham, Judith R. Brown, Robert P. Burton, and Mark Ohlson. 1988. Varieties of computer graphics courses in computer science. *SIGCSE Bull.* 20, 1 (1988), 313–313.

David H. Eberly. 2001. *3D game engine design - a practical approach to real-time computer graphics*. Morgan Kaufmann.

Heinrich Fink, Thomas Weber, and Michael Wimmer. 2012. Teaching a Modern Graphics Pipeline Using a Shader-based Software Renderer. In *Proc. Eurographics (Education Papers)*.

Pedro Pablo Gómez-Martín and Marco Antonio Gómez-Martín. 2006. Fast application development to demonstrate computer graphics concepts. In *Proc. ITICSE*. 250–254.

Christiaan P. Gribble. 2008. Ray Tracing for Undergraduates. *ASEE Computers in Education Journal* 18 (2008).

Scott Grissom, Bill Kubitz, Jack Bresenham, G. Scott Owen, and Dino Schweitzer. 1995. Approaches to teaching computer graphics (abstract). *SIGCSE Bull.* 27 (Mar. 1995), 382–383. Issue 1.

Lewis E. Hitchner, Steve Cunningham, Scott B. Grissom, and Rosalee Wolfe. 1999. Computer graphics: the introductory course grows up.. In *SIGCSE'99*. 341–342.

Helen H. Hu. 2010. Teaching introductory computer graphics via ray tracing. *J. Comput. Sci. Coll.* 26, 2 (2010), 30–38.

Matthew Jones and Amit Shesh. 2013. Scene Graph Creation And Management For Undergraduates. In *Proc. Eurographics (Education Papers)*. 11–12.

Reinhard Klein, Frank Hanisch, and Wolfgang Straßer. 1998. Web-based teaching of computer graphics: concepts and realization of an interactive online course. In *SIGGRAPH Conference abstracts and applications*. 88–93.

Michael Kutner, Christopher Nachtsheim, John Neter, and William Li. 2004. *Applied Linear Statistical Models* (5 ed.). McGraw-Hill/Irwin.

Jordi Linares-Pellicer, Empar Carrasquer-Moya, Javier Esparza-Peidro, and Pau Mic-Tormos. 2010. Computer Graphics for Information System Programmers. In *Proc. Eurographics (Education Papers)*. 57–62.

John McDonald and Stephen Luecking. 2002. Three Alternatives for an Introductory Computer Graphics Sequence. In *Proc. CGE*. 69–78.

G. Scott Owen. 1992. Teaching computer graphics using RenderMan. In *Proc. SIGCSE*. 304–308.

Eric Paquette. 2005. Computer Graphics education in different curricula: analysis and proposal for courses. *Computers & Graphics* 29, 2 (2005), 245 – 255.

Dino Schweitzer, Jeff Boleng, and Paul Graham. 2010. Teaching introductory computer graphics with the processing language. *J. Comput. Sci. Coll.* 26, 2 (2010), 73–79.

Dino Schweitzer, Jeff Boleng, and Lauren Scharff. 2011. Interactive tools in the graphics classroom. In *Proc. ITiCSE*. 113–117.

Peter Shirley, Kelvin Sung, Erik Brunvand, Alan Davis, Steven Parker, and Solomon Boulos. 2007. Rethinking graphics and gaming courses because of fast ray tracing. In *Proc. SIGGRAPH 2007 educators program*. Article 15.

Kelvin Sung and Peter Shirley. 2003. A top-down approach to teaching introductory computer graphics. In *Proc. SIGGRAPH 2003 Educators Program*. 1–4.

Romero Tori, João Luiz Bernardes, Jr., and Ricardo Nakamura. 2006. Teaching introductory computer graphics using java 3D, games and customized software: a Brazilian experience. In *Proc. SIGGRAPH 2006 Educators program*. Article 12.

Mark Allen Weiss. 2006. *Data Structures and Algorithm Analysis in C++* (third ed.). Addison Wesley, 121–123.

Rosalee Wolfe. 2000. Bringing the introductory computer graphics course into the 21st century. *Computers & Graphics* 24, 1 (2000), 151–155.

Received June 2012; revised January 2013; accepted August 2013

Online Appendix to: Towards a Singleton Undergraduate Computer Graphics Course in Small and Medium-sized Colleges

AMIT SHESH, Illinois State University

A. PROPOSED ASSIGNMENTS

The following is an abridged description of the proposed assignments, as they were assigned in Fall 2012. During the course students were provided with source code in C++ with many of the assignments. This appendix does not include the source code but refers to it.

A.1. 1. 2D Graphics

Description: Turtle graphics is based on the notion of a turtle moving on screen, drawing as it moves. The turtle's "state" is described by a position (x,y) and a direction in which it is pointing, in the form of a unit vector (v_x,v_y). Through standard commands, you can make the turtle simply walk, walk and draw, and change its direction.
What to do: Implement a program that accepts a set of turtle commands from a file and correspondingly moves the turtle and makes it draw. The starting state for the turtle is at the origin pointing in the +X direction. The following are the turtle commands that your program must implement: *turn* θ that turns the turtle to its left by angle θ in degrees, *trace* r that moves the turtle in the direction it is facing by distance r and draw its path as a line, *move* r that moves the turtle in the direction it is facing by distance r but does not draw, *push* that saves the current state of the turtle in a stack, and *pop* that retrieves the state of the turtle from the stack.
What is provided: Example text files with turtle commands, a short description of their format and the expected picture for one simple input file.

A.2. Modeling

Description: In this assignment you will implement a program that reads and draws simple 3D objects constructed from simple primitives. The provided C++ classes represent and draw simple primitives (bounded XY plane, sphere, cylinder). Please refer to documentation in these files to know more about their default sizes and positions. Create a new project with the provided source files and verify that the pictures shown below (in the actual assignment) are rendered for the corresponding files shown.

The XML file describes an object composed of the above primitives. The XML file represents each primitive, its transformations and color as XML tags. The provided XML parser reads the input XML file and if valid, creates an array of objects that are drawn by the main rendering function.
What to do: Create two new primitives (a) a cone with its circular face of unit radius centered at the origin and on the X-Z plane, and height 1 along +Y axis (b) a box of unit side centered at the origin (you may use several planes to create the box or create it from scratch. Verify that these primitives work correctly by trying the provided input files that contain them.

Model a locomotive using the above 5 primitives. You are free to design its structure, so long as it obeys 4 constraints: (1) The locomotive must have 4 wheels (2) Each wheel must have at least four spokes (3) The locomotive must have a body (i.e. not just four

DOI: http://dx.doi.org/10.1145/0000000.0000000

wheels and axles) (4) The locomotive must have at least one propeller with at least 2 blades.
What is provided: C++ source code that creates and draws planes, spheres and cylinders in a canonical position, example input XML files, XML parser that parses provided input files into a list of primitive objects, an image of an example locomotive.

A.3. Hierarchical modeling and animation

Description: A scene graph is a hierarchical representation of a model. It contains two types of nodes: leaf nodes that represent an actual primitive and group nodes that group several sub-groups or primitives and transform them together. Each node has its coordinate system and stores a transformation that transforms it into its parent node's coordinate system. In this assignment you will extend your Assignment 2 program to process, render and animate objects represented as a scene graph.
What to do: Start with your own code for Assignment 2 or use the provided solution.

1. Read the provided input files to see how a hierarchical model can be represented in XML. Your XML parser should be able to parse these files.
2. Create classes that represent scene graph nodes (*Basic* node that contains the node's name and a transformation, two subclasses: *group* node that contains a list of basic nodes as children and *leaf* node that is one of the primitives from Assignment 2). Write functions in each type of node that will draw itself and delegate drawing to its children if any.
3. Change the XML parser from Assignment 2 so that it supports additional tags for groups and leaves and assembles a scene graph as it parses a provided valid input XML file. Modify the main rendering function so that it renders this scene graph instead of the earlier list of objects.
4. For the provided XML file that shows a "Jack-in-the-box" face, write an animator function that will make the face nod "yes". You can do this by suitably naming various nodes in the XML files, identifying them in your program and transforming them with time. You may find it useful to maintain a map of node names and node pointers.
5. Animate the locomotive model from Assignment 2 so that it moves along a circular track at constant speed. The wheels of the locomotive should roll convincingly (i.e. they should neither skid nor move too slowly relative to the speed of the locomotive).

What is provided: Solution to Assignment 2, input files representing hierarchical objects with the expected picture for a simple input.

A.4. Lighting and Texture Mapping

Description: In this assignment you will extend the program from Assignment 3 by adding support for point lights and textures. You will use these features to enhance the animated scene that you set up in Assignment 3.
What to do: Start with your own code for Assignment 3 or use the provided solution. Use the provided C++ classes that represent a light, a material and a texture.

1. Modify the node classes so that any type of node can contain a list of lights. Add functions to add a light source to an existing node.
2. Modify the leaf nodes so that instead of color, they have material. Modify the draw functions so that they use materials.
3. Write functions in the node classes that enable and disable lights attached to them. This way all the lights can be " switched on" before rendering and "switched off" thereafter.
4. Modify the leaf node classes so that they may contain one texture and its associated transformation. Incorporate this texture in the drawing functions.

4. Modify the XML parser to support the additional tags for lights and textures (see provided input files for an example). The provided texture class uses Qt classes discussed during lectures, so that you can specify textures in standard image formats.

5. Modify the main rendering function so that it renders the scene graph using lighting and textures.

6. Modify the locomotive scene set up in Assignment 3 so that (a) the locomotive runs on a circular track (use a thin cylinder for the track) (b) the track looks more convincing (use provided "road" texture or your own for the track) (c) the locomotive has at least one headlight (d) the scene contains at least two stationary lights (e) the locomotive has at least one texture.

What is provided: C++ classes for light, material and texture, and sample input files.

A.5. Ray tracing

Description: In this assignment (divided in two parts) you will extend your program from Assignment 4 with a ray tracer that renders a 3D scene made from planes and spheres with lighting and texturing.

What to do (Part 1):

1. Write classes that represent a 3D ray and a hit record that stores (a) time 't' on the ray where it intersects an object (b) the point of intersection in world coordinates (c) the normal at the point of intersection (d) material properties. You may use the provided Point3D and Vector3D classes.

2. Write a function *raytrace* that iterates through each pixel on screen, creates a 3D ray in world coordinates and uses the *raycast* function below to obtain a color for that pixel.

3. Write a function *raycast* that takes a ray in world coordinates as input, checks its intersection with the scene graph and returns the appropriate color (returns background color if the ray does not hit any object).

4. Write *intersect* functions in each of the node classes that accept a ray in its parent's coordinate system and check for intersection with itself. If a group node, it computes the nearest valid intersection of the ray with its children. Make sure that if an intersection occurs, the returned hit record is populated correctly and fully.

5. Write a function *shade* that takes the point of intersection, the normal at that point, material, light and texture properties and returns the color. This function should implement the provided *OpenGL shading model* and should support point, directional and spot lights. Use the provided texture class to determine the texture color at the provided location.

6. (Extra credit) Create an XML model that showcases the capability of your ray tracer. The model must contain at least 10 objects.

What to do (Part 2):

1. In the *shade* function, spawn a shadow ray per light source and test if it hits any other objects. If it does, skip this light source. This will effectively implement shadows.

2. Change the material class and the XML parser so that a material can be reflective and/or transparent. Look at the provided input files for an example.

3. In the *raycast* function, if the hit object is reflective, spawn a reflection ray (using the math discussed in class) and use *raycast* recursively to determine its color. Incorporate this color into the final color as discussed in class.

4. (Extra credit) In the *raycast* function, if the hit object is reflective, spawn a reflection ray (using the math discussed in class) and use *raycast* recursively to determine its color. Incorporate this color into the final color as discussed in class.

5. (Extra credit) Support cylinders and cones in your ray tracer.

What was provided: C++ classes for points and vectors, input files, example images created by students from previous years.