Renaissance: a functional shading language

by

Chad Anthony Austin

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Human Computer Interaction

Program of Study Committee:
Dirk Reiners, Major Professor
Gary Leavens
Adrian Sannier

Iowa State University

Ames, Iowa

2005

Copyright © Chad Anthony Austin, 2005. All rights reserved.

Graduate College
Iowa State University

This is to certify that the Master's thesis of

Chad Anthony Austin

has met the thesis requirements of Iowa State University

Major Professor

For the Major Program

TABLE OF CONTENTS

1 GENERAL INTRODUCTION

Introduction

Real-time computer graphics hardware has experienced a revolution in the last decade. The days of low-polygon, flat-shaded scenes has long passed: modern graphics accelerators are capable of staggering amounts of geometric detail as well as custom light and material properties on every triangle. Consumer hardware, driven primarily by games, has caught up to and surpassed the traditional workstation graphics processors. In terms of sheer floating point operations per second, modern graphics processors (GPUs) are significantly faster than even general-purpose CPUs. [11]

With this increase in raw power, we have also seen a drastic increase in the flexibility of GPUs. Early GPUs had limited shading capabilities – the best one could hope for was the ability to blend between multiple textures, maybe even with a custom blend operation. Now we have the ability to perform arbitrary calculation on any geometric or color data that's passed down the graphics pipeline, by uploading custom programs, called shaders, written in an assembly language or even C-like languages [16, 22] to the graphics card. With effectively infinite instruction limits and large amounts of temporary storage, the limiting factor in a shading algorithm is the GPU's speed.

The programmability of these processors has come at a cost: if a programmer wishes to execute any custom operations, the entire programmable stage of the pipeline must be implemented. The vertex transformation part of the pipeline can't be customized without implementing lighting as well. This is unlike the traditional fixed-function pipeline, where lighting, texturing, and other graphics processing could all be enabled or configured separately. This problem of shader non-orthogonality could prevent shaders from becoming a standard part of every graphics program. Additionally, shaders require a fair amount of additional work over the fixed function pipeline for most graphics tasks.

We have a developed Renaissance, a new shading language on top of the OpenGL Shading Lan-

guage that addresses these problems. It was designed with three goals in mind:

- Allow shaders to have functionality switched on and off without reducing functionality at run-time, like the fixed-function pipeline.
- Tight integration with the host language to bring shader development to a wider audience.
- To explore a functional programming model for the programmable graphics pipeline.

Thesis Organization

This thesis is organized in the form of three scientific papers prepared for submission to conferences or journals. Each paper captures one core aspect of the system as a whole, and they are all written for different audiences. We will now introduce each paper and show how it fits into this thesis.

The first paper, Renaissance: A Functional Shading Language, is intended for a graphics audience, likely very familiar with existing shading systems. It introduces the project and shows how the existing shading systems have led up to this research. It also explains the design of the language and briefly discusses the operation of the compiler and runtime.

The second paper, The Implementation of a Functional Shading Language, focuses, as the title implies, on the algorithms and techniques required to implement the language. It shows implementation feasibility and some of the interesting results of applying a pure functional language to the shading problem domain. This paper is written for a programming languages audience.

Finally, the third paper is an evaluation of the Renaissance with a human factors approach. An analysis of the language is provided with the help of a heuristic tool, the cognitive dimensions, and a usability study is designed to test some parts of the system. This paper is written for an HCI audience.

Each paper contains a review of existing literature on the relevant subjects.

2 RENAISSANCE: A FUNCTIONAL SHADING LANGUAGE

A paper to be submitted to *I3D 2006: Symposium on Interactive 3D Graphics and Games*

Chad Austin[1] and Dirk Reiners[2]

Abstract

Programmable graphics hardware is growing in capability and flexibility at a rapid pace. Existing languages for programming this hardware make it difficult at best to build collections of custom graphics algorithms that can be combined as needed. We present a pure functional shading language, Renaissance, that uses the concepts of computational frequency and frequency inference to naturally allow composition of shader concepts without generating redundant code. We also provide most of the benefits of metaprogramming languages without the restriction of requiring a full host environment.

Introduction

The most important innovation in computer graphics hardware over the last decade has been the introduction of programmability. Textures were a first step towards fine-grain control over the rendered pixels, and together with multi-pass rendering and later multi-textured pipeline configurability they allowed some basic implementations of user-specific calculations in the graphics hardware. But mapping general algorithms to the very limited and non-intuitive operations that were possible in this way remained something of a black art, as witnessed by the many papers that were published on mapping specific algorithms to graphics hardware, e.g. [12, 14].

[1]Graduate student, primary author and researcher
[2]Assistant Professor, Department of Computer Science, Iowa State University

Offline rendering for animation had been using much more general languages for a long time [10], and some attempts were made to map them to a slightly extended version of the fixed-function OpenGL pipeline [20]. But the real breakthrough came with actual programs that were executed on the graphics hardware.

The first steps were assembly languages for register machines. This was a great step forward for generalizing graphics hardware, but it had its limitations. The shading algorithms were not easy to follow and it was hard to create building blocks of functionality on which the rest of the shader was built. The next natural step was a high-level language built on top of the assembly. These languages often look like C, both in syntax and semantics. There are also metaprogramming languages built on top of a host language. These allow tight integration between the host language and the graphics processor as well as straightforward shader specialization.

With the advent of Cg's interface features and looking at shaders (i.e. a program or programs that runs on the GPU) as elements of an algebra [18], we're just now starting to see support for composable shaders.

In this work we introduce a shading language built on modern functional languages and their pure semantics instead of the procedural roots used before. The functional approach significantly simplifies compilation and analysis of the program, opening up new avenues for more general optimizations and compositions.

Related Work

Multi-Pass Texture Blending

Real-time programmable shading appeared in an early form as multi-pass rendering along with multi-texturing blend modes. The Quake 3 engine for example provided a simple shader scripting language to control the number of passes, texture stages, and rendering states. This isn't a complete solution for general shading, but it goes a long way towards allowing the implementation of several surface appearances. Peercy, Olano et al. discovered that an OpenGL implementation, with some key extensions, can be treated as a general-purpose SIMD computer in their OpenGL Shader work [20]. OpenGL Shader can support arbitrary shading computation, using multiple rendering passes.

However, the trend for processors in general and graphics processors specifically has gone towards higher clock speeds on the processor, but slower and higher latency memory access. This precludes large-scale multipass implementations of shading from being viable, due to the very high memory bandwidth requirements.

RTSL

Stanford's real time programmable shading system, RTSL [21], introduced the concept of computational frequency. They defined four frequencies: constant, primitive group, vertex, and fragment. Constant computation is done at shader compile time and not during the processing of geometry. Primitive group computation is done per batch of geometry, while vertex and fragment computations are done per vertex and per fragment, respectively. RTSL has a retargetable backend that can map vertex computation to either the CPU or to basic programmable vertex hardware. Fragment computation is mapped to multi-pass OpenGL, as in OpenGL Shader above, or early fragment processing hardware like NVIDIA's register combiners. Their shading language did not separate vertex and fragment as the compiler was responsible for splitting the work up among the various computational stages. They allowed explicit specification of where computation is to be done; for example, to easily compare two lighting algorithms, one per vertex and the other per fragment.

Assembly Languages (ARBfp, ARBvp, DirectX shader models)

The next generation of shading languages allowed full programmability at the vertex and pixel levels via assembly languages for a vector based register machine architecture. Although the instruction sets were limited at first, the languages allowed arbitrary computation per vertex and per fragment. They are more efficient than the multi-pass approaches above, because they require much less memory bandwidth. One obvious disadvantage of assembly languages is that they are difficult for people to write and understand, as well as maintain, especially when programs get larger. One principal advantage of assembly languages is that they are directly executed by the underlying hardware. Due to the variability of graphics hardware, between and within vendors, this is rarely the case for shader languages, making them less attractive.

Cg, HLSL

Naturally, the next step beyond an assembly language is a high-level language that compiles to it. Cg [16] and HLSL were created by NVIDIA and Microsoft, respectively, as C-like, high-level shading languages. HLSL compiles to the Microsoft-defined DirectX vertex and pixel shader models, which are loaded into the card at runtime. Cg, on the other hand, compiles to a variety of back ends and is graphics API neutral. The most recent NVIDIA cards have support for Cg in the driver itself, requiring no special compilation step.

When referring to the language used by both Cg and HLSL, I will call it simply Cg. For the sake of compatibility with other shading systems, and transparent access to the underlying hardware, Cg does very little work for the user. She is required to specify how data is transferred into the shaders and which attribute channels map to what. By design, Cg also does not virtualize any resources, if a feature is not available. One of Cg's primary goals is to be as close to the hardware as possible while maintaining a higher level of abstraction.

GLSL

While Cg and HLSL were being developed, 3DLabs and the OpenGL Architecture Review Board were designing a shading language for the future of OpenGL. The OpenGL Shading Language (GLSL [22]) had different goals than Cg and HLSL. It was intended to become part of the OpenGL standard, replacing the assembly languages. OpenGL implementers must have the shader compiler in the driver itself, as opposed to an external process. This increases driver complexity, but means that applications that use GLSL benefit from driver upgrades and compiler improvements for free. It is also a forward thinking language design in that it requires all implementers to support things like conditionals and loops even if they can't do it in hardware. It requires virtualization of resources not visible to the shader writer, such as temporary registers and instruction count.

Sh

Sh [17] isn't exactly a language, per se. It is a metaprogramming system on top of C++ designed for building shaders. Sh is implemented through a set of custom C++ objects that build an internal

program representation when operations are applied to them. This program is compiled to an underlying shader that is run directly on the graphics card. The advantage of a metaprogramming system such as this is that it has very tight integration with the host language. If the shader references a global variable, and assignments are made to that global variable outside the definition of the shader, the data is automatically passed in as a uniform. Also, it is natural to use the host language's features in order to specialize shaders. For example, if the shader contains an `if` statement, two different shaders may be generated, based on the outcome of the condition.

Sh's primary disadvantage is that it requires a full C++ compiler to use a shader. Thus, shaders can't easily be passed along with 3D models, limiting their usefulness to people who aren't programmers. That said, there are some uses for shaders where a metaprogramming approach is ideal; such as implementation of custom graphics algorithms tightly bound to the application.

Vertigo

Vertigo [7] is a metaprogramming system like Sh, but built on top of Haskell instead of C++. The interesting aspects of Vertigo are that it is a functional language and uses expression rewriting for optimization. Expression rewriting allows it to do an optimal search of expression space to reduce the amount of computation necessary in a particular evaluation. A compelling example is that of vector normalization. Vector normalization is a common operation in graphics programs. When writing a procedure, there is a choice between accepting a normalized vector or a potentially non-normalized vector and then normalizing it explicitly. Since normalization is expensive, normalizing a vector twice should be avoided. However, in a functional language it is possible to take advantage of referential transparency and expression rewriting to reduce the expression `normalize (normalize v)` to `normalize v`. Once this optimization is available, there is no reason not to normalize a vector, if it needs to be. Redundant normalize calls are optimized away. Vertigo shows how this is done in an elegant and automatic way.

Contributions

In this paper we introduce a programming language for real-time graphics hardware that we believe addresses many of the problems in the existing languages, discussed above. This language draws from research in modern, pure functional languages, such as Miranda, Haskell, and Clean. We base our design on functional languages for a variety of reasons. First, functional languages are a very natural fit to the programming model exposed by graphics hardware. Second, functional languages are traditionally easier to efficiently compile than imperative languages with side effects, such as C. Third, our language is designed to have a minimum of extraneous syntax, making it much easier to learn and read.

This paper's primary contributions are the following:

- A pure functional programming language with a straightforward semantic model and syntax
- Automatic computational frequency inference for optimal partitioning of program execution to four stages of the graphics pipeline
- Natural shader composability that follows naturally from the simple execution model and frequency inference

Key Design Decisions

Functional Model

Renaissance is based on the syntax and semantics of modern, typed, pure functional languages, specifically the family consisting of Miranda, Haskell, Clean. Since we don't expect our audience to be familiar with the syntax or semantics of these languages, the following will introduce the look and feel with an example.

```
pi = 3.1415927
square x = x * x
circumference r = pi * square r
```

The first line defines a name `pi` to have an approximate value of pi. The second line defines a function called `square` that takes one parameter and returns its square. The third line defines the circumference, given a radius, to be pi times the square of the radius. `square r` is the syntax for function application, and it means "apply the function square to the value r". Notice that the example does not make any types explicit. Types are inferred based on a definition's expression and any arguments. So, above, `pi` has type `float`. square's type is `t -> t`, meaning "a function that takes type t and returns type t", where t is the type of the arguments. So `square 10` has type `int` and `square 10.0` has type `float`. This type inference is discussed in detail later.

There are no loops or variable assignments in this language. Every object, once created, cannot be changed. This is called referential transparency, which refers to the fact that if the same function twice is called twice with the same arguments, the same result will be returned.

Modern GPUs have a stream programming model: there is a stream of data elements (vertices or fragments) and a function is applied across all of them. This function, in stream programming terminology, is called a kernel. Since all of the stream elements are independent, the function can be run in parallel without any sort of synchronization or data dependency analysis. This is largely the reason why graphics processors these days are so efficient: performance increases linearly with the number of processors available. Previous shading languages have semantic models similar to C; that is, variables that can be modified and read from. Further, the order statements are executed is critical. Consider the C-like code in figure 2.1.

The value of `a` at the end of `main()` is either 9 or 5, depending on whether `foo()` or `bar()` is called first. In general, this restriction complicates the compiler's task of optimization and static analysis. A functional language, on the other hand, is given a lot of freedom to reorder evaluations, because all dependencies are explicit and no evaluation has side effects. For specialized tasks, functional languages have been shown to perform much more efficiently than equivalent C code.

As hardware programmability increases in capability and shaders get longer and larger, we believe a functional language will scale in both performance and maintainability more than a language based on the imperative model of C.

```
int a = 1;
int foo() {
  a += 2;
  // Some code.
  return 10;
}
int bar() {
  a *= 3;
  // Other code.
  return 15;
}
void main() {
  int sum = foo() + bar();
  // do something with a
}
```

Figure 2.1 C code example

Even ignoring the performance and "compiler-friendly" issues, functional languages are a better mental model for the humans writing shaders as well. They make explicit that an operation on a stream element has no side effects beyond the output value. Other shading languages must explicitly document that modifications to global variables do not affect the program's operation on other stream elements.

Frequency and Type Inference

Renaissance is a statically typed language, as in C++, other shading systems, and most pure functional languages. That is, the type of an expression is associated with its name, not its value. However, Renaissance infers the type of an expression from context, so no types need be specified explicitly. Consider:

```
foo a b = a + bar b

bar b = b + 2

result = foo 10 4
```

Notice that no types are explicitly specified. However, when `result` is evaluated, `foo` is called with two integers and returns the sum of the first and `bar` of the second. The result of this addition is an

integer as well, so the value `result` has type `int`. Consider the definition of a function that normalizes a vector:

```
normalize v = v / length v
```

The operation of the function is clear even though its argument and return types are not specified. This has a surprising side effect: the `normalize` function actually represents several functions, each of different type. Given that division by a scalar and the length function can operate on multiple types of vectors, normalize will work with any vector. This is similar in practice to C++ template functions.

Alongside each expression's type, we also maintain a computational frequency, a concept introduced by Stanford's RTSL. There are four frequencies: constant (per compile), uniform (per primitive group), vertex (per vertex), and fragment (per fragment). Built-in shader inputs each have a specified frequency. For example, `gl_Vertex` has the frequency `vertex`. `gl_FragCoord` has the frequency `fragment`. If an operation on two expressions that have different frequencies is performed, the resulting expression usually has the higher of the two. One exception is the `if` construct: if the condition has constant frequency, the `if` is evaluated at compile-time, and, if true, the resulting frequency is the frequency of the `if-true` expression. Otherwise, it is the frequency of the `if-false` expression.

Outputs have a required frequency as well. The `gl_Position` output has frequency `vertex` and `gl_FragColor` output has frequency `fragment`. It is an error to define `gl_Position` to be an expression with frequency `fragment`. Outputs must have frequency less than or equal to their definition. Now assume that `gl_FragColor` depends on the normalized, transformed normal:

```
gl_FragColor = dependsOn (
                normalize (gl_NormalMatrix * gl_Normal))
```

`gl_NormalMatrix` has frequency `uniform` and `gl_Normal` has frequency `vertex`. Thus, the normal transformation can be done on the vertex processor. It looks at first glance like the `normalize` call can be moved up to the vertex processor too, but, since it is a nonlinear operation and the fragment interpolators linearly interpolate, the normalization must be done on the fragment processor. Conceptually, all operations are done on the fragment processor, and lifted to earlier stages of the pipeline if possible.

Single Shader for Both Vertex and Pixel Pipelines

In contrast with the most popular real-time shading languages today, Cg, HLSL, and GLSL, we decided to blur the distinction between vertex shaders and fragment shaders. One concern raised by NVIDIA in the design of Cg is that the different processors support different functionality, and by making the programs explicitly separate, the differences are made clear[16]. However, recent trends suggest that the vertex and fragment processors will grow closer in functionality, rather than farther apart. Microsoft's new graphics standard, the Windows Graphics Foundation (WGF, aka DirectX 10) is pushing for a unified processor architecture for both the vertex and fragment parts of the pipeline [2]. ATI technology has also recently been issued a patent on a multi-threaded graphics core that hides the distinction between vertex and fragment units [15, 1]. With this in mind, we feel the potential confusion caused by executing "one" program on two potentially-different processors (in addition to the CPU) is worth the benefit in improved shader clarity, maintainability, and optimization.

To mitigate the potential confusion brought about by this approach, we may allow specification of computational frequency explicitly, as RTSL does. If a lower frequency is specified for a result than the values it depends on (for example, if it is claimed that a result has a frequency of `vertex` but it depends on the `fragment`-frequency `gl_FragCoord` value), a compile-time error is raised. Conversely, explicitly specifying a higher frequency than would be inferred would force computation to occur later in the pipeline, which could be a performance improvement in some cases.

Shaders As Data Files

Following the example set by Cg and GLSL, it is critical that shaders can be treated as data files so that they can travel along with the models whose surfaces they describe. Requiring a compilation step before being able to load or use a shader greatly increases the amount of time it takes to iterate changes, especially for shader building tools and people who aren't programmers. For this reason, the approach taken by metaprogramming shading systems is infeasible for many uses of shaders, such as in games and modeling software. The convenience of being able to use a fully-featured general-purpose language for generation of shaders is offset by the requirement of having a complete C++ or Haskell compiler in order to use them at all. Further, the basis of functional programming languages,

the lambda calculus, provides a high degree of abstraction and notational convenience even with a naive implementation [13]. Therefore, we can provide many of the important features of other high-level languages, such as higher-order functions and specialization, with a minimum of effort. Also, Vertigo shows that an optimizing compiler from a functional language to GPU architectures is relatively straightforward, especially compared to an optimizing C compiler. In short, we believe a “small” functional language with a simple and powerful semantic model can satisfy the needs of shaders just as well as the metaprogramming systems, without the requirement of a host environment.

System Overview

The Renaissance system is implemented in C++ and split into two pieces: the language, including its compiler, and the shader management API. For simplicity of implementation and to leverage the extensive design work that went into the OpenGL Shading Language, we have chosen GLSL as the basis for a large portion of our language.

When the program loads a shader, it is parsed, validated, and type checked into an intermediate program structure. The program can then set the value of any constant inputs. When the program is bound, it is compiled into code that can run on the GPU, optimized for the constant values that have been set. This part is what enables efficient specialization and composition. The generated code is cached with the constants used to create it so recompilation is not necessary when switching back and forth between shader states.

Setting uniforms and attributes does not invoke recompilation, since their values do not affect the structure of the generated code.

One of the niceties of metaprogramming languages is that the interface between the host program and the shader is very convenient, since it can use native data types and structures. Contrast this with the OpenGL Shading Language APIs which require querying and managing uniform indices, and use function names with ’warts’ to distinguish between setting different uniform types: glUniform1f and glUniform2i etc. We can get close to the convenience of a metaprogramming language by providing custom C++ types that hide the internal data transfers.

```
ren::Bool enableBones(program, "enableBones");
```

```
enableBones = true;
program->bind();  // Compiles if necessary.

enableBones = false;
program->bind();  // Compiles if necessary.

enableBones = true;
program->bind();  // Does not compile, already done.
```

The next two sections define the language and the compiler in more detail.

Language Description

The syntax and semantics of Renaissance are very similar to the languages Miranda, Haskell, and Clean.

A program consists of two components: inputs and definitions. Each is separated by a newline. (Renaissance is whitespace-sensitive.)

Inputs

There are three types of inputs, one for each of the first three computational frequencies: constants, uniforms, and vertex attributes. Constant values are taken into account at compile time, uniforms at primitive group time, and attributes per vertex. Since their type cannot be inferred, it must be made explicit:

```
constant bool enablePerPixelLighting
uniform mat3 colorMatrix
attribute float temperature
```

Definitions

A definition either specifies a value or a function, with the general form: name (arguments)* = expression

```
value = 2 + 2
function arg1 arg2 = arg1 / arg2
```

`value` is a value of type int and function is a function of type `s * t -> u` (takes two values of potentially different types and returns the type of dividing the first by the second). `function`'s return type is not evaluated until it is called with arguments. In this sense, `function` actually refers to a template of possible functions which are instantiated when called.

Expressions consist of infix operators and function applications. Precedence of operations is the same as in GLSL. Operators are discussed more fully in a later section.

Evaluation of functions is done lazily, as in Miranda, Haskell, and Clean. This prevents redundant code generation:

```
constant bool doExpensive
choose expensive cheap =
        if doExpensive then expensive else cheap
gl_FragColor = choose ExpensiveCalculation CheapCalculation
```

The arguments to `choose` are only evaluated if necessary; that is, if `doExpensive` is true at compile time, then only `ExpensiveCalculation` will be performed. Otherwise, only `CheapCalculation` will be performed. Lazy evaluation is necessary for optimal specialized code generation.

Types

Following the conventions set by GLSL, we provide the following types: `bool`, `int`, `float`, and vectors of 2 to 4 elements of each. (`vec2` is a vector of two floats, `vec3b` is a vector of three bools, `vec4i` is a vector of four integers, etc.) There are also three square, float matrix types: `mat2`, `mat3`, and `mat4`. Texture samplers have type `sampler1D`, `sampler2D`, etc. just as in GLSL.

Arrays have type `[t]` where `t` is the type of its elements. Since shading hardware does not yet support variable-length arrays, the length of the array must be specified at `constant` frequency. In order to access the i-th element of an array, an array access is treated as a function and called with parameter `i`.

In Renaissance, there are no implicit type conversions. `2 + 2.0` is a type error, requiring a constructor conversion: `float 2 + 2.0`

Built-In Functions, Operators, and State

As with types, we provide access to all GLSL built-in functions, with the same names, types, and overloads. Texture access is done as in GLSL, with the exception that sampler types may be called as functions with the lookup coordinates as the parameter.

All of GLSL's built-in infix operators are available in Renaissance, with the same precedence. Function calls have the highest precedence, but parentheses are available and operate as expected. A new ++ operator is defined as vector concatenation, replacing GLSL's vector constructors. Given two floats, concatenating them with ++ returns a 2-element vector. For example, `(vec3 1.2 3.4 5.6) ++ 7.8` evaluates to `vec4 (1.2 3.4 5.6 7.8)`

All GLSL state is exposed in Renaissance as expected.

Overloading and Swizzling

Renaissance supports what is known as ad-hoc polymorphism, or overloading, based on the number and type of arguments. For example, the expressions `vec4 1.0` and `vec4 1.0 1.0 1.0 1.0` are equally valid and have the same result, since the first is an overloaded constructor that fills the vector with its one argument. There is a built-in `length` function which takes any vector of size 1 to 4 and returns its length. Renaissance defines a special dot operator (`.`) (similar to the language Nice) that calls the right hand side with the left hand side as its argument. This means `length vec` and `vec.length` are equivalent. This has the nice property that vector swizzling (`vec.xyz`) can be defined entirely within the standard library, although, for performance reasons, it is special-cased.

Composability

As graphics teams begin to replace the entire fixed function pipeline with their own shading algorithms, the restriction that shaders must replace the entire pipeline becomes an increasing problem. Moreover, it is nontrivial to write two independent pieces of the shading algorithms and combine them into one shader at runtime, even if they are independent in definition. Some have solved this problem with elaborate preprocessors that combine the pieces into one shader that does not do any redundant computation. Valve's Half-Life 2, for example, builds over 1500 shaders as part of their build process by combining pieces of them with a preprocessor.

As a consequence of the functional programming model and frequency inference, Renaissance naturally supports composition, as demonstrated by the following example code:

```
constant bool useLightingModel1
lightModel1 = ...  # calculations for light model 1
lightModel2 = ...  # calculations for light model 2
gl_FragColor = if useLightingModel1 then lightModel1
                                    else lightModel2
```

Since the variable `useLightingModel1` has constant frequency, it is evaluated at shader compilation time. Thus, the shader is specialized based on its value, with no extra computation per fragment.

Abstraction Facilities

Traditionally a vertex program that applies skeletal animation bone transformations to each vertex looks something like this:

```
uniform [mat4] bones
attribute vec4 boneIndices
attribute vec4 weights

v0 = weights.x * ((bones boneIndices.x) * gl_Vertex)
v1 = weights.y * ((bones boneIndices.y) * gl_Vertex)
```

```
v2 = weights.z * ((bones boneIndices.z) * gl_Vertex)
v3 = weights.w * ((bones boneIndices.w) * gl_Vertex)
vertex = v0 + v1 + v2 + v3
gl_Position = gl_ModelViewProjectionMatrix * vertex
```

This program has much duplicated logic and is hard-coded for the number of bones applied to each vertex. One improvement would be to use a for loop or iteration construct to iterate over the bone references. This would reduce the duplicated logic, but compilers for these languages do not claim to unroll loops and may even cause the shader to be virtualized onto the CPU if loops aren't supported by the underlying hardware. Given frequency inference and higher-order-functions, however:

```
constant bool enableBones

uniform [mat4] bones
attribute vec4 boneIndices
attribute vec4 weights

skinnedVertex =
    sum [(weights i) * (bones (boneIndices i)) * gl_Vertex)
                    for i in (range 0 3)]
vertex = if enableBones then skinnedVertex else gl_Vertex
gl_Position = gl_ModelViewProjectionMatrix * vertex
```

The syntax `[expr for var in list]` is called a list comprehension. A new list is created by evaluating `expr` on every item in `list`. In this case, the new list contains weighted vertices, which must be summed to get the result. The sum function takes a list and returns the result of adding all its elements. Since the length of the list has constant frequency, it is automatically unrolled.

It may seem strange that the vector `weights` is being called as a function, with an index as a parameter. But, since the index has `constant` frequency, `weights 0` is compiled into `weights.x`, `weights 1` is compiled into `weights.y`, etc...

This version of the shader provides a simple switch to enable and disable bone application at compile time.

Runtime Description

Compiler Backend

As mentioned above, we are building Renaissance upon GLSL. It is a strong foundation for our functional language. Also, several functional languages compile to C as it makes a very effective portable assembly language. Nothing in the language itself prevents other backends from being added in the future, however.

Shaders have special output definitions that are the ones actually responsible for generating code. If `gl_Position` is defined, for example, it must have type `vec4` and frequency of `vertex` or less. Its evaluation becomes part of the vertex program. If it and all other `vertex`-frequency outputs are not defined, a vertex program is not generated at all and the fixed function pipeline is used. If any other `vertex`-frequency is defined, `gl_Position` must also be defined. (In GLSL, vertex programs must output at least a position.) `gl_FragColor` has the same restriction for `fragment`-frequency outputs. These output variables can also be assigned the special value `undefined`, which is equivalent to not giving a definition at all. This is used in the following situation:

```
gl_FragColor = if enablePerPixelShading then getColor
                                        else undefined
```

The reason the special value `undefined` is necessary can be demonstrated by a shader that can switch between per-vertex and per-fragment lighting. When vertex lighting is enabled, we may not need a fragment program at all: the fixed function pipeline may do just fine. In that case, we want a way to define what `gl_FragColor` is, while providing a switch that specifies whether it should generate an output or not.

Fig. 2.2 shows the standard OpenGL brick shader translated directly into Renaissance.

Future Work

While Renaissance satisfies our expectations, there are clearly areas that we feel we could improve it. First, composition and specialization of shaders in our system requires that everything is written and compiled in one file. A “linking” or “module” system would allow users to write independent concepts by themselves and then combine them as needed. Similarly, we would like to extend the concept of functional graphics up to the level of multi-pass effects and state specifications. As Vertigo [7] shows so eloquently, functional programming is a perfect fit for many concepts in computer graphics.

Our research was focused on implementing high-level optimizations such as specialization without redundant code. We would like to apply Vertigo’s expression rewriting system so that we can generate efficient code at the instruction level as well. Along the same lines, additional backends for the assembly language shading languages are an obvious improvement.

Finally, since a functional language provides a clear, unambiguous specification of the dependencies in the pipeline, implementing shader debugging and virtualization on top of Renaissance is a nice opportunity.

Conclusion

As programmable graphics hardware becomes more prevalent and instruction and memory limitations are lifted and removed, a next generation shading language will need to reduce the complexities associated with transferring data and calculations from the host application all the way down to the pixels on the screen.

This paper describes Renaissance, a shading language for modern programmable GPUs that, through the benefits of functional programming, enables efficient and clear algorithm specifications across multiple stages of the graphics pipeline. Through a simple semantic model and frequency inference, natural composability of shading “concepts” is possible, which existing languages make difficult at best. Extending this simple concept, we can imagine a programmable shading system with configurable state that can be flipped on and off, just like the interface to the fixed function pipeline.

Acknowledgments

We would like to thank Conal Elliot for his work on Vertigo while at Microsoft Research – without it we would not have gotten far. Thanks also goes to Simon Peyton-Jones at Microsoft Research for his work on the Haskell language and for releasing his out-of-print book The Implementation of Functional Programming Languages. Finally, Dusty Leary and his infectious love of functional programming greatly influenced the design of the language.

```
# Uniforms.
uniform vec3 LightPosition
uniform vec3 BrickColor
uniform vec3 MortarColor
uniform vec2 BrickSize
uniform vec2 BrickPct
# Constants.
SpecularContribution = 0.3
DiffuseContribution = 1.0 - SpecularContribution
# Transform.
gl_Position = ftransform
ecPosition = (gl_ModelViewMatrix * gl_Vertex).xyz
tnorm = normalize (gl_NormalMatrix * gl_Normal)
# Lighting.
lightVec   = normalize (LightPosition - ecPosition)
reflectVec = reflect (-lightVec) tnorm
viewVec    = normalize (-ecPosition)
diffuse = max (dot lightVec viewVec) 0.0
spec = if (diffuse > 0.0) then s else 0.0
    where s = pow (max (dot reflectVec viewVec) 0.0) 16.0
LightIntensity = DiffuseContribution * diffuse +
                 SpecularContribution * specular
# Brick.
position = gl_Vertex.xy / BrickSize + (vec2 xoffset 0.0)
    where xoffset = if fract (position.y * 0.5) > 0.5 then
0.5 else 0.0
useBrick = step (fract position) BrickPct
color = mix MortarColor BrickColor amount
    where amount = useBrick.x * useBrick.y * LightIntensity
# gl_FragColor should have
type vec4gl_FragColor = color ++ 1.0
```

Figure 2.2 Brick shader

3 THE IMPLEMENTATION OF A FUNCTIONAL SHADING LANGUAGE

A paper to be submitted to *The International Conference on Functional Programming*

Chad Austin

Abstract

Renaissance is a functional programming language for developing programs that run directly on real-time programmable graphics hardware. It provides a novel approach to solving the problem of efficient shader specialization by using frequency analysis and automatic lifting. We show the feasibility of such a design by providing algorithms and techniques in the language's implementation.

Introduction

The advent of programmable hardware is perhaps the most important real-time graphics hardware innovation in the last five years. As programmable shading replaces operations traditionally done with the fixed function pipeline and selected extensions, the lack of mechanisms to elegantly and efficiently combine shading code, without resorting to preprocessors that run on the shader text, is a large barrier to wide adoption of shaders.

We introduce a functional programming language, in the style of Haskell, Miranda, and Clean, for implementing shading algorithms on modern, programmable, real-time graphics hardware. We discuss previous systems for interfacing with programmable hardware and show why they don't facilitate orthogonality. We discuss the implementation of Renaissance in detail and provide an example of efficient specialization that isn't straightforward in existing systems.

Related Work

This research draws mainly upon three other fields: previous shading languages, functional programming, and staged computation.

Shading Languages

Graphics Pipeline as a Shading Language

Before graphics hardware allowed user-defined programs to replace the vertex and fragment processing portions of the pipeline, some systems used multipass texture blending techniques available on existing hardware to perform somewhat arbitrary shading calculations. The Quake 3 engine, for example, provided a simple shader scripting language to control the number of passes, texture stages, and rendering states. Peercy, Olano et al. discovered that a standard OpenGL 1.2 implementation, with a handful of key extensions, could be treated as a general-purpose SIMD computer in their OpenGL Shader work. [20] OpenGL Shader can support arbitrary shading, using multiple rendering passes. This approach is not viable in the long term, as multipass algorithms depend on a very high memory speed, and arithmetic unit clock speeds are increasing faster than memory.

RTSL

Stanford's real time programmable shading system, RTSL [21], introduced the concept of computational frequency. They defined four frequencies at which computation can be performed: constant, primitive group, vertex, and fragment. Constant computation is done at shader compile time and not during the processing of geometry. Primitive group computation is done per batch of geometry, while vertex and fragment computations are done per vertex and per fragment, respectively. RTSL has a retargetable backend that can map vertex computation to either the CPU or early programmable vertex hardware. Fragment computation is mapped to multi-pass OpenGL, as in OpenGL Shader above, or early fragment processing hardware like NVIDIA's register combiners. Their shading language did not logically separate the vertex and fragment stages as the compiler was responsible for splitting the work up among the various computational stages. Keywords allowed explicit specification of where

computation is to be done; for example, to easily compare two lighting algorithms, one per vertex and the other per fragment.

Assembly Languages (ARBfp, ARBvp, DirectX shader models)

As fully programmable graphics hardware appeared, they exposed an assembly-language-like mechanism for programming the pipeline. The assembly language models a vector-based register machine architecture. Although the instruction sets were limited at first, they were leaps and bounds more general than the previous programmability options. Over time they have grown more advanced features such as dynamic branching. They are more efficient than the multi-pass approaches above, because they require much less memory bandwidth; instead, they depend on increasing ALU clock speeds. One obvious disadvantage of assembly languages is that they are difficult for people to write and understand, as well as maintain, especially when programs get larger. Their principal advantage is that they expose the capabilities of the hardware in a direct way and place most of the cost of optimization on the application developers.

Cg / HLSL

Naturally, the next step beyond an assembly language is a high-level language that compiles to it. Cg [16] and HLSL were created by NVIDIA and Microsoft, respectively, as C-like, high-level shading languages. HLSL compiles to the Microsoft-defined DirectX vertex and pixel shader models, which are loaded into the card at runtime. Cg, on the other hand, compiles to a variety of back ends and is graphics API neutral. The most recent NVIDIA cards have support for Cg in the driver itself, requiring no distinct compilation step. When referring to the language used by both Cg and HLSL, I will call it simply Cg, even though they have diverged since. For the sake of compatibility with other shading systems, and transparent access to the underlying hardware, Cg does very little work for the user. She is required to specify how data is transferred into the shaders and which attribute channels map to what. By design, Cg also does not virtualize any resources, if a feature is not available. One of Cg's primary goals is to be as close to the hardware as possible while maintaining a higher level of abstraction.

GLSL

While Cg and HLSL were being developed, 3DLabs and the OpenGL Architecture Review Board were designing a shading language for the future of OpenGL. The OpenGL Shading Language (GLSL [22]) had different goals than Cg and HLSL. It was intended to become part of the OpenGL standard, replacing the assembly languages. OpenGL implementers must have the shader compiler in the driver itself, as opposed to an external process. This increases driver complexity, but means that applications that use GLSL benefit from driver upgrades and compiler improvements for free. It is also a forward thinking language design in that it requires all implementers to support things like conditionals and loops even if they can't do it in hardware. It requires virtualization of resources not visible to the shader writer, such as temporary registers and instruction count.

Sh

Sh [17] isn't exactly a language, per se. It is a metaprogramming system on top of C++ designed for building shaders. Sh is implemented through a set of custom C++ objects that build an internal program representation when operations are applied to them. This program is compiled to an underlying shader that is run directly on the graphics card. The advantage of a metaprogramming system such as this is that it has very tight integration with the host language. If the shader references a global variable, and assignments are made to that global variable outside the definition of the shader, the data is automatically passed in as a uniform. Also, it is natural to use the host language's features in order to specialize shaders. For example, if the shader contains a branch expression, two different shaders may be generated, based on the outcome of the condition. Sh's primary disadvantage is that it requires a full C++ compiler to use a shader. Thus, shaders can't easily be passed along with 3D models, limiting their usefulness to people who aren't programmers. That said, there are some uses for shaders where a metaprogramming approach is ideal; such as implementation of custom graphics algorithms tightly bound to the application.

Vertigo

Vertigo [7] is a metaprogramming system like Sh, but built on top of Haskell instead of C++. The interesting aspects of Vertigo are that it is a functional language and uses expression rewriting for optimization. Expression rewriting allows it to do an optimal search of expression space to reduce the amount of computation necessary in a particular evaluation. A compelling example is that of vector normalization. Vector normalization is a common operation in graphics programs. When writing a procedure, there is a choice between accepting a normalized vector or a potentially non-normalized vector and then normalizing it explicitly. Since normalization is expensive, normalizing a vector twice should be avoided. However, in a functional language it is possible to take advantage of referential transparency and expression rewriting to reduce the expression `normalize (normalize v)` to `normalize v`. Once this optimization is available, there is no reason not to normalize a vector, if it needs to be. Redundant normalize calls are optimized away. Vertigo shows how this is done in an elegant and automatic way.

Staged Computation

The concepts of computational frequency and shader specialization have generalized analogs in recent work in staged computation and partial evaluation. [23] It's not obvious how staged computation research – which is intended for efficient specialization of programs on one architecture – can be applied to custom programmable hardware, but there are certainly similarities, not the least of which is that they tend to be based on pure functional languages.

Functional Programming

A lot of Renaissance's design depends on research in modern functional programming languages, starting primarily with Miranda. [24] The algorithms in the implementation owe much to the techniques specified in The Implementation of Functional Programming Languages. [13]

Motivation and Contributions

Existing shading systems have generally been designed to provide a reasonably close representation of the underlying hardware. Since the underlying hardware requires full replacement of the vertex and fragment stages of the pipeline, the orthogonality of the fixed function pipeline is lost. If any customizable shading is desired, everything else must be implemented as well. Several people have built preprocessors and ad hoc shader composition systems to alleviate this problem. Through frequency analysis, Renaissance allows efficient shader specialization without a higher-level preprocessor.

Since existing languages don't hide the distinction between the vertex and fragment processors, the implementation of a particular shading algorithm requires the developer to explicitly mark which calculations are performed on which stage of the pipeline. Our system allows this to be inferred from the frequencies of the input in the calculations being performed. This is discussed in detail later.

The metaprogramming systems, Sh and Vertigo, have the disadvantage that they depend on a host programming language, and thus cannot reasonably be implemented in graphics drivers or scene graphs, for example. One of Renaissance's goals is simplicity of implementation, so that it can be implemented in or used from many languages and systems. Basing our computational theory on the lambda calculus facilitates this. Compilation and evaluation is also discussed later.

The programmable graphics hardware found in modern graphics acceleration cards, is stateless. That is, there is a set of inputs, a set of temporary registers in which to perform calculation, and a set of outputs for passing data on to the next stage. Elements in a stream that are being processed have no way to communicate with each other. We believe a functional programming model matches this hardware better then C-like languages. For optimization purposes, shaders written in C-like languages are often transformed into single-static-assignment form (similar to pure functional programs) anyway. If that's the case, why not just have a functional programming language in the first place? More importantly, a functional language has a cognitive model such that no values can be modified, matching the human mental model of the underlying hardware better.

Finally, if Renaissance aims to provide the backbone for implementing an orthogonal rendering pipeline with shaders to replace the fixed function pipeline, selection of renderer state (and thus, specification of shader input values) must be as convenient as `glEnable(GL_LIGHTING)`. The Renaissance

runtime system make specification of these values effortless.

In short, our primary contributions are as follows:

- The implementation of a pure functional programming language for programming graphics hardware.
- Efficient shader specialization through partial specialization.
- Automated computational frequency inference for optimal partitioning of program execution to four stages of the graphics pipeline.
- Simple implementation allowing for a wide variety of uses.

Shading Pipeline

An extremely simplified overview of the graphics pipeline can be split into four stages, roughly.

Application

The application configures the rendering state (current shader, lighting parameters, textures, materials, etc.) and pushes groups of triangles to the vertex processor.

Vertex processor

The vertex processor transforms, clips, performs vertex calculations, and begins rasterization. It runs once per vertex.

Interpolators

Any given triangle can generate an arbitrary number of fragments, depending on its size on the screen. The attributes passed from the vertex processor to the fragment processor must be linearly interpolated across the triangle (perspectively correct of course). The interpolators perform this operation.

Fragment processor

The fragment processor performs any final shading calculations on the pixel before it is written into the frame buffer and displayed on the screen. These include per pixel lighting calculations, depth buffer tests, reading from textures, and alpha blending.

Language Overview

Introduction

In syntax and semantics, Renaissance looks and feels very similar to languages in the Haskell, Miranda, and Clean family. For those familiar with functional languages, Renaissance is a strongly typed, pure, non-strict language. For the other, it's probably best to start with examples. Names can be given definitions:

```
gross = 12 * 12
```

Functions over arguments are defined as such:

```
square x = x * x
```

Functions can be applied to arguments:

```
sumOfSquares x y z = square x + square y + square z
```

Certain definitions have special meaning – they are the outputs of a particular stage of the GPU. For example, the special output `gl_Position` refers to the vertex position generated by the vertex processor. `gl_FragColor` is the color of the pixel that gets placed in the framebuffer.

Frequency

Because graphics hardware is split into stages, we borrow the concept of "computational frequency from RTSL. We support four frequencies: `constant`, `uniform`, `vertex`, and `fragment`.

- Values of `constant` frequency are known at the time the shader is compiled into instructions that actually run on the hardware.

- Values of `uniform` frequency may be specified by the application per every batch of vertices.
- Values of `vertex` frequency are given to the shader per vertex. These are generally attributes such as the vertex normal, texture coordinates, and color.
- Values of `fragment` frequency are given to the shader per fragment.

At first glance, the result of an operation applied to two values should have frequency equivalent to the highest frequency of the values. Since the literal constants in `2 + 2` have `constant` frequency, they can be evaluated at compile time, to produce the literal value `4`. (This is known as constant folding in the traditional compiler literature. It also can be thought of as partial evaluation in the staged computation literature.) The generalized concept of frequency allows us to do the same thing for the other stages of execution. Let's say that a fragment stage output, namely `gl_FragColor`, depends on the normal given to the vertex stage transformed into eye coordinates:

```
gl_FragColor = generateColor (normalize (gl_NormalMatrix * gl_Normal))
```

`gl_NormalMatrix` has `uniform` frequency (it does not change per primitive group) and `gl_Normal` has `vertex` frequency. Given our algorithm above, normalizing the transformed normal should have vertex frequency. But remember that there are interpolators between the vertex stage and the fragment stage. A vertex normal can't be referenced directly by the fragment stage – it can only reference the output of the interpolator at that particular fragment. Notice specifically that it makes no difference if `gl_NormalMatrix * gl_Normal` is calculated on the vertex or fragment stage of the pipeline because the normal matrix does not change between primitives in a group. However, normalizing a vector on the vertex processor, and then interpolating that, is different from interpolating the vector and normalizing it on the fragment processor. Now we see that we can only lift computation from the fragment shader to the vertex shader (that is, across the interpolators) if the operation is linear. Multiplication of a uniform matrix by a vector, addition of two vectors, multiplication or division by a constant, and several other operations are linear and can be lifted to the vertex processor. The naive algorithm above works properly for lifting anything from the vertex shader to the CPU, such as multiplication of uniforms, however.

Types

Since this language is based on GLSL, all primitive types are borrowed directly. There are integers, floating-point numbers, and booleans, as well as 2-, 3-, and 4-vectors of each. Three square matrix types are provided: 2x2, 3x3, and 4x4. There are also six sampler types which cannot be manipulated but are used to read from texture units.

Renaissance has two compound types: tuples and functions. A value of type `(int, float)` has two elements, the first of which has type `int`, and the second `float`. A function that accepts an integer and returns a float has type `(int -> float)`. A function that accepts two integers and returns their sum has type `((int, int) -> int)`.

Ad Hoc Overloading

Types in the language are inferred entirely from context, not specified explicitly. Rather than using the Hindley-Milner type inference or class-based polymorphism [25] common in other functional languages, Renaissance uses an ad hoc overloading system similar to C++ [3]. A particular source level function definition actually refers to a template of functions, depending on the types of the arguments. For example, given the definition of the function `square` above, `(square 0)`, `(square 0.0)`, `(square (vec2 0.0 0.0))` are all valid, having types `(int -> int)`, `(float -> float)`, and `(vec2 -> vec2)`, respectively.

The reasons we chose this approach over the standard Hindley-Milner approach are twofold. First, the most common language known in our target audience is C++. Thus, it makes sense to use C++'s template and overloading model. Secondly, and most importantly, GLSL uses ad hoc overloading in the definition of its standard functions and operators. The primary disadvantage of choosing an ad hoc approach over the Hindley-Milner inference is that the types of arguments cannot be inferred directly from their usage in a function. So, to pass a function into another function, it has to be explicitly instantiated with its arguments, as in C++. We believe passing functions into other functions is a rare enough operation for shader developers that this trade-off is acceptable.

Expressions

In an expression, there are four primary types of syntax elements: unary and binary operators, function application, and conditional branches.

```
negativeNormal = -normal          # Unary operator.
eight = 2 + 2 * 3                 # Binary operators.
myLength v = v / sqrt (dot v v)   # Function application.
lightingModel = if dot v1 v2 < 0.0 then model1 else model2  # Branch.
```

Function application has the highest precedence. The precedence of the other operators matches the OpenGL shading language specification.

Inputs and Outputs

Shaders replace a section of the programmable graphics pipeline. They have a set of inputs that they process and pass on to the next stage through a set of predefined outputs. If an output is not defined, the next stage of the pipeline gets a default value.

The two required outputs are `gl_Position` and `gl_FragColor`. `gl_Position` represents the transformed vertex after processing by the vertex pipeline. `gl_FragColor` is the color of the fragment written to the framebuffer after fragment processing.

OpenGL defines a standard set of inputs: `gl_Vertex`, `gl_Normal`, `gl_Color`, texture coordinates, lighting parameters, transformation matrices, etc. Additional inputs can be defined by the user through three keywords: constant, uniform, and attribute.

- Constant inputs have `constant` frequency and cause a recompilation of the generated code when they are changed. They are typically used to reconfigure the shading pipeline while retaining efficient generated code.

- Uniform inputs do not cause shader recompilation but are used to configure parameters that remain constant over a batch of primitives. Lighting parameters and transformation matrices are typically uniforms. Uniform inputs have `uniform` frequency.

- Attribute inputs become part of the vertex stream and are used to store any data that changes often or is associated with a vertex. Common examples include tangent and binormal vectors, temperature, weight, density, additional colors, etc. Attribute inputs have `attribute` frequency.

Compiler Implementation

Parsing

Upon loading, a Renaissance shader is lexed and parsed into an abstract syntax tree (AST), using standard parsing techniques. After parsing, the shader program is stored in a direct representation of the syntax. The `Program` object contains a list of constants, uniforms, attributes, and definitions. A `Definition` has a name and an expression tree, in which nodes correspond to elements of the source. There are only two types of syntax nodes: values and function applications. The program:

```
uniform mat4 TransformMatrix
gl_Position = TransformMatrix * gl_Vertex
```

Builds a program structure with one uniform and one definition, defining `gl_Position` as in Figure 3.1.

Building the Lambda Graph

The compiler, given a program object, knows about the built-in output names, and thus "instantiates" the definitions. This process of instantiation converts syntax nodes into nodes in the lambda calculus, named concrete nodes in this implementation. While syntax nodes only have strings, concrete nodes actually represent objects in the language, and thus they have type and frequency. There are six types of concrete nodes: function application, abstractions, arguments of abstractions, branches, built-in functions (with strict call semantics), and values. Figure 3.2 contains a graphical representation of the following program after instantiation into the lambda calculus.

```
foo a b = a + b
bar = 10
output = foo bar 36
```

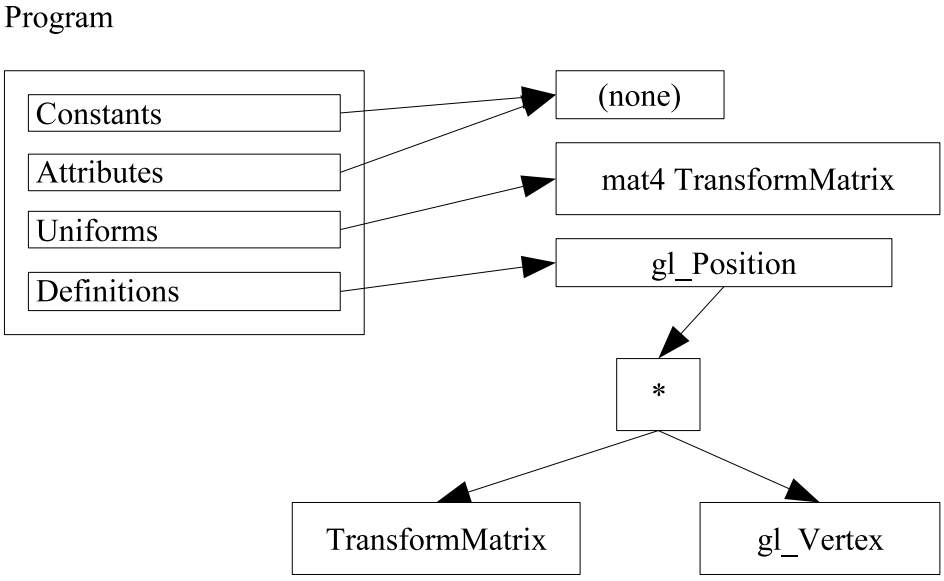

Figure 3.1 Example program structure

When instantiating a function application, the types of the arguments are evaluated, and the correct function is chosen based on those types. No explicit type coercion is performed. If no function accepts the argument types, an error is raised. User defined functions have no type constraints, so the function is chosen entirely on the number of arguments.

Lambda Reduction

Once the program is converted into the lambda calculus, it is evaluated using normal form reduction, explained in detail in the lambda calculus and functional programming literature. In short, when the top of the DAG is the application of an abstraction, the arguments are substituted into the abstraction's subtree. [13] Figure 3.3 shows the value of `output` after one evaluation step.

As evaluation is done as part of compilation, a naive approach is suitable. As abstractions are evaluated, another graph is built as the result of evaluation. This new graph represents the set of computations that must be done to calculate the output, and contains only built-in operators and function calls.

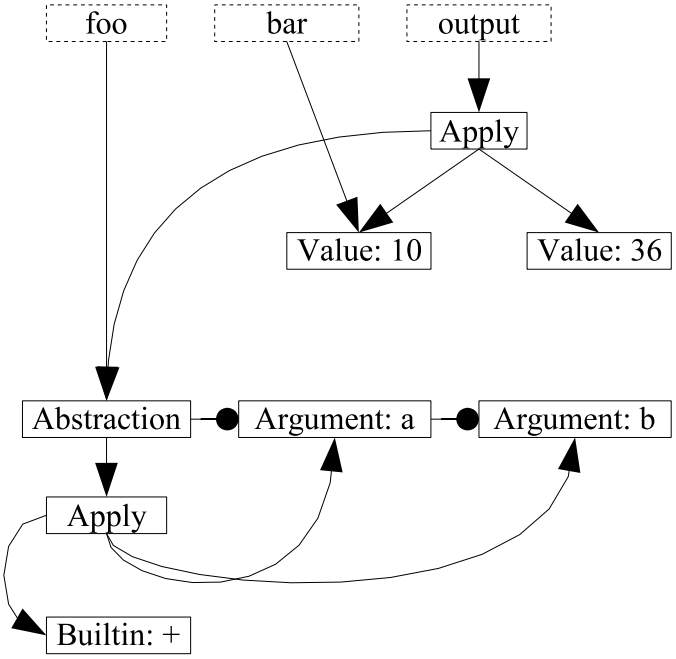

Figure 3.2 Example concrete node graph

The nodes in this graph are called `CodeNodes`. There are only three types of code nodes: branches, native function calls[1], and references to names or constants. Figure 3.4 gives an example `CodeNode` structure.

The generated, rooted, DAGs are inserted into a structure called the shade graph, which represents the shader pipeline as a whole. It is not yet suitable for conversion into GLSL though. Some further processing is necessary.

[1]Operators are considered functions that evaluate to a special syntax in the generated code.

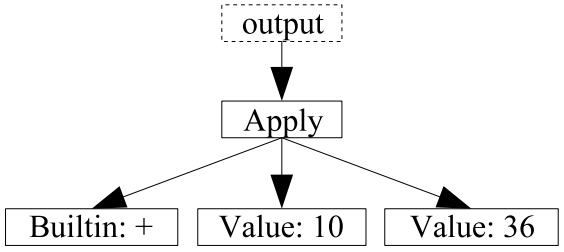

Figure 3.3 output after application of foo

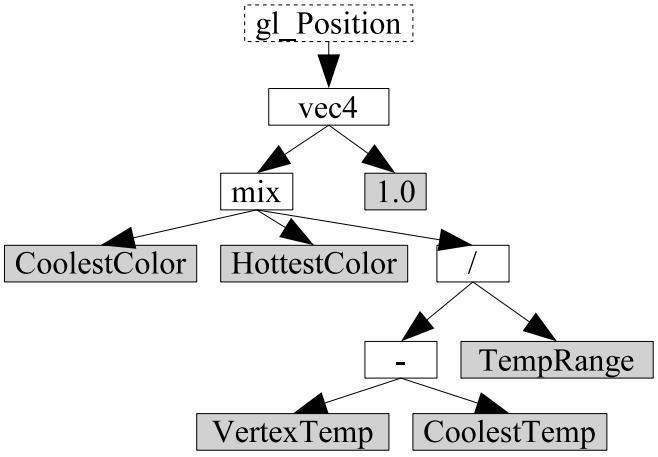

Figure 3.4 Example CodeNode DAG

Constant Evaluation

First, expressions with constant frequency should be evaluated and inserted back into the shade graph tree. This step subsumes constant folding in traditional compilers. It also facilitates efficient specialization: branch code nodes where the condition has a constant frequency can be directly evaluated and replaced with the true part or false part depending on the value of the condition. If nothing else depends on the computation that was removed from the graph, it is never calculated in the generated code. Another subtlety here is that the true part and false part can have different frequencies. This means that the computation that depends on a result of the branch may have a different frequency, depending on the value of the condition. Figure 3.5 is an example of a constant boolean that enables texturing in the shader.

GLSL Shader Generation

At this point the shade graph is converted into a data structure that is conducive to direct conversion into GLSL code. The structure has a list of constants, uniforms, attributes, and varying. It also contains a statement tree to represent things such as assignments, bracketed compound statements, and if statements. The statement nodes may reference subgraphs in the evaluation DAG. For example, assignment statements must reference the value they assign and branch statements must reference their condition.

```
# Basic gouraud shading modulated with a texture if enabled

# Shader inputs
constant bool     texturing   # enabled?
uniform mat4      colormatrix
uniform sampler2D texture

# Output vertex position
gl_Position = ftransform

# Output fragment color
transformedColor = colormatrix * gl_Color
white = vec4 1.0 1.0 1.0 1.0
texel = if texturing then (tex2d texture gl_TexCoord) else white
gl_FragColor = texel * transformedColor
```

Figure 3.5 Example compile-time switch

It would be possible to convert this structure directly into GLSL that can be executed on the GPU, but it's certainly not as efficient as it could be, yet.

Lifting

Next, computational lifting is performed. In most cases, the earlier a computation is performed in the graphics pipeline, the more efficient the pipeline as a whole is. Shaders are compiled much less often than they are drawn, so constant inputs result in the most efficient code, especially if they are used to select code paths in the shader.[2] Each triangle (three vertices) can produce an arbitrary number of fragments, so computation should be performed on the vertex processor, instead of the fragment processor, if possible. As described above, not all computations can be lifted to the vertex processor: only those that have vertex frequency and are linear and thus can be lifted across the interpolator units. The algorithm for lifting is pretty simple: while the fragment portions of the shader have liftable computations, replace them with a reference to a varying, and define the varying in the vertex shader. The lifting stage is done before optimization because lifting may generate common subexpressions in the vertex program which then can be coalesced.

[2]Branching at runtime may not be supported in hardware at all, so not generating unnecessary branches is of critical importance.

After constant evaluation converts the shade graph into the minimum number of calculations that must be executed in the shader, computations in the fragment stage that can be lifted across the interpolators (see above) to the vertex stage are converted into GLSL varying outputs in the vertex stage, and referenced by name in the fragment stage. The algorithm is as follows:

```
find liftable computation in fragment shader:
  in vertex shader:
    add new varying to vertex
    add new assignment statement setting that varying output to the expression
  in fragment shader:
    add new varying (of same, but arbitrary, name)
    replace reference to computation with reference to varying name
  repeat while there are more
```

A computation is liftable if it has certain linearity properties. Unary functions are linear if `c * f(x)` = `f(c * x)` and `f(a) + f(b)` = `f(a + b)`. Binary functions are linear if `c * f(x, y)` = `f(c * x, c * y)` and `f(a, b) + f(c, d)` = `f(a + c, b + d)`. Multiplication by itself is not always linear, but it can be, depending on the frequencies of the results. For example, multiplication of two scalars with vertex frequency is not liftable, but multiplication of a uniform scalar by an attribute scalar is liftable, as the uniform is not interpolated.

Optimization

A side effect of the lifting process is that a liftable computation that is referenced on both the vertex and fragment stages is now duplicated on the vertex side. Common subexpression elimination removes this redundancy. This is the point in the process where other optimizations could be performed too.

Code Generation

Finally, the shader data structures are converted directly into strings which then can be uploaded to the graphics card. This final phase is the most straightforward.

Runtime Implementation

The compiler is only one portion of the Renaissance system (albeit, an important one). A shading system should provide a mechanism for 1) passing data into the system and 2) interfacing the compiled shaders with the underlying graphics API. This section will describe that part of Renaissance. We will describe interfacing compiled shaders with the graphics API first, because it influences decisions in the data input system.

Compilation

A a Renaissance shader can generate many GLSL shaders, depending on the values of constant inputs. Consider a scene of objects, all rendered with different shaders and shading parameters. Recompilation of the Renaissance shader every time an object is drawn with different parameters is undesirable. Clearly, the compiled shader objects should be cached, along with the values of the constant inputs used to generate those objects.

In more detail, the process works like this:

- When an OpenGL program loads a shader, memory is allocated for the values of all the constant inputs.
- Before an object is drawn, the values of constants are set and the shader is bound as the current shader on the graphics device.
- In the shader's bind() call, if this set of constants has already generated a GLSL shader, that one is used. Otherwise, the shader is compiled with the new constants.

The code looks something like this:

```
ren::Shader shader("shader.rs");  // Loads the shader.
ren::Bool useVertexLighting(shader, "useVertexLighting");  // Defaults to false.

shader.bind();  // Compiles for the first time.
drawObject();
```

```
useVertexLighting = true;
shader.bind();  // Compiles again, with vertex lighting.
drawAnotherObject();
```

Data Input

There are three types of input that must be specified in a Renaissance shader: constants, uniforms, and attributes. Constants were described above. The setting of uniforms works in much the same way. If a uniform changes between objects, however, the shader does not need to be recompiled. The uniform's new value must be uploaded to the graphics card before the object is drawn, though. The shader object also maintains a registry of current uniform values so that it knows which values have changed, and thus, which to upload to the card.

Constants and uniforms are largely abstracted away. The user of Renaissance never has to use graphics API calls directly to set or query them. More importantly, the Renaissance input types are much more convenient to use than the graphics API calls. Attributes are different story, however. Since attributes are more tightly bound to the vertex specification mechanism of OpenGL, Renaissance can only provide a mechanism to get the GLSL attribute ID, which the program can then use directly when specifying attribute values or arrays. Using the same approach for attributes as for constants and uniforms would be vastly inefficient.

Future Work

As our research has merely focused on high-level optimizations – such as compiling away unneeded code – we have ignored most low-level optimizations. There are several obvious improvements that can be made to the code generator. It would also be interesting to add expression rewriting, such as that in Vertigo. Support for additional backends, such as DirectX HLSL, is another obvious area Renaissance could be improved.

One of the primary goals of Renaissance is to provide a foundation for creating shaders that can flip functionality on and off. A good way to test the system would be to implement a sizable portion

of the fixed function pipeline as a Renaissance shader. This would also provide the opportunity to do comprehensive performance comparisons between various configurations of the fixed function pipeline and the equivalent (minimal) shader, as well as testing the performance of shader compilation in Renaissance itself. It would also provide a basis for switching entire programs to using shaders all of the time, rather than switching in and out of the fixed function pipeline. In the long run, we could envision GPUs as being extremely efficient and specialized vertex processors that only execute the code given to them. Shading logic would be entirely specified in custom shaders.

As shaders grow, we expect shader developers will want to split them into reusable components, each component representing one concept. For example, a module for doing varying forms of lighting calculations could be shared across multiple shaders and projects.

Debugging shaders can be very difficult because the compilation and evaluation processes are so opaque. We envision a shader debugging mode and tool where intermediate results can be viewed, perhaps by selecting temporary evaluations and rendering them to temporary buffers.

Finally, this project was guided largely by the belief that a functional shading model is a better cognitive model for shading in computer graphics than a C-like language is. This assumption must be tested in a usability study for it to be considered valid.

Conclusion

We've shown the design and implementation of a functional shading language. Renaissance is intended to provide a framework for maintaining the orthogonality of concepts in shading calculations. By basing the language on the syntax and semantics of pure functional languages and adding the concept of computational frequency, we provide this functionality with implementation simplicity. We also show the algorithms that demonstrate that this approach is feasible and effective, perhaps even more efficient than a language based on C.

4 A HUMAN FACTORS ANALYSIS OF A FUNCTIONAL SHADING LANGUAGE

A paper to be submitted to *Communications of the ACM*

Chad Austin

Abstract

Renaissance is a functional programming language for specifying computation to be performed on programmable graphics hardware, addressing some major problems in existing systems. Additionally, it was designed with a focus on usability. This paper provides an analysis of Renaissance, not from a technical perspective, but from a human factors one. It examines Renaissance using the cognitive dimensions framework and proposes a usability study.

Introduction

In the last half-decade, computer graphics hardware has grown the capability for almost arbitrary computation at the vertex and fragment levels. This capability enables a slew of new surface and lighting effects.

In the very early days of programmable shading, everything had to be programmed directly in an assembly language. Collections of simple operations were performed on registers of floating point 4-vectors, then written to outputs. Assembly languages have never been designed for writability and readability, but they got the job done for a while. As more and more effort went into the development of shaders, high-level languages (drawing from experience in shading from the offline rendering world) were developed. These languages are a vast improvement in a variety of ways, but do not go the entire

way towards making shader development realistic as a complete replacement for the fixed function pipeline. [reference first paper]

We have developed Renaissance to address two problems: first, to provide a framework that makes it easy to combine shading ideas without building a complicated preprocessor on top of existing systems and, second, to attend to build a system that is better than existing ones in terms of cognitive load and usability.

In this paper, we provide an analysis of the Renaissance shading language in the context of human factors. We briefly discuss the system's intended audience. We then utilize a heuristic framework, the cognitive dimensions, to provide a "broad brush" analysis of the language and system. Finally, we design a user study to qualitatively measure the effectiveness of the system.

Related Work

Two areas have primarily led up to this research: realtime computer graphics shading and the psychology of programming. This paper primarily focuses on the second, but we will briefly discuss the first for background.

By far the most popular shading language in the off-line shading world is Pixar's RenderMan. RenderMan looks similar to C and programs that describe the characteristics of lights and materials are written in it. RenderMan has had much influence on the design of real-time languages. As mentioned above, the real-time shading languages have evolved away from assembly languages to higher-level languages commonly based on C, including HLSL, GLSL [22], and Cg [16]. Cg has recently grown "interfaces", based loosely on Java, that begin to address the problem Renaissance solves, but not in a satisfactory way. There are also metaprogramming systems, such as Vertigo and Sh, that utilize a host language to define the shaders. These are not suitable for systems where the shaders should be considered part of the 3D model file on disk. (The shader is as much a description of the appearance as the textures are.)

Psychology of programming has been an active area of research since the 1970s [19, 26]. The Psychology of Programming SIG was established in 1987 to coordinate research in the areas of cognitive psychology in software development as a whole. Recently however there has been an increased interest

in the psychology of programming in the mainstream. Microsoft applies a heuristic analysis, called the cognitive dimensions (CDs), to their C# and .NET development tools [5, 4]. The cognitive dimensions are a set of mostly orthogonal axes in which a system can be evaluated. They are not intended to provide a rigorous analysis, but instead give the designer a rough idea of some of the human factors issues inherent in the system. From there, the designer can come up with potential tests and changes that will improve the system.

Renaissance consists of two primary pieces: the shading language itself and a runtime system that hooks the shaders into the underlying graphics API, such as OpenGL, so that they can actually be used. We will often discuss them together, since we took a holistic view when designing them. Either part would not be terribly useful on its own.

Approach

Analyzing a programming language in the context of human factors on multiple axes can be difficult, especially when the language isn't designed for general-purpose programming, but also depends on a great deal of specific knowledge – such as computer graphics, linear algebra, and the physics of light. We do not pretend to know the best way to evaluate the system, and we certainly do not assume that our design is necessarily better than existing systems (although we certainly hope so!). What we really want to know is how to test our system in comparison with existing ones and discover whether the design we created is actually better for some, if not most, users. We would also like to learn which areas of the system could use improvement.

The first thing we will do is a traditional user analysis, to get a rough idea of their relationship with the software. Then we will dig into the cognitive dimensions and apply them if possible. Note that all of the dimensions are not equally valuable in this analysis. Some might not even apply. But overall they do a good job of capturing several important ideas. Finally, we design a potential usability study to test the effectiveness of the system.

User Analysis

Using the persona definition technique from interaction design [6], we have defined three primary user personas. Alan Cooper claims that reasoning about design is made more natural and effective when designing in the context of realistic (although hypothetical) people instead of differentiating “experts” and “beginners”. Our three personas follow.

Aaron is an experienced graphics programmer. Having worked on several modern PC games, he’s intimate with the operation of programmable graphics hardware and how to use it to efficiently implement many shading algorithms. He has traditionally developed the shaders in an assembly language, but recently has been using a high-level language (still checking the generated code, of course!) because every frame per second counts. He reads every major computer graphics research paper and has substantial linear algebra experience. It is very unlikely that Aaron will be receptive to a language that hides the generated code. For shader specialization, he has implemented a preprocessor that runs at compile time and generates all of the different shaders he needs for various configurations.

Steven has been working for a CAD/engineering company for ten years. He has developed and maintained several key pieces of the graphics rendering system in their software over the years and is very familiar with rendering large amounts of geometry, levels of detail, and efficient culling algorithms. He understands some of the concepts in programmable shading but has not had the need or desire to sit down and play with shaders.

Jennifer is a student in computer science with a minor in art and is taking an introductory course in computer graphics. She has written a few programs that render a shaded and lit model, but has only used the fixed function pipeline and doesn’t understand all of the details of how the fixed function pipeline works. She knows that when she positions and colors lights properly, they affect the appearance of the scene. She would like to learn how to write shaders so that she can implement her own lights and material properties.

These three hypothetical users represent the audience we are targeting with Renaissance.

Cognitive Dimensions

The cognitive dimensions [8] are a popular heuristic framework initially created by Thomas Green that are designed for quickly and easily evaluating a system. There are 13 dimensions, each representing an aspect of the system and has an impact on the ability of users to work with it. They allow the designer to get a general feel for the characteristics of the system before running expensive usability tests. Here, we use them to help design a usability study tasks in the third section of the paper.

The cognitive dimensions take a complete view of a system. The notation is the textual or graphical view into structure. The environment is the way the notation is manipulated. The system is defined as both the notation and the environment.

For each of the dimensions, we will briefly discuss what it is and then go into how it applies to Renaissance. For more detailed description of the dimensions, see the literature. [9]

Abstraction Gradient

For clarity, let us define *abstraction* as a conceptual object, a group of elements treated as one. In a traditional programming language, a function is an abstraction over a list of statements or operations. A class in OOP is an abstraction for a data structure and a set of related operations. The abstraction gradient represents whether users are required to learn abstractions before effectively using the system and whether they are allowed to use abstractions if they want to. At first glance, programming systems can be defined as abstraction-hating, abstraction-tolerant, or abstraction-hungry.

We consider Renaissance an abstraction tolerant system. The two main abstractions in our language are functions and types. It is never required to define functions to write a useful shader. The simplest shader looks like this:

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex
```

reading, “Define the output position as the multiplication of the model-view-projection matrix and the input vertex”. However, it is certainly possible to define new functions if necessary. In the current state of the implementation, it is not possible to define user types, but we do intend to implement that in the future.

One potential sticking point is Renaissance's concept of frequency. Frequency can be thought of as an abstraction over the shading stages in the graphics pipeline. For most tasks, Renaissance's frequency analysis provides performance equal to or better than making the calculations explicit for each stage of the pipeline. However, when optimizing a shader, the implicit nature of frequency and frequency analysis may hide the impact of calculations done in the shader on the generated code's performance.

The runtime support, since it is interfaced through C++, has a similar abstraction gradient to C++. That is, the creation of new abstractions is never necessary, but is possible if needed.

Closeness of Mapping

Closeness of mapping represents the similarity of the structures in the problem domain to the structures in the program. In an ideal language, concepts and the problem domain would have a one-to-one mapping with structures in the program. This would prevent the user from having to break goals down into programming sub-tasks in order to solve the problem.

We feel Renaissance has a very high closeness of mapping, especially compared to existing systems. The lack of explicit types and frequency allows the user to specify just the computations required for the desired effect. There is no extraneous syntax. Built-in inputs and outputs are made implicit for this reason as well. The syntax of expressions in the language also comes very close to traditional mathematical notation. Unlike C-like languages, functions with return types and parameter lists do not have to be notated. Temporary variables and order of calculation are also made implicit in Renaissance, under the premise that the compiler can do a better job than a human anyway.

Since every Renaissance object is constant once calculated[1], it does not have the problem where users may think the value of global variables will persist between multiple shader executions on elements in the processing stream. This provides a closeness between the syntax, the semantic model of the shader, and the model of the shading pipeline as a whole.

In the above situations we have considered the problem domain to be that of shading computation and effects. If the problem domain is considered to be the actual instructions executed by the hardware (for example, when optimizing or benchmarking code), Renaissance has a low closeness of mapping

[1]This is called referential transparency in the functional programming literature and means that a function applied to the same arguments will always have the same value, no matter when it is evaluated.

rating. Minor changes in syntax, or even the values of constant input, can effect huge changes in the generated code. The algorithm for splitting computation onto the different stages is neither obvious nor made explicit in the notation.

Consistency

Consistency refers to the "guessability" of a system. Given knowledge of some of the program structure, how much of the rest can be guessed? Note that simplicity brings about consistency simply because there aren't that many types of definitions, expressions, etc. For this reason, Renaissance is consistent with itself. Examples follow. Uniform and constant inputs are specified with the same objects in the runtime library. Also, the input specification objects have the same name as the types that they define as well as behaving like native types in both Renaissance and C++ (bool can be implicitly converted to and from Bool, etc.).

```
# Renaissance snippet
uniform bool booleanUniform

// C++ snippet
ren::Bool booleanUniform(program, "booleanUniform");
booleanUniform = true;
```

Another area we improved consistency over traditional approaches is with swizzles. Swizzles are a window into the operation of the underlying graphics hardware that has followed into high-level shading languages; the elements of a vector can be arbitrarily reordered or ignored. `vec.wxyz` is the same as `(vec4 vec.w vec.x vec.y vec.z)`. Rather than treating swizzles as special syntax, we treat them as normal functions (`(wxyz vec)` is valid) and define the syntax `A.B` to be equivalent to `(B A)`, that is, apply the function B to A. This has an advantageous consequence. In object-oriented languages, fields are accessed with dot notation: vec.length is the length of the vector. In Renaissance, that syntax is also valid; it calls the length function on the vector.

Another area of consistency involves function definition and application. Since we based the syntax of this language on the family of languages derived from Miranda, functions look like this:

```
sumOf a b c = a + b + c
theResult = sumOf 15 25 35
```

Notice that the function definition `sumOf a b c` and the function application `sumOf 15 25 35` have the same syntax.

Renaissance is also consistent with the language it is built on top of, the OpenGL shading language (GLSL). All built-in functions, inputs, outputs, and types are taken directly from GLSL.

Diffuseness - Terseness

A notation with high diffuseness simply uses a larger amount of notation or screen real-estate to express its meaning. In other languages (APL, for example), symbols carry a lot of meaning, by themselves, so fewer are needed to express a concept. Renaissance is also very terse: types and frequencies are implicit, and parentheses and commas are not necessary in function calls. It also does not suffer from LISP's paren-hell because most operators are used in infix notation and have the same precedence as in GLSL.

Often, terseness is a positive thing, as it implies the programmer needs to keep a smaller amount of text in memory and on the screen. However, we feel Renaissance might be too terse for users. It might not be easy for them to quickly scan text and understand its meaning. Hopefully, a user study will shed some light on this issue.

Error-proneness

It is difficult to discuss the issue of error proneness when a system is new and has only a few users. That said, we have come up with a set of potential sources of errors, in no particular order:

In the Renaissance language, if an output is not defined, it is given a default or unspecified value. This means that if an output variable is misspelled, the calculation won't even be performed and the output will have an unexpected value. Since all built-in outputs begin with the prefix `gl_`, it is possible to make it an error to provide custom definitions that begin with `gl_`. This would make output misspellings an error.

Renaissance functions are defined without explicitly restricting the types of the arguments. The type of the result and the operations performed depends on the types of the arguments given. Take the following example.

```
add a b = a + b
```

The `add` function will work as expected if called with two integers or two floats. If called with an integer and a float, however, an error message is generated from inside the add function, since a floating point number and an integer can't be added directly. In this situation, the problem pretty obvious, but as a shader grows, it won't be as obvious where the error is coming from. One way to address this would be to add a type constraint system so that if a function only works with patterns of certain types, that restriction can be expressed directly in the notation, improving the error messages.

An area of concern is that we expect most users of this system to be very familiar with C, C++, or similar languages. In those languages function calls take the form `functionOver(a, b, c)`. In Renaissance, they take the form `(functionOver a b c)`. `functionOver (a b c)` may seem to be equivalent at first glance, but it has a very different meaning: Apply `functionOver` to the result of applying function `a` to `b` and `c`.

These error situations are certainly not an exhaustive list. The usability study should bring up ones we haven't thought of, as well as showing the actual impact of the ones suggested.

Hard Mental Operations

Hard mental operations refer to notational constructs that, when two or three are combined, vastly increase the difficulty of understanding, especially compared to the individual constructs. We have so much experience with the system that we do not feel qualified to decide which operations are especially hard at the notational level. A larger user base and the usability study should enlighten us on this.

Hidden Dependencies

Hidden dependencies are relationships between objects in the system that are implicit and difficult to uncover. If two functions call one to do some common work, changing that function may break one or

both of the callers. This is a hidden dependency. The cognitive dimensions are meant to be evaluated in terms of both the notation and the environment used to manipulate that notation. Given that this research is on the language by itself, no special editor has been developed. When using a standard text editor, the relationship from callees to callers is a hidden dependency. It would be possible to develop an editor that would make it easy to find these back-references.

Another hidden dependency is between the notation and the generated code, when compiled with a certain set of constant values. A primary feature of Renaissance is that branches on constant values can be used to specialize a shader for certain operations, only generating code for the calculations that must be performed. In a sufficiently large shader with enough of these compile-time switches, there is no easy way to tell which functions actually generate code.

Finally, functions in Renaissance are actually syntactic templates for a set of possible functions. This means that if a function is called twice with arguments of different type, it will actually generate two functions – one specialized for each type. When looking deep inside a function, the types and values of the arguments are not directly visible. This may impair understanding.

Premature Commitment

When outlining a document in a notebook with a pen, you need to make sure to leave enough room for the content between the headings. This is an example of premature commitment, where the user is required to make decisions when not ready. A problem inherent in all programmable shading systems is that when shaders are used at all, they must replace the entirety of the fixed function pipeline. Renaissance does not address this problem. Otherwise, the only decision required of the user is what to output to the required vertex and fragment outputs, gl_Position and gl_FragColor. Beyond that, everything can be developed incrementally.

Progressive Evaluation

Progressive evaluation refers to the ability of users to evaluate their progress frequently. We feel Renaissance allows for iterative changes at least as much as traditional programming languages. Two things must be true before a shader can be tested. The required outputs must be defined and the program

must be syntactically correct.

This is an area where Renaissance, GLSL, HLSL, and Cg have an advantage over metaprogramming systems. Since shaders can be directly loaded from files, iteration time is reduced.

Role-Expressiveness

Role expressiveness is related to diffuseness and terseness above. It refers to the self-describability of the notation. When a user looks at a particular piece of a shader, a high degree of role expressiveness means that the user will rapidly discover what that piece does or is for. Role expressiveness can be improved through well named identifiers, comments, and standard idioms. Renaissance supports these secondary notations.

One part of the syntax we think might cause problems is the notation for applying functions to arguments. In traditional mathematical notation, `f(x, sin(y)) + 10` would be the representation for "f applied to x and the sine of y, added to 10". Since Renaissance uses Miranda's syntax, the above could be written `f x (sin y) + 10` which seems to hide the order of evaluation. Users may tend to explicitly insert parentheses to make the the order more explicit: `(f x (sin y)) + 10`

Secondary Notation

Secondary notation encompasses all of the ways a programmer can encode meaning in the program, beyond that which is required for correct execution. These include identifier names, comments, use of whitespace, and grouping of similar structures. Renaissance allows most of these, but it uses whitespace as end-of-line punctuation so arbitrary use of whitespace is restricted.

Viscosity

Viscosity refers to the amount of work required to make a small conceptual change. Renaming a class in C++ is an example of something that has very high viscosity – the header file must be changed (and even renamed), and the name must be changed in the corresponding source file and anywhere the class is used. The notation has high viscosity, but a refactoring browser may make this operation much simpler, reducing its viscosity. (Remember: System = Notation + Environment)

Renaissance is such a small and experimental system that we don't see any highly viscous areas. One thing that requires multiple changes would be changing the name of an input. The name would have be changed in the shader, in the code that referenced that name, and perhaps the name of the C++ input objects would be updated. Metaprogramming systems have an advantage here because inputs are tied directly to the host language – updating and referencing the shader inputs is more direct.

Visibility and Juxtaposability

A system with low visibility makes it cognitively difficult to bring related structures into view. Juxtaposability refers to the ability to view objects side-by-side. These dimensions primarily focus on the editing environment. Since a Renaissance shader is (in the current implementation) entirely in one file, visibility and juxtaposability depend on the text editor used. This dimension would be much more relevant for a specialized shader editor.

Design of a User Study

Now that the CDs have given us a look at some of the cognitive aspects of the system, we present a usability study to begin to explore these issues. We have two goals with this study: 1) perform a comparison between Renaissance and an existing, standard shading system, GLSL, and 2) gain qualitative information about the effectiveness of our design. We are not necessarily interested in specific quantitative results – they would be too specific and could hide the overall value of the system. The forest for the trees, so to speak.

Subject Selection

Ideally, we would like to test Renaissance with subjects who are at least moderately familiar with computer graphics, but do not have much shading language experience. If possible, we would like a sufficiently large group to cover the three roles we have defined above. Since this is such a specialized audience, we actually expect to perform the test on people who have moderate to extensive graphics experience, even if they've used existing shading languages.

Format

We will split the group at random into two. One will be using GLSL and the other Renaissance for the tasks below. The first thing we'll do is administer a questionnaire to discover the background and previous experience of the individual participants. Then we will provide a classroom-type lecture to make sure everyone understands the shading pipeline and how data flows through it. There are three things we must make clear about shaders in general:

- How vertices get put into the pipeline by the application.
- What the fixed function pipeline does, and how it can be replaced by programmable shaders.
- How resulting fragments get written to the screen.

Since Renaissance and GLSL have the same data types and built-in functions, we then must discuss them:

- Scalars, vectors, and matrices
- The functions and operators in the language.
- Provided inputs and outputs.

This educational session is given after the questionnaire so as to avoid affecting the results. It also mitigates the effect of the subject's previous experience on the test.

Tasks

The test itself consists of five tasks. They are designed to build an understanding in a linear order. The tasks happen sequentially, in one session, expected to last about two hours. The subject sits at a computer while the administrator watches, takes notes, and provides assistance if necessary. A "cheat sheet" and reference documentation are available for both the GLSL group and the Renaissance group.

Task 1

The first task is designed to get the subject familiar with writing Renaissance shaders. The instructions are "Write a shader that writes a vertex transformed by the projection and model-view matrices to the output position, and writes the color white to the output color." The result in Renaissance should look something like:

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex
gl_FragColor = vec4 1.0 1.0 1.0 1.0
```

The equivalent in GLSL:

```
// simple.vert
void main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}


// simple.frag
void main() {
  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

The subject is given a program that loads the shader and draws a teapot with it.

Task 2

The second task is the other half of the first one: the subject develops the parts of the program that must load and execute the shader. A program skeleton is provided that initializes OpenGL, handles the window, and other tasks unrelated to the usability study. The subject is required to load the shader, check for errors, and bind it to GL.

The first two tasks are primarily intended to judge the initial overhead required in using Renaissance. Can understanding at first impression be improved by reorganizing documentation? Renaming some of the objects in Renaissance? How long does it take to get up and running compared with GLSL?

Task 3

Task #3 concerns itself with the effort involved in a relatively simple and straightforward change to the shader: adding an input uniform. The task instructions are “Add an input uniform to the shader, and set the output color to the value of this uniform. Then vary the color in the program to verify that it works.” This task has two components: adding the uniform to the shader itself and setting its value from the program. The first should be very straightforward. We expect the second to be significantly easier in Renaissance than GLSL, simply because GLSL is a C++ API that is intended to look familiar.

Task 4

The fourth task takes advantage of a primary Renaissance feature: automatically lifting computation from the fragment shader to the vertex shader. We want to test whether the unified vertex-fragment shader model of Renaissance makes cross-stage calculations such as lighting substantially easier. The task is to add a new set of uniforms representing a light and perform a per-fragment lighting model in the shader. The subject is given the required parameters and the mathematical equations needed to implement the light model and is expected to develop a shader to properly shade an object.

The GLSL group, due to the separation of fragment and vertex shaders, have two possible implementations. They may perform all of the computation in the fragment shader at reduced performance but increased readability. More experienced users are likely to automatically perform vertex-dependent calculations on the vertex pipeline and pipe them through varyings, even though this is more complex, time-consuming, and prone to error.

An area of interest here is whether GLSL users have trouble dealing with the vertex-fragment separation or if it comes naturally. It’s also possible that they will try to put too much computation in the vertex stage, attempting to interpolate the results of non-linear calculations.

Task 5

The fifth and final task focuses on the compile-time frequency analysis. Since GLSL has no analog of Renaissance’s constant frequency calculation, besides a preprocessor on top of the source text, which would take substantial effort to implement, this task is only performed with the Renaissance subject

group. The goal is to implement one shader with two shading algorithms and a compile-time switch to select which one to use. A `constant bool selector` input will have to be added and `gl_FragColor` must be defined as `if selector then algorithm1 else algorithm2`. We are interested in three things:

- Does it make sense to use the same data input mechanism for both constants and uniforms?
- Does the user understand that the shader is recompiled when a constant is changed?
- Does the user understand that the branch statement is not actually executed in the instructions sent to the hardware?

Comments

It is important to remember that this usability study is qualitative – it is not intended to show statistically significant differences. Instead, we intend for it to uncover future areas of investigation. That said, we believe several important results would be obtained.

Future Work

The next step is obvious: The usability study must be performed, the results gathered and analyzed, and Renaissance changed or clarified in response. Beyond that, since Renaissance is such a simple functional language and GLSL is such a simple imperative language, shaders could be used as a reasonable basis for quantitative, statistically significant comparisons of imperative and functional programming in general.

Conclusion

We have shown a preliminary human factors analysis of the shading language design we have created. The cognitive dimensions have allowed us to paint a high-level picture of the impact of our system on human behavior and uncovered areas for future research and investigation. We then presented a design for a usability study to test some of our assumptions and decisions. The study is intended to lead into specific follow-on studies measuring the effectiveness of specific other aspects of the system.

Based on our initial observations, Renaissance's usability looks favorable compared to other system, primarily because it handles the work of splitting what the shader developer writes efficiently onto the different processors. Since it also addresses the major problem of nonorthogonality in existing shading systems, we have hopes that it or a similar design will become the primary way people develop real-time shaders.

5 GENERAL CONCLUSIONS

General Discussion

Chapter 2, the first paper, introduced the design of a functional shading language that provides an elegant design for addressing the problems of shader non-orthogonality and automatic lifting of computation onto previous shader stages. The second paper shows that the design is feasible and provide specific algorithms for implementation. Finally, the third paper evaluates the language with a human factors approach.

By basing Renaissance on the lambda calculus and a form of staged computation, we have designed a language that has syntax and semantics straightforward enough to address computation at all four stages of graphics hardware. This method enables efficient shader specialization without any preprocessors or additional syntax. Additionally, we believe it has the benefit of being easier to use than existing shading systems.

Recommendations for Future Research

In the future, we would like to see this system or a similar one provide the foundation for shading on graphics hardware. We can envision a time when graphics hardware evolves into very fast, parallel vector processors that don't have any silicon dedicated to specific graphics operations. That logic would be provided by a shader in software which could be customized by arbitrary applications. In some sense, this is possible today, but there is no way to tell OpenGL to use a different lighting model or to use per-pixel shading without writing a shader for the entire pipeline. As a proof of concept, we would like to develop a shader for the majority of the fixed-function pipeline in Renaissance.

As we are in control of the software used to generate the instructions in the shading pipeline,

there are opportunities in shader debugging tools. It would be possible to build a debugger on top of Renaissance that interactively renders temporary results to the framebuffer. Something like this will become increasingly important as shaders grow in complexity, especially if we have the bulk of the fixed-function pipeline implemented in Renaissance.

There are obviously several improvements that could be done to the code generation portion. One interesting aspect of optimization is addressing the tension between temporary storage and number of instructions. There is also a restricted number of interpolator units, so an algorithm to efficiently stay within the allowed number would be important.

Since human factors were a concern throughout the project's design, we would like to validate our decisions. Chapter four of this thesis presents a design for a usability study. Once it is performed, additional insights into the usability of Renaissance should guide further design and refinements.

BIBLIOGRAPHY

[1] Dave Baumann. Ati multi-threading patent - for wgf2.0 and xenon graphics? Beyond 3D (www.beyond3d.com), 2005.

[2] David Blythe. Windows graphics foundation. Presentation at WinHEC 2004, 2004.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[4] Steven Clarke. Evaluating a new programming language. In *G Kadoda (Ed), Proceedings of the thirteenth Annual Meeting of the Psychology of Programming Interest Group,* pages 275–289, April 2001.

[5] Steven Clarke. Api usability and the cognitive dimensions framework, 2003. http://blogs.gotdotnet.com/stevencl/PermaLink.aspx/ae9f669d-09df-4bc7-b882-ea8361cdcc30.

[6] Alan Cooper. *About face 2.0 the essentials of interaction design*. Wiley, 2003.

[7] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*, 2004. http://conal.net/papers/Vertigo/.

[8] T. R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[9] Thomas Green and Alan Blackwell. Cognitive dimensions of notations resource site. http://www.cl.cam.ac.uk/ afb21/CognitiveDimensions/.

[10] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298, New York, NY, USA, 1990. ACM Press.

[11] M. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of Graphics Hardware '02*, 2002.

[12] W. Heidrich and H.-P. Seidel. Realistic, hardware accelerated shading and lighting. *ACM Trans. Graph.*, pages 171–178, 1999.

[13] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[14] Jan Kautz and Hans-Peter Seidel. Towards interactive bump mapping with anisotropic shift-variant BRDFs. In *Proceedings of the 2000 ACM/SIGGRAPH Graphics Hardware Workshop*, pages 51–58, 2000.

[15] Laurent Lefebvre, Andrew Gruber, and Stephen Morein. U.s. patent: Multi-thread graphic processing system, 2005.

[16] W. Mark, S. Glanville, and K. Akeley. Cg: A system for programming graphics hardware in a c-like language. In *Siggraph 2003, Computer Graphics Proceedings*. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2003.

[17] M. MCCOOL, Z. QIN, and T. POPA. Shader metaprogramming. In *Proceedings of the 2002 ACM/SIGGRAPH Graphics Hardware Workshop*, pages 57–68, 2002.

[18] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[19] Lance A. Miller. Programming by non-programmers. *International Journal of Man-Machine Studies*, 6(2):237–260, 1974.

[20] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings,*, pages 425–432. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[21] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Siggraph 2001, Computer Graphics Proceedings,*. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2001.

[22] Randi J. Rost. *OpenGL(R) Shading Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[23] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

[24] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[26] G. M. Weinberg. *The psychology of computer programming*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

ACKNOWLEDGMENTS

I would like to take this moment to thank the people in my life that, without their friendship, guidance, and support, this work would never have been completed. Dr. Dirk Reiners for giving me the opportunity to work on such a great project and keeping me headed in the right direction; Dr. Adrian Sannier for showing me that the success of a project depends on much more than the talent and effort of the programmers, advice that will stick with me throughout my career; and Dr. Gary Leavens for his advice, in the classroom and out, on how to be a better scientist, engineer, and person. I would also like to thank, in no particular order:

- Andres Reinot, for letting me bounce ideas off of him throughout the research's evolution.
- my girlfriend, Laura Manor, for supporting me in these stressful times and making me smile when I needed it the most.
- my parents, Gregg and Sue Austin, and the rest of my family, who were always there for me, even when school was the last thing that I wanted to deal with.
- Bryan and Jessica Walter, Jared Knutzon, and Jon Heseman, without whom, I would not have even started graduate school.
- Michael Van Waardhuizen, Benjamin Garvey, and Christopher Tasler for being great friends throughout college and after.
- all of my friends in the indie game development community, to whom I owe both my ability to argue cogently and years of entertainment.