Joseph Seaton

Shader Compositor

Part II Computer Science

Fitzwilliam College

April 4, 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 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] Use of shader program vs shader is messy and technically incorrect throughout. In their modern incarnation, OpenGL shaders are quite a simple

concept to understand at a high level. A shader is a simple program - 'simple' referring to certain constrains we shall ignore - written in a dataflow style. Such programs take in one item of data at a time, for example a vertex, and output another item of data - the new vertex location, or a pixel value. In a modern OpenGL implementation, there are two commonly used types of shader (and a few more which will not be discussed). These are vertex shaders, and fragment shaders.

1.2.1 A look at some OpenGL calls

As discussed above, the use of shaders sounds simple. However, as we can see from [REF listing], a short except of the OpenGL calls from an actual WebGL program [REF], the paraphanalia of mostly boilerplate OpenGL calls required for compiling shader programs, specifying parameters and so on can result in a very large codebase for even the simplest program.

```
rawData = new Uint8Array(noisepixels);
texture_noise_l = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture_noise_l);
gl.pixelStorei(gl.UNPACK_ALIGNMENT, 1);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, sizeX, sizeY, 0, gl.RGBA, gl.UNSI
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
```

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 [REF]. It was decided that the use of web-based technologies would help lower the barrier for use of this project, and therefore shader development in general, since a user needs only visit the correct page using a modern webbrowser. Using WebGL has the added advantage of requiring less familiarisation given my prior experience with Javascript.

2.1.2 Emacs

Surprisingly for such an old editor, with the correct extensions Emacs is very well equipped for modern Javascript development. While there are other editors availiable that could claim most or all of the features these extensions add, my previous experience with Emacs makes this a favourable choice. The extensions used add auto-completion, syntax checking, and an inline node.js (see below) console [REF http://blog.deadpansincerity.com/2011/05/setting-up-emacs-as-a-javascript-editing-environment-for-fun-and-profit/].

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

Given my own lack of familiarity with the technologies used, and the somewhat experimental nature of WebGL, an iterative development process was deemed necessary.

It was decided to use a combination of a test-driven and evolutionary software development model.

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. I found particularly useful [REF http://eloquentjavascript.net] and [http://w3future.com/html/stories/hop.xml], both of which are unusual among texts on Javascript in being willing to discuss the more sophisticated functional aspects of Javascript, and their relation to Javascript's own unusual prototype-based object system.

2.4 Libraries

TODO: is this the right place?

2.4.1 JQuery

JQuery is a general utility library for Javascript, providing many helper functions not provided by browsers, and also greatly simplified DOM (Document Object Model) manipulation

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 General Outline

The project consists of the following parts:

Parser

The parser extracts relevant parameters from a given shader.

Pipeline Specification

The user-specified shader tree [TODO use shader tree rather than pipeline more] is extracted and converted into a usable format

Parameter UI Generation

An appropriate user interface is generated based on the pipeline specification and the shader parameters extracted by the parser.

Pipeline Generation

A list of shaders is generated from the shader tree such that the shaders can be rendered in sequence.

Pipeline Initialization

Actual GLOW shader objects are created. FBOs are assigned to shaders as necessary.

3.2.1 Rendering

Render the actual pipeline, resulting in a preview image in the UI.

3.2.2 User Interface

3.3 Parser

The main job of the parser is to take a shader program, and provide an array of parameter names and their corresponding data types, which must be supplied to a shader program for it to run. The parser may also return information for the syntax highlighter.

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.

Struct Parsing

The GLSL Specification [REF] allows a shader program to define and use new types of structures. Structure names and field names are subject to the same restrictions as normal identifiers. This makes parsing GLSL as parsed by e.g. Google Chrome technically context-sensitive. However, the parser used, Jison, only supports LALR grammars. The grammar as provided in the specification introduces extra tokens for structure and field names, ignoring the context sensitivity. Therefore by way of an initial, straightforward implementation of struct parsing, I chose to make the parser simply append new structure/field names to

the appropriate part of the matching logic of the lexer. While this will fail to parse shaders that use e.g. some field name as the name of a variable,

3.4 Pipeline Specification

The purpose of the pipeline specification stage is to take input from the user specifying the shaders to be used, and the connections between them - that is, when a shader uses the output FBO of a previous shader as an input texture. For the actual pipeline specification, I could either develop my own simple specification language, or reuse some existing language. Since this project uses WebGL, it was decided to use Javascript. This had the added bonus of enabling the user to specify a more dynamic pipeline that takes e.g. browser differences into account, with no extra work. However, the objects used internally by my project are sufficiently complicated that I did not want to expose them directly to the user. Therefore, a simple environment containing methods for creating dummy shaders is created. These dummy shaders can then be converted into the internal format later.

3.5 Parameter UI Generation

This stage takes a tree of dummy shaders, and returns the appropriate HTML user interface. For each parameter of the shader which needs specifying - that is, is of type [TODO typeset]uniform and is not already specified - it produces an appropriate set of input elements. In the initial implementation, the appropriate shader values would be updated when a new render was requested. However this is inefficient, and makes the UI less responsive. Later implementations therefore use a different method: each input element has associated event listeners that will update the value within the shader, and correctly invalidate the GLOW cache and request the preview be re-rendered.

3.5.1 Struct Handling

Since GLOW expects structure parameters to be specified by ordinary Javascript objects, which are also used to specify parameters, structs can be easily and simply handled by recursing on the parameter generation function. While for languages like C, this could produce an infinite loop if the structure references itself, GLSL does not permit this. However [TODO test this].

3.6 Pipeline Generation

The primary task of the pipeline generation stage is converting the shader DAG into a list. This can be acheived using a topological sort. Initially the algorithm decribed by Cormen et al [REF Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "Section 22.4: Topological sort", Introduction to Algorithms (2nd ed.), MIT Press and McGraw-Hill, pp. 549552, ISBN 0-262-03293-7.] was used. The algorithm is shown in 0.

$L \leftarrow$ Empty list that will contain the sorted nodes
$S \leftarrow$ Set of all nodes with no outgoing edges
for all node n in S **do** visit(n)
end for
function VISIT(node)
    if n has not been visited yet **then** mark n as visited
        for all node m with an edge from m to n **do** visit(m)
        end foradd n to L
    end if
end function

However, the algorithm shown in 0 cannot detect when the graph contains a loop - in such a case, the program should produce an error.

3.7 Pipeline Initialization

In this stage, each shader object in the pipeline must be assigned an actual GLOW shader. This is the object that contains the actual compiled shader that will be called at rendertime. Initially, this was done in the most straightforward manner possible, by simply iterating through the pipeline and generating new GLOW objects after any modification. This was later modified such that changing parameters would not require this - GLOW objects are created before the Parameter UI Generation stage, with dummy parameter values.

3.8 Rendering

The rendering process itself is actually quite straightforward since we try to offload as much work as possible to other stages. This is important for later extensions involving animation, where we want the render loop to run as quickly as possible. The render loop simply iterates through the shader pipeline list, binding the FBO associated with each shader, rendering the shader, and then

unbinding the FBO. The FBO binding is skipped for the final shader. While the initial render function cleared the GLOW cache at the start of each call, we can avoid this as discussed under [REF glow cache]

3.9 Testing

Parser Correctness

In order to assist with development of the parser, and in particular to assist in regression testing, a test harness was developed for the parser. Since the parser does not access any browser-specific APIs, it was possible to perform this testing in a simple, automated way on the command line.

3.10 Optimisation

3.10.1 GLOW Early Initialization

As discussed in [REF UI params, Pipeline init], in the initial implementation all stages following Parameter UI Generation must be called, at the users request, following each change to a parameter. This also includes a step in which every parameter in the UI is updated. This is obviously suboptimal, both since all parameters must be updated and redundant steps be re-run, and since the user is required to request re-rendering, which reduces the level of interactivity. The solution to this problem is briefly mentioned in [REF pipeline init].

3.10.2 Modified Shader Detection

The implementation as discussed above performs the entire shader parsing through pipeline initialization process, throwing away all previous data, whenever a shader program is modified. However, since only one shader program can be modified at once, this is very sub-optimal. This is particularly important since we would like to be able to render a new preview of the pipeline output as quickly as possible.

3.10.3 Parameter Value Propagation

Some of the advantages in interactivity introduced by the optimisations made in [REF msd] will be useless, since parameters specified for the modified shader will

be lost, and will need to be re-entered by the user. However we cannot simply reuse the previous shader parameters since these may have changed.

3.10.4 GLOW Cache

[TODO: this is broken]

3.10.5 FBO Reuse

In the initial implementation, FBOs are simply discarded and new FBOs are allocated after each pipeline change. This is time consuming and could be avoided. However, since FBOs come in different sizes, we cannot naively allocate some 'pool' of available FBOs. This is achieved by simply keeping

3.11 User Interface

3.11.1 Editor

As discussed in [REF], I chose to use the CodeMirror text editor to provide GLSL syntax highlighting to users. While CodeMirror does not provide GLSL highlighting, it does provide a general, 'C-like' highlighting mode, and simple hooks for custom parsers. Initially, I based my syntax highlighting on the 'C-like' mode provided. This was a simple matter of providing the correct keywords. However, this only provides simple highlighting. Given that each shader is parsed anyway, it would seem sensible to modify the parser used for parameter extraction to also provide syntax highlighting information. [TODO: actually do this]

3.11.2 Layout

The initial layout of the UI was straightforward, consisting of a pair of text editors for each shader program, an editor for the pipeline and parameter specification, a sequence of control buttons and a preview box. [TODO picture]. While this layout was spartan and not user friendly, it was sufficient for initial testing.

3.11.3 Improved Layout

The layout described above suffers from a number of usability problems. Firstly, the UI does not fit at all on the screen of the average user. This requires the user to be constantly scrolling to use it. Secondly, the control buttons are unintuitive, and increase the expected length of a user's development cycle. From [TODO

another picture] we can see the improved layout. Shader programs occupy the top left side of the screen, and can be switched between by clicking on the shader's name. The right hand side of the screen is occupied by the parameter specification UI at the bottom, and the preview at the top. The control buttons have been eliminated entirely using the hooks developed in [REF optimisation]

Chapter 4

Evaluation

4.1 Parser Correctness

As discussed under [REF testing], a test harness was developed for testing the correctness of parameters returned by the parser. The tests consist of pairs of shaders and an object containing the correct parameters for that shader. In [REF figure], we see a graph of the percentage of the test shaders correctly parsed vs. git revision.

4.2 Pipeline Correctness

4.3 Speed

Effect of Modified Shader Detection

In [REF optimisation], we discuss the modification of the pipeline initialization process to avoid unnecessary shader re-parsing, compilation, and pipeline generation.

4.3.1 Effect of FBO Reuse

Also in [REF optimisation], we discuss the modification of the pipeline initialization process to reuse FBOs where possible. While the main motivation for this is to avoid allocating textures unnecessarily, this has possible speed benefits. [TODO graph]

4.4 Texture Usage

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

The combination of WebGL and Linux is still a little unstable at times, and as such I would be apprehensive about relying so much on a very recent technology for future projects.

5.3 Further Work

As this project was only partially focused on the development of a user interface, the current UI is fairly basic. Further work could be done to make the UI more useable, and possibly to conduct a user study to test its effectiveness.

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

Appendix A

Project Proposal