SIGRAD 2015
L. Kjelldahl and C. Peters (Editors)

Teaching OpenGL and Computer Graphics with Programmable Shaders

J. Nysjö[1] and A. Hast[1]

[1]Centre for Image Analysis, Dept. of Information Technology, Uppsala University, Sweden

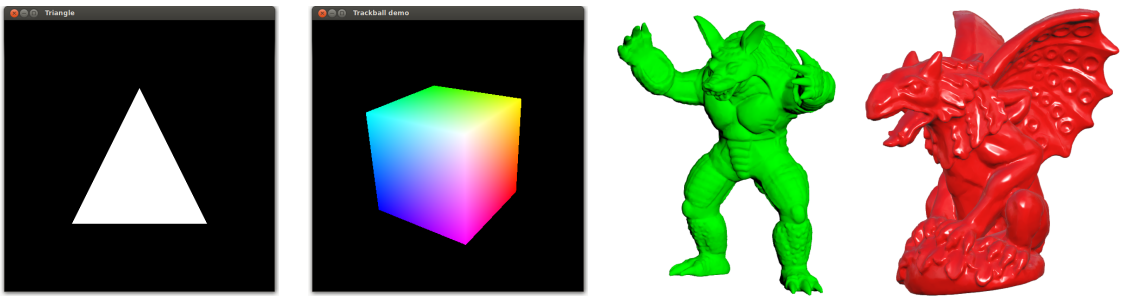

Figure 1: *Our course assignments are all implemented with modern shader-based OpenGL and range from manipulating the vertices and fragments of a single triangle to implementing a 3D model viewer with per-fragment Blinn-Phong shading.*

Abstract

This paper presents our approach and experiences of transferring an introductory computer graphics course from the fixed-function OpenGL pipeline to modern shader-based OpenGL. We provide an overview of the selected course structure and the C++-based programming environment that we use for assignments and projects, and discuss some of the technical and pedagogical challenges, e.g., multiplatform support and shader debugging, that we ran into. Based on course evaluations and the outcome of programming assignments, we conclude that introducing shaders early and skipping the fixed-function pipeline completely is a sound and viable approach. It requires more initial effort from teachers and students because of the added complexity of setting up and using shaders and vertex buffers, but offers a more interactive and powerful programming environment, which we believe helps promoting the creativity of students.

Categories and Subject Descriptors (according to ACM CCS): I.3.0 [Computer Graphics]: General—

1. Introduction

The transition from fixed-function OpenGL to programmable shaders has slowly reached academia and prompted a change in how graphics programming is taught [AS11]. One of the initial challenges in teaching an introductory computer graphics course based on modern shader-based OpenGL (which here means OpenGL version 3.x or higher) is to help the students overcome the hurdles of compiling shaders and uploading vertex data to the GPU via buffer objects. Another challenge is to set up a programming environment that supports the learning process and works on a variety of platforms and GPU configurations [PPGT14]. This paper presents our approach and experiences of adopting modern shader-based OpenGL in the introductory computer graphics course offered at Uppsala University. The course, which is supposed to represent 10 weeks of full-time studies and usually have between 50 and 70 enrolled students, is targeted to Bachelor's and Master's CS or engineering students who have taken basic courses in linear algebra and computer programming. Typically, a few of the students have prior experience of shader programming or the fixed-function OpenGL pipeline, but most are complete novices.

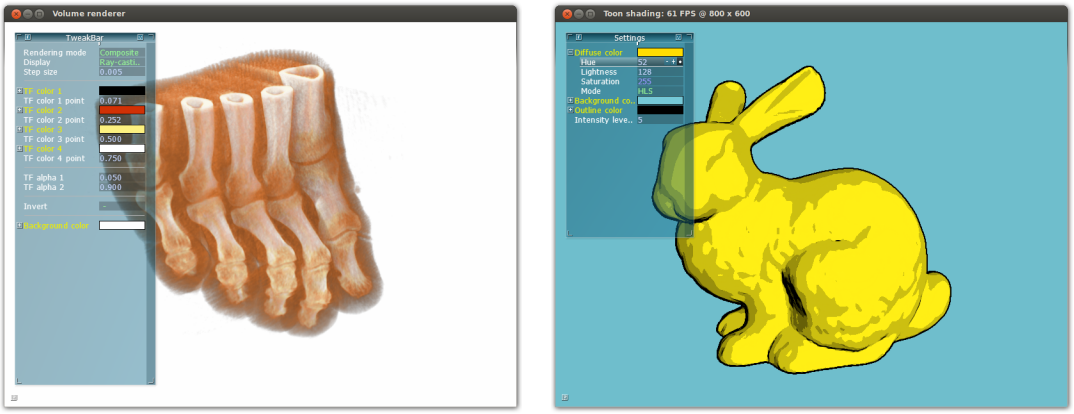

Figure 2: *Two of the six available course projects students may choose from. Left: Volume rendering. Right: Toon shading implemented as a post-processing effect. The AntTweakBar GUI library is used for interactive parameter tweaking.*

2. Course structure

Whereas shader programming was introduced as an advanced topic towards the end of our former graphics course, we now introduce it already in the second lecture, along with a simplified overview of how the programmable graphics pipeline works. We start by showing how to manipulate vertices and fragments via shaders and how to upload vertex data to the GPU memory via buffer objects, and then move on to cover transformations, viewing, and shading. Subsequently, we introduce more advanced rendering topics such as texture mapping, global illumination, and splines, along with fundamental topics such as rasterization and clipping. The fixed-function OpenGL pipeline is briefly mentioned during the course but not covered in detail.

Our course includes three programming assignments (Fig. 1) and one project. While the assignments specify exactly what the students are supposed to do to assure that they are familiar with the fundamental graphics programming tools and concepts, the project is more loosely specified. The students can choose from six different projects, which cover more advanced rendering techniques such as volume rendering and post-processing effects (Fig. 2). The projects have different degrees of specification so that the students can choose a project according to their own interest but also according to how much freedom they desire and how much specification they need. We also encourage students to suggest projects on their own. In addition to implementing a working solution, each student or project group must submit a 1-2 pages graphical abstract and a short ($\sim$1 minute) video demonstrating the solution. The compact format of the presentation enables quick assessment of the quality of the solution, which is appealing from an instructors point of view.

At the end of the course, there is a conventional exam testing the students' teoretical knowledge about the various rendering topics covered in the lectures. The final grade is determined by the grade on the exam and by eventual bonus points obtained in the assignments.

3. Programming environment

The assignments and project are programmed in C++ using the widely supported OpenGL 3.2 core profile and GLSL 1.50. Although we would have preferred to use a compatibility profile, the 3.2 core profile is the only reasonable modern OpenGL profile that is supported under Windows, Linux, and Mac OS X on up to five years old integrated or discrete GPUs. The students can work either on their own computers or on the lab computers, which are equipped with Windows 7, Visual Studio 2013, and Nvidia 6XX series GPUs. We use the FreeGLUT library for creating and managing windows (with GLFW as alternative for Mac OS X users), GLEW for loading OpenGL extensions, and GLM [Cre15] for mathematics. We also use the easy-to-integrate AntTweakBar GUI library [Dec15] to set up a widget for interactive parameter tweaking. Compilation on different platforms is enabled via the CMake [Kit15] build tool, which can generate Visual Studio project files on Windows and Unix makefiles on Linux or OS X.

Compared with the fixed-function OpenGL pipeline, modern shader-based OpenGL has, as noted in [AS11], a steeper learning curve and requires more initial programming to display something on the screen. One of the initial barriers for newcomers to modern OpenGL is to perform the somewhat complex and error-prone steps of loading and compiling shader programs and setting up vertex buffer and vertex array objects. In the first assignment, we provide utility functions for performing these tasks. This allows students to get started quickly and focus on the shader programming, without abstracting away too much of the underlying graphics API. In our experience, wrapping OpenGL calls into a custom framework of C++ classes tend to confuse students

with limited programming and C++ background. Thus, we prefer to use plain OpenGL calls, for which the students easily can find documentation and examples. The only additional utility code we provide for the assignments is a virtual trackball implementation and a simple OBJ file reader.

4. Challenges

Many of our students find it difficult to debug OpenGL applications and shader code. Shader variables can not be easily printed, and sometimes it is not obvious whether the problem lies in the host application code or in the shader code. Commenting out code and rendering vector or scalar variables as colors to the screen are the typical shader debugging techniques that we suggest for the assignments. Adding a keyboard shortcut for reloading shader programs on-the-fly is also highly useful since it allows the students to interactively modify shaders without restarting the OpenGL host application. The `ARB_debug_output` extension can be enabled to facilitate debugging on the host side. Some students have found graphical debuggers like APITrace useful later in the course.

Live coding during the initial lectures is helpful to illustrate how shaders can affect the appearance of the rendering and how the different types of shader variables are used. As a supplement to the course material, we also encourage the students to work through some of the many excellent modern OpenGL programming tutorials (e.g., [dV15]) that are available online. To avoid that the students try to use legacy OpenGL functions in the assignments, we provide a quick-reference listing the most common fixed-function OpenGL commands that are deprecated or removed in the OpenGL 3.x and 4.x core profiles.

The programming environment described in Section 3 can be a bit complex to set up for the first time. CMake is less ideal for the GUI-based workflow on Windows, and some of the students who worked on Mac OS X ran into issues with third-party libraries. A possible future direction of the course could be to move away from desktop OpenGL and C++ to WebGL and JavaScript, so that the students only would require a text editor and a WebGL-enabled browser to develop and run their OpenGL applications. This approach has been successfully adopted in, for example, massive open online courses (MOOC) [Cou15].

5. Conclusion

According to Romeike [Rom08] there are three drivers for creativity in computer science education: 1) the person with his motivation and interest, 2) the IT environment, and 3) the subject of software design itself. Allowing the students to choose a project is promoting the first driver. We strongly believe that using programmable shaders instead of the old fixed-function OpenGL pipeline promotes the second driver. Having control of powerful shaders is simply much more fun as it allows the student to try out new ideas quickly and easily and implement more sophisticated rendering techniques. The third driver is promoted by the fact that the response is immediate and graphical. The student instantly sees the result on the screen from changing the code and that itself helps the student in the exploration of different parameters and studying the result of code changes.

This is the second year we teach the revisited course, and overall the transition from the old fixed-function OpenGL pipeline to modern shader-based OpenGL has been a positive experience. Based on course evaluations, student feedback, and the outcome of programming assignments, we conclude that introducing shaders early and skipping the fixed-function pipeline completely is a sound and viable approach, particularly since the OpenGL 3.2 core profile is now widely supported. Although the shader-based approach requires more initial effort from both teachers and students, it appears to provide the students a more fundamental understanding of how the graphics pipeline works, smoothens the transition to advanced rendering topics, and pays off later in the form of more spectacular course projects. Key issues to address in the future are improved support for shader debugging and simplification of the programming environment.

6. Acknowledgments

The 3D models are courtesy of the Stanford 3D Scanning Repository and the AIM@SHAPE repository. The volume dataset is courtesy of Philips Research.

References

[AS11] ANGEL E., SHREINER D.: Teaching a shader-based introduction to computer graphics. *Computer Graphics and Applications, IEEE 31*, 2 (2011), 9–13. 1, 2

[Cou15] COURSERA: Interactive Computer Graphics with WebGL. https://www.coursera.org/course/webgl, 2015. Accessed on April 14, 2015. 3

[Cre15] CREATION G.-T.: OpenGL Mathematics (GLM). http://glm.g-truc.net/, 2015. Accessed on April 14, 2015. 2

[Dec15] DECAUDIN P.: AntTweakBar. http://anttweakbar.sourceforge.net/doc/, 2015. Accessed on April 14, 2015. 2

[dV15] DE VRIES J.: Learn OpenGL. http://learnopengl.com/, 2015. Accessed on April 14, 2015. 3

[Kit15] KITWARE: CMake. http://www.cmake.org/, 2015. Accessed on April 14, 2015. 2

[PPGT14] PAPAGIANNAKIS G., PAPANIKOLAOU P., GREASSIDOU E., TRAHANIAS P.: glGA: an OpenGL Geometric Application framework for a modern, shader-based computer graphics curriculum. *Eurographics 2014, Education Papers* (2014), 1–8. 1

[Rom08] ROMEIKE R.: Towards Students' Motivation and Interest: Teaching Tips for Applying Creativity. In *Proceedings of the 8th International Conference on Computing Education Research* (New York, NY, USA, 2008), Koli '08, ACM, pp. 113–114. 3