Joseph Seaton js845

Shader Compositor

Part II Computer Science

Fitzwilliam College

May 4, 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

The aim of this project was to produce a novel user interface for easy composition and testing of a certain class of graphical effects known as shaders. The user interface was to be highly interactive and allow easy modification of the code for the effects themselves, the manner in which they are composed and, via graphical means, their associated parameters. Previews of these changes were to be provided as quickly as possible; therefore the project also included a number of optimisations.

Work completed

The core of the project has been completed and works satisfactorily. Shaders can be created and a tree 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 all types of parameter that have been tested, and respects some annotations. A number of optimisations have been applied, including elimination of unnecessary shader recompilation and pipeline

re-generation. The project is also now capable of reusing framebuffer objects 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

1 Introduction **9**
- 1.1 Motivation . . . 9
- 1.2 Brief introduction to shaders . . . 10
  - 1.2.1 A look at some OpenGL calls . . . 10

2 Preparation **11**
- 2.1 Requirements analysis . . . 11
- 2.2 Preparatory learning . . . 12
  - 2.2.1 A short introduction to OpenGL . . . 12
  - 2.2.2 Background reading . . . 12
- 2.3 Development environment . . . 13
  - 2.3.1 Libraries . . . 14
  - 2.3.2 Version control and backup strategy . . . 16
- 2.4 Software development process . . . 16

3 Implementation **19**
- 3.1 Parser . . . 20
- 3.2 Pipeline specification . . . 24
- 3.3 Parameter UI generation . . . 26
  - 3.3.1 Struct handling . . . 27
- 3.4 Pipeline generation . . . 27
- 3.5 Rendering . . . 28
- 3.6 Optimisation . . . 29
  - 3.6.1 GLOW early initialization . . . 29
  - 3.6.2 Modified shader detection . . . 30
  - 3.6.3 Parameter value propagation . . . 30
  - 3.6.4 GLOW cache . . . 30
  - 3.6.5 FBO reuse . . . 31
- 3.7 User Interface . . . 31

3.7.1 Editor 31
3.7.2 Layout 31
3.7.3 Improved Layout 32

4 Evaluation **33**
4.1 Parser correctness 33
4.2 Pipeline correctness 36
4.3 Performance 37
4.3.1 Effect of FBO reuse 38

5 Conclusion **39**
5.1 Main Results 39
5.2 Lessons learned 39
5.3 Further work 41

Bibliography **43**

A Project Proposal **45**

Chapter 1

Introduction

My project provides a novel user interface for easily composing multiple graphical effects known as shaders together, in which individual shader parameters can be easily modified and the results viewed immediately. Furthermore the interface is sufficiently optimised to provide a low round-trip time between making a change and seeing the effect of this change. I have successfully implemented the core of my project, as well as a number of proposed extensions, and a few optimisations not initially proposed.

1.1 Motivation

Working with raw OpenGL is 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. Since 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

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. Vertex shaders operate on individual vertices, transforming the position and also attaching information such as a colour. Fragment shaders roughly speaking generate pixel values. Information can be passed from vertex shaders to fragment shaders using `varying` parameters, the values of which are interpolated from those of the surrounding vertices.

1.2.1 A look at some OpenGL calls

As discussed above, the use of shaders sounds simple. However, as we can see from Listing 1, a short excerpt of the OpenGL calls from an actual WebGL program[15], 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.

Listing 1 Example WebGL calls

```
rawData = new Uint8Array(noisepixels);
texture_noise_1 = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture_noise_1);
gl.pixelStorei(gl.UNPACK_ALIGNMENT, 1);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, sizeX, sizeY, 0, gl.RGBA,
              gl.UNSIGNED_BYTE, rawData);
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 Requirements analysis

Before beginning the project, it was important to establish the general architecture. In 2.1 we can see a summary of the main objectives of the project. [Is this reasonable detail?]

In order to properly plan the development of the project, it was necessary to construct a graph of the dependencies between parts of the project. Priority could then be given to those parts of the project that were necessary for other parts of the core. Figure 2.1 shows the general dependencies between larger parts of the project.

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

Table 2.1: Main goals

2.2 Preparatory learning

2.2.1 A short introduction to OpenGL

OpenGL is a hardware, Operating System and vendor independent graphics API. It has a stateful, client-server architecture, in which the client sends commands – OpenGL API calls – to the server. These commands alter the current state, specifying for example that the following commands will specify a polygon. These commands enter a pipeline, a very simplified version of which is shown in [TODO diagram]. The vertex and fragment shaders discussed in Section **??** are contained within the per-vertex and per-fragment operation blocks respectively. Of particular concern to for this project is the final stage in this pipeline, in which fragments are rasterized and (following some other processing e.g. z-buffering) written to a framebuffer. A Framebuffer Objects or FBOs contains several buffers including a colour buffer, which may accessed as a texture by other stages.

2.2.2 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' [11]. I also read through the GLSL ES specification provided by the Khronos Group [13], 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 *Eloquent Javascript* [9] and *Higher Order Programming [in JavaScript]* [14], 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.

2.3 Development environment

WebGL

WebGL is a very recently developed API allowing web-based applications access to OpenGL ES contexts, via JavaScript [7]. 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 [7], we can see that WebGL uses essentially the same API as OpenGL ES for C.

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 [10].

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

WebGL inspector is a Google Chrome extension intended to provide inline information on current WebGL contexts, giving easy access to state information including currently allocated textures. While this tool would be very useful for debugging, I found the tool to be insufficiently stable on my Operating System (Linux) to use reliably. This meant that most inspection of OpenGL state had to be performed manually.

2.3.1 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.

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 Listing 2.3.1 we see the difference in concision with and without jQuery for some simple common tasks.

Listing 2 Comparison of simple JavaScript tasks with and without jQuery

```
// jQuery
$("a").clickfunction() {
  ...
})

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

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).

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 [5] 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 [2], a toolkit specifically designed to make working with shaders simple, but otherwise providing little abstraction.

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 [1], 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 including a simple tabbing interface. Using a pre-existing library is especially useful for User Interface work in JavaScript, since widgets will have been tested against multiple browsers, a time consuming and tricky process.

2.3.2 Version control and backup strategy

All project files, including the dissertation, were placed under git version control. Git is a distributed version control system, thus it was possible to maintain multiple complete copies of the repository in different locations. 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.

2.4 Software development process

WebGL is still quite a new technology, and as such the libraries surrounding it are not yet mature. Given that I was also not overly familiar with OpenGL at the beginning of the project, I deemed it sensible to apply an iterative software development model, to allow the requirements to be updated after prototyping.

I decided to use an evolutionary software development model. This model employs rapid prototyping, enabling fast experimentation and evaluation of the feasibility of a given route. The model also incorporates feedback from development into the specification.

Test-driven development was also employed for some parts of the project, most prominently the parser. The use of unit testing enables early, automated detection of regressions, making it easy to spot bugs that might otherwise go unnoticed. Git pre-commit hooks were used to enforce running tests regularly. [TODO pic that isn't obviously nicked]

Chapter 3

Implementation

[TODO: 'shaders /shader instances' as discussed actually correspond to vertex/fragment shader pairs. How to sensibly make this clear? Comment in intro? Maybe an introduction here?]

The project essentially consists of two parts. The main flow of control within the program can be viewed as a pipeline, with each stage depending on results from the previous stage, and passing results to the next stage. The user interface provides new input, and either runs through the entire pipeline in the simple case, or updates individual stages in the more complex case. In either case, changes to earlier stages necessitates updating all later stages. The project can be further divided into the following parts:

Parser

The parser extracts relevant parameters from a given shader, and their associated types, precisions and arity. Using annotations extra information for the User Interface is also extracted.

Pipeline specification

The user-specified shader graph is extracted and converted into a usable format, with dummy objects being extended with actual shader instances.

Parameter UI generation

An appropriate user interface for specifying parameters for each shader instance in the shader tree is generated based on the pipeline specification and the shader parameters extracted by the parser. Callbacks are generated for changes to widgets.

Pipeline generation

A list of shaders is generated from the shader tree such that the shaders can be rendered in sequence. Actual GLOW shader objects are created. FBOs are assigned to shaders as necessary.

Rendering

Render the actual pipeline, resulting in a preview image in the user interface.

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. Provides feedback for errors.

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. For each parameter, the parser returns an object specifying:

- A parameter qualifier: we are primarily concerned with `uniform` parameters since these are exposed to the user
- The type: this may be a fundamental type like `int` or a structure, in which case the type is an object like a parameter object, barring the parameter qualifier

- The precision: unused, but could provide error information
- Arity: if the parameter is an array, the size of the array (this is required to be constant)
- Range: range annotation if present
- Colour: presence of a colour annotation

All of these other than the type may be left unspecified. 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 1.0.17 [13][7]. This simplifies the construction of the parser, as we need not worry about different GLSL versions.

Since GLSL's grammar 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 [4], 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 [3], a JavaScript port of GNU Bison. Bison is a venerable parser generator which is part of the GNU project, but is only capable of generating C, C++ and Java parsers. Jison was able to parse even quite complicated input in a reasonable amount of time, as is discussed in my evaluation. As an alternative to using Jison, which is relatively untested compared to Bison, I could have used server-side parsing. However, this would have imposed the need for websites using my project to compile code to run on their servers, and configure their webservers appropriately. Conversly using Jison, no server-side processing is required at all, much reducing the barrier to using my project.

Grammar conversion

The grammar for GLSL is provided for GLSL ES in the specification [13], in BNF form. Since Jison's specification language is roughly based on BNF, the

conversion is mostly straightforward as we can see in Listing 3.1. 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. Care must also be taken with the ordering of token definitions, in particular the IDENTIFIER token is capable of capturing most keywords and must be placed after them.

Listing 3 GLSL grammar rules

```
//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};
        }
        ;
```

Preprocessor

GLSL contains a preprocessor language much like that used in C. It is very restricted and mostly only allows a few simple substitutions and if/else statements. It is often used to make GLSL version-based decisions, but since WebGL currently only supports one version it is not often used in WebGL programs. Implementation would require a simple initial pass over the shader code before forwarding it to the main parser. However this was dropped for time reasons [TODO: if there's time this is easy].

Annotations

In addition to parsing raw GLSL, the parser was intended to be able to detect additional annotations to parameters. These annotations would provide extra information to enrich the usability of the parameter UI, for example by specifying the expected range of values taken by a parameter, or in the case of vectors, 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 Listing 4. According to the GLSL specification [13], comments are not to be handled by the parser, but rather be stripped out by a preprocessor. Therefore annotations are acheived by way of my own preprocessor that translates GLSL into annotated GLSL by replacing of a certain form, an example of which we can see in Listing 5. Other types of comment are simply removed, as required by the specification. 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 to accept these annotations. An example of the rule changes can be seen in [REF TODO broken].

Listing 4 Simple annotated GLSL parameter

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

Listing 5 Transformation to annotated GLSL

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

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

Struct parsing

The GLSL Specification [13] 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 concerning self-references. Structure names and field names are subject to the same restrictions as normal identifiers. Since this means that a token may be considered an identifier or a type or a field identifier depending on its usage, this makes parsing GLSL as parsed by e.g. Google Chrome technically context-sensitive. However, the parser used,

Jison, only supports LALR grammars. General context-free parsers are notoriously complicated, and tend to have high time complexity. Their use was therefore immediately ruled out as unsuitable.

The grammar as provided in the specification introduces extra tokens for structure and field names, skipping over the problem of context sensitivity entirely, but suggesting that the lexer should be able to decide. Therefore by way of an initial, straightforward implementation of struct parsing, I chose to make the parser simply append structure/field names to the appropriate part of the matching logic of the lexer whenever a new structure definition was encountered. While this will fail to parse shaders that use e.g. some field name as the name of a variable, or more likely, some field name as a struct name, it was deemed to be sufficient for this project.

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. This is particularly useful given the slight variations in WebGL implementation between web browsers. For example, the pipeline could choose to use a slightly faster version of a shader for Firefox, given the worse performance on that browser (in my personal experience), or even display a notice to Internet Explorer users explaining that WebGL is not supported on that browser.

While I could have required pipeline specifications to produce an actual shader graph in its entirety, 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 the interface as presented to the user, most of the internal complexity of a shader instance is hidden. As far as the user is concerned, an instance has parameters, a name and a size, but nothing else. Using the output of another shader instance uses the same interface as any other parameter. The actual

behaviour, in which an FBO is written to by one instance and read from by another, is left implicit.

To construct this environment, we evaluate the pipeline specification text (as JavaScript) within a function containing the appropriate function definitions. These definitions include shorthand functions like `sampler2D()` for specifying parameters (in this case, a texture), and shorthand objects with convinient names like `gaussHShader` in Listing 6 for creating shader dummy objects. These objects merely contain the shader instance name, type, size and parameters. The resulting tree (actually, a Directed Acyclic Graph or DAG) of dummy objects is then be walked, creating an appropriate tree of actual shader instances. In the initial implementation, this includes separating parameters into an object as needed by GLOW, attaching a reference to the appropriate shader and adding name and size information. Later, this was modified to also include initialisation of GLOW objects. This is discussed in Section 3.6.

In Listing 6 we see a simple example of a pipeline specification as provided by a user. This corresponds to the shader graph in Figure 3.1. The variable "output" is special, in that it corresponds to the highest node in the 'tree' and therefore the final node in the pipeline – the shader to be rendered to the screen. 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 on initialisation, and the size of the FBO to which the shader is drawn can also be specified. When these are not specified, an identifier based on the shader name and the size of the canvas are used respectively. 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).

Listing 6 Example pipeline specification

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

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

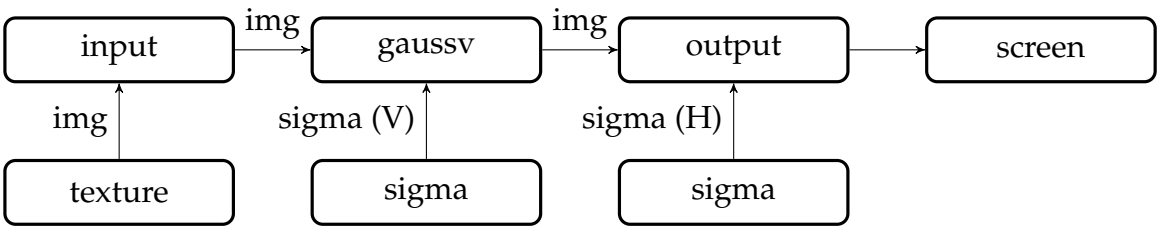


Figure 3.1: Shader graph

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 `uniform` and is not already specified – it produces an appropriate set of input elements. For each parameter a label based upon either the provided name or a hopefully informative generated name is generated, along with a type-specific label and widget.

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, correctly invalidate the relevant GLOW cache and request the preview be re-rendered. This is achieved cleanly thanks to the presence of closures in JavaScript: the UI generating function has access to the relevant GLOW object for the current parameters, and as such event handles attached within it also have access to the correct object, without the need for clumsy navigation of the shader graph.

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 the web colour picker by Stefan Petre [12], as can be seen in [TODO PIC]. Another addition which would make use of the range parameter annotations (see Section 3.1) would be sliders for numeric parameters. Texture parameters assume sensible defaults rather than requiring the user to specify the verbose information required by GLOW.

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, the parser will not reject such a self-referential structure definition, which would cause infinite recursion. This is avoided using a counter set to a sufficiently large value that no valid GLSL program could reach it. This is possible since GLSL programs are required to compile to a sufficiently small amount of instructions, due to the limitations of stream processors.

3.4 Pipeline generation

Pipeline linearisation

At this stage in the process, we have essentially a Directed, Acyclic Graph (DAG) of shader instances, with connections representing dependencies between instances – that is, instances that write to textures that will be read by other instances. In order to actually render the DAG, we must linearise it such that the node labeled "output" is last. This is the primary task of the pipeline generation stage, namely generating a list containing the nodes of the shader DAG. Since JavaScript uses pass-by-reference for objects, we can retain the original DAG at no extra cost. We achieve this linearisation using a topological sort. Initially the algorithm decribed by Cormen et al [8] was used. The algorithm is shown in Listing 3.4. Note that we only have one node in S, the output node. Additionally, all nodes must be marked as not visited before running this algorithm, as the linearisation procedure may be run multiple times. However, the algorithm shown in Listing 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.

```
L ← Empty list that will contain the sorted nodes
S ← 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
```

Pipeline initialisation

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. All parameter values would be retrieved from the UI at this point. This required the user to manually request the pipeline be updated before rendering.

Recreating GLOW objects and reading in all parameter values after a single parameter change is very inefficient, especially since this process would run after each individual parameter change. Therefore this stage was later modified such that changing parameters would not require a complete update – GLOW objects are created before the Parameter UI Generation stage, with dummy parameter values, and pipeline initialisation is only run after a change to the pipeline (as discussed in Section 3.6.

3.5 Rendering

The rendering process itself is kept 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 for each shader instance at the start of each call, we can avoid this as discussed under Section 3.6.4.

3.6 Optimisation

As stated in the introduction, one of the main aims of this project is to make the User Interface as interactive as possible. To be more specific, the delay between making some change and seeing the effect of this change should be as short as possible. Since changes on this timescale will usually be quite small, for example changing a single parameter, or modifying a single shader, by reusing existing state where possible this delay can be significantly reduced.

3.6.1 GLOW early initialization

As discussed in Section 3.3 and Section 3.4, in the initial implementation all stages following Parameter UI Generation must be called, at the users request, following each change to a parameter. This includes a step in which every parameter in the UI is updated, initialisation of GLOW shader objects, and pipeline linearisation. 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 Section 3.4. 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 shader tree creation (that is, during the pipeline specification stage). 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. Parameter changes now generate events which update the relevant GLOW object's parameters, and request a new render. The entire pipeline initialisation stage can now be skipped. Updates also invalidate the relevant GLOW cache as discussed below.

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. The most obvious optimisation here is to only update shader instances in the pipeline that use the modified shader. This is achieved by tagging shader instances with the name of their shader. Shader update calls then only need to attach new GLOW shaders to the appropriate shaders, using their existing parameters. Note that this is only applicable if the required shader parameters have not been added to or changed, since all parameters must be specified. In this case the old behaviour can be falled back on.

3.6.3 Parameter value propagation

Some of the advantages in interactivity introduced by the optimisations made in Section 3.6.2 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 (these extra parameters are ignored, and 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

For efficiency reasons, GLOW itself tries to cache as much as possible in GPU-accessible memory. In particular, parameter values are cached and updates to parameters must respect this and invalidate the cache before the next render call. In the initial implementation, the GLOW cache was cleared at the start of each render call. This is inefficient, and was soon modified such that the GLOW cache was only cleared after parameter changes and pipeline changes.

However, this is still not particularly efficient, as these gains are mostly only noticable during animation. [TODO: can do better?]

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. Shader updates using the event-based system described above can reuse the existing FBO for each instance, this is an additional benefit of the optimisations discussed.

3.7 User Interface

3.7.1 Editor

As discussed in 2.3.1, 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. A particularly common feature provided by IDEs is highlighting the line on which a syntax error occurs. [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, a text editor for the pipeline and parameter specification, the parameter UI, 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, and allowed for easy debugging of separate stages.

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 an average-sized screen. This requires the user to be constantly scrolling to use it, adding a delay between the user making a change, and seeing its effect. Secondly, the control buttons are unintuitive, and the need for the user to use them whenever they wish to see the effect of a change further increases the expected length of a user's development cycle. A new layout was designed, with inspiration taken from the typical construction of IDEs, where tiling is employed to maximize screen usage. 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 Section 3.6. Compared to the old layout, this provides:

- Minimal scrolling – shaders/pipeline specification/parameters can be changed without having to scroll to see the result
- Improved interactivity – updated render provided as soon as possible, rather than on request
- Improved usability – no confusing buttons to press requiring knowledge of internal program workings
- Error feedback – introduction of an error popup means users are informed of errors, but screen space is not wasted otherwise

Chapter 4

Evaluation

Evaluation of this project chiefly consists of two parts, correctness and performance. Correctness evaluation is intended to demonstrate correct functioning of the core project. This consists of two parts: parser testing, to ensure parameters are correctly provided to the user interface, and output testing, to verify that the output of shader pipelines is visually as expected. Performance evaluation provides an analysis of the effect in terms of both performance and resource usage of some of the optimisations. In particular, the effect of avoiding unnecessary shader recompilation and later steps on performance is evaluated.

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 Listing 7 we see a small sample of the complete test results for an individual version, verbatim from actual command line output. The second column lists individual tests – here, we see some very basic tests verifying that for example `uniform int` is parsed correctly. The third column lists the sub-tests for each test: these are identical for each test, with the exception of tests that are intended to fail to parse. The right column shows whether the sub-test is passed. We can see that for the `varying lowp vec4 `, the parameter types are incorrect for this git revision. Listing 8 shows a sample of the test summary for some of the later git revisions. Each line represents a different git revision. Note that there are two sub-tests consistently failed: these correspond to tests testing correct struct parsing for structs sharing names with variables or fields, which cannot be parsed with the current parser implementation, as discussed in Section 3.1.

Listing 7 Sample of verbose test results

```
Module                                  Test                            As
glsl                                    uint                            Su
                                                                        On
                                                                        Co
                                                                        Co
                                                                        Co
                                                                        Co
                                        ufloat                          Su
                                                                        On
                                                                        Co
                                                                        Co
                                                                        Co
                                                                        Co
                                        vlvec4                          Su
                                                                        On
                                                                        Co
                                                                        Co
                                                                        Co
                                                                        Co
```

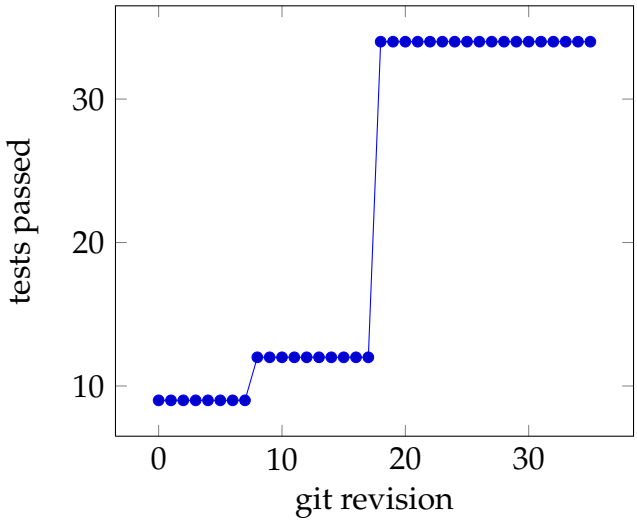
Listing 8 Sample of test summary

```
Summary:

 File                                                Failed    Passed    Total     Runt

 /home/joseph/ws/clone/shomp/parser/glsl.js          2         34        36        177

 /home/joseph/ws/clone/shomp/parser/glsl.js          2         34        36        179

 /home/joseph/ws/clone/shomp/parser/glsl.js          2         34        36        179

 /home/joseph/ws/clone/shomp/parser/glsl.js          2         34        36        173

 /home/joseph/ws/clone/shomp/parser/glsl.js          2         34        36        175

 /home/joseph/ws/clone/shomp/parser/glsl.js          2         34        36        178
```

In Section 4.1 we see a graph showing the percentage of tests correctly completed, from the first revision including a parser. The notable jumps in tests passed correspond to implementation of the currently used parameter format, scoping corrections (i.e. discarding parameters hidden within functions) and structure support. Not all tests are passed at the final version – these tests are those that contain structure definitions and variables with the same name, the parsing of which is not possible with my current parser, as discussed in Section 3.1. Also tests that use preprocessor commands will fail since a preprocessor was not implemented.[TODO more tests]

4.2 Pipeline correctness

[TODO: don't really have this, getting output is a real pain. Possibly construct a set of demonstrations that clearly work? Good excuse to make pretty pictures.] In order to demonstrate visually correct shader pipeline output, a small number of example pipelines were constructed. This was achieved by taking a sample of simple WebGL example programs found online, and modifying them to work within my project. Output from the modified versions could



then be compared with the originals. Ideal example programs would use a few different shaders connected together to produce a single visual effect. Below we can see samples of the output of the original programs alongside my reimplementations.

4.3 Performance

Effect of modified shader detection

In Section 3.6, we discussed the modification of the pipeline initialization process to avoid unnecessary shader re-parsing, compilation, and pipeline generation after modification of a shader. In order to compare the performance of the updating procedure before and after these optimisations were introduced, a custom test harness was developed. Since the existing test harnesses for JavaScript that provide performance testing are mostly quite heavyweight, being intended for large-scale website testing, I implemented this from scratch.

Each test specification consists of a set of shaders, and a shader tree specification in which all necessary parameters are specified. Once a shader pipeline has been generated for a given specification, a shader update event is generated for a particular shader, and its completion timed. This is repeated 10 times to obtain an average and range. Similarly the completion of a complete re-generation of the shader pipeline is timed 10 times. This is repeated for each

shader in the test specification. From [TODO graph] we can see the result of these tests for a particular test specification.

4.3.1 Effect of FBO reuse

Also in Section 3.6, we discussed the modification of the pipeline initialization process to reuse FBOs where possible. While the main motivation for this is to avoid allocating new textures unnecessarily, this has possible speed benefits. [TODO graph]

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 Table 5.1 we can see that the core critera were completed, and from Table 5.1 we can see that a significant number of the extensions were completed. Of the extensions not implemented, two of them were deemed unsuitable. Texture unification as originally envisaged is supported trivially by the pipeline specification stage, and code separation would be sufficiently complex to be outside the scope of this project, and is unlikely to be worth the extra effort. 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 less inclined to rely so much on a very recent technology for future projects.

While I was initially apprehensive about using JavaScript given it's reputation as an ugly language to work with and my experience of Internet Explorer 6-era JavaScript, I was pleasantly surprised by it's novel object model and quite functional underpinnings. Particularly useful was the ability to apply a class

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	No [SEE EVAL]
Provide basic syntax highlighting of GLSL in interface	Yes

Table 5.1: Core Criteria

Criterion	Successful?
Reuse of FBOs in shader pipeline	Yes
Automatic unification of identical textures	No [1]
Splitting shared code off into separate shaders	No [2]
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

Table 5.2: Extensions

constructor to an already existing object, and the ability to treat objects much like associative arrays. Combined with a good set of libraries like jQuery, I found modern JavaScript about as pleasant to work with as languages like Python or Ruby.

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. More sophisticated widgets and better error feedback would be a good start. The extensions listed in Table 5.1 were dropped for time reasons, but could also be incorporated with not too much additional effort. A number of techniques are used for dealing with context sensitivity in C without using a general context-sensitive parsing algorithm, which could potentially be applied to parsing GLSL with structs correctly.

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 [6]. 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] Codemirror. http://codemirror.net/.

[2] I am glow. http://i-am-glow.com/.

[3] Jison. http://zaach.github.com/jison/.

[4] Js/cc parser generator project homepage. http://jscc.jmksf.com/.

[5] three.js. https://github.com/mrdoob/three.js/.

[6] Webgl and html5 challenges for the future. http://codeflow.org/entries/2011/sep/11/webgl-and-html5-challenges-for-the-future/.

[7] Webgl specification. http://www.khronos.org/registry/webgl/specs/latest.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 22.4: Topological sort, pages 549–552. The MIT Press, 2 edition, 2001.

[9] M. Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press Series. No Starch Press, 2011.

[10] David Miller. Setting up emacs as a javascript editing environment for fun and profit. http://blog.deadpansincerity.com/2011/05/setting-up-emacs-as-a-javascript-editing-environment-for-fun-and-profit/, May 2011.

[11] Opengl, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[12] Stefan Petre. Colorpicker – jquery plugin. http://www.eyecon.ro/colorpicker/.

[13] Robert J. Simpson. *The OpenGL ES Shading Language*. 2009.

[14] Sjoerd Visscher. Higher order programming. `http://w3future.com/html/stories/hop.xml`, October 2005.

[15] Felix Woitzel. Webgl shader lab reactor-diffusion mix. `http://cake23.de/diffusion-mix.html`.

Appendix A

Project Proposal