School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

# Functional Rendering Engine in Haskell

**Naval Bhandari**

**Submitted in accordance with the requirements for the degree of BSc (Ind) Computer Science**

2014/2015

The candidate confirms that the following have been submitted.

| Items | Format | Recipient(s) and Date |
|---|---|---|
| Deliverable 1 | Report | SSO (13/05/15) |
| Deliverable 2 | Source code repository: `https://bitbucket.org/Gentatsu/functional-rendering-engine-in-haskell-freed/overview` | Supervisor, Assessor (13/05/15) |
| Deliverable 3 | Library source code: `http://onibaku.co.uk/FreeD.zip` | Supervisor, Assessor (13/05/15) |
| Deliverable 4 | Library documentation: `http://onibaku.co.uk/freed/` | Supervisor, Assessor (13/05/15) |

Type of project: **Exploratory Software**

# Summary

This project is intended to analyse the current state of rendering engines available for open source use in Haskell, and to deliver a new one which is based upon that analysis.

## Acknowledgements

I would like to thank my supervisor, David Duke, for all of his support and guidance throughout the project, and for giving me several books as reference and reading material. I would also like to thank my assessor, Anthony Cohn, for providing valuable feedback during my mid-project presentation. I am also grateful for one of my fellow students in the School of Computing, Frasier Murray, for creating a Haskell library for me to use. I would also like to thank my girlfriend for her continued support throughout my degree.

# Contents

# 1 Introduction

This section sets out the aims, objectives and minimum requirements for this project. It lists deliverables, and briefly mentions further enhancements; the things that there is an interest to do, but, due to the limited time constraint, will or may not be possible in the current project scope.

## 1.1 FreeD

The name of the library is called *FreeD*, a wordplay on 3D, where the initial *fre* is an acronym for *Functional Rendering Engine*.

## 1.2 Introductory References

Here, I introduce some terminology to add context in the following section. They are further explained throughout the report.

- **Haskell** - A functional, high-level programming language.
- **Hackage** [16] - A community-driven website archive of open-sourced Haskell libraries. Many of the libraries that are used in this project can be found here, and it is intended for FreeD to be added to this repository.
- **OpenGL** [28] - De facto standard 2D/3D graphics rendering API written in C.
- **HOpenGL** [30] - Mid-level bindings in Haskell to OpenGL functions and calls.
- **The Pipeline** - A sequence of steps to create a 2D raster representation of a scene is referred to as a graphics pipeline. In OpenGL, this is simply known as *the pipeline*.
- **Vertex Buffer Object** - An OpenGL data type which provides a way of uploading vertex data (position, normals, colour, etc) to the video device [39].
- **Shaders** - A program that determines how a geometric, 2D primitive is rendered.
- **GLSL** - OpenGL Shading Language - A language for writing shaders in OpenGL.
- **eDSL** - Embedded Domain-Specific Language - A small language written for a specific application domain.

## 1.3 Aims

The main aim of this project is to deliver a library which can be used to render 3D graphics and use shader programming in a functional, high-level way. There are a number of libraries that have similar capabilities, however, many of them are experimental and/or no longer maintained. Several of these libraries have common functionality, and are compared in Figure 2.12; these will be discussed further on.

OpenGL is currently the industry standard for rendering 2D/3D graphics, and it will be built upon for the purpose of this library. OpenGL has two ways of rendering: immediate mode rendering and shaders. Immediate mode relies on the top-layered OpenGL state machinery, accessed and modified by a large collection of specialised functions, which are further explained in section 2.5.

Shaders are a way of programming graphics, which are based on general purpose programming on the graphics processing unit (GPU). One of the aims is to incorporate the use of shaders in FreeD. Shaders are becoming the dominant way of rendering graphics, as rendering in immediate mode is now becoming deprecated, and may be removed in the future [7] [15]. This is because hardware has had to become more specialised in order to support the functions introduced by updates in OpenGL. Instead of hardware being created to support additional functions, each time they're added, a general-purpose, programmable piece of hardware has been made. This enables the programmer to implement their own functions, rather than using the fixed ones found in older versions of OpenGL, giving them more flexibility. Many of the existing Haskell graphics libraries currently use immediate mode, and there is only very basic shader programming support at the moment. This means that, to be useful, shaders should be the fundamental building blocks of FreeD. This presents itself as its own challenge, as the library needs to keep track of several aspects of the state, which have been deprecated using the fixed-function pipeline (FFP).

Although efficiency is not one of the main priorities of this project, is it important that the data structures are implemented in such a way that allows efficient rendering. As this is a graphics project, there are many applications requiring fast rendering, such as games. Although a compromise can be made in order to abstract the code, we need to be mindful of not losing speed in this process.

## 1.4 Objectives

My objectives are to conduct the following:

- An analysis of current libraries; an in depth breakdown of critical aspects of each of them.
- A comparative study of the techniques used in current libraries. Linked with the previous objective, this will be presented via a table of what the libraries do, and whether they tick the boxes of what is to be accomplished.
- A package or set of packages containing Haskell modules which can be used for the purposes of the project. This can be measured by the evaluation, and the functionality it provides. This means that I shall attempt to abstract as much of the Haskell OpenGL bindings into my library as possible to see if it provides the same result.

## 1.5 Deliverables

I aim to deliver the following:

- A requirement specification (Individual functions to be implemented).
- A qualitative evaluation study of my library.

## 1.6 Minimum Requirements

My minimum requirements are as follows:

- To conduct a study on the current state of 3D functional rendering engines in Haskell.
- To implement a scene graph (explained in section 2.1) API through use of Haskell's data types, so that it can capture all of the FFP's capabilities.

## 1.7 Further enhancements

I aim to build upon the minimum requirements, and enhance the library by:

- Implementing a data structure for geometry, which can be defined as a vertex buffer object (VBO) for rendering, explained in section 2.5.2.
- Making use of the GPU to render instead of the FFP (i.e. shader programming), and using the same scene graph API.

## 1.8 Evaluation Criteria

FreeD will be tested in 3 main ways: Frames (rendered) per second (FPS), which is a common way of approximating the speed at which a scene is being rendered; lines of code (LOC) for each way tested (independent of window calls); Visually checking whether the results are identical when rendered under the same pretense of geometry, lighting, camera, etc. The latter method can be tested by taking a screenshot after one render pass, and then analysing it using a visualisation tool to check the pixel values against each other. I would like to comparatively test each of these:

- FFP calls in C++.
- Shader Programming calls in C++.
- Direct OpenGL bindings for FFP calls in Haskell.
- Direct OpenGL shader bindings for shader programming calls in Haskell.
- FreeD library API calls using the FFP in Haskell.
- FreeD library API calls using shader programming in Haskell.

Another area of evaluation, is the cost in speed and memory of handling geometry, measured in terms of the number of triangles rendered in the scene. The constraint of constructing the object can be abstracted once the mesh has been loaded and tested when it comes to constructing the scene graph with the loaded mesh and the rendering of it. This is coupled with the evaluation of stress-testing the abstraction of the library. As mentioned earlier, there needs to be a balance between speed and the abstraction functionality of the library. This will be tested by drawing as many objects as possible with as little stress placed on the GPU, so that we can test the CPU bounds and abstraction purely.

## 1.9 Relevance to Degree

This project is focused around rendering, which is encapsulated by my Computer Graphics module. This module gave me the core knowledge to understand what happens throughout each part of the graphics pipeline as well as geometric data structures, in terms of how they are represented, as well as used efficiently. Software Systems Engineering, and my year in industry also helped me managed my time for a big project, and helped me become comfortable using version control for both my source code and project report.

## 1.10 Initial Schedule

The initial schedule was created at the start of my project when defining the aims and objectives, as shown in Figure B.1 in appendix B. Each stage is intentionally vague as, ultimately, several of the stages were tied in with each other. This is especially true for the implementation and evaluation. Changes in the implementation meant that some original evaluations were invalid, and needed to be performed again, or in a different way.

# 2 Background Research

## 2.1 Scene Graphs

Scene graphs are a type of tree data structure, and are a way of representing a graphical model. They are constructed from types of node representing either the settings of a mesh, or a mesh itself. A mesh is simply the geometrical data of an object. They are referred to as graphs, and not trees, as parent nodes can share a child node.

### 2.1.1 The Inventor Mentor

*Open Inventor* [50] is a C++ library for rendering scene graphs, developed originally by Silicon Graphics [38]. It introduced the concept of scene graphs, and is implemented using an imperative, object-oriented style of programming. *The Inventor Mentor* [50] is a book which goes through the details of Open Inventor and scene graphs. I had used this book to learn how scene graphs are handled using an object-oriented programming (OOP) approach. Although I have not taken anything explicitly from the book, as I had sufficient information from other sources, it has affirmed that scene graphs are a sensible data structure for representing a graphical scene.

### 2.1.2 State

Rendering, ala OpenGL, is stateful. Nearly all OpenGL commands set the global state. Some examples of these are `glColor`, `glMatrixMode`, and `glLoadIdentity`, which modify the current colour and matrix, and set the current matrix as the identity matrix, respectively. Commands which generate output read the state to determine what is rendered, and thus, the output is state-dependent.

Scene graphs lift parameters which determine appearances into a high-level data structure that can be inspected, shared, and modified. Rendering is one such traversal in which parameters in scene graphs are applied to the internal OpenGL state to affect image synthesis. When rendered, these data structures are then used to perform the low-level commands which alter the state, based on abstracting them from the user.

### 2.1.3 Traversal

Although scene graphs may contain shared nodes, scene graph traversal is most simply understood as a depth-first tree traversal. Optimisations for sharing nodes are discussed later in section 6.3. In a functional language like Haskell, it is possible to express traversal generically, but, in keeping with common practice in graphics, discussion will focus on specialised traversals, most importantly rendering traversal. If the rendering function is called on a scene graph, then it recursively calls the rendering function on it's children until it reaches a mesh. For example, if we have an initial state, we can add lighting, then perform a rotation on two different scene graphs already constructed with their own settings. It will then apply the initial lighting and rotation, before performing the other operations in those two scene graphs when rendered. The downside, however, is that if we wish to draw an object with different light settings, we are forced to perform the same traversal twice to get our desired result. This can be optimised in several ways, which we shall explore further into the report.

### 2.1.4 Parameters

A scene graph is represented as a recursive data type with multiple constructors, one per type of node in the scene. Node constructors will have fields carrying node-specific data and/or the sub-tree rooted under the node. The fields are used as function parameters so as to change the state dependent on the node. The given scene graph argument is then rendered after the state has been changed.

### 2.1.5 Use in FreeD

The starting point for my design, was to use two existing libraries which use the scene graph data structure, so I have taken influence from them to justify the use of scene graphs in my approach. Scene graphs are easy to visualise in comparison with stateful graphics, whereby once a property has been set, it does not change unless told otherwise. It's a paradigm that enforces construction of scenes encapsulated by the properties of itself.

## 2.2 Haskell

Haskell is a strongly-typed, general-purpose functional programming language. Learning it was my first priority leading up to and at the start of the project. For this, I studied a book titled *Learn You a Haskell for Great Good* [47], which heavily aided my learning of a functional style of programming. This is vastly different to an imperative style, which is the only style of programming I have used previously.

The main challenge when using Haskell as a solution language for this project is the difficultly in building an *elegant, robust, extensible*, and functional high-level API on top of OpenGL's imperative, stateful bindings. As mentioned earlier, there are several functional libraries available, which are further explored in section 2.3. We will attempt to incorporate aspects of them into FreeD, to build such a solution that fits our criteria. There are varying levels of abstraction used to achieve a functional API, and the challenge lies in finding the correct balance. Although OpenGL has transitioned from a stateful approach by incorporating shaders, it is still highly dependent on setting global states, shown by its need to bind textures, vertex buffers/arrays, and shader programs before use.

I shall explain certain exclusivities to Haskell (or other functional languages) and why I have made use of them in FreeD in the following sections.

```
data Vertex a = Vertex2 !a !a | Vertex3 !a !a !a
  deriving (Show, Eq)

data Person = Person {
       age :: Int
      ,height :: Int
          }
          deriving (Show, Eq)

instance Functor Vertex where
  fmap f (Vertex2 x y) = Vertex2 (f x) (f y)
  fmap f (Vertex3 x y z) = Vertex3 (f x) (f y) (f z)

addNum :: Int -> Int -> Int
addNum a b = a+b

addVertex :: Vertex -> Vertex -> Vertex
addVertex (Vertex2 x1 y1) (Vertex2 x2 y2) = Vertex2 (x1+x2) (y1+y2)
addVertex (Vertex3 x1 y1 z1) (Vertex3 x2 y2 z2) = Vertex3 (x1+x2) (y1+y2) (z1+z2)
```

Figure 2.1: Haskell snippets

### 2.2.1 Algebraic Data Types

The algebraic data types used in Haskell are very different to imperative language with regards to subclasses and abstract classes. We can define a data type with several constructors, given different names, which means that we can represent several types of node of a scene graph. As shown in Figure 2.1, the `Vertex` data type has two constructors: `Vertex2` and `Vertex3`, which take `a`, this is a constructor argument of `Vertex` to define the type of values that is used for the arguments of both `Vertex2` and `Vertex3`. This shows just how easy it is to define them.

#### 2.2.1.1 Pattern Matching

Pattern matching goes hand in hand with the data types mentioned. Functions defined over an algebraic data type can be expressed by a set of equations over the input, using pattern matching on the data value to select the appropriate equation, and to unpack the fields of the constructor.

Figure 2.1 shows two functions, `fmap` and `addVertex`, for which `Vertex2` and `Vertex3` are both defined. This is possible via pattern matching on the correct constructor of `Vertex`. Pattern matching can also be used in case statements, given a value to determine what constructor was used.

#### 2.2.1.2 Record Syntax

In the same vein as above, record syntax is a way of a describing a data structure, and is similar to C's structs. Constructor fields are defined by their name and type. The name of the field constructors are functions, and are used to access the fields. They can therefore be used like any other function in Haskell. Figure 2.1 illustrates the differences in defining a data type using record syntax. We could have defined `Person` as:

```haskell
data Person = Person Int Int
```

This does not clarify what each individual constructor field is, and, to get access to the fields, we would have to implement separate functions. The following example demonstrates the usefulness of record syntax.

```haskell
> let person = Person {age = 10, height = 170}
> :t age
age:: Person -> Int
> age person
10
> let person2 = person {age=20}
```

Figure 2.2: Record syntax usage example

Figure 2.2 shows the binding of the `person` variable. The function `age` takes in a `Person`, and returns a `Int`, as seen by calling it on `person`. We then use `person` as a constructor, specifying that `age` is now bound to the value `20`. The rest of `person2`'s fields point to `person`, as they haven't been specified to be different.

#### 2.2.1.3 Type Synonyms

Type synonyms are a way of giving a different name to another type. They can give context to data types, or simply make it easier and quicker for the programmer to type. Here is how the standard library defines `String` as a synonym for `[Char]` [47].

```haskell
type String = [Char]
```

If we define a function to take a `String`, we can instead give it a `[Char]` and it won't think anything different of it. We are **not** making a new type, simply another name for that type. Here are a few other ways we can use the `type` keyword.

```haskell
type DVertex = Vertex3 GLfloat
type Params = Int -> Int -> Int
```

Figure 2.3: Type synonym usage

In Figure 2.3, we have defined `DVertex` as `Vertex3 GLfloat`. `Vertex3` is both a type, and a constructor name. `DVertex` is the type, where `Vertex3`'s constructor values are applied to be of type `GLfloat`. When we define `Params` to be `Int->Int->Int`, we can use it in a function definition to simply be `Func :: Params`. This will give us a function which takes two `Int` types as parameter and returns an `Int`.

#### 2.2.1.4 Maybe Data Type

The `Maybe` data type is defined as:

```
data Maybe a =  Just a | Nothing
```

I have made use of `Maybe` several times in FreeD. The `Nothing` constructor is analogous to `NULL` in imperative languages such as C or Java. This means that if it is not for certain that there will be a value in a function, or even in other data types, then we can use the `Nothing` constructor, and if there is a value, we can use the `Just` constructor. When we need to use that value, we can pattern match on the variable to handle whether it is a `Nothing` or inside a `Just` constructor.

### 2.2.2 Type Classes

A type class provides a way of defining a common set of functions over a wide variety of data types (i.e. function overloading). We can make a data type an instance of a type class, and implement the functions which are defined in that type class. Two widely used examples of type classes are `Eq`, whereby two variables of the same data type can be compared to return an equivalence value, and `Show`, which provides a string representation of the data type.

In Figure 2.1, `Vertex` derives `Eq` and `Show`. Type class instances can be specified explicitly, or in some cases, can be constructed mechanically from the data type via the "deriving" directive, if all the fields of the constructor of a data type are valid instances of the same type class. Below, we show an explicit definition of the instance `Show` for `Person`, by defining the function `show` for it to return a string.

```
instance Show Person where
  show person = ''person''
```

#### 2.2.2.1 Functor

Functor is a type class for conserving the context of the data type whilst applying a function to it. The data type is unpacked into it's constructor(s) using pattern matching. The function is then applied to it's contents and wrapped around with the constructor, so that the context is preserved. This is an area in which I did not fully take advantage, mainly due to my inexperience in Haskell.

`Vertex` is made an instance of `Functor` in Figure 2.1, and `fmap` is defined for both constructors of `Vertex`. Looking at `Vertex2`, it is deconstructed into it's arguments, and the function is applied to both arguments. They are then wrapped around again in a `Vertex2` and returned as such so as to preserve the context.

#### 2.2.2.2 Monoid

Monoid is a type class that captures the algebraic structure of a monoid. A data type is an instance of Monoid if it has an identity element, `mempty`, and an associative binary operation, `mappend`. Haskell does not check whether these functions abide by the mathematical rules of a monoid, so it is up to the programmer to enforce this.

```haskell
instance Monoid [a] where
  mempty = []
  mappend x y =  x ++ y

example = [1,2,3] `mappend` [4,5,6] `mappend` [] -- Result is [1,2,3,4,5,6]
```

Figure 2.4: Monoid instance definition of a list

Figure 2.4 shows a list being made an instance of `Monoid`. The identity (`mempty`) is an empty list, and the associative binary operator (`mappend`) is the addition of two lists, as we can see from the function `example`.

### 2.2.3 Folding

Folding is the notion of reducing a data structure to a single value. This goes hand in hand with the `Monoid` instance earlier discussed. `Foldable` is another instance which tells us whether a data type can be folded for it's defaults, and uses the `mempty` and `mappend` functions. `Foldable` defines a function `fold`, which uses `mempty` as the initial accumulator value, and calls `mappend` on each value in the data structure, until it has reached the end. This relies on the programmer implementing the instance functions, such that they traverse the whole structure correctly. Here is an example:

```haskell
sum' :: [Int] -> Int
sum' xs = foldl addNum 0 xs

eleven = sum' [3,5,2,1] -- Answer is 11
```

Figure 2.5: Folding example

Figure 2.5 shows a function which takes in a list and returns it's sum. As can be seen, the starting accumulator is 0, and it uses the function `addNum`, defined in Figure 2.1, to take the accumulator and add it to the next number to become the new accumulator, which then applies that to the next element in the list, and so on until it reaches the end. The function `eleven` gives an example on a list of numbers to return the value 11.

### 2.2.4 Curried Functions

Currying functions is the notion of treating a function which takes multiple arguments as functions taking only one argument. This is then applied to the first argument to return another function, which is applied to the next one, and so on until a result is returned.

Figure 2.1 introduces a function `addNum`, which can be applied partially to 3 and binded to `add3`, which is a function that takes in one argument (an `Int`), and adds it to 3.

```
> addNum 3 5
8
> let add3 = addNum 3
> add3 5
8
> fmap add3 [1,2,3]
[4,5,6]
> add3 $ addNum 2 $ addNum 4 5
14
```

Figure 2.6: Currying example

### 2.2.5 $ (Dollar Sign)

Ordinary function application is treated as left-associative by Haskell's grammar, i.e.

`f x y === (f x) y`

`$` is a right-associative operator which simply performs function application:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Figure 2.7: Dollar definition in Haskell

It is often used to construct right associative applications, avoiding the use of parentheses, e.g.

`f $ g u $ h x`

rather than:

`f (g u (h x))`

### 2.2.6 Bangs

One such way to provide some optimisation is by specifying the strictness of a variable in a data constructor by using bangs (exclamation marks before a variable). We are essentially telling the compiler that the value will **always** be required, and should be evaluated to weak head normal form [1] when the value is created. This means we only evaluate the value until you reach the outermost constructor.

[1]The outermost part of the expression should be either a data constructor or a lambda. [40]

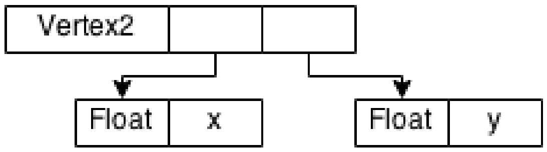


Figure 2.8: Lazy evaluation of `Vertex2`

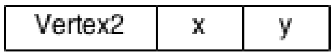


Figure 2.9: Strict evaluation of `Vertex2`

The `Vertex` data type defined in Figure 2.1 makes use of bangs. If we were not to use bangs on `Vertex2`, the values would be evaluated as shown in Figure 2.8. Assuming that each value takes one space of memory, we take up 7 spaces of memory: 1 for the constructor of `Vertex2`, 2 more for each pointer to the `Float` constructor, and another 2 each for the constructor of `Float` and the value. Figure 2.9 shows how the unboxed values would be evaluated under strict annotations in conjunction with the compiler directive to unbox the strict values, taking up only 3 spaces: 1 for the constructor of `Vertex2` and 2 for both of the values. If a value is boxed, we store a pointer to the value. When scaled to larger numbers, this can have implications on memory and speed.

## 2.3 Existing Functional Rendering Engines

Several libraries were researched for the purpose of becoming the foundation of FreeD, as well as to construct and meld ideas that FreeD build upon. My supervisor had mentioned two specific libraries start with: Gloss and Fizz. These two indeed became the foundations of FreeD, and I have drawn ideas extensively from both of them.

### 2.3.1 Gloss [10]

Gloss is a simple 2D functional rendering engine which was authored in 2010. Since then, it has become the most downloaded 2D graphics engine on hackage, and is still currently maintained. This is one of the reasons I have chosen to use it. The other being that my supervisor advised me to look at it first, so naturally, when FreeD was being implemented, I drew from the library I had paid most attention to.

Gloss uses a scene graph approach to build up a 2D scene. I have taken inspiration from the data structure used to represent each node in the scene graph, called `Picture`, and the method of rendering of them. In appendix F.1, we show an example of how Gloss's data structures can be used to build scenes.

As the scene graph is traversed, different actions are taken, dependent on the primitive being rendered in the Gloss library. A `Picture` primitive is enumerated to be nothing, a line, a circle, or a polygon. Gloss attempts to optimise by reducing the amount of calls to `preservingMatrix` by rendering items based on the position given, instead of manipulating the modelview matrix directly. One such example with the `Circle` and the `Translation` node is, instead of using `preservingMatrix`, it renders the circle with an offset to the x and y positions in its constructor, as shown by the following code:

```
Translate posX posY (Circle radius)
  -> renderCircle posX posY circScale radius 0
```

Figure 2.10: Translation optimisation in Gloss

### 2.3.2 Fizz Data Structures

Fizz [43] also uses scene graphs to render objects. The scene graph data structure used by Fizz, `HsScene`, consists of nodes which represent camera, geometry, transformers or a group. Each node takes in different argument: for the camera node, a scene and the viewing properties; the geometry node, a list of different geometrical data structures; the transformation node, a transformation to be applied and the scene the apply it to; and the group, a list of scenes.

I have used the geometry nodes from the Fizz library extensively to build my own geometric data structures. Initially, for the FFP approach, I have used almost identical calls to render geometry. This is only in my first revision of the FFP version of FreeD. I have since deprecated the use of the rendering calls in turn for more optimised methods, which use VBOs and shaders, the former being introduced as an optimisation to the FFP, and then carrying over to the shader version of FreeD.

One of the potential applications for the FreeD library is to be used as a backend engine for Fizz.

### 2.3.3 Other Engines

There are a number of other engines I have looked at, which have provided great insight into the current state of 3D functional graphics engines in Haskell. Some examples are as follows:

#### 2.3.3.1 not-gloss [26]

This library is very similar to FreeD, in that it emulates heavily upon Gloss' data types and uses scene graphs to traverse when rendering. The main difference is that it does not support OpenGL 2+.

#### 2.3.3.2 SceneGraph [36]

This is an old (2010) library which was developed for the purpose of research into scene graph implementations in Haskell, by the community on hackage [36]. Similarly to FreeD, it has operations to perform translation, rotation, camera settings, lighting, but, instead of using data constructors build the scene graph, it uses functions. An example would be:

```
runScene $ osgt $ cube 0.5 `colour` Green `scale` v1x 40 `translate` vy 2 <+> ↩
    camera `translate` vy (−10) <+> light `translate` vz 10
```

Figure 2.11: SceneGraph rendering example

This is traversed from the right to the left, whereas in FreeD, it is traversed from left to right.

#### 2.3.3.3 GPipe [12]

GPipe is a purely functional library for programming the GPU. It does not use scene graph traversals. It instead focuses on a lower-level layer upon OpenGL's binding to provide a purely functional way of writing low-level graphics in much the same way as OpenGL. It has, however, not been updated in some time, and does not compile under current libraries.

#### 2.3.3.4 fwgl [8]

fwgl is a 2D/3D game engine which has support for building in javascript. It uses shaders, and can handle loading of custom meshes. It also provides an external interface for a GLSL embedded domain-specific language (eDSL), which allows users to build shaders inside the program. I have taken pointers from this in terms of building an eDSL and also general shader support.

#### 2.3.3.5 Additional Libraries

Additional libraries used for inspiration were:

- **LambaCube 3D** [22] - A very rich eDSL library and rendering engine. eDSL ideas were used from this.
- **Lambency** [23] - Rendering framework using functional reactive programming (FRP) and focusing towards games.
- **helm** [19] - A functionally reactive game engine.

#### 2.3.3.6 Summary

Gloss is the library that I have chosen to use as the basis for the core data structure equivalent in FreeD, known as `Drawable` to Gloss' `Picture`. I will also take a similar approach to rendering, using case statements to determine the node at the current point in the scene.

I have chosen to use Fizz for rendering geometry as well as the data structures to represent geometry, lighting and camera.

There are a number of other libraries that are available to look at, but given lack of use (evident by number of downloads), and/or maintenance (modified date), have chosen not to investigate further.

| **Features** | FreeD | Gloss | not-gloss | SceneGraph | GPipe | fwgl | LambdaCube 3D | Lambency |
|---|---|---|---|---|---|---|---|---|
| Shader Support | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Model Loading | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| eDSL | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Scene Graphs | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Default Shaders | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Interaction | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Figure 2.12: Comparison table of engines

Although I have stated FreeD offers model loading and an eDSL, they are both very limited in that the eDSL is non-external, and the model loading is only for one file type (OBJ) and does not contain full support (material properties and only one type of geometry).

## 2.4 Interaction

I have looked into two ways of handling interaction: event handlers and functional reactive programming (FRP).

### 2.4.1 Event Handling

In terms of event handling, I have mainly looked at Fizz, which treats event handlers as a property of each node encased in a `Maybe` type, so that `Nothing` can be passed to the node instead of an event handler. There are several functions which take in properties and return new properties given parameters. These can be assigned to an event handler. There are several pre-defined functions, such as pan and zoom for the camera. There are also pre-define event handlers which map certain events to specific functions, such as the right mouse button being pressed and dragged to zoom. Event handling is more suitable to the FFP approach, as state is not manually kept during traversal.

```
handle :: Event -> HsScene -> HsScene
handle e (Camera h view g)   = Camera h (response h e view) (handle e g)
handle e (Geometry h mode g) = Geometry h mode (response h e g)
handle e (Transform h t)     = Transform h (response h e t)
handle e (Group h g)         = Group h (map (handle e) (response h e g))

response :: HsHandler a -> Event -> a -> a
response Nothing _ node  = node
response (Just f) e node = f e node
```

Figure 2.13: Fizz's event handling

In Figure 2.13, I have included several of the functions for dealing with given event handlers with certain nodes. As can be seen, the node is reconstructed after being given to the `response` function.

### 2.4.2 Functional Reactive Programming

FRP [46] is a paradigm which revolves around taking a **time-evolving** data input from any source and producing an output. FRP modifies behaviours over time, both continuous and discrete. Similarly to event handling, we can create functions on what to do given the values, and then simply poll for the input data on how to transform that information into our available functions. However, we do have the choice of adding in some very general functions which simply map over the state. This could involve making instances of new classes to perform functions generally over our state. FRP uses a substantial library of combinators, and requires bindings to general events, both of which require the investment of a considerable amount of time into both learning and implementing. Since interaction is not a priority, I have chosen not to pursue FRP further.

### 2.4.3 Summary

I have not spent very much time on dealing with interaction, as this project, first and foremost, promises to deliver an engine with a focus on rendering. Because of this, I have not had time to add interaction into FreeD. This has forced me to use a layer of interaction above FreeD in which the variables I use to construct my scene graph are changed via `IORef` values. `IORef` values are values which are akin to a mutable global variable. If the values are updated, anywhere they are referenced has access to the updated value. In this respect, it is easy to add interaction the engine, but it is very limited to specific event handling. I have opted to not incorporate such a rigid interaction system into my engine for this very reason.

## 2.5 OpenGL and HOpenGL

OpenGL [28] is the industry standard API for rendering 2D/3D graphics written in C. There are Haskell bindings to the underlying C code available as a library called HOpenGL [30], which is what we will use as the basis of our library. I have researched into the two ways of rendering graphics in OpenGL: using the FFP and shader programming.

### 2.5.1 Fixed-Function Pipeline (FFP)

The FFP was the original and dominant way of rendering graphics until shaders came along. It was released in 1992 and was used for a vast amount of applications. It focused on specific function calls to modify the state in the rendering pipeline, which includes calls like `glBegin, glEnd, glEnable, glDisable, glVertex, etc...`. To tell the OpenGL state machine to draw, we use a function called `glBegin(GL_PRIMITIVE)`, where the primitive mode is specified as a parameter. We can then use `glVertex` commands to construct the primitive. The vertices are sent down the pipeline when enough `glVertex` calls to make up the primitive have been made. The primitive is then rasterised and rendered. `glEnd` is used to specify that we want to stop drawing.

### 2.5.2 Vertex Buffer Objects (VBOs)

VBOs are a way of defining vertex data, such as the position, normal, texture coordinates, colour, etc. for the video card. They are an alternative to immediate-mode rendering, which consists of `glBegin` and `glEnd` functions. They offer a large performance gain over immediate-mode rendering, primarily because the data is on the video card memory rather than on the system memory, enabling it to be rendered directly by the video card instead of sending one vertex at a time [39]. These can be further improved with the use of element buffer objects, explained in appendix F.3.

### 2.5.3 Shaders (OpenGL 2.0+) [31]

Shaders were introduced to the OpenGL graphics pipeline in 2004 [37], and have ever since evolved and adapted to become the modern way of rendering graphics. The reason for their introduction is that more and more functions were being added to the FFP until it was decided that instead of having fixed functions, it is better to let the programmer have full versatility in the functionality, enabling much more complex and specialised functions.

A graphics processing unit (GPU) is a specialised piece of hardware designed to accelerate the rendering of graphics. They are optimised to run commands, such as matrix and vector calculations, which are heavily used in rendering.

A shader is a small program executed by the GPU, which performs functions for several parts of the graphics pipeline, from the initial geometry to the final pixel. This gives developers much more control over what is drawn, and is much faster than the old FFP method. This is mainly because the vertex information for geometry resides on the graphics memory. I have used the book titled *Graphic Shaders: Theory and Practice* [41] to learn about shader programming, and how they have evolved. I have also used another book, titled *OpenGL Insights* [49] to learn how to transition from rendering graphics using the FFP to shader programming. Another resource I have used in aid of learning shading programming is *Open.GL* [29], following tutorials on their website.

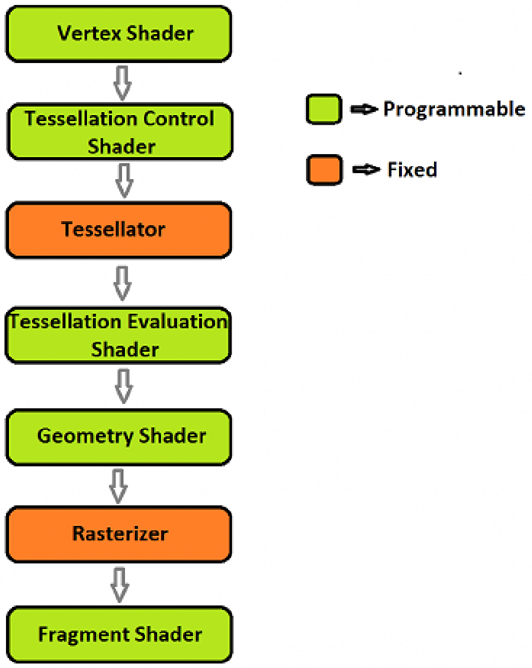


Figure 2.14: Graphics Shader Pipeline [21]

As can be seen in Figure 2.14, there are a total of 5 shaders in the pipeline. We are only taking into account the vertex and fragment shaders for the purpose of FreeD. Essentially, at each stage, information can be passed onto the next part in the pipeline. Shaders are intended to replace parts of the FFP in an attempt to emulate them, but with much more control. Certain parts, as can be seen, are still fixed, so shaders have not fully replaced the whole of the rendering pipeline.

```
uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;

in vec3 aVertex;
in vec3 aColor;

out vec4 fColor;
void main()
{
  fColor = vec4(aColor, 1.0);
  gl_Position = uProjectionMatrix * uModelViewMatrix * vec4(aVertex, 1.0);
}
```

Figure 2.15: Simple vertex shader

#### 2.5.3.1 Variables in Shaders

Variables are needed to bind to data in shader programs, and/or to be global parameters. We define binded data by declaring them as `in/out` types, and global parameters using `uniform` type.

There are several ways of declaring a variable in shaders. There are also a number of different variable data types, which include `float, vec*, mat*`, where `*` is a placeholder for the size (2,3,4) of the variables. Vectors and matrices are among the most commonly used, thus there are many capabilities built in for handling these types, such as the cross and dot product, and normalisation.

Vertex array objects (VAOs) are used alongside VBOs. They are used to describe the positions at which each attribute of a vertex lies.

`Uniform` variables are a type whereby every shader has access to them and their value does not change. `in` variables are *read-only*, and are passed to the shader through the VAO, and, depending on what shader they are in, they can be received from the prior part of the pipeline. The vertex shader takes them in from the VAO, and the fragment shader takes them in from the shader directly beforehand as interpolated values. `out` variables are simply the opposite of `in` variables, in that they are output to the next part of the pipeline and are *write-only*. The `out` variable from a vertex shader feeds into the `in` variable of the next shader in the pipeline and so on. There are also `flat` variables, which enforces that the value is not interpolated when given to the next shader; it will use the last `out` variable processed by the previous shader as the `in` variable, with respect to the primitive. For example, if we were to draw a triangle, specifying that the colour from the vertex shader is `flat`, the next shader would use the colour given from third 3rd vertex processed by the vertex shader.

#### 2.5.3.2 Vertex Shaders

Vertex shaders are applied to each vertex as it is processed through the pipeline. These are sent via a VAO whose attributes are specified at runtime by calls to OpenGL commands. Each `in` variable is set using offsets per vertex. We demonstrate this in Figure 2.15, as we have `aVertex` and `aColor` as `vec3`. This is represented in our VAO as 6 elements per vertex in the array linearly set (assuming no element buffer object). These are only sent to the vertex shader, as for each of the succeeding stages in the pipeline, the information is received from the previous stage.

As shown in Figure 2.15, all that is passed to the shader is the modelview matrix, projection matrix, vertex, and colour. This does mean, however, that changes to the modelview or projection matrix by way of transformations or perspective modification in the camera have to be done manually, and then passed to the shader. This behaviour is abstracted from the library so that the user does not need to directly deal with matrices, but still uses shaders. This is not to say that the user can't change the matrices manually, and pass them through instead, which is perfectly fine. Even a mix-and-match approach works the same.

#### 2.5.3.3 Fragment Shaders

Information from the vertex shader is passed to the next stage, which is the fragment shader in FreeD's case. We have an `out` variable which is interpolated for each fragment on the screen. `fColor` is passed into our fragment shader shown in Figure 2.16, and then assigned to `outColor`, which is what the RGBA value of the pixel on the screen will be (if we define more than one out variable in the fragment shader, we must specify which is the final pixel value), subject to blending and depth-tests.

```
in vec4 fColor

out vec4 outColor;
void main()
{
  outColor = fColor;
}
```

Figure 2.16: Simple fragment shader

We could make our fragment shader even simpler by specifying `fColor` as a `varying` and not setting anything in the main function. This would be less clear, however. The position has already been interpolated after rasterising the object, so it is not necessary to mention the position for this example (it is needed, however, for certain shading/lighting effects, and other techniques).

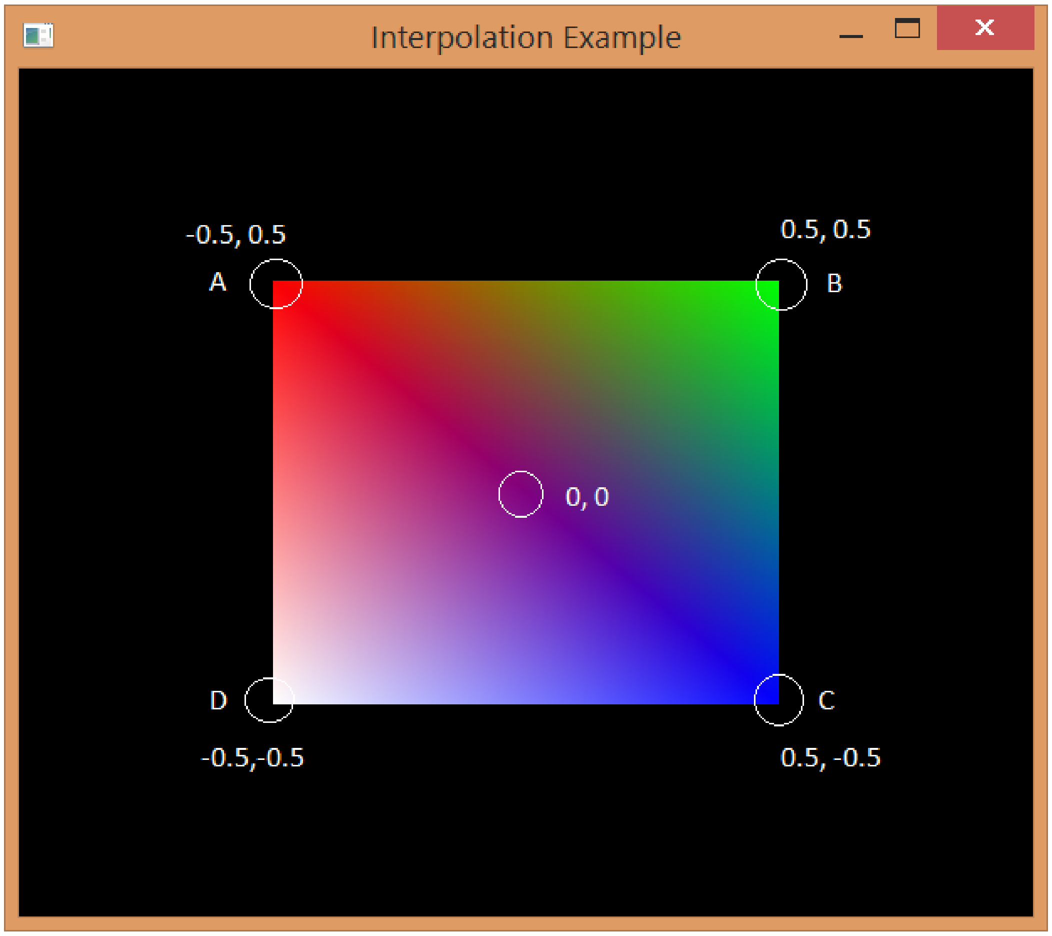


Figure 2.17: Linear interpolation effects from above shaders

I have used the aforementioned shaders to demonstrate the power of shaders in a very simple way, as shown in Figure 2.17. As can be clearly seen, each corner of the square takes red, green, blue, and white respectively. These are vertex attributes, and so are passed along in the VAO. This means that when output to the fragment shader, they are linearly interpolated. This is further explained in Appendix F.2.

#### 2.5.3.4 Other Shaders

There are 4 types of other shaders: Geometry, Tessellation Control, Tessellation Evaluation and Compute shaders (not in rendering pipeline). These are newer than vertex and fragment shaders, introduced in OpenGL 3.2, OpenGL 4.0 (both tessellation shaders) and OpenGL 4.2 respectively. I have chosen not to include these in my library due to time and complexity constraints.

Geometry shaders can generate new primitives from the given one in the previous part of the pipeline. Say the primitive mode of `Triangle` is used, we can generate lines, points, etc. They can "emit" several primitives per original primitive or none, even. An example of this could be approximation a curve using lines. Passing it through a geometry shader could generate extra lines which provide a better approximation [37]. Geometry shaders are not generally used for level-of-detail (LOD) type calculations, however. This is usually provided in the tessellation shaders.

Tessellation shaders can also create primitives, but they must be of the same type as the original. This is very useful for LOD calculations and subdivision techniques.

Compute shaders are not part of the rendering pipeline. Although they *can* be used to render, they are usually not. They are generally used to compute arbitrary information.

Every stage interpolates the `out` values from the previous one, as well as the position of vertices.

#### 2.5.3.5 Shaders in Haskell

There are Haskell bindings to OpenGL's use of shaders, though I have opted not to use them directly, and have instead utilised *GLUtil* [11], a library which provides a convenient way of loading a simple shader program given vertex and fragment shaders as filename or string sources. It does not support the other shaders discussed above, but they can easily be incorporated into the library's loading facilities. GLUtil also provides convenient functions to set attribute and uniform variables using Linear's [24] data types, which include fixed vectors (`V1, V2, V3, V4`) and matrices (e.g. `M44`, 4x4 matrices).

#### 2.5.3.6 Embedded Domain-Specific Language (eDSL)

An eDSL is essentially a small language written for a very specific use within an application. In our case, it's GLSL shader code. There are no GLSL eDSL libraries freely available for Haskell, or, at least, no usable ones. This has prevented me from being able to generate shaders in FreeD, using an external library. Although several of the rendering libraries compared earlier provide an eDSL for creating shaders, they are tightly coupled to their respective engines, and do not provide an easy way of using these shaders to FreeD's needs. I have taken elements from several of these libraries, as well as other libraries which have briefly been looked over. One in particular: Häte [18] provides an easy to use API, which produces shader strings, excluding the body of the main function of a shader. I have used this as the basis of creating an eDSL for FreeD.

## 2.6 Model Loading

There are many different file types for holding mesh data. Several of the more prominently used ones are OBJ, Collada, and PLY. I have researched the following libraries for this purpose:

- GPipe-Collada [13]
- collada-output [3]
- graphics-formats-collada [14]
- collada-types [4]

- obj [27]
- ply-loader [34]

There are unfortunately no libraries available on hackage that allow us to load in these file types easily to use with FreeD's data types for holding geometry. One of my friends has implemented one on my behalf, and thus I have used it to load OBJ files [48].

## 3 Core Implementation

This section outlines the core implementation common to both of my implementations of FreeD.

### 3.1 Drawable

`Drawable` is the data structure for representing a scene graph in FreeD. FreeD follows the nature of building a scene graph using constructor arguments as is done in Gloss [10].

```haskell
data Drawable
  -- | A blank Drawable, with nothing in it.
  = Blank
  -- | Geometry with the primitive drawing type (Lines, Triangles, Quads, etc)
  | DObject DGeometry PrimitiveMode
  -- | A Sphere with given radius
  | Sphere GLfloat
  -- | A Cube with given width, height and depth.
  | Cube GLfloat GLfloat GLfloat
  -- | A Drawable drawn with this color.
  | Color (Color4 GLfloat) Drawable
  -- | A Drawable drawn with these material properties when lighting is used.
  | Material  Material  Drawable
  -- | A Drawable translated by the given x,y and z coordinates.
  | Translate GLfloat GLfloat GLfloat Drawable
  -- | A Drawable rotated clockwise by the given angle (in degrees) , and axis.
  | Rotate GLfloat  GLfloat GLfloat !GLfloat Drawable
  -- | A Drawable scaled by the given x, y and z factors.
  | Scale   GLfloat GLfloat GLfloat Drawable
  {- | Camera, to do perspective, can have multiple, but at the moment assumes only↩
     one
     Given eye, facing towards, and up vector.
     For perspective: angle, aspect ratio, znear, zfar. -}
  | Camera DView Drawable
  -- | Lighting, given position, ambient,diffuse, and specular properties.
  | Lighting DLighting Drawable
  -- | Texture to be used for Drawable (if texture coordinates are given).
  | Texture TextureObject Drawable
  -- | To turn off the current Texture.
  | TextureOff Drawable
  -- | A Drawable consisting of several others.
  | Drawables [Drawable]

instance Monoid Drawable where
  mempty    = Blank
  Blank `mappend` a = a
  a `mappend` Blank = a
  mappend a b = Drawables [a, b] -- Puts two Drawable objects into a list
  mconcat   = Drawables -- Appends all Drawables and sticks them in one.
```

Figure 3.1: `Drawable` data type

We show the definition of the `Drawable` data structure in Figure 3.1, and can see several types of node defined. `Blank` is used to render nothing, which is useful when we are converting other data structures into a `Drawable`, and, if there are no distinguishable types of node to use, we can use `Blank` instead. It is also useful for monoidal purposes, in that

`Blank` is the identity element of `Drawable`, and, when appended with another `Drawable`, the other `Drawable` is returned. This is illustrated in Figure 3.1.

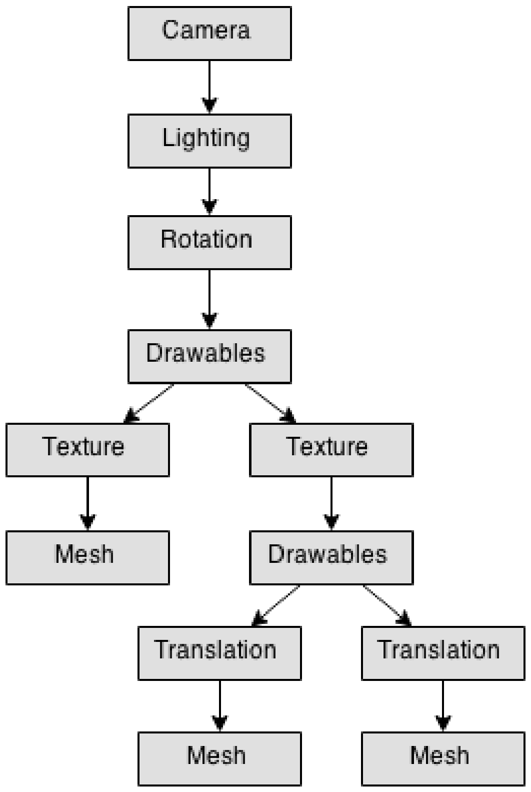


Figure 3.2: Simple scene graph

```
-- tex1 and tex2 are previously binded TextureObjects
let texturedbox = Texture tex1 box
let trans = Translate 5 10
let objects = Drawables [texturedbox, Texture tex2 $ Drawables [trans 5 object, ↩
    trans 10 object]]
-- view and light are previously binded viewing and lighting properties.
render $ Camera view $ Lighting lighting $ Rotate 10 0 0 1 objects
```

Figure 3.3: `Drawable` construction of Figure 3.2

Figures 3.2 and 3.3 demonstrate the versatility of the `Drawable` data structure. It is simple to construct the scene in Figure 3.2 using FreeD's API.

## 3.2 Geometry

Simply put, the geometry of an object is information about it's structure, in terms of vertices, normals and/or texture coordinates, as well as their connectivity. It is generally held in files and can be read in to match data structures. As mentioned in subsection 2.6, I did not find an effective library to load in these file types on hackage. I opted to use an

OBJ file loading package created by a fellow student [48], specifically for the use of FreeD . I made intermediate functions which transformed the read-in data structures to match my own. At the moment, it only supports reading in vertices, normals and texture coordinates, but OBJ files can read in material information, which decides on what texture to use as well as the material properties. This can give a full description of a model.

Geometry is very easily introduced into the program by way of the `DObject` data structure shown in Figure 3.1. For the loaded files, I have used a data structure called `DGeometry` shown in Figure 3.4, which contain several types of geometric data structure, as we try to capture common patterns of geometric specification (e.g. a list of vertices, vertices with normals and/or texture coordinates). This is taken from Fizz. Of the `DGeometry` constructors, `DGeometryVnt` is used for loading in an OBJ file which consists of a list of triples containing a vertex, normal and texture coordinate.

Loading geometry via files is not the only option, as they can be manually constructed algebraically, although I have not included any examples of this.

```haskell
type DVertex = Vertex3 GLfloat
type DNormal = Normal3 GLfloat
type DTexCoord = TexCoord2 GLfloat

data DGeometry = DGeometryV [DVertex]
             | DGeometryVn [(DVertex, DNormal)]
             | DGeometryNv DNormal [DVertex]
             | DGeometryNt [(DNormal, (DVertex, DVertex, DVertex))]
             | DGeometryVnt [(DVertex, DNormal, DTexCoord)]
             | DGeometryVt [(DVertex, DTexCoord)]
```

Figure 3.4: `DGeometry` data type

There is one main weaknesses to the data structure shown in Figure 3.4, which is the amount of overhead needed for holding the geometry. Using `DGeometryV` as an example, it's constructor takes in a list of `DVertex` values.

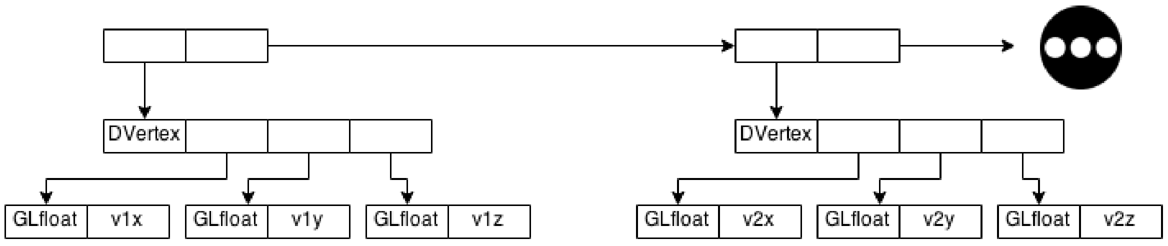


Figure 3.5: `DGeometryV` memory structure

Figure 3.5 shows the how the values are stored. The constructor points to the `DVertex` constructor and the next item in the list, and then the `DVertex` constructor points to each of it's respective values. These point to the `GLfloat` constructor, which contains the values. This means that there are 12 values stored per vertex in a list using this representation. As this data structure was used for Fizz, it is deemed efficient enough to represent geometry in FreeD. This has not been pursued further, as it is not a priority.

## 3.3 Primitives

Identifying primitives in 3D is an important aspect of rendering engines, as they are what will be rendered on the screen. Primitives in 3D are common models used for rendering, which are built from polygonal meshes. Some of the commons ones found in rendering engines are:

- **Spheres**
- **Cubes/Boxes**
- Toroids
- Cylinder
- Pyramids

Of these, I have implemented spheres and cubes (highlighted in bold) in my engine, as they are commonly used. Ultimately, they are types of geometry built from meshes, and can be represented as such. They were first implemented using functions from a library called *GLUT* [17], but in the latest implementation of FreeD, they are read in via files, like any other type of geometry. They are adjusted based on parameters by transforming the values in the geometry data type using functions: For cubes, the width, height and depth; and for spheres, the radius. The rest of the primitives I have left out, since I do not think they are needed for the purpose of this library.

## 3.4 Representing Floating Point Numbers

Haskell has it's own primitives for representing floating point numbers called `Float`, as does OpenGL, whose representation is called `GLfloat`. Converting between them is trivial, but a nuisance. Any OpenGL calls that require floating point numbers require it in the `GLfloat` primitive, which would mean converting between them whenever one of these calls are made. Therefore I have opted to use `GLfloat` in my constructors instead of `Float` for consistency, and it's what OpenGL functions expect, so it would need to be converted at some point anyway.

# 4 Fixed-Function Pipeline (FFP) Implementation

## 4.1 Transformations

All calls in this section make use of the `preservingMatrix` function found in the OpenGL bindings, which mimics `glPushMatrix/glPopMatrix` found in the C version of OpenGL. This allows us to preserve the current matrix on the stack. It also ensures that we are maintaining a hierarchical approach to rendering, and that the transformations only affect the children of the current node.

All transformations take in a `Drawable`. When traversed, they are wrapped in a `preservingMatrix` call, the transformation is applied, and then then the render code is recursively called on the `Drawable` variable passed by the parameter.

### 4.1.1 Translation

Translation is the addition to the respective x, y and z components of the modelview matrix, which is achieved by calling the `translate` function on given x, y and z variables as parameters.

### 4.1.2 Rotation

In 3D, we have to handle 3 dimensions of rotation: pitch, yaw, and roll. This is defined by a vector and an angle. Rotation can be done using euclidian angles and 3x3 matrix multiplications. This, however, gives us something called ”gimbal lock”, where, if given pitch, yaw and roll, it is ambiguous which intermediate steps to take to achieve the rotation, which can give us a false rotation. To solve this, I have used quaternions. Quaternions are homogeneous co-ordinates represented by complex numbers.

I had originally used the **vect-floating** package from hackage to handle `GLfloat/GLdouble` values, but have since changed to the **Linear** package for better consistency with the HOpenGL functions, as well as providing many more functions. This way, we apply the rotation using quaternions and then convert it to a 4x4 matrix to be multiplied with the current modelview matrix on the stack.

### 4.1.3 Scaling

Scaling is a simple matter of multiplying the current modelview matrix's diagonal (excluding the fourth row/column) with the x, y, and z values each way the scene would like to be scaled in, respectively. This is implemented via a simple call of the `scale` command.

## 4.2 Camera and Lighting

Camera and lighting are very important in 3D graphics. FreeD is built using Gloss as a foundation, which does not implement these as they are not needed in 2D. Therefore, I have had to explore elsewhere to find ways of implementing this in an efficient manner. I thus decided to use Fizz's implementation as inspiration for handling these two aspects in FreeD.

### 4.2.1 Camera in 3D

The camera is handled by specifying where the eye is, the centre of it (or the point it's facing towards), and what direction is up for the eye. We also need to specify the field of vision for the eye in terms of an angle in radians, the aspect ratio of the screen, and the depth distance in the form the closest and the farthest z value. Once given these values, any object drawn, is drawn from this perspective. There can be multiple camera points, and a default one is given if none is supplied. The functions these use are commonly known as `perspective & lookAt`.

The data structure for storing the camera properties is called `DView` for the sake of not having several parameters whilst calling the render function. This also allows for easy changing of singular properties due to record syntax's nature.

### 4.2.2 Lighting in 3D

Lighting can be specified given a vertex position, and ambiance, diffuse and specular level (as colours). The `Drawable` specified after it will then follow the lighting properties of the ones set, so that multiple lights can be used to illuminate different items. This is useful if one wants to specify the dominant light source for a set of different objects.

Likewise to the camera node, we have given lighting a properties data structure called `DLighting` for the same reasons as above.

## 4.3 Colour

Colour is handled by specifying the RGBA values and simply setting the state of the colour to those values. The previous colour is recorded in a temporary variable and then set again to that temporary variable after the scene graph has been traversed so as to contain the colour to that branch. I have used the American spelling of colour, that is, `Color` for consistency with OpenGL's version, as well as other libraries.

## 4.4 Compiled Nodes

Many times, we do not wish to re-calculate all of our vertices/normals again. For this, we need to provide the user with capabilities of compiling a `Drawable`, and then drawing the compiled node. We make use of something called a `DisplayList`, which "is a series of graphics commands that define an output image" [5]. The commands are stored and compiled for later use. This is done by rendering the `Drawable` as we would to draw it on the screen, but instead, we assign it to a `DisplayList` and store the pointer as a `Compiled` node in our `Drawable` data structure. For this purpose, I have created a `compile` function, which is taken from Fizz. When we traverse the scene graph and reach a `Compiled` node, we call the `DisplayList` to draw. This saves a lot processing power by reducing the necessity to traverse the scene graph constantly. An example of this is shown in Figure 4.1.

```
compiled <- compile $ Texture texture1 box
render compiled
```

Figure 4.1: `Compiled` node example

## 4.5 Textures

Textures are very important in rendering engines, as they are often used in place of geometry, or show change in objects. For example, if in a game in which an object can be damanged, a texture depicting each stage of the damage could be used. Each time the object is damaged, the texture used to render it would be changed. There are a vast number of uses for textures, which makes them worthwhile in an rendering engine.

FreeD handles textures by taking in a `TextureObject` and binding it, and then turning off the binding after the `Drawable` has been rendered. I have included a texture loading function: `loadTex` for the sake of loading bitmaps, JPEGs and PNGs from filenames. This function is reliant on the GLUtil package [11].

## 4.6 Optimisation

### 4.6.1 Vertex Buffer Objects

I have attempted to optimise the way in which geometry is handled by making use of VBOs. This essentially means that the geometry is stored on the GPU as opposed to the CPU, and therefore it can be rendered directly by the GPU. It also has the added advantage of being the type data structure that shaders use.

Since using this data structure, I have abandoned my current rendering approach towards geometry, which is based upon Fizz, and adopted a different way. This involved specifying vertex attributes for each type of geometry over where it is stored inside the VBO.

Several attributes can be stored in VBOs, such as the vertex position, normal, colour, and even texture coordinates. When rendering, we need to specify how big each *chunk* of data is in the VBO, and where each bit of information is. The chunk of data is called the *stride*, which is calculated by multiplying the amount of memory of each element (usually a `GLfloat`) by the amount of elements per vertex. Then we set the offset for each element. For example, if we have a VBO with 3 floats for the vertex and 3 for the normal, we set the stride to 6 multiplied by the size a `GLfloat` uses. We then specify how many elements there are for the vertex position, the stride, the type we expect, and the offset from the start. This is best demonstrated by a code example:

```
displayVbo :: GL.BufferObject -> GLUT.NumArrayIndices -> IO ()
displayVbo buff size = do
    let stride = fromIntegral $ sizeOf (undefined::GL.GLfloat) * 6
        offset = fromIntegral $ sizeOf (undefined::GL.GLfloat) * 3
        vxDesc = GL.VertexArrayDescriptor 3 GL.Float stride $ offset0
        nDesc = GL.VertexArrayDescriptor 3 GL.Float stride $ (offsetPtr offset)
    GL.bindBuffer GL.ArrayBuffer $= Just buff

    GL.arrayPointer GL.VertexArray $= vxDesc
    GL.arrayPointer GL.NormalArray $= nDesc

    GL.clientState GL.VertexArray $= GL.Enabled
    GL.clientState GL.NormalArray $= GL.Enabled

    GL.drawArrays GL.Triangles 0 $ size `div` 6
    GL.bindBuffer GL.ArrayBuffer $= Nothing $
```

Figure 4.2: Code for drawing a VBO onto the screen

The example in Figure 4.2 takes in a VBO and it's size. As you can see, `stride` is calculated by multiplying the size of a `GLfloat` by 6, as we expect 6 values per vertex. `offset` is 3 multiplied by the size of a `GLfloat`, as we are specifying that this is when we encounter the normal for this vertex. We point to where the vertex is using `VertexArray` and we point to the normal similarly using `NormalArray`. We enable both of them, and then we draw given the amount of vertices we want. We have done `size ‘div‘ 6`, as we have 6 values per vertex, so the amount we are drawing is the size of the VBO divided by 6. We are also drawing using Triangles, and the offset for the VBO is 0. We can see this in effect via Figure 4.3.

We also use Figure 4.3 to show different combinations of storing vertex information in a list. Each box is a value in the VBO: One with vertex position, normal and texture coordinates; one with vertex position and normals; one with vertex position and texture coordinates.

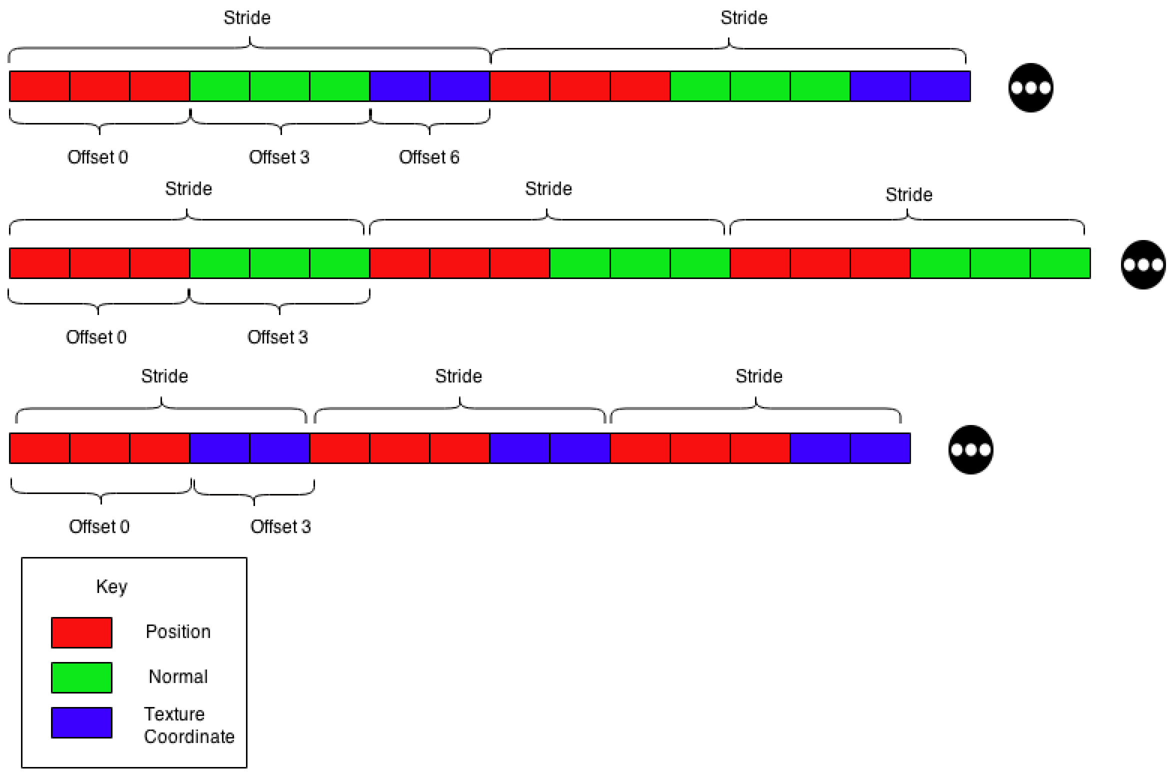


Figure 4.3: VBO storage examples

Instead of using `glVertex` calls to specify vertex information, we construct the VBO and send it to the GPU. Assuming the vertex information does not change, we simply tell the GPU to draw using the information stored.

Severals objects can be defined in the same VBO, using a different offset for each object and changing all the properties. However, this is out of the scope of this project.

# 5 Shader Implementation

This is the idea of rendering using the GPU, that is, to perform general purpose programming on the GPU in the form of vertex, fragment, geometry, tessellation, and compute shaders. The FFP is becoming obsolete [15] [7] due to many of it's functions being deprecated in the latest version of OpenGL. The aim is to migrate the current library to use shaders after branching from the original. We can subsume several parts of the FFP by using certain vertex/fragment shaders. For example, we can perform the following using a vertex shader, which would've otherwise been handled by the FFP:

- Vertex transformations.
- Normal transformations.
- Normalisation.
- Handling of per-vertex lighting.
- Handling of textue coordinates.

Shaders can deliver everything that the FFP can do, but has the flexibility to do more, e.g. beyond flat/Gourad shading, we can do Phong, exact, and anisotropic shading (explained in section 5.2.5).

## 5.1 State Inheritance

In my solution, I have made a state data structure called `GLState` (Figure 5.1) to carry properties which would've otherwise been held by the FFP. This is because modern OpenGL has deprecated functions, such as `rotate`, `translate`, `scale`, etc. This enables me to use functional Haskell code to modify the state, and have much more flexibility over what values are stored. I carry the state explicitly as a value, using functions to take in the current state and values, outputting a new state, via record-syntax pattern matching. One such function, is shown in Figure 5.2 to demonstrate this functionality. Changes in state do not affect the siblings in the scene graph, and so, do not act on a global scale. They are purely hierarchical. Changes to state are maintained when the `Drawable` is traversed, such that if a `Drawable` modifies the state, the child of the `Drawable` will be traversed with the modified state.

Since using shaders, we do not have access to `preservingMatrix (glPushMatrix/glPopMatrix)` anymore, and therefore the state of matrices is maintained manually. This means that we have to have state inheritance in our scene graph for the child nodes of the state. For this, we have created a data type:

```
data GLState = GLState
  {
   mMatrix ::  (M44 GLfloat) -- ^ Model Matrix (for transformations).
  ,vMatrix :: (Maybe (M44 GLfloat)) -- ^ If a view Matrix is set (for camera ↩
      settings).
  ,pMatrix :: (Maybe (M44 GLfloat)) -- ^ If a projection matrix is set (for ↩
      perspective settings).
  ,lighting :: (Maybe DLighting) -- ^ If lighting properties are set.
  ,texture :: (Maybe TextureObject) -- ^ If a texture has been set.
  ,sProgram :: (Maybe ShaderProgram) -- ^ If a custom shader program is used.
  ,shaderProgramType :: ShaderProgramType -- ^ Record syntax of default shaders.
  ,material :: (Maybe Material) -- ^ Material properties.
  ,color :: (Color4 GLfloat) -- ^ Color.
  }
```

Figure 5.1: Data structure for holding the current state in the scene graph

The model matrix is modified for transformations such as translation, rotation and scaling, whereas the view matrix is for the camera settings, and the projection matrix, for the projective view of the scene. The lighting holds all the properties for lighting a scene, as well as the type of shading model to be used. Material properties can also be recorded for lighting, but they are not direct properties, so they are contained in a separate variable. Colour can be set and is used when there is no shading. It is also possible to have a `TextureObject`, which we can bind to display a texture with items which have texture coordinates.

Several of the variables are of the `Maybe` data type, meaning that they do not have to be set and are initialised to `Nothing`. When the scene is constructed and the user has passed one of these nodes, the node is replaced by a `Just` type. Then, when the scene is rendered, these are checked upon to decide which type of shader to use (stored in `ShaderProgramType` (explained in section 5.1.1)).

I have separated the modelview matrix into model and view matrices so that the camera can be abstracted from the model. This enables us to simply set the view and projection matrix when we use the `Camera` node. When sending the matrices to the shaders, I multiply them in my library instead of in the shader, so that the modelview matrix can be used to calculate the normal matrix, as this would be an expensive operation in the shader (due to calculating the inverse of a matrix).

```
translate :: GLfloat -> GLfloat -> GLfloat -> GLState -> GLState
translate x y z state = let newM = mMatrix state !*! translationMat x y z
        in state { mMatrix=newM}
```

Figure 5.2: State transformation example

In Figure 5.2 a function called `translate` is shown, which takes in a `GLState` and three `GLfloat` values, constructs a translation matrix, and multiplies them with the current matrix in the given state. It then returns the state with the newly updated matrix.

### 5.1.1 ShaderProgram and ShaderProgramType

```
data ShaderProgramType = ShaderProgramType
  {
     sSP :: ShaderProgram  -- ^ Simple shader
    ,tSP :: ShaderProgram  -- ^ Texture shader
    ,pSP :: ShaderProgram  -- ^ Phong shader
    ,ptSP :: ShaderProgram -- ^ Phong + Texture shader
    ,fSP :: ShaderProgram  -- ^ Flat shader
    ,ftSP :: ShaderProgram -- ^ Flat + Texture shader
    ,gSP :: ShaderProgram  -- ^ Gourad shader
    ,gtSP :: ShaderProgram -- ^ Gourad + Texture shader
  }
```

Figure 5.3: Data structure for holding the default shaders

We have a `ShaderProgram` type, which can be defined if the user wishes to use their own shaders instead of the preset ones. The preset shaders are defined in `ShaderProgramType`, which holds the `ShaderProgram` for all of the default shaders, enabling us to reuse them, shown in Figure 5.3.

## 5.2 Replacing the fixed-function pipeline (FFP)

To replace the FFP, we shall incorporate several simple shaders to mimic the behaviour we would expect by using different nodes in our `Drawable`. Firstly, we shall explore how vertex data is retrieved and used for the purposes of shader programming.

### 5.2.1 Geometry

Before FreeD transitioned over to render using shaders, the FFP implemented VBOs to optimise geometry. Since shaders expect VBOs, this involved very few changes to the code. Instead of enabling states, and setting the pointers for the `VertexArray, NormalArray, TexCoordArray` as is done on using the FFP, I use a VAO to describe the vertex attributes in terms of sending vertex information down the pipeline. Assuming there are no changes in the vertex information, we can simply bind the VAO created, and then tell OpenGL to draw using the currently bound VAO.

### 5.2.2 Transformations

Transformations are handled in the model matrix, which is passed through to every shader as a `uniform` variable. This matrix is kept in the state, and, when encountering a transformation, we simply multiply the current matrix on the stack with the constructed matrix needed for the particular transformation, instead of using OpenGL's `multMatrix` function. The constructed matrices are formed from an identity matrix (Figure 5.5), and then the values whose position is dependent on the type of transformation are placed in the identity matrix.

$$\begin{array}{ccc|c} sx & r & r & x \\ r & sy & r & y \\ r & r & sz & z \\ \hline 0 & 0 & 0 & 1 \end{array}$$

Figure 5.4: Position of variables which are changed based on the transformation

$$\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}$$

Figure 5.5: Identity matrix

#### 5.2.2.1 Translation

This is achieved by constructing a translation matrix from the $x, y, z$ variables given in Figure 5.4, demonstrated by the function in Figure 5.2.

#### 5.2.2.2 Rotation

The rotation matrix is calculated by using quaternions, as it is done in the FFP version. We use all of the $r$ variables, as well as $sx, sy, sz$, given in Figure 5.4, for different rotations (depending on the axis).

#### 5.2.2.3 Scaling

Constructing a scaling matrix is similar to translation, except we use the $sx, sy, sz$ variables in Figure 5.4.

### 5.2.3 Colour and Material Properties

Colour is handled by setting the `color` property in the state to the colour given, and then passed into the shader as a uniform variable. Material properties have also been added for use with the different shading models when lighting is used. These are separated into two different nodes: `Color` and `Material`. The material properties are the ambient, diffuse and specular aspects of the material. This is explained in more detail in paragraph 5.2.5.2. When colour is set, it checks if there if the lighting has been set in the state, and if it has, sets all of the material properties to equal the colour. Likewise, if a lighting node is encountered, the material properties are checked to exist, and if not, sets them to be the current colour set. The colour is set to white as default.

### 5.2.4 Camera

Camera is handled in two steps: The camera view matrix, and the perspective projection matrix. I have opted to construct these matrices using the *Linear* package's functions: `lookAt` and `perspective` respectively. I replace the current matrices in the state with the newly constructed ones. There are several ways of handling this. One option is to enable support of multiple camera points, rendering everything up until the second camera node is encountered using the first one's properties, and the ones after with the second one's properties. Another option is to throw an error, as we only allow for one set of viewing properties. It is very unlikely that there will be two camera nodes in the scene graph, so I have opted to replace the matrices, and render everything in the scene graph from the second camera's perspective for sake of simplicity. If we reach a compiled node which has camera settings, we replace these as well. The projection and view matrices are also wrapped in `Maybe` types, as there may be no camera settings, in which case, it sets them to the identity matrix.

### 5.2.5 Lighting

There are several shading models for lighting, as we have highlighted earlier. We will discuss them below and describe how to achieve them using a variety of shaders. We can use the Phong lighting model (explained in section 5.2.5.2) to determine the colour at a particular point. For all of these models, we need the vertex and normal to be given as attributes from a VAO passed to the vertex shader.

#### 5.2.5.1 Normal Matrix

The normal of a vertex is used to calculate the light intensity at that vertex. To calculate the normal for a surface respective to the current space, a normal matrix is needed. Firstly, we remove the last column and row in the modelview matrix to convert it from a 4x4 matrix to a 3x3 matrix. The *Linear* package [24] is used to calculate the normal matrix from the 3x3 modelview matrix by taking the inverse of it, and then transposing it [25]. The resulting matrix and sent to the shaders.

This does not guarantee a correct normal matrix, as the modelview matrix must be orthogonal to ensure correctness, and the `Scale` node does not guarantee this. Unless the scaling is done uniformly, the modelview matrx is not orthogonal [25]. If the scaling is uniform, the lengths are affected, but are corrected when normalised. There will not be changes to rectify this, as there is not enough time nor is it's implementation deemed sufficiently important.

#### 5.2.5.2 Phong Lighting Model

The Phong lighting model is a formula used to determine the light intensity at a point given several parameters.

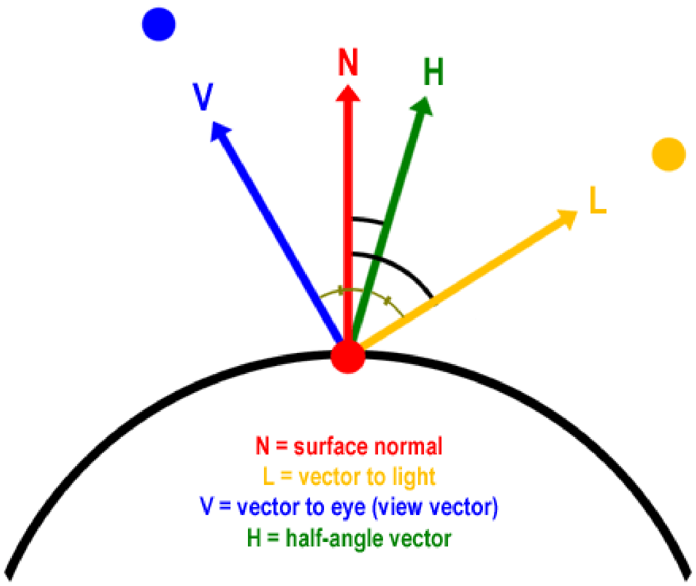


Figure 5.6: Vectors for calculating Phong lighting model [33]

In addition to the above vectors, we are also given several colour inputs:

- Light Ambiance $= l_{ambient}$
- Material Ambiance $= m_{ambient}$
- Light Diffuse $= l_{diffuse}$
- Material Diffuse $= m_{diffuse}$
- Light Specular $= l_{specular}$
- Material Specular $= m_{specular}$

The light intensity equation is:

$$I = A + D + S \tag{1}$$

Where $I, A, D$, and $S$ are the light intensity, ambiance, diffuse and specular components respectively. These are made up of the above parameters and shall be shown below:

$$A = l_{ambient} m_{ambient} \tag{2}$$

$$D = l_{diffuse} m_{diffuse} \frac{N \cdot L}{||N||||L||} \tag{3}$$

#### 5.2.5.3 Blinn Specular

We could use a reflection vector from the vector to the light to calculate the specular term, but this is expensive [33]. Instead, we can use the Blinn specular model, which uses the half vector, calculated from the vector from the surface to the light and to the eye. This is for efficiency, as well as to provide a more accurate model than one which uses the reflection vector [2]. The term is raised to a power of 0.3, an arbitrary constant given, for the point of completeness. This can be parameterised as well, but I have chosen not to do so in my implementation due to time constraints.

$$S = l_{specular} m_{specular} \left( \frac{N \cdot H}{||N||||H||} \right)^{0.3} \tag{4}$$

The lighting properties (and position) are specified when using the `Lighting` node, and the material properties are specified when using the `Material` node. The view vector is calculated in the shader, and the normal is specified as a vertex attribute. This model is used in all of the shaders specified below. There are variations of the lighting model to include a *specular highlight coefficient* (which I have as a 0.3 constant), *emitted light*, and *attenuation*, but I have not included them in my model.

As will be evident during the evaluation, the shading looks different between the FFP and shader versions. This is due to the different lighting models used. We have shown, however, how much capability using shaders for shading has over the FFP, where we are bound by one model.

#### 5.2.5.4 Flat Shading

To achieve flat shading, we calculate the light intensity in the vertex shader and specify it as a `flat out` variable to the fragment shader. This means it won't interpolate, and shall use the same light intensity for each fragment in the polygon. The light intensity is either multiplied with the interpolated texture pixel colour and outputted or outputted by itself.

#### 5.2.5.5 Smooth (Gourad) Shading

This is even simpler, as we can do the same thing as in flat shading, except we omit the `flat` property. In this case, it shall interpolate the light intensity at each pixel.

#### 5.2.5.6 Phong Shading

Phong shading is similar to Gourad shading, but, instead of interpolating the light intensity, we interpolate the normals and calculate the light intensity in the fragment shader.

#### 5.2.5.7 Exact Shading

I will not have time to implement this shader, but it is important to discuss it's capabilities. If we have a function, $f$, to determine the position of a vertex and pass that into the fragment shader, and we have two partial derivatives of $f$, giving us two tangents to the vertex surface, which in turns gives us a surface normal by taking the cross product of them. This gives us the exact normal at that particular interpolated position, which we can then use to calculate the light intensity as we did before. It is also important to note that, as there may be nonlinear variation in the region inside the polygon, exact shading gives us better approximation results than Phong.

There are a large variety of shading models, such as anisotropic and toon shading, but we will not look at these in any further detail, as the above mentioned ones are the most common.

### 5.2.6 Compiled Nodes

`DisplayLists` have been deprecated through the introduction of VAOs. This is because the information about the displayed item is retained on the GPU memory.

I have implemented compiled nodes by carrying around the state created when the `Drawable` was first traversed. If a call is made to render the scene graph, it can be assigned to a variable and rendered without knowing any of the original `Drawable` nodes. This means we can apply additional types of `Drawable` to the compiled node, and it will apply the changes to the state. When we compile a `Drawable` , the mesh data is stored in terms of the primitive mode, number of vertices per primitive, number of numbers per vertex, and the bound VAO.

```
CompiledDrawable
{
   pMode :: !PrimitiveMode
  ,vboSize :: !NumArrayIndices
  ,divSize :: !NumArrayIndices
  ,vao :: !VertexArrayObject
  ,glState :: !GLState
}
```

Figure 5.7: Data structure for holding a `CompiledDrawable`

This leads to having another data structure called `CompiledDrawable`, as shown in Figure 5.7, which is part of the `Drawable` data structure as one of the constructors. A `CompiledDrawable` is the `Drawable` that is returned when a `Drawable` is compiled, holding information about the object that is required for when drawing. It still needs to, however, have transformations and its properties changed as part of the call. This is why the state is part of it; to keep track of those changes in a new render call. When it is finally rendered, the state is queried every time to ensure the correct shader program is being used.

For example, if we compile a `Drawable` with its `Texture` node set, it would use a simple texture shader. If we then render the `Drawable` wrapped around in a `Lighting` node, then it would need to use the correct lighting and texture shader. This is reflected by changes to the state, and can therefore be reliably checked upon rendering. It can be overridden by manually setting the shader program in the state, so that it does not try to determine what shader to use.

I had an issue with the camera settings when using `CompiledDrawable`s. Previously, the view and projection matrices were not of type `Maybe`, so we could handle the camera settings of a `CompiledDrawable` in three ways: Replace the matrices with the ones set in the render stage; ignore any settings set in the render stage, and use the `CompiledDrawable`'s; or multiply them with each other. I had opted for the latter option, as this seemed most sensible, as it is unlikely that there will be two camera nodes, so they would simply be multiplied by the identity matrix. The other two have the problem of being replaced by an identity matrix. This goes against the decision I had made earlier to replace camera settings with the new one, if specified. This would have given incorrect matrices if two camera settings were specified separately in the `CompiledDrawable` and render stage. This has been rectified by wrapping these matrices in a `Maybe` constructor, and are only replaced if the values given are not `Nothing`. We follow the same principle of replacing the camera settings, if we have one for both the render stage and `CompiledDrawable`, so we use the one specified during the render stage.

### 5.2.7 Textures

Textures are handled fairly similarly to the FFP. They are set up in the same way, and the texture coordinates are given at each vertex as an attribute, and given to the fragment shader as an `out` variable. The pointer to the texture is given to the fragment shader as a `uniform` variable, which is then looked up using the `texture` function, and is returned as a `vec4` RGBA value. This can be the fragment colour, or multiplied with the light intensity to add lighting. The `TextureObject` needs to be bound before setting the `uniform` variable, so that the correct one is set, which is why it is carried in the state.

### 5.2.8 Compiling and Rendering

In the FFP implementation, we had two functions: `compile` and `renderDrawable`. This meant we separated the two stages. When moving over to shaders, I had implemented the render function, such that it **always** returned a `CompiledDrawable` when called. This let us bound the value and render that instead. One of the disadvantages of this was the difficulty in deciding whether or not the item should be rendered in this call. If it is being bound to a variable, it would make sense for it not to be rendered.

I identified two ways of handling this. The first one was to add another `Drawable` node called `Direct` which would needed to be applied as the outermost node when constructing the scene graph. When encountered during our rendering function, it would set a flag in our state to true, which would be checked after compiling the mesh. The advantage of this approach is that we have one function call, regardless of compiling or rendering. The disadvantage is that a flag has to be set, which is set to false by default, and setting the flag to true would be easy to miss out.

Ultimately, I reverted to it's original implementation, with separate functions, although there are 4 this time instead of 2. This is to account for the states, and, that an initial one may be provided. The 4 functions are `render, renderWithState, compile,` and `compileWithState`. The `compile/render` functions create a state with each matrix set to the identity matrix, and other properties set as `Nothing`, whereas the other two take in an initial state. `compileWithState` is the core function which does all the work, whereas the others simply make calls to it. The `render` function simply calls `compile` and then renders the result. Likewise for `renderWithState` calling `compileWithState`. I believe this to be the better approach, as the user is able to decide which distinct functionality they want, and there is no ambiguity.

## 5.3 Default Shaders in Library

To some degree, the library needs to be able to support the simple and common cases for rendering. I have identified 8 different default shaders which provide this functionality. These are:

- Simple shader with projection, model, view and colour changes. (1)
- Simple shader with lighting and material properties set:
  - Phong. (1)
  - Gourad. (1)
  - Flat. (1)
- All of the above lighting shaders with one texture per mesh. (3)
- Simpler shader with no lighting properties, but a texture has been set. (1)

This accounts for all the current default shaders in the library. If a certain node is used in the scene graph, then the state is changed to reflect that, and thus, when rendering, the state is checked so that the correct shader may be used. However, this does not restrain the user from using their own shaders. A `ShaderProgram` can be set by using the scene graph node `Shader`, which shall override any state changes and use the specified `ShaderProgram` instead. This assumes that the user has set all relevant attribute and uniform variables, though.

### 5.3.1 Constructing the Shaders

Initially, I had opted for files, as is the norm for shaders. The vertex and fragment shaders are held in plain text files, which can have any extension, as they are read as plain text files when compiled. This is true for all languages, and is no different to using shaders in Haskell or C.

There are, however, several problems with this approach. GLSL is written in a C-style approach, thus it is an imperative language. This means that the user has to switch between an imperative and functional writing style when programming with shaders in Haskell, which may present consistency problems. Another issue is that there is a lot of duplication between shaders. In fact, there is only one word different to Gourad and flat shading shaders. There is also the issue of variable naming. This convention leads to specifying attribute and uniform variables via strings, which can lead to bugs and inconsistency. This leads onto the final issue, which is that there is no verification that the input is a valid program at the compile stage of the main program. This means the program can fail during runtime whilst it is compiling the shader code. This is due to shaders being read in as a string, and only compiled at runtime. All of these issues can be solved by using an eDSL, or at the very least be minimised.

I have opted to create a pseudo-eDSL due to time constraints, and because it is not a requirement to build a GLSL eDSL API. I have used some code which generates GLSL code in an eDSL style for everything in a shader except functions and the main body function. This is from a library called Häte [18], which uses GLUtil [11] itself, so it is known for it's capability to generate compatible shader programs. In using this, I can maintain consistency in variable naming by assigning a variable one value and exporting it to the module that sets up the shader program, and using that when setting my attribute/uniform variable. This means that there are no inconsistencies in variable naming for my default shaders. Although inherently imperative, these modules provide a much more functional approach to writing GLSL. The issue of duplication is tackled by declaring many functions to take the place of strings provided in the main body function. I have used the tool *hlint* to ensure any such duplication is not in the module, and have thus provided very small snippets of code which can be chained together to create a shader program.

This is not unlike an eDSL, but it lacks the same versatility, as one would expect, simply because I have written neither general cases for variable assignment, nor vector concatenation. Although this is very useful for FreeD's non-external module, it is not useful as a general purpose eDSL for writing GLSL code.

There is one main issue with using an eDSL, however. If the code is modified to produce a different shader (any changes at all), it would need re-compiling before we see any changes to the outputted GLSL shader code. When using plain text external files, this is not the case, as the files are compiled at runtime, and could even be changed during runtime so that we can see the updates. As mentioned earlier, for the purpose of this library, given that these are default shaders which expect no changes, this is not a disadvantage, as these shaders would never need to be changed. In using this approach, we have also lessened the possibility of runtime shader compiling errors by standardising the calls made to produce shader code. The issue is still present when constructing the library, since we are simply procedurally generating the code using strings. But, for the purposes outside of the library for a user, this is not an issue, as it is not an external aspect. Creating an external eDSL library is also not in the scope of this project.

```
p = MedPrecision
vers = Version330
--Vertex Shader
tVert = shader vers p ins outs unifs body
  where
        ins = [vPosition, vTexCoord]
        outs = [outFTexCoord]
        unifs = [uModelViewMatrix, uProjectionMatrix]
        body = vToFTexCoord ++ glPosition

--Fragment Shader
tFrag = shader vers p ins outFColor unifs body
  where
        ins = [inFTexCoord]
        unifs = [uTexUnit]
        body = texColor ++ (fColorVar `toEqual` texColorVar)
```

Figure 5.8: Texture vertex and fragment shader eDSL code

```
//Vertex Shader
#version 330 core
precision highp float;
in vec3 vPosition;
in vec2 vTexCoord;
out vec2 fTexCoord;
uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
void main() {
fTexCoord = vTexCoord;
gl_Position = uProjectionMatrix * uModelViewMatrix * vec4(vPosition, 1.0);
}

//Fragment Shader
#version 330 core
precision highp float;
in vec2 fTexCoord;
out vec4 fColor;
uniform sampler2D uTexUnit;

void main() {
vec4 texColor = texture(uTexUnit, fTexCoord);
fColor = texColor;
}
```

Figure 5.9: Texture vertex and fragment shader output from eDSL

Figure 5.8 shows the code used to produce the vertex and fragment shaders for a textured object, and Figure 5.9 shows the output from each one, respectively. Variables are presented in the form of a list. Several parameters such as `vers` and `p` are used to give the version number of the shader, and the floating-point precision used, respectively. All shaders share this, so it makes sense to define it only once. The body takes in a string which has been given as several variables/functions. Although I did mention that it does not allow functions to be added to the GLSL output, we can have functions in the eDSL, which would be equivalent in terms of coding. This would increase the output size, as the text is being repeated, but this is abstracted from the user, and is negligent as the sizes of the produce strings are miniscule (<50).

# 6 Further Optimisation

## 6.1 Caching Default Shaders

Since our default shaders might be used very often, it is unwise to create several copies. For this, I have made a cache data structure `ShaderProgramType`, explained in section 5.1.1, which has compiled and linked the necessary shaders for our default shaders. This is carried around in our `GLState` data structure. This also means that carrying the shader program and texture object is now redundant in our `CompiledDrawable`, as we carry the state as a field of `CompiledDrawable`. The state carries the texture, and we can query against the state to determine what shader program to use. This is necessary due to possible changes in the modelview/projection matrices.

This does not negate the use of a custom shader program, because, as mentioned above, the shader program is no longer a field in the `CompiledDrawable`; it is instead carried around by the state, if set. When drawing, the state is queried to check whether one has been set, else, the default ones are used. Since the properties are at liberty to change, we have to query it every time it is drawn, but this depends upon only a few simple case statements per render pass, so it requires very little computation.

## 6.2 Further Caching Default Shaders

We had introduced a cache to optimise our program by reducing the amount of times we construct, compile, link and set the attributes for each shader. This had only proven useful where we had compiled a `Drawable`, and then called `render` on it, without wrapping it around more `Drawable` constructors. This is because when the `render` function is called, if the first node it encounters is not a `CompiledDrawable`, it will initialise and record the state until it reaches the `CompiledDrawable`, at which point it will combine the states. This is problematic, as every time we initialised the state, we constructed every default shader again. To solve this, a function called `getSPT` had been created, which traverses the scene graph until it finds a node with an initialised `ShaderProgramType`. If `getSPT` encounters a `Drawables` node, it folds over it, so we can reduce it to one variable. To enable folding, another constructor, `Nada`, had been added to `ShaderProgramType`, and it had also been made an instance of `Monoid`. The implementation of this instance is shown below.

```haskell
instance Monoid ShaderProgramType where
  mempty = Nada
  Nada `mappend` a = a
  a `mappend` _ = a
```

Figure 6.1: `ShaderProgramType Monoid` instance

This allows `ShaderProgramType` to be in a list, and for it to be flattened into one value. The `Nada` constructor is never used; it is there for completeness.

There were still issues with this approach, however. If there are no `CompiledDrawable` nodes when rendering, the rendering function will initialise the default shaders every render pass. We have solved this problem by using the `IORef` data type, which will be explained in the next section.

### 6.2.1 Accessing the Default Shaders via a Mutable Reference

An `IORef` is a data type that allows us to make a mutable reference to a variable.

```
getDefaultShadersRef :: IORef ShaderProgramType
getDefaultShadersRef = unsafePerformIO $ newIORef $ unsafePerformIO $ initSPT
```

Figure 6.2: Defining ShaderProgramType as a top-level global function

Above, we discussed the issue of having to reconstruct an identical data structure every render pass. We solve this by creating a function `getDefaultShadersRef`, shown in Figure 6.2. This function is called when we initialise the state for compiling a `Drawable`. When `getDefaultShadersRef` is first called, it initialises itself using the function `initSPT`, which builds, compiles, and links all of the default shaders, and binds them to the `ShaderProgramType`. Any subsequent calls to `getDefaultShadersRef` return the reference to this value instead of initialising it again. This means that it is only initialised **once** at runtime. It has caused our need for making the `ShaderProgramType` an instance of `Monoid` to become redundant.

## 6.3 Caching Geometry

Normally, one would compile geometry during the initialisation stage of their program, as geometry is unlikely to change. However, if the geometry isn't compiled during initialisation, but is instead rendered in the display callback, it incurred a great cost. Every time the scene graph is traversed and reaches geometry, it would have to bind itself to a VBO and VAO, as well as set the attributes. This is very expensive, in terms of computation. Using the chessboard demo in section 7.5 as an example, if the geometry is compiled, the frame rate is $\sim$ 590 FPS. Otherwise, the it is $\sim$6 FPS I have used the notion mentioned earlier of utilising `IORef` constructors to cache geometry by compiling it once, and then caching it. I used a map to store the rendered geometry. This is stored in an `IORef (Map (DGeometry, GLState) Drawable)` value. We use a tuple of `(DGeometry, GLState)` as the key to our map, so that if we have used the same geometry, we can differentiate between them by checking the state, to see if it has texture and/or lighting. The `Drawable` value in the map is of type `CompiledDrawable`. We lookup whether the key is in the map, and if it is, return the `CompiledDrawable`. Otherwise, we compile the geometry into a `CompiledDrawable`, add it to the map, and modify the `IORef` map to include the new one in, so that next time, it is looked up and returned. This has increased the frame rate of the chessboard demo from $\sim$6 FPS to $\sim$60 FPS.

It is not ideal to have the key be a tuple of `(DGeometry, GLState)`, as the geometry may be large, and when the map is traversed to search for the key, it may take a long time to differentiate between two pieces of geometry. This can be addressed by introducing an identifier linked to the `DGeometry` to use as the key. We could either use the VAO ID, which is assigned when the VAO is generated for the geometry, or introduce our own identifier. The VAO ID is unique, so this would be suitable. Due to time constraints, I have not been able to implement this, but it is important to note the flaw in this approach.

## 6.4 Caching Cube/Spheres

As we have the ability to parameterise cubes and spheres, we can specify different sizes without using the `Scale` node. Before every render pass, however, we read the OBJ file that held the primitive, deconstructed it, and used the parameters to adjust the geometric values. These are expensive to perform, as it involves an IO action. This is shown by the frame rate of the chessboard demo (∼60 FPS), given above after geometry optimisation, which is still much lower in comparison to the compiled version. We can use the same technique as above to cache the creation of a cube using `IORef` constructors. We use a three-tuple for the cube as the key, containing it's width, height, and depth, and we use the radius as the key for the sphere, so that only distinct values for these are constructed again. We then store the constructed `DObject` as the value. When coupled with the optimisation above, the frame rate rises to ∼164 FPS for our chessboard demo.

Caching objects that are read in using IO are common practice, as IO actions are very slow. This is why it is important that they are used as little as possible. Another IO action that can be cached is loading in textures from file. This has not been done, due to time constraints. Common practice is to load anything from file during the initialisation stage of a program.

# 7 Evaluation

To recap the criteria for evaluating the library, we shall be comparing 6 different methods of rendering, comparing LOC, speed (frame rate), and visual consistency. The base rates I shall be using are the *C shader-based* and *C FFP* approach respectively. In theory, these should be the fastest in terms of rendering speed. They are what OpenGL is originally implemented in, and do not have any overhead like Haskell does to use the underlying C code.

I shall also be evaluating the performance of FreeD against the number of triangles it can render, whilst still maintaining a reasonable FPS rate. A reasonable FPS rate is anything above 30, as this is the rate that eye comfortably perceives motion. This does place dependency on the model loading library, and support of loading a variety of objects of certain types and sizes. The timings shall only be considered during the rendering stage, and the object shall be both pre-loaded into a `DObject`, and compiled for our tests to compare. In addition to the original evaluation criteria, I have chosen to demonstrate that the programmer's skill is important in the usage of FreeD in section 7.4, as the order in which tasks are performed can give very different results (computationally speed-wise). I have also made a demo to showcase the ease of using FreeD, to show that it is integrated easily into a functional style of programming in section 7.5.

## 7.1 Pretext

I have run all of these demos on an ASUS 303XLA with i7 5510U processor, 6GB of RAM, and an Intel(R) HD Graphics 5500 GPU. The Operating System used is Windows 8.1. Any Haskell code is compiled using GHC 7.10. Any C/C++ code is compiled under GNU GCC.

Unless stated otherwise, the demos are performed using the latest version of FreeD (shader-based). For the C versions, I have used C++ for window context creation, but all OpenGL calls are performed in C. Frame rates are calculated using an average of 5 seconds across 5 runs to get an accurate result.

LOC is recorded in setup+rendering style.

## 7.2 Simple Textured Cube with Gourad shading

I shall be evaluating how my library copes with drawing a simple, lone, textured cube with Gourad shading.

Due to the differing nature of FFP and shader-based rendering, it would not make sense to compare them in a visual sense. This is because there are different assumptions made with each approach. Since we are effectively replacing parts of the rendering pipeline, I have not been able to capture the same lighting model that is used in the FFP, due to unknown assumptions made by the FFP. Most noticeably, the lighting is different between them. This is why we shall compare them against each other in their respective rendering forms.

Matlab was used to check the visual differences between the rendered images. The titles in the rendering windows were omitted when comparing. The difference and the sums of the two images (base image being from the C++ implementation) were calculated and then a percentage was computed from this using the following formula:
$200 * mean(diff)/mean(sum)$ [20].

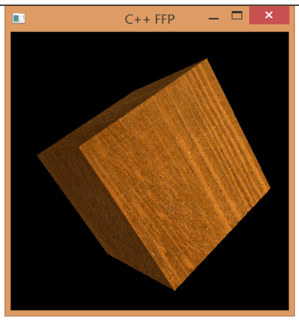
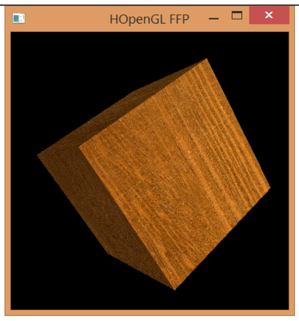
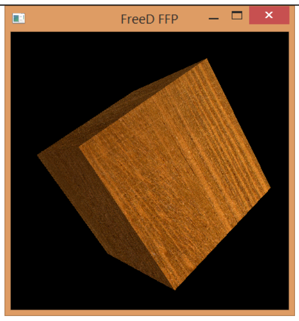
| **Rendering library** | **Language** | **Frame rate (FPS)** | **LOC** | **Image comparison** | **Visual consistency** |
|---|---|---|---|---|---|
| OpenGL [28] | C++ | 2083.3 | 23+14 |  | 0% |
| HOpenGL [30] | Haskell | 2899.2 | 50+2 |  | 1.5825% |
| FreeD | Haskell | 2527.8 | 18+1 |  | 2.7748% |

Figure 7.1: Comparison table - Cube FFP example

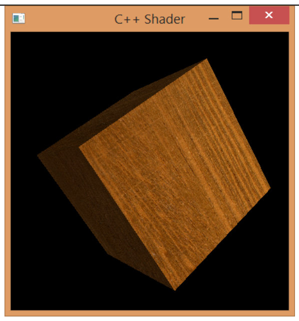
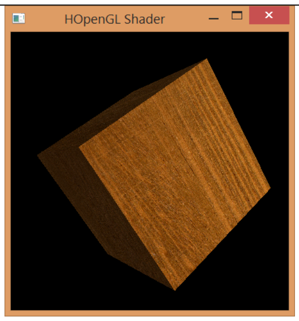
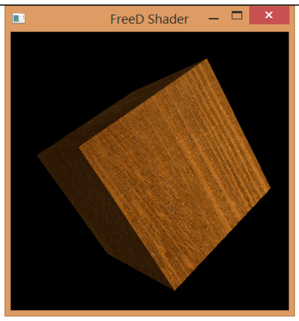
| **Rendering library** | **Language** | **Frame rate (FPS)** | **LOC** | **Image comparison** | **Visual consistency** |
|---|---|---|---|---|---|
| OpenGL [28] | C++ | 2703.4 | 48+1 |  | 0% |
| HOpenGL [30] | Haskell | 3448.4 | 107+1 |  | 1.3943% |
| FreeD | Haskell | 2749.2 | 15+1 |  | 1.3943% |

Figure 7.2: Comparison table - Cube Shader example

As can be seen in Figures 7.1 and 7.2, the two C++ implementations show that the base FPS rates do not differ much from the Haskell implementations. Due to the simple nature of the scene, the FPS of each method is high (1000+). There is large margin for error, as the most significant factor in determining the FPS, is the background applications running, causing it to range from 2000-4000 for each implementation. What we can deduce from this simple demo, is that there is no particular difference in rendering speed over the various languages for the simplest case. This means that we can build on these scenes and develop on the relationship of speed using different methods, knowing that there isn't an initial bottleneck. What is meant by an initial bottleneck is that there are no underlying differences which are apparent in the simplest of cases between the two languages, and, within Haskell, between FreeD and HOpenGL.

The visual differences are very hard to see by eye, so an image comparison tool has been utilised to compare the rendered images. As can be seen from the results, the difference is minimal. There is still a difference, though. This is to be expected through rounding errors, and the fact that the comparative functions used in HOpenGL may not directly do the same as they would in C++'s OpenGL. The interesting result is between HOpenGL and the FreeD FFP implementations, as their results differ. This is unexpected, as they are both using the same underlying calls to render, though in a different order. This can mean one of two things: either the order is different, and that has had some side effects, or the programmer has not taken care to make sure everything is like-for-like between the two programs. We also wouldn't expect a difference between the C++ and Haskell shader implementations as all of the rendering code is in the shaders, which are identical for each implementation. This can be attributed to the calculation of the normal matrix which is used to calculate the lighting. This requires an inverse of the modelview matrix, which may have different precisions depending on the libraries used. The normal and modelview matrices were calculated using the same library in both Haskell implementations, so they are identical.

The FreeD libraries require much less initialisation code. This is one of the key advantages of the library - it contains an abstraction of all the set up needed, as well as the organisation of rendering different objects with different properties.

## 7.3 Numerous Objects

We shall attempt to display many objects in one scene to stress the abstraction functionality of the library. We shall also evaluate the CPU bounds purely by stripping out any transformations, and simply drawing several copies of the same object. The shader used shall be the simplest one, so as to have the results rely on the abstraction of the library, as opposed to the shader code. There are 4056 polygons being rendered in this per object (the same object), in the same position. I have gone from having 1 object to 1024, multiplying by 2 in between each.

Here we have a table showing the decreasing frame rate (FPS) as the number of objects/triangles being rendered on the screen increases.

| **Object Count** | **Triangle count** | **Frame rate (FPS)** | **Frame rate $\times$ Object count** |
|---|---|---|---|
| 1 | 4096 | 2967.4 | 2967.4 |
| 2 | 8192 | 2583.2 | 5166.4 |
| 4 | 16384 | 1955.3 | 7821.2 |
| 8 | 32768 | 1509.9 | 12079.2 |
| 16 | 65536 | 1263.6 | 18132.8 |
| 32 | 131072 | 847.481 | 27119.392 |
| 64 | 262144 | 487.8 | 31219.2 |
| 128 | 524288 | 241.4 | 30899.2 |
| 256 | 1048576 | 133.465 | 34167.04 |
| 512 | 2097152 | 67.6482 | 34635.8784 |
| 1024 | 4194304 | 34.14 | 34959.36 |

Figure 7.3: Numerous Object Table

Here is the plot of the frame rate (FPS) against the number of triangles being rendered on screen on a logarithmic base 2 scale.

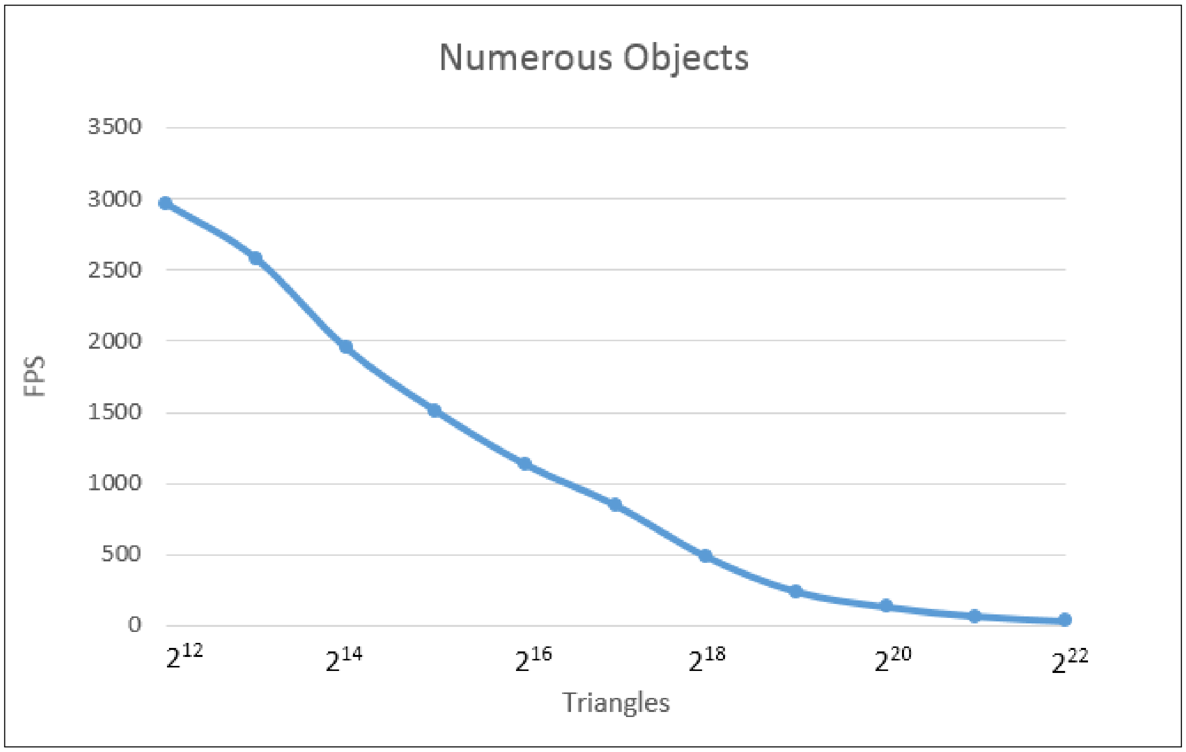


Figure 7.4: Numerous objects chart on a logarithmic base 2 scale

Figures 7.4 and 7.3 show that the number of objects has an inverse relationship with the FPS, and thus every time we double the amount of objects being drawn, the FPS drops roughly in half. This becomes more apparent as the number of objects increases, as the constant of proportionality converges to one value. The less objects being drawn, the more erratic the FPS is, and thus the averages over 25 seconds have been used for each test.

## 7.4 Efficiency in Coding

When performing my testing for section 7.3, I noticed that the initialisation of creating the `Drawable` items was incredibly slow when creating 512 or 1024 objects. I shall attempt to demonstrate the extent to which this is dependent on the type of object being replicated.

`replicate` is a function which takes in two arguments, an `Int` and any other object. It returns a list of the second argument, of the size of the first argument. That is, `replicate 5 10` will return `[10,10,10,10,10]`. I used this function to create the number of objects for testing in section 7.3.

I had ran this test before optimising to cache the geometry, so after the geometry has been compiled once, any subsequent calls to compiling the same thing simply returns the reference to the original one. This has decreased the loading time considerably, and shall be presented in the table.

```
let drawables = Drawables $ replicate 1024 obj
cd <- compile drawables
```

Figure 7.5: Replicating before compiling

```
cd <- compile c2
let drawables = Drawables $ replicate 1024 cd
```

Figure 7.6: Replicating after compiling

| **When replicated** | **Time taken (seconds)** |
|---|---|
| Before Compiling (pre-optimisation) | 113.273 |
| Before Compiling (post-optimisation) | 1.528 |
| After Compiling (pre-optimisation) | 0.38 |
| After Compiling (post-optimisation) | 0.22 |

Figure 7.7: Comparison table between compilation times

Figures 7.5 and 7.6 shows the use of the `replicate` function before and after compiling, respectively. We can see from Figure 7.7 that there is a huge difference between the results. They both return the same value, and thus render at the same speed. We can deduce from this that producing a `CompiledDrawable`, and replicating it is much more efficient than replicating the uncompiled `Drawable`, and then compiling each one. Although the optimisation made compiling much faster, if each object had transformations, the cache would not treat the geometry as the same object and would compile it again. The extent to which FreeD can be used effectively is highly dependent on the ability of the programmer using it.

## 7.5 Chessboard Demo

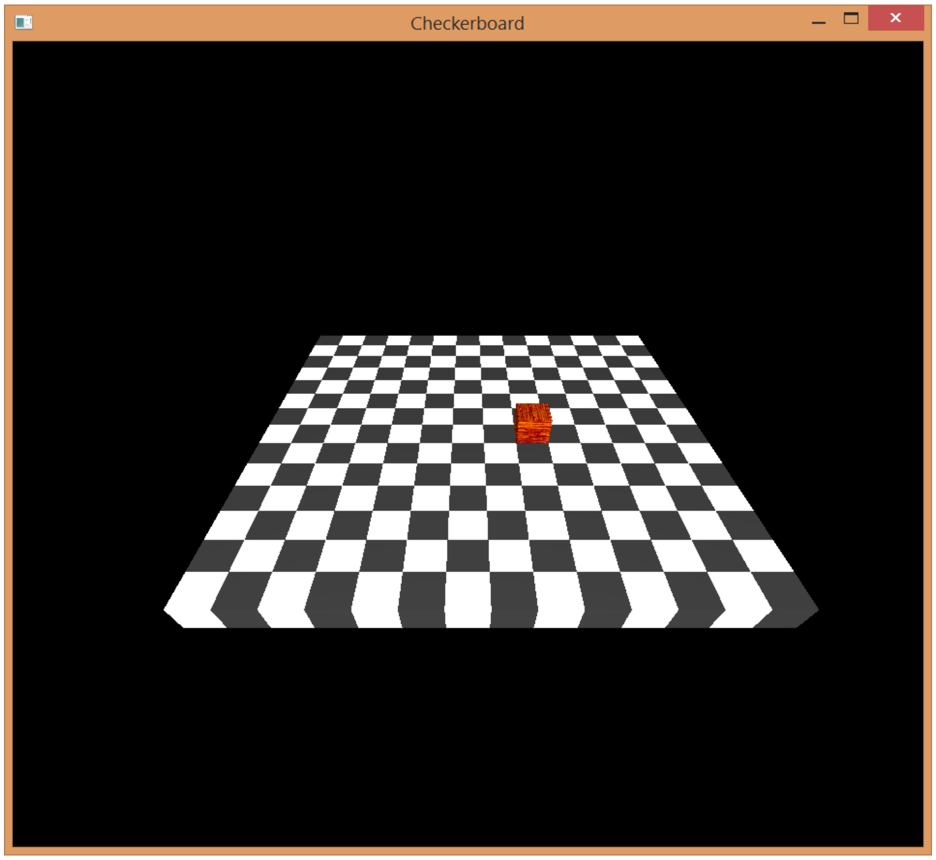


Figure 7.8: Chessboard demo with movable piece

```
buildRow n mult offset d = [Translate ((x*mult)+offset) 0 0 d |  x <- [(-n)..n]]
buildCol n mult offset d = [Translate 0 ((x*mult)+offset) 0 d |  x <- [(-n)..n]]
```

Figure 7.9: Board building code

This demo showcases how FreeD data structures can be used with conventional Haskell code to create graphic scenes. The block in Figure 7.8 above the chessboard is movable via keystrokes. `IORef` values are used in a `Translate` constructor to enable the block to move. As can be seen in Figure 7.9, building the chess board using the `Drawable` nodes is incredibly easy. `n` signifies twice the number of blocks desired; `mult` is the size disparity between each block; `offset` is the offset from the calculated position. We first build two rows of black and white cubes with alternating patterns. We then build columns from each alternate row.

## 7.6 Fizz Backend Replacement

One of the initial aims for this project was to use FreeD as a backend for Fizz [43] in terms of rendering. Here, we show that this has been accomplished to an extent. FreeD does not offer the interactivity that Fizz does, so it shall not be shown. Fizz loads in data sets from file to produce geometry. I have implemented the demo so that FreeD converts Fizz's geometry into it's own. There was another data set that I had intended on testing (144MB), but due to insufficient memory on the machine it was being tested on, this could not be achieved. Both Fizz and FreeD versions ran out of memory.

We will evaluate the performance by measuring speed and memory usage. For this, we need to enable profiling when compiling our source. The compile flags we will be using are:

```
-O2 -funbox-strict-fields
```

### 7.6.1 Profiling

We enable Haskell to compile and run using profiling options to gauge the space and time usage of the program. For this evaluation, we will also use the following profiling flags when compiling, in addition to the ones mentioned above:

```
-rtsopts -prof -auto-all
```

And at runtime:

```
 +RTS -p -hy
```

This will output a textual profile and a heap profile, which can converted to postscript and viewed graphically. The heap profiles are shown in appendix G.1 for each implementation.

There is another constraint to profiling when rendering graphics. We must perform only one render pass, and then exit the program. This is so the profiler can accurately profile the render code.

### 7.6.2 Results

We use the same method to approximate speed as other demos: by frame rate. The nodes have been compiled using their respective functions for each library, and are passed to the display callback to be rendered.

| Data set size (MB) | Rendering library | Frame rate (FPS) |
|---|---|---|
| 2.2 | Fizz | 94.43 |
|  | FreeD | 1363.4 |

Figure 7.10: FPS comparison between Fizz and FreeD

The following tables are produced from a profiling report generated when running the programs. They are limited to the rendering functions, including the setup.

| **Function** | **Time(%)** | **Allocation(%)** |
|---|---|---|
| openGraphics | 4.0 | 0.0 |
| render | 3.1 | 0.0 |
| vertUpper | 2.9 | 0.8 |
| vertLower | 2.7 | 0.8 |
| gExtent | 1.0 | 2.2 |
| vertExtent | 0.4 | 1.6 |

Figure 7.11: Fizz time and allocation, with respect to rendering

| **Function** | **Time(%)** | **Allocation(%)** |
|---|---|---|
| drawComponent | 8.6 | 6.6 |
| display | 2.6 | 0.0 |

Figure 7.12: Fizz using FreeD time and allocation, with respect to rendering

Figure 7.10 shows that, when using FreeD as the rendering engine, the speed is significantly faster, by a factor of 14.5. We also show the time and allocation percentages for both Fizz and FreeD to render one frame and then exit in Figures 7.11 and 7.12. Fizz, in total, takes 14.1% and 5.4% for time and allocation, respectively, compared to FreeD's 11.6% and 6.6%. Fizz does, however, draw an additional bounding box around the data set, which FreeD does not.

We can see, when comparing FPS, that FreeD is more efficient in terms of speed. However, Fizz uses less memory. Since FreeD's geometry data structures are the same as Fizz's, with a different outermost constructor, this does not account for the extra memory. The reason for the extra memory usage is primarily due to the fact that FreeD has to unpack, and flatten the geometry into a list of `GLfloat` values so that it can be binded to a VBO. This can be avoided by optimising the code so that additional memory storage can be kept to a minimum, by incorporating techniques known as fusion [42] and deforestation [45] [44]. These refer to the concept of removing intermediary data structures by combining functions, so that the functions applied to the values in the original geometry are combined and placed in the resulting data structure only after each function has been applied to the value. Currently, we apply a function to the geometry, place it in another data structure, and then apply another function to obtain the desires results. This has not been optimised due to time constraints.

Figures G.1 and G.2 from appendix G.1 also illustrate the differences between Fizz and FreeD. Most notably, Fizz's equivalent rendering functions are not seen, whilst FreeD's are. Using this, and the above data, we can conclude the FreeD is less memory efficient than Fizz, but is faster in terms of setup as well as rendering.

# 8 Conclusion

The aims and objectives for this project can be summarised into two points:

- A study on the current state of functional graphics in Haskell, and how I intend to improve on it.
- The implementation and evaluation of FreeD using the findings of the study.

In this section, I will explore the extent to which each of these points have been achieved.

## 8.1 Functional Graphics in Haskell

In section 2.3, I summarised the capabilities of the existing available libraries in Haskell. I have used a table (Figure 2.12) to compare the points that I thought were important to include in a functional rendering engine. Lambency [23] and LambaCube3D [22] were among some of the libraries which had as many features as FreeD, but did not implement scene graphs. The issue with FreeD, along with these libraries, is that they are experimental and in their early stages. Several of them had better model loading support and GLSL eDSLs. FreeD does not aim to be an all-encompassing library to be used for every graphics problem in Haskell. It is, however, an alternative to the current set of libraries, providing a different set of tools to solve *some* graphics problems.

## 8.2 Evaluation of Implementation

The implementation was split into three categories: core, FFP, and shader. The core involved judging the amount of abstraction between the user and the library. This was the most important aspect of the project, as it quantifies the control the user has over what is being drawn. To enable easy and quick implementations of building a scene, I abstracted the nodes and rendering calls at a very high level, not allowing the user as much flexibility in terms of camera, lighting, and texture properties. However, the user can benefit from other lower-level libraries along with FreeD to minimise any problems this may cause.

I had implemented FreeD using the FFP due to my lack of shader programming knowledge, so that I could build upon it to achieve my shader implementation. Since Gloss [10] and Fizz [43] both use this approach, it was very easy to incorporate into FreeD.

The shader implementation was originally set as an optimisation, but has since become an objective. This is because the FFP implementation has already met the original objectives. Due to the nature of the core implementation, the external data types and functions have not changed, besides optimisations which would benefit both FFP and shader version. I have not included full shader (geometry, tessellation, compute) support, as FreeD only supports vertex and fragment shaders. This does not limit the user from using other shaders in FreeD. However, there is no native support for them in terms of default shaders.

My evaluation has shown that FreeD can be used to construct scenes with varying detail, and that the abstraction layer does not hinder the visual result, nor the speed at which it is being rendered. It also shows that we can create scenes with much less code than is needed in either OpenGL in C or HOpenGL.

## 8.3 Further Work

There are many areas upon which I could improve in FreeD. Here, we will discuss the potential extensions, as well as the shortcomings of the presented solution.

### 8.3.1 Model Loading

I could add additional support for one file type, or introduce the ability to load different ones. At the moment, only OBJ files are supported, and this support is very limited. The loader does not support material files, and only has the ability to load **triangles** that have vertices, normals and texture coordinate attributes. Adding more loaders would involve integrating existing file loaders, so that they load into FreeD's geometry types, or creating our own parsers. I would like to implement the loading of scenes, as well as singular meshes. This means that we can load camera and lighting settings as well as the mesh, so that the whole scene can be read in and rendered in fewer lines of code.

### 8.3.2 Additional shader support

I would like to add default shader support for geometry, tessellation and compute shaders. The current support for vertex and fragment shaders is also lacking, in that there are many more variations of these. The current solution does not scale, as it implements every possible default shader, instead of providing the tools to build shaders using a built-in eDSL.

### 8.3.3 Bounding Box Camera Calculation

Having a bounding box is useful for determining the properties of the camera view in a scene graph. We can recursively traverse the scene graph to calculate the bounding box by using minimum and maximum functions for the x, y, z of the box on the meshes. Unlike the way in which state is carried through when traversing a scene graph for rendering, the bounding box is available as a global variable, as sibling meshes need to be aware of each other's sizes. This would be implemented using an `IORef`. The rotation and translation transformations would also be applied to the current bounding box, as they may modify the extent of the mesh. If we had a list of two meshes, both of which have had translations applied, in opposite directions, we would need to take this into account, and the bounding box would need to envelop both meshes. After calculating the bounding box, we could use it to determine the camera's position, the centre that it is looking toward, and z-depth. This would be particularly useful if a mesh of unknown size were to be loaded in, and we wanted the whole mesh to be encapsulated without tweaking the camera settings when rendering.

### 8.3.4 Generalising scene graph data structure

The `Drawable` data structure cannot be directly used in conjunction with several higher-order functions, such as `fmap`, as `Drawable` takes in explicit types in it's constructor fields. This is better suited to the FFP implementation, as many OpenGL functions require explicit types, whereas in the shader version, all of the functions are self-defined. These functions also take explicit types, but can be easily modified to take general values. There is also a tree data structure defined in Haskell, and I could have made use of this instead of creating my own data structure for the scene graph.

### 8.3.5 Unboxing Values

In my approach, I do not fully make use of bangs and unboxing values, such that there's a noticeable difference in speed and memory efficiency. I have not used the compiler directive to unbox strict values in any of the evaluations besides for Fizz. I would like to better identify areas where specifying strictness would be most appropriate, to further improve the efficiency of FreeD.

## 8.4 Methodology

I had used an agile approach towards my project, whereby several tasks were performed alongside each other. This approach worked well, since I had no knowledge of Haskell or shader programming prior to the project, which meant that I had to focus on breaking up the background research into several stages so that I could continually implement parts of FreeD. Although this approach led me to deprecate the FFP version of FreeD, it was a necessary step in order to implement the core functionality, so that I could then use it to build upon the shader implementation. I met with my supervisor at least once a week in order to share my findings and remain focused on my objectives. My supervisor would generally guide me into areas required to achieve the functionality I wanted to provide in FreeD. I would research these topics and implement them into my project. This worked well, due to the small scale deliverables which were added each week, and because they could be challenged very quickly afterwards, if deemed unsatisfactory.

## 8.5 Final Conclusion

In conclusion, I have achieved my goal of implementing a functional rendering engine in Haskell, called FreeD, which uses a scene graph data structure to construct scenes, with the use of shader support for rendering. FreeD has been evaluated in terms of performance, visual correctness, lines of code, abstraction, simplicity of use, and compatibility with higher-order functions. It has also been used as a backend rendering engine for Fizz, highlighted as one of the original aims, and has provided a much faster way of rendering than Fizz's current method. The evaluation of each of these points has verified that FreeD has met the aim of delivering a library which can be used to render 3D graphics, using shader programming in a functional, high-level way.

## References

[1] Bitbucket. https://bitbucket.org/. Accessed: 13/05/2015.

[2] Blinn-phong shading model. http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model. Accessed: 13/05/2015.

[3] collada-output. https://hackage.haskell.org/package/collada-output. Accessed: 13/05/2015.

[4] collada-types. https://hackage.haskell.org/package/collada-types. Accessed: 13/05/2015.

[5] Display list. http://en.wikipedia.org/wiki/Display_list. Accessed: 13/05/2015.

[6] Dummy wood. http://www.turbosquid.com/3d-models/free-obj-mode-dummy/662719. Accessed: 13/05/2015.

[7] The end of fixed-function rendering pipelines. http://gamedevelopment.tutsplus.com/articles/the-end-of-fixed-function-rendering-pipelines-and-how-to-move-on--cms-21469. Accessed: 13/05/2015.

[8] fwgl. https://hackage.haskell.org/package/fwgl. Accessed: 13/05/2015.

[9] Git. http://git-scm.com/. Accessed: 13/05/2015.

[10] Gloss. https://hackage.haskell.org/package/gloss. Accessed: 13/05/2015.

[11] Glutil. https://hackage.haskell.org/package/GLUtil-0.8.5. Accessed: 13/05/2015.

[12] Gpipe. https://wiki.haskell.org/GPipe. Accessed: 13/05/2015.

[13] Gpipe-collada. https://hackage.haskell.org/package/GPipe-Collada. Accessed: 13/05/2015.

[14] graphics-format-colllada. https://hackage.haskell.org/package/graphics-formats-collada. Accessed: 13/05/2015.

[15] Graphics pipeline. http://en.wikipedia.org/wiki/Graphics_pipeline. Accessed: 13/05/2015.

[16] hackage. https://hackage.haskell.org/. Accessed: 13/05/2015.

[17] Haskell bindings for glut. https://hackage.haskell.org/package/GLUT. Accessed: 13/05/2015.

[18] Hate.
https://github.com/bananu7/Hate/blob/master/src/Hate/Graphics/Shader.hs.
Accessed: 13/05/2015.

[19] helm. http://helm-engine.org/. Accessed: 13/05/2015.

[20] Image difference. http://uk.mathworks.com/matlabcentral/answers/
10960-how-to-measure-intensity-difference-between-2-image. Accessed:
13/05/2015.

[21] Introduction to shaders. http://nomone.com/2014/05/introduction-to-shaders/.
Accessed: 13/05/2015.

[22] Lambacube 3d. https://lambdacube3d.wordpress.com/. Accessed: 13/05/2015.

[23] Lambency. https://github.com/Mokosha/Lambency. Accessed: 13/05/2015.

[24] Linear. http://hackage.haskell.org/package/linear-1.18.0.1. Accessed:
13/05/2015.

[25] Normal matrix.
http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/.
Accessed: 13/05/2015.

[26] not-gloss. https://hackage.haskell.org/package/not-gloss. Accessed:
13/05/2015.

[27] obj. https://hackage.haskell.org/package/obj. Accessed: 13/05/2015.

[28] Opengl. https://www.opengl.org/. Accessed: 13/05/2015.

[29] open.gl. https://open.gl. Accessed: 13/05/2015.

[30] Opengl-2.11.1.0. http://hackage.haskell.org/package/OpenGL-2.11.1.0.
Accessed: 13/05/2015.

[31] Opengl shading language.
http://en.wikipedia.org/wiki/OpenGL_Shading_Language. Accessed:
13/05/2015.

[32] Opengl tutorial. http:
//www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/.
Accessed: 13/05/2015.

[33] Phong for dummies.
http://www.gameprogrammer.net/delphi3dArchive/phongfordummies.htm.
Accessed: 13/05/2015.

[34] ply-loader. https://hackage.haskell.org/package/ply-loader. Accessed:
13/05/2015.

[35] Realtime collision detection.
http://realtimecollisiondetection.net/blog/?p=12. Accessed: 13/05/2015.

[36] Scenegraph. https://wiki.haskell.org/SceneGraph. Accessed: 13/05/2015.

[37] Shader. http://en.wikipedia.org/wiki/Shader. Accessed: 13/05/2015.

[38] Silicon graphics. http://www.sgi.com/. Accessed: 13/05/2015.

[39] Vertex buffer object. http://en.wikipedia.org/wiki/Vertex_Buffer_Object. Accessed: 13/05/2015.

[40] Weak head normal form. http://stackoverflow.com/questions/6872898/haskell-what-is-weak-head-normal-form. Accessed: 13/05/2015.

[41] Mike Bailey and Steve Cunningham. *Graphic Shaders: Theory and Practice*. CRC Press, 2012.

[42] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP'07*, 2007.

[43] M.Wallace C.Runciman D.J. Duke, R. Borgo. Huge data but small programs: Visualization design via multiple embedded dsls. Technical report, University of Leeds/University of York, 2009.

[44] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. pages 223–232. ACM Press, 1993.

[45] Andrew John Gill and Andrew John Gill. Cheap deforestation for non-strict functional languages, 1996.

[46] Paul Hudak. *The Haskell School of Expression Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[47] Miran Lipovača. *Learn You a Haskell for Great Good*. No Starch Press, 2011.

[48] Frasier Murray. Obj-reader. https://github.com/intolerable/obj-reader. Accessed: 13/05/2015.

[49] Patrick Cozzi & Christophe Riccio. *OpenGL Insights*. CRC Press, 2012.

[50] Josie Wernecke. *The Inventor Mentor*. Addison Wesley, 1994.

# Appendices

## A Personal Reflection

There were several areas of this report which I found more difficult than initially expected. These are highlighted below.

Soon after I started the project, I was tasked with defining it's scope. I found this particularly difficult, as I had no experience with many of the concepts that this project required. With the advice of my supervisor, I managed to come up with a scope that was deemed feasible and accurate.

The writing of my report was iterative, as I wrote alongside my implementation. This led to a sporadic layout of the report, which I found difficult to rectify once I had started. The main issue I found when compiling my report, was that I had discussed topics which hadn't been introduced, or repeated myself in another section. A solution would've been to have separate documents for each of my discussed sections, and then to discuss with my supervisor the best approach to lay them out, and where to introduce certain concepts.

Implementing interaction and a robust eDSL were tasks that I struggled with, as I did not have sufficient time. They were not part of the objectives, but they would have provided FreeD with much more extensibility. In Figure 2.12, we compared FreeD's features to several other libraries, and it lacked interaction where all others didn't. Every other library with shader support implemented their own eDSL, several of which could be used externally to build shaders to use alongside their libraries. This is an aspect which FreeD lacked, in that it could only build shaders to produce the preset ones. These features were not implemented mainly due to time constraints, and lack of experience with Haskell and shader programming.

There were many times that I took a wrong approach, which had adverse consequences to the project. In these instances, I opted to delete all the related code, and start again from scratch. This stopped me from trying to save code that was designed inefficiently, and consequently eliminated the time and effort which would've been required to rectify the problem.

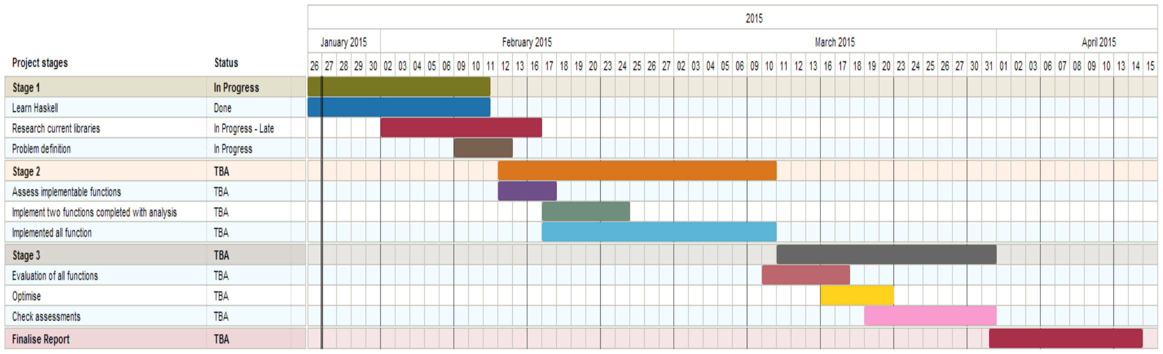
Much of my time has been spent studying core topics, so that I could comfortably use, or implement them in my solution. The two topics I spent the majority of my time studying were Haskell and shader programming. I studied Haskell prior to any coding, as I had no knowledge of it, or any other functional programming languages before starting this project. I had only knowledge of using OpenGL, with the FFP in C. This meant that I had to learn how to use the FFP in Haskell, and then shader programming in both Haskell and C, so that I could compare them. This, I feel, was the biggest challenge during my project.

In learning Haskell, I have a newfound respect for functional programming languages, and strongly encourage any programmer to learn one. It has given me an entirely different outlook on programming, and I look forward to using it for many more projects to come.

# B Methodology

## B.1 Initial Schedule

Here, I have included my initial schedule when drafting my scoping and planning document.



| Project stages | Status |
|---|---|
| Stage 1 | In Progress |
| Learn Haskell | Done |
| Research current libraries | In Progress - Late |
| Problem definition | In Progress |
| Stage 2 | TBA |
| Assess implementable functions | TBA |
| Implement two functions completed with analysis | TBA |
| Implemented all function | TBA |
| Stage 3 | TBA |
| Evaluation of all functions | TBA |
| Optimise | TBA |
| Check assessments | TBA |
| Finalise Report | TBA |

Figure B.1: Gantt Chart of initial project schedule

I largely stuck to my planned schedule, up until the evaluation and finalising my report. I began my evaluation on time, but it took much longer than initially anticipated. This is largely due to the fact that I had to implement several of the same programs with completely different approaches, shown in section 7.2. I had no prior knowledge of shader programming in C, or several libraries which allowed me to use the same geometric model and texture as the Haskell implementations. Since much of the report relied on the completion of the evaluation, it had pushed back the writing of the report, and I had less time to spend on it, leading me to complete it much later. Since I had given myself a 3 week grace period, I still had ample time to finish the evaluation and finalise my report.

## B.2 Version Control

I have opted to use version control software to maintain the source code for my project as well as my report. I have used BitBucket [1] as my choice of server repository and git [9] as my choice of version control software. I was also able to use it for my report, because it is written in LaTeX, a document markup language, which stores the report in plain text and can be typesetted to produce a PDF report.

# C Record of External Materials Used

## C.1 Data

The following OBJ file was used:

- Cube [32]

The following texture file was used:

- Dummy Wood [6]

## C.2 Code

Aside from the Haskell base library, and libraries found on hackage [16], the following external libraries were used:

- **Fizz** - Used as a basis for FreeD [43]
- **Häte** - Used as a basis for GLSL eDSL [18]
- **Obj-reader** - Written by Frasier Murray to load OBJ files in [48]

These are all open-source and allow re-use of code as per their licences, provided they are given due credit.

## D How Ethical Issues were Dealt With

No ethical issues arose during my project.

## E List of Acronyms

- VBO - Vertex Buffer Object
- VAO - Vertex Array Object
- CPU - Central Processing Unit
- GPU - Graphics Processing Unit
- FPS - Frames Per Second
- LOC - Lines of Code
- OOP - Object-Oriented Programming
- RGBA - Red Green Blue Alpha
- LOD - Level of Detail
- GLSL - OpenGL Shading Language
- DSL - Domain-Specific Language
- eDSL - Embedded Domain-Specific Language
- FFP - Fixed-Function Pipeline
- FRP - Functional Reactive Programming

# F Examples

I have included several examples, simply for reference, and further explanation of certain concepts and libraries.

## F.1 Tree from Gloss-examples

Firstly, we will look at the *Tree* example from Gloss-examples. This is a simple use of tree fractals. A `Picture` is returned from the main function. This consists of a `tree` defined in the main file. It is a recursive function, which sprouts more trees from the initial position, to form a tree structure. This is achieved by giving each `Picture` a list of `Pictures`, which are its children, and are therefore rendered as part of the parent `Picture`'s transformations. The base `Picture` is a polygon, which takes coordinates several x-y coordinates as it's arguments. A polygon is rendered primitively using "Polygon" or "LineLoop" (depending on whether it's a wireframe polygon, or a solid) in the OpenGL package.

## F.2 Interpolation

The primitive mode used for the example in Figure 2.17 is *Triangles*, so that the values are only interpolated when there are enough vertices to complete one primitive; in this case it's 3. To demonstrate this further, we choose an arbitrary point: (0,0) for simplicity's sake. The range of our example is -0.5 to 0.5 and thus (0,0) is the midway point by way of interpolating at $t = 0.5$ between $A$ and $C$. Thus
$0.5 \times (-0.5, 0.5) + 0.5 \times (0.5, -0.5) = (0, 0)$. We also interpolate the colours at these respective vertices, where, at $A$, we have (1,0,0,1), and at $C$, we have (0,0,1,1), we calculate $0.5 \times (1, 0, 0, 1) + 0.5 \times (0, 0, 1, 1) = (0.5, 0, 0.5, 1)$, which is purple, as depicted on the screen. This is a very simple, yet powerful concept of graphics.

We could have also specified for the colour to be `flat,` which would force it not to interpolate. This works by taking the last given value in vertex shader for both primitives, and using that as the colour, so that we end up with two distinct blue and red triangles, as they are given in order of ABC and CDA.

## F.3 Element Buffer Object

We can record the vertex position once in the VBO, and specify the index of the vertex position in the VBO using an element buffer object.

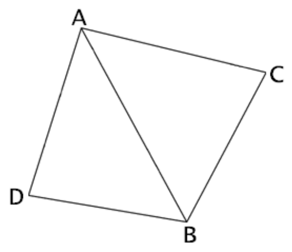


Figure F.1: Adjacent triangles [35]

As can be seen in Figure F.1, *A* has a shared vertex between the two triangles. If we used a VBO by itself, we would store the vertices as `[A, D, B, A, B, C]`, but if we instead used an element buffer object, the VBO would be `[A, B, C, D]` and the element buffer object would be `[0, 3, 1, 0, 1, 2]`.

# G Figures

This section contains figures not present in the main report.

## G.1 Fizz Comparison Figures

The following figures and tables are in reference to section 7.6, using the dataset of size 2.2MB.

Here we have the memory usage, separated by functions for both libraries.

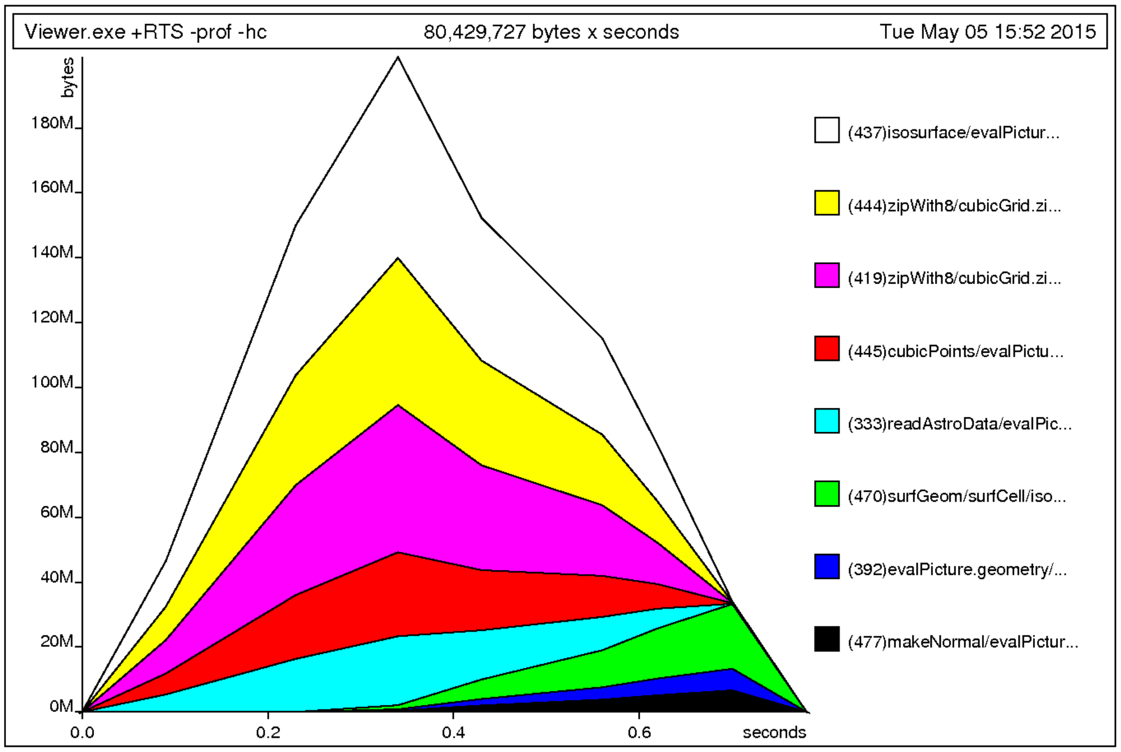


Figure G.1: Fizz memory usage

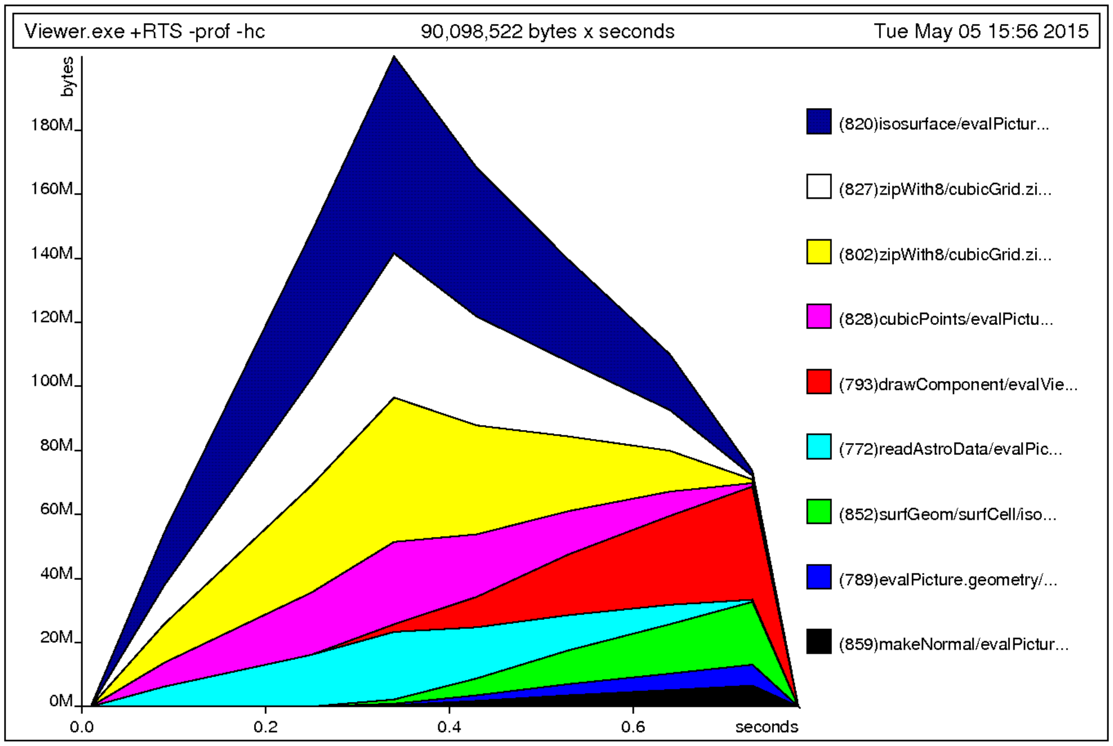


Figure G.2: Fizz memory usage using FreeD to render

Also in the report, is the total time to execute and bytes allocated during execution:

**Fizz**:

Total time taken to execute = 0.77 secs (769 ticks @ 1000 us, 1 processor)
Total amount of memory allocated = 1,021,180,808 bytes (excludes profiling overheads)

**FreeD**:

Total time taken to execute = 0.87 secs (872 ticks @ 1000 us, 1 processor)
Total amount of memory allocated = 1,033,259,720 bytes (excludes profiling overheads)

### G.1.1 Summary

From our analysis in section 7.6, we can deduce that the increased memory usage is from one of FreeD's functions. It was also shown that FreeD was quicker, but as we can see, it has taken 0.1 seconds longer than Fizz to execute. As the data set was so small, we can assume that the main factor in determining speed are background applications to explain this discrepancy.

# H Code Source, Documentation, and Module Structure

## H.1 Source Code

The source code for FreeD, including the previous FFP version, and all demos are included at: `https://bitbucket.org/Gentatsu/functional-rendering-engine-in-haskell-freed/overview`

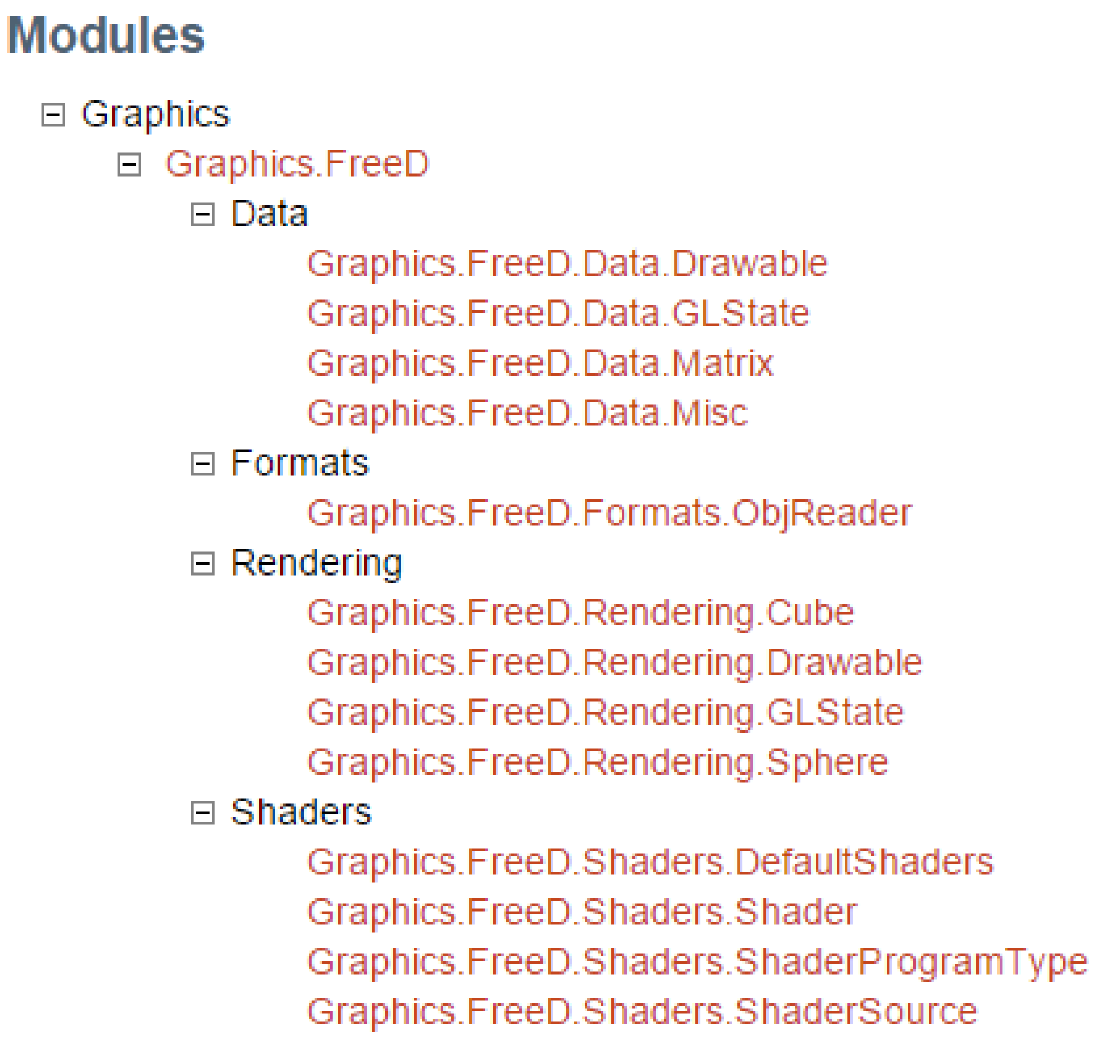
The zip for just FreeD can be found at: `http://onibaku.co.uk/FreeD.zip`

## H.2 Documentation

The documentation is held here: `http://onibaku.co.uk/freed/`

## H.3 Module Structure

In this section, I explain the module structure, and the reasoning behind it.



Modules

- Graphics
  - Graphics.FreeD
    - Data
      - Graphics.FreeD.Data.Drawable
      - Graphics.FreeD.Data.GLState
      - Graphics.FreeD.Data.Matrix
      - Graphics.FreeD.Data.Misc
    - Formats
      - Graphics.FreeD.Formats.ObjReader
    - Rendering
      - Graphics.FreeD.Rendering.Cube
      - Graphics.FreeD.Rendering.Drawable
      - Graphics.FreeD.Rendering.GLState
      - Graphics.FreeD.Rendering.Sphere
    - Shaders
      - Graphics.FreeD.Shaders.DefaultShaders
      - Graphics.FreeD.Shaders.Shader
      - Graphics.FreeD.Shaders.ShaderProgramType
      - Graphics.FreeD.Shaders.ShaderSource

Figure H.1: Module structure for FreeD

I have taken a hierarchal approach to structuring my modules, as several had common functionality, so it made sense to group them together. The library is inside a *Graphics* folder, as it is a graphics library. This helps identify straight away that FreeD is used for graphics purposes on hackage. I have 4 sub-folders: Data, Formats, Rendering, and Shaders. Files in the *Data* folder are used to define the data structures and simple functions for each of my data types in FreeD, except for shaders. Files in the *Formats* folder are for parsing different formats for loading in geometric data. Files in the *Rendering* folder are for functions that are used for rendering, which are related to the data type. Files in the *Shader* folder are for any files that are related to shaders.

We now go through each module and explain the functions and data structures defined in them:

- **FreeD** - Used as a convenience module to export relevant modules, so that the user only has to import this module.
- *Data*
  - **Drawable** - Data structure for `Drawable` and instance functions.
  - **GLState** - Data structure for `GLState` and initialisation function for `GLState`.
  - **Matrix** - Constructor functions for the matrix type used in FreeD (`M44`).
  - **Misc** - Data structures for `Material`, `DView`, `DLighting`, `ShadingModel`, `DGeometry` and loading function for `TextureObject`
- *Formats*
  - **ObjReader** - Parser for OBJ files to load into `DGeometry` type.
- *Rendering*
  - **Cube** - Functions for rendering a `Cube`
  - **Drawable** - Render and compile functions for a `Drawable`.
  - **GLState** - Functions for modifying the `GLState` when rendering.
  - **Sphere** - Functions for rendering a `Sphere`
- *Shaders*
  - **DefaultShaders** - Functions for initialising shaders, and setting uniform variables.
  - **Shader** - Slightly modified code from Häte [18]. Data structures for shader eDSL and function to build shader code in the form of strings from input variables.
  - **ShaderProgramType** - Data structure for `ShaderProgramType` and initialiser function.
  - **ShaderSource** - Functions to build default shader code in the form of strings.