

Using a Software Framework to Enhance Online Teaching of Shader-Based OpenGL

James R. Miller

Electrical Engineering and Computer Science

1520 W. 15 Street; 2001 Eaton Hall

University of Kansas; Lawrence, KS 66045

1-785-864-7384

jrmiller@ku.edu

ABSTRACT

Shader-based OpenGL is a powerful and exciting tool for individuals with data visualization needs, interests in gaming, a need to create and manipulate synthetic environments, and a variety of other high-performance tasks. It is well-known that shader-based OpenGL is difficult to learn for programmers with no prior graphics API experience – some have even claimed it's impossible to teach to individuals without such prior exposure. In this paper, we report on our experiences developing an educational approach that we believe has contributed to the base of evidence that it is not only possible, but also desirable to do so. Our overriding goal has been to learn and exploit the extent to which mastering a complex graphics API like modern shader-based OpenGL can be enhanced by using a software design framework into which relevant concepts can be placed, thus facilitating more rapid assimilation and mastery of the concepts. We have been using this technique in our Introduction to Computer Graphics course in which we have two primary goals: teach shader-based OpenGL to students who have no prior experience using a graphics API, and present a framework to students that will scale up to medium and large scale applications, both in terms of code size as well as data size. We do not in any way suggest that our architecture is the “best” for all advanced graphics applications, or even for teaching. Instead we simply claim that use of such a framework helps students master complex OpenGL concepts and develop nontrivial interactive 3D applications. Once students fully understand the basics, they should find it easy to migrate to other perhaps quite different architectures.

Categories and Subject Descriptors

K.3.1 [Computers and Education]: Computer Uses in Education – *distance learning*

K.3.2 [Computers and Education]: Computer and Information Science and Education – *computer science education; self-assessment*

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCSE '14, March 5–8, 2014, Atlanta, Georgia, USA.

Copyright © 2014 ACM 978-1-4503-2605-6/14/03...\$15.00.

<http://dx.doi.org/10.1145/2538862.2538892>

General Terms

Design, Experimentation, Languages.

Keywords

Computer-aided Learning; Educational Technology; Computer-Graphics; Self-paced learning; Mastering Shader-Based OpenGL.

1. INTRODUCTION

OpenGL [4] has regained its status as the preeminent interactive 3D modeling and rendering API for scientific, engineering, and gaming applications. It has done so in part by changing the nature of how graphics data are specified, processed, and rendered, explicitly exposing an intimate cooperation between the CPU and the GPU. In so doing, the learning curve for so-called shader-based OpenGL has become a little steeper. So much so, in fact, that many have opined that it is not practical to teach to introductory students at the undergraduate level without at least some exposure to the older, simpler version of the API. See, for example, [1] in which this prevailing opinion is cited, but rebutted. We agree with the rebuttal, and the purpose of this paper is to present the approach we have adopted to make it work.

We have been teaching shader-based OpenGL in our introductory computer graphics course since the Fall of 2011 when [1] first appeared. This course is targeted towards senior undergraduate computer science and computer engineering majors, although we frequently attract several graduate students as well since the material serves as a prerequisite to several graduate courses in visualization and modeling. Moreover, graduate students are allowed to apply this course towards their graduate degree.

While the Fall 2011 semester was a bit rocky, we have developed an approach based on the use of a consistent software architectural framework that allows students to progressively learn increasingly sophisticated aspects of OpenGL, in part by relating them to the framework. The bulk of the material is online and is embodied in an increasingly sophisticated set of examples that share the framework. Packaged with each example is a detailed description of the new features introduced in the example along with complete source code so that students can run, modify, and experiment with the example programs.

The framework we use serves a dual purpose. Obviously the main purpose is that it helps students organize their growing understanding of how portions of the OpenGL system operate and cooperate. Secondly, it is a C++ object-oriented framework that serves as a realistic example of several advanced C++/OO design

patterns that students have typically only seen previously in unrealistically small abstract contexts, if they have even seen them at all.

A major goal is to present a framework that will scale up to medium and large scale applications, both in terms of code size as well as data size. However we also recognize and emphasize to students that this framework is not intended to represent the “best” architecture for all advanced graphics applications. Indeed, very different architectural styles and organizational methods may well be needed in different application contexts. Our philosophy is instead that this is a useful framework with which to master OpenGL concepts and develop nontrivial interactive 3D applications. Once students fully understand the basics, they will be well-prepared to migrate their expertise into other architectures.

2. RELATED WORK

There are online tools developed to aid students in mastering complex concepts related to interactive 3D modeling and rendering. The Exploratories project [2], for example, offers a wealth of free software and tools for teaching fundamental concepts including color theory, viewing in 3D, illumination models, and others. Standalone applications illustrating such concepts are available as well as component tools to help developers with basic tasks like drawing axes and 3D shapes (boxes, cylinders, etc.). These are high-quality tools that serve an important purpose, but one that is somewhat different from what we are trying to accomplish here.

Most existing standard graphics textbooks focus on teaching the theoretical concepts of computer graphics modeling and rendering and then relating them to a concrete API like OpenGL (e.g., [1]). This is unquestionably an important and valuable function, and we definitely exploit that fact in our course by using such a textbook for readings on those underlying graphics concepts. Again, we believe our contribution is best seen as serving a companion, but orthogonal purpose: namely the development of a software architectural framework that both helps students organize their new knowledge and lets them develop skills that will help them design and implement large scale applications that make effective use of the graphics functionality they are learning to solve real-world problems in data visualization and other areas.

Talton and Fitzpatrick describe their rationale and approach to teaching GLSL in the introductory computer graphics course [6]. They argue that exposing introductory students to GLSL is ultimately an advantage, in part because it forces them to come to grips with the internal mathematics, algorithms, and data structures required while removing the “black box” aspect of the process. They go on to state their belief that students getting this exposure are “more likely to produce visually and technologically impressive course projects than those who are taught to use only the fixed-function OpenGL API.” Since their paper appeared, the entire fixed-function pipeline to which they allude has been deprecated as has a large portion of the basic geometry definition API. These developments appeared in OpenGL 3.3 released in March 2010, and they have reinforced the steep learning curve, making the issue much more critical.

3. MODEL-VIEW-CONTROLLER

Our basic framework is based on a simplified version of the well-known Model-View-Controller (MVC) design pattern. We begin by briefly explaining the motivation for and use of the MVC

pattern in general, and especially its popularity for interactive modeling and rendering applications. We then observe that in introductory graphics applications such as the ones they will develop in this course, it is frequently the case that the “model” will not really have a use outside the context of the graphics program itself. Indeed, there may not even be a standalone representation of the model aside from its implicit representation in code and data declarations. Hence we introduce a simplification of the MVC pattern by proposing two generic classes: a combined “ModelView” and a “Controller”. Since there really is no separable “Model”, our merged ModelView makes sense because we only access the model in the context of generating a view of it. This simplified pair of classes then forms the core of the framework we develop for the course.

An OpenGL program can produce and interact with images in multiple windows, and each window is managed by a single Controller. Multiple models (i.e., ModelView instances) can be created; each can be added to one or more Controllers and hence rendered in the corresponding windows.

4. MAIN CONTRIBUTIONS OF THIS EFFORT

We believe the approach to teaching the introductory computer graphics course described here accomplishes several useful goals:

- Students get significant experience with the common MVC design pattern (albeit a somewhat simplified version of it). They get the big picture while using our simplified ModelView – Controller architecture for projects.
- Of course the primary goal is to master the somewhat complex art of shader-based OpenGL, including getting significant experience with cooperative CPU-GPU software design, development, and debugging issues. Students see how to exploit the massively parallel GPU for their rendering code, and we now very briefly introduce them to general-purpose GPU-based computation between frame refreshes via OpenGL’s Compute Shaders. Included is an introduction to the idea that large numbers of GPU program instances simultaneously receive and process “work assignments”, the common paradigm for data parallel computational models.
- Students gain practical experience with several intermediate-level object-oriented concepts, such as class hierarchy design, and the need to understand, embrace, and exploit the differences between instance and class methods as well as between instance and class variables. Students typically have been minimally exposed to these ideas to various extents, but they typically have not yet seen realistic examples of where and how they can be effectively used. We are exploiting a synergy here: learning graphics in the context of a useful software architecture helps students master the graphics concepts; conversely, employing useful software design patterns in a realistic setting helps them become more comfortable with sophisticated design patterns, and hence better software engineers.
- Students find it both a blessing and a curse that they gain experience studying and understanding an established design and code infrastructure inside of which they will build their own projects. Some have expressed a lack of enthusiasm for reading and studying existing code bases, but the counterargument is that that is more representative of what they will be doing in industry than is starting a program from scratch

every few weeks. Moreover, the established code base they must understand in order to do the course projects is small.

- Once they have mastered the framework itself, students can focus more attention on “normal” graphics issues such as how to build, manipulate, illuminate, render, and interact with large 2D and 3D models.
- All the examples, code, and explanatory information are on the web site (<http://people.eecs.ku.edu/~miller/Courses/OpenGL/OpenGL.html>), hence students can – to a certain extent – proceed through the material at their own pace. Specifically, if a student is eager and wants to move faster than our in-class discussions are proceeding, they are certainly able to do that.

5. PROGRESSION THROUGH THE ONLINE PORTION OF THE COURSE

The primary online web site for the shader-based material just cited above is structured generally as follows. There are initial pages that describe the relevant background and history of OpenGL as well as the general architecture of an OpenGL program. Following a review of the overall course goals, an initial “Hello, OpenGL” application is presented which simply defines and renders a triangle. In spite of the simplicity of this example, students see the bulk of the control and communication patterns that are used throughout the course. Following this simple example, we briefly summarize review the primary OpenGL “draw modes”, relating them to the order in which points are placed in buffers sent to the GPU.

Following this overview, the primary online materials that develop the software framework are presented as a succession of increasingly sophisticated example programs organized into “Sample Program Sets”. Each introduces two or three new concepts, the first by extending “Hello, OpenGL” so that students do not have to start from scratch understanding a new model.

5.1 Hello, OpenGL

We begin by explaining the high-level execution model employed by our ModelView–Controller architecture (Fig. 1). The main program is very small, basically just creating a Controller (and hence its window on the screen) and a ModelView. It then adds the ModelView to the Controller’s list of managed models and hands off to the event handling loop of OpenGL’s window system interface GLUT [4]. The following briefly summarizes the key concepts introduced in each step.

Hello, OpenGL: Execution Overview

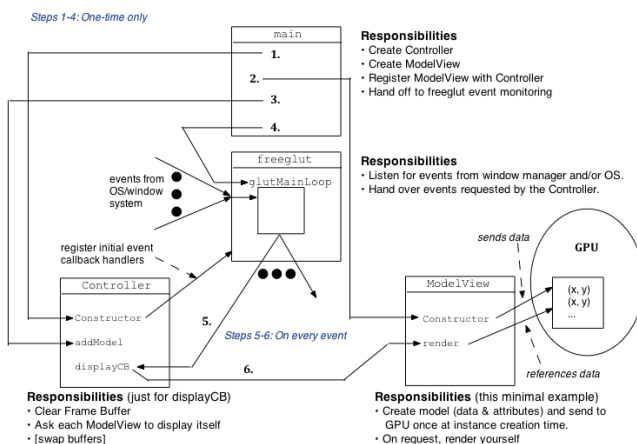


Figure 1: Basic Communication and Control Flow.

Step 1 creates the Controller along with its Rendering Context (RC – an encapsulation of OpenGL state in a window). Event handling in OpenGL is based on event callback registration. Client code registers a function in the OpenGL RC that is to be invoked when a given event occurs. OpenGL is not object-oriented, hence only external functions (as opposed to, say, class instance methods) can be registered. C++ class methods can be registered as callback functions since they are called in a manner identical to that for regular C functions. This observation allows us to illustrate a common pattern to students for translating non-object-oriented API callbacks to object-oriented instance methods. In the simplest case, for example, a Controller class method is registered as the callback function, and a Controller class variable is used to remember the last Controller instance created. When the class method is called in response to a registered event, it uses the saved Controller class variable pointer to hand off responsibility for processing the event to the appropriate Controller instance method.

The following captures the essence of the idea as presented to the students:

```
Controller* Controller::theController = NULL;
Controller::Controller()
{
    ...
    theController = this;
}
// A class method registered with GLUT:
void Controller::anEventCallback(...)
{
    if (theController != NULL)
        theController->handleEvent(...);
}
// The instance method to handle the event:
void Controller::handleEvent(...)
{
    ...
}
```

While this simple scheme of translating class methods registered as callback functions to instance methods works in single-window (and hence single Controller) applications, alternative designs are discussed with students that are capable of handling multiple Controller situations. The typical solution is to use a mapping from GLUT window IDs retrievable during event processing to the corresponding Controller pointer.

The second step in this generic execution model creates one or more ModelView instances. From the perspective of teaching core shader-based OpenGL concepts, we need to reinforce the distinction between what happens at model creation and/or modification time as opposed to rendering time. Model data is copied to buffer objects on the GPU; storage for these objects must be managed in much the same way normal dynamically allocated storage is on the CPU. The ModelView constructor is responsible for dynamically allocating GPU buffer storage and initializing it with the required model data. The destructor is then responsible for releasing this buffer storage. If model modifications are performed (typically in response to events), other ModelView methods are used to rewrite all or part of the data in the buffers stored on the GPU. It is clear to students that these creation and modification operations are performed infrequently as compared to the much simpler operations performed to support dynamic updates such as animation, which we shall revisit below in steps 5 and 6.

Core rendering is performed using shader programs written in OpenGL's GLSL GPU shader language. These programs must be explicitly compiled, linked, stored on the GPU, and executed on demand by the host CPU program. Typically several instances of a `ModelView` class will share the same shader program(s), so we use this as an opportunity to illustrate how class variables can be used to manage the storage and variable location data for the shader programs, allowing a single compiled and linked shader program to be shared among all the instances. We illustrate how a reference counting scheme can also be used so that – when the last `ModelView` instance has been deleted – the storage associated with the common compiled and linked program can be released.

Step 3 simply registers the `ModelView(s)` with the Controller so that each instance can be located for rendering during display callbacks. Step 4 then transfers control to the main GLUT event monitoring loop.

When the GLUT recognizes the need to recreate the display in a window, it calls the registered display callback (a class method in Controller). That callback delegates to an instance method as discussed above which in turn invokes a render method of each `ModelView` instance registered with the Controller. Here students observe that very little CPU-GPU communication is required – typically just simple directives involving minimal CPU-GPU data transfer that establish values for certain attributes and trigger the rendering of data stored in buffers already on the GPU.

This simple framework and mode of operation will require only minor extensions to get through the bulk of the introductory online material that needs to be conveyed to students during the first half of the course. Following this basic introductory example students progress through a series of sample program sets, a brief summary of which follows.

5.2 Sample Program Set Structure

Once we have completed discussion of “Hello, OpenGL”, we move on to the portion of the web site that contains a series of increasingly sophisticated example applications. Each sample program has the same MVC structure, of course, and this allows us to focus on just the new features being considered in each example. Each set is presented in a web page in which each example of the set is introduced using the common general format, a portion of which is illustrated in Fig. 2.

2. **twoTriangles_V2**
NEW: More Elaborate Fragment Shader Coloring; Event Handling

More Elaborate Fragment Shader Coloring

Rather than pass an explicit color to the shader program, this example passes a per-primitive integer attribute to the fragment shader that is used to select one of a variety of schemes to compute a color. In some cases, the attribute is used to simply select a constant color (for example, `colorMode 1` or `2`); in other cases, it is used to specify a general algorithm that, for example, can use the current pixel position to compute colors in very general ways. To enable this, we see a small, but important, change in the vertex shader: it passes the computed logical device space coordinates into the fragment shader using “out” variables which are then imported into the fragment shader as “in” variables. That is, the same values used to set the *x* and *y* components of `gl_Position` are also separately output as “out” variables. It is necessary to do this because the coordinates passed to `gl_Position` get mapped to pixel coordinates and would not

- `twoTriangles.cpp`
- `ModelView.h` ; `ModelView.cpp`
- `twoTriangles_V2.vsh` ; `twoTriangles_V2.fsh`
- `Makefile`




Figure 2: Introducing Example 2 of Sample Program Set 1

The description contains a textual overview of the example, beginning with a short list of new features presented followed by a brief narrative of each new feature. Links to all source files for the examples are given, and a thumbnail of the generated output is presented. Each sample program set also contains a link to a master tar file that contains the source code for all examples in that program set.

5.3 Sample Program Set 1

Using programs that continue to generate only small numbers of triangles, this set introduces, in turn, the following important concepts:

- Coordinate system issues including generalized communication between the Controller and its several managed `ModelView` instances in which each `ModelView` instance reports its spatial extent to the Controller; the Controller uses this information to maintain a master spatial extent that can then be used to map from world coordinate data to logical (and later physical) coordinates on the device.
- Aspect ratio preservation, including when it is and is not appropriate.
- Attributes – both “per-primitive” and “per-vertex” attributes.
- General event handling schemes including inverse coordinate mapping that allows pixel coordinates to be mapped back to world coordinates.
- Program design patterns illustrating different common approaches to how responsibility for geometry and appearance generation can be divided between the CPU program and the GLSL shader program running on the GPU. Included is a glimpse into how procedural modeling in GLSL can be used to completely determine the appearance of geometry stored in buffers on the GPU. For example, Fig. 3 is a screen shot from a demonstration program showing how two simple triangles can be rendered in a half dozen very different ways based strictly on choices made in the GLSL fragment shader program.

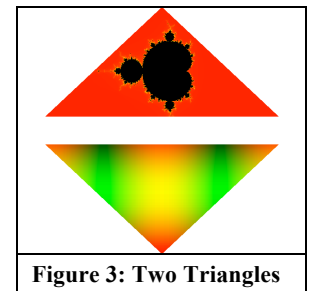


Figure 3: Two Triangles

5.4 Other Sample Program Sets

The remaining online sample sets continue to add to the students' repertoires by introducing concepts involving fonts, general color space and color modeling issues, simple animation using timer callbacks, and modeling techniques. We use a font library based on utilities developed by George Sealy [5], slightly modified to work in the context of our execution environment. It is the general modeling techniques introduced at this stage that merit the most attention here.

While certain variations can be reasonably introduced in different `ModelView` instances, students quickly run into limitations as they attempt more elaborate scenes. Our first approach to overcome these limitations is to convert `ModelView` to an abstract base class. Keeping `ModelView` simple – basically preserving only those interfaces needed to communicate effectively with the Controller and factoring out a few common tools required by all concrete `ModelView` subclass instances – we quickly arrive at a modified design that allows multiple unique instances of models to be maintained and rendered in this environment. Our first example of such a scene results in a “mountain village” as shown in Fig. 4.

Three concrete subclasses of `ModelView` are used: House, Tree, and Mountain. Each can be parameterized on creation with

different relative and absolute sizes, labels, etc. Students see how different subclasses can have their own GLSL shader programs for rendering their instances. For example, the House subclass render method sends to its GLSL program information describing the part of the house being rendered, and the GLSL program is then responsible for creating an appropriate rendered appearance.

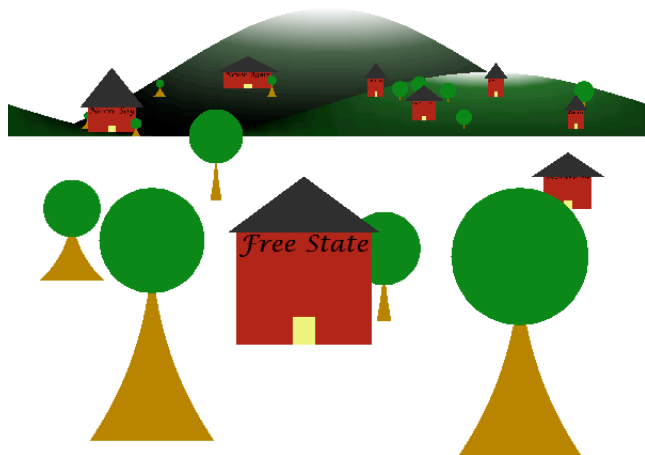


Figure 4: A Mountain Village.

5.5 Moving to 3D

Making the transition from 2D to 3D is always difficult. Of course 3D coordinate data is required; 3D normal vector descriptions and material reflectance properties are required as well to support lighting models. Students must cope with a more complex viewing model involving line of sight, field of view, etc. The good news is that neither a new nor an expanded set of APIs is required with shader-based OpenGL. It is of course necessary to explain the mathematical aspects and discuss what computations must be performed and where (e.g., on the CPU or on the GPU using GLSL). But no new APIs are required in order to render a 3D scene with a given lighting environment. Once students know what needs to be done and what data needs to be sent to the GPU, they simply use interfaces they have been using all along to make that happen. Even implementation of dynamic 3D viewing operations like rotations, panning, and zooming can all be accomplished using APIs they are now well on the path towards mastering.

This base familiarity with the core API means we can also begin to back away from presenting complete running examples, focusing instead on conceptual material and code snippets that introduce new modeling and rendering techniques. We include focused code samples where needed, but expect the students to exercise their understanding of the overall structure so as to be able to incorporate these into shader programs and/or CPU code as needed.

Subsequent online and in-class materials continue to develop more advanced 3D modeling techniques. Some new APIs are required along the way for certain advanced techniques, most notably texture mapping. Given their overall familiarity with the

architecture and the OpenGL API, we find that students are able to simply focus on mastering the concepts while plugging implementations of the new functionality they are learning into the architecture as needed.

Figures 5 and 6 below show examples of student-generated projects from the last time the course was offered. In addition to the 3D scenes, it was a requirement to support walking through or otherwise exploring the 3D environments, and students generally had few problems doing so.

6. DISCUSSION

As mentioned in the Introduction, a very common perception is that it is impractical to teach pure shader-based OpenGL programming without first exposing students to required introductory concepts using the older, simpler, and now deprecated interface. There are a number of problems with this approach that makes seeking ways to avoid that initial experience worth pursuing. One obvious problem is that students must learn two different APIs. Somewhat worse is the fact that two overlap considerably, so when “making the move” to the pure non-deprecated shader-based OpenGL API, they must pay careful attention to what they can (and in fact must) still use as opposed to functionality they must eschew. It was the goal of the online material briefly introduced here to help fast-track students into an adequate comfort level with shader-based OpenGL so that they could demonstrate solid skills by the end of a single semester.

It should come as no surprise that the first time through the course was a bit rough. Students struggled as I tried to identify the best order in which to cover the various required concepts. Exam scores were noticeably lower than usual. Submitted student projects were much less sophisticated and more quirky and buggy than those in previous semesters.

Typical comments from the first offering of the course included:

- “Don’t know if he realizes how difficult some of the stuff is for beginners.”
- “More frequent smaller projects would help build a better understanding.”

These comments along with my general observations and experiences from the first offering led to several modifications including a significant restructuring and reordering of the online topics covered, better up-front explanations of high-level concepts, and more and smaller projects.

By the second – and especially the third – offering of the course in this new format, significant improvements were noticed as evidenced by all measures noted above. Submitted student projects were less buggy and much more sophisticated (see Fig. 5 and 6 below), many including elaborate indoor or outdoor scenes. Performance on exams was markedly better, and end-of-semester course evaluations reflected a higher level of satisfaction. For example:

- “This is the most challenging and one of the most rewarding classes I have taken.”

To be sure, many students still commented that the material was quite difficult and that they struggled. But the overall sense was much more positive than that from the first offering.

7. FUTURE DIRECTIONS

The jump to 3D has always been a difficult one – even when teaching the older deprecated API. It continues to be a significant hurdle for students when using pure shader-based OpenGL. It is my sense, however, that we have reached the point where the primary difficulties have reverted to those that are intrinsic to 2D versus 3D rather than being unique to the use of pure shader-based OpenGL. Primary difficulties encountered by students in this class include conceptual issues surrounding (i) 3D modeling and (ii) generating appropriate views of 3D scenes. Problems surrounding the latter are heavily related to visualizing the relationships among the various coordinate systems that APIs like OpenGL use. Specifically, coordinates are mapped in turn across a series of 3D coordinate systems: modeling coordinates to world coordinates to viewing (“eye”) coordinates to logical device space coordinates and ultimately to pixel coordinates. Students must fully understand these different coordinate systems and know which they are dealing with at various points in time when creating and manipulating data on either the CPU or in GLSL on the GPU. We continue to use our *metaview* to help students master these concepts [3].



Figure 5: Student project: Garden with thatched roof hut.

Our future plans to address the other ongoing concerns are multifaceted. We plan to incorporate specific “Modeling 101” materials to both the online web site as well as in-class discussions. While especially needed for 3D modeling, such materials will also be helpful in 2D, and the 2D materials can be constructed in such a way as to feed into the 3D version of “Modeling 101”.

We would also like to find a way to incorporate at least introductory material on some of the other programmable stages. For example, geometry and tessellation shaders could be introduced, and possibly even the new compute shaders. The course is fairly tightly packed, however, so some of these advanced topics might be relegated to “optional extensions” status of some sort and/or used by follow-on advanced graphics courses.

8. SUMMARY

We have discussed our approach to the teaching of pure shader-based OpenGL, an approach that has evolved over the past couple of years. We have found that teaching the concepts in the context of a useful software architecture – specifically, a simplified version of the well-known Model-View-Controller architecture –

has allowed us to present material typically found by undergraduate students to be quite complex in an incremental style that most find to be manageable, albeit still challenging.

This approach has also been found to have several additional benefits, for example:

- exposure to certain intermediate and advanced object-oriented concepts in a realistic setting.
- experience with the common and useful Model-View-Controller design pattern. They use our simplified version, but also learn enough of the complete pattern so that they should be able to recognize and comfortably move to this pattern if and when appropriate.
- experience with data-parallel GPU programming, GPU work assignments, and CPU-GPU coordination and communication.

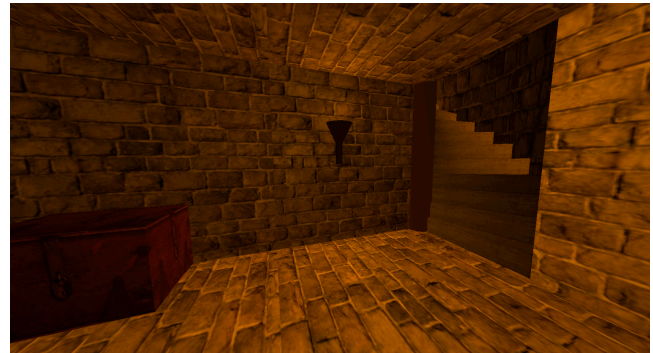


Figure 6: Student project: Castle basement with trunk.

9. ACKNOWLEDGMENTS

I am most indebted to the students who enrolled in the first couple of offerings of this new shader-based version of the graphics course. In spite of my warnings at the start of the class, their willingness to be a part of this grand adventure was truly appreciated, and their comments were taken to heart.

REFERENCES

- [1] Angel, E. and Shreiner, D., 2012. *Interactive Computer Graphics: A Top-Down Approach With Shader-Based OpenGL*. Addison-Wesley, Boston, MA. (6th edition).
- [2] Brown. 2013. *Exploratories*. Retrieved August 8, 2013 from <http://www.cs.brown.edu/exploratories>.
- [3] Miller, J. R. *metaview: A Tool for Learning About Viewing in 3D*, *SIGCSE '12: Proceedings of the ACM Conference on Computer Science Education*, Feb-March 2012, pp. 135-140.
- [4] Schreiner, D., Sellers, G., Kessenich, J. and Licea-Kane, B., 2013. *OpenGL Programming Guide*. Addison-Wesley, Boston, MA. (8th ed.).
- [5] Sealy, G., 2013. *Adding Font Support in OpenGL*, Retrieved August 20, 2013 from <http://acornheroes.com/2009/04/adding-font-support-in-opengl/>.
- [6] Talton, J. O. and Fitzpatrick, D., Teaching Graphics with the OpenGL Shading Language, *SIGCSE '07: Proceedings of the ACM Conference on Computer Science Education*, March 2007, pp. 259-263.