colorlinks, linkcolor=red!50!black, citecolor=blue!50!black,
urlcolor=blue!80!black

Object Oriented GPU Programming Using CLOS

Nicolas Hafner
Shirakumo.org
Zürich, Switzerland
shinmera@tymoon.eu

ABSTRACT

In this we present a new approach to applying object oriented principles to GPU programming via GLSL shaders. Shader code is attached to classes within standard Lisp code and automatically combined according to inheritance rules using the CLOS Meta Object Protocol. This results in a native coupling of both CPU and GPU logic, while preserving the advantages of object oriented inheritance and behaviour.

CCS Concepts

•**Software and its engineering** → **Object oriented architectures;** *Abstraction, modeling and modularity; Object oriented frameworks;* Parsers; •**Computing methodologies** → *Computer graphics;*

Keywords

Common Lisp, GLSL, OpenGL, GPU, CLOS, Object Orientation

ELS'18 April 16–17, 2018, Marbella, Spain

ACM ISBN .

DOI:

1. INTRODUCTION

In modern real-time computer graphics applications, advanced effects and GPU computations require the use of a programmable graphics pipeline. This pipeline is usually programmable using a specific language that is compiled to GPU instructions by the graphics card drivers.

For OpenGL, this language is called GLSL[1] and follows a mostly C-like syntax. A program written in GLSL is called a shader and is used to fill one of several stages in the graphics pipeline. Whenever a primitive is rendered, these shaders are executed on the GPU, each providing a particular stage of processing until finally an image is produced.

However, only a single shader can live in a particular stage at a time. This presents an issue for modularity, as effects represented by shaders cannot be easily combined. Instead, it is usually the task of a programmer to craft a single shader for each stage that produces the desired results.

In this paper we present a new approach to this problem that is accomplished in two steps. The first step is the parsing and manipulation of native GLSL code. The second step is the integration of shader code into the Common Lisp Object System to allow for automatic combination through inheritance.

2. RELATED WORK

Several different approaches to shader combination exist today.

Trapp et al[2] use additional constructs introduced to GLSL and a preprocessor to combine effects. This differs from our approach in that we do not extend GLSL syntax in any way and do not present a standard interface to use between shader fragments.

McCool et al.[3] present an extension of the C++ language to allow writing GLSL-like code in C++ source. Shader fragments are parsed into an abstract syntax that can be used to perform static analysis and combination of fragments. Combination is however not automatic and needs to be explicitly requested by the programmer.

Kuck[4] presents an evolution of McCool's approach by allowing the use of C++ method declarations for shader function definitions. In this approach method dispatch is mirrored in the emitted GLSL code using unique identifiers

for each class.

In VTK[5] GLSL's ifdef precompiler macros are used to conditionally exclude or include certain shader functionality. This means that a full shader program of all possible combinations is meticulously hand-crafted, and features are then added or removed as needed by the program by prepending appropriate preprocessor definitions to the shader source.

A GLSL extension[6] for an include preprocessor directive that would splice other shader source files in at the requested position, similar to C's include facility. This would merely allow controlling concatenation of source text within GLSL itself, though.

3. GLSL PARSING AND MANIPULATION

4. SHADER INTEGRATION WITH CLOS

5. CONCLUSION

6. FURTHER WORK

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] Randi J Rost, Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL shading language*. Pearson Education, 2009.

[2] Matthias Trapp and Jürgen Döllner. Automated combination of real-time shader programs. In *Eurographics (Short Papers)*, pages 53–56, 2007.

[3] Michael D McCool, Zheng Qin, and Tiberiu S Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68. Eurographics Association, 2002.

[4] Roland Kuck. Object-oriented shader design. In *Eurographics (Short Papers)*, pages 65–68, 2007.

[5] Inc. Kitware. Visualization toolkit.

[6] Jon Leech. $\mathrm{Gl}_a rb_s hading_l anguage_i nclude$, 2013.