Using Computer Graphics to Explore Object Oriented Concepts Using C*

William Kreahling
Department of Mathematics and Computer Science
Western Carolina University
Cullowhee, NC 28723
01-828-227-3951
wkreahling@email.wcu.edu

ABSTRACT

Most recent graphics courses are what we would call *top-down* courses. Courses that focus on using graphical packages to implement and teach graphics. The course discussed in this paper takes the opposite approach. We use a bottom-up approach to teach the basic concepts of graphics. In this class graphics is used as a learning tool, to delve into the underlying data structures and concepts that are needed to create photo-realistic images. The students build a program to create images from scratch using the C programming languages. While there have been other graphics courses to take a bottom-up approach, this paper builds the graphical projects using only the C programming language, no graphics APIs are used in the class. During the course of the semester object-oriented concepts and differences between object-oriented languages and procedural languages are highlighted and discussed in detail. The students not only learn the basic concepts of graphics, but become familiar with the C programming language (students at our university begin programming with Java). The course could be easily tailored for several different classes in a typical CS curriculum, from a beginning programming course, to a programming languages course. At the conclusion of the course a student survey was also conducted to gather student feedback about the course.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: [Inheritance, Polymorphism]; K.3.2 [**Computer and Information Science Education**]: [Computer Science Education, Curriculum]

General Terms

Languages

*This research was supported in part by NSF grant 0305381, and is currently supported by CPATH EAE grant 0722313.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACMSE '09 March 19-21, 2009, Clemson, SC, USA.
Copyright 2009 ACM 1-58113-000-0/00/0004 ...$5.00.

Keywords

Inheritance, Polymorphism, $\tau\varepsilon\chi\nu\eta$

1. INTRODUCTION

Most recent graphics courses are what we would call *top-down* courses. Courses that focus on using graphical packages to implement and teach graphics [1, 6]. The course discussed in this paper takes the opposite approach. We use a bottom-up approach to teach the basic concepts of graphics. In this class graphics is used as a learning tool, to delve into the underlying data structures and concepts that are needed to create photo- realistic images. The students build a program to create images from scratch using the C programming language. While there have been other graphics courses to take a bottom-up approach [5], this paper builds the graphical projects using only the C programming language, no graphics APIs are used in the class. During the course of the semester object-oriented concepts and differences between object-oriented languages and procedural languages are highlighted and discussed in detail.

Our computer science program has recently become involved with the $\tau\varepsilon\chi\nu\eta$ project, which is a new approach for the design of computer science curricula. In the original $\tau\varepsilon\chi\nu\eta$ proposal [2] several introductory courses were developed (CS1 and CS2) that focused on teaching students object-oriented concepts using a non object-oriented language, specifically the C programming language. One of the goals is to help students learn the basics of programming and introduce them to object-oriented concepts without the hurdles that come from learning about *objects*.

Another key aspect of the $\tau\varepsilon\chi\nu\eta$ project is the integration of art with computer science, in the case of the original $\tau\varepsilon\chi\nu\eta$ project, this took the form of semester long ray-tracing project to produce graphical images. The original $\tau\varepsilon\chi\nu\eta$ project was focused on teaching the basics of programming to beginning students in CS1 and CS2. The students in the original project were novice programmers, most of whom has not been exposed to *any* programming language. The classes were taught using the C programming language, but introduced object-oriented concepts as the ray-tracing program was designed and developed over the course of one or two semesters.

When we decided to become involved in the $\tau\varepsilon\chi\nu\eta$ project, our goal was different than that of the original project. We already have two successful CS1 and CS2 classes that are taught using the Java programming language using an "objects first" paradigm. We did not want to revamp two suc-

cessful beginning programming courses, however, many of our students have been requesting a course in the C programming language as well as a course in computer graphics, neither of which has been offered at Western Carolina University for quite some time. The computer science program at Western Carolina University is part of a joint Math and CS department and many of our students lament the fact that they do not always see how the math that they are learning as part of their studies, applies to computer science (a common complaint among our lower level students). We saw the $\tau\varepsilon\chi\nu\eta$ project [3] as a way we could design a course that would address all these concerns. We designed a course using the ray-tracing project from the original $\tau\varepsilon\chi\nu\eta$ project with the goal of communicating the underlying principles involved in graphics, but also instructing students in the use and syntax of the C programming language as well as integrating more math into a typical CS class.

Most recent graphics courses are what we would call *top-down* courses. These courses focus on using graphical packages to implement and teach graphics [1, 6]. The course discussed in this paper takes the opposite approach. We use a bottom-up approach to teach the basic concepts of graphics. In this class, graphics is used as a learning tool, to delve into the underlying data structures and concepts that are needed to create photo-realistic images. The students build a program to create images from scratch using the C programming language. While there have been other graphics courses to take a bottom-up approach [5], the course discussed in this paper used the $\tau\varepsilon\chi\nu\eta$ model as a guide to help design the course.

The remainder of this paper will discuss the design and outcome of offering this new hybrid graphics course. Section 2 will discuss the layout of the class, the assignments and lectures used to tailor the class to our needs. Section 3 will discuss the problems and benefits observed while teaching the class. Finally in Section 4 we conclude with our observations about the successes and failures of this course, including information garnered from student questionnaires.

2. COURSE DESIGN AND GOALS

We first started thinking about offering a graphics course in early 2006, computer graphics had not been taught at Western Carolina University for some time, and students kept asking if we could offer such a course, or incorporate more graphics into their existing classes. Our program tends to focus more on *why* things work in computer science and programming, not just the overall outcome. This has meant that graphics and GUI's (perennial student favorites) have fallen by the wayside in our curriculum. However, we did not just want to offer a graphics course that was a study in software packages or APIs used to create and manipulate graphics.

Around this same time we had been getting requests from students to learn other programming languages (besides Java) in the classroom setting, the top choice of languages to learn was the C programming language. Another comment we kept hearing from students was the fact that the math they were learning, did not seem to tie directly into what we they doing in their CS classes. We started thinking about ways we could address student comments and concerns as well as help build future interest in our CS program. We came up with the following goals for a possible new course:

1. Incorporate computer graphics in a non-trivial manner.
2. Use the C programming language.
3. Incorporate more math (in a meaningful fashion).

Around this time we became aware of the $\tau\varepsilon\chi\nu\eta$ project that incorporated art and graphics as a way to introduce basic programming concepts to novice programmers. Obviously this was not a perfect fit, as our graphics course would be targeted to sophomores and juniors, students who had already taken their introductory programming courses. However, the $\tau\varepsilon\chi\nu\eta$ approach approach seemed like a good beginning.

We decided to keep the same core structure as a programming class designed at Clemson University by Mike Westall as part of the $\tau\varepsilon\chi\nu\eta$ project [2]. The original course used in the $\tau\varepsilon\chi\nu\eta$ project assumes no knowledge of programming and thus started with the basics of programming. The course also included weekly labs, allowing the students time to complete portions of their assignments with faculty help. Neither of these would be the case for the course we wanted to offer. However, the basic structure of the course was a good starting point, and with some modifications, I was able to tailor the course for our needs.

I decided to switch the emphasis from learning the basic concepts of programming to comparing and contrasting the concepts and data structures of C with those in Java, a language with which all the students here at Western Carolina are familiar with. This "programming languages" approach allowed me to discuss topics ranging from garbage collection, inheritance, polymorphism, and type checking to linking, loading, compiling, and interpreting as well as the basics of computer graphics and some basic linear algebra and calculus.

The class utilized a semester long project called a ray-tracer, that served as a framework for all our discussions over the course of the semester. A ray-tracer is a program that renders graphical images of a virtual scene in three-dimensional space, as shown in Figure 1.

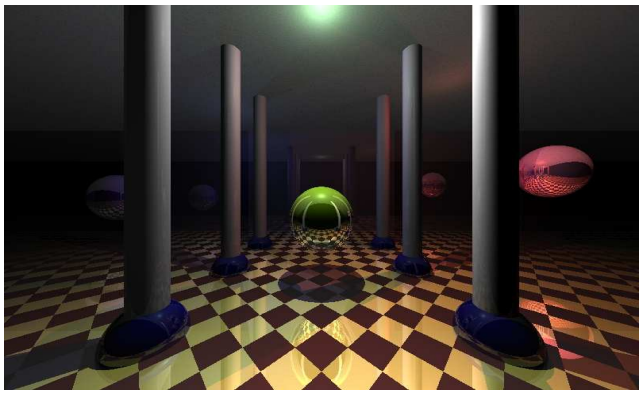

Figure 1: 3D Scene Created by a Student

The basics of the ray-tracer for this course starts with a *viewpoint* which is the location from which the scene is being viewed. An imaginary *ray* is fired from the viewpoint through every pixel on the screen. (The screen acts as a window between the viewpoint and the 3D scene we are attempting to render). If the ray hits an object in the scene

then the corresponding pixel though which it travels is 'lit' up. The color and intensity of the pixel depends on the object hit within the scene and the distance from the viewpoint (in general objects farther from the viewpoint will appear darker, ignoring light sources). This project is not trivial and each student builds their own version of the program, without the use of any pre-existing graphical packages.

There is a lot of effort involved to build a working ray-tracer. I classified the assignments the students would be doing into three categories: Homeworks, programs, and projects. The vast majority of work done for both the homeworks and programs would be needed for their major project (i.e., the ray-tracer).

The work classified as *homework* were assignments the students should be able to do in a day or two (in some cases just several hours).

Homeworks:

- Sample C program
- A collection of vector library functions
- Linked lists and makefiles

The only assignment that was not related to the ray-tracing program was the first homework. The purpose of the first assignment was to let students familiarize themselves with the tools we would be using during the semester as well as the C compiler, *gcc*. For the other assignments I supplied the students with test programs and data so they could check the correctness of their programs.

The work classified as *programs* were larger tasks that students would typically need a week or more to complete. The programming assignments included:

Programs:

- Read in ppm image header files
- Convert color images to grey scale and sepia tones
- Stretch, shrink, and tile images
- Read and parse data files for a 3D scene

By separating the work into small assignments the students were not overwhelmed and were able to see tangible progress throughout the course of the semester. The homeworks and programs were able to be run and tested as stand alone programs, not just as a piece of a larger whole.

The last set of assignments were the *projects*. Each project was a different version of a working ray-tracer. For each project, the students had to produce their own working version of the ray-tracer. Each successive version of the ray-tracer incorporated more functionality.

Projects:

- Ray Tracer Version 1 (basic framework, spheres and infinite planes)
- Ray Tracer Version 2 (lights, finite planes, shadows)
- Ray Tracer Version 3 (reflections, more complex shapes, cones, texture mapping, etc,)

3. CLASSROOM OBSERVATIONS

The first several weeks of class were spent discussing the syntax of the C programming language, talking about different data types, those that are common to both C and Java, and those that are not. For many students this was a review of material covered before, but brought the ideas of lifetime and scope into much tighter focus, as the students could not use the *class* construct that they were used to working with in Java.

At this point in the class, pointers were introduced as a robust way of accessing memory with more functionality than references, but with the associated higher risks. The concept of pointers led to some interesting discussions of language design and issues of type checking. Pointers (especially function pointers) are essential to mimicking object-oriented behavior in our C programs, so we spent a good amount of time discussing how pointers worked, including dangling pointers, and pointer arithmetic. These discussions dovetailed nicely into discussion about the *printf* and *scanf* functions and the purpose of the *address of*(&) operator.

During the time that we were discussing the basics of the C programming language, several homeworks were assigned to allow the students to become more familiar with the tools we were using, such as *gcc, DDD* and *lint*. For one assignments the students were asked to write a program using linked lists that could hold some random structure. (These structures were not related to the overall ray-tracing project, but were created specifically for only this assignment). A number of transfer students found it difficult to create the linked lists from scratch and were not clear on the the exact details of what constituted a linked list. Several students indicated that they had previously only used the linked lists built into the Java programming language. (Although most students who completed CS2 here at Western were familiar with linked lists as that material is covered in detail in that class).

At this point in the class, we began to develop the data structures needed to build the ray-tracer that would be the culmination of the work done in the class. Several data structures were defined using *void* pointers. In this manner, we could connect the void pointer from one structure to another structure and mimic the concept of inheritance found in many object-oriented languages.

Figure 2 shows three abridged data structures used in the ray-tracing project [2]. The ray-tracer displays a scene in 3D space composed up of different items, including spheres, planes, cones, etc. Items that existed within each scene were defined within "model" files which are parsed when the ray-tracer is started. For each item in a scene an *object_type* structure is created, and initialized with the proper data from the model file, then depending on whether the item models a plane, a sphere, or some other type of item, the appropriate data structure is created and a link from the object structure to the new structure is created. This mimics the behavior of inheritance.

Figure 3 depicts an *object_type* data structure containing a *void* pointer named *priv*. The *priv* pointer can point to any other type of data, in this case either a sphere_type structure or a plane_type structure.

Figure 4 shows a more complicated example of inheritance within the ray-tracer. A *textured_plane_type* structure is needed to represent a plane that can have different textures

```
struct object_type
{
   struct object_type *next;
   int objid;
   int object_type;
   double(*hits)(double *base, double *dir,
                 struct object_type *);
   ...
   void *priv;
}

struct plane_type
{
   double normal[3];
   double point[3];
   ...
   void *priv;
}

struct sphere_type
{
   double center[3];
   double radius;
   void *priv;
}
```

Figure 2: Inheritance in C, Using Void Pointers

or images mapped onto it's surface. The *textured_plane_type* structure is a child of several other data structures, which are all ancestors of the original *object_type* (analogous to the class *Object* in the Java programming language).

One of the interesting discussions that took place during this class occurred when students tried to de-reference the void pointers that exist within these data structures. In the beginning almost all the students had difficulty when trying to de-reference the void pointers. Many students were confused when the compiler would not let them de-reference the void pointers until they explicitly type cast the void pointer to correctly indicate the type of structure that it was pointing to. This error lead to detailed discussions about type casting, multiple inheritance, as well as static and dynamic typing.

Polymorphism was also mimicked in the ray-tracer using the C language through the use of function pointers, as depicted by the *hits* function pointer in the *object_type* structure in Figure 2. Each item in the scene needed a function that determined if an imaginary ray, when fired from a certain location, would hit a spot on the item. Different items (e.g., planes, finite planes, spheres) needed different hits functions. In the first iteration of the ray-tracer, *switch* or *if* statements were used to correctly call each item's *hits* function. However, after function pointers were discussed function pointers were used to mimic polymorphic behavior.

When the structures for each item comprising the 3D scene were created and initialized, the proper *hits* function was connected though the use of the function pointer in the *object_type* data structure for the specific item. Thus when a *hits* function needed to be called during run-time, the *hits* function pointer was accessed to call the correct method and execute the proper code. Each of the items in the 3D scene had a different set of mathematical formulas to determine if

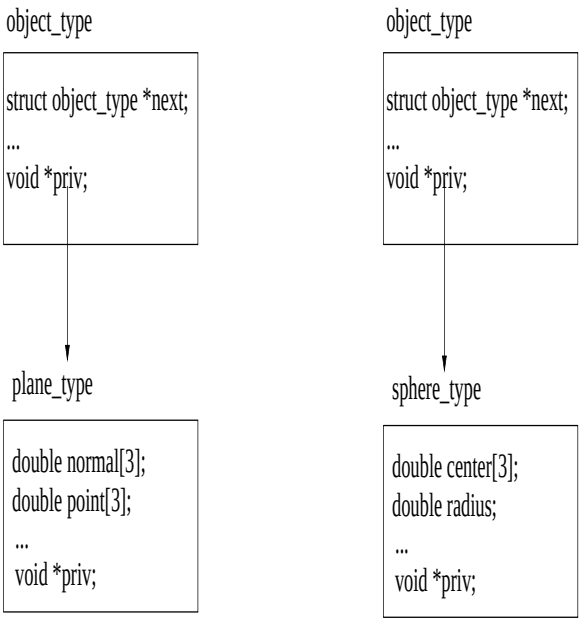

Figure 3: Inheritance Using C

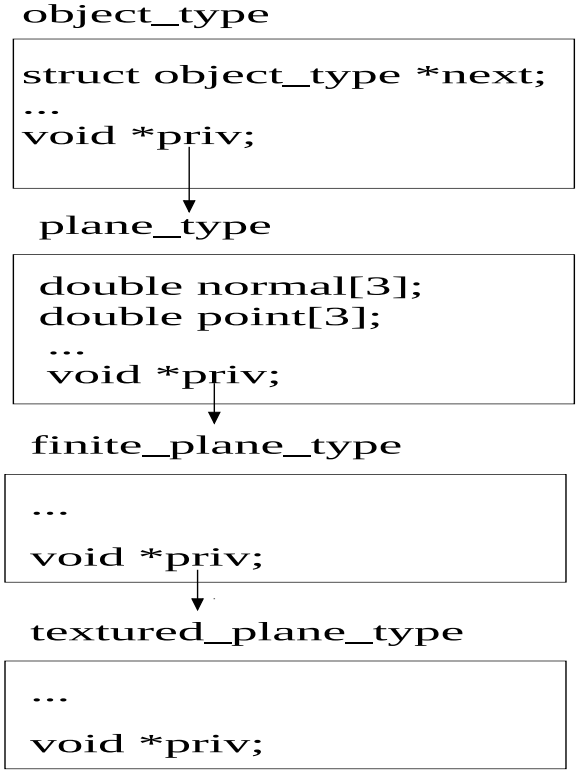

Figure 4: Structure With Several Ancestors

a ray fired from the viewpoint would hit the object, and the object's distance from the viewpoint. While these *hits* functions were different for different items in the scene, they did have similarities depending on how closely related the objects were, further tieing in the concepts of polymorphism and inheritance. These *hits* functions also involved the use

of matrices, quadratic formulas, and many other math formulas from linear algebra and calculus.

When function pointers were covered in class, many students expressed dismay and confusion over the purpose of using them, however by the end of the semester, quite a few students told me that while painful at first, the use of function pointers really helped them grasp the concept of polymorphism and its practical uses.

During the the semester I was inundated during office hours (and non-office hours alike) with student questions about C, graphical concepts (such as anti-aliasing and texture mapping), debugging techniques, and revision control systems. Many students would stop by to show me the latest images their ray-tracer had produced and ask for help adding non-required features to their programs. I had more student interaction (outside the classroom) for this class than any other class I have taught.

Several students went above the requirements necessary for their completed programs. For example, I had asked students to add a simple version of anti-aliasing to their ray-tracer programs. The anti-aliasing algorithm discussed in class was pretty basic, but worked for our purposes. Several students took it upon themselves to research better approaches to anti-aliasing and incorporate the new methods they discovered into their programs.

Several students informed me they had signed up for either linear algebra or calculus 3 courses to get more information on the math we used in this course. Other students expressed interest in continuing to work with their ray-tracer programs as a means to do their senior projects. Still other students began producing extra images to use as wallpapers on their computers and cell phones.

4. CONCLUSION

The basic tenants of the $\tau\varepsilon\chi\nu\eta$ project were successfully applied in an upper level course, bring together the basic principles of graphics, the C programming language and basic linear algebra and calculus. At the beginning of the semester students seemed a little overwhelmed by the amount of new information being presented: the syntax and behavior of a new language, a large program to develop images and a fair amount of math. I tried to "manage" the students' workload by giving out detailed notes and organizing the class into small manageable homeworks, tasks, and quizzes. When I assigned the first project that actually produced graphical images, the student's interest increased dramatically.

At the end of the semester the students were given a questionnaire from the DELES survey [4] and were allowed to complete the survey without the instructor present. Figure 5 show several charts depicting student reaction to their learning in this class. Roughly half the class felt that the class contained authentic learning, which is not surprising since the semester long project was a relatively simplistic ray-tracer and not a real-world graphics package. However, the majority of students felt that they were active participants in this course and had autonomy with regards to their classwork and learning. Also, the vast majority of students were also very satisfied with the amount of instructor support that they were given in this class. The survey was taken by 15 (of the 18 current) students with one week of classes left in the semester.

One of our goals for this new course was to more fully integrate math concepts into a CS course in a meaningful fashion and there is evidence to show the graphics course is helping us meet this goal. During the presentations of senior projects, one student noted similarities between the work he had done in the graphics course involving rotation matrices and some of the graphical work he had done for his senior project. Also, several of the comments on the exit interviews for graduating seniors, conducted the first year we taught the graphics course indicated that the students found topics covered in their math courses (including calculus and vector math) very useful in completing the several projects in their graphics course.

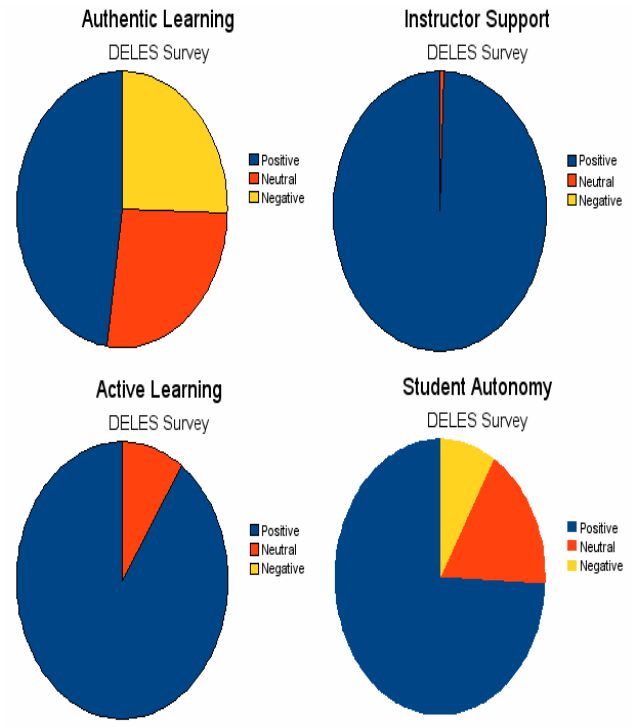

Figure 5: Student Responses to selected question from the DELES Survey

5. REFERENCES

[1] E. Angel, S. Cunningham, P. Shirley, and K. Sung. Teaching computer graphics without raster-level algorithms. In D. Baldwin, P. T. Tymann, S. M. Haller, and I. Russell, editors, *SIGCSE*, pages 266–267. ACM, 2006.

[2] T. Davis, R. Geist, S. Matzko, and J. Westall. $\tau\varepsilon\chi\nu\eta$: a first step. In J. Impagliazzo, editor, *Proceedings of the 35th SIGCSE technical symposium on Computer science education (SIGCSE-04)*, volume 36, 1 of *SIGCSE Bulletin*, pages 125–129, New York, Mar. 03–07 2004. ACM Press.

[3] T. A. Davis, R. Geist, S. Matzko, and J. Westall. $\tau\varepsilon\chi\nu\eta$: trial phase for the new curriculum. In I. Russell, S. M. Haller, J. D. Dougherty, and S. H. Rodger, editors, *SIGCSE*, pages 415–419. ACM, 2007.

[4] M. J. Garica. Comparison of student perceptions of classroom instruction: Traditional, hybrid and distance education. *Turkish Online Journal of Distance Education.*, 7(2):46–51, APril 2006.

[5] M. B. Gousie. Teaching computer graphics in a small department. 2000.

[6] J. O. Talton and D. Fitzpatrick. Teaching graphics with the openGL shading language. In I. Russell, S. M. Haller, J. D. Dougherty, and S. H. Rodger, editors, *SIGCSE*, pages 259–263. ACM, 2007.