Joseph Seaton

Shader Compositor

Part II Computer Science

Fitzwilliam College

April 18, 2012

Proforma

Name:	**Joseph Seaton**
College:	**Fitzwilliam College**
Project Title:	**Shader Compositor**
Word Count:	**About 6000**
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 created, 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 Seaton of 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

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 constraints we shall ignore for now – 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 excerpt 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.UNSIGNED_BYTE,
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
```

1.3 Requirements analysis

Before beginning the project, it was important to establish the general architecture, and establish dependencies between parts of the project in order to determine the order of their development. In [REF] we can see a summary of the main objectives of the project. [TODO expand]

Goal	Priority
Simple interface	High
Parameter detection and presentation	High
Pipeline generation	High
Pipeline rendering	High
Syntax highlighting	Medium
Optimisations	Medium
Annotations	Medium
Additional UI work	Low
Animation	Low

Chapter 2

Preparation

2.1 A short introduction to OpenGL

[TODO] Framebuffer Objects, or FBOs...

2.2 Development environment

WebGL

WebGL is a very recently developed API allowing web-based applications access to OpenGL ES 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. In [REF], we can see that WebGL uses essentially the same API as C does OpenGL ES, except

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 available 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/].

node.js

node.js is a JavaScript platform built 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.

Google Chrome

WebGL naturally requires the use of a web browser. I have found that Google Chrome provided the most reliable WebGL implementation on my development machine, which runs Linux on an ATI HD 6850 graphics card. 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 using Google Chrome. Google Chrome also includes a set of 'developer tools' including

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.

2.3 Software development process

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

I decided to use a combination of a test-driven and evolutionary software development model. This model enables rapid prototyping for exploring new concepts. The use of unit testing enables early, automated detection of regressions.

2.4 Preparatory learning

2.4.1 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 embarking 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.

2.5 Libraries

JavaScript as provided by web browsers is usually quite minimal in terms of provided libraries. Thanks to between-browser variation, there are also subtle (and not so subtle) differences in behaviour between browsers. It was therefore deemed pertinent to use a set of pre-existing, cross-browser libraries as discussed below.

2.5.1 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. In **??** we see the difference in concision with and without jQuery for some simple common tasks.

```
// jQuery

\$("a").clickfunction() {
  ...
})

// JavaScript

[].forEach.call(document.querySelectorAll("a"), function(el) {
  el.addEventListener("click", function() {
    ...
  });
});
```

2.5.2 Testing framework

There are many unit testing frameworks available for JavaScript, of varying levels of complexity. Many of these frameworks are designed with very large projects with extensive tests in mind. For my project I considered these to be overly complicated to work with. Eventually I settled on QUnit, a very simple unit testing framework that is part of jQuery, and it's port to node.js (a fork of the original code).

2.5.3 WebGL toolkit

Given that OpenGL and therefore WebGL itself can be quite awkward to work with, I decided to use an existing library to abstract away some of the boilerplate code irrelevant to my project. 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.

2.5.4 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.

jQuery UI

jQuery UI is a User Interface library built on top of jQuery. It provides a number of common, basic widgets.

2.6 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 regular compressed backups of my current repository using git bundle.

Chapter 3

Implementation

The project consists of the following parts:

3.0.1 Parser

The parser extracts relevant parameters from a given shader, and their associated types.

3.0.2 Pipeline specification

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

3.0.3 Parameter UI generation

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

3.0.4 Pipeline generation

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

3.0.5 Pipeline initialization

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

3.0.6 Rendering

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

3.0.7 User Interface

Connects the above stages together in an event-driven manner, and allows the user to actually input shaders and their connections and parameters.

3.1 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. Since WebGL is based on a very specific version of OpenGL, OpenGL ES 2.0, the parser only needs to support GLSL ES as defined by [REF http://www.khronos.org/registry/gles/specs/2.0/GLSL$_E S_S pecification_1.0.17.pdf].[REFhttp:$
$//www.khronos.org/registry/webgl/specs/latest]Thissimplifiestheconstructionoftheparser, sincewen$

Since GLSL's is (mostly) LALR, it was deemed sensible to use a parser generator rather than expend effort writing a parser by hand. 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 too 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.

Grammar conversion

The grammar for GLSL is provided for GLSL ES by [REF], in BNF form. Since Jison's specification language is roughly based on BNF, the conversion is mostly straightforward as we can see in [REF]. The additional annotations between curly braces are constructing parameter type objects to be passed upwards. Token specification is likewise mostly straightforward, with the exception of struct related tokens as discussed below, and integer / floating point literals which must be represented via regular expressions.

```
//BNF form from specification
type_specifier:
        type_specifier_no_prec
        precision_qualifier type_specifier_no_prec

//Jison version
type_specifier:
        type_specifier_no_prec { \$\$ = {type:\$1}; }
        | precision_qualifier type_specifier_no_prec { \$\$ = {type:\$2,prec:\$1};
        ;
```

Annotations

In addition to parsing raw GLSL, the parser was intended to be able to detect additional annotations to parameters, providing extra information for the parameter UI, for example the range of values taken by a parameter, or whether to provide a colour picking interface. In order to keep compatiability with GLSL, it was decided to provide these annotations via comments, as we can see in [REF]. This is acheived by way of a preprocessor that translates GLSL into annotated GLSL, an example of which we can see in [REF]. Since the syntax of these annotations is quite simple, this can be done using regular expressions. This annotated GLSL can then be passed to the parser, the grammar of which is modified as can be seen in [REF]. [TODO broken]

```
uniform highp int x; //range 0,100
```

```
//GLSL
uniform highp float y; //range 0,100

//Annotated GLSL
uniform highp float y RANGE 0,100;
```

Struct parsing

The GLSL Specification [REF] allows a shader program to define and use new types of structures. These structures behave in essentially the same way as structs in C, subject to certain restrictions. 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,

3.2 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.

In [REF] we see a simple example of a pipeline specification as provided by a user. This corresponds to the shader graph in [REF]. The variable "output" is special, in that it corresponds to the highest node in the tree and therefore the final node in the pipeline. Notice that some parameters are left blank – these can be provided later via the parameter UI. Individual shader instances can be given names for identification within the UI, and the size of the FBO to which the shader is drawn can also be specified (with sensible defaults provided for both). Unfortunately, since JavaScript lacks a way to dynamically find the variable names within a given context, it is not possible to automatically infer shader instance names without using a JavaScript parser, which is beyond the scope of this project (and would not always be able to infer names since shader instances can be anonymous).

```
output = new gaussHShader(``output'');
gaussv = new gaussVShader();
input = new textureShader(``input'',{width:800,height:600});
```

```
output.img = gaussv;
gaussv.img = input;
```

3.3 Parameter UI generation

This stage takes a tree of dummy shaders, and creates 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 user interface 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.

While simple text boxes provide complete control over the parameters, for certain types of parameter more useful widgets can be provided. The most obvious example is for vec3 and vec4 (vectors of length 3 and for) parameters, which are often used to specify colours. For such parameters, I constructed a colour picking widget based on [REF], as can be seen in [TODO].

3.3.1 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].

3.4 Pipeline generation

Pipeline linearisation

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

```
$L \leftarrow$ 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 for**add n to L
    **end if**
**end function**
```

0-262-03293-7.] was used. The algorithm is shown in 3.4. However, the algorithm shown in 3.4 cannot detect when the graph contains a loop. This situation will only occur if the user inputs an invalid shader graph – in such a case, the program should produce an error.

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.

3.5 Rendering

The rendering process itself is kept as simple, 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, which is instead rendered to the preview context. 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]

3.6 Optimisation

3.6.1 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]. Rather than creating GLOW objects – and therefore incurring compilation overhead – after all parameters have been specified, we substitute dummy parameters of the appropriate type, and create the GLOW objects during pipeline initialization. Dummy parameters are obtained by modifying the parameter UI generation stage to generate a dummy value for each parameter as well as the actual UI.

3.6.2 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. Since only one shader program can be modified at once, this is 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.

3.6.3 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 the parameters taken by said shader may have changed. The set of parameters for the modified shader must first be compared to the new required parameters, and dummy parameters generated where appropriate and any extra parameters must be discarded (although we could try to keep these parameters in case they are used in the future [TODO: if I get really bored]).

3.6.4 GLOW cache

[TODO: this is broken]

3.6.5 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 separate pools of already allocated FBOs for each size of FBO that has been used in the past.

3.7 User Interface

3.7.1 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]

3.7.2 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.

3.7.3 Improved Layout

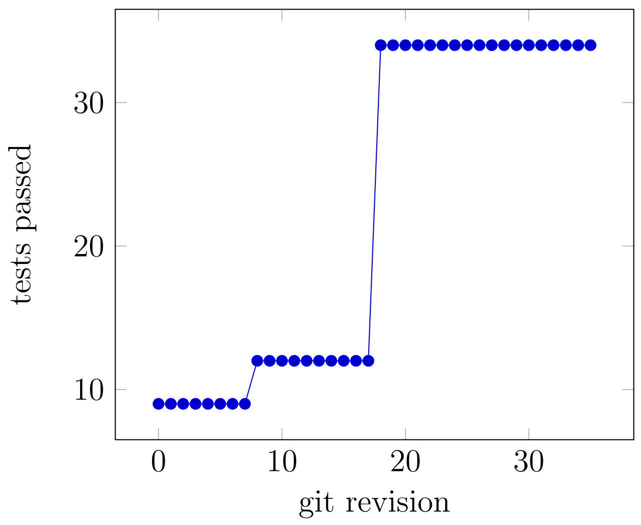
The layout described above suffers from a number of usability problems. Firstly, the UI does not fit at all on the screen of the average user. This requires the user to be constantly scrolling to use it. Secondly, the control buttons are unintuitive, and increase the expected length of a user’s development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

In order to assist with development of the parser, and in particular to assist in regression testing, a test harness was developed for the testing the correctness of parameters returned by 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. Each test is specified by a shader program, and the result that should be returned by the parser – the parameters and their associated information, any struct definitions, and whether the parser should fail to parse it and raise an exception, in the case of an invalid program. Parser output for each test specification is then compared against the actual output for a set of tests. These include correctly throwing/not raising an exception, returning the correct parameter names/a correct subset of the parameter names, all/some of the correct parameter types, the correct struct definitions, and so on. An additional script was written to perform these tests over all git revisions. In [REF] we see a graph showing the percentage of tests correctly completed, from the first revision including a parser.



4.2 Pipeline correctness

[TODO: is this worth it?]

4.3 Performance

Effect of modified shader detection

In [REF optimisation], we discussed 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 sufficiently interactive to be useful. I have made the project source code availabe to others via GitHub, and I hope that it will prove useful. In [REF] we can see that the core critera were completed, and from [REF] we can see that a significant number of the extensions were completed. In addition, a number of optimisations were made not listed in the table below.

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

Criterion	Successful?
Interface to construct shaders, specify connections between shaders	Yes
Detection of parameters of set of shaders, parameters presented to the user somehow	Yes
Sample scene output for single shader	Yes
Demonstration of correct composition (with sample scene output) for various shaders with pipeline specifications	Yes [TODO graph]
Provide basic syntax highlighting of GLSL in interface	Yes

Criterion	Successful?
Reuse of FBOs in shader pipeline	Yes
Automatic unification of identical textures	No[silly]
Splitting shared code off into separate shaders	No[also silly]
Avoiding recompiling shaders unnecessarily	Yes
Presentation of sliders with range detected from range annotations in shaders	Yes[TODO broken]
Some facility for animation	No
Shader and/or scene test suite	No

for future projects.

While I was initially apprehensive about using JavaScript given it's reputation as an ugly language to work with, I was pleasantly surprised by it's novel object model and quite functional underpinnings. Particularly useful was the ability to apply a class constructor to an already existing object [TODO why].

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 usable, 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

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.

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.

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.

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.

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.

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.1 Timetable and Milestones

A.1.1 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.1.2 Weeks 3-5: 12th November 2nd December

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

A.1.3 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.1.4 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.1.5 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.1.6 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.1.7 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.1.8 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.1.9 Weeks 24-26: 7th April 27th April

Finish first draft of dissertation.

A.1.10 Weeks 27-29: 28th April 18th May

Proofreading and submission.