HaGPipe
Programming the graphics pipeline in Haskell

Tobias Bexelius (tobias.bexelius@hotmail.com)
Mälardalen University

June 14, 2009
Abstract

In this paper I present the domain specific language HaGPipe for graphics programming in Haskell. HaGPipe has a clean, purely functional and strongly typed interface and targets the whole graphics pipeline including the programmable shaders of the GPU. It can be extended for use with various backends and this paper provides two different ones. The first one generates vertex and fragment shaders in Cg for the GPU, and the second one generates vertex shader code for the SPUs on PlayStation 3. I will demonstrate HaGPipe’s many capabilities of producing optimized code, including an extensible rewrite rule framework, automatic packing of vertex data, common sub expression elimination and both automatic basic block level vectorization and loop vectorization through the use of structures of arrays.
Contents

1 Introduction 2
2 Haskell 101 5
3 The pipeline 11
4 Programs as abstract syntax trees 13
5 General numerics 18
6 The expression types 21
7 Rewrite rules 23
8 Border control 27
9 Post pass 31
10 Wrapping it all up 35
   10.1 Cg vertex and fragment shader backend 36
   10.2 SPU vertex shader backend 36
11 Results & conclusions 39
12 Discussion & future work 42
13 Related work 44
14 Acknowledgments 45
Chapter 1

Introduction

3D-graphics is a computationally heavy duty; in the current state of the art a graphics pipeline model shown in figure 1.1 is used in which a stream of 3D objects represented as triangles are transformed and rasterized into pixel fragments, which gets lit and merged into a 2D image\(^1\) shown on the screen.

![Figure 1.1: The graphics pipeline.](image)

Since no global state is updated, this process is highly parallelizable, and this is why most personal computers have a special purpose GPU (Graphics Processing Unit) dedicated for this task. For the last decade, the vertex and fragment shader stages has become programmable,\(^2\) providing a highly customizable graphics pipeline.

All triangles sent into the pipeline are represented by three vertices where

---

\(^1\)This is called the front buffer.

\(^2\)Newer hardware also have a programmable geometry shader stage between the vertex shader and the rasterizer that operates on whole triangles.
each vertex have a position in 3D and usually some additional attributes, such as normal vectors, texture coordinates and color values. The vertex shader is a function that (in parallel) transforms each incoming vertex with attribute data into a new vertex with 3D-positions in canonical view space (i.e. where $x, y, z \in [-1, 1]$ will be visible in the resulting frame) and possibly some altered attribute data.

The rasterizer uses the transformed positions of each triangle’s vertices to generate pixel fragments for all pixels covered by the triangle in the resulting image. The vertices attribute data outputted from the vertex shader will also get interpolated for all pixel fragments over the triangle.

The fragment shader is then executed (in parallel) for each generated pixel fragment and computes each fragment’s final color and depth value. The depth values can be used by the output merger to filter out fragments hidden by other triangle fragments.

Two major APIs (Application Programming Interfaces) have evolved for the task of managing and programming GPUs: OpenGL and DirectX. For the programmable shader stages, both APIs have incorporated their own shader languages; OpenGL has GLSL and DirectX has HLSL. The graphics card manufacturer NVidia has also created Cg, a shading language supported by both OpenGL and DirectX. Such shading languages are examples of domain specific languages, as opposed to general purpose languages such as C++, Java and Haskell.

When programming the graphics pipeline using one of the APIs above, the common workflow consists of first writing vertex and fragment shaders in one of the supported shader languages, and then writing a “host” program (usually in C or C++) that loads and uses those shaders as well as sets up the other non-programmable stages of the pipeline. In this paper I suggest another approach, and present HaGPipe\(^3\), a DSL (Domain Specific Language) where we program the graphics pipeline as a whole. The difference between HaGPipe and the shader languages described earlier is that HaGPipe is embedded in Haskell, a functional language for general purpose programming with several interesting properties. A listing of Haskell’s many features are provided in the next chapter for those of you that haven’t had the opportunity to try this amazing language out yet.

The goal of HaGPipe for the purpose of this paper is to generate the vertex and fragment shader programs. I will provide backends for generating Cg-code and for generating C++-code for the SPUs (Synergetic Processing Units, the coprocessors) of PlayStation 3. HaGPipe is implemented as a Haskell library that may be imported by a programmer in order to write shaders in Haskell. The programmer won’t even need to define what backend should be used. Instead, a third party may use the shader written with HaGPipe to generate code for several different backends. The actual code generation will occur when we run the Haskell program in which the backend’s generator function is applied to the HaGPipe shader. The produced code may then be compiled by a third party shader compiler and used in a host program. If the host program is written in Haskell, there is also the opportunity to embed the HaGPipe code directly into the host program, in order to generate and compile the shaders at the same time.

\(^3\)“HaGPipe” is an abbreviation of Haskell graphics pipeline, but also reminiscent to “bag-pipe”, a common instrument in the hometown of GHC, the Glasgow Haskell Compiler.
as the host program that will use them is being compiled. I’ll briefly discuss this and other possible additions to HaGPipe in chapter 12. Instead, I’ll let the main focus of this paper be the process of transforming HaGPipe into optimized shader code.
Chapter 2

Haskell 101

Haskell [16] [10] [9] has shown to be an ideal host language for EDSLs (Embedded Domain Specific Languages) [6] [8] [18], and there is a wide range of different EDSLs implemented in Haskell already, even for graphics programming. Haskell is a deep language, and takes some time to fully understand. The purpose of this chapter is not to give a complete tutorial of the language, but rather to demonstrate the benefits of using it in the HaGPipe project, and to explain its rather uncommon syntax.

Haskell is a functional language, and as such the use of functions is emphasized. The syntax for applying functions in Haskell is very concise; actually a separating whitespace is used for function application. Instead of writing \( f(x,y) \) as is common in other languages, we simply write \( f \ x \ y \). Another syntactic feature of Haskell is that almost every non-alphanumeric character string may be used as an operator, e.g. `++`, `>>=` or `:> >`. All operators in Haskell are binary, except for the unary minus (\(-\)) operator. Any function may be used as an operator by enclosing it in ampersands, e.g. \( x 'f' y \) is equal to \( f \ x \ y \). Dually, any operator may be used as a standard prefix function by enclosing it in parentheses, e.g. \( (+) \ x \ y \) is equal to \( x + y \).

Haskell has many interesting, and in some cases unique, features:

**Purity.** The result of a function in Haskell is only defined on its input; given the same input twice, a function will always return the same output. This makes expressions in Haskell referentially transparent, i.e. it never matters if two identical expressions share the same memory or not. As a consequence, the sub expression \( f \ a \) in the expression \( g (f \ a) (f \ a) \) only needs to be evaluated once, and the result can safely be reused for both parameters of the function \( g \).

**Immutable data.** All data in Haskell is persistent, i.e. never changes once created. All functions operating on data in Haskell will always return a new version of the data, leaving the old version intact. Without mutable data, Haskell has no for- or while-loops, and recursion is the only way to create loops.

---

1Another common abbreviation is DSEL, Domain Specific Embedded Languages.
2See chapter 13.
Laziness. Since pure functions are only defined on their input, and since all data is persistent, it doesn’t matter when an expression is evaluated. This makes it possible to delay the evaluation of a value until it’s needed, a strategy called lazy evaluation. This will optimize away unneeded computations, as well as make it possible to define infinite data structures without running out of memory.

Higher order functions. In Haskell, functions are first class values and may be passed as arguments to or returned as results from other functions. In fact, a function taking two parameters is actually a function taking one parameter, returning a function taking another parameter, that in turn returns the resulting value. You could partially evaluate a function in Haskell by supplying only some of the left-most arguments, a technique called “currying”\(^3\). With operators both operands may be curried, e.g. \((4/)\) is a function taking one parameter that returns 4 divided by that parameter, and \((/4)\) is a function also taking one parameter but returns the parameter divided by 4.

Closures. Creating functions in Haskell is easy: you could for instance use a function to create another function, curry a function with one or more parameters, or use a lambda expression. With lambda expressions, an anonymous function can be defined at the same time it’s used, which for instance is useful when you need to define a function to be used as argument to another function. A lambda expression can be used whereever a normal function name can be used. An example of a lambda expression is \((\lambda a \ b \rightarrow \text{abs } a + \text{abs } b)\) which is a function taking two arguments, \(a\) and \(b\), that returns the sum of their absolute values. Common for all these ways of creating functions is that they may contain variables from the surrounding scope, which requires those values to be retained in memory for as long as the created function may be used. The retaining of a value in a function is called a closure, and is a feature not only found in Haskell, but most languages with automatic memory management, such as C# and Java.

Strong polymorphic types with type inference. All data types in Haskell are statically determined at compile time, and types can also be parameterized, like templates in C++ or generics in Java and .Net. Haskell incorporates type inference, which means that the type of an expression or variable can be deduced from its context by the compiler so the programmer won’t have to explicitly write it. This is not the same as dynamic typing where expression has no type at all; in Haskell the type checker will generate an error if no unambiguous type could be deduced from the context.

Data types in Haskell are declared with this syntax:

```haskell
data BinaryTree a = Branch (BinaryTree a) (BinaryTree a) | Leaf a
```

In this example, `BinaryTree a` is a parameterized type with two data constructors, `Branch` and `Leaf`, which holds different kind of member data. All

\(^3\)After the logician Haskell B. Curry.
type names and their data constructors must start with an upper case character, and a data constructor can have the same name as its type. Functions on the other hand must start with a lower case character. **BinaryTree** is called a type constructor, since it’s expecting a type parameter \( a \).

A **Branch** has two **BinaryTree**’s (with the same parameter type \( a \)) and the **Leaf** has a single value of type \( a \). A **BinaryTree** value has the form of one of these constructors, i.e. either it’s a branch with two sub trees, or it’s a leaf with a value. A type could have any number of constructors.\(^4\) A data constructor could be seen as a function only that it won’t get evaluated into some other value. A binary operator could be used as a type or data constructor if it starts with a colon (\( : \)), such as \( :> \) as we’ll see later on in this paper. The colon operator itself is a data constructor in the Haskell list type that concatenates a list element with a trailing sub list.

Some built in data types that are commonly used in Haskell are lists and tuples. Lists can contain a variable number of elements of one type and tuples can contain a specific number of elements of different types. A list type is written by enclosing the element type inside square brackets (e.g. \([\text{Int}]\)) and a tuple is written as a comma separated sequence of types enclosed by normal parantheses (e.g. \((\text{String},\text{Int},\text{Int})\)).

Deconstructing a data type is done using pattern matching, as in the following example:

\[
\begin{align*}
\text{count} \ x \ (\text{Branch} \ y \ z) &= \text{count} \ x \ y + \text{count} \ x \ z \\
\text{count} \ x \ (\text{Leaf} \ y) \ | \ x == y &= 1 \\
| \ otherwise &= 0 
\end{align*}
\]

In this example, the function **count** is a function taking two parameters that counts the number of occurrences of a specific element (supplied in the first parameter) in a **BinaryTree** (provided as second argument). It has two definitions where the first matches the constructor **Branch** for the second parameter, and binds the variable \( x \) for the first parameter and the variables \( y \) and \( z \) for the **Branch**’s data members (the sub trees). The second definition matches the constructor **Leaf** on the second parameter instead, and also has two guarded paths. The first path is taken if the variables \( x \) and \( y \) are equal, otherwise the second path is taken. **otherwise** is actually just a constant with the value **True**. Function patterns are tried in order from the top down, and the definition of the first matching pattern with a passed guard expression will be the result of the function application. Since Haskell is strongly typed, all definitions of a function must have matching types. In this case for example, the second parameter must always be a **BinaryTree** \( a \).

The last example also highlights another syntactic feature of Haskell, layout rules. In Haskell, the indentation actually has a syntactic meaning and is used instead of a lot of curly braces (\{\}) to group statements together, as could be seen in other languages. This is why the second guarded line was intended to match the previous one. One big advantage with the layout rule is that the programmer is forced to write well formatted code.

\(^4\)In fact, a type may even have no constructors, but in this case no values can have this type. Such a type may only be used as a type parameter in a phantom type, explained later in this chapter.
Type synonyms could be declared with the `type` keyword as in this example:

```haskell
type BinaryStringTree = BinaryTree String
```

The type of a function is expressed with `->`. For example, the type of the function `count` in our example above will be inferred to `a -> BinaryTree a -> Int`, i.e. a function taking a value of type `a`, a `BinaryTree a` and returns an integer value. Actually, this is only partly true. The actual type would be `Eq a => a -> BinaryTree a -> Int`, saying that `a` must be an instance of the type class `Eq`. The `=>` keyword is used to denote a context for a type. The type variables (e.g. `a` in our previous example) can be restricted to belong to specific type classes by stating this on the left side of `=>`. Type classes are used in Haskell to share interface between different data types, and is similar to interfaces in Java or .Net. Type classes are actually the only way to overload a function or operator for different data types in the same scope in Haskell. The declaration of a type class looks like this:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

This class has one type variable `a` and two operators `==` and `/=` that all instances of this class must implement. The `::` keyword is used to denote the type of a function or expression in Haskell. A type class is implemented by this syntax:

```haskell
instance Eq (BinaryTree a) where
  Branch x y == Branch z w = x == z && y == w
  Leaf x == Leaf y = x == y
  _ == _ = False
  x /= y = not (x == y)
```

In this example we also see the use of pattern wildcards `_` that is matching any expression and not bound to any variables. In an extension to standard Haskell, a type class may have more than one type variable, and could be used to express relations between types, something we’ll utilize frequently in HaGPipe. Subclasses can be defined by restricting what types that can be instances of a class by using `=>`, as in the following example where `Ord` is a subclass of the `Eq` class:

```haskell
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```
Given that functions in Haskell have no side effects, one might wonder how to actually perform any work in Haskell. Monads [20] are one of the more advanced features in Haskell that provides the means for effectful computing to Haskell. A monad is a “wrapper” type with two defined operations. These two operations are called return and >>= in Haskell, and is declared in a type class as:

```haskell
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

This class is actually not implemented on types, but on type constructors. We can see that \texttt{m} in the method declarations above is always used with a parameter (\texttt{a} or \texttt{b}) and thus must be a type constructor expecting one type parameter. A monadic value is called an action, e.g. the type constructor \texttt{IO} implements the \texttt{Monad} class so the type \texttt{IO String} is an \texttt{IO}-action that results in a \texttt{String}. The method \texttt{return} creates a simple action that wraps up a value in a monad. The resulting value from an action could be used by the >>= operation to create a new action in the same monad. Writing monadic actions in Haskell is made easy thanks to the do-notation. By using the keyword do we could instead of

\[ m = a >>= (\lambda x \rightarrow b >>= (\lambda y \rightarrow c >>= (\lambda z \rightarrow d x y))) \]

just write

\[ m = \]
\[ \mathrm{do} \]
\[ x <- a \]
\[ y <- b \]
\[ c \]
\[ d x y \]

Every row of a do statement is a monadic action, in our example the actions are \texttt{a}, \texttt{b}, \texttt{c} and \texttt{d x y}. The order of evaluation is implied by the order of the rows. The <- keyword is used to bind the (unwrapped) result of a monadic action to a local variable. Besides the wrapped up values, monads may also contain an internal state that is passed along by the >>= operator. For example with the \texttt{Reader} monad its easy to pass along an read-only environment value without the need of explicitly wire it through all functions. In the \texttt{State} monad, the internal state value is also writeable. With the \texttt{IO} monad, the internal state is the entire world, so this monad could be used to write stateful code that for instance accesses the file system or writes to the screen. The \texttt{main} function in a Haskell program is a value of type \texttt{IO ()}, i.e. an \texttt{IO}-action returning a trivial () value (i.e. nothing). Since the constructor of the \texttt{IO} type is private, this \texttt{main} function is the only way of getting a “handle” to the outside world. The definition of the \texttt{Monad} class ensures that the internal state of the \texttt{IO} monad never may be copied or retrieved, neither can we execute \texttt{IO}-actions from within
a pure function.\textsuperscript{5} As an example of how the \texttt{IO} monad is used, consider this simple Haskell program:

```haskell
main =
  do
    putStrLn "Enter your name:"
    name <- getLine
    putStrLn ("Hello " ++ name)
```

The first row writes a prompt to the screen, the second reads input from the keyboard, and the third writes a greeting back to the screen. Thanks to the \texttt{do} notation, imperative programming in Haskell is as simple as that.

\textsuperscript{5}Actually, with the function \texttt{unsafePerformIO} this could indeed be done, but as the name suggests, it is unsafe and only used in extremely rare cases.
Chapter 3

The pipeline

We start by defining a polymorphic data type \texttt{GpuStream }\texttt{a}, which represents a stream of \texttt{a}s on the GPU. A stream is an abstraction of vertices, primitives, fragments, pictures etc streaming through the graphics pipeline. As mentioned before, for the sake of this paper we’ll limit ourselves to streams of vertices and fragments.

\begin{verbatim}
newtype GpuStream a = GpuStream a

instance Functor GpuStream where
    fmap f (GpuStream a) = GpuStream (f a)

The data type \texttt{GpuStream} is polymorphic, not only to incorporate both vertices and fragments, but also the different attribute combinations they may have. A “vertex” or “fragment” may be any combination of types representable in a shader, e.g. floating point values or vectors. A vertex may for instance be represented as a position vector and a normal vector. By letting \texttt{GpuStream} become an instance of the standard library \texttt{Functor} class, we can operate on the vertices or fragments running through the stream. The \texttt{Functor} class is defined as:

\begin{verbatim}
class Functor f where
    fmap :: (a -> b) -> f a -> f b

fmap takes a function and in our case an \texttt{GpuStream}, and applies the function to all elements of the stream, resulting in a new stream. This corresponds to using a vertex or fragment shader, but in a more modular way. It is possible to fuse several operations together by applying \texttt{fmap} several times on different functions. We may also use the same polymorphic functions both on general purpose code as well as in shaders.

To transform a stream of vertices to a stream of fragments we use the function \texttt{rasterize}. The declarations of this function’s type is shown below:

\begin{verbatim}
rasterize ::
    ShaderRasterizable v f =>

rasterize takes a function and in our case a \texttt{ShaderRasterizable}, and applies the function to each element of the input stream, resulting in a new stream of type \texttt{v}.
This function takes a stream of vertex positions paired with some data and returns a stream of fragment positions, grouped with their facing coefficients, window positions and rasterized data.¹ The type class ShaderRasterizable is used to constraint the data that can be transferred over the shader boundaries. This type class will be defined as we cover the border control in chapter 8. We can also see a lot of new types such as V4, V2, :, Vertex and Fragment in the definitions above.² These types will also be defined in the following chapters.

To generate shader code, we use the functions vertexProgram and fragmentProgram to extract information for the different shader stages:

```haskell
type VPosition = V4 (Float :> Vertex)
type FPosition = V4 (Float :> Fragment)
type FFace = Float :> Fragment

type FWinPos = V2 (Float :> Fragment)
```

vertexProgram ::
  (VertexIn v, FragmentOut r) =>
  (GpuStream v -> GpuStream (SMaybe r Fragment)) ->
  Shader()

fragmentProgram ::
  (VertexIn v, FragmentOut r) =>
  (GpuStream v -> GpuStream (SMaybe r Fragment)) ->
  Shader()
```

Both of these functions take a pipeline function as argument and return a Shader () type. The Shader type will be explained in detail in the next chapter. A pipeline function is a transformation of a vertex stream of some input type (i.e. an instance of the VertexIn class) to a fragment stream of some output type (i.e. an instance of the FragmentOut class). These type classes will also be defined in chapter 8.

¹One might expect the rasterize function to require input on how to rasterize the vertices, e.g. what kind of primitives are used and if multi sampling should be used. To keep things simple in this paper, we assume that these parameters are already set by the host program.

²Remember that since :: starts with a colon, it could be used as a parameterized type with two parameters, e.g. Float and Fragment.
Chapter 4

Programs as abstract syntax trees

Now that we’ve defined the pipeline wide combinators, let’s focus on what we can do with those GpuStreams when we use the fmap function. As an example before we dig into the details, let’s see how a Phong [17] shader would look like in HaGPipe:

```haskell
main = do
  (writeFile "vertex.spu.cpp" . generateSPU . vertexProgram) pipeline
  (writeFile "fragment.cg" . generateCg . fragmentProgram) pipeline
  pipeline = fmap fragShader . rasterize . fmap vertShader

vertShader (vInPosition, vInNormal) =
  (canonicalPos, (vertexNormal, vertexWorldPos))
  where
    model        = uniform "modelmatrix"
    proj         = uniform "projmatrix"
    norm         = convert model
    homPos       = vInPosition :. V4 x y z (const 1)
    worldPos     = model * homPos
    canonicalPos = proj * worldPos
    vertexNormal = norm * vInNormal
    vertexWorldPos = convert worldPos

fragShader (fragmentViewPos, _, _, (fragmentNormal , fragmentWorldPos)) =
  toSMaybe (Just (fragmentColor, z fragmentViewPos))
  where
    a          = 30
    lightPos   = uniform "lightpos"
    material   = uniform "material"
    eyePos     = uniform "eyepos"
    n          = normalize fragmentNormal
```

13
What this program does (in the **main** action), is that it generates source code for a vertex shader and a fragment shader and writes it to disc. The function **pipeline** in this program is our complete render pipeline. Note that we won’t actually “run” this pipeline with any data in this program, just turn it into two text files containing source code written in some other shader languages.

The functions **vertShader** and **fragShader** is mapped over the streams of vertices and fragments respectively, and this is where all the work gets done. The . operator is used to compose functions together just as ◦ in mathematics. In **vertShader** we transform the position in model space to world space with a model matrix, and then use a perspective projection matrix to get the coordinate in the canonical view space. The vertices’ normals are transformed accordingly using the model matrix. The rasterizer uses the canonical view coordinate to interpolate the world position and normal over the generated fragments. The **fragShader** function then use these interpolated values to calculate a Phong shaded color for each fragment. The resulting color consists of three components: ambient color, diffuse color and specular color. Ambient color is a constant component, diffuse color is computed from the angle between the normal and the incoming light, and the specular is computed from the angle between the reflected light and the viewers direction. The dot product is used to get the cosine of the angle between two vectors. We need to use some **iff** expressions\(^1\) to make sure that fragments that aren’t facing the light doesn’t get diffuse or specular lighting. The function **pure** above is used to create a vector of values from a scalar value, and .\* is used for component-wise multiplication of vectors. The operator .: is called “swizzling” and is defined in the next chapter. The normal arithmetic operators +, - and \* are extended to apply to vectors and matrices as well, e.g. \* between a matrix and a vector (as in worldPos above) is a matrix multiplication, while it’s defined as the dot product when used between two vectors (as in l * n above). a ** b is defined in Haskell as “a raised to the power of b”.

We now want to take the functions **vertShader** and **fragShader** and somehow turn them into shader code, but there is no way to inspect a function programmatically in Haskell. The solution is to use a special data type for the output of those functions that instead of containing a resulting value contains the expression used to compute that value. This expression can then be turned into shader code, e.g. in Cg, and saved to file. In order to get a resulting expression from the functions we need to feed them a symbolic input value that acts as a placeholder for the actual input that will be given to the shaders. We need to define this expression data type and the operators we use on it in the vertex and fragment shaders. For the expression type, we use an abstract syn-

\(^1\)See chapter 5.
tax tree (AST). An abstract syntax tree is a data structure used in compilers that is generated by the source code parser [3]. As an example, the AST for the variable *fragmentColor* in the Phong shader above would look something like figure 4.1.

Figure 4.1: AST for the *fragmentColor* variable.

```
Apply * Constant 0
Apply >
Apply +
Apply saturate
fragmentColor
ambient
ambient
l n
diffuse specular
```

The AST data type is defined in HaGPipe as an algebraic data type representing a dynamically typed expression tree. The leaves are either symbols represented by the Symbol data type that represents the shader input attributes and other shader variables, or constant values represented by the OrdDynamic data type, that is a variant of the type Dynamic\(^2\) that also supports comparison and equality. This is important because we need to be able to sort and compare different ASTs in many rewrite rules. The branch nodes, created with the Apply constructor, are applications of operators (represented by the type Op) to lists of operand ASTs:

```haskell
data AST =
  Symb Symbol |
  Constant OrdDynamic |
  Apply Op [AST]
```

data Symbol =
  Variable Int TypeRep | -- A common sub expression
  SamplerAttribute String TypeRep | -- A global sampler attribute
  UniformAttribute String Int | -- A global uniform attribute
  VertexAttribute Int | -- A custom vertex input attribute
  FragmentAttribute Int TypeRep | -- A custom fragment input attribute
  FragmentPosition | -- A fragments canonical view position
  FragmentFace | -- The fragments' triangles' normals' z-component
  FragmentWPos | -- The fragments 2D window position
  FragmentColor Int | -- The resulting color of a fragment
  FragmentDepth Int | -- The resulting depth of a fragment
  FragmentDiscard -- A boolean telling if the fragment is discarded

\(^2\)This type can wrap any value that implements the Typeable class, i.e. types that supports type introspection.
Some things are worth mentioning regarding the AST. Firstly, the number and order of arguments are dependent on the operator, e.g. the iff operator always has three arguments where the first denotes the conditional expression, the second contains the result if it’s true and the third if it’s false. Some operators can also have a variable number of operands, e.g. the + operator in figure 4.1 has three parameters. Secondly, all sub expressions will get aggressively inlined, e.g. ambient in figure 4.1 gets instantiated two times for the different branches of the iff expression. In section 9, I will present a technique that eliminates such common sub expressions.

The creation and parsing of ASTs will be split into three phases:

1. Creation, local optimization and normalization
2. Global transformation
3. Code generation

Each of those steps is dependent on which backend we want to use, e.g. there are backends where vector operations aren’t intrinsic and must be expanded into scalar operations, and since we want to support many different backends we’ll need to create some intermediate value that the backend can consume later on. We’ll also see that it’s advantageous to perform local optimizations and normalization as the AST is being built, so here we have a problem; on one hand we want to start transforming the tree right away, but on the other hand we want to wait for the backend. An elegant solution to this problem is to use monads.

The Shader monad we’ve already seen is actually a type alias, defined as:

```haskell
type Shader a = ErrorT String (StateT Declarations (Reader Capability)) a

type Declarations = Map Symbol AST

type Capability = Op -> Bool
```

The Shader monad consists of several monad transformers wrapped up together. Monad transformers [11] provide a convenient way of creating a composite monad by adding features to a base monad. The mtl package contains several useful monad transformers, some of which we’ll use here. Simply put, a monad transformer wraps another monad instead of just a value. For instance StateT is a monad transformer that adds an internal state to another monad, thus giving it the same features as the State monad. For our Shader monad, we choose to start with a Reader monad as base that can hold a backend-specific capability function of type Op -> Bool with which we could ask the backend which operators it supports. On top of that, we wrap a StateT monad transformer that adds an internal state remembering symbols declared throughout the pipeline. Lastly, we would like to add the ability to abort the creation of an AST and to use alternative ways instead. The ErrorT monad transformer we use for this makes our monad an instance of the MonadPlus type class, which provides abort capabilities with the mzero constant, and the possibility to add alternative paths with the mplus method. The ErrorT could also hold an error message in case of an aborted action. Another common Haskell data type that is an instance of the MonadPlus type class is Maybe a, that has the constructors Just a and Nothing and can denote a value (of the parameter type a) or the
lack of one. Having a `Maybe` inside all other layers of monad transformers has unfortunately shown to result in a monad that consumes all available memory and terminates with an exception. Using an `ErrorT` monad transformer on the top of our monad transformer stack solves that problem and makes the code run in constant memory instead. We’ll see how to use the `mplus` and `mzero` methods when we discuss rewrite rules later in chapter 7. We will hang around inside the `Shader` monad for the rest of the compilation process, and it’s not until we wrap it all up in chapter 10 that we’ll find our way out of it.

Now that we have a better understanding of the `Shader` monad, let’s go back to the functions `rasterize`, `vertexProgram` and `fragmentProgram` we declared in the previous chapter. They all have in common that they operate on the internal state of the `Shader` monad. `rasterize` actually stores away the symbols the vertex shader produces into the `Shader` monad’s state. Using `vertexProgram` will throw away everything happening after rasterization, and `fragmentProgram` everything before. These two functions both return a `Shader ()`, i.e. a shader but no return value; the backend will get what it needs from the `Shader`’s internal state.

Having the AST dynamically typed greatly simplifies its usage as the different constructors to match against are kept at a minimum. However, we do need to be able to reflect the type at some point. For applications, the operator data type `Op` has a data member of type `TypeRep` that represents the return type of the operation. `TypeRep` is also used in the data type `Symbol` to alleviate the need to look it up in the definition table of the `Shader` monad. Some symbols are always of a specific type (e.g. `FragmentPosition`) and consequently doesn’t need a `TypeRep`. The data type `OrdDynamic` used for constants also uses a `TypeRep` internally to save the original type. But since the AST is dynamically typed, we don’t want to expose it to the shader programmers directly. Instead we’ll wrap it in some other expression types on which we can define strongly typed functions and operators.

---

3The data type `Op` will be defined in chapter 7.
Chapter 5
General numerics

For the expression types I’m about to define, it would be swell if we could overload numerical literals, as well as inherit the standard prelude operators +, -, *, /, etc, in order to make shader programming as transparent as possible. We also need all overloaded operators to be closed under expression types, i.e. any operator taking expression types as input must return an expression type as output. Haskells built in numerical classes are indeed overloadable, but they are unfortunately not general enough for our specific needs:

- All classes are subclasses of the Show class, and thus must be able to convert into a String type.
- Most of the classes are subclasses of the Eq class, and methods in this class return values of the Bool type.
- The * operator are defined as type a -> a -> a, and can neither be used for type safe matrix multiplications nor for dot products.

The first of these issues are according to myself just a result of bad design choices. The latter two on the other hand origins from the lack of multi parameter type classes (which isn’t standard Haskell). We could choose to create new operators with the same name and just hide the prelude ones, but in that case we would lose the ability to overload numerical literals. The solution to these problems is preventing the standard prelude from being imported in GHC with the {-# LANGUAGE NoImplicitPrelude #-} pragma, which will make it possible for us to define new numerical classes without these limitations and let numerical literals use our classes instead. Since our classes will be more general than the ones in the prelude, we may even let all types that are instances of the old ones become instances of the new ones. One drawback of using this approach is that shader code cannot make use of existing functions that rely on the old numerical classes.

The new numerical classes we define differ from the old ones in the following ways:

- We define a class Boolean for boolean operators such as && that we implement both for the standard Bool type as well as our own bool expression type.
• We define a class \texttt{If} that provides a method \texttt{iff} that generalizes the built in \texttt{if} keyword to operate on different \texttt{Boolean} instances. Note that using \texttt{iff} on a bool expression type will return the conditional expression itself.

• We define a class \texttt{Equal}\footnote{We still want to be able to access the old class \texttt{Eq}, hence we don’t use that name for our \texttt{Equal} class.} of two parameters to parameterize the kind of \texttt{Boolean} that the equality and non-equality operators return.

• The new \texttt{Ord} class will also take two parameters to parameterize the kind of \texttt{Boolean} that is returned.

• The class \texttt{OrdBase} provides the methods that don’t rely on a \texttt{Boolean} type, like \texttt{min} and \texttt{max}.

• The class \texttt{RealFloatCond} provides the methods of the \texttt{RealFloat} class that needs to be parameterized on the type of \texttt{Boolean}, e.g. \texttt{isInfinity}.

• The class \texttt{Mult} provides the \texttt{*} operator.

Every graphics API provides some vector math modules and this package is no exception. To be brief, my \texttt{Vector} module defines the types \texttt{V1 a}, \texttt{V2 a}, \texttt{V3 a} and \texttt{V4 a} which essentially are homogenic tuples, parameterized on the type \texttt{a}, of length 1, 2, 3 and 4 respectively. These types all have one data constructor each of the same name as the type. Matrix types are simply defined as type synonyms for vectors of vectors. The module also defines all vector and matrix operations as methods in type classes with some appropriate default implementations, along with vector and matrix instances for all the new numerical classes we defined above.

The \texttt{*} operator defined in our numeric classes is defined for (type safe) matrix-matrix, vector-matrix, matrix-vector and vector-vector multiplication. The latter is simply defined as the dot product. To keep the number of different operators at a minimum,\footnote{This increases the number of opportunities for rewrite rules to fire. See chapter 7 for more information on these rules.} we define the matrix-vector and vector-matrix multiplications as matrix-matrix multiplication on column and row vectors respectively. For this we use the conversion functions \texttt{toCmat} and \texttt{toRmat} to convert the input vectors to matrices, and \texttt{fromRmat} and \texttt{fromCmat} to convert the results back to vectors. We see that \texttt{fromRmat . toRmat} and \texttt{fromCmat . toCmat} equals the identity function, but also that \texttt{toRmat . fromCmat} and \texttt{toCmat . fromRmat} both translates into the transpose function. These properties can be utilized by our rewrite rule framework that we’ll encounter in chapter 7.

Swizzling is an operator in shading languages that creates a new vector by selecting components from another vector. In the language \texttt{Cg} for example, \texttt{v.wyy} returns a three component vector with the first element set to the \texttt{w} component of vector \texttt{v} and the second and third one set to component \texttt{y}. This is a nifty feature that makes it easy for a shader programmer to create a new vector by selecting components from an old one, and this API should incorporate a similar facility too. We define the functions \texttt{x}, \texttt{y}, \texttt{z} and \texttt{w} that extracts single components from vectors. With appropriate type classes, the function \texttt{x} work on all vector types, \texttt{y} on vectors of at least two elements, and so on. Since the dot (\texttt{.}) that is used for swizzling in other languages already is defined as
function composition in Haskell, we define the operator .: as our swizzling operator instead. On its left hand is the vector to extract the elements from, and on the right is a vector of functions to use on the left hand vector. The expression \( \mathbf{v} .: \mathbf{v3 w y y} \) will create a \( \mathbf{v3} \) vector with components set to the \( w-, y- \) and \( y\)-components from the vector \( \mathbf{v} \). Any function may be elements of the right hand vector, so our swizzling operator could also for instance be used for homogenizing vectors with the expression \( \mathbf{a} .: \mathbf{v4 x y z} \ (\text{\texttt{const 1}}) \). I present the definition of .: below, just to demonstrate how compact definitions may be in Haskell when utilizing higher order types and currying (see chapter 2):

\[
\text{(.:) } \mathbf{v} = \text{\textbf{fmap}} (\$ \mathbf{v})
\]

where the standard library define

\[
\text{\texttt{f \$ x = f x}}
\]
Chapter 6

The expression types

With all these new classes in place, we can go on with defining the expression types. We need different types for different kinds of expressions, e.g. bools, ints and floats. We could define many different expression types, but a simpler solution is to make a polymorphic data type that uses a phantom type [7] parameter. A type parameter that isn’t used in any of the parameterized type’s constructors is called a phantom type parameter. It’s used to restrict the parameterized type further.

\[
\text{newtype } x :> c = E \text{ (Shader AST)}
\]

\[
data \text{ Fragment}
\]

\[
data \text{ Vertex}
\]

This type is just a newtype of an AST wrapped in a Shader monad. newtype works as a data construct, with the restriction that it can only have one constructor (in our case E) with one datum (in our case a Shader AST) but the benefit of having no memory overhead. This type has actually two phantom type parameters: one for the type being expressed, and one for the context in which the type is expressed. For the purpose of this paper, we have only two contexts: Vertex and Fragment. These symbols are simply defined as data types without constructors. The use of an infix type constructor makes it easier to understand the meaning of such types, e.g. \text{Float :> Vertex} means “a Float in a Vertex”.\(^1\) The context parameter ensures that values in a vertex shader cannot be used in fragment shaders, and vice versa. The function rasterize defined in chapter 3 is the only way to convert from \text{Vertex} to \text{Fragment}, and the opposite is not allowed at all. But most importantly, with the declaration of contexts we can define functions that are only allowed in one context. E.g. texturing functions are only allowed in fragment shaders.

Now let’s stop for a moment and consider an important issue: pattern matching on expression types. Up to this point, the special :> data type may seem to suffice for all possible expressions. But what happens if we want to represent vectors in shaders? Well, couldn’t we just, for instance, use the data type \text{V4 Float :> Vertex}? This would be rather unpractical, because we would

\(^1\)Remember that operators starting with : can be used as a type constructor. See chapter 2.
need to define some kind of extraction function to retrieve the elements, since the whole vector would have been wrapped into a single AST. We would like to keep the ability to use pattern matching to bind variables to the elements of the V4 constructor. The solution is to wrap only the atomic types, like Float, which in our previous example gives us the type V4 (Float :> Vertex). The difference is subtle but important. Now we have a vector of (Shader wrapped) ASTs instead of a single AST. The next question is: what happens with operations that take such vectors as input? For element wise operations like +, we can simply map that operation on to the elements, as one would do with a “normal” vector. But for operations like vecLength that returns the length of a vector, we need to be able to take the whole vector as input (and thus as a single AST).

For this reason the type class ::> is defined:

\[
\text{class } a :> t \mid a->t \text{ where}
\]

- \text{fromAST} :: Shader AST -> a
- \text{toAST} :: a -> Shader AST
- \text{liftedType} :: a -> TypeRep

This class has methods to convert an unwrapped value (e.g. a vector) into a single AST and back again. It also contains the method \text{liftedType} to return the TypeRep of the “lifted” type (i.e. the left operator of the :> type). With this class we no longer define our operations only for the :> data type, but for all types instantiating the ::> type class. The constraint \( a :> c \) could be read “\( a \) is expressible in context \( c \)”. This class also declares a functional dependency between its type variables with the expression \( a->t \) that means that type \( t \) is dependent on type \( a \), i.e. the compilers type inferer can infer the second type variable if it knows the first. The instance \( (a :> t) :> t \) is trivial (e.g. \text{fromAST} = E), but for tuples, vectors and other composite types we need to create some structure AST to be able to represent the type as a single AST. HaGPipe defines many instances of the ::> class that uses an special structure AST node\(^2\) on the elements of the data type.

One big caveat with the ::> class is that the instances must be single constructor types, since we can’t know at run time (of the Haskell program) what constructor to convert back to when fromAST is used. With this limitation, and with the fact that the type :> is not an instance of the Eq class (since the values can’t be compared at run time of the Haskell program, but only when the shader executes on the GPU), there will actually never be more than one pattern to match against, and the pattern matching facility will just be used to bind variables to the constructor’s arguments. This is however in my opinion valuable enough to justify the need for the ::> class. In chapter 8 I will present the data type SMaybe that actually has two constructors, but only one of them will be used by the fromAST function.

\(^2\) This AST operator has the empty string ("")) as opcode since it’s so common.
Chapter 7

Rewrite rules

As I stated earlier, the Op data type represents an operation in our abstract syntax tree. Internally, it is represented by a String, a TypeRep and a RewriteRule:

```haskell
data Op = Op String TypeRep RewriteRule

type RewriteRule = Op -> [AST] -> Shader AST
```

The String denotes the operator’s opcode and is used to identify the operation. The TypeRep is used to save the type of the operations returned expression. The last parameter, the rewrite rule, is a function that is used to locally simplify and normalize a sub AST. Normalization in this context is the process in which an AST is transformed into a certain “normal form” to increase the chance of it being equal to another sub AST later on. E.g. in order to get \((a*b) + (b*a)\) to be simplified into \(2*a*b\), the operands of one of the multiplications needs to be reordered. The “normal form” in this case could be to order the operands in alphabetical order, yielding \((a*b) + (a*b)\) where the plus operator now can see that it has two identical operands.

A rewrite rule is invoked by the function apply as soon as the AST node is created,\(^1\) thus avoiding an additional traversal of the created AST.

```haskell
apply :: Op -> [AST] -> Shader AST
apply op xs = getRewriteRule op op xs
  where
    getRewriteRule (Op _ _ r) = r
```

Rules that uses functions or values with the expression’s lifted type in\(^2\) must capture that function or value in a closure at the same time as the operator is created, due to the lack of full type reflection in Haskell.\(^3\) If we did had full type reflection, we could have had a function taking an \(Op\) and return the rewrite rule. But, instead we need to save the rewrite rule as a part of the operation, in

\(^1\)Or, actually, since Haskell uses lazy evaluation, as late as when the value of the AST is needed the first time.
\(^2\)An example of such a type specific rule is constant folding.
\(^3\)We can get a TypeRep from a type by using the typeOf function on an undefined value of that type. However, we can’t create a type or value from that TypeRep.
case a node on a higher level in the AST want to create a new AST node using
the same operator but different operands.

The most trivial rewrite rule is `ruleEnd` that just returns an Apply node
with the provided operation and operands. But before returning, this rule also
invokes the monadic action `support` to ensure that the operation is supported
by the backend. Here will our Shader monad come into play:

```haskell
ruleEnd :: RewriteRule
ruleEnd op xs =
  do
    support op
    return (Apply op xs)
```

Executing the `support` action will retrieve the capability function from the
Reader monad, applying it on the provided operation and if it returns False
result in the action throwing an exception using our ErrorT monad transformer,
i.e. becoming `mzero`. Note that we don’t need to know the capability as the
expression is being built. This is what’s so ingenious with monads; the HaGPipe
expression won’t actually return an AST, but a (rather large) action that, when
run by the specific backend, will return our AST. Haskell’s laziness and the way
ErrorT is defined ensures that no performance penalty is paid.

But what other rules than `ruleEnd` can we use, and how do we declare alter-
native paths in the Shader monad? Executing the `support` action might result
in the containing action aborting with an exception, so what should happen
then? For this, we use some rewrite rule combinators:

```haskell
type RewriteRuleC = ([AST] -> Shader AST) -> RewriteRule

(>>+) :: RewriteRuleC -> RewriteRule -> RewriteRule
(>>-) :: RewriteRule -> RewriteRule -> RewriteRule

infixr 5 >>+
infixr 4 >>-

(>>+) f g op xs = f (g op) op xs `mplus` g op xs
(>>-) f g op xs = f op xs `mplus` g op xs
```

The `infixr` keyword declares the operator as right associative and states
it’s precedence. Besides the combinators `>>+` and `>>-` we also defined the type
`RewriteRuleC`, a rule continuation. A continuation is a function that receives
a function as argument that it will apply to its result instead of returning it.
With `>>+` we can bind several rules together, or more precisely a rule contin-
uation with a rule. Since `>>+` is right associative, we can bind several continu-
ations together and terminate the chain with the `ruleEnd` rule. The rewrite rule
`a >>+ b >>+ c >>- ruleEnd` will run transformation a first, giving it the op-
portunity to either return some AST or pass the (possibly altered) operand list
on to rule b, and so on until `ruleEnd` returns an Apply node with the resulting
operands (unless the `support` action yield `mzero`). Every step created with `>>+
also creates an alternative path in the MonadPlus with the `mplus` combinator
where the alternative is to skip the left hand rule continuation. This means that if rule \(a\) in the example above decides to transform the expression or any of its operands into a non-supported application,\(^4\) rule \(a\) will be abandoned and the input to rule \(a\) will be fed to rule \(b\) instead. If the same happens in rule \(b\), rule \(c\) will fire and so on.

The definition of the >>-> combinator looks very similar to the one for >>+, but its behavior is quite different. This combinator is used to declare fallback rules. First of all, both its operands are RewriteRules, secondly its precedence is lower than the one for >>+. \(a >>+ b >>+ c >>+ d\) means “try rule \(a >>+ b\) first and if no valid path is returned, run rule \(c >>+ d\)”. Several fallbacks can be attached in a chain where highest priority comes first.

Now that we have all these rewrite rule combinators we can finally define some rewrite rules! The HaGPipe package defines over 30 rewrite rules that is used by the :> type’s instances of the numerical and vector classes. As an example, consider the following \(a :> t\) instance of the OrdBase class (defined in our own generic numeric class module, see chapter 5):

```haskell
instance (OrdBase a) => OrdBase (a :> t) where
  max = apply2 "max" (associative >>+ multiIdemPotent >>+ distributive "+" >>+ distributive "min" >>+ commutative >>+ constantFoldNUniType (undefined :: a :> t) (max :: a->a->a) >>+ associativeFilterSingle >>+ ruleEnd )
```

The class OrdBase also declares a method \(\text{min}\) that is defined by \((a :\_ : t)\) in a similar way.

The rule commutative reorders the operands to get a normal form, and associative flattens binary operators into n-ary operators (i.e. operators with a variable number of operands) for normal form. multiIdemPotent will remove all duplicate operands from the operand list. Some rewrite rules are parameterized, e.g. the distributive rule needs an argument telling what operator (by supplying its opcode) the current one distributes over. constantFoldNUniType is an example of a rule that captures a function in a closure to capture its types. It also takes an undefined value just to get its type; undefined cannot be evaluated or it will generate an error. Many of the rules exploit the fact that >>+ attaches the alternative path where the left hand rule is omitted, and simply returns mzero where the rule doesn’t apply. E.g. associativeFilterSingle will simplify the operation into it’s operand if it only has a single one, but result in mzero if it has two or more operands. The order in which one attaches the rules also makes a difference, since they may depend on each other’s normal forms.\(^5\)

\(^4\)This happens if apply is used on a non-supported sub expression that ultimately runs its own ruleEnd rule and throws an exception.

\(^5\)E.g. associative must be run before multiIdemPotent.
apply2 is a helper function that simplifies the creation of Shader expressions such as \texttt{max} above, defined as:

\begin{verbatim}
apply2 \ s \ r \ a \ b = fromAST ( 
  do 
    a' <- toAST a 
    b' <- toAST b 
    apply (Op s (liftedType (undefined::c)) r) [a', b']
  )
\end{verbatim}

Similar definitions are given for \texttt{apply1}, \texttt{apply3} and \texttt{apply4}. 
Chapter 8

Border control

One issue we haven’t dealt with so far is loading and returning data to and from the individual shaders. In our streaming model, a stream of vertex data is fed to our vertex shader, then outputted to the rasterizer that feed the rasterized data to the fragment shader, which outputs color and depth data to the front buffer. In this model, our data is transferred across four different kinds of borders:

1. Loading of vertex data into the vertex shader.
2. Return of data from the vertex shader to the rasterizer.
3. Load of rasterized data to the fragment shader.
4. Return of color and depth data from the fragment shader.

In the first kind of border transfer, vertex attribute buffers containing vectors of up to four \texttt{Float} are set up on the GPU and data is then uploaded by the CPU. Since attributes aren’t processed by the pipeline during transfer, several pieces of information can be packed into a single attribute to save bandwidth. The packing of data may then take place offline on static vertex lists, or may be done on the fly on generated vertex data. We define a class \texttt{VertexIn} to provide the interfaces for unpacking:

```haskell
class VertexIn a where
    unpack :: Consumer (Float :> Vertex) a
    unpackList :: Int -> Consumer (Float :> Vertex) [a]

    type Consumer s a = State ([s], Int) a
```

The methods \texttt{unpack} and \texttt{unpackList} converts \texttt{Float}s to the desired value, and lists of the desired value respectively. By using \texttt{unpackList}, several values could be loaded from a single \texttt{Float}, where \texttt{unpack} only provides the ability to load one value at a time (but maybe from several \texttt{Float}s). The facility used to feed the unpacker with \texttt{Float}s is a monad called \texttt{Consumer}. This is just a type synonym for a \texttt{State} monad that has a list and an \texttt{Int} as state. The list contains all the \texttt{Float}s to be unpacked, and the \texttt{Int} denotes how many elements (\texttt{Float}s in our case) that has been extracted from the input list already. The reason we introduce the \texttt{Consumer} monad here is because we want to be able
to control how the loaded \texttt{Float}s are aligned to the four-component vectors that are loaded by the GPU. If we for instance pack one \texttt{Float} value in the first component of an input vector and then want to load a homogenic position vector (represented by a four component vector), we would need to pad the first input vector with three dummy values (that are\textquoteleft\textquoteleft not used on the GPU\textquoteright\textquoteright’s receiving end) to avoid using components from two input vectors on the GPU for the homogenic position vector. To keep the discarded floats to a minimum and thus the data bandwidth to a maximum, one should order the input data in an appropriate order, e.g. after a vector of three \texttt{Floats}, one should load a single \texttt{Float} value instead of another vector. Thanks to the tuple data type instances of \texttt{VertexIn}, it’s possible to load more than one vertex attribute to the \texttt{Vertex} shader.

When writing shader programs for GPUs, it’s often advantageous to use the 16-bit wide half precision floating point data type. We define \texttt{Half} in Haskell as a \texttt{newtype} of a normal \texttt{Float}, and make use of GHC’s ability to derive any classes for \texttt{newtypes} to make it useable. It’s possible to pack two halves into a single float if the values are unit clamped. A normalized vector of two halves may also be packed into a single float even though the elements are not unit clamped if we renormalize the unpacked vector. To incorporate this into our packing facility provided by the \texttt{VertexIn} class, we define two \texttt{newtypes} \texttt{UnitClamped} and \texttt{Normalized} to be used on scalars and vectors respectively. We will even take this further and create the types \texttt{Third} and \texttt{Fourth} to support packing of three and four values into a single float. These types doesn’t exist on the GPUs of course, but will be represented by half values.

\begin{verbatim}
newtype Half = Half Float
newtype Third = Third Float
newtype Fourth = Fourth Float
newtype Normalized a = Normalized a
newtype UnitClamped a = UnitClamped a
\end{verbatim}

Next border to cross is from the vertex shader stage into the rasterizer. The rasterizer on GPUs operates on triangle vertices and accepts vectors of up to four \texttt{Floats} as input. It interpolates the components of those vectors for all fragments inside the triangle, and in this setting we cannot pack more than one value into every float, but we could pack several values into different components of the same vector. For this we use the class \texttt{ShaderRasterizable}:

\begin{verbatim}
class (v :>> Vertex, ShaderLoadable f Fragment) =>
  ShaderRasterizable v f | v->f, f->v
where
  toRasterizer :: v -> [[Float :> Fragment]]
  listToRasterizer :: [v] -> [[Float :> Fragment]]
\end{verbatim}

Instances of this class must be expressible (with the :>> class) in a \texttt{Vertex} context, and also loadable in a \texttt{Fragment} context (with the \texttt{ShaderLoadable} class, defined below). We still want to align every data entity into as few vectors as possible, and for this reason \texttt{ShaderRasterizable} has two methods
to support rasterization of single values and lists of values, both that produces lists of lists of Floats. This class also has tuple instances to provide support for more than one attribute. The dual class ShaderLoadable is then used to retrieve the rasterized values in the fragment shader stage:

\[
\text{class } (a :\Rightarrow t) \Rightarrow \text{ShaderLoadable } a t\text{ where }
\text{load } :: \text{Consumer (Float } :\Rightarrow t) a
\]

This class is similar to VertexIn to the extent that it uses the Consumer monad to consume Floats and also will align the data to the input vectors. ShaderLoadable is also used for uniform constants:

\[
\text{uniform } :: \forall a t. (\text{ShaderLoadable } a t) \Rightarrow \text{String } \Rightarrow \text{a}
\]

The \texttt{forall} sign is used to declare the type variables when some of them only appears on the left side of the \texttt{=>} sign. \texttt{uniform} tags a uniform with a name that will show up in the generated code. Texture sampler, though not instances of the ShaderLoadable class, are referred to in a similar way with a function named \texttt{sampler} that takes a string and returns a value that is an instance of the \texttt{Sampler} type class. The instances of the \texttt{Sampler} class are just constructor-less data types\(^1\) since they’re only going to be used as phantom types in \texttt{=>} anyway.

Using text labels for constants and samplers may seem a little too uncontrolled for our type safe graphics pipeline DSL. In the current setting where HaGPipe only generate shaders, and doesn’t actually binds them to a program that uses them, this is satisfactory. If HaGPipe for example were extended to be used with Template Haskell as I will suggest in chapter 12, we would instead need to take a more controlled approach.

The final border, the most restrictive one, is controlled by the class \texttt{FragmentOut}:

\[
\text{class } (r :\Rightarrow \text{Fragment}) \Rightarrow \text{FragmentOut } r \text{ where }
\text{getFragmentDecls } :: \text{Int } \Rightarrow r \Rightarrow \text{Shader } \text{Int}
\]

A fragment shader normally returns one color value and one depth value, which are saved in the internal state of our Shader monad for later retrieval by the backend. There are actually several front buffers on most GPUs and for this reason the FragmentOut has instances for (possibly nested) tuples of color/depth-pairs. \texttt{getFragmentDecls} takes an \texttt{Int} saying how many color/depth-pairs are loaded so far, and returns how many are loaded when it’s done (inside the Shader monad since we’re writing to it’s internal state). Tuples are used instead of lists to ensure that every path of the shader returns the same number of color depth pairs. A fragment shader may also choose not to return any value at all to the post processing stage, and for this purpose we would like to wrap our resulting FragmentOut value in something like a \texttt{Maybe}\(^2\). The \texttt{Maybe} type won’t suffice however, since it’s determined to be \texttt{Nothing} or \texttt{Just} something in runtime of the Haskell program, i.e. compile time of our shader program. We need some deferred \texttt{Maybe}, just as we needed expression types for \texttt{Bool}, \texttt{Int} and

\(^1\)HaGPipe defines \texttt{Sampler} instances for \texttt{Sampler1D}, \texttt{Sampler2D} and \texttt{SamplerCube}.

\(^2\)See discussion of monads in chapter 2.
Float. Due to the single constructor restriction of the \( :> : > \) class (see chapter 6), we can’t just wrap the Maybe in a \( :> \) value. The solution is creating a new data type \( S\text{Maybe} \) with the two private constructors \( \text{Sometimes} \) and \( \text{Never} \):

\[
\text{data } S\text{Maybe } a \ t = \begin{cases} \text{Sometimes } (\text{Bool } :> t) \ a & \mid \text{Never} \\
\end{cases}
\]

\[
toS\text{Maybe } (\text{Just } a) = \text{Sometimes } \text{true } a \\
toS\text{Maybe } \text{Nothing} = \text{Never} \\
is\text{SJust } (\text{Sometimes } c \ _) = c \\
is\text{SJust } \text{Never} = \text{false} \\
is\text{SNothing} = \text{not } . \ is\text{SJust}
\]

\( \text{Never} \) corresponds to \( \text{Nothing} \) and \( \text{Sometimes} \) to \( \text{Just} \), with the addition that \( \text{Sometimes} \) also holds a bool expression that when evaluated in the shader may return \( \text{false} \) (thus rendering the \( S\text{Maybe} \) value equivalent to \( \text{Never} \)). This is also the reason we could define an instance for the \( :> : > \) class even though we have two constructors; when converting to an AST we treat \( \text{Never} \) as being \( \text{Something} \) with the constant bool expression \( \text{false} \), and converting back from an AST will always return a \( \text{Sometimes} \) value. Creating a \( S\text{Maybe} \) value in a shader is done by converting from a normal \( \text{Maybe} \), where \( \text{Just} \) values translates to \( \text{Some} \)times with the bool expression being a constant \( \text{true} \), and \( \text{Nothing} \) translates to \( \text{Never} \). It’s not until we use the \( \text{iff} \) function on \( S\text{Mays} \)es that we get \( \text{SMatchings} \) where the bool expression isn’t constant. In the end of the day, from the shader creator’s point of view, using a \( S\text{Maybe} \) will be very similar to using a normal \( \text{Maybe} \), except that we can’t pattern match on it since its constructors are private.\(^{3}\)

There is actually one more kind of border data transfer we haven’t dealt with so far: The one taken by constant literals. \( \text{ShaderLoadable} \) were indeed also used for constants that are set per model, but we don’t want to declare all constants in that way; those that never change should be written as literals directly in the code. For this purpose, we define the \( \text{ShaderConstant} \) class:

\[
\text{class } (\text{Typeable } b) \Rightarrow \text{ShaderConstant } a \ b \text{ where}
\begin{align*}
\text{mkConstant} & : b \rightarrow a \\
\text{fromConstant} & : a \rightarrow \text{Shader } b \text{ --Could be mzero}
\end{align*}
\]

As we might have guessed, the instances of this class uses the \( \text{Constant} \) constructor\(^{4}\) of the \( \text{AST} \) type to store the literals. If \( \text{fromConstant} \) is used on a expression that not is a constant literal, it will yield \( \text{mzero} \), which we used to our advantage in the rewrite rules defined earlier.

---

\(^{3}\)Instead, we have to resort to the functions \( \text{isSJust} \) and \( \text{isSNothing} \) that returns a value of type \( \text{Bool } :> \) context

\(^{4}\)See chapter 4
Chapter 9

Post pass

The rewrite rules we defined earlier will simplify AST nodes locally as an expression is being built. There are still however the need for transformations that can’t be done by the rewrite rules. They include:

- Transformations into non-normal form (as defined in chapter 7).
- Global optimizations.

These kinds of transformations are applied by the individual backends after all ASTs have been declared, and operate directly on the global state saved in the Shader monad. A backend typically use post transformations to make the declared ASTs to better fit the target platform, e.g. change operators’ names, reorder their operands, combine operators, etc. We don’t go into detail on the different kinds of transformations into non-normal form that backends may use, but instead focus on the more interesting group of transformations: global optimizations. In this group we find common sub expression elimination (CSE) and vectorization. We will represent the ASTs as a directed acyclic graph (DAG) in an intermediate data type, but before we do that we need to undo the flattening done by the rewrite rule associative and recreate binary operators from n-ary ones in a disassociation pass:

\[
\text{disassociate} :: (\text{Op} \rightarrow \text{Bool}) \rightarrow (\text{Op} \rightarrow \text{Bool}) \rightarrow \text{Shader} ()
\]

We’ll have to use caution when we split an n-ary operation into binary ones, since we want to maximize the number of opportunities to CSE and vectorize the code. So disassociation starts with finding all common pairs of commutative and non-commutative operands in all associative operations. Because of this, we not only need to specify which operators are associative, but also which are commutative and which are not, hence the two predicate arguments of type \(\text{Op} \rightarrow \text{Bool}\). The disassociation pass then splits all n-ary operations into several binary operations by first trying to create binary pairs matching the common pairs. If no matching pair is found it splits the operands into a balanced binary tree. This will increase the number of opportunities to parallelize code, not only by our own vectorization pass to be defined below, but also by the compilers that consume our generated code since the execution of independent sub expressions can be interleaved in parallel on modern pipelined processors.
Creating a balanced binary tree won’t result in the most optimal program under all circumstances however; factors as instruction delays and number of processor registers may affect what strategy is the best. On the GPU’s and the SPU’s though, where delays are rather long and registers plenty, a full balancing has shown to generate the best results.

There is a trade off in the disassociation pass however: for floating point numbers, the order in which we apply the operations really does matter. For instance, consider the expression \((a+b)+c\). If \(a\) is a large number and \(b\) and \(c\) are small, the sub expression \((a+b)\) might cause some or all of \(b\)'s contribution to be lost when rounding off to match \(a\)'s precision. And when we then add with \(c\), the same happens again. If we instead would add \(b\) and \(c\) first, using the expression \(a+(b+c)\), less precision is rounded off in the sub total that then may influence the total sum more. In other words, floating point addition is not actually associative. However, this potential loss of precision is something I believe we could sacrifice for the performance gain.

When all associative operators have been disassociated, we turn our AST declarations into a DAG with the `createDAG` function:

```
type DAG = (Map Int DAGNode, Map Symbol Int)
data DAGNode = DAGNode Op [TypeRep] [Int]
  | DAGAST AST deriving (Eq, Ord)
createDAG :: Shader DAG
```

A DAG consists of two maps\(^1\). The first is a set of temporary variables, indexed by Ints, where each temporary variable is either represented by an operator application of some other temporary variables, or by a reference to a AST leaf (i.e. a Symb or Constant). The second map associate each previously declared symbol with a temporary variable index. In the creation of the DAG, all temporary variables with equal content will be merged so that common sub expressions are eliminated. This property of the creation process is what turns the ASTs into a DAG instead of just another tree. After the DAG has been created, the backend may perform transformations on the DAG before returning them to AST declarations. One such transformation is vectorization:

```
vectorize :: Int -> (Op -> Bool) -> DAG -> DAG
```

Most research concerning vectorization deals with loop vectorization.\(^2\) This is on the other hand a basic block level vectorization that groups operations in a basic block together and forms vectorized SIMD (Single Instruction Multiple Data) operations, effectively minimizing the number of operations performed \([14]\). For example all Intel based processors since Pentium III supports SSE (Streaming SIMD Extensions) instructions that can operate on four scalar values at the same time. All GPU’s also have SIMD instructions supporting operations on four scalars at once, so we want to find groups of four scalar

---

\(^1\) Associative set. Not referring to the same “map” as in `fmap`. Also called “dictionary” in some other languages.

\(^2\) We will use loop vectorization ourselves when using the backend for SPU code in section 10.2.
operation applications, and replace them with the application of SIMD instructions instead. Figure 9.1 shows an (simplified) DAG before vectorization, and figure 9.2 shows how it would look like after vectorization. Note how the vectorized result contains fewer + and * operator nodes, and how new “structure” and “get” nodes are used to group and extract the components of the vectors to and from these operators. These added nodes are assumed to be cheaper than the + and * operator nodes, so the whole vectorized DAG will be cheaper even though it has more nodes. Also note that the operators dependent on each other aren’t vectorized together.

When using the vectorization transformation, the backend provides a predicate telling which operations should be vectorized, as well as the preferred length of the vectorization. The transformation will vectorize all input to the operations selected by the predicate, which won’t work well for if-clauses. So before vectorization is run, the function packIfs will embed the conditional expression in the operator so that the branches can be vectorized for two or
more if-clauses with the same conditional expression. The function `unpackIfs` is then used after vectorization to restore the conditional expression and make things look right again:

```haskell
packIfs :: DAG -> DAG
unpackIfs :: DAG -> DAG
```

A programmer using HaGPipe will probably write code that seemingly uses vectorized operations when using the numerical operations that are overloaded on vectors. Those vector operations are however implemented so that they map the operation on the elements, to increase the number of opportunities for rewrite rules to fire, but leaving the AST in desperate need of revectorization. Using the vectorization transformation will not only find those vectorization opportunities implied by the programmer, but also many more. Since we created balanced trees in the disassociation pass earlier, we can also vectorize the computation of a sum or product, e.g. 

\[(a + b) + (c + d) + (e + f) + (g + h)\]

may be computed in only three steps. Having the code represented as a DAG really helps to reduce the complexity of the vectorization process. Besides from avoiding to vectorize dependant applications together, no special heuristic is used to choose which next group to vectorize in the process, and it’s possible that using such a heuristic might increase the total number of vectorization possibilities. Nevertheless, the current solution has shown to produce code with a high amount of vectorizations anyway.

When all DAG transformations are complete, the function `injectDag` is used to recreate AST declarations from the final DAG:

```haskell
injectDag :: (Op->Bool) -> DAG -> Shader ()
```

During this process, common sub expressions will be declared as `Variable` symbols in the AST, and it’s also possible to limit the number of produced `Variable` symbols by supplying a predicate that defines which operators are “free” on the target platform. In Cg for instance, the unary negation operator (-) as well as extracting an element from a vector is free, and using those operations alone won’t motivate for the creation of `Variable` symbols.
Chapter 10
Wrapping it all up

When all post pass transformations are done, all that’s left for the specified backend is to generate the code. The AST declarations can be returned from the inner state of the Shader monad by the action allDecls (for functional target languages where the order doesn’t matter), or with allDeclsSorted (for imperative languages, e.g. Cg and C++):

\[
\text{allDecls :: Shader } [(\text{Symbol, AST})]  \\
\text{allDeclsSorted :: Shader } [(\text{Symbol, AST})]
\]

The actual code generation is quite straightforward at this point in the process. Using a monad transformer is a great aid in producing the code during the final traversal of the AST declarations. With the `execWriterT` action, we can add a WriterT layer to our Shader monad, thus extending it with the capability of outputting text. We then traverse the final form of our AST declarations and use the action `tell` as we go along to write code to the output that finally will be returned by the `execWriterT` call. The backend wraps it all up in a call to `generateCode` that runs the Shader monad with the backends capability function used for the `support` action (see chapter 4) and unwraps the resulting code String:

\[
\text{generateCode :: Capability } \rightarrow \text{ Shader Code } \rightarrow \text{ Code}  \\
\text{type Code = String}
\]

In case no supported rewrite path were found, the `generateCode` function will abort execution and output the error message of the last path tried, e.g. “Backend doesn’t support operation vecLength with lifted return type Float”. This will be helpful when constructing backends for HaGPipe.

A backend will probably wrap `generateCode` into a function of their own, taking as input a Shader () returned from either of the functions `vertexProgram` or `fragmentProgram` (see chapter 3), and return a String with code as output. Now, let’s examine the approaches taken by two different backends: One Cg code generator with GPUs as target platform, and one C++ code generator with the SPUs on PlayStation 3 as target.
10.1 Cg vertex and fragment shader backend

The implementation of the Cg backend is quite straightforward since HaGPipe
is designed with the functions and operators of Cg in mind:

\[
generateCg :: Shader () \rightarrow Code
\]

\[
generateCg \text{ prog } = \text{generateCode } (\text{const True}) (\text{do})
\]

\[
\text{prog}
\]

\[
\text{removeToAndFromMatFunctions}
\]

\[
\text{disassociate commutativeCgFuncs nonCommutativeCgFuncs}
\]

\[
dag <- \text{createDAG}
\]

\[
\text{injectDAG freeCgOps dag}
\]

\[
\text{replaceWithCgFunctionNames}
\]

\[
declarations <- \text{allDeclsSorted}
\]

\[
\text{execWriterT } (\text{printCgShader declarations})
\]

The capability function that directs the support action in our Shader monad
is simply \text{const True}, i.e. all operators are supported by the Cg backend. Before
the ASTs are traversed for code generation, some post processing is performed. We remove the conversions between vectors and matrices, and perform
CSE as we convert the AST to a DAG and back. We don’t use our vectorizer
since it turns out that the Cg compiler has a vectorizer of its own, probably
with some search heuristics, and using our vectorizer would actually make the
code less efficient.

The actual code generation is easily done. First we write out declarations
for all inputs and outputs, and then the main functions body. For fragments
shaders, if the \text{FragmentDiscard} symbol is defined (i.e. a path in the fragment
shader results in a \text{Never}) then an if-statement will be written that conditionally
terminates the fragment with the \text{discard} keyword. Other than that, all
statements consist of assignments of expressions to either temporary variables
or output variables. Any if-clauses will be rendered in the inlined (?:) form and
swizzling expressions will also be generated where possible.

10.2 SPU vertex shader backend

The second backend implemented for this project is an interesting one. Its
purpose is to generate C++ code that will be compiled for the SPUs found
on IBM’s Cell processor, used in PlayStation 3. To keep it simple, we limit
ourselves to only generate vertex shaders, but the techniques used in this section
could also be employed for fragment shaders. This backend defines a function
\text{generateSPU} that handles the AST in similar ways as the \text{generateCg} function.
The code generation is done quite differently though.

There are seven SPUs in a Cell processor. Each has its own local memory,
and uses DMA (Direct memory access) to transfer data to and from the main
memory. SPUs have two pipelines, one dedicated for arithmetic operations

\footnote{See discussion on type \text{SMaybe} in chapter 8.}
and one for memory accesses. In contrast to vertex shaders on GPUs, our SPU program will handle the entire vertex array transformation, i.e. loading it from main memory, looping through it to perform the transformations, and transferring the accumulated results to the GPU for rasterization. In order to utilize pipelining, we don’t want to accumulate all vertices before transferring them to the GPU; instead we work on chunks (max 16 kB) of vertex data at once. The SPUs operate solely on vectors of data, e.g. four floats, by using SIMD instructions [2]. If we do scalar computations on an SPU, only one component would be used of the vectors, thus wasting a lot of computational bandwidth. One solution would be to make use of our excellent vectorizer. But even if we did pack all scalars in vectors, it’s likely that we still would have some spare components in the end. A better solution is to operate on 4 vertices at once, arranging our code in a SoA (Structure of Arrays, as opposed to AoS, Array of Structures) format, hence utilizing a form of loop vectorization [12]. We do it in the following way:

1. First, we use the backends capability function to only allow operations performed on scalars. Vector and matrix operations will effectively be expanded into scalar form.

2. We don’t use the vectorizer defined in chapter 9 in order to keep the operations in scalar form.

3. When we loop through the vertices in our SPU program, we step ahead four vertices a time.

4. In the beginning of each iteration, we transpose the vertex attributes from one-vector-per-vertex into four-vertices-per-vector. For example: The vertices 3D positions are represented as one vector with three components for each vertex. We transpose the positions of four vertices into three vectors with four components each, the first vector having the x-positions of all vertices, the second having all the y-positions, and so on.

5. All operations in our shader are now performed in vector form, i.e. operating on 4 scalars at once.

6. When the four vectors output values have been calculated, we transpose them back into one-vertex-per-vector form and saves them in the output buffer for transmission to the GPU.

The transposition to and from vector form is done using the spu_shuffle intrinsic [1]. Eight such statements are needed in order to transpose four vectors of four components each. Even better would be if the initial vertex data were already transposed into groups of four to begin with, then we could save these instructions from our for-loop. The output, however, must still be transposed to be used by the GPU. We could also improve the pipeline utilization of the SPUs if we unroll the loop and process 8, 12 or even 16 vertices at a time in the loop body.

The generated code is structured as some data definitions and two functions. The first function has the name `splatUniforms` and creates vectors for each uniform scalar. The other function has the name `shadeChunk` and processes a chunk of vertices in a for-loop that is incremented in a multiple of four steps.
each turn. In the beginning of each iteration, the indata will be transposed to temporary variables in SoA format. Then the actual shader code takes place, and before the end of the iteration the output in SoA format is transposed into the resulting buffer. The transposition of in- and output are never parts of the AST being parsed,\textsuperscript{2} but simply generated as we write the code.

\textsuperscript{2}And they couldn’t, since the AST represent the operations made on one vertex, and these transpositions happens on multiples of four.
Chapter 11

Results & conclusions

Two different pipelines are used in the tests, the Phong shader presented in chapter 1 and a more advanced pair of vertex and fragment shaders used in production in a game by Avalanche Studios. Cg vertex and fragment shaders are generated by HaGPipe for both pipelines. Manually written Cg shaders are used to compare the generated shaders with. The generated Cg shaders and the reference shaders are compiled for PlayStation 3, and the program NVShaderPerf from NVidia is then used to get some statistics from the compiled binaries, i.e. number of cycles per vertex or fragment, and number of vertices or fragments per second. The former should be as low as possible and the latter as high as possible. The results for the vertex shaders is shown in table 11.1 and for the fragment shaders in table 11.2.

<table>
<thead>
<tr>
<th></th>
<th>Cycles / vert.</th>
<th>Vert. / sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phong</td>
<td>13</td>
<td>307,692,320</td>
</tr>
<tr>
<td>HaGPipe</td>
<td>13</td>
<td>307,692,320</td>
</tr>
<tr>
<td>reference</td>
<td>13</td>
<td>307,692,320</td>
</tr>
<tr>
<td>Av. game</td>
<td>28</td>
<td>142,857,136</td>
</tr>
<tr>
<td>HaGPipe</td>
<td>28</td>
<td>142,857,136</td>
</tr>
<tr>
<td>reference</td>
<td>27</td>
<td>148,148,144</td>
</tr>
</tbody>
</table>

The results are inconclusive. We can see that HaGPipe outperforms the reference Phong fragment shader and does an equally good job on the Phong vertex shader, but performs worse on the more advanced shaders. Some experiments show that using imperative programming style as done in the reference shaders, e.g. using in-place operations such as += and *=, won’t give any performance gains over using temporary variables as the Cg compiler will optimize them away. I also found that HaGPipes CSE on the other hand does improve performance, since the Cg compiler only does some rudimentary CSE.

SPU vertex shaders are also generated for both the Phong and Avalanche game pipelines. Different number of loop unrolls are tested in which 4, 8, 12 or 16 vertices are processed in each iteration. I also test SPU code that transpose the input vertex data during iteration, as well as code in which the data is assumed to be already transposed. The tool SPUSim from the PlayStation 3 SDK is used to inspect the number of cycles needed in the loop body of the compiled SPU program. This number is then divided by the number of vertices.
produced per iteration. The SPU code is loaded and run on a single SPU on a PlayStation 3 and the number of vertices per second is measured in run time. The results is shown in table 11.3. It is uncommon to hand write vertex shaders for the SPU, so we won’t have any reference shaders to compare against, except the Cg-shaders we tested above.

Table 11.2: Cg fragment shader statistics as reported by NVShaderPerf.

<table>
<thead>
<tr>
<th></th>
<th>Cycles / frag.</th>
<th>Frag. / sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phong</td>
<td>14</td>
<td>857,142,848</td>
</tr>
<tr>
<td></td>
<td>reference</td>
<td>16</td>
</tr>
<tr>
<td>Av. game</td>
<td>24</td>
<td>500,000,000</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>600,000,000</td>
</tr>
</tbody>
</table>

From this table we see that pre-transposing the data is beneficial under all circumstances. We get the peak performance at 179 million fragments per second from the Phong shader if we do 12 vertices at a time in the processing loop. This number should be lower for bigger shaders, and for the Avalanche game shader 8 vertices are sufficient, which gives us a throughput of approximately 40 million vertices per second.

If we compare the Cg shaders with the SPU shaders we see that the SPU Phong shader needs approximately 30% more cycles per vertex than the Cg Phong shader. In the more advanced shader however, about 190% more cycles are needed. 2 SPUs would outperform the GPU in the Phong case, but for the more advanced shader we would need 5 SPUs to perform better than the Cg shader. Nevertheless, this is still pretty remarkable for a general purpose processing unit as the SPU. We could also use more advanced shading techniques on the SPUs since we’re not bound to a single vertex in the shader as we have access to the entire loop. For instance recursive geometry shaders could easily be incorporated on the SPUs with good performance. We could also use a combined approach where we do some vertex processing on the SPU, preferably data demanding operations such as skinning, and other vertex operations in the vertex shader of the GPU.
Table 11.3: SPU vertex shader statistics as measured on PlayStation 3.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Phong</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transposing</td>
<td>4</td>
<td>21.50</td>
<td>144,462,634</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>19.13</td>
<td>162,573,913</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>17.83</td>
<td>173,263,193</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>18.19</td>
<td>170,294,062</td>
</tr>
<tr>
<td>pre-transposed</td>
<td>4</td>
<td>19.00</td>
<td>162,815,096</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>18.25</td>
<td>169,880,902</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>17.17</td>
<td>179,052,632</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>17.44</td>
<td>176,226,093</td>
</tr>
<tr>
<td>Av. game</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transposing</td>
<td>4</td>
<td>86.50</td>
<td>36,528,911</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>80.50</td>
<td>39,212,584</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>83.58</td>
<td>37,682,875</td>
</tr>
<tr>
<td>pre-transposed</td>
<td>4</td>
<td>83.75</td>
<td>37,695,291</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>79.00</td>
<td>39,951,767</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>83.00</td>
<td>37,960,946</td>
</tr>
</tbody>
</table>

41
Discussion & future work

What have we actually gained by embedding a graphics DSL in Haskell? Isn’t writing the code in Cg from the start best? Well, first of all, since the graphics pipeline conceptually is functional, I believe that a functional language is better suited than an imperative one for programming it. Secondly, the rewrite rule framework presented in this paper provides an extensible and modular way of defining algebraic rules to simplify expressions with, independent of what backend will be used to generate the code. The results in the previous chapter show that there is still need for more optimizing rewrite rules and other improvements. The vectorizer wasn’t used at all in the backends implemented for this paper as the Cg compiler actually did a better job on its own; if we did do vectorization, it performed worse. Using a search heuristics in the vectorizer might make it useable also for the Cg case.

But we’ve just touched upon the real potential of having a graphics DSL in Haskell. In chapter 3 we defined the graphics stream type GpuStream and the function rasterize to convert between streams of vertices and fragments. Here we gain something not present in standard shading languages: type safety between the different shader stages. The stream concept in HaGPipe could easily be extended to cover additional pipeline stages and multi pass rendering as well. Some examples:

- Geometry shader stage\(^1\) could be incorporated by removing the rasterize function and replacing it with two new functions, one from vertices to geometry primitives, and one from primitives to fragments. Other than that, this stage behaves like the vertex or fragment shader.

- The output merger (blending) stage could be modeled as a fold operation where fragments gets folded on to the front buffers.

- The stream output capabilities of newer hardware, where the output of the vertex or geometry shader can be saved on the GPU and later fed to the pipeline as input, fits functional programming particular well as it can be modeled as having a recursive vertex and geometry shader.

- Having a dedicated data type for front buffers would make it possible to model multi pass rendering. A conversion function from buffer to sampler

\(^1\)Added in Direct3D 10 and OpenGL 2.1 on supported hardware.
could be used to declare the result one pass as input as a texture in another.

- If the type GpuStream were made an instance of the class Monoid, two or more streams of vertices, primitives or fragments could be concatenated. This would imply loading one set of shaders, running some models, loading another set of shaders (if the concatenated streams are different) and then run some other models through the pipeline.

For applications where performance is an issue (e.g. games) we will probably want to use the generated shaders in a program written in a lower level language such as C++. But for many applications, we could benefit from using the shaders in a host program written in Haskell. Template Haskell [19] is an extension to Haskell that gives us the opportunity to extend the compilation by running custom programs and generating Haskell code on the fly. This is done by inserting “splices” in the source code of a Haskell module. A “splice” has the syntax $( e )$, where $e$ is an ordinary Haskell expression of the type $Q\ Exp$. $Q$ is a special monad and $Exp$ is a data type that holds a Haskell expression tree (like HaGPipe’s AST). The spliced $Q$ action is executed during compilation, and the returned $Exp$ is injected back into the source code where it gets type checked and compiled with the rest of the file. The $Q$ monad also has the ability to run certain $IO$ actions, and could for example write to the file system and invoke third party compiler programs. As an example, consider this Haskell program:

```haskell
main =
  do
    world <- loadModel "world.obj"
    window <- createRenderTarget FullScreen
    mainLoop world window

mainLoop world window =
  do
    uniforms <- getCameraAngleAndOtherConstants
    $(createIOsplice pipeline) uniforms world
    mainLoop world window
```

During the compilation of this program, the `createIOsplice` function will be invoked with the provided HaGPipe pipeline. This function will now generate vertex and fragment shaders, save them to file and even invoke the third party shader compiler, and finally return a code fragment. The returned code fragment is an $IO$ action that when fed with render target, uniform constants and a list of vertices, will load the prepared shader binaries and run the vertices through them. In this manner, no shader compilation will take place during runtime, only the loading of the binaries. The binaries could also be wrapped directly in the $IO$ action instead of being loaded from file. The use of Template Haskell really gives us the opportunity to seamlessly integrate our shaders with the program that uses it, and finally makes it possible to create a graphics application in a single program written in a single language.
Chapter 13

Related work

There are several code generating graphics EDSLs implemented in Haskell already. Pan [6] is an EDSL targeting image generation by generating C code. PanTHeon [18] improves it by using Template Haskell for compile-time optimization. An interesting EDSL that generates GPU-shaders is Vertigo [5], in which the surfaces and their textures are described procedural. There are also some GPGPU (General Purpose GPU programming) DSLs out there, targeting nVidia’s CUDA platform, e.g. Obsidian [4] and the GPU kernels by Lee et. al [13]. As far as I know, HaGPipe is the only DSL in Haskell that targets shader code generation for graphics purposes and doesn’t add an extra layer of abstraction over the graphics pipeline. Much of its implementation is however inspired by the previous work above. The rewrite rule framework has also been inspired by the work of van Noort et. al [15].
Chapter 14

Acknowledgments

Thanks to Avalanche Studios for all resources, especially Christian Murray for all help with the PlayStation 3 platform and Gustav Taxén for all other guidance. Also thanks to my supervisor Björn Lisper at Mälardalens Högskola for all useful feedback on this paper. And finally a big thanks to my wife and son for all positive support.
Bibliography


