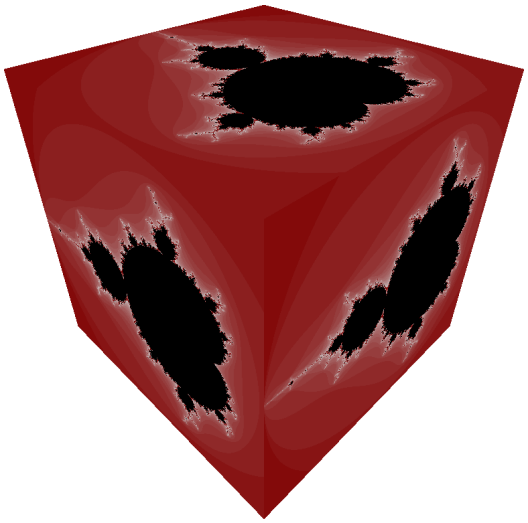
Ben Challenor

bdc25@cam.ac.uk

Funslang: A lightweight shading language for graphics processors



Computer Science Tripos Part II

St John's College

2008

Proforma

Name:	**Ben Challenor**
College:	**St John's College**
Project Title:	**A lightweight shading language for graphics processors**
Examination:	**CST Part II, 2008**
Word Count:	**10426 (TEXcount)**
Project Originator:	Ben Challenor
Project Supervisor:	Christian Steinruecken

Original Aims of the Project

Noting that existing shading languages have many shortcomings, the aims of this project were the design of a new shading language and the implementation of a compiler. The language had to allow typical shader use cases: in particular, standard vertex transformations and texture application. The compiler had to generate code that could run on a real graphics processor.

Work Completed

All project goals have been completed. The result, 'Funslang', is a pure functional shading language. Its novel type system allows dimensionally-safe, polymorphic linear algebra. Despite being functional, it can be compiled to an architecture with no concept of function calls. As an extension, the compiler has been integrated with OpenGL to allow applications to use the new language at run time. Two graphical applications have been created to demonstrate the success of the project.

Special Difficulties

None.

Declaration of Originality

I, Ben Challenor of St John's 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

Acknowledgments

This project would not have been possible without Eben Upton, who introduced me to both compilers and shading languages.

For agreeing to supervise this project, I would like to thank Christian Steinruecken: your crazy ideas were an inspiration.

Thanks are also due to Alex Davies, whose discussion and encouragement was invaluable.

Contents

Chapter 1

Introduction

In recent years, graphics processors have become more flexible. Instead of fixed-function hardware, graphical effects are produced in software on massively parallel arithmetic units.

Until now, that software has traditionally been written in specialized assembly languages [Brown, 2002–2004, Lipchak, 2002–2003] or 'high-level' languages [Kessenich, 2006, Mark et al., 2003, Peeper and Mitchell, 2003] that closely resemble C. However, I felt that many aspects of C were not appropriate in this application.

I decided to design and implement a new language, to be as far as possible free of legacy features. To make it easy to deploy on new, mobile architectures, it had to be lightweight, with a simple semantics.

The result is *Funslang*: a functional shading language. With this language I have achieved all the goals of my Proposal, as well as several extensions. I have implemented a compiler and integrated it successfully with OpenGL, allowing my language to be used at run time.

While work has been done on functional languages for stream processors [Frankau and Mycroft, 2003], this is one of the first to be optimized for graphics. Graphical computation has an exciting future, and I hope that this dissertation might contribute a few ideas.

Chapter 2

Preparation

Shaders are introduced, and the potential benefits of a programmable graphics pipeline are explained. Existing shading languages are analysed. Appropriate tools are selected. Design considerations are summarized such that implementation may begin.

2.1 Introduction to Shaders

Modern real time graphics libraries, such as OpenGL[1] and Direct3D[2], currently use the *rasterization* model of rendering.

The scene to be rendered is represented by a set of polygons, with each defined by its vertices. Each vertex has at least a three-dimensional position vector, but it usually has other attributes, such as a colour or a normal vector.

The vertices are transformed into two-dimensional screen space by applying a series of matrices. The two-dimensional scene can then be rasterized, converting the abstract shapes into coloured pixels.

Graphical effects, such as lighting, texturing and fog, used to be applied by fixed-function hardware at certain points in the pipeline. However, the range of effects was limited, and general purpose hardware was introduced to give the user more control.

[1] See `http://www.opengl.org`.
[2] See `http://msdn.microsoft.com/directX`.

The pipeline was opened up to users at two programmable points: the *vertex shader* and the *fragment shader* (also known as the *pixel shader*). Given a particular polygon, the vertex shader is run once per vertex, and the fragment shader is run once per pixel. Each shader run is independent, and so the graphics hardware will typically schedule many runs in parallel.

Some recent architectures provide another programmable point, the *geometry shader*. This is located between the vertex and fragment shaders in the pipeline, allowing the programmatical generation or deletion of polygons. Geometry shaders are not currently supported by all graphics libraries, and so I will not deal with them further. However, once they are standardized, Funslang could quite easily support them.

2.1.1 Shader Pipeline

The shader pipeline is outlined in Figure 2.1. Briefly:

- the user specifies a polygon, associating with each vertex any desired *attributes* (e.g. position in 3D, normal vector)
- the graphics library invokes the user's vertex shader once per vertex, passing its attributes
- the vertex shader transforms the vertex and prepares any *varyings* (e.g. texture coordinates) for the fragment shader
- the graphics hardware uses the transformed vertices to decide which pixels will need colouring
- it then interpolates the varyings across the polygon, with one set for each pixel
- the graphics library invokes the user's fragment shader once per pixel, passing its interpolated varyings
- the fragment shader decides the colour of the pixel

2.1.2 Example Applications

The trend in existing graphics libraries is towards removing fixed functionality, allowing the user to recreate such effects as they desire. This has the

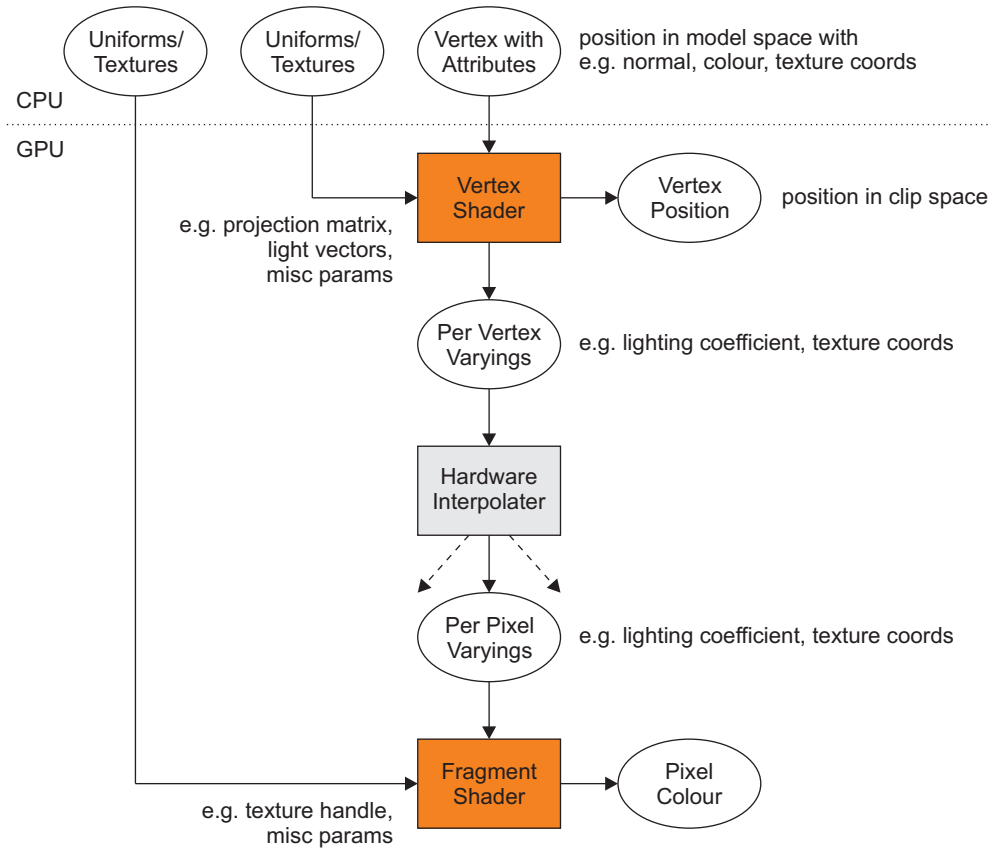


Figure 2.1: The graphical shader pipeline.

advantage of greatly reducing library complexity, while giving the user more flexibility.

Many useful graphical effects are possible under this model. To name just a few:

- lighting
- texture application, including multitexturing and blending
- fog, to simulate atmospheric effects on distant objects
- bump mapping, to simulate non-smooth surfaces
- iterative noise generation, e.g. for wood or clouds
- reflection mapping and refraction, e.g. for realistic water

It is difficult to list the applications of the vertex and fragment shaders separately: in reality, they are often tightly coupled, to the point of coming in pairs. Most production shaders will combine several or even all of the above effects.

2.1.3 Using Shaders

First, note that the shading language does not replace the language in which the user's application is written. The majority of the control logic, such as setting up the graphics library, will be executed on the CPU.

We must also distinguish 'compile time' for the application from 'compile time' for the shaders. Graphics architectures are not standardized, and an application may be deployed on many different devices. Thus, in both of the major graphics libraries, shaders are not compiled until application run time.

Given a pair of shaders, the graphics library needs to know three sets of data, shown on the CPU side of Figure 2.1:

- constants (including textures) accessible to the vertex shader
- constants (including textures) accessible to the fragment shader
- polygon vertices and their associated attributes

In OpenGL these are called *vertex uniforms*, *fragment uniforms*, and *vertex attributes* respectively.

The graphics library passes the data to the graphics hardware, and the polygon is rendered.

2.2 Language Research

Before designing Funslang, I first considered the existing shading languages. Which features were necessary, and which could be improved on?

This section will focus mainly on the OpenGL Shading Language (GLSL) [Kessenich, 2006], but most of the points are also applicable to the other 'high-level' shading languages, Cg [Mark et al., 2003] and HLSL [Peeper and Mitchell, 2003].

2.2.1 Necessary Features

To help the programmer new to Funslang, I wanted to ensure that it was not *unnecessarily* different to the existing languages.

While I had no qualms about having the *language* of Funslang depart radically from the existing ones – and as a functional language, I believe it does – I tried to keep other features as close to GLSL as possible.

For example, the libraries share many function names, and the terminology I use in Funslang is similar to GLSL. The C application programming interface (API) of Funslang replaces very few OpenGL functions.

2.2.2 Undesirable Features

However, many features of GLSL are legacies inherited from C, adding to the complexity of the semantics and the compiler.

Separation of Syntax and Semantics

The main function in a GLSL shader is declared to be of type $void \rightarrow void$, and output is achieved by writing to magic variables.

For example, here is a fragment shader that outputs constant red:

```
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

This approach certainly works, but its meaning is not clear. It also has unfortunate consequences [Kessenich, 2006, p.48]:

> *Reading from these variables before writing to them results in an undefined value ... If subsequent fixed functionality consumes fragment color and an execution of the fragment shader executable does not write a value to* `gl_FragColor` *then the fragment color consumed is undefined.*

Assuming that it is undesirable to have shaders with undefined behaviour, enforcing such semantics would require ad-hoc checks.

GLSL in fact requires many of these checks – a list is given in Simpson and Kessenich [2007, pp.97–99]. For example, in non-`void` functions, all control paths must end with `return` statements. In `void` functions, `return` statements are optional, but they must not return values. In both kinds of function, the compiler should detect 'unreachable code' below a `return`.

Ideally we would prefer to push the checks into the syntax, or a type system. For example, in ML and Haskell, the syntax ensures that all control paths return a value, and the `void` value ('unit') does not require special treatment.

Eliminating undefined behaviour would also be a step towards a more formal semantics.

Impurity

Most shaders require little or no control flow; for example, loops are statically sized, and conditional branches are rare. Under this assumption, the hardware on which they are executed can be greatly simplified: many shaders can be executed in lockstep by a single execution unit.

However, the syntax of C (and thus GLSL) is biased towards control flow, which the shader compiler must undo. One such compiler phase is *if-*

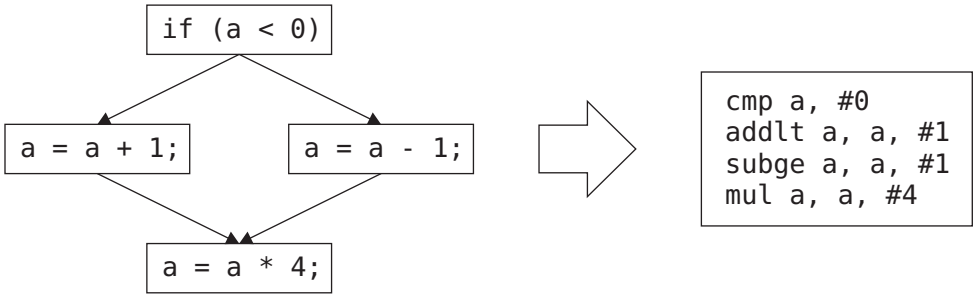


Figure 2.2: An example of if-conversion.

conversion, in which conditional branches are replaced by predicated instructions. These always execute, but only write to the register file if their predicate is true. An example is given in Figure 2.2.

Such control flow elimination phases are much more complex if the language is *impure*, with mutable variables. Global mutable variables are even worse – consider a function call inside an if-statement.

But shading cannot cause external side effects, and a shader can always be described as a pure function on numerical data. While pure general purpose languages are rare, I would argue that purity is quite appropriate for a shading language.

Purity has other benefits for compilation, and for the interested reader, I recommend *SSA is Functional Programming* [Appel, 1998].

Complex Control Flow Mechanisms

Many of the architectures to which GLSL will be compiled require control flow to be eliminated. However, GLSL provides the user with complex control flow mechanisms such as `break` and `continue`, which are non-trivial to linearize. The static analysis of even 'for' and 'while' loops can be difficult.

Unfortunately, these features are rarely used by shaders, and so the effort required to support them is disproportionate. Most loops in shaders are used to apply a function to elements of a vector or array. Here, a functional 'fold'[3] or 'map' could capture the essence of the operation, while being much easier to analyse.

[3] `http://foldl.com` and `http://foldr.com` are quite entertaining.

Restrictive, Hard Coded Types

In GLSL, vector types are dimensionally restricted, and the type system is not compositional, so vector types are entirely unrelated to the type of their elements. For example, the floating point vector types are `vec2`, `vec3`, and `vec4`, and unrelated to `float`. Boolean vectors are different again, with `bvec2`, `bvec3`, and `bvec4` separate from `bool`.

GLSL also lacks one-dimensional vectors, which would be good if only for consistency. In GLSL, a two-dimensional texture requires a two-dimensional vector coordinate (`vec2`), and a three-dimensional texture requires a three-dimensional vector coordinate (`vec3`). However, a one-dimensional texture is accessed with a scalar co-ordinate (`float`).

The vector dimension restriction dates back to hardware which had four-component vector arithmetic units. However, vectors are only a syntactic convenience for a modern shader compiler, which will perform all vector operations with scalar instructions [NVIDIA Corp., 2006, p.27]:

> *NVIDIA engineers have estimated as much as 2× performance improvement can be realized from a scalar architecture that uses 128 scalar processors versus one that uses 32 four-component vector processors, based on architectural efficiency of the scalar design.*

Thus there is no reason for a language to artificially restrict the dimension of a vector. Indeed a more flexible approach, allowing the user to form arbitrarily sized vectors from the element type, may make the compiler implementation simpler.

Ad-hoc Polymorphism

Overloading, or ad-hoc polymorphism, is where multiple operations with *different implementations* are known by the same name. Most of the library functions in GLSL are overloaded. For instance, the dot-product function `dot` can be applied to pairs of `vec2`, `vec3`, or `vec4`.

But different implementations can be a problem for both clarity and maintainability.

If you are actually trying to apply the same algorithm to the different types, then it might be better to change the type system so that it can be achieved

with a single implementation. This is known as parametric polymorphism, and is seen in functional (e.g. ML, Haskell) as well as imperative languages (e.g. Java's 'generics').

If on the other hand, the algorithm differs, then for clarity you might consider giving the operations different names.

Most of the ad-hoc polymorphism in GLSL is of the first type: a single algorithm, such as the dot product, can be applied to vectors of different sizes. Parametric polymorphism could reduce the size of the GLSL library by a factor of four.

Polymorphism of the second type can be confusing. For example, the GLSL definition of the '`*`' operator takes nearly a page [Kessenich, 2006, p.37], explaining the different semantics of its application: scalar-scalar, scalar-vector, scalar-matrix, vector-scalar, matrix-scalar, vector-vector, matrix-matrix, matrix-vector or vector-matrix.

If I write `m1 * m2 * v`, with `m1` and `m2` matrices and `v` a vector, do I mean `(m1 * m2) * v` or `m1 * (m2 * v)`? Though mathematically they give the same result, the former takes $O(n^3)$ operations and the latter $O(n^2)$. Unfortunately, as `*` is left-associative in GLSL, we get the former.

Programming-in-the-Large Features

'Programming-in-the-large'[4] features are those that help the management of large software projects.

However, shader programming is programming-in-the-small: shaders are typically a few hundred lines at most, and in a single file. Shading languages do not require the same features as general purpose languages.

The GLSL type system is extensible: through the `struct` mechanism familiar to C programmers, new aggregate types can be defined. However, this is probably unnecessary. A type system that allowed the construction of new anonymous types, such as the tuples of ML, Haskell and Python, would give similar benefits, but be simpler to specify and implement.

GLSL also requires a preprocessor. However, the string manipulating phases of a compiler are often the most expensive, and the preprocessor seems an unnecessary burden, especially on mobile devices. At this scale, preprocessor macros can be replaced by language-level features such as constant definition

[4] See [DeRemer and Kron, 1975] if you are interested in the origins of this term.

and type polymorphism.

Type Annotations

All variable declarations in GLSL require a type annotation. I recognize that type annotations provide useful documentation, especially in function headers, so they should not be removed entirely. However, they make a program more verbose, and it should be possible to infer most or all of the types in a shader.

2.2.3 Other Sources of Inspiration

For further inspiration, I looked to general purpose languages.

Array programming languages, such as APL and J, interested me. They are notoriously terse; matrix multiplication in J can be written:

```
A +/ . * B
```

However, array programming languages make it easy to apply operations across arbitrarily dimensioned, nested arrays, and this is something I tried to recreate in Funslang.

As a functional language, though, most credit is due to Hindley-Milner languages such as ML and Haskell. The Haskell programmer may note that much of Funslang's syntactic sugar is borrowed from that language.

2.3 Further Planning

Thorough planning was an important factor in the success of this project.

2.3.1 Requirements Analysis

The first step was to make clear exactly what the project required:

- Selection of appropriate software tools

- Design of a shading language, including its syntax and type system
- Example shaders in the language
- Compiler front end: implementation of a lexer and parser
- Compiler back end: code generation
- Integration with a graphics library
- Live graphical demonstrations

The last two requirements were extensions to the Proposal, but I felt that demonstrating that the language could be used in a real graphical application would add significantly to the value of the project.

2.3.2 Project Dependencies

Each of these requirements represents a goal, but to plan properly I had to understand the dependencies between those goals. Each dependency complicates the project, makes the implementation less flexible, and is a potential point of failure. To quote Spolsky [2004, pp.250–251]:

> *Find the dependencies – and eliminate them.*

The dependencies in this project are shown in Figure 2.3.

2.3.3 Tools

The next task was to choose suitable tools.

Compiler Implementation

I have previously used C for a compiler, but found it lacking. In my opinion,[5] functional languages like ML and Haskell are better suited to writing a compiler, with their tagged-union data types and ease of expressing structural recursion. Automated garbage collection is also incredibly useful, as data structures tend to be complex and short-lived.

[5] . . . though others [VandenBerghe, 1998] agree

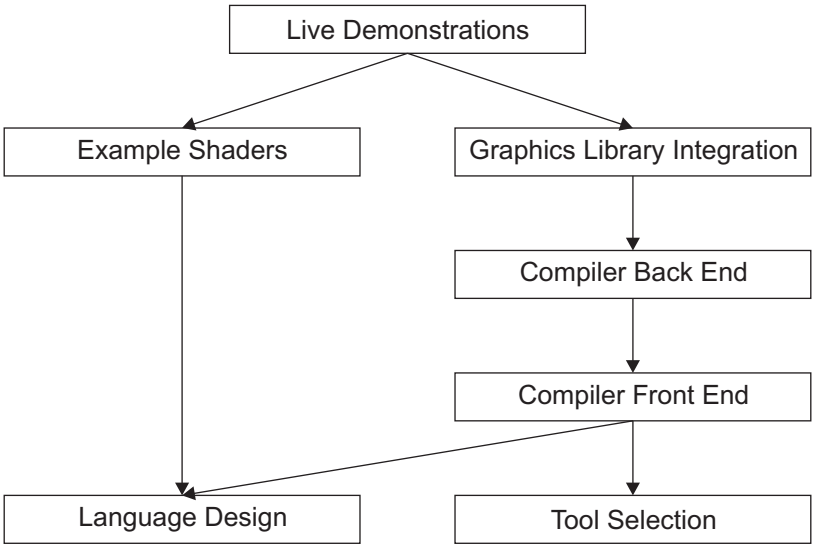


Figure 2.3: Project Dependencies.

The choice of Haskell over ML was personal, and mainly because I wanted to try something new. I also hoped that in learning Haskell I might find new ideas for use in my own language.

Haskell has many excellent tools available. In particular, I made use of the lexer and parser generators *Alex* and *Happy* – equivalent to the C tools `lex` and `yacc` – as well as its functional graph library and O($log(n)$) map and set implementations.

Haskell is a pure language, in that expressions cannot have side effects. However, it provides an interesting category theoretical abstraction called the *monad* to make state-passing simple and safe. I used monads extensively in the lexer and parser, the type checker, the interpreter and the C interface.

Compiler Target

Initially I thought that my compiler might target existing low-level shading languages [Brown, 2002–2004, Lipchak, 2002–2003]. However, while these might look like 'assembly' languages, they are not in the traditional sense – the architectural model is no longer representative of the underlying hardware. If I wanted to support features possible with modern hardware (such as texture accesses in the vertex shader) I would have to find something different.

Another option was one of the more recent vendor extensions to these languages [e.g. Brown, 2006], but I ruled this out as it would restrict my project to running on specific hardware.

I decided, therefore, to target one of the high-level languages: GLSL. This is widely supported, and could provide access to the features that I wanted. Obviously the extra layer of compilation would not be sensible in actual deployment; however, the point of this project is to demonstrate that Funslang could *replace* those high-level languages. Besides, the compiler outputs a three-address code that would be easy to convert to the desired machine code.

Graphics Library Integration

While Haskell is a good programming language for writing a compiler, it is not commonly used for writing graphical applications. For this, I would have to support C, the native language of OpenGL.

Thankfully, Haskell can be compiled to native code in standard C object files, and integration with C is well supported through its Foreign Function Interface [Chakravarty, 2002–2003].

Graph Visualization

I found `graphviz`, a graph visualization package from AT&T Research, invaluable for debugging compiler data structures.

2.3.4 Design Considerations

Another useful aspect of planning was to clarify my intentions for the language.

- The target architecture is unified, and there are no differences in capability between vertex and fragment shading. The language should be the same for both.
- The architecture is scalar-based, but the language should make vector and matrix mathematics as easy as possible.

- Programs with detectable runtime errors should be rejected by the compiler. This should include problems such as attempting to multiply matrices with incompatible dimensions.
- The language should not make assumptions about contiguity of memory; in particular, indirect memory accesses may not be possible. Thus any array references must be static, and the language cannot allow dynamic recursion [Mark et al., 2003, p.8].
- The architecture might not support dynamic branching, so it must be possible to eliminate control flow.

The static analysis guarantees imply that the language cannot be Turing-powerful. Interestingly, however, many graphical effects are still possible.

With these points in mind, the design and implementation process could begin.

Chapter 3

Implementation

The syntax, type system and library of Funslang is presented. Important design decisions are explained with examples. The implementation of the compiler is discussed. The compiler is integrated with OpenGL in C, and a 'development kit' prepared for the Evaluation.

3.1 Language Design

The design of Funslang was integral to this project. It borrows much from existing functional languages, but the design considerations of the previous chapter allowed me to make careful concessions in the interest of a 'lightweight' language.

In particular, I am very happy with the type system – it does without the theorem prover of a dependently typed language, yet it provides dimensional safety for vectors and matrices, and remains expressive enough for complex shaders. Funslang is indeed functional, but it can be compiled to an architecture with no concept of function calls.

3.1.1 Abstract Syntax

Funslang provides a lot of syntactic sugar, so its full grammar is long. However, the abstract syntax of Funslang is simple. The following recursive definitions can be represented directly as algebraic data types in Haskell.

Expressions

Expressions are defined similarly to ML or Haskell:

e	::=	()	unit literal (the 0-tuple)
		real	'real' (floating point) literal
		boolean	boolean literal
		identifier	variable
		$e_1\ e_2$	function application
		$[\ e_1\ ,\ \ldots e_n\ ]$	array construction
		$(\ e_1\ ,\ \ldots e_n\ )$	tuple construction
		if e_1 **then** e_2 **else** e_3	conditional expression
		let $p = e_1$ **in** e_2	let expression (non-recursive)
		$\boldsymbol{\lambda}\ p \longrightarrow e$	anonymous (lambda) function

A *shader* in Funslang is just a function. It takes certain parameters as passed by the graphics library, returning a value such as a colour. There is no need for any special syntax above expression level.[1] In Funslang, the entire program is a single expression.

The type signatures permissible for vertex and fragment shaders will be discussed later.

Note that there is no sequential composition operator ('`;`' in ML, '`seq`' in Haskell) as it would be pointless: Funslang is a pure language, and no side effects are possible. The persistent programmer could easily define it, however:

```
> let seq a b = let _ = a in b
<function>
a -> b -> b

> 0 `seq` 1
1.0
Real
```

The syntactic sugar used in this example will be explained in Section 3.1.5.

[1] Though the interactive debugging environment makes use of a top-level 'let' declaration, with no 'in' clause.

Patterns

In Funslang, lambda and let expressions can bind not just single identifiers, but arbitrary patterns. Patterns can optionally be annotated with a type, which the type inference algorithm will enforce.

p	::=	_	matches any expression and discards it
		()	matches unit expression
		$identifier$	matches any expression and binds it
		[p_1 , ... p_n]	matches array expression of same size
		(p_1 , ... p_n)	matches tuple expression of same size

Types

The rasterization model of graphics makes heavy use of *textures*, bitmap images that can be applied to surfaces to make them appear more lifelike.

No Funslang expression can create a texture handle, so they must be passed as parameters into the shader function by the graphics library. This is possible because textures are value types in Funslang. Native texture accesses are supported via the library functions `sample{1D,2D,3D,Cube}`.

Funslang provides four kinds of OpenGL texture:

texkind	::=	**1D**	texture line, indexed with 1D coordinates
		2D	texture plane, indexed with 2D coordinates
		3D	texture volume, indexed with 3D coordinates
		Cube	six 2D textures, arranged as the faces of a cube

The types of Funslang can thus be defined:

τ	::=	()	type of unit expressions
		Real	type of real numbers
		Bool	type of booleans
		Tex *texkind*	type of textures
		τ *dim*	type of arrays
		(τ_1 , ... τ_n)	type of tuples
		$\tau_1 \longrightarrow \tau_2$	type of functions
		α	type variable

Most of these types will be familiar to users of functional languages. In particular, note that functions can be passed as values.

The *array* type is new, though. Unlike the tuple, which is a heterogeneous

collection, the array is homogeneous. All the elements that it contains must be of the same type, allowing us to define higher-order functions such as 'map'.

However, arrays are not the same as lists in ML or Haskell. Unlike lists, arrays of different dimensions have different types:

dim	::=	*integer*	fixed dimension (≥ 1)
		δ	dimension variable

So the Funslang array lies somewhere between the familiar concepts of the tuple and the list. What makes arrays more interesting than either of those, however, is that dimensions can be unified in much the same way as types in the standard Hindley-Milner algorithm.

Arrays turn out to be remarkably powerful, and can fulfil the roles of vectors, matrices and (C-style) arrays from GLSL.

3.1.2 Type System

As stated above, Funslang's type system is a variant of that presented by Hindley [1969] and Milner [1978].

The typing rules of Funslang are given in Figure 3.1.

Polymorphism for Dimensions

Type variables are a concept key to the polymorphism of traditional functional languages. A type variable can 'stand in' for any other type, acting as a placeholder until that type is known. For example, a function that has been determined to have type $\alpha \to \alpha$ can be used in a situation where one of type $\mathbf{Bool} \to \mathbf{Bool}$ is required. Then we know that everywhere we see an α, we must replace it with **Bool**.

In Funslang, as in ML and Haskell, polymorphism is introduced by the 'let' typing rule, which universally quantifies free type variables to give a *type scheme*.

Funslang introduces a similar idea for array types: *dimension variables*. All array types have a dimension, either fixed (an integer), or a variable. This allows the type system to enforce dimensional constraints, which is critical for statically detecting errors in language that is to natively support vectors and matrices.

$$\text{(unit)}\ \frac{}{\Gamma \vdash () : ()}$$

$$\text{(real)}\ \frac{}{\Gamma \vdash \mathit{real} : \mathbf{Real}}$$

$$\text{(bool)}\ \frac{}{\Gamma \vdash \mathit{boolean} : \mathbf{Bool}}$$

$$\text{(var}\succ\text{)}\ \frac{}{\Gamma \vdash \mathbf{x} : \tau}\ ^{a}$$

$$\text{(app)}\ \frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1\ E_2 : \tau_2}$$

$$\text{(array)}\ \frac{\Gamma \vdash E_1 : \tau \ \ldots\ \Gamma \vdash E_n : \tau}{\Gamma \vdash [\ E_1, \ldots E_n\ ] : \tau\ n}$$

$$\text{(tuple)}\ \frac{\Gamma \vdash E_1 : \tau_1 \ \ldots\ \Gamma \vdash E_n : \tau_n}{\Gamma \vdash (\ E_1, \ldots E_n\ ) : (\ \tau_1, \ldots \tau_n\ )}$$

$$\text{(if)}\ \frac{\Gamma \vdash E_1 : \mathbf{Bool} \qquad \Gamma \vdash E_2 : \tau \qquad \Gamma \vdash E_3 : \tau}{\Gamma \vdash \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : \tau}$$

$$\text{(let)}\ \frac{\Gamma \vdash E_1 : \tau_1 \qquad \Gamma' \vdash E_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ p = E_1\ \mathbf{in}\ E_2 : \tau_2}\ ^{b}$$

$$\text{(lambda)}\ \frac{\Gamma' \vdash E : \tau_2}{\Gamma \vdash \boldsymbol{\lambda}\ p \rightarrow E : \tau_1 \rightarrow \tau_2}\ ^{c}$$

[a] where $\Gamma(\mathbf{x}) \succ \tau$, i.e. the type τ is a specialization of the type scheme $\Gamma(\mathbf{x})$

[b] where Γ' is the minimal extension of Γ such that $\Gamma' \vdash p : \tau_1$, and each new type scheme avoids capturing free variables of Γ

[c] where Γ' is the minimal extension of Γ such that $\Gamma' \vdash p : \tau_1$

Figure 3.1: Funslang typing rules.

Consider Funslang's `dot` function, which calculates the dot product of two vectors. Its type scheme is $\forall\delta$. **Real** $\delta \rightarrow$ **Real** $\delta \rightarrow$ **Real**. If we tried to apply it to vectors with different dimensions, say **Real 3** and **Real 4**, the type inference system would complain:

```
> dot [1..3] [1..4]
could not unify <Real 3> with <Real 4>
in expression: ((((dot) [1.0, 2.0, 3.0])) [1.0, 2.0, 3.0, 4.0])
```

ML and Haskell offer dimensional 'polymorphism', in a crude sense, because they do not care about the length of a list. It is not part of the type. However, they do not provide the same safety; I have been caught out more than once in Haskell by accidentally passing lists with mismatched lengths.

Everywhere type variables are used in the standard Algorithm W, the Funslang type inference system also operates on dimension variables.

For example, an important part of the algorithm concerns variable *substitutions*: which type variables correspond to which types – and in Funslang, which dimension variables correspond to which dimensions? Among other things, we need to be able to calculate the composition of two substitutions.

For ML, this is easy: we just apply the type variable substitution of one mapping to all the values of another mapping. For Funslang, the situation is more complicated, and we must think carefully about how the two pairs of mappings are combined. Code is given in Appendix A.1.3.

Dependent Types?

If dimensionality constraints are so useful, why don't ML and Haskell have them already?

The answer lies with *dependent types*. In a general purpose programming language with dimensionality constraints, we would probably want to allow the type (dimension) of an expression to depend on its value. However, we quickly run into problems with undecidability; how we might rescue ourselves is a current topic of research: the language Epigram [Altenkirch et al., 2005] is an interesting example.

Funslang gets away with it by cheating, and disallowing dependent types. However, this does not seem to matter at all for a shading language; all the library functions of GLSL are expressible in Funslang.

Type Checking is Optional

Type checking is incredibly useful when developing a shader on a workstation, but it might not be desirable in deployment. The algorithm is costly at run time: profiling the Mandelbrot shader indicates that inference takes 30–50% of the time and 20% of the total memory accesses. Besides, if shaders fail to type check on your customer's mobile phone, it is already too late.

Thus, another advantage of Funslang's type system is that it is decoupled from program evaluation. In Funslang, type checking is optional. This is in contrast to GLSL, where ad-hoc polymorphism means that typing information is needed to resolve function overloads.

Vectors and Matrices

In Funslang, vectors are represented by arrays, and matrices by arrays of arrays. This is similar to APL and J. Funslang takes the view that matrices are arrays of rows – an arbitrary decision, but one which allows us to write matrices in source form as we would on paper:

```
let getRotX a =
  let s = sin a in
  let c = cos a in
    [ [ 1,  0,  0, 0],
      [ 0,  c, -s, 0],
      [ 0,  s,  c, 0],
      [ 0,  0,  0, 1] ]
```

The GLSL syntax for constructing vectors uses functions, with different functions for `float` and `bool` vectors. I think that my syntax is more clear, but of course, if you prefer the old way, Funslang will let you recreate it:

```
> let vec2(x::Real, y) = [x, y]
<function>
(Real, Real) -> Real 2

> vec2(1,2)
[1.0, 2.0]
Real 2
```

Note that a type restriction is required if we want `vec2` to construct only **Real** arrays.

Unification, and Free Lunches

The type inference system uses *unification* to solve type constraints. If we know that a function has type $\alpha \to \mathbf{Bool}$, and is used in a context requiring $\mathbf{Real} \to \beta$, then we can infer that $\alpha \equiv \mathbf{Real}$, $\beta \equiv \mathbf{Bool}$, and that the function overall has type $\mathbf{Real} \to \mathbf{Bool}$.

This allows us to avoid some ad-hoc checks that are necessary with some shading languages. For example, in the low-level language ARB, we can declare a texture without explicitly saying which kind of texture it is, but once we have used it one way, its type is fixed. In Funslang, this behaviour is free.

Recursion

Due to its design considerations, Funslang intentionally disallows recursion. This is achieved by making the let rule non-recursive: the variable being defined cannot be accessed in its own definition.

Numeric Types

You may have noticed that there is only one numeric type, **Real**. GLSL supports integers, but Funslang could not without either introducing new operators for integers, or ad-hoc polymorphism.

Ad-hoc polymorphism does not interact well with type inference, but see Haskell's type classes for a very neat solution [Jones, 1998, pp.38–39].

However, we should ask whether integers are actually necessary [Kessenich, 2006, p.15]:

> *There is no requirement that integers in the language map to an integer type in hardware. It is not expected that underlying hardware has full support for a wide range of integer operations.*

Limitations

It is impossible to define a general append operator in Funslang's type system. There is no way to express a type like $\alpha\ \delta \rightarrow \alpha\ \epsilon \rightarrow \alpha\ (\delta + \epsilon)$. To solve this, we would need something like the theorem prover employed by Epigram.

However, this restriction actually turns out to be helpful, by reducing the number of type annotations. Most dimension changing operations in shaders are between vectors of length 3 and 4, because colors and positions can have 3 or 4 components, depending on whether you include the w-axis or the alpha channel.

The library defines two operations, `pad` and `strip`, to convert between the two representations. Because they only work with fixed dimensions, the type inference system learns more about their arguments.

Summary

Implementing type inference took a very long time, but I am pleased with the results. I found the examples given in Grabmüller [2006] to be a useful foundation; however, many cases were not treated. I added principal typing and unification for tuples, textures, and pattern-based let and lambda expressions, and of course everything related to arrays.

3.1.3 Shader Types

Shaders are simply functions. The permissible types of those functions are different for vertex and fragment shaders, but they follow the same general form: *uniforms* → *textures* → *attributes/varyings* → *result*.

- *uniforms* are those variables that do not change per shader run, such as the projection matrix.
- *textures* are those texture handles that the graphics library has initialized. These also do not change per shader run; however, because textures are fundamentally different to numeric parameters, I do not treat them as uniforms in the way that GLSL does.
- *attributes/varyings* are those variables that change per shader run, such as vertex positions or texture coordinates.

- *result* depends on the shader kind.

These correspond directly to the shader pipeline of Figure 2.1 on page 5.

For vertex shaders, *result* ≡ (**Real 4**, *varyings*), where **Real 4** is the transformed vertex position, and *varyings* are the variables to be interpolated as parameters to the fragment shader.

As *varyings* appears in both the vertex and fragment shaders, the inference system can unify the two types to complete a definition. Amazingly, this means that it is often possible to write (type-safe) shaders with no type annotations at all!

For fragment shaders, there is a choice: either *result* ≡ **Real 4**, a color, or *result* ≡ (**Bool**, **Real 4**), where the extra boolean is a flag that determines whether the pixel should be kept (**True**) or discarded (**False**). The latter allows certain transparency effects to be achieved; however, it is rarely used, so the type checker will only try it if the former unification failed.

Note that in keeping with good design principles for functional languages, the parameters that change most frequently appear last. In general, this means that functions can be easily composed, allowing code reuse.

3.1.4 Library

The library of Funslang is an important part of the language design. In designing the library, I tried to maintain a clean, consistent approach without compromising on the functionality provided by GLSL.

Magic

As with most high level languages, the Funslang library must provide 'magic' functions. The user has to trust that these work as described, because there is no way of reproducing them in the language. Indeed, while it is not Turing-complete, a proof that all Funslang programs terminate would need to consider the implementation of these functions.

However, in Funslang the magic functions are all typeable. For example, the arithmetic operators are handled just like any other function.

Funslang provides 81 library functions, of which around 60% are magic. Most of these functions encapsulate hardware operations, such as arithmetic and

texture lookups. The rest are functions that are polymorphic in array dimension, such as transpose and higher-order map.

Dimensional polymorphism cannot be derived in the type system, because there is no primitive in Funslang for manipulating arrays of general length. However, the functional nature of the language means that it is easy to define new polymorphic functions once you have a few higher-order building blocks.

The remaining 40% of functions are literally implemented in the compiler as Funslang source code.[2] Not only is this convenient, it also ensures automatic testing. For example, the matrix-vector linear algebraic multiply is defined as:

```
> \m v -> map (dot v) m
<function>
Real m n -> Real m -> Real n
```

Note that the correct dimensional constraints are inferred from the `map` and `dot` functions.

Implementation

The library is constructed in two phases. First, type and value environments are populated with the magic functions, then the remaining functions are derived using these environments.

The library is stored as two lists of tuples. For example, here is the `sin` tuple:

```
("sin", "Real -> Real", "sine (radians)", liftRR sin DFRealSin)
```

It holds the name of the function as well as its type, description and implementation. The type is stored in Funslang syntax because the internal representation of types – representing fresh type and dimension variables with integers – is not easily maintainable. The implementation is kept simple by using a higher order 'lift', which steals Haskell's sine function.

In this way, all the implementation details for a particular function are kept together. These details are never duplicated – for example, the type of a derived function is not specified, as it can be inferred.

[2] In a production compiler, efficiency could be gained by serializing this part of the library.

Documentation

A full list of library functions as well as their types can be found in Appendix A.3. This documentation was generated automatically.

3.1.5 Syntactic Sugar

The phrase *syntactic sugar* refers to syntax constructs that are useful to a human programmer, but which can easily be translated – 'desugared' – to more primitive forms. Sugar does not change the semantics of a language, but it can make it sweeter!

Much of Funslang's sugar will be familiar to Haskell programmers.

Function Definitions

Syntactic sugar for function definitions allows us to define a multiple argument function directly. This can replace the verbose:

```
> let add = \x -> \y -> \z -> x + y + z
<function>
Real -> Real -> Real -> Real
```

with the concise:

```
> let add' x y z = x + y + z
<function>
Real -> Real -> Real -> Real
```

This is a very useful feature, and I quite proud of its implementation. In Haskell, it can be achieved in a single line, using the higher order `foldl` function on the list of arguments to create nested lambdas:

```
ExprLet
  (PattVar identifier optional_type)
    (foldl (flip ExprLambda) e1 pattern_list)
      e2
```

Array Ranges

Array ranges allow you to create sequential arrays easily, with the notation `[a..b]`.[3] This can be neatly implemented using Haskell's version of the same sugar.

Sections

Infix operators such as addition are standard functions in Funslang's abstract syntax. However, their concrete syntax prevents you from easily accessing that function – you have to wrap the operator in a lambda abstraction.

Sections let you refer directly to the function by placing the operator in parentheses, optionally with a left or right argument.

For example, the library function `sum` is concisely defined:

```
> let sum = foldl1 (+)
<function>
Real m -> Real
```

Backtick Expressions

While sections let you convert infix operators into functions, backtick expressions let you convert functions into infix operators. For example:

```
> [1,2,3] `cross` [4,5,6]
[-3.0, 6.0, -3.0]
Real 3
```

This requires just a single grammar production.

Swizzles

Swizzles are a concept from GLSL that allow you to extract vector components by name. For example, `v.x` selects the first component of `v`. Interestingly, though, swizzles also allow multiple and repeated components: `vec2(1,2).xyy` would give `vec3(1,2,2)`.

[3] Because Funslang does not have dependent types, the range must involve compile time constants.

In Funslang, swizzling is implemented with a function, '`!!`'. It can even recreate the original GLSL syntax:

```
> let xyy = [0,1,1]
[0.0, 1.0, 1.0]
Real 3

> [1,2]!!xyy
[1.0, 2.0, 2.0]
Real 3
```

3.2 Compilation

Bearing in mind that the target architecture cannot dynamically allocate or even reference memory – ruling out both a heap and a stack – it was not obvious to me that a functional shading language was even possible.

3.2.1 'Pre-Runtime' Interpretation

In short, I needed to prove that it is always possible to simulate the execution of a Funslang shader without the use of function closures, and then find a way to do so.

I realized that this followed from two key points:

- The lack of recursion guarantees termination, regardless of input parameters.
- The type checker guarantees that neither the parameters to a shader nor its return type include functions.

The first point tells us that we can actually simulate the shader at compile time, and the second means that we can use the simulation to emit code without function closures. While function types may be constructed during execution, they must be fully applied if the result is to depend on them.

My idea was to pass the shader dummy arguments, and keep track of what hardware operations (such as addition) were applied to them. In effect, the compiler would build a dataflow graph. Any valid topological sort of this

graph would produce code corresponding exactly to the execution of the shader.

Because our architecture does not support dynamic branching, control flow has to be eliminated. However, the referential transparency of Funslang makes this easy. The only source of divergent control flow in Funslang is the conditional if-expression; we can simply evaluate both arms and merge back based on the condition.

Note that this is much easier than in GLSL, which in addition to conditional expressions also has if-*statements* and side effects.

If dynamic branching became possible in the future, we would obviously choose to execute only one arm. Such an architecture would also allow us to implement dynamic looping, or primitive recursion. Funslang could be extended to support this without affecting the termination guarantee.

3.2.2 Phases

Compilation proceeds in four main phases: generating the abstract syntax tree, type checking, interpretation and code emission.

Generating the Abstract Syntax Tree

Haskell's algebraic data types make the representation of recursive data structures easy. A sample is given in Appendix A.1.1, following directly from the definitions of Section 3.1.1.

A small sample of the LALR(1) grammar is given in Appendix A.1.2. I tried to keep the grammar actions as clean as possible, so I hope that it is readable.

Type Checking

The type system was described in Section 3.1.2. Code extracts from the substitution composition and principal typing algorithms can be found in Appendices A.1.3 and A.1.4 respectively.

Interpretation

The foundation of the interpreter is the `Value` data type:

```
data Value
  = ValueUnit
  | ValueReal !DFReal
  | ValueBool !DFBool
  | ValueTex !DFTex
  | ValueArray ![Value]
  | ValueTuple ![Value]
  | ValueFun !(Value -> InterpretM Value)
```

There is a `Value` case for every Funslang type, so it can represent the result of any expression. Note that Funslang functions are represented by Haskell closures.

As simulation unfolds, `Value`s are manipulated. The dataflow graph representing the computation can be recovered from the resulting `DFReal`, `DFBool` and `DFTex` nodes.

It turns out that functional languages are very good at expressing other functional languages: the core interpreter, defined inductively on the syntax of expressions, is quite straightforward. Code is given in Appendix A.1.5.

Most of the work involved in creating dataflow graphs is actually in the library – each of the magic functions requires a `ValueFun` definition. This is where the `lift*` functions from Section 3.1.4 are useful. Recall that all we have to do is pass them the dataflow constructor:

```
liftRR sin DFRealSin
```

Figure 3.2 shows a simple graph generated by a fragment shader with two texture accesses. However, the graphs of complex shaders can be intimidating – see the Evaluation for more.

Code Emission

Dataflow nodes represent the results of operations, while edges represent (true) data dependencies. To generate linear code from the graph, we just need to respect those dependencies by performing a topological sort.

I designed the granularity of the operations to suit the machine instructions of a 'reasonable' graphics architecture. For example, there are nodes for memory loads, addition, texture lookups and trigonometric functions. For reasons

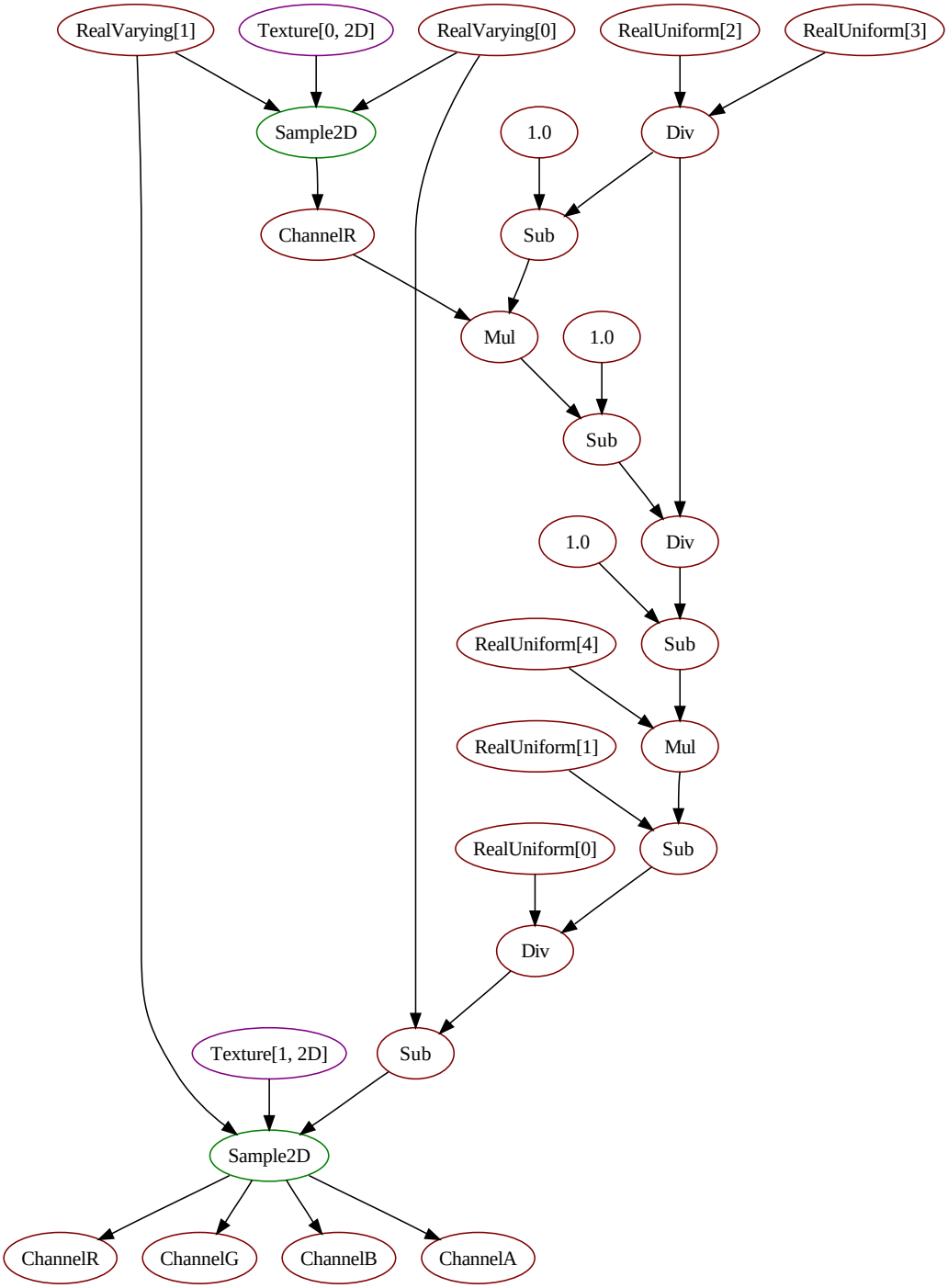


Figure 3.2: Dataflow graph of autostereogram fragment shader.

discussed in Section 2.3.3, my compiler emits GLSL code; however, it would not be hard to retarget the compiler to a proprietary graphics processor.

All directed acyclic graphs (DAGs) have at least one possible topological sort. In practice, most shaders have many different topological sorts, corresponding to different code schedules. Choosing the 'best' such sort is impossible without knowledge of the machine architecture, but again, this would not be difficult.

The code to emit GLSL source is tedious, and I will not reproduce it here. Some sample output is given in the Evaluation.

3.2.3 Errors

The Funslang compiler responds to user errors gracefully and consistently. When an error is detected, all the information required to convey it is stored in a type-safe data structure. As the error propagates, further information may be added, recording contextual information such as the expression stack of the interpreter.

The data structure used thus forms an exhaustive list of the possible user errors. It is shown in Appendix A.1.6.

In normal use, the error data is simply converted to a user-friendly error message. However, the type safety can also be used by the automated test suite to determine exactly which error occurred, permitting the expression of negative assertions. The test suite is discussed further in the Evaluation.

3.2.4 Command Line Interface

The compiler is accessible from the command line, giving the options shown in Figure 3.3. Of note are the creation of dataflow graphs (in `graphviz` format), the interactive debugging environment, the automated test suite, and the generation of library documentation.

The implementation of this interface is decoupled from the compiler core. It was constructed using the Haskell module `System.Console.GetOpt`, which dispatches control to the appropriate compiler functions based on the command line arguments.

```
Usage: main [options] [input files]
  -g    emit dataflow graph (must be combined with -c or -e)
  -c    compile and link vertex and fragment shaders from
        separate files
  -e    evaluate expression from file
  -i    enter interactive environment
  -t    run automated tests
  -l    print library in LaTeX format
```

Figure 3.3: Options given by the command line compiler.

3.3 Graphics Library Integration

To present my shading language as a full solution, I realized that I should demonstrate integration with an existing graphics library.

I therefore decided to make the compiler accessible from C.

3.3.1 Haskell and the Foreign Function Interface

Luckily, Haskell's Foreign Function Interface [Chakravarty, 2002–2003] makes exporting Haskell functions relatively painless. A single line in the Haskell source indicates that a function is to be exported, for which the Haskell compiler will generate stub `.c` and `.h` files.

In fact, only two functions were exported: `fsCompile`, which encapsulates the compiler, and `fsFree`, which allows the C code to indicate to Haskell that it may deallocate memory.

The Haskell code accepts pointers to the Funslang source, and returns the type of the shaders, details of its parameters, and GLSL source. In true C spirit, the function manages to return multiple values by mutating pointers belonging to the calling code.

3.3.2 Funslang API

Of course, we should not expect the user to know these details, so a clean programming interface is provided. There are functions for:

- initializing and terminating the Haskell runtime
- compiling Funslang source
- setting up OpenGL shader parameters (uniforms/attributes/varyings) automatically
- setting up OpenGL texture hardware automatically
- loading textures from JPEG files[4]

The header file is presented in Appendix A.1.7.

3.3.3 Funslang 'Development Kit'

The Haskell compiler provides C object files that could be linked into the user's application. However, it would be unreasonable to require the user to have a Haskell compiler installed, so I created a shared library to store the compiled code. As I was developing on Windows, this was a dynamic link library (DLL), but a Unix port would not be difficult.

The resulting memory management issues are awkward – there are three separate allocators in the user code, the DLL and the Haskell runtime, not to mention Haskell garbage collection – so care was required to avoid memory leaks.

To ease development, I wrote a script to automatically generate Visual Studio projects with all the necessary OpenGL and Funslang libraries and headers included. The interactive development environment, of which you have already seen sample output, is also helpful.

3.4 Summary

In Funslang, I designed a shading language that is simple, yet flexible. The syntax, library and type system make vector and matrix mathematics easy and safe.

The library is comprehensive, but the compositionality of a functional language keeps it small.

[4] using `libjpeg` from `http://www.ijg.org`

The type system guarantees to detect most run time errors statically, without requiring the ad-hoc checks of existing languages. Unlike those languages, type checking in Funslang is fully decoupled from the execution model, so devices with limited computational power could omit it altogether.

The target architecture is limited, but my compiler embraces those limitations, emitting code with no run time overhead.

As an extension, I successfully integrated the compiler with OpenGL, creating several example applications that will be the topic of the next chapter.

Chapter 4

Evaluation

Testing methods are outlined, referencing output from the automated test suite. Two novel examples are presented, demonstrating the power and flexibility of computation on the graphics card. The six required Funslang shaders are explained. Results are produced, giving screen shots and dataflow graphs, and positive conclusions drawn.

4.1 Testing

The purpose of this chapter is to convince the reader that the Funslang 'development kit' works reliably, and as intended. Example applications are the best indication of this, but I should first like to mention the thorough testing that was carried out throughout the implementation phase. Certainly, testing was not limited to the end of the project, and I believe that this was key to its success.

4.1.1 Prototyping

In the Preparation chapter I stressed the importance of straightforward module dependencies: most of the modules in this project depend on one module, and have only one dependent. This made it possible to always have a usable prototype of the compiler. I began with a lexer and parser, and later added type checking, interpretation, and code emission. Any bugs in one module

were fixed before the next was started.

With no complex module interactions, I could test modules individually and be confident that the system would work.

4.1.2 Interactive Debugging Environment

The interactive Funslang environment allows the user to modify variable bindings dynamically, and query the values and types of any Funslang expression. It proved to be very useful for quick tests.

Example output can be found throughout the dissertation.

4.1.3 Automated Test Suite

Informal testing played an important role in my project, but I also designed an automated test suite to systematically test features and bug regressions. In fact, the build process ran the test suite after compilation, so that the build would actually fail if any of the tests did not pass.

The test suite can express both positive and negative assertions. It can test not only the result of good input, but also the response to bad input. Each test case is encapsulated in a Haskell algebraic type.

For example, the type unification algorithm must reject recursive type variable constraints such as $\alpha \equiv \alpha \rightarrow \beta$. This is called the *occurs check*. The lambda term $\lambda x.xx$, which applies x to itself, is untypeable due to the occurs check. We expect the Funslang compiler to reject it:

```
TestExprCompileError {
  name = "TypeErrorOccursCheck",
  expr = "\\x -> x x",
  expect_error =
    \ e -> case e of
      TypeError _ (TypeErrorOccursCheck _ _) -> True;
      _ -> False
  }
```

Negative assertions are only possible because the test suite can distinguish between the different errors – in this case, `TypeErrorOccursCheck` is required.

```
C:\Users\Ben\Trees\ben_rlaptop\project\code\compiler>bin\main -t
Testing typing rules:
  unit... OK
  real 1... OK
  real 0... OK
  real -1... OK
  bool T... OK
  bool F... OK
  var... OK
  app fn... OK
  app res... OK
  array... OK
  tuple... OK
  if... OK
  let... OK
  lambda... OK
Testing errors:
  LexerErrorNoLex... OK
  LexerErrorIdentBeginsUpper... OK
  ParserErrorNoParse... OK
  ParserErrorBadFixedDim... OK
  TypeErrorOccursCheck... OK
  TypeErrorCouldNotUnify... OK
  TypeErrorUnboundVariable... OK
  TypeErrorDuplicateIdentsInPattern-array... OK
  TypeErrorDuplicateIdentsInPattern-tuple... OK
  ShaderErrorBadShaderType... OK
  ShaderErrorBadUniformType-tex... OK
  ShaderErrorBadUniformType-fn... OK
  ShaderErrorBadTextureType-real... OK
  ShaderErrorBadTextureType-fn... OK
  ShaderErrorBadVaryingType-tex... OK
  ShaderErrorBadVaryingType-fn... OK
  ShaderErrorBadOutputType... OK
  ShaderErrorCouldNotLink... OK
  InterpreterErrorIndexOutOfBounds... OK
  InterpreterErrorDynamicUnroll... OK
  InterpreterErrorDynamicIndex... OK
  InterpreterErrorFunctionEquality... OK
  TargetErrorGLSLDynamicTextureSelection... OK
Testing bug regressions:
  should not trigger occurs check... OK
  typing variable shadowing in lambda... OK
  interpreting variable shadowing in lambda... OK
  typing variable shadowing in let... OK
  interpreting variable shadowing in let... OK
  iterate stops off by one... OK


43 tests passed, 0 tests failed and 0 tests had unexpected errors.
```

Figure 4.1: Sample output from the automated test suite.

If any compilation failure were acceptable, the test might pass erroneously. For instance, all negative assertions with syntax errors would pass.

An example test run is shown in Figure 4.1.

4.2 Example Applications

4.2.1 Autostereograms

The first example I will present is an autostereogram generator. Commonly known by the *Magic Eye* trademark, autostereograms are two-dimensional images that give the illusion of three-dimensional depth.

By modulating a periodic background image with depth information, the brain can be tricked into thinking that each eye is viewing a different angle of a three-dimensional scene. It infers that the image has depth.

While the rasterization model of rendering is already optimized for calculating depths, it would be difficult to take advantage of this efficiently without shaders. Therefore, this example is a particularly good demonstration of Funslang.

However, the main reason for choosing autostereograms is that I think they are good fun. I had always wondered how they were generated, and this was a chance to find out.

Autostereograms reproduce well in print, so you will be able to try them for yourself. This often takes practice, as you must learn to decouple your focussing from your convergence: you still focus on the image, but you converge your eyes *behind* it.

If you want to view the final image before I reveal what is in it, look at Figure 4.2 now!

Method Outline

In the interest of brevity, I will not explain the biology or trigonometry behind autostereograms. Many websites do this already. The novelty of this example is that the autostereograms are generated on the graphics processor.

In outline, this is the method that we will use:

1. Render a three-dimensional scene to create a *depth texture*, a monochromatic image where the intensity of a pixel represents its depth.

2. Fill a strip down the left hand side of the image with a seamless, repeating texture. This will form the background of the autostereogram.

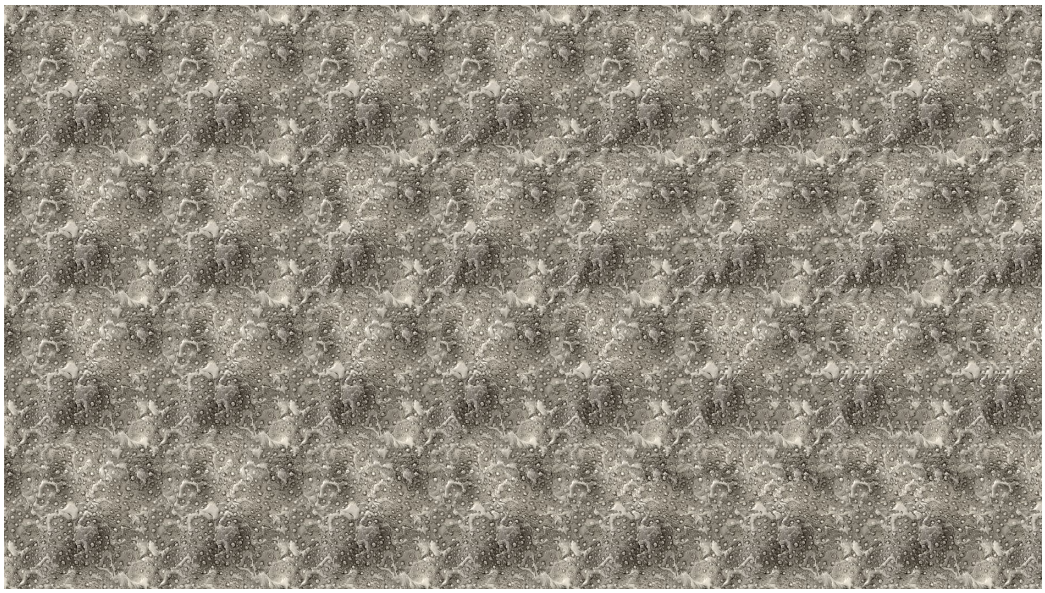


Figure 4.2: A mystery autostereogram.

Any pattern will do, so it can be chosen on an aesthetic basis. However, the width of the tile is important: too narrow, and the representable depth levels will be too coarse; too wide, and no stereo information will be encoded at all. I found that a tile that fits five or six times into the width of the image works well.

3. Proceed from left to right across the image, colouring pixels as follows:
   (a) Calculate the depth level d for this pixel using the depth texture.
   (b) Where w is the width of the background tile, choose the colour of this pixel to be the same as that $w - d$ to the left.

When colouring a pixel, the fragment shader must look up the colour of a pixel to its left. As the output pixels are not all independent, we cannot colour the autostereogram in a single pass. After each pass we must copy newly-coloured pixels back to a texture ready for the next pass to access.

A naïve and safe approach would be to have one pass per pixel column. However, the overhead would be unnecessarily high: we can actually render strips of width $w - D$, where D is the maximum depth level. To see why this is the case, note that all the pixels in such a strip are necessarily independent. Even a pixel on the far right depends on a pixel at least $w - D$ to its left, and thus not in its strip.

In total we need to write two pairs of shaders: one pair to seed the background texture, and another to calculate the remaining autostereogram strips.

Shaders to Seed Background Texture

The shaders used here are a good introduction, as they are about the simplest possible examples.

The vertex shader takes two vertex attributes: the position of the vertex, and its associated texture coordinates. It outputs the position directly, and passes the texture coordinates to be interpolated for the fragment shader:

```
\ () () (p, c) -> (p, c)
```

The fragment shader takes a texture and a coordinate, and samples the texture using that coordinate:

```
\ () (t) (c) -> sample2D t c
```

However, note that this can be written equivalently and even more neatly:

```
\ () -> sample2D
```

Try writing a 13 character shader in any existing language!

Shaders to Calculate Autostereogram

The autostereogram vertex shader is still very simple. Given coordinates relative to the window, all it does is output the coordinates in one range for the graphics library and another for the fragment shader:

```
\ () ->
\ () ->
\ [windowCoordX, windowCoordY] ->

  ([windowCoordX * 2 - 1, windowCoordY * 2 - 1, 0, 1],
   [windowCoordX, windowCoordY])
```

The fragment shader is more complex, and deserves a more detailed explanation. First, we take some uniform parameters, a pair of textures and the interpolated window coordinates.

```
\ (windowW, tileW, zNear, zFar, numDepthLevels) ->
\ (depthTex, outputTex) ->
\ (windowCoords) ->
```

We then sample the depth buffer, and undo the projection matrix to give a meaningful depth value.

```
% Get depth buffer value.
let [projectedDepth,_,_,_] = sample2D depthTex windowCoords in
% Convert to linear depth
% so 1 is at camera (not zNear!) and 0 is at zFar.
let zRatio = zNear / zFar in
let linearDepth = 1 - zRatio/(1 - (1 - zRatio)*projectedDepth) in
```

With this depth value we can locate the pixel from which to source our colour.

```
% Find horizontal shift (in pixels) based on linearDepth.
let shift = linearDepth * numDepthLevels in
% Find horizontal period (in pixels) for this depth.
let period = tileW - shift in
% Find the coordinates with which to index the previous strip.
let prevOutputCoords =
  [windowCoords!0 - period/windowW, windowCoords!1] in
```

Finally, we can look up the colour of that pixel.

```
    sample2D outputTex prevOutputCoords
```

The type of this shader is:

```
(Real, Real, Real, Real, Real) ->
  (Tex 2D, Tex 2D) ->
    Real 2 ->
      Real 4
```

Note that this can be inferred without *any* type annotations.

The dataflow graph for this shader was given as an example on page 33, while the GLSL code emitted by the compiler is relegated to Appendix A.2.1.

Graphics Library Code

The shaders are the interesting part of the example, but of course they do not work by themselves. The control logic requires a fair amount of boilerplate code, which is too long to reproduce in full. However, the code specific to initializing Funslang shaders is quite simple. Here is a taste:

```
g_ProgramStereo.vertex_shader_path = "../../funslang/Stereo.vp";
g_ProgramStereo.fragment_shader_path = "../../funslang/Stereo.fp";
if (!fsCompile(&g_ProgramStereo)) exit(1);
glUseProgram(g_ProgramStereo.glsl_program);
fsSetVertexUniforms(&g_ProgramStereo, NULL);
fsSetFragmentUniforms(&g_ProgramStereo, (GLfloat*)&g_StereoFragmentUniforms);
fsSetTextureImageUnits(&g_ProgramStereo);
```

In particular, note the `fs*` functions that I provide for initializing shader variables.

Results

The final image was shown in Figure 4.2. The depth texture used to create that image is visualized in Figure 4.3, revealing it to be a teapot.

Thanks to parallel execution on the graphics card, autostereogram shaders add little overhead. The performance of this example is very good – even on my laptop, it runs at 150 frames per second.

The shaders could easily be appended to existing OpenGL code, making 'real' three-dimensional gaming a reality. However, initial experiments indicate that this is quickly fatiguing!

4.2.2 Mandelbrot Fractals

For the next example, I wanted to try something that would stress the graphics card. I chose, therefore, to generate the Mandelbrot set: by increasing the resolution or the number of iterations, fractal generation can be made arbitrarily hard.

To show off the power of shaders, I thought that it would be interesting to render the fractals on the faces of a spinning cube.

Figure 4.3: A depth map reveals the mystery autostereogram.

The fragment shader is based on an existing GLSL shader by 3Dlabs; for comparison, that code is in Appendix A.4.1. The vertex shader is entirely original, and demonstrates how easily standard vertex projections can be expressed in Funslang.

Method Outline

Again, I will only outline the algorithm here. If you are interested in more detail, there are many textbooks and websites that explain the mathematics.

At the heart of the Mandelbrot fractal is the iteration formula:

$$\begin{aligned} z_{n+1} &= z_n^2 + c \\ z_0 &= 0 \end{aligned}$$

Each point c in the complex plane corresponds to a different iteration. Most points in the complex plane diverge to infinity; those which do not escape are said to be in the Mandelbrot set.

By colouring points based on how quickly they diverge, we generate the Mandelbrot fractal.

Vertex Shader

Because we are interested in rendering a rotating cube, the vertex shader must be capable of performing the necessary vertex transformations to transform the cube from model to clip space. To achieve this, incoming vertices are subjected to model, rotation, view and projection matrices.

Rather than pass all of these matrices component by component, I decided to construct some of them in the vertex shader. This wastes negligible computation, because the vertex shader is only called four times per quadrilateral. However, it saves effort: matrices are much easier to work with in Funslang than in C.

The rotation matrices are defined in terms of an angle about a primary axis:

```
let getRotX a =
  let s = sin a in
  let c = cos a in
    [ [ 1,  0,  0, 0],
      [ 0,  c, -s, 0],
      [ 0,  s,  c, 0],
      [ 0,  0,  0, 1] ] in
```

`getRotY` and `getRotZ` are defined similarly.

Next is the view matrix, defined in terms of the camera position, its subject position, and the 'up' direction:

```
let lookAt from to up =
  let n = normalize $ from -- to in
  let u = normalize $ up `cross` n in
  let v = n `cross` u in
      [ [u!0, u!1, u!2, -(from `dot` u)],
        [v!0, v!1, v!2, -(from `dot` v)],
        [n!0, n!1, n!2, -(from `dot` n)],
        [ 0 ,  0 ,  0 ,        1       ] ] in
```

With those matrices defined, the shader simply applies them to the incoming vertex:

```
\ (proj, model, rotx, roty, rotz, from, to, up) ->
\ () ->
```

```
\ (p) ->

let view = lookAt from to up in
let rot = getRotZ rotz # (getRotY roty # getRotX rotx) in

  (proj #. view #. rot #. model #. pad p, (p![0,1]))
```

Funslang actually conceals a great deal of complexity here: four-dimensional linear algebra is expensive. The dataflow graph is shown in Figure 4.4.

Fragment Shader

The fragment shader is where the Mandelbrot fractal is calculated.

First, we define a sensible limit to the number of iterations, determined by experiment.

```
let maxIterations = 50 in
```

We then take some parameters, and the position interpolated from the vertex shader. Note that because the colours are parameterized, they can be changed at run time in the C code.

```
\ (zoom, center, innerColor, outerColor1, outerColor2) ->
\ () ->
\ (pos) ->
```

We calculate a value for c, our position on the complex plane.

```
% starting values
let [cr, ci] = pos **. zoom ++ center in
```

Now we are in a position to define the iteration function. To understand this, we have to understand the higher-order library function `iterate`, with type:

```
(a -> (Bool, a)) -> Real -> a -> a
```

`iterate f n z` will apply `f` repeatedly to `z`, stopping either when `n` iterations have occurred, or when `f` returns `False`. So we define `f` to do a single Mandelbrot iteration:

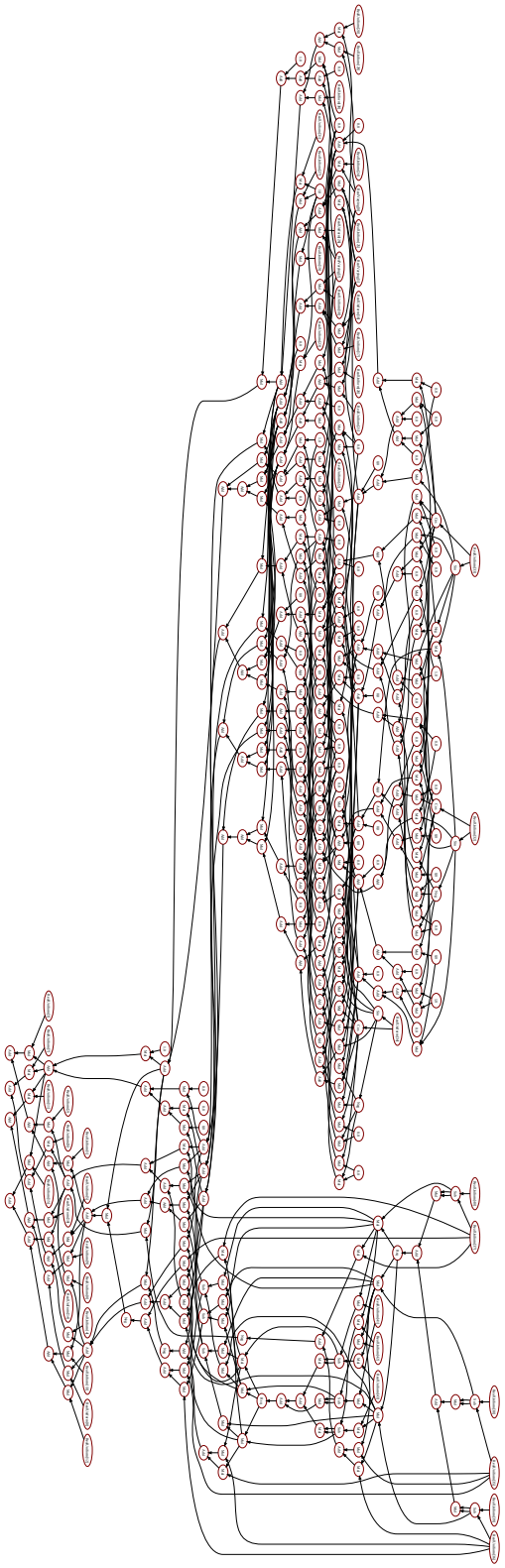


Figure 4.4: Dataflow graph of Mandelbrot vertex shader.

```
% iteration function
let f ([r, i], [r2, i2], iter) =
  let r' = r2 - i2 + cr in
  let i' = 2 * r * i + ci in
  let r2' = r' * r' in
  let i2' = i' * i' in
    (r2' + i2' < 4.0, ([r', i'], [r2', i2'], iter+1)) in
```

Note that if $|z_n|^2$ ever exceeds 4, it is guaranteed to diverge. Thus, we use this as the termination condition.

Finally, we can perform the iteration...

```
% iterate
let (_, _, iter) = iterate f maxIterations ([0.0, 0.0], [0.0, 0.0], 0) in
```

...and colour the point based on how soon it terminated:

```
% color
let basecolor =
  if iter == maxIterations
    then innerColor
    else zipWith (mix (fract (iter * 0.05))) outerColor1 outerColor2  in

  pad basecolor
```

The dataflow graph for a 10 iteration version of this shader is shown in Figure 4.5. You can quite clearly see the individual iterations.

Results

As promised, the fractals are shown on the faces of a cube in Figure 4.6. Unfortunately, you will have to take my word that it spins!

On my desktop graphics card, the example runs smoothly at 60 frames per second. That is very impressive: consider that the fragment shader runs once per pixel, and a 50 iteration shader requires around 500 floating point operations. At 2000 by 1000 pixels, that is 60 billion operations per second.

When I originally translated this shader from GLSL, and looked at the dataflow graph, it was apparent that it was recalculating some multiplies

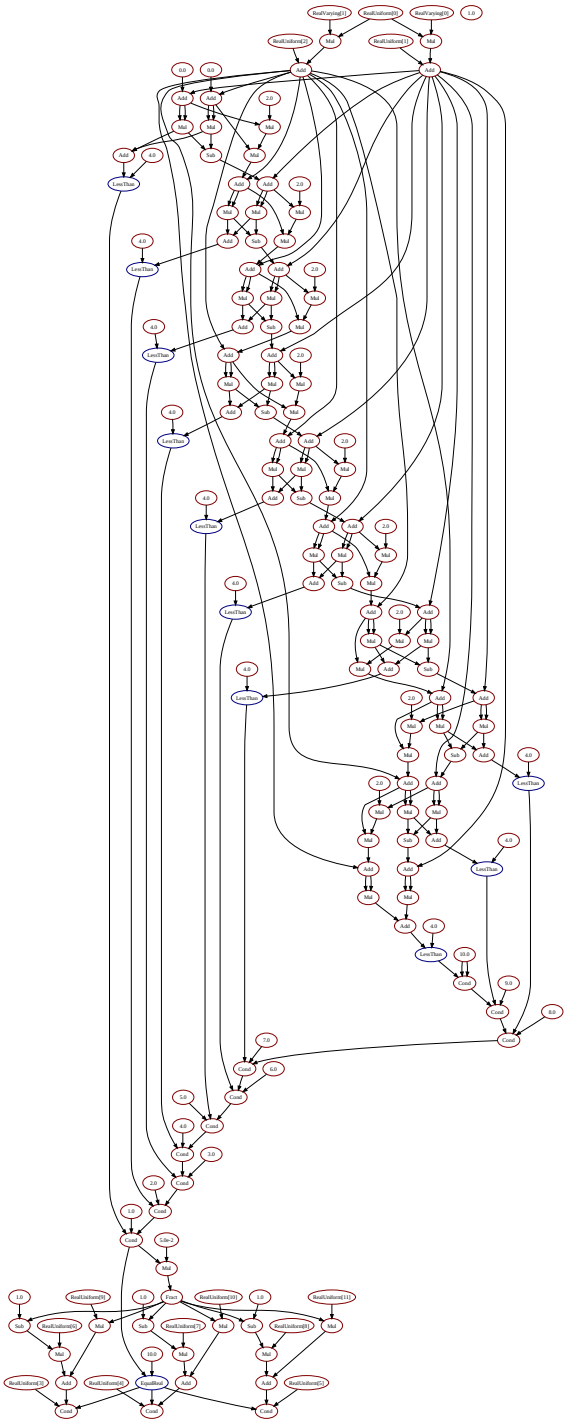


Figure 4.5: Dataflow graph of Mandelbrot fragment shader.

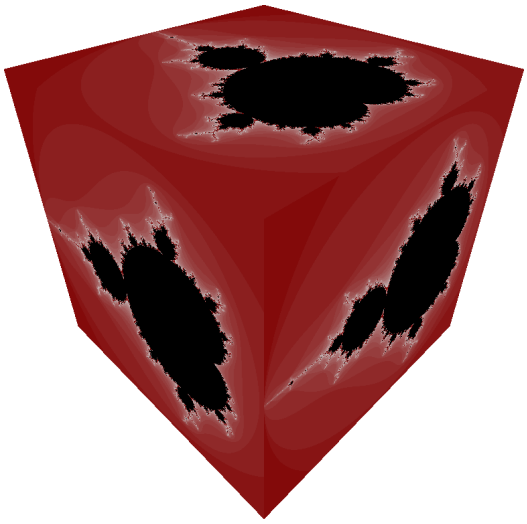


Figure 4.6: Mandelbrot fractals on the faces of a (spinning) cube.

between loop iterations. This must have escaped the attention of the GLSL authors. With an optimization in place, around 100 instructions are saved – 20% of the shader.

Compared to the GLSL code, I hope you will agree that the Funslang code is more declarative, and more succinct. Again, it requires no type annotations.

4.3 Summary

Six Funslang shaders have been presented in two applications. These work reliably and perform well. None of these shaders require any type annotations, neatly demonstrating the type inference system.

The compiler was written in a strongly-typed language, and has been tested thoroughly both manually and with an automated test suite. Funslang was designed to have few 'corner cases', and I have confidence in its implementation.

Chapter 5

Conclusions

Having no familiarity with language design, I was perhaps ignorant of the risk I was taking with this project. Indeed, looking back on my original ideas, some of them are quite humorously bad. However, Funslang met or surpassed the goals of my Proposal, and the project has been an experience that I would happily repeat.

Learning Haskell was a challenge, but it has given me new insight and greatly improved my technical skills. I would very much recommend a functional language to anyone planning to implement a compiler.

Graphics hardware is evolving quickly. If I were to repeat this project in several years' time, I would extend the language to support upcoming architectural features such as dynamic branching, and new pipeline abstractions such as the geometry shader.

Currently, however, there is little in Funslang that I would change. I set out to create a lightweight shading language, and I think I have succeeded. Indeed, Funslang is more flexible than I expected: it takes advantage of hardware-imposed restrictions, but these could be lifted without greatly affecting the language.

Bibliography

Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Technical report, University of Nottingham and University of St Andrews, 2005. URL `http://www.dcs.st-and.ac.uk/~james/RESEARCH/ydtm-submitted.pdf`. 3.1.2

Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4): 17–20, 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/278283.278285. URL `www.cs.princeton.edu/~appel/papers/ssafun.ps`. 2.2.2

Pat Brown. *ARB_vertex_program*. Silicon Graphics, Inc., 45 edition, 2002–2004. URL `http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt`. 1, 2.3.3

Pat Brown. *NV_gpu_program4*. NVIDIA Corp., 2 edition, 2006. URL `http://developer.download.nvidia.com/opengl/specs/GL_NV_gpu_program4.txt`. 2.3.3

Manuel Chakravarty. *The Haskell 98 Foreign Function Interface*, 1.0 edition, 2002–2003. URL `http://www.cse.unsw.edu.au/~chak/haskell/ffi/ffi.pdf`. 2.3.3, 3.3.1

Frank DeRemer and Hans Kron. Programming-in-the-large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, pages 114–121, New York, NY, USA, 1975. ACM. doi: http://doi.acm.org/10.1145/800027.808431. 4

Simon Frankau and Alan Mycroft. Stream processing hardware from functional language specifications. In *36th Hawaii International Conference on System Sciences*, Hawaii, January 2003. URL `https://www.publications.cl.cam.ac.uk/246/01/hicss_2003-1.pdf`. 1

Martin Grabmüller. Algorithm W step by step. Draft Paper, 2006. URL `http://uebb.cs.tu-berlin.de/~magr/pub/AlgorithmW.pdf`. 3.1.2

Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969. 3.1.2

Simon Peyton Jones. *The Haskell 98 Report*, 1998. URL `http://haskell.org/definition/haskell98-report.pdf`. 3.1.2

John Kessenich. *The OpenGL® Shading Language*. 3DLabs, Inc. Ltd., 1.20-8 edition, September 2006. URL `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`. 1, 2.2, 2.2.2, 2.2.2, 3.1.2

Benj Lipchak. *ARB_fragment_program*. Silicon Graphics, Inc., 26 edition, 2002–2003. URL `http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt`. 1, 2.3.3

William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *SIGGRAPH*. The University of Texas at Austin and NVIDIA Corporation, 2003. URL `http://www-csl.csres.utexas.edu/users/billmark/papers/Cg/cgpaper.pdf`. 1, 2.2, 2.3.4

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, April 1978. URL `http://diku.dk/undervisning/2006-2007/2006-2007_b2_246/milner78theory.pdf`. 3.1.2

NVIDIA Corp. *NVIDIA GeForce 8800 GPU Architecture Overview*. NVIDIA Corp., 2006. URL `http://www.nvidia.com/object/IO_37100.html`. 2.2.2

Craig Peeper and Jason L. Mitchell. *Introduction to the DirectX® 9 High Level Shading Language*. Microsoft Corporation and ATI Research, 2003. URL `http://ati.amd.com/developer/ShaderX2_IntroductionToHLSL.pdf`. 1, 2.2

Robert J. Simpson and John Kessenich. *The OpenGL® ES Shading Language*. The Khronos Group, 1.00-14 edition, March 2007. URL `http://www.khronos.org/files/opengles_shading_language.pdf`. 2.2.2

Joel Spolsky. *Joel on Software*. Apress, Berkeley, CA, 2004. ISBN 1-59059-389-8. 2.3.2

Dwight VandenBerghe. Why ML/OCaml are good for writing compilers. comp.compilers, July 1998. URL `http://compilers.iecc.com/comparch/article/98-07-220`. 5

Appendix A

Appendices

A.1 Code Extracts

A.1.1 Abstract Syntax Representation

```
data Expr
  = ExprUnitLiteral
  | ExprRealLiteral !Double
  | ExprBoolLiteral !Bool
  | ExprVar !String
  | ExprApp !Expr !Expr
  | ExprArray ![Expr]
  | ExprTuple ![Expr]
  | ExprIf !Expr !Expr !Expr
  | ExprLet !Patt !Expr !Expr
  | ExprLambda !Patt !Expr

  deriving (Show, Eq)
```

A.1.2 Pattern Grammar

```
--
-- Patterns
--

tuple_patt_inner :: { [Patt] }
  : patt COMMA patt { $3:$1:[] }
  | tuple_patt_inner COMMA patt { $3:$1 }
  ;

tuple_patt :: { Patt }
  : LPAREN tuple_patt_inner RPAREN opt_type { PattTuple (reverse $2) $4 }
  ;

array_patt_inner :: { [Patt] }
```

```
  : patt { $1:[] }
  | array_patt_inner COMMA patt { $3:$1 }
  ;

array_patt :: { Patt }
  : LBRACKET array_patt_inner RBRACKET opt_type { PattArray (reverse $2) $4 }
  ;

patt :: { Patt }
  : WILDCARD opt_type { PattWild $2 }
  | LPAREN RPAREN opt_type { PattUnit $3 }
  | IDENTIFIER opt_type { PattVar $1 $2 }
  | tuple_patt { $1 }
  | array_patt { $1 }
  | LPAREN patt RPAREN { $2 }
  ;

patts :: { [Patt] }
  : patt { $1:[] }
  | patts patt { $2:$1 }
  ;
```

A.1.3 Substitution Composition

```
-- Substitution composition, finding s3 t = s2(s1(t))
-- s3 contains:
-- a) all the bindings in s1,
--    with s2 applied to their right hand sides
-- b) all the bindings in s2,
--    except if they clash with a)
--    in which case a) takes precendence
-- Note that Map.union prefers its first argument
-- when duplicate keys are encountered.
composeSubst :: Subst -> Subst -> Subst
composeSubst (tsub2, dsub2) (tsub1, dsub1) = (
  (Map.map (applySubstType (tsub2, dsub2)) tsub1) `Map.union` tsub2,
  (Map.map (applyDimVarSubst dsub2) dsub1) `Map.union` dsub2
  )
```

A.1.4 Principal Typing

```
-- Type inference! The cool stuff.
-- Returns the principal type of the expression, and the substitution that must
-- be applied to gamma to achieve this.
principalType :: SchemeEnv -> Expr -> TI (Subst, Type)

principalType _ (ExprUnitLiteral) = return (nullSubst, TypeUnit)

principalType _ (ExprRealLiteral _) = return (nullSubst, TypeReal)

principalType _ (ExprBoolLiteral _) = return (nullSubst, TypeBool)

principalType gamma a@(ExprVar ident) = registerStackPoint a $
  case Map.lookup ident gamma of
    Just sigma -> do
      t <- instantiate sigma
```

```
      return (nullSubst, t)
    Nothing -> throwError $ TypeError [] $ TypeErrorUnboundVariable $ ident

principalType gamma a@(ExprApp e1 e2) = registerStackPoint a $ do
  (s1, t1) <- principalType gamma e1
  let s1gamma = applySubstSchemeEnv s1 gamma
  (s2, t2) <- principalType s1gamma e2
  alpha <- freshTypeVar
  s3 <- mgu (applySubstType s2 t1) (TypeFun t2 alpha)
  return ((s3 `composeSubst` (s2 `composeSubst` s1)), applySubstType s3 alpha)

principalType gamma a@(ExprArray es) = registerStackPoint a $ do
  alpha <- freshTypeVar -- the type of elements of the array
  (s1, _) <- Foldable.foldrM (
    \ e (s1, s1gamma) -> do
      (s2, t2) <- principalType s1gamma e
      s3 <- mgu t2 (applySubstType s1 alpha)
      return (s3 `composeSubst` (s2 `composeSubst` s1), applySubstSchemeEnv s3 (applySubstSchemeEnv s2 s1gamma))
    ) (nullSubst, gamma) es
  return (s1, TypeArray (applySubstType s1 alpha) (DimFix $ toInteger $ length es))

principalType gamma a@(ExprTuple es) = registerStackPoint a $ do
  (s1, _, ts1) <- Foldable.foldrM (
    \ e (s1, s1gamma, ts1) -> do
      (s2, t2) <- principalType s1gamma e
      return (s2 `composeSubst` s1, applySubstSchemeEnv s2 s1gamma, t2:ts1)
    ) (nullSubst, gamma, []) es -- the empty tuple is invalid, but es has valid length
  return (s1, TypeTuple ts1)

principalType gamma a@(ExprIf e1 e2 e3) = registerStackPoint a $ do
  (s1, t1) <- principalType gamma e1
  s2 <- mgu t1 TypeBool
  let s2s1gamma = applySubstSchemeEnv s2 (applySubstSchemeEnv s1 gamma)
  (s3, t3) <- principalType s2s1gamma e2
  let s3s2s1gamma = applySubstSchemeEnv s3 s2s1gamma
  (s4, t4) <- principalType s3s2s1gamma e3
  s5 <- mgu (applySubstType s4 t3) t4
  return (s5 `composeSubst` (s4 `composeSubst` (s3 `composeSubst` (s2 `composeSubst` s1))), applySubstType s5 t4)

principalType gamma a@(ExprLet p e1 e2) = registerStackPoint a $ do
  (s2s1, _, gamma') <- principalTypeBinding gamma p e1
  (s3, t3) <- principalType gamma' e2
  return (s3 `composeSubst` s2s1, t3)

principalType gamma a@(ExprLambda p e) = registerStackPoint a $ do
  (t1, m) <- inferPattType p
  -- generalize over nothing when converting to type schemes
  let params = Map.map (Scheme emptyVarRefs) m
  -- it is critical that Map.union prefers its first argument
  (s, t2) <- principalType (params `Map.union` gamma) e
  return (s, TypeFun (applySubstType s t1) t2)


-- Binds p to e, returning the substitution required and the new environment produced.
-- Helper function used both by normal let expressions and by debug let commands.
principalTypeBinding :: SchemeEnv -> Patt -> Expr -> TI (Subst, Type, SchemeEnv)
principalTypeBinding gamma p e = do
  (s1, t1) <- principalType gamma e
  (t1', bindings) <- inferPattType p
  -- we should remove any mapping for the identifiers in p, because we're going to shadow them,
  -- and we don't want existing defs to stop us from generalizing their free variables
  let gamma_shadowed = Map.differenceWithKey (\_ _ _ -> Nothing) gamma bindings
```

```
  let s1gamma_shadowed = applySubstSchemeEnv s1 gamma_shadowed
  s2 <- mgu t1 t1'
  let s2s1gamma_shadowed = applySubstSchemeEnv s2 s1gamma_shadowed
  let generalize t = Scheme (fvType t `differenceVarRefs` fvSchemeEnv s2s1gamma_shadowed) t
  let s2bindingspoly = Map.map (generalize . applySubstType s2) bindings
  let gamma' = s2s1gamma_shadowed `Map.union` s2bindingspoly
  return (s2 `composeSubst` s1, applySubstType s2 t1, gamma')
```

A.1.5 Interpreter

```
-- The inner interpreter function.
-- Returns the value, or on error, a message explaining the error and the expression that triggered it.
interpretExpr :: ValueEnv -> Expr -> InterpretM Value

interpretExpr _ (ExprUnitLiteral) = return ValueUnit

interpretExpr _ (ExprRealLiteral b) = do
  n <- freshNode
  return $ ValueReal $ DFRealLiteral n b

interpretExpr _ (ExprBoolLiteral b) = do
  n <- freshNode
  return $ ValueBool $ DFBoolLiteral n b

interpretExpr env (ExprVar ident) = do
  case Map.lookup ident env of
    Just v -> v
    Nothing -> error $ "variable <" ++ ident ++ "> undefined, should have been caught by type checker!"

interpretExpr env a@(ExprApp e1 e2) = registerStackPoint a $ do
  ValueFun f <- interpretExpr env e1
  v <- interpretExpr env e2
  f v

interpretExpr env a@(ExprArray es) = registerStackPoint a $ do
  vs <- mapM (interpretExpr env) es
  return $ ValueArray vs

interpretExpr env a@(ExprTuple es) = registerStackPoint a $ do
  vs <- mapM (interpretExpr env) es
  return $ ValueTuple vs

interpretExpr env a@(ExprIf eb e1 e2) = registerStackPoint a $ do
  ValueBool dfb <- interpretExpr env eb
  v1 <- interpretExpr env e1
  v2 <- interpretExpr env e2
  conditionalize dfb v1 v2

interpretExpr env a@(ExprLet p e1 e2) = registerStackPoint a $ do
  (_, env') <- interpretBinding env p e1
  v2 <- interpretExpr env' e2
  return v2

interpretExpr env a@(ExprLambda p e) = registerStackPoint a $ do
  -- it is critical that Map.union prefers its first argument
  return $ ValueFun (\v -> let env' = matchPattern p v `Map.union` env in interpretExpr env' e)


-- Binds p to e, returning the new environment produced.
```

```
-- Helper function used both by normal let expressions and by debug let commands.
interpretBinding :: ValueEnv -> Patt -> Expr -> InterpretM (Value, ValueEnv)
interpretBinding env p e =
  registerStackPoint e $ do
  v <- interpretExpr env e
  -- it is critical that Map.union prefers its first argument
  let env' = matchPattern p v `Map.union` env
  return (v, env')
```

A.1.6 User Errors

```
data LexerError
  = LexerErrorNoLex !Char
  | LexerErrorIdentBeginsUpper !String

  deriving Show

data ParserError
  = ParserErrorNoParse !Token
  | ParserErrorBadFixedDim !Integer

  deriving Show

data TypeError
  = TypeErrorOccursCheck !Type !Type
  | TypeErrorCouldNotUnify !Type !Type
  | TypeErrorUnboundVariable !String
  | TypeErrorDuplicateIdentsInPattern !Patt

  deriving Show

data ShaderError
  = ShaderErrorBadShaderType !Type
  | ShaderErrorBadUniformType !Type
  | ShaderErrorBadTextureType !Type
  | ShaderErrorBadVaryingType !Type
  | ShaderErrorBadOutputType !Type
  | ShaderErrorCouldNotLink !Type !Type

  deriving Show

data InterpreterError
  = InterpreterErrorIndexOutOfBounds !Int
  | InterpreterErrorDynamicIterate
  | InterpreterErrorDynamicIndex
  | InterpreterErrorFunctionEquality

  deriving Show

data TargetError
  = TargetErrorGLSLDynamicTextureSelection

  deriving Show
```

A.1.7 Funslang C API

```
FS_API void fsInit(int* argc, char*** argv);
FS_API void fsExit(void);

// Funslang compilation.
// Requires p->vertex_shader_path and p->fragment_shader_path to be set.
// The rest of p can be uninitialized.
FS_API FS_BOOL fsCompile(FSprogram* p);

// Shader data initialization.
// You _must_ ensure that p is active by calling glUseProgram(p->glsl_program) first.
FS_API void fsSetVertexUniforms(FSprogram* p, const GLfloat* data);
FS_API void fsSetFragmentUniforms(FSprogram* p, const GLfloat* data);
FS_API void fsSetVertexVaryings(FSprogram* p, const GLfloat* data);

// Shader texture image unit assignment.
// You _must_ ensure that p is active by calling glUseProgram(p->glsl_program) first.
FS_API void fsSetTextureImageUnits(FSprogram* p);

// Loads texture from file and assigns it new name (using glGenTextures).
// Creates mipmap and enables trilinear filtering.
// Currently only supports JPG.
// Returns name, guaranteeing that the new texture is currently bound.
// On error, returns 0 (note 0 is not a valid GL texture name).
FS_API GLuint fsLoadTexture2D(const char* fn, unsigned int* width, unsigned int* height);

// Loads JPG pixel data from file to byte array in RGB order.
// Uses the given function to allocate memory.
FS_API unsigned char* fsLoadJPG(void* (*alloc_pixel_buffer)(size_t),
  const char* fn, unsigned int* width, unsigned int* height);
```

A.2 Compiler-Generated Code

A.2.1 Autostereogram Fragment Shader

```
uniform float FragmentUniforms[5];
uniform sampler2D Tex1;
uniform sampler2D Tex0;
varying vec2 FragmentVaryings0;

void main()
{
//sampler2D t6, t5;
float t33, t32, t31, t30, t27, t26, t24, t23, t22, t21, t20, t19, t18, t17,
  t16, t15, t14, t10, t8, t7, t4, t3, t2, t1, t0;
vec4 t29, t9;

// node 28 is not required
// node 25 is not required
t17 = 1.0;
t16 = 1.0;
t15 = 1.0;
// node 13 is not required
// node 12 is not required
// node 11 is not required
```

```
t8 = FragmentVaryings0[1];
t7 = FragmentVaryings0[0];
// noting texture: DFTex (DFTexConstant 6 2D 1)
// noting texture: DFTex (DFTexConstant 5 2D 0)
t9 = texture2D(Tex0, vec2(t7, t8));
t10 = t9.r;
t4 = FragmentUniforms[4];
t3 = FragmentUniforms[3];
t2 = FragmentUniforms[2];
t14 = t2 / t3;
t18 = t17 - t14;
t19 = t18 * t10;
t20 = t16 - t19;
t21 = t14 / t20;
t22 = t15 - t21;
t23 = t22 * t4;
t1 = FragmentUniforms[1];
t24 = t1 - t23;
t0 = FragmentUniforms[0];
t26 = t24 / t0;
t27 = t7 - t26;
t29 = texture2D(Tex1, vec2(t27, t8));
t33 = t29.a;
t32 = t29.b;
t31 = t29.g;
t30 = t29.r;

gl_FragColor = vec4(t30, t31, t32, t33);
}
```

A.3 Funslang Library Reference

A.3.1 Base Functions

```
negate :: Real -> Real
not :: Bool -> Bool
! :: a m -> Real -> a
+ :: Real -> Real -> Real
- :: Real -> Real -> Real
* :: Real -> Real -> Real
/ :: Real -> Real -> Real
< :: Real -> Real -> Bool
<= :: Real -> Real -> Bool
> :: Real -> Real -> Bool
>= :: Real -> Real -> Bool
== :: a -> a -> Bool
/= :: a -> a -> Bool
&& :: Bool -> Bool -> Bool
|| :: Bool -> Bool -> Bool
tx :: a m n -> a n m
map :: (a -> b) -> a m -> b m
foldl :: (a -> a -> b) -> a -> b m -> a
foldl1 :: (a -> a -> a) -> a m -> a
foldr :: (a -> b -> b) -> b -> a m -> b
foldr1 :: (a -> a -> a) -> a m -> a
iterate :: (a -> (Bool, a)) -> Real -> a -> a
```

```
zipWith :: (a -> b -> c) -> a m -> b m -> c m
zipWith3 :: (a -> b -> c -> d) -> a m -> b m -> c m -> d m
sin :: Real -> Real
cos :: Real -> Real
tan :: Real -> Real
asin :: Real -> Real
acos :: Real -> Real
atan :: Real -> Real -> Real
pow :: Real -> Real -> Real
exp :: Real -> Real
exp2 :: Real -> Real
log :: Real -> Real
log2 :: Real -> Real
rsqrt :: Real -> Real
abs :: Real -> Real
floor :: Real -> Real
ceiling :: Real -> Real
round :: Real -> Real
truncate :: Real -> Real
fract :: Real -> Real
min :: Real -> Real -> Real
max :: Real -> Real -> Real
sample1D :: Tex 1D -> Real 1 -> Real 4
sample2D :: Tex 2D -> Real 2 -> Real 4
sample3D :: Tex 3D -> Real 3 -> Real 4
sampleCube :: Tex Cube -> Real 3 -> Real 4
```

A.3.2 Derived Functions

```
$ :: (a -> b) -> a -> b
. :: (a -> b) -> (c -> a) -> c -> b
negates :: Real m -> Real m
!! :: a m -> Real n -> a n
++ :: Real m -> Real m -> Real m
-- :: Real m -> Real m -> Real m
** :: Real m -> Real m -> Real m
// :: Real m -> Real m -> Real m
**. :: Real m -> Real -> Real m
//. :: Real m -> Real -> Real m
pi :: Real
sum :: Real m -> Real
product :: Real m -> Real
any :: Bool m -> Bool
all :: Bool m -> Bool
sqrt :: Real -> Real
mod :: Real -> Real -> Real
dot :: Real m -> Real m -> Real
cross :: Real 3 -> Real 3 -> Real 3
length :: Real m -> Real
normalize :: Real m -> Real m
#. :: Real m n -> Real m -> Real n
.# :: Real m -> Real n m -> Real n
# :: Real m n -> Real o m -> Real o n
clamp :: Real -> Real -> Real -> Real
step :: Real -> Real -> Real
mix :: Real -> Real -> Real -> Real
smoothstep :: Real -> Real -> Real -> Real
faceforward :: Real m -> Real n -> Real n -> Real m
reflect :: Real m -> Real m -> Real m
```

```
refract :: Real m -> Real m -> Real -> Real m
pad :: Real 3 -> Real 4
strip :: a 4 -> a 3
```

A.4 Third Party Shaders

A.4.1 3Dlabs

3Dlabs-License.txt

```
/**************************************************************************
*                                                                         *
*               Copyright (C) 2002-2006  3Dlabs Inc. Ltd.                 *
*                                                                         *
*                        All rights reserved.                             *
*                                                                         *
* Redistribution and use in source and binary forms, with or without      *
* modification, are permitted provided that the following conditions      *
* are met:                                                                *
*                                                                         *
*     Redistributions of source code must retain the above copyright      *
*     notice, this list of conditions and the following disclaimer.       *
*                                                                         *
*     Redistributions in binary form must reproduce the above             *
*     copyright notice, this list of conditions and the following         *
*     disclaimer in the documentation and/or other materials provided     *
*     with the distribution.                                              *
*                                                                         *
*     Neither the name of 3Dlabs Inc. Ltd. nor the names of its           *
*     contributors may be used to endorse or promote products derived     *
*     from this software without specific prior written permission.       *
*                                                                         *
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS     *
* "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT       *
* LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS       *
* FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE          *
* COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,   *
* INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,    *
* BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;        *
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER        *
* CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT      *
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN       *
* ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE         *
* POSSIBILITY OF SUCH DAMAGE.                                             *
*                                                                         *
**************************************************************************/
```

Mandel.frag

```
//
// Fragment shader for drawing the Mandelbrot set
//
```

```
// Authors: Dave Baldwin, Steve Koren, Randi Rost
//          based on a shader by Michael Rivero
//
// Copyright (c) 2002-2006 3Dlabs Inc. Ltd.
//
// See 3Dlabs-License.txt for license information
//

varying vec3  Position;
varying float LightIntensity;

uniform float MaxIterations;
uniform float Zoom;
uniform float Xcenter;
uniform float Ycenter;
uniform vec3  InnerColor;
uniform vec3  OuterColor1;
uniform vec3  OuterColor2;

void main(void)
{
    float   real  = Position.x * Zoom + Xcenter;
    float   imag  = Position.y * Zoom + Ycenter;
    float   Creal = real;   // Change this line...
    float   Cimag = imag;   // ...and this one to get a Julia set

    float r2 = 0.0;
    float iter;

    for (iter = 0.0; iter < MaxIterations && r2 < 4.0; ++iter)
    {
        float tempreal = real;

        real = (tempreal * tempreal) - (imag * imag) + Creal;
        imag = 2.0 * tempreal * imag + Cimag;
        r2   = (real * real) + (imag * imag);
    }

    // Base the color on the number of iterations

    vec3 color;

    if (r2 < 4.0)
        color = InnerColor;
    else
        color = mix(OuterColor1, OuterColor2, fract(iter * 0.05));

    color *= LightIntensity;

    gl_FragColor = vec4 (clamp(color, 0.0, 1.0), 1.0);
    //gl_FragColor = vec4 (color, 1.0);
}
```

Ben Challenor
St John's College
bdc25

Computer Science Part II Project Proposal

A lightweight shading language for graphics processors

October 17 2007

Project Originator: *Ben Challenor*

Resources Required: See Project Resource Form

Project Supervisor: *Christian Steinruecken*

Signature:

Director of Studies: *Robert Mullins*

Signature:

Overseers: *Neil Dodgson* and *Timothy Griffin*

Signatures:

1 Introduction and Description of the Work

Graphical rendering, from vertex manipulation to texture application, is numerically intensive. Even modern CPUs are poorly suited to performing such calculations in real time. Often, however, operations on individual vertices or pixels are independent, and so the problem is highly parallelizable.

Traditionally, specialized hardware has handled the rendering process. But this has been restrictive: effects have been limited to so-called *fixed functionality*, with standard lighting, texture application and fog.

The trend in recent years, however, has been towards general-purpose graphics processors. These adopt the stream processing, or 'gather, operate and scatter' model [Gummaraju and Rosenblum, 2005]. Input, for example, of vertices and their associated data, is *gathered* into streams. These streams are then *operated* on by programmable functions called *shaders*. The output stream is then *scattered* back to where it is needed—in this case, the framebuffer.

Shaders are 'compute heavy', but small. They require little memory beyond the input stream and registers, allowing chip area to be dedicated to arithmetic units instead of the caching hierarchy found on a CPU [Venkatasubramanian, 2003]. Furthermore, they tend to require little or no control flow, so many stream objects can be processed in parallel by the same execution unit.

By opening up the graphics hardware to application programmers, many effects are now possible that would have been infeasible on the CPU, from soft-shadowing to bump mapping, from refraction to complex particle systems. My current graphics card is even (apparently) good enough to dynamically generate an animated Mandelbrot fractal!

The first languages available for shader programming were abstract assembly languages, with separate standards for the different exposed parts of the 3D pipeline [Brown, 2002–2004, Lipchak, 2002–2003]. High-level shading languages, such as GLSL [Kessenich, 2006], Cg [Mark et al., 2003] and HLSL [Peeper and Mitchell, 2003] unified the two pipeline stages and introduced C-like syntaxes. An example is given in Figure 1.

Stream processing is now moving to mobile devices, and a cut-down version of GLSL—GLSL ES—has been specified [Simpson and Kessenich, 2007]. Unfortunately, its main advantage appears to be familiarity; it retains many legacies of its C ancestry.

One of the key features of all the shading languages mentioned is run-time compilation, allowing dynamic manipulation of shader source code. However, the current high level shader languages are complex; for example, they all retain the C preprocessor. Writing a compiler—which hardware vendors

```
attribute vec4 vertex;
uniform mat4 modelViewProjectionMatrix;

void main()
{
    gl_Position = modelViewProjectionMatrix * vertex;
}
```

Figure 1: A GLSL vertex shader to perform a simple projection.

must do when providing a graphics library implementation—is unnecessarily difficult.

I propose that a lightweight language, designed formally from the ground up, could be easier to reason about, and better suited to compilation on mobile devices.

Searching the literature leads me to believe that nothing like this has been done before. While a functional language has been designed for generic stream processors [Frankau and Mycroft, 2003], it compiles directly to hardware, supports generic stream manipulation, and is not optimized for graphics.

2 Resources Required

- My personal computer, for prototyping, with OpenGL 2.0-compatible graphics card
- OpenGL and associated libraries

3 Starting Point

I wrote a compiler for GLSL ES in a summer placement, which explains my desire for a more appropriate language. I have a good grounding in imperative languages, such as C and Java, but if I decide to implement the parser in a functional language then this will require some mental adjustment. I also have no background in language design, so this project should be a learning experience!

4 Substance and Structure of the Project

4.1 Core

- Research into language design issues, and aquisition of background knowledge.
- Decisions on what features the new language should support—and equally, what features it should not. This will involve:
  - studying the languages (Cg, HLSL) that are unfamiliar to me; and
  - identifying common shader uses.
- Specification of language.
- Implementation of parser to generate abstract syntax tree (AST), including semantic checks. This will require the choice of a suitable programming language, and possibly some learning curve.
- Implementation of test rig to ensure semantic checks are respected.
- Hand-translation of existing shaders into new language for comparison.
- Implementation of compiler back end. This is a proof-of-concept, so the likely target will be my desktop graphics card.
- Writing the Dissertation.

4.2 Possible Extensions

- Preparation of live graphical demonstration with shaders written in my own language
- . . . and with those shaders compiled at run time.

5 Limitations

It should not be considered a part of this project that the new compiler be demonstrably more efficient in compilation than a compiler for an existing shading language. Firstly, it may not be possible to access an existing compiler in such a way as to be able to measure its performance; and secondly, this is a proof-of-concept, and should not be expected to be production quality.

I also do not plan the new language to be 'better' than the existing languages in its expressive power. In fact, it may well end up with fewer features, or being more restrictive. My intention is that the new language will be simpler to write a compiler for, free of legacy features, and easier to reason about.

6 Success Criteria

My main objective is the design of a new shading language and the implementation of a compiler. The language must at least allow standard vertex transformations and texture application.

By the end of this project I should be able to write shaders in this language and compile them to some format that can be used on a real graphics processor. Though this may require some interaction (such as manual copying of intermediate files) I should be able to give example shaders and screenshots that prove that my shader compiler works.

7 Timetable and Milestones

7.1 Work Packet 1: Oct 26 – Nov 9

Acquisition of background knowledge. Researching existing languages. Gathering ideas about language features.

7.2 Work Packet 2: Nov 9 – Nov 23

Making decisions about language features. Milestone: Fragments of example code in new language.

7.3 Work Packet 3: Nov 23 – Dec 7

Choice of implementation language for compiler. Starting to implement front end. Milestone: Lexer, parser and pretty printer.

7.4 Work Packet 4: Jan 11 – Jan 25

Implementing semantic checks and testing compiler front end. Writing progress report. Milestone: Progress report ready for submission.

7.5 Work Packet 5: Jan 25 – Feb 8

Decisions about compiler target. Starting to implement back end. Milestone: Back end tests on some simple shaders.

7.6 Work Packet 6: Feb 8 – Feb 22

Further work on back end. Fixing bugs. Gathering ideas for dissertation. Possibly working on project extensions. Milestone: Compiler works for range of shaders with some tangible results (e.g. screenshots).

7.7 Work Packet 7: Feb 22 – Mar 7

Starting to write first draft of dissertation. Further bug fixes.

7.8 Work Packet 8: Mar 7 – Mar 21

Milestone: First draft of dissertation basically ready.

7.9 Work Packet 9–10: Apr 18 – May 2 – May 16

Writing dissertation final draft. Milestone: Ready for submission on May 16.

8 References

P Brown. *ARB_vertex_program*. Silicon Graphics, Inc., 45 edition, 2002–2004. URL `http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt`. 1

S G Frankau and A Mycroft. Stream processing hardware from functional language specifications. In *36th Hawaii International Conference on System Sciences*, Hawaii, January 2003. URL `https://www.publications.cl.cam.ac.uk/246/01/hicss_2003-1.pdf`. 1

J Gummaraju and M Rosenblum. Stream programming on general-purpose processors. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 343–354, 2005. URL `http://merrimac.stanford.edu/publications/micro38_streamingGPP.pdf`. 1

J Kessenich. *The OpenGL®Shading Language*. 3DLabs, Inc. Ltd., 1.20-8 edition, September 2006. URL `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`. 1

B Lipchak. *ARB_fragment_program*. Silicon Graphics, Inc., 26 edition, 2002–2003. URL `http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt`. 1

W R Mark, R S Glanville, K Akeley, and M J Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *SIGGRAPH*. The University of Texas at Austin and NVIDIA Corporation, 2003. URL `http://www-csl.csres.utexas.edu/users/billmark/papers/Cg/cgpaper.pdf`. 1

C Peeper and J L Mitchell. *Introduction to the DirectX®9 High Level Shading Language*. Microsoft Corporation and ATI Research, 2003. URL `http://ati.amd.com/developer/ShaderX2_IntroductionToHLSL.pdf`. 1

R J Simpson and J Kessenich. *The OpenGL® ES Shading Language*. The Khronos Group, 1.00-14 edition, March 2007. URL `http://www.khronos.org/files/opengles_shading_language.pdf`. 1

S Venkatasubramanian. The graphics card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003. URL `http://www.research.att.com/~suresh/papers/mpds/mpds.pdf`. 1