Joseph Seaton

Shader Compositor

Part II Computer Science

Fitzwilliam College

March 15, 2012

Proforma

Name:	**Joseph Seaton**
College:	**Fitzwilliam College**
Project Title:	**Shader Compositor**
Word Count:	**0**
Project Originator:	Christian Richardt
Supervisor:	Christian Richardt

Original Aims of the Project

To implement a novel user interface for the creation, testing and easy modification of pipelines of shaders. To automatically detect shader parameters and provide a simple interface to change them. To apply some optimisations to parts of this process.

Work Completed

The core of the project has been completed and works satisfactorily. Shaders can be inputted, and a pipeline of shaders can be specified using Javascript, the result of which is shown in-browser. An interface for modifying shader parameters is correctly generated for [TODO: all] most types of parameter. A number of optimisations have been applied, including elimination of unnecessary shader recompilation and pipeline re-generation. [PENDING] The project is also now capable of reusing FBOs between pipeline changes where possible.

Special Difficulties

None.

Declaration

I, Joseph Seatonof Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

This document owes much to an earlier version written by Simon Moore [1]. His help, encouragement and advice was greatly appreciated.

Chapter 1

Introduction

My project provides a novel user interface for easily compositing multiple shaders together, in which individual shader parameters can be easily modified and the results viewed immediately. I have successfully implemented the core of my project, including a number of proposed extensions.

1.1 Motivation

Working with OpenGL is usually undeniably messy. OpenGL is a verbose and complicated API. When developing shaders, developers tend not to be interested in writing large amounts of OpenGL API code just to tweak shader parameters or forward the texture output of one shader to the input of another. But this is exactly what developers have to do. Given that many graphics applications are written in compiled languages like C and C++, this significantly increases the length of a development cycle. Given that a large part of shader development constitutes small tweaks to (often subjectively) improve output, this is not ideal.

For students learning about shaders, OpenGL is often an intimidating and confusing mess that gets in the way of understanding a fundamentally quite simple concept.

This project aims to provide a way of developing shaders that is both useful for developers, and may be used as a tool for students learning about shaders for the first time.

1.2 Brief introduction to Shaders

TODO: move to under background reading?

Chapter 2

Preparation

2.1 Development Environment

2.1.1 WebGL

WebGL is a very recently developed API allowing web-based applications access to OpenGL contexts, via Javascript.

2.1.2 Emacs

2.1.3 node.js

node.js is a Javascript platform build on top of Google's V8 Javascript engine. It provides among other things, a command line interface, including a REPL (read-evaluate-print-loop). This provides a fast, light-weight way to test non-browser-bound code using traditional command line tools, can be easily interfaced with git and cron, and can easily write to files. All of this can be done without requiring messy extra code to interact with the webbrowser or server-side code to report back statistics.

2.1.4 Google Chrome

WebGL naturally precludes the use of a webbrowser. I have found for my particular Operating System / hardware combination, WebGL on Google Chrome is more stable. Since WebGL is a relatively recent development that has only recently gained browser support, WebGL can still be unstable and buggy in some circumstances even under Google Chrome. Google Chrome also includes a set of 'developer tools' including

2.1.5 WebGL Inspector

While the premise of the WebGL inspector tool - a Google Chrome extension that provides inline information on current WebGL contexts - is promising, I found the tool to be too unstable to be helpful in practice.

2.2 Software Development Process

Test-driven development Evolutionary

2.3 Preparatory Learning

2.3.1 Background Reading

OpenGL and GLSL

Prior to beginning this project, I had very little knowledge of OpenGL barring a small amount of personal experience writing toy programs in my spare time. As such I needed to gain a better understanding of OpenGL, and some knowledge of GLSL before I could begin this project. For this purpose I obtained a copy of the famous 'Red Book' [REF]. I also read through the GLSL ES specification provided by the Khronos Group [REF], which discusses the syntax of GLSL in detail.

Javascript

While, like many people I already have some experience of Javascript programming from web development work, I thought it would be wise to refresh myself before starting on a more complex project such as this one.

2.4 Libraries

TODO: is this the right place?

2.4.1 JQuery

2.4.2 WebGL Toolkit

Given that OpenGL itself is quite awkward to work with, I decided to use an existing library to abstract away some of the uglier parts. However, there are currently many libraries available claiming to do exactly this, of varying degrees of completeness and varying levels of abstraction. Initially I settled on three.js [REF] on the basis of its popularity, and high rate of development. However it soon became apparent that in the level of abstraction provided by three.js actually made the task more difficult, given that interacting with FBOs was non-obvious. After some more searching I came across GLOW [REF], a toolkit specifically designed to make working with shaders simple, but otherwise providing little abstraction.

2.4.3 GUI Toolkit

Editor

A core aim of the project was to provide good GLSL syntax highlighting. There currently exist many web-based editors offering some degree of syntax highlighting, but I chose to use CodeMirror [REF], given its active development, and the presence of a modular way to add support for highlighting new grammars.

2.5 Version Control and Backup Strategy

All project files, including the dissertation, were placed under git version control. Git was then configured to push to multiple remote repositories in different locations: an external harddrive, my PWF account, my SRCF account, my web hosting account, and GitHub. While this could be considered excessive, the use of SSH keys meant that post-setup, this required no extra effort on my part. Since I am conscious that git is capable of history rewriting, I also configured a regular cron job to make copies of my current repository using git bundle.

Chapter 3

Implementation

3.1 WebGL

TODO: where to talk about this.

3.2 Parser

It was decided that due to time constraints, and the lack of a pressing need for an unusual parser, a parser generator would be used. Initially I chose to use JS/CC [REF], since it seemed well-documented and quite popular. However, once I had used it to generate an appropriate parser for GLSL, it quickly became apparent that the parser was insufficiently fast to use for such a complicated grammar. In fact, I was unable to obtain any timing for the parser, as under Google Chrome, even with the simplest inputs it would fail to produce any output before Chrome itself decided to kill the process. While it may have been possible to obtain timing information, this was clearly to slow to be of any value. I then found Jison [REF], a JavaScript port of Bison. This was able to parse even quite complicated input in a reasonable amount of time, as is discussed in my evaluation.

3.3 Testing

3.4 Optimisation

3.5 User Interface

Chapter 4

Evaluation

4.1 Parser Correctness

4.2 Pipeline Correctness

4.3 Speed

Chapter 5

Conclusion

5.1 Main Results

The project has been successful. The core of the project has been completed and functions correctly, and with the addition of the various optimisations and extensions, is quite usable.

5.2 Lessons Learned

5.3 Further Work

User study.

Both the specification and implementation of WebGL have changed significantly since their first proposal, and I expect them to continue to do so for some time yet. As such further work may be necessary in the future as these change. Of particular note is the current lack of multiple render targets in current implementations [REF]. If in the future implementations begin to support multiple render targets, adding support for these to my project will be a moderately non-trivial task.

Bibliography

[1] S.W. Moore. How to prepare a dissertation in latex, 1995.

Appendix A

Project Proposal

Part II Computer Science Project Proposal

Shader Compositor

Joseph Seaton, Fitzwilliam College

Originator: Christian Richardt

21 November 2000

Special Resources Required

Project Supervisor: Christian Richardt

Director of Studies: Dr R. Harle

Project Overseers: [TODO]

Introduction

Designing shaders can be laborious work, involving endless back and forth between tweaking the code (often minor visual tweaks) and examining the visual output. While programs exist to provide previews of individual shaders, modern software, especially for computer game engines, often involves multiple shaders chained together, and little software exists to aid shader writers in testing such shader pipelines. Furthermore for beginners, the initial process of learning of how to use shaders is complicated by this extra legwork to compose shaders. The aim of this project is to alleviate these problems to some degree.

A.0.1 Aside on Shaders

A 'shader' is a piece of code that runs directly on a GPU. Shaders work on the data flow model, in that once the code is set up, many data may be fed through it. In OpenGL, which this project will focus on, such shaders are divided into a number of types. The two types considered here are vertex shaders and fragment shaders although there are others including geometry shaders which could be considered as an extension. Vertex shaders operate on a vertex, and its associated data, performing such operations as transforming a vertex to the screen coordinates. Fragment shaders (approximately speaking) operate on individual pixels, and are often used for operations such as texturing a model. Vertex shaders may pass information to fragment shaders, and the underlying program using the shaders may pass information to both.

A.0.2 The Project

This project intends to provide an interface to allow a user to create a set of shaders, written in a slightly annotated version of GLSL. The attributes of these shaders will be detected, along with the type of each attribute, and, given a specification of how to connect these attributes (provided either via a GUI or a simple language), a pipeline of these shaders will be constructed. It should be noted that the construction of said pipeline will require a number of stages, namely construction of a simple GLSL parser, generation of a DAG of shaders, linearisation of said DAG, and finally application of this DAG to FrameBuffer Objects for offscreen rendering, finishing in some rendered model. Since FBOs can be reused, an efficient mapping is non-trivial. Furthermore, there is room for optimisation of this DAG consider unification of identical textures, or extraction of common code. The interface should also provide some facility to allow the user to easily modify shader parameters by e.g. selecting textures, or setting the colour of a vector. Addition of simple annotations to the GLSL may be useful here, such as specifying the range of a vector. Since this project is intended to be used partly as a learning tool, I propose to use WebGL, a standard for using OpenGL (including GLSL) on the web, via JavaScript. Since WebGL is cross platform by design and requires only a recent webbrowser to use, As WebGL is quite a new technology, it is possible that this may turn out to be infeasible, in which case I would use Python.

A.0.3 Resources Required

Since this project will involve OpenGL shaders, a recent graphics card supporting a recent version of OpenGL will be required. To this end I would use my own computer with its AMD HD5450 graphics card. Depending upon the speed at which my test harness runs, I may consider upgrading to a more powerful card such as the HD6770.

A.0.4 Starting Point

I have some existing knowledge of OpenGL and GLSL from personal experience, but only to the extent of small personal projects. I have a reasonable amount of experience with JavaScript having used it in the past for web development work but in conjunction with a server-side language. Substance and Structure of the Project The project involves writing software providing a user interface in which to input a set of shaders and the connections between them and view a sample output of the generated shader pipeline on a sample model. Most notably to do this the project requires software to be written that given a set of shaders and a specification of how to connect them, can parse each shader, extract its parameters, and construct the specified pipeline. Furthermore the user interface should provide the ability to easily vary the parameters of each shader on the fly. This and the above comprise the core of the project. By way of extensions, there is much scope for fleshing out the interface into a fully-featured development tool, with such additions as example shaders, a more complete set of sample scenes, and so on. There is also some room for optimisation of the composition process, for example the unification of identical textures or extraction of common code between shaders. For evaluation, in order to demonstrate correct composition, a testing framework will be written that takes a set of semantically obvious or simple shaders and composites them in arbitrary ways and tests whether the output matches the expected result. Also pipeline speed/memory use comparisons of before/after the implementation of the extensions will be performed, along with comparisons of recomplilation speed before/after optimisation (i.e. detection of modified shader). Further comparisons include comparisons of a generated pipeline against a hand-coded shader, or a comparison of the time complexity of a given pipeline to a JavaScript program carrying out the same task.

A.1 Success Criteria

The following should be achieved:

1. Interface to construct shaders, specify connections between shaders
2. Detection of parameters of set of shaders, parameters presented to the user somehow
3. Sample scene output for single shader
4. Demonstration of correct composition (with sample scene output) for various shaders with pipeline specifications
5. Provide basic syntax highlighting of GLSL in interface

Extensions: Demonstration of some optimisations:

1. Reuse of FBOs in shader pipeline
2. Automatic unification of identical textures
3. Splitting shared code off into separate shaders
4. Avoiding recompiling shaders unnecessarily

Demonstration of some additional GUI work:

1. Presentation of sliders with range detected from range annotations in shaders,
2. Some facility for animation (e.g. parameters that are automatically incremented)
3. Shader and/or scene test suite

A.2 Timetable and Milestones

A.2.1 Weeks 0-2: 22nd October 11th November

Initial reading. In particular, the OpenGL ’red book’. Setup of barebones GUI, associated libraries. Preparation of automatic backups. Milestone: dummy interface for single shader.

A.2.2 Weeks 3-5: 12th November 2nd December

Construction and testing of GLSL parser. Further familiarisation with GLSL. Milestone: ability to parse simple GLSL

A.2.3 Weeks 6-8: 3rd December 23rd December

Write shader DAG generating code, DAG linearisation code. Initial work on application to FBOs. Milestone: generation of linearised shader DAGs

A.2.4 Weeks 9-11: 24th December 13th January

Complete actual pipeline given some DAG, the project should be able to render the pipeline to a quad. Complete interface such that the pipeline order can be specified, either via a GUI or via some written specification. Basic parser may be sensible. Milestone: ability to render a given shader pipeline

A.2.5 Weeks 12-14: 14th January 3rd Feburary

Preparation of progress report. Presentation of shader parameters in GUI. Initial work on texture unification and other optimisations. Initial work on test suite. Milestone: Progress report prepared. Ability to vary shader parameters from the GUI. Ability to unify some identical textures (or reduced FBO or code usage, depending on extension). Some tests written.

A.2.6 Weeks 15-17: 4th February 24th February

Presentation of progress report. Further testing of core functionality, expansion of test cases. Bug fixing. Possibly initial optimisation work. Milestone: Progress report given. Large test suite. Lack of obvious bugs in core.

A.2.7 Weeks 18-20: 25th February 16th March

Additional GUI work and/or optimisation. Slack time for unexpected problems. Optimisations can be skipped if more time is needed for core work. Milestone: Better featured GUI and/or some further optimisations.

A.2.8 Weeks 21-23: 17th March 6th April

Begin work on dissertation. Programming work should be finished by now, although some expansion of the test suite may be acceptable. Milestone: complete initial sections of dissertation.

A.2.9 Weeks 24-26: 7th April 27th April

Finish first draft of dissertation.

A.2.10 Weeks 27-29: 28th April 18th May

Proofreading and submission.