Teaching Programmable Shaders: Lightweight versus Heavyweight Approach

G. Scott Owen Ying Zhu Jeff Chastine Bryson R. Payne

Georgia State University Georgia State University Georgia State University Georgia College & State University

Abstract

The most exciting recent advance in computer graphics has been the development of programmable Graphics Processing Units (GPUs). We discuss different approaches to and some of the issues involved in teaching the use of GPUs.

1. Introduction

The most exciting recent advance in computer graphics has been the development of commodity level graphics hardware that supports programming. These Graphics Processing Units (GPUs) can be programmed in C-like languages for different shading models, procedural texturing, and other special effects. This presents a pedagogical challenge in how to best take advantage of these GPUs in graphics courses. This paper compares two different ways to use GPUs in courses; one is very programming intensive and the other requires a minimum of programming. Also, as with any new rapidly evolving technology, there are resource support issues and these are also discussed.

2. Background

Recently rapid technology advances have been made in computer graphics hardware, especially at the consumer level. These advances have been driven by the multi-billion dollar computer game industry that is constantly striving for faster and more realistic 3D graphics. As an example, current graphics boards priced at $200-500 are more than an order of magnitude more powerful than systems costing a hundred thousand dollars just a few years ago. The speed and capability of these boards is doubling about every nine months which is twice the rate of the traditional 18 months of Moore's law. These boards have developed to the point of being programmable and are known as Graphics Processor Units or GPUs.

University of North Carolina Professor Frederick P. Brooks, Jr., who coined the term "Computer Architecture", received the ACM Turing Award, considered the “Nobel Prize” of computing, from ACM in 1999. He also received the 2005 ACM/IEEE Computer Society Eckert-Mauchly Award for outstanding contributions to the field of computer and digital systems architecture. In his award acceptance speech, Dr. Brooks stated that GPUs are "...very powerful scientific computers installed in many homes... I think exploring that design space and its utilization... is one of the most exciting areas in computer architecture today."

The emergence of programmable commodity GPUs is a recent development. In 2001, Nvidia Corporation, released the GeForce 3 GPU with limited programmability (Lindholm et al. 2001). In the GeForce 3 series the programmer had access to only part of the geometry pipeline, i. e., vertex processing. In addition, this system allowed for no branching, and only simple if/then/else evaluation through sum-of-products testing. While all these limitations made early programs both relatively difficult and limited, the programming model had just enough capability for the exploration of vector processing on GPU hardware to begin.

Also at SIGGRAPH 2001, Proudfoot, et al. (Proudfoot 2001), presented a paper on the Stanford Real-Time Programmable Shading Project, which was the first attempt at a high-level procedural shading language for GPUs. The Stanford shading language system consisted of a high-level language and compiler front end and a retargetable compiler back end in order to accommodate various vendors' API's. While the Stanford shading language was more accessible for programmers, the level of abstraction was still relatively low. Also, the work was largely theoretical, in that it showed great promise and the feasibility of the concept but it would require significant support from several graphics vendors to make learning the language worthwhile.

Consequently, Nvidia developed a C-like programming interface to make their hardware more accessible to programmers (Mark 2003). Cg (C for graphics) was different than the Stanford shading language in that it was intended to be a hardware-oriented, general-purpose language rather than an application-specific shading language. Cg had several design goals, including ease of programming, portability, complete hardware functionality support, performance, extensibility, and support for non-shading use of the GPU.

Cg runs on any GPU from any manufacturer and works on Microsoft Windows, Mac OS X, and Linux systems. As the hardware capability increases the Cg language is also being extended. Similarly, Microsoft and Nvidia developed the High Level Shading Language (HLSL), which is compatible with Cg, in DirectX version 9.0. The OpenGL Architecture Review Board has released the OpenGL 2.0 specification that includes the OpenGL Shading Language (GLSL).

The Cg language provides structures, arrays, matrices, looping, branching, Boolean types and operators, increment/decrement and assignment expressions. Cg is very similar to the C language upon which it was based. The primary difference between the two languages is the hardware for which each was designed. Without a high-level language for implementing complex algorithms on the GPU, as is possible in Cg, it would be prohibitively difficult to enable applications to perform scientific calculations of any significant kind on the GPU. By the publication date of Mark's description of Cg, he was already able to refer to research using Cg to implement fluid dynamics simulations.

These GPUs are parallel processors and current systems can perform up to thirty two parallel operations. This makes them potentially much faster at some types of computation, e. g., ones that can be vectorized, than CPUs. Software environments and C-like languages have been developed to take advantage of the programmability of these GPUs. Computer Science researchers are beginning to take advantage of this computational power for applications beyond the rendering of images. Special purpose, non-graphics languages are being developed for general scientific

email: (owen@siggraph.org, yzhu@cs.gsu.edu, jchastine1@student.gsu.edu, Bryson.payne@gcsu.edu)

programming on GPUs, e. g., Brook [Buck 2004]. There is a website [GPGPU 2005] dedicated to General Programming on GPUs.

3. Reasons to Teach GPU Programming

One obvious reason to teach programmable shaders is to expose students to the latest developments in the field. As mentioned above, even if the students do not become computer graphics professionals, they may use GPUs in scientific programming. However there are other excellent reasons to teach them. In a standard OpenGL course the student can be exposed to different specular and diffuse illumination models, but cannot implement them because the pipeline is fixed to Gouraud shading of the Phong Illumination model with Lambertian diffuse shading. But using programmable shaders, the students can implement these algorithms themselves. They can also implement the Phong/Lambertian model in a fragment shader to get per pixel shading.

They can also study and implant more advanced specular illumination algorithms such as the Blinn-Fresnel algorithm [Blinn 1977] or anisotropic specular shading [Ward 1992, Heidrich 1998] Similarly they can implement better diffuse illumination algorithms such as the Oren-Nayer [Oren 1994] for rough surfaces, such as clay or the moon. There are many procedural texturing and modeling methods that can be implemented on the GPUs, e. g., as in [Ebert 2003].

4. Approaches to Teaching Programming on GPUs

We are assuming a second course in Computer Graphics and that the first course was a standard course using OpenGL and C/C++. Our second course is the graduate course and so not all of the students will have had a first course using OpenGL. This is especially true of non-US students who have taken their undergraduate courses elsewhere. We are also assuming a focus or at least significant emphasis on procedural texturing and modeling, as in [Ebert 2003]. In such a course there are several choices to make, which are discussed below.

The first choice is which shader language to use. The three main Graphics shader languages, Cg, HLSL, and GLSL all have advantages and disadvantages. We chose to go with Cg. It is identical to HLSL but is multi-platform and there are many excellent tutorial books and examples for Cg and HLSL. In our course we used "The Cg Tutorial" [Fernando 2003]. Another good book is [Fosner 2003]. It is for DirectX and HLSL and is available on line to ACM members. There is a good book on OGSL by [Rost 2004].

A second choice is what level of hardware support to require. This can be defined by the level of DirectX vertex shader and/or pixel shader supported by the GPU. In our course we required VS 2.0 and PS 2.0 since this level can run everything in the Cg Tutorial book and is quite capable of complex shading programs. This corresponds to an Nvidia Fx 5XXX GPU or an ATI 9XXX GPU. The current GPUs by Nvidia (the 6XXX series) support VS 3.0 and PS 3.0. Our students all own their own computers and prefer to do their work at home. If they had a recent PC then they probably already had this level of graphics board. If they had to purchase a new one they could obtain an Nvidia or ATI board for the course for about $80.00.

There are some tools that students can use to create shaders. Nvidia has both the Cg Browser and FX Composer (HLSL only) (Nividia 2005). ATI (ATI 2005) has RenderMonkey (for HLSL and GLSL). All of these tools are Microsoft Windows specific. However, they make it easier to develop shaders and then to export them for use in other programs. These tools also allow the instructor to easily show the programming code and visual results of the algorithms that are discussed in class. However, these tools are not very good for debugging complex code.

4.1 Heavyweight Approach

A programming intensive approach to teaching Programmable Shaders would be to use a traditional graphics API, such as OpenGL or Direct X. We characterize this as the “heavyweight” approach. Most introductory computer graphics courses use OpenGL with C or C++ since it is well established and platform independent. In order to integrate Cg program with their OpenGL or DirectX programs, students can use either Nvidia’s Cg Runtime runtime library, DirectX native API, or OpenGL extensions. Currently, Cg Runtime is the easiest to use and it is cross-platform. If you were to just have a brief taste of programmable shaders in a course, then having the students integrate it into their OpenGL programs is still difficult because of the OpenGL setup required. We have used programmable shaders this way, i. e., the bulk of the course uses OpenGL and just one or two assignments involve GPU programming.

However, if the focus of the course is on the more advanced techniques that can be done on GPUs, then using OpenGL with all of its complexity becomes even more of a burden.

4.1.1 Advantages to Using OpenGL

- Students gain more experience in general C/C++ programming and in using OpenGL
- Easier to incorporate other, non-GPU, advanced algorithms in the course, e. g. different geometry based algorithms such as surface subdivision.
- More flexibility in turning on and off shaders at runtime.
- More flexibility in passing parameters to shaders at runtime.

4.1.2 Disadvantages to Using OpenGL

- The added complexity of OpenGL programming means that less time can be spent on GPU related algorithms and programming.
- Some of the students may be learning OpenGL along with Cg.
- In order to use Cg with OpenGL, students need to learn either the Cg Runtime library or OpenGL extensions. This may decrease their time to study the shading techniques and Cg language.

4.2 Lightweight Approach

An alternative approach, that we characterize as “lightweight”, is to use X3D/VRML to teach programming GPUs. The X3D Working Group on Programmable Shaders (Web3d 2005) has developed proposals for incorporating programmable shaders into X3D. The vertex and fragment shader code is placed in a ShaderAppearance node that replaces the normal Appearance node. Thus it is very easy to incorporate a shader into a VRML world. There exist many resources for learning VRML such as (Nadeau 1998), which is an introduction and [Carey 1997] which is an excellent on line reference manual.

Currently the only browser plugin that supports programmable shaders is BS contact [Bitmanagement 2005]. BS contact runs only on Microsoft Windows and works in both Microsoft Internet Explorer and Firefox. There are multi-platform Open Source browsers that support X3D, e. g., FreeWrl (FreeWrl 2005) and it is hoped that they will soon incorporate support for the programmable shaders. The BS contact browser plugin supports the original 2003 X3D Programmable Shaders proposal [Carvalho 2003]. This plugin supports either X3D or VRML files. For simplicity the students used VRML files.

As discussed above, there are many tools to create shaders and also there are many shader examples on the Web. Since this was a computer science class, we wanted the students to have some experience in programming their own shaders. So, we did not introduce them to the above tools until later in the course. As an early assignment they were asked to implement the Blinn specular model (Blinn 1977) and the Oren-Nayer diffuse model (Oren 1994).

Later assignments had students create VRML worlds and use the shaders in these worlds. The students could use the package of their choice in creating them and most used [Blender 2005], which is a free cross-platform modeling and animation package that will export VRML and/or X3D files. They then insert their shader code into the VRML files.

Since the students are using modeling packages, it is fairly easy for them to create complex VRML Worlds to test their shaders.

4.2.1 Advantages to Using X3D/VRML

- Students learn about X3D/VRML. A recent survey of Google hits for 3D Web technologies showed that approximately 83% of all current 3D Web content was VRML and 11% was X3D. Of the other two major entries, Macromedia Shockwave 3D had 4% and Adobe Atmosphere (which has been discontinued) had 2%. However, the students did not need to understand VRML in depth as they could just insert their shader code into existing VRML files.
- Students can use existing complex VRML worlds or can easily create new ones using Blender. This is much easier than importing models and creating worlds in OpenGL.
- Students learn a modeling and animation package such as Blender.
- Testing the results of the shaders is much easier since the students can follow this process (with no program compilation):
    - write the shader
    - apply it to an object in VRML
    - load the VRML world and view the result
    - modify the shader
    - reload the VRML world

4.2.2 Disadvantages to Using X3D/VRML

- Currently no cross-platform browser support (but most students use Microsoft Windows).
- Less flexible runtime control of the shaders.

4.3 Middleweight Approach

There is yet another approach to teaching programmable shaders that combines some of the favorable elements of both of the above. This approach, that we will characterize as "middleweight", is to use Python. Python (Lutz 2004 and Python 2005) is an Object-Oriented language similar to C++ but it is interpreted rather than compiled. Python's features are midway between scripting languages, such as Perl, and a systems language such as C. Python is widely used in the scientific community, for example its application in computational molecular biology (BioPython 2005).

Python is cross-platform and implementations exist for Microsoft Windows, Mac OS X, and Linux. There exists an OpenGL binding for Python (PyOpenGL 2005). There is a recent effort to enable Python to use programmable shaders (GLEWpy 2005). We have not yet used Python in our courses, but may in the future.

5. Conclusion

We have incorporated GPU programming into both our first and second graphics courses. The first undergraduate course uses OpenGL and has one or two assignments doing GPU programming. The second, graduate, course uses X3D/VRML and makes extensive use of GPU programming. In this paper we have compared that approach to the second course with a potential one that uses OpenGL and GPU programming.

Since no paper should be published at SIGGRAPH without an image, in Figure 1 we show a screenshot from a VRML file of a teapot shaded with a modified blue-white marble texture from [Ebert 2003].

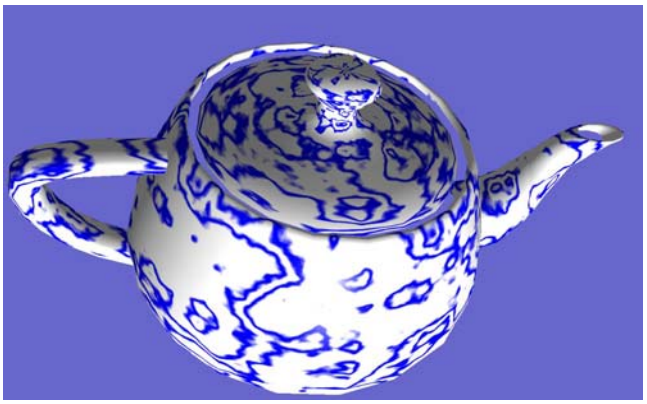

Figure 1. VRML Teapot with blue-white marble procedural texturing.

References

BLENDER, 2005, http://www.blender.org/

BioPython, 2005, http://www.biopython.org/

Bitmanagement, 2005, http://www.bitmanagement.com/

BLINN, J. F. ,"Models of Light Reflection for Computer Synthesized Pictures", **SIGGRAPH 77 Conference Proceedings**, pp. 192-198.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., and HANRAHAN, P., "Brook for GPUs: Stream Computing on Graphics Hardware", **ACM Transactions on Graphics**,

23:3, pp. 774-783. Proceedings of ACM SIGGRAPH 2004.

CAREY, R. and BELL, G. “The Annotated VRML97 Reference Manual”, http://www.cs.vu.nl/~eliens/documents/vrml/reference/BOOK.HTM

DE CARVALHO, G. N. M., COUCH, GILL, J. T., and PARISI, T., “X3D Programmable Shaders”, http://www.bitmanagement.com/developer/contact/examples/shader/spec/X3DProgrammableShadersProposal.htm

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., and WORLEY, S., Texturing and Modeling: A Procedural Approach, 3rd Edition, Morgan Kaufmann, 2003.

FOSNER, R., **Real-Time Shader Programming**, Morgan Kaufmann Publishers, 2003. This book is available online at the ACM Professional Development Centre (www.acm.org)

.

GLEWpy 2005, http://glewpy.sourceforge.net/

GPGPU, 2005, http://www.gpgpu.org

HEIDRICH, W. and SEIDEL, H.-P., “Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware”, Image and Multi-dimensional Digital Signal Processing Workshop, 1998.

LUTZ, M. and ASCHER. D. , “Learning Python”, O'Reilly, 2nd ed., 2004.

NADEAU, D. R., MORELAND, J. L., and HECK, M. M., “Introduction to VRML 97” SIGGRAPH 98 Course Notes at http://www.siggraph.org/education/materials/siggraph_courses/S98/18/vrml97/vrml97.htm

OREN, M. and NAYAR, S. K. "Generalization of Lambert's Reflection Model", **SIGGRAPH 94 Conference Proceedings**, pp. 239-246.

Python, 2005, http://www.python.org

PyOpenGL, 2005, http://pyopengl.sourceforge.net/

ROST, R., **OpenGL Shading Language**, Addison-Wesley, 2004.

WARD. G. J., “Measuring and Modeling Anisotropic Reflection”, **SIGGRAPH 92 Conference Proceedings**, pp. 265-272.

Web3d, 2005, http://www.web3d.org/x3d/workgroups/x3d-shaders.html