An Embedded Shading Language

by

Zheng Qin

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

Author's Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modern graphics accelerators have embedded programmable components in the form of vertex and fragment shading units. Current APIs permit specification of the programs for these components using an assembly-language level interface. Compilers for high-level shading languages are available but these read in an external string specification, which can be inconvenient.

It is possible, using standard C++, to define an embedded high-level shading language. Such a language can be nearly indistinguishable from a special-purpose shading language, yet permits more direct interaction with the specification of textures and parameters, simplifies implementation, and enables on-the-fly generation, manipulation, and specification of shader programs. An embedded shading language also permits the lifting of C++ host language type, modularity, and scoping constructs into the shading language without any additional implementation effort.

Acknowledgements

I would like to thank my supervisor Michael D. McCool for his help throughout my study in Waterloo. I was always very excited by his continuous stream of new ideas. It was my great pleasure to study under his supervision.

Many thanks to my readers Craig Kaplan and Mark Giesbrecht for giving very detailed comments, regarding both the grammar and the organization of my thesis.

I enjoyed working in CGL very much. I would like to thank all CGL members for the help they offered. Especially, I would like to thank Kevin Moule for his willingness to help in many different problems.

Also, I would like to thank my previous roommates and friends Rodolfo Gabriel Esteves Jaramillo, Willem van Heiningen, and Kim Sehoon very much for staying up very late. I always had company no matter how late I was working. Gabriel was always there for help whenever I had a question, Sehoon did all the snow shovelling during the winter and lawn mowing in the summer, and Will's car took us wherever we wanted to go. I would also like to thank another previous roommate Rick Leung for being very considerate and doing all the cooking.

Thanks to CITO and NSERC for their financial support of my research work.

Finally, I would like to thank my parents, my husband and my daughter for their invaluable support.

Contents

List of Figures

Chapter 1

Introduction

The purpose of computer graphics rendering is to create pictures or animations with computers. These pictures or animations can be used in entertainment products, such as movies and games, architectural applications, such as urban planning, medical applications, such as volume rendering of the data taken by CT or MRI, and also technical research, in which computer simulation can be used to simulate situations where it is precarious for real people to work. Applications of computer graphics are surely not limited to the above areas. They are found in a wide variety of areas from daily life to academic research.

The purpose of the research described in this thesis was to develop an embedded real-time shading language. In section 1.1, we describe the role of shading in graphics. Section 1.2 introduces the architectures of graphics accelerators. In particular, we emphasis the role of programmable shading units in modern accelerators. Our shading language targets these programmable shading units. In section 1.3, we motivate the use of a high-level shading language. Finally, section 1.4 summarizes

contributions of this thesis.

1.1 Shading

For different reasons, either artistic or technical, some of the renderings need to be based on real-world physics. Users want objects rendered as realistically as possible. For example, in movies, many scenes of ocean storms, fires, earthquakes, and other natural disasters are simulated by computer graphics. The rendering of these scenes has to be based on the physics of the natural phenomena to make the scenes realistic. Because of the complexity of the physics, the simulations are normally simplified to speed up the computations. But the renderings have to be convincing enough. There are also some renderings that need special rendering effects that have nothing to do with the underlying physics. They can be used either artistically for striking effects or used technically for users to understand better the data rendered. No matter what kind of rendering, we need to figure out how to assign colour to each pixel on the surface of an object in the scene. This is called shading [37].

Shading is a very important area in computer graphics. It is also a very complicated and time-consuming procedure. Imagine we need to render a scene with a few objects and a few light sources. For each light source, we have to figure out the effects of the light rays on each point on the objects. A point light source will shoot rays in all directions. So all the directions have to be taken care of. When a single light ray hits an object surface, it can be reflected, refracted and diffracted. These reflected, refracted or diffracted rays become new light sources. Theoretically, this

procedure will go on forever. Since the objects can be made of different materials, the objects may have various reflection, refraction and diffraction features. For an area light source, we have to compute the effect integrated over the area of the source. To make the scene more convincing, normally techniques such as, bump mapping, displacement mapping, and anti-aliasing are applied to generate realistic images.

1.2 Graphics Hardware Acceleration

For some rendering tasks, time is not a big problem although it is not a pleasant thing to wait a long time for the final results. We can still perform the rendering with software only. But for real-time rendering, a software-only rendering system usually will not be fast enough, and we need the help of hardware accelerators.

A hardware accelerator can dramatically speed up rendering. It can implement special graphics algorithms, such as rasterization and compositing, in the hardware. Also, a hardware accelerator can have an architecture tuned to the needs of graphics. In particular, the data access patterns for graphics applications can be predicted, so a graphics accelerator does not have to depend so much on cache.

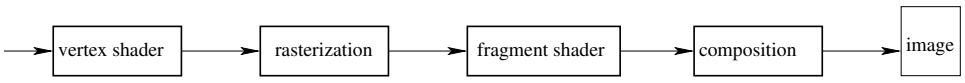

Figure 1.1: Architecture of graphics accelerator

A hardware accelerator is generally organized in a pipeline. Normally, the primi-

tives represented by triangles are set up by the application program. These triangles are sent to one end of the hardware accelerator and the color for each pixel comes out at the other end of the hardware accelerator. A rough illustration of the data flow in the hardware accelerator is shown in Figure 1.1.

Triangle data, including position, normal, and some other parameters of each vertex, are sent into the accelerator. First, they enter the vertex shader, which performs a set of computations either fixed in hardware or coded in software. The vertex shader performs independent computations on each vertex's data. The computations can be vertex transformations, normal transformations and normalizations, texture coordinate generation or any other arithmetic computations to compute the values needed in the following module.

Next, since the final goal of rendering is to find a color for each pixel in the final image, we need information for the pixels in between the vertices. To compute the color for each pixel, two approaches can be used. The first approach is to compute the color in the vertex shader and interpolate it. This is known as Gouraud shading. The second approach is to interpolate the parameters of the lighting model, then compute the color for each pixel in a fragment shading unit. This is known as generalized Phong shading. Gouraud shading ignores some important information such as surface normal when it interpolates the pixel colors, so it is faster. But the disadvantage is that it makes the images appear flat across each triangle, and causes artificial edges along the triangle borders. Whereas Phong shading can render curved surfaces more smoothly. Therefore Phong shading is more realistic than Gouraud shading.

This brings up the next module in the accelerator architecture: the rasterization module. The main job of rasterization is to figure out which pixels each triangle covers, and then interpolate the triangle vertex data to get corresponding data for each pixel. The rasterizer also does some clipping or culling if the triangles are outside of the view, perspective projection, etc. Then all the interpolated data are sent down to the next module: the fragment shader.

Just as in the vertex shader, the fragment shader is also a computation either fixed in hardware or coded in software that performs some arithmetic operations and/or texture lookups to compute the color for each pixel in the final image with the interpolated data passed in from the rasterization. For instance, the fragment shader can evaluate a lighting model to implement Phong shading.

The last module in the pipeline is the composition module that performs alpha testing, stencil testing, depth testing, blending and dithering to generate the final image and put it into the framebuffer.

When graphics accelerators were first developed, the functions of the vertex shader, the rasterizer, the fragment shader and the composition module were all fixed in the hardware. The rendering abilities were quite restricted to the fixed rendering functions provided by the hardware. Then came more flexible accelerators that added programmability in some important modules, such as in the vertex shader and the fragment shader [7, 27].

Programmability of modules means the functions of some modules in the accelerators are no longer fixed. Users can write their own programs and load the programs into these modules to control how the modules should work. Of all the

modules in the accelerators, the modules that actually determine shading are the vertex shader and the fragment shader. Although rasterization or composition can be implemented in different ways, they do not affect the shading effects very much. Generally the most efficient algorithms are fixed in hardware in these two modules to make the rendering fast. So the programmable modules in the hardware accelerators are basically the vertex shader and the fragment shader. The first programmable hardware accelerator was developed in 1992 on the Pixel-Planes 5 research graphics machine by University of North Carolina, and the first commodity graphics hardware that had similar abilities appeared in 2001 with the NVIDIA GeForce3. A programmable graphics accelerator is now called a GPU (Graphics Processing Unit). Research in this area has been quite active. New techniques have been found to implement sophisticated lighting models using a relatively small number of programmable operations [17, 18, 20, 21, 30, 37] and high-level shading languages have been developed or modified to target the GPU [15, 6, 29, 38, 41].

Computation in the vertex shader and the fragment shader in the GPU are both performed in a streaming SIMD (Single Instruction Multiple Data) fashion. Data streams enter the shaders, and programs are executed element by element producing an output stream. Computations on each element are independent and so can be executed in parallel. Also, shaders do not support branches, so all parallel executions can use the same instruction stream.

Some data required by the shader programs vary from element to element, some are unchanged during the rendering of a set of primitives or don't change at all. Data for shaders are therefore classified into different categories according to the

frequency of their modification. For instance, one way of classification will put the data that vary with element into a *varying* category, and others into *uniform* category. This classification of data will be talked more about later when high level shading languages are introduced.

1.3 High Level Shading Languages

Graphics accelerators provide assembly languages for users to write their own programs for both the vertex shader and the fragment shader. However assembly language programs are hard to develop, debug and maintain, which prevents many users from using them. Some of these assembly languages have never been used by any users except the developers themselves. Thus, people started to develop high level shading languages [2]. The main advantages of high level shading languages are that all the hardware details and optimizations are hidden from the users, so that users can focus on the applications only and work more efficiently. Also, the hardware itself has diverged from the assembly language standards, so the existing assembly languages are really an intermediate representation. There is no point in performing extensive low-level optimizations on an assembly program if it will be transformed by the hardware driver, so we may as well use a high level language.

When users employ a rendering system to write their own applications, they normally use one high level shading language for the programmable shaders and other programming languages for the rest of the application. For example, the programmable shaders could be written in Cg [28], while the rest of the application is written in C and OpenGL. There are quite a few high level shading languages.

Among them, the RenderMan shading language [6, 15] is the most popular offline (non-realtime) shading language. Some other shading languages are the Pfman shading language [35], the Interactive Shading Language(ISL) [43], the Stanford Real-Time Shading Lanugage [42], the OpenGL 2.0 shading language [2], and the Microsoft High Level Shading Languages [33]. Some of these shading languages will be introduced in more detail in the next chapter.

1.4 Contributions

In this thesis, we designed and implemented an embedded real-time shading language called Sh. Sh is built on top of C++. We implemented the Sh compiler, including some optimizations; designed and prototyped the basic language features, including data types, operators, and a library; explored ways of using C++ features to organize the Sh code; explored ways of lifting error checking to compile time; and finally, we implemented two advanced texture data types using the basic Sh features to experiment with the abstraction mechanisms made available by our language.

Chapter 2

Shading Languages

In this chapter, we will discuss a few well-known high-level shading languages relevant to our work. Also, we will talk about our work and its comparison with previous shading languages. Unlike general programming languages that target the CPU, shading languages that target the GPU have a much shorter history. The time and effort devoted to the development of GPU languages cannot be compared with those for the CPU. We cannot expect GPU languages to be as sophisticated and mature as CPU languages. However, to develop shading languages for the GPU, the simple idea of transplanting a general programming language to a shading language is not feasible for two reasons. First, GPUs have some special features, such as operations on tuples instead of scalars, texture lookups, stream computation. These features are not reflected in the design of general-purpose programming language. Second, some features of more general programming languages are not supported by the GPU. For instance, current GPUs don't support branches, and compiled programs are restricted to a certain length. Despite the differences between GPU

and CPU architectures, basic compiler principles still apply to the development of shading languages.

We will survey the features of existing shading languages. The following shading languages will be introduced roughly in chronological order of development. Some earlier shading languages were not designed for GPUs, but have inspired more recently developed GPU-oriented languages.

2.1 RenderMan Shading Language

RenderMan [45, 1] is a software rendering system designed by Pixar Animation Studios for high quality, photo-realistic image synthesis. It can be considered as an interface between modelling programs and rendering programs. The RenderMan shading language is an essential part of the rendering system designed to provide customization of the shading and lighting functions. Shaders are used to simulate different materials, the distribution of light in an environment, and the effects of special lenses and film response.

Like GPUs, RenderMan uses a SIMD model of execution. Surfaces are tesselated into small pixel-sized fragments which are shaded in parallel. In fact, the original RenderMan interface targeted a hardware implementation, although today RenderMan implementations are mostly performed in software.

2.1.1 Language Features

Data Types

The RenderMan shading language is a C-like shading language. The `float` type is the only basic data type for all scalar variables. There are also composite data types like `color`, `point` and `normal` that are composed of groups of `floats`. The shading language also has a data type for 4×4 matrix internally represented by 16 `floats`; this matrix type is used to represent 3D affine and projective transformations. One-dimensional arrays of all data types are also supported.

Data Type Modifiers

There are two data type modifiers in the language: `uniform` and `varying`. The `uniform` modifier indicates the variables that are constant over a portion of surface being shaded. The `varying` modifier indicates the variables that have different values at different locations on the surface being shaded. These modifiers are necessary to optimize SIMD execution.

Operators

Arithmetic operators ($+$, $-$, $*$, $/$), vector operators (cross product, dot product) and C conditional expressions are defined for the corresponding data types. On matrices, $*$ is used for matrix multiplication.

Control Constructs

The language supports `if-then-else`, `for`, `while` and `do` control constructs with some restrictions on boolean expressions. It also has three domain specific block statement constructs: `illuminance`, `illuminate` and `solar`. The constructs `illuminate` and `solar` are used to specify directional properties of light sources. The `illuminance` construct performs integration over all incoming lights and accumulates the lighting contributions.

Functions and Libraries

Built-in or user-defined functions are called as in the C programming language. The RenderMan shading language has a large library that provides mathematical functions (such as basic math functions, derivative functions and noise functions), shading, colouring and lighting functions (such as ambient illumination functions, Phong illumination functions and ray tracing functions). Map access functions (such as texture access functions, environment map functions and shadow map functions) and geometric functions (such as point component access functions, vector normalization functions and refraction functions) are also provided. Functions are in-lined and parameters are passed by reference.

Shaders

In the RenderMan rendering system, there are six kinds of shaders. The keywords for these shaders, `light`, `surface`, `volume`, `transformation`, `displacement` and `imager`, must be placed in front of shaders to distinguish shaders from normal

functions. Each shader has an implicit parameter list that is actually a list of global variables that have default values. When users instantiate a shader, they can set new values to the parameters for their special cases or just use the default values. There are two kinds of data in this implicit parameter list. One is `uniform`: the data don't change over the whole shading surface. The other is `varying`: they vary with the position on the surface shaded. The results of shaders are put into global variables.

Compilation

Shaders are placed in files separate from the user of the RenderMan system. These files will be compiled by the RenderMan shading language compiler into machine code and then put in a place where the rendering program can find them. When the rest of rendering program is compiled, if there are RenderMan Interface commands invoking shaders, the appropriate shaders will be found and linked with the rendering program.

2.1.2 Example

Figure 2.1 is an example shader written in RenderMan shading language.

2.2 Pfman Shading Language

The Pfman shading language [35] was designed especially for PixelFlow graphics hardware developed by the University of North Carolina in 1995. In 1992 the University of North Carolina designed Pixel-Planes 5, a research hardware testbed,

that supported real-time rendering using a high-level shading language, but only for simple shaders. PixelFlow evolved from Pixel-Planes 5 plus a technique called deferred shading first used by Whitted and Weimer [47]. It was the first graphics system that supported hardware-accelerated rendering for arbitrarily complex shaders.

```
/*
 * plastic(): give the appearance of a plastic surface
 */

surface
plastic(
       float ks              = .5,
             kd              = .5,
             ka              =  1,
             roughness       = .1;
       color specularcolor   =  1)
{
       point Nf = faceforward(normalize(N), 1);
       point V = normalize(-1);

       Oi = Os;
       Ci = Os * (Cs * (Ka * ambient() + Kd * diffuse()) +
             specularcolor * Ks * specular(Nf, V, roughness));
}
```

Figure 2.1: RenderMan surface shader for a plastic appearance

2.2.1 Language Features

Data Types

Since the Pfman shading language targeted PixelFlow, it was the first real-time high level shading language. The Pfman shading language was modeled after the RenderMan shading language, and was quite similar to it. At this time, the RenderMan shading language was used extensively in movie production. The developers of Pfman wanted to make it easy to port RenderMan shaders to PixelFlow.

However, Pfman also has some improvements and differences. First, it has a new fixed-point data type in addition to a floating-point data type. In PixelFlow, the pixel processor did not support floating-point operations directly. The floating-point operations were done by combining a few integer operations plus shifting to align the decimal points. A single precision floating-point is represented by 4 bytes. It can represent numbers from 10^{-38} to 10^{38}. For operations with small numbers (from 0 to 100, for instance) and low precision (integer computation, for instance), it is much more efficient to use small fixed-point numbers (represented by 1 byte, for instance) rather than floating-point numbers (represented by 4 bytes). Users could therefore declare fixed-point data and specify the bits for the integer part and the bits for the fraction. In cases requiring high precision over a narrow range of values, fixed point types make more efficient use of memory.

Unlike the RenderMan shading language, that has types like `point`, `color` and `normal`, the Pfman shading language used simple tuples of floating-point or fixed-point numbers instead. RenderMan shaders also have implicit parameter lists for input and output data. In contrast, all parameters in Pfman shading language have

to be declared explicitly.

Control constructs are supported in the Pfman shading language. However, since Pfman also used a SIMD execution model, implementation of these control constructs could be inefficient.

Shaders

In the PixelFlow rendering system, there are several shader types. However, only two of them are actually implemented in hardware: the `surface` shader and the `light` shader.

Compilation

The compilation of the Pfman shading language has three stages. In the first stage, the compiler transforms the shading program into a C++ program. Every PixelFlow rendering node has both a serial microprocessor and a parallel SIMD array. In the second stage, a C++ compiler cross-compiles this program into machine language for the serial microprocessor. When this machine code runs, it executes the uniform computations directly and generates an instruction stream for the SIMD array to execute the varying computations. Several optimizations are performed to minimize storage cost in the SIMD array, which is the main resource limit of the PixelFlow architecture.

The RenderMan system is a complete rendering system that supports all stages of rendering from setting up the models to obtaining the final image. The separate shaders written in the RenderMan shading language are invoked by API function calls in the rendering system. The Pfman shading language does not have such a

complete rendering system. It accomplishes rendering by working with OpenGL. OpenGL is an interactive API that also supports the whole rendering pipeline from establishing models to generating the final images except (at the time PixelFlow was implemented) that it did not support arbitrary shaders. Pfman makes OpenGL support arbitrary shaders by adding some new API calls to OpenGL.

2.2.2 Example

Figure 2.2 is a portion of a Pfman shader for a brick wall.

```
    // figure out which row of bricks this is (row is 8-bit integer)
fixed<8,0> row = tt/height;
    // offset even rows by half a row
if(row % 2 == 0) ss += width/2;
    //wrap texture coordinates to get "brick coordinates"
ss = ss % width;
tt = tt % height;
    // pick a color for the brick surface
float surface_color[3] = brick_color;
if(ss < mortar || tt < mortar)
    surface_color = mortar_color;
```

Figure 2.2: Code from a simple Pfman brick shader

2.3 ISL Shading Language

2.3.1 Language Features

While shading languages like Pfman were developed to extend the programmability of OpenGL, SGI, who designed OpenGL, also developed their own high level shad-

ing language for procedural shaders. This language, called the Interactive Shading Language (ISL) [43], is built on top of existing OpenGL capabilities. The ISL compiler does not translate files written in ISL into machine code directly, but instead converts ISL files into pass description files. These pass description files will then be translated into C/OpenGL code. So the ISL compiler is indeed a compiler that converts the ISL language into C/OpenGL code. Each OpenGL call can be seen as a parallel SIMD operation. Combinations of OpenGL rendering passes can achieve arbitrary shading computations. ISL is known as a multi-pass shading language.

ISL is not a full-featured shading language like the RenderMan and the Pfman shading languages. This was because of the limitations of existing SGI OpenGL hardware. One example of such a limitation is that texture coordinates for texture lookups could not be computed values. Another limitation is that when users define shaders, they could not declare their own parameters if the parameters change with position. If the users do want such a parameter, they have to put it in a texture map and do a texture lookup to get it.

Data Types

Data types in ISL are quite similar to those in the RenderMan shading language. ISL also has only `float` for all scalars, has type `color`, `point`, and `vector` composed of groups of `floats`, has a 4×4 matrix composed of 16 `floats`, and allows 1D arrays of the above data types.

Data Type Modifiers

ISL has three modifiers for its variables: `varying`, `parameter`, and `uniform`. The `varying` data change with different positions, the `parameter` data change only with different surfaces or frames, and the `uniform` data will not change after the shader is compiled. The allowable arithmetic operations for `varying`, `parameter` and `uniform` variables are different. A `uniform` variable can be converted to a `parameter` variable and a `parameter` variable to a `varying` variable.

Control Constructs

ISL supports control constructs `if/else` and `repeat` that will run the `repeat` body for a certain number of times. However, `if/else` results in inefficient execution and `repeat` only supports a constant (non-data dependent) number of iterations.

Functions and Shaders

Functions and shaders in ISL are of the same format. Each has a return type, a function name, a formal parameter list, and a body. The return type can be one of any of the ordinary types or one of the following shader types: `surface`, `atmosphere`, `ambientlight`, `distantlight` and `pointlight`. If the return type is a shader type, the function is actually a shader, otherwise it is an ordinary function.

A big difference between ISL and most other shading languages is that ISL allows more than one `surface` shader. These `surface` shaders are applied one by one and their results can be composited together.

2.3.2 Example

A simple ISL shader is shown in figure 2.3.

```
surface shadertest(
    uniform color c = color(1, 0, 0, 1);
    uniform float f = .25;
{
    FB = diffuse();
    FB *= C*f;
    return FB;
}
```

Figure 2.3: A simple ISL shader

2.4 Stanford Real-Time Shading Language

The Stanford Real-Time Shading Language (RTSL) [42] is another high level shading language designed on top of OpenGL, but this time targeting OpenGL extensions for modern GPUs that support programmable vertex and fragment units.

2.4.1 Language Features

Data Types

RTSL is very similar to C. Compared with the RenderMan shading language, Pfman and ISL, RTSL adds a boolean type `bool`, a texture reference type `texref` and `clamped` types (for instance `clampf1` is a floating-point scalar clamped to [0-1]). RTSL does not have data types for `color`, `point`, or `vector`, instead it has generic

data types that have more than one component, such as `float3` and `float4`. RTSL also supports 3×3 and 4×4 matrices for transformations.

Data Type Modifiers

In RTSL, variables are classified into four categories indicated by four modifiers: `constant`, `primitive group`, `vertex`, and `fragment`. The `constant` modifier means the variables will not change after the shaders are compiled, so it cannot be assigned any new value later. The `primitive group` modifier means the variable can be assigned a value for each primitive group, but not more frequently. The `vertex` modifier means the variables can be different for each vertex and the `fragment` modifier means the variables change per pixel. RTSL calls these four categories *computation frequencies*; `constant` has the lowest computation frequency and `fragment` has the highest computation frequency.

There are restrictions on what operations can be performed at certain frequencies. For instance, matrix-matrix multiplication must be at `primitive group` computation frequency. Theoretically, it is not wrong to put matrix-matrix multiplication at the `vertex` or `fragment` frequency. However, matrix-matrix multiplication normally is constant for each primitive group. It is not necessary to repeat the operation for each vertex or fragment.

When users declare variables, they can put modifiers in front of the variable types. Otherwise the system will figure out the computation frequencies of the variables automatically. The variables will be computed in the appropriate computation frequencies.

RTSL supports explicit type conversion and implicit promotion of types. RTSL

also supports computation frequency conversion from low computation frequency to high computation frequency, but not vice versa.

RTSL provides some functionality borrowed from C++. It supports function overloading similar to C++, conditional compilation (it has seven C preprocessor directives) and global variables. RTSL does not support control constructs.

Operators

RTSL has the basic math operators such as $+$, $-$, $*$, / for different types, a special operator "{}" to group scalars into vectors or vectors into matrices, and functions to access one or more components in a vector. However, assignment to vector components is not allowed. Grouping scalars into vectors is not only for the convenience of users, but also for taking advantages of graphics hardware features. The registers in the graphics hardware are normally 4-tuples. Operations for four unrelated scalars are less efficient than grouping the scalars into a vector and performing one operation.[1]

RTSL has a blend function to blend two 4-component vectors together with respective coefficient for each vector. RTSL also has comparison operators and a conditional select operator. Operator precedence is the same as in C.

[1] Assuming the hardware actually operates on tuples. Unfortunately, hardware developers do not release the internal details of their hardware and also perform optimization of assembly code submitted to their API. There is some indication that modern GPUs are actually multithreaded VLIW (Very Large Instruction Word) machines, in which case the driver does extensive rescheduling of parallel operations anyway.

Libraries and Functions

RTSL provides a library that has math functions for scalars, vectors and matrices, texturing and lookup functions, and some other functions. All user-defined functions must be defined before they are used, since RTSL does not support function declaration. Functions are called just as in the C language. RTSL has a function `integrate` that has similar semantics to the `illuminance` construct in the Renderman shading language. The function `integrate` evaluates the accumulated value of an expression over all light sources.

Shaders

RTSL has two kinds of shaders: `surface` and `light`. In the high level shading languages that we have introduced so far, logical modules do not map one-to-one to the GPU architecture. GPUs mainly have only programmable vertex shaders and fragment shaders. The shading languages that we will introduce later will have logical shaders that do map one-to-one to GPU structures.

Shaders are basically the same as functions, except that the return types of shaders have to be modified by `surface` or `light`.

Compilation

RTSL is connected to OpenGL by a set of interfaces designed by RTSL. These API extensions are prefixed by `sgl` in OpenGL.

Shaders are in files separate from the rest of the program. The `surface` shader and `light` shader are first loaded. Then the `surface` shader and corresponding

`light` shaders are combined together. Before this combined shader can be used, the combined shader has to be compiled for the graphics hardware. The compilation of shaders happens at run time because the compilation depends on the hardware which may be different on different runs of the application.

Example

Figure 2.4 is a simple Stanford RTSL shader for a diffuse lighting model.

```
// useful constants

constant float4 Zero = 0, 0, 0, 0;
constant float4 Black = 0, 0, 0, 1;
constant float4 White = 1, 1, 1, 1;

constant float pi = 3.14159;

surface float4
lightmodel_diffuse (float4 a, float4 b)
{
    perlight float diffuse = dot(N, L);
    perlight float4 fr = select(diffuse > 0, d * diffuse, Zero);
    return a * Ca + integrate(fr * C1);
}
```

Figure 2.4: A simple diffuse Stanford RTSL shader

2.5 OpenGL 2.0 Shading Language

OpenGL 2.0 [3, 4] will support a shading language, which has been nicknamed `glslang`. This language was originally designed by 3DLabs, Inc. Ltd. but is

now being standardized by the OpenGL Architecture Review Board (ARB). The OpenGL 2.0 shading language tries to incorporate the good features, such as vector and matrix computations, from previous shading languages to make graphics computations more convenient for users. It also added more C/C++ features to form a full real-time shading language. Many of the OpenGL 2.0 features are similar to features of previous shading languages.

Unlike previous shading languages that have supported logical models not matching the GPU model, OpenGL 2.0 supports a logical model with only vertex and fragment shaders which matches perfectly the vertex and fragment shader in the GPU. This model makes compilation simpler since the effort of splitting, reorganizing or combining logical shaders into GPU shaders is avoided. In OpenGL 2.0, vertex shaders and fragment shaders are loaded directly into the corresponding modules in the GPU.

2.5.1 Language Features

In terms of general language features, OpenGL 2.0 supports conditional compilation, C/C++ style comments, structures, arrays, control constructs, C-like scoping, C-like operator precedence, function calls, function declaration, constructors (just like the constructors in C++ that initialize instances from constructor parameters), conversions between variable types. Note that some of these features still have restrictions. For instance, the preprocessor directives in OpenGL is only a subset of those in C/C++, the `if/else` control construct cannot be nested, and users cannot define their own constructors. Also it is not clear if the OpenGL 2.0 shading

language can be supported on current GPUs. In OpenGL fashion, if a given shader cannot be compiled to hardware, then an implementation is free to execute it in software—likely at a unacceptable cost in performance.

In terms of graphics, OpenGL 2.0 supports arithmetic operators and comparison operators for vectors and matrices, more complex variable types including texture handles, composition of vectors and matrices from scalars or vectors, vector and matrix component access, and so on.

Data Type Modifiers

There are two categories of modifiers for data types in the OpenGL 2.0. One category includes `const`, `uniform`, `attribute` and `varying`. These modifiers (except `const`) are mainly used for global variables. Their functions are similar to the modifiers in other shading languages. The `const` modifier is for variables that never change. The `uniform` modifier is for variables that are constant for the primitives rendered but may change between shader bindings. The `attribute` modifier is for vertex-based variables. The `attribute` variables (data about each vertex) are passed from OpenGL to vertex shaders, processed by the vertex shader and the results are written into `varying` variables. These `varying` variables are then interpolated and copied to the corresponding `varying` variables in fragment shader. The fragment shader reads from its own input `varying` variables and processes them. So the `varying` variables are actually the interface between the vertex shader and the fragment shader.

Communication between application programs, OpenGL, the vertex shader and the fragment shader in OpenGL 2.0 is handled through global variables. For exam-

ple, vertex data are passed from application programs to the vertex shader through some built-in global `attribute` variables. Data between the vertex shader and the fragment shader are transmitted through global `varying` variables as long as the declaration of the output `varying` variables in the vertex shader matches the declaration of the input `varying` variables in the fragment shader.

Another category of modifiers including `in`, `out`, `inout` are for function parameters. The `in` modifier indicates read-only function parameters, the `out` modifier indicates write-only function parameters, and the `inout` modifier indicates both read and write function parameters. Parameters are basically passed by reference, since functions are to be implemented by in-line code expansion.

Compilation

Just as in RTSL, the connection of shaders with other part of the OpenGL rendering pipeline is accomplished by an OpenGL API extension. Source code for a shader is stored in a separate file. Before a shader can be used, the API loads the shader into a shader object. The shader object is then compiled and attached to a program object. The program object is then linked to generate an executable that can be run on the programmable unit in the OpenGL pipeline. This executable can be chosen when the rendering needs it. The loading, compiling and linking of shaders are all done at run time.

Example

The OpenGL 2.0 shading language example in Figure 2.5 is a light map applied to a base texture map with linear fog.

2.6 Cg (C for Graphics)

Cg (C for Graphics) is a high-level shading language developed by NVIDIA. It was developed in parallel with the OpenGL 2.0 shading language.

2.6.1 Language Features

Generally speaking, Cg has the same logical model as OpenGL 2.0 because both support only vertex shaders and fragment shaders. The language features of Cg are also similar to that of OpenGL 2.0. However, there are certainly some differences between these two languages. Cg is independent of API, while OpenGL 2.0 is very specific to OpenGL. But the differences are not so basic as to cause big differences in functionality.

```
void main (void)
{
    float fog;
    vec3  color;
    vec3  baseMap = texture3(0, gl_TexCoord0);
    vec3  lightMap = texture3(1, gl_TexCoord2);

    color = baseMap * lightMap;

    fog = (gl_FogEnd - abs(gl_EyeZ)) * gl_FogScale;
    fog = clamp(fog, 0, 1);

    color = mix(color, gl_FogColor, fog);
    gl_FragColor = vec4(color, 1);
}
```

Figure 2.5: OpenGL 2.0 example: a light map applied to a base texture map

Cg introduced the concept of language profiles to distinguish the different capabilities of vertex processor, fragment processor and of different commodity graphics hardware. By choosing a certain profile, a subset of the full Cg language is chosen such that this subset of language is fully supported on a particular hardware platform. Functions in Cg can also be overloaded on profiles, providing a simple mechanism to specify shaders specialized for different hardware platforms.

Cg also includes a mechanism for defining the input/output structure of each shader in a `struct`. These structs are then bound to attributes, and shared definitions are used to pass data between vertex and fragment programs.

2.6.2 Compilation

Shaders written in Cg are also put in separate files. The compiling and linking of Cg shaders to application programs can be done at either compile time or run time. Run time compiling and linking are supported by Cg's runtime library, as in OpenGL 2.0.

2.6.3 Example

Figure 2.7 is a simple Cg vertex shader program that calculates diffuse and specular lighting. The input-output specification for this shader is given in Figure 2.6.

2.7 Sh Shading Language

All previous shading languages put the shader program in a separate string or file and then implement a relatively traditional compiler to convert this specification to a machine language representation. Using a separate language has some advantages. For instance, a "little" language can be more tightly focused and can leave out unnecessary features while supporting a domain specific syntax. However, binding the shader program to the application program can be a nuisance. In fact, this binding code can easily be longer than the shader itself.

Our contribution is the development of a high-level shading language, Sh, that is not really a separate language, but a library embedded in C++. Since it is an embedded shading language, the binding of the shader program with the application is implicit and controlled by C++ scoping rules. Also, all the modularity constructs

```
// define inputs from application
struct appin
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
};

// define outputs from vertex shader
struct vertout
{
    float4 Hposition : POSITION;
    float4 Color0 : COLOR0;
};
```

Figure 2.6: An input-output specification for a simple Cg vertex shader

and syntax of C++ are available for use in Sh programs.

Shading languages are domain specific languages (DSLs) designed specifically

```
vertout main(appin In,
             uniform float4x4 ModelViewProj : C0;
     uniform float4x4 ModelViewIT : C4;
     uniform float4 LightVec)
{
    vertout out;

    Out.Hposition = mul(ModelViewProj, In.Position);

    // transform normal from model space to view space
    float4 normal = normalize(mul(ModelViewIT, In.Normal).xyzz);
    // store normalized light vector
    float4 light = normalize(LightVec);
    // calculate half angle vector
    float4 eye = float4(0.0, 0.0, 1.0, 1.0);
    float4 half = normalize(light + eye);
    // calculate diffuse component
    float specular = dot(normal, half);
    // calculate specular component
    float specular = dot(normal, light);
    specular = pow(specular, 32);
    // blue diffuse material
    float4 diffuseMaterial = float4(0.0, 0.0, 1.0, 1.0);
    // white specular material
    float4 specularMaterial = float4(1.0, 1.0, 1.0, 1.0);
    // combine diffuse and specular contributions
    // and output final vertex color
    Out.Color0 = diffuse * diffuseMaterial
               + specular * specularMaterial;
    return Out;
}
```

Figure 2.7: A simple Cg vertex shader program that calculates diffuse and specular lighting

for computer graphics. The evolution of shading languages has followed the general evolutionary path of all domain specific languages. Normally a DSL is first designed to capture precisely the semantics of an application domain. However, users of DSLs always find that some general programming language features are very helpful. So eventually a DSL ends up as a general programming language with extra features in a special domain. This has also been the case in the development of shading languages. From the very beginning, shading languages followed the syntax of the C programming language with some additional semantics to support computer graphics, such as vector and matrix operations. Shading languages are now borrowing more and more general programming language features from C/C++, such as structures, function definition and overloading, preprocessing directives. So we can expect that future shading languages will be languages like C/C++ with extra application features in computer graphics.

The evolution of a DSL also tells us that to develop a DSL that has just the semantics of an application domain can be quick, but to make the DSL have general language features will take lots of time and effort. This gave us the idea of building an embedded shading language within an existing language. This saves us from developing general language features so we can focus on the semantics of computer graphics. The general language features of the existing host language are inhereted "for free". Since the host language is normally well developed and mature, and still being improved by its own developers, the newest features of the host language will always be available to the users of such a DSL. An API is also a language, so our contribution can also be thought of as developing an API with the expressiveness

of a shading language.

Embedding a DSL within an existing language is not a new idea [10, 12]. For example, SQL database queries are often embedded in C. Embedded DSLs have been implemented by using specilized preprocessors [19], or by taking advantage of language features to build pure embedded DSLs. Embedded DSLs implemented the second way are actually libraries constructed on top of the existing language. The C++ template mechanism, for example, has been used to construct "active libraries" which will generate more efficient code because some of the computations are done by the preprocessor when the templates are instantiated. Embedded DSLs have also been popular in functional programming languages like Haskell or ML because some of their features, such as higher-order functions, lazy evaluation, polymorphism and type classes, make the implementation of the embedded DSLs easy. In the past, it was hard to build embedded languages in imperative languages. Sh uses both macros (preprocessing) and operator overloading in C++ to build an embedded DSL for graphics.

Sh is built on top of standard C++ features. The decision to embed the shading language within C++ has three motivations. First, existing shading languages all follow the syntax of C++ and are borrowing more and more features from it, so why not use C++ directly? Second, most real-time graphics applications are written in C++. Third, the features of C++, such as classes, operator overloading, templates and inheritance, have made it possible to construct an expressive embedded language. The operator overloading feature in C++ in particular is crucial in the construction of Sh. Unlike conventional language compilers that use a lexical

analyzer and a parser to generate tokens and parse trees, the Sh compiler uses operator overloading to construct parse trees directly. This saves lots of work but the code generation that follows can use any methods that are applicable to general compilers. Since the embedded shading language is actually a shading library in C++, the Sh shaders can be put anywhere in the application program and don't need to be in separate files. The embedding of Sh in the application also gives new power to the application, which can now easily generate custom shaders on the fly. The compilation of an application program is just a one-pass compilation using C++ compiler, unlike the other shading languages introduced before that have multi-pass compilation.

In this thesis we describe the design and development of a prototype Sh implementation. This prototype was targeted at a GPU simulator called Sm [31], because the commercial graphics cards didn't support control constructs or floating point at that time and we wanted Sh to target the new generation of graphics cards. In the next chapter, we will introduce our GPU simulator.

It should be noted that all features in this simulator are now basically supported by GPUs, and the current version of Sh targets real GPUs. However, we will focus on the prototype for the thesis and discuss the current state of Sh in the conclusion.

Chapter 3

Testbed

The Sh prototype targets a software GPU simulator called Sm, implemented by Michael McCool, Kevin Moule and Stefanus Du Toit. When the Sh prototype was implemented, commercial graphics cards did not support control constructs or floating point values. The new graphics products support floating point values but not control constructs—yet. However, indications are that the next iteration of Direct3D will require control constructs, and this will drive hardware to support them. The Sm testbed tries to predict new directions in graphics hardware. Sm is similar to the NVIDIA graphics hardware API except that it has new features such as control constructs and noise functions. We did not want Sh to be obsolete when it was released, so we targeted the testbed whose features we expected to see in commercial graphics cards later.

Another reason the Sh prototype targeted the testbed instead of real hardware is that when the Sh was first developed, there were not many optimizations in the compiler. The assembly programs generated were definitely not efficient or concise.

While commercial graphics cards have limitations on the length of shaders, our testbed supports shaders with arbitrary length. So it was easy for us to develop Sh on Sm at an early stage. However, the new version of Sh has targeted real hardware.

The architecture of the Sm testbed will be roughly introduced in this chapter. Hopefully, it will help our readers understand the Sh shading language better.

3.1 Architecture of Testbed

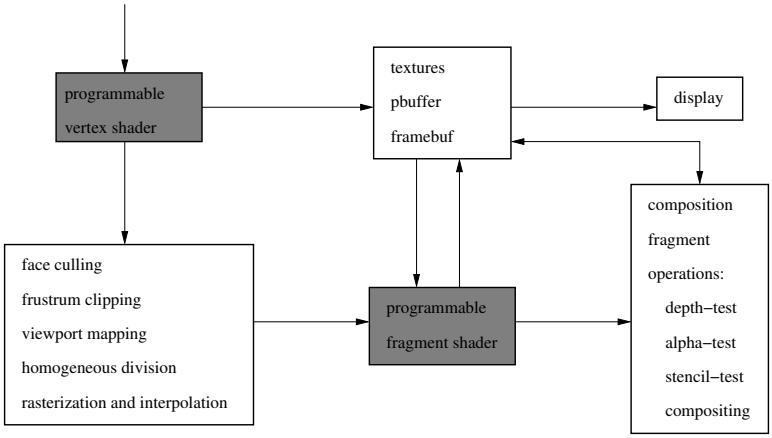

Figure 3.1: Architecture of testbed

The architecture of the Sm testbed is shown in Figure 3.1. The shaded blocks in the figure are the programmable modules and the other blocks are fixed-function modules.

3.1.1 Vertex Shader

The processing of application program data by the Sm testbed starts from the vertex shader. Vertex data streams are sent to the programmable vertex shader by the rendering system. In the vertex shader module, operations such as vertex position and normal transformation, vector normalization, texture coordinate generation, advanced lighting models, or advanced vertex operations for special effects are performed. At this stage, no culling or clipping is performed to remove vertices from the data stream. Each vertex has its own corresponding input and output data. Computations on one vertex cannot use data from neighboring vertices.

3.1.2 Rasterization and Interpolation

The output from the vertex shader is then sent to the next fixed-function module. Here, vertices are assembled to reconstruct the triangles (primitives).

Processing primitives that are expected to be invisible is wasteful. Before rasterization and interpolation, hidden primitives are eliminated to speed up processing. Face culling discards the primitives that are facing away from the viewer. Frustum clipping, in general, clips off the primitives that are out of the viewing frustum and may possibly generate new triangles if the primitives intersect with the viewport boundaries. Clipping may not be a separate path from rasterization. As a matter of fact, clipping is not done in Sm. Sm uses a clipless rasterization algorithm [32].

Rasterization and interpolation are then applied to the triangles. Data for each pixel on a rasterized triangle, including pixel coordinates, normals, texture coordinates are interpolated from the triangle vertex data. Interpolated pixel data may

also go through further procedures such as homogeneous division, which performs the perspective projection, and viewport mapping, which maps the pixel coordinates onto the 2D screen.

3.1.3 Fragment Shader

The fragment shader uses the interpolated pixel data to compute the color for each pixel on the screen. The computation of colors may involve arithmetic operations and texture lookups. The results of the fragment shader are sent to the composition module for further processing.

3.1.4 Composition

At the composition stage, the depth-test, the alpha-test, and the stencil-test are applied to the output data from the fragment shader to generate the color for each pixel. The depth-test determines the pixels that are closest to the viewer. These pixels are not hidden by other pixels and will appear on the screen. The alpha-test eliminates pixels whose alpha values do not pass a certain threshold. The stencil-test compares a reference value with the contents in a stencil buffer, and further operations are only applied to the pixels that pass the comparison. The results of the composition module are sent to the framebuffer in the memory module, which is then read by the display device.

3.2 Vertex/Fragment Shader Architecture

In our work, all we care about are the programmable modules in the graphics cards. So we will only focus on the vertex shader and fragment shader from now on. Here, we will introduce the architectures of the programmable modules, and their API features.

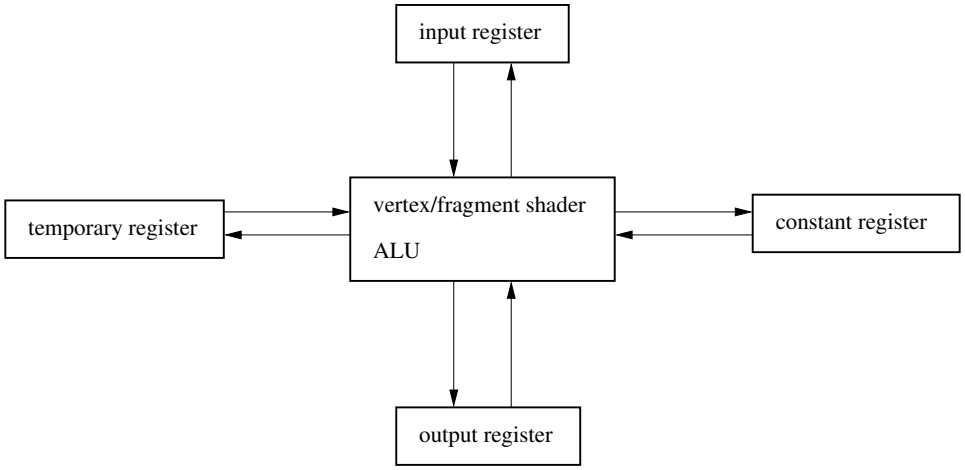

Figure 3.2: Architecture of vertex/fragment shader

Normally in commercial graphics cards, such as ATI and NVIDIA graphics cards, the architectures of vertex shader and fragment shader are designed differently to maximize overall performance because the vertex shader and fragment shader have distinct requirements and functionalities. But in Sm, the vertex shader and fragment shader were designed in a general way without distinguishing the differences between them because we expect this will be the trend in hardware design. As the vertex shader and fragment shader become more and more powerful, their functionalities will eventually converge. The architectures of the vertex shader and

fragment shader in Sm are shown in Figure 3.2. A vertex/fragment shader has an ALU and four kinds of registers: `input` registers, `output` registers, `constant` registers and `temporary` registers. Input data come in from the `input` registers and final results are written to the `output` registers. Sm instructions are function-call based.

3.2.1 Registers

In Sm, registers have to be declared and allocated before they are used. Each register in Sm is a four-tuple of floating-point numbers. Operations on registers are component-wise. Unlike real GPUs, Sm has an unlimited number of registers of all types, and all are both readable and writable.

Users can give any names to registers, or even allocate arrays of registers. In the following illustration, we will assume preallocated arrays of registers for the convenience of explanation: `iR` for input registers, `oR` for output registers, `tR` for temporary registers and `cR` for constant registers.

Input registers hold incoming attribute data; these are the varying parameters of the shader. Output registers hold outgoing attribute data; these are the results of the shader. Note that in Sm (and in Sh) shaders can have multiple outputs. Constant registers are loaded explicitly by the host. They hold uniform parameters. Finally, temporary registers hold intermediate results.

3.2.2 Swizzling and Write Masking

Registers in Sm support both swizzling and write masking. The swizzling operation accesses and rearranges components of a register. The write masking operation writes only certain components of a register. The letters "`x`", "`y`", "`z`" and "`w`" are used respectively to represent the first, second, third and forth components in a register. A string composed of the four kinds of letters grouped in arbitrary order is used for swizzling and write masking. If the string is appended to a register read from, the string is used for swizzling; if the string is appended to a register written into, the string is used for write masking.

3.2.3 Instructions

Most Sm instructions have the form `sm`*OP*`(dest, src1, src2)` or `sm`*OP*`(dest, src)`. There are also a few instructions with four parameters or one parameter. *OP* is the type of operation which can be `ADD` (addition), `MUL` (multiplication), `DIV` (division), `MAX` (maximum), and so on. Parameters `src1`, `src2` and `src` are the source registers read from. Parameter `dest` is the destination register written into.

The instruction `smMOV(oR[1]["xy"], iR[2]["xx"])` is an example with both swizzling and write masking. String "`xx`" is for swizzling and "`xy`" is for write masking. This instruction copies the first component of input register `iR[2]` to the first and second components of output register `oR[1]`.

Instruction `smADD(oR[1], iR[1], iR[2])` is an example that adds the values in register `iR[1]` and `iR[2]`, and puts the result in register `oR[1]`.

Sm shader instructions are similar to NVIDIA and OpenGL ARB shader in-

structions, except that in Sm, a shader can have unlimited number of instructions.

3.2.4 Control Flow

Sm supports branches. The following instructions are used for flow control.

command	parameters	function
`smLBL(`n`)`	n: index of label	declare a label
`smJR(`d`)`	d: register for label	jump to label stored in d
`smBNE(`d`, `l`)`	d:register for label l:register compared with 0	if(l != 0) goto label stored in d
`smBE(`d`, `l`)`	d:register for label l:register compared with 0	if(l == 0) goto label stored in d

3.2.5 Texture Access

For texture lookups, there are no special texture registers in Sm as in other commercial graphics cards. Texture lookup is done with the constant registers in the following steps.

1. Declare a texture object.

```
T1 = smNewTexture2D(res, channel, SM_FLOAT);
```

Parameter `T1` is the texture object returned by the function, parameter `res` is the resolution of the texture, parameter `channel` is the number of channels in the texture, and parameter `SM_FLOAT` is the data type in the texture.

2. Load texture data into the texture object.

```
smTexImage2D(T1, 0, databuffer);
```

Parameter `T1` is the texture object object, and the `databuffer` parameter is a pointer to a buffer holding the texture data.

3. Bind the texture object to a texture unit in Sm.

```
smBindShader(SM_TEXTURE_UNITn, T1);
```

Parameter `SM_TEXTURE_UNITn` is an integer indicating the number of the texture unit in the GPU. Parameter `T1` is the texture object.

4. Texture lookup in a texture unit.

```
smTEX(creg, reg1, reg2);
```

Parameter `creg` has the texture unit number. If texture lookup is done in texture unit `SM_TEXTURE_UNITn`, the content of constant register `creg` has to be the same as `SM_TEXTURE_UNITn`. Parameter `creg` must be a constant register. Parameter `reg1` is the register for the color returned by the texture lookup. Parameter `reg2` is the register for texture coordinates. Register `reg1` can be any type of register except constant. Register `reg2` can be any types of register.

3.2.6 Noise Functions

Sm provides noise functions to generate procedural textures. Procedural textures are functions that compute color for each pixel in the texture with respect to the texture coordinates. The advantage of a procedural texture is that no texture buffer is needed to store the color for each pixel. All we need is the function. Procedural textures save the memory resources in GPU. The following noise functions are

supported in Sm:

```
SMuint smDNS1(SMreg dest, SMreg src1);
SMuint smDNS2(SMreg dest, SMreg src1);
SMuint smDNS3(SMreg dest, SMreg src1);
```

These functions implement 1D, 2D and 3D Perlin noise functions respectively. No current GPUs support noise functions, although they can be simulated.

3.2.7 Shader Definition

After we know the registers and instructions in Sm, the last step is to use them to construct a shader. Here we use the example in Figure 3.3 to give the readers an idea of what an Sm shader looks like.

First, the registers used in the shader are declared and allocated. The Sm function `smDeclareShader` returns a shader object. The parameter `ShaderModule_ID` indicates whether the shader is a vertex shader or a fragment shader. Vertex shaders are indicated by 0, and fragment shaders are indicated by 1. The shader object is passed as parameter to the `smShaderBegin` function which begins the shader definition. The instructions that follow define the body of the shader. The shader definition ends with the `smShaderEnd` function.

Multiple vertex/fragment shaders can be loaded into the GPU. But only one vertex/fragment shader is active at a time. When a shader is loaded into the GPU, it is made active by default. To reactivate a shader, the `smBindShader` function is used.

```
SMreg oPos, v0, v1, r0, r1;

oPos = smOutputReg(0);
v0 = smInputReg(0);
v1 = smInputReg(1);
r0 = smReg();
r1 = smReg();

shader = smDeclareShader(ShaderModule_ID);
smShaderBegin(shader);

    smDP3(r0, v0["xyz"], v1["zyx"]);
    smADD(r1, v0, r0);
    smMUL(r0, v0, r1);
    smSUB(oPos, r0, r1);

smShaderEnd();
```

Figure 3.3: Example of Sm shader

Chapter 4

Sh Compiler

Almost all shading languages developed previously are independent languages with their own compilers. Most of the time, programs written in shading languages are put in separate files. They are compiled with their own compilers first, then the compiled code is linked with the rest of the application program. The compilations normally involve two passes.

Our goal is to develop a shading language that is powerful, expressive, easy to develop, easy to learn, and easy to use. The powerful features of C++ made our goal possible. The Sh shading language is an embedded shading language built in C++. In addition to domain specific features of its own, Sh gets many powerful C++ features for free. We also use the operator overloading function of C++ to construct the parse tree for arithmetic expressions automatically, which saves us from constructing a parser from scratch. Since C++ is already a popular programming language, users of C++ can learn the Sh shading language very quickly, because the Sh shading language is in fact a C++ library. Also, Sh shaders do not

have to be separated from the application. They can be put anywhere in a C++ program, and compilation is a one-pass process.

In this chapter, we will explain how the prototype Sh compiler was constructed. We start with the parse tree construction and code generation. Then we introduce how the memory in Sh is managed to avoid memory leaks. Last, we introduce the optimizations that the Sh prototype supported.

4.1 Parsing

Since Sh is embedded in C++, most of its syntax features are inherited from C++. For instance, a semicolon is used to end a statement, curly braces are used to define scopes, and functions are called the same way as in C++. The main difference between embedded Sh shader statements and normal C++ statements is that at run time Sh shader statements are not computed immediately in the CPU. Instead, side effects of Sh shader statements and constructors are used to generate a parse tree, that keeps all the information about the shader. At the end of the shader definition, optimizations and code generation are applied to generate code for the GPU. This code is then downloaded to the GPU.

Statements in Sh can be classified into two categories: expressions and control constructs. The parsing of expressions and control constructs are different in Sh.

4.1.1 Expression Parsing

Expression parsing in Sh is accomplished by using operator overloading in C++. Operators, including arithmetic operators, such as “+”, “−”, “∗”, “/”, logic oper-

ators, such as "|", "&", ">", "!", and the assignment operator "=", use overloading to delay execution. Instead, a simple parse tree is constructed for each operator. For example, the expression $a + b$ will not compute the sum of variables a and b immediately. Instead a parse tree as in Figure 4.1 will be generated. The parent node "+" points to its two child nodes a and b. An operator node may have two child nodes or one child node depending on whether the operator is binary or unary. At the end of the shader, machine code will be generated according to this parse tree and eventually the sum of a and b will be computed in the GPU.

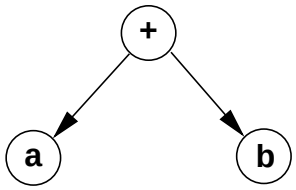

Figure 4.1: Parse tree for $a + b$

All operators are overloaded in a similar way except the assignment operator. The assignment operator "=" is overloaded not only to generate a parse tree, but also has the side effect of putting onto the current token list a token referring to the assignment node. This token list is used to keep all the statement tokens in an Sh shader. When the shader definition is finished, all the tokens, including expression tokens and control construct tokens, are read from the token list to generate a complete parse tree for the shader. Code generation is applied to this complete parse tree.

As an example of assignment, the expression $d = a + b * c$; generates a parse tree as in Figure 4.2. An expression token referring to the assignment node is pushed

into the token list. When the tokens in the list are read from the token list, they are combined together to generate a complete parse tree. The parse tree for this expression will become a part of the general one.

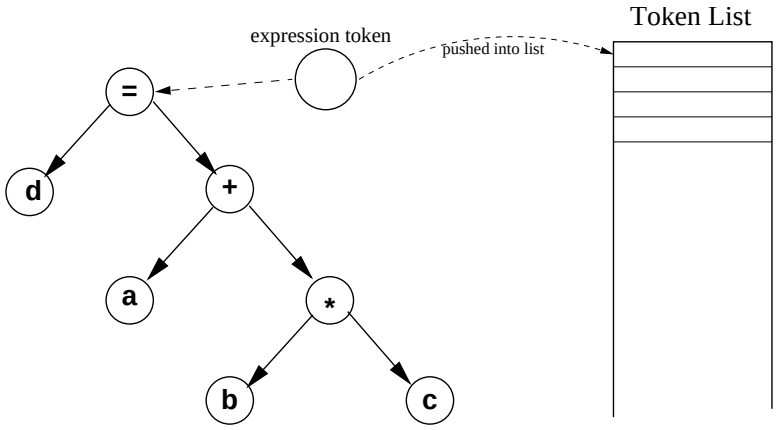

Figure 4.2: Parse tree for $d = a + b * c$

The precedences of the operators cannot be modified from the C++ standard. The operator precedences in Sh are naturally consistent with the ones in C++. In this example, operator "$*$" has higher precedence than "$+$", so "$*$" is parsed earlier than "$+$". The parse tree for "$*$" is deeper than that for "$+$" in the parse tree. The order of the parse tree generation is "$*$", "$+$", and "$=$". The code generation procedure performs a bottom-up travers of the parse tree. It generates code for $b * c$ first, puts the result in a temporary register t_1, for example, then generates code for $a + t_1$, puts the result in another register t_2, and finally generates code for $d = t_2$.

Expressions (other than arguments to control constructs) without an assignment operator have no effect in the shader, therefore they are simply discarded and no

tokens are put into the list.

The copy constructors of all data type classes in Sh are rewritten so that they will generate an assignment parse tree that assigns the initial value to the newly declared variable. We can then initialize the variable when we declare it, and don't have to separate declaration and initialization. The additional assignment will be optimized away.

4.1.2 Control Construct Parsing

In Sh, we can use either C++ control constructs or Sh control constructs. C++ loops will expand the statements within a loop the corresponding number of times. Also, C++ `if/else` permits the conditional inclusion of Sh code. Use of such control constructs results in no jumps or loops in the generated GPU assembly code. In order to generate assembly code that has data dependent branches, we implemented a mechanism to specify Sh control constructs.

Control construct keywords in Sh are actually macro definitions. The control keywords are defined as follows:

```
#define  SH_WHILE(c)   shWHILE(push()&&process(c)); {
#define  SH_ENDWHILE   shENDWHILE(); }
#define  SH_IF(c)      shIF(push()&&process(c)); {{
#define  SH_ELSE       shELSE();
#define  SH_ENDIF      shENDIF(); }}
#define  SH_DO         shDO(); {{{
#define  SH_UNTIL(c)   shUNTIL(push()&&process(c)); }}}
```

```
#define  SH_SWITCH(c)  shSWITCH(push()&&process(c)); {{{{
#define  SH_ENDSWITCH  shENDSWITCH(); }}}}
#define  SH_CASE(c)    shCASE(push()&&process(c));
#define  SH_DEFAULT    shDEFAULT();
#define  SH_FOR(a,b,c) shFOR(push()&&process(a)
                           &&push()&&process(b)
                           &&push()&&process(c)); {{{{{
#define  SH_ENDFOR     shENDFOR(); }}}}}
#define  SH_BREAK      shBREAK();
#define  SH_CONTINUE   shCONTINUE();
```

Note that the "control keywords" in Sh are really function calls. They are wrapped in macros just for cleaner syntax. The curly braces "{" and "}" at the end of the macro definitions are used for compile time error detection. For example, `shFOR` has five "{" and `shENDFOR` has five "}". If `shFOR` and `shENDFOR` do not appear in a pair, the C++ compiler will complain about it. The number of braces for each pair of control keywords is unique.

Control construct functions that do not accept arguments simply insert a keyword token into the current token list. For example, `shELSE()` puts an `ELSE` token into the token list.

Each argument, for instance represented by c, in the control macros is expanded into two functions in the corresponding functions: `push()` and `process(`c`)`. These two functions, which always return *true*, are linked together with operator &&. In C++, the operator && guarantees its two operands are executed from left to

```
shader definition {
     expressions before WHILE loop...
     SH_WHILE(c) {
          expressions inside WHILE loop...
     } SH_ENDWHILE
     expressions after WHILE loop...
} end of shader definition
```

Figure 4.3: Shader with `WHILE` loop

right. Function `push()` returns a *true* value, and so `process(`c`)` is always executed. If there are multiple arguments, the functions are linked together as in `shFOR()` with &&, such that the arguments are processed in sequence. The arguments are processed in such a way to solve some problems described below.

Suppose we are defining a shader and the shader has a current token list for storing expression and control construct tokens. For instance there is a `SH_WHILE` loop in the shader, as is shown in Figure 4.3. The `SH_WHILE` loop keyword takes one argument c. The argument c is normally an expression such as $a >= b+d$ *or* $a\ != b$. The loop is ended by `SH_ENDWHILE`.

We would like the tokens generated for this shader to be in the order shown in Figure 4.4a. The `WHILE` token separates the expression tokens before `SH_WHILE` from c expression tokens. The "{" token separates the c expression tokens from the expression tokens inside the loop body, and the "}" token separates the expression tokens inside the loop from the expression tokens after the loop. If each group of tokens is separated from other groups, it is then easy to construct a general parse tree with these tokens and generate GPU assembly code.

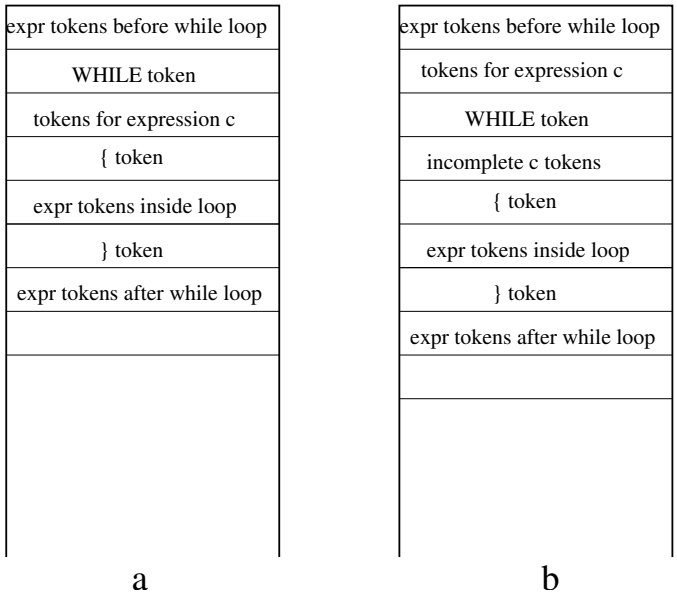

Figure 4.4: Tokens in the token list

To achieve this, the `shWHILE(`c`)` function was originally designed to push a `WHILE` token into the token list, then push tokens generated for expression c, and finally push a "{" token to indicate the end of the c expression tokens and the beginning of the expression tokens inside the loop. Function `shENDWHILE()` was designed to push a "}" token into the token list to end the `WHILE` loop. Tokens for expressions before, inside, and after `WHILE` loop are pushed into the token list automatically because of assignment operator overloading.

The problem with this method is that when the C++ compiler sees the `shWHILE(`c`)` function, it evaluates the argument before it executes the function. When the c expression is evaluated, any statment or assignment will push tokens into the token list. This does not happen for simple expressions, but expressions can contain function calls which can issue arbitrary sequences of statements. In this case, the

tokens in the token list will be as in Figure 4.4b. Extra tokens required for the evaluation of c now appear before the `WHILE` token in the list. This situation arises frequently. For instance, the arithmetic operations sometimes involve matrices. Matrix operations normally generate multiple assignment tokens.

There is no way to tell how many expression tokens before `WHILE` are extra ones. Even worse, if a control function has multiple arguments, like `shFOR(`a`, `b`, `c`)`, the C++ compiler expands the three arguments before `shFOR` is executed and puts the tokens for all three of them into the list. Tokens for a, b and c are piled together. It is impossible to tell which tokens belong to which argument. Another problem is that different C++ compilers may evaluate arguments in different orders. The parameters a, b and c could be processed from left to right or in reverse order. The straightforward way to do control constructs, simply pushing a control keyword token, does not work. Therefore, we introduce the functions `push()` and `process(`c`)`.

To solve these problems, we use the data structure shown in Figure 4.5. Each rectangular block is a struct that has a pointer to its parent struct, a token list for storing tokens and a child stack for storing pointers to its child structs.

If a keyword macro has no argument, we simply push a keyword token into the current token list. If a keyword macro has an argument c, the argument is expanded by the macro preprocessor into two functions connected with the "&&" operator: `push()`&&`process(`c`)`. The "&&" operator guarantees `push()` is executed before `process(`c`)`. The function `push()` creates a new child struct and sets the current struct and token list to this child struct and token list. The function `push()`

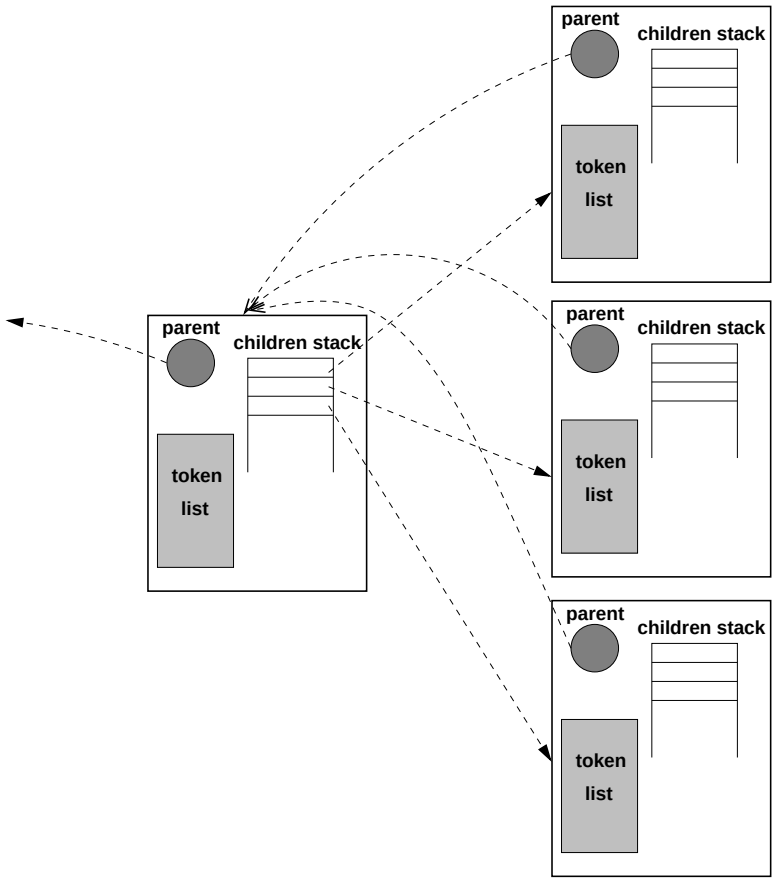

Figure 4.5: Data structures in control constructs

returns a *true* value to force the execution of `process(`c`)`. When the C++ compiler evaluates c, the tokens for c are now pushed onto the new child token list. Tokens for c are thus separated from the previous tokens and the whole set of c tokens can be used later. Function `process(`c`)` simply sets the current struct and token list back to the previous parent struct and token list and returns a *true* value.

Now the function `shWHILE()` pushes a `WHILE` token into the current token list, moves tokens from the child token list for argument c to the current token list and pushes a "{" token. Function `shENDWHILE()` still pushes a "}" token into the token list. The tokens in the final token list are now the same as in Figure 4.4a.

Other control flow functions are implemented in a similar way. Here we will only deliberate on the `shFor` function a bit more to show how a function with more than one argument works.

In the `shFOR(`a`, `b`, `c`)` macro, each argument is processed in the same way and therefore the tokens for each parameter are put into three separate child token lists. Theoretically, the control construct functions can have any number of arguments by using the data structure in Figure 4.5. The `push` and `process` functions for each argument are also connected with "&&" operator so that we can control the processing order of a, b and c. At the end, the `shFOR` function rearranges the segments of code into the correct order.

By implementing control constructs this way, and using a recursive descent parser as introduced in the next section, Sh can support a full set of control constructs. Theoretically, it allows unlimited number of control construct embeddings and also allows the keyword arguments to have functions with control constructs

inside them.

4.1.3 Parse Tree Generation

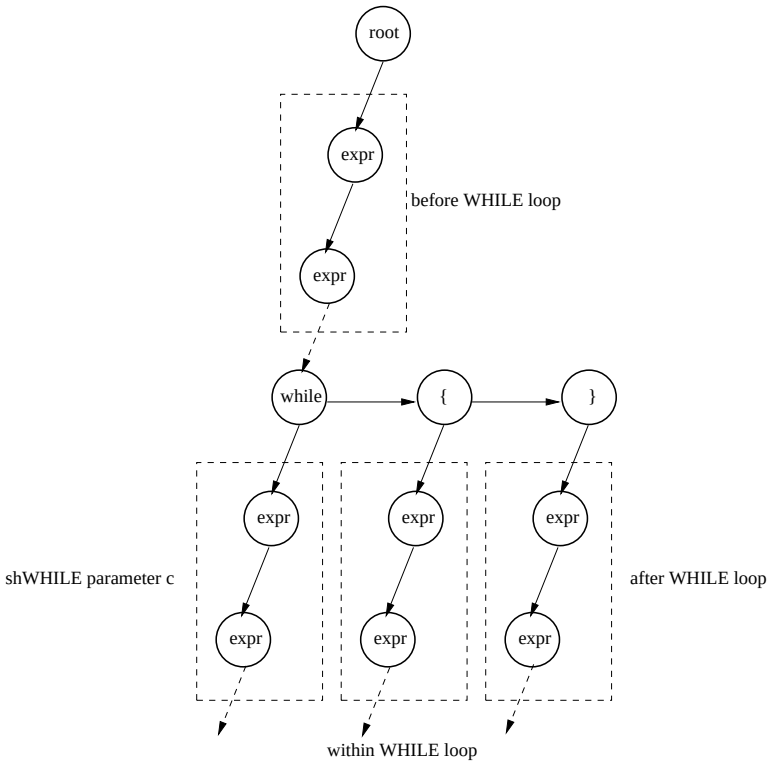

Figure 4.6: Parse tree for a shader with a WHILE loop

After the shader definition is closed, the tokens are stored in one token list. A recursive descent parser is then applied to construct a general parse tree for the shader. The parse tree for a `WHILE` loop is shown in Figure 4.6.

If there are no control constructs, assignment tokens are linked in sequence. Each token is set to the child of the previous token. Each expression token points to parse trees for its left and right expressions. The parse trees for expressions are

generated automatically by operator overloading. The emphasis of the recursive descent parser is to parse the control constructs. For a set of control functions, for instance `shWHILE` and `shENDWHILE`, all the keywords generated by them, in this case keywords `WHILE`, "{" and "}", are put in sequence as brothers. The expressions for arguments, conditional or loop bodies, and the expressions after the control constructs are set as the children of the corresponding keywords. As a matter of fact, each expression token in the above example can be substituted by a control construct, which makes the parse tree support arbitrary combinations of the control constructs.

4.1.4 Code Generation

Code is generated via a top-down travers of the parse tree. The code for a parent is always generated before the code for a child. If a node in the parse tree has both a child node and a brother node, the child node has higher priority than the brother node. The code for the control keywords generates either labels or branches.

Semantically, Sh control constructs work just as C++ control constructs do. Sh supports `if/else`, `while`, `do/until`, `for`, `switch`, and also supports `break` and `continue`. Arbitrary nesting of control constructs is allowed.

Syntactically, Sh differs slightly from C++. Sh has `SH_END...` keywords for ending control constructs, while C++ does not. Sh separates the arguments in `SH_FOR` with "`,`" instead of "`;`" as in C++.

Although Sh has implemented the `SH_SWITCH/CASE` control construct, it is now not in use, because the GPUs do not support integers, and the `SWITCH/CASE` control

construct needs integer comparisons. This notwithstanding, we still implemented the syntax for the `SWITCH/CASE` control construct in Sh in order to have a complete set of control constructs for Sh, and if integer operation are supported in GPUs later, the `SWITCH/CASE` will be right there for use.

4.2 Memory Management

When Sh shaders are defined, they require some memory resources. Users may declare variables outside shaders (uniform parameters) or inside shaders; the parse trees are constructed, and some internal data structures are allocated to store information. The memory taken by the variables and the shaders must be reclaimed when they are no longer needed. For simplicity's sake, we need to free users from explicit memory management. Therefore, in Sh, reference-count smart pointers are used to clean up memory automatically. This simple garbage collection scheme depends on the fact that all data structures in Sh are directed acyclic graphs.

References to shaders and variables are maintained via smart pointers. For variables, each variable points to a variable node, which keeps the reference count of the variable. When there are new references to the variable, the reference is increased automatically, and when the referring objects are killed, the C++ destructors of these refering objects automatically decrease the reference count and delete the variable as necessary. References to shaders work the same way. Shaders may be referenced for more than once because they can be involved in shader algebra operations. When a shader is killed, all the memory taken by this shader is automatically released.

4.3 Optimizations

In the Sh prototype, we don't perform complete optimizations because of the time and effort required. But we did try to do some optimizations on constant registers which are very important with limited resources. Our optimizations are really just a beginning. Much more work remains to be done in the future.

4.3.1 Optimizations on Constant Variables

Since constant registers are a very limited resource, we implemented a few optimizations on them.

In Sh, there are three kinds of "constants": uniform parameters declared outside shaders, numeric literals, and newly declared temporary variables. Newly declared temporary variables are regarded as constant until they are modified by non-constant variables. Initially, only the non-constant values are stored in input registers.

The following constant optimizations are implemented to save constant registers and improve the efficiency of the assembly code.

Scalar Packing

For numbers that have to be loaded into constant registers, each number uses only one component in a constant register. A constant register has four components, so one constant register can hold four numbers. Each component can be accessed by swizzling. If a number is used in more than one place, the number is put in a constant register only once. A map is used to keep track of the place where

the number is loaded. When the same number is used in an expression again, the correct constant register and component can be retrieved.

Immediate Parameter Operation

Arithmetic expressions defined outside shaders (operations on uniform parameters) and inside shaders are implemented in different ways. Computations defined outside shaders or computations define inside shaders on parameters and on constant variables are evaluated immediately by the host, and the results are used as constants inside the shaders. Arithmetic expressions defined inside shaders but on non-constant variables are not executed immediately. Parse trees are generated for such expressions.

Constant Folding

Constant folding is a way to improve run-time performance by doing constant evaluation at compile time [5, 48, 34]

If there is no constant folding optimization, each constant will be loaded into a constant register. Constant evaluations are parsed to generate assembly code. With constant folding optimization, the constant evaluations are performed on the host. Only the results are loaded into the constant registers.

Especially when all the variables on both sides of an assignment operator are constant variables, the expression is executed immediately. No parse tree and assembly code is generated for this expression. The reason Sh regards newly declared temporary variables as constant is that shorter assembly code is generated because of constant folding.

```
1    ShMatrix4x4f mata;
2    ShMatrix4x4f matb;
3    matb = inverse(mata);

4    ShShader exp = SH_SHADER_BEGIN(0); {
5        ShInputPoint3f pc;
6        ShPoint3f pa(0.2, 3.2, 1);
7        ShPoint3f pb;
8        ShOutputPoint pd;

9        pb = pa | matb;
10       pd = pb - pc;
11       pb = pc;
12       ...
13   } SH_SHADER_END;
```

Figure 4.7: Example shader for constant folding

There are often matrix operations in shaders, such as the computation of the inverse, adjoint, or determinant of a matrix. The length of the machine code for these operations can be drastically reduced by the use of immediate parameter operation and constant folding.

The segment of code in Figure 4.7 illustrates constant folding in Sh. Line 3 is executed immediately because of the immediate parameter operation. Line 9 is also executed immediately because variables `pb`, `pa` and `matb` are constant. Line 10 is not executed immediately because `pc` is an input variable. So a parse tree is constructed for this line. On line 11, `pb` is assigned by `pc` which is an input variable, so a parse tree is generated for this line and `pb` is also marked as temporary variable.

Parameter Modification

Parameters in Sh act like global variables in C++. When parameters are modified, Sh automatically reloads the new values into the GPU without further action by the users.

One parameter may be used multiple times in one shader and used in multiple shaders. Because of constant folding, a parameter may not be loaded into register directly, but the constant folding results involving the parameter are loaded. Changing a parameter may affect several constant registers in the GPU. The new values have to be loaded into the GPU before the shader is used again. We use a data structure to keep track of this information. If a parameter is used in a shader, a node list for this shader is added for this parameter to record which nodes in the parse tree are influenced by this parameter. Each node is a constant folding point in the parse tree. Sh also keeps record of the constant register each node is using. One node list is added for each shader that uses the parameter.

When the parameter is changed outside a shader, Sh marks the parameter as `modified`. When the shader is used, Sh checks each parameter to see if any of them have been modified previously. If Sh finds a parameter that has been modified, it will follow the node list for this shader in the parameter variable to recompute the values at each node in the node list. Then only the new values are loaded into the GPU. Also, if a parameter currently in use by the active shader is modified, the recomputation and reloading are done immediately.

4.3.2 Matrix Loading

When a matrix is loaded into the GPU, it may take more than one constant register depending on its dimensions. Normally the matrix is loaded row by row. The worst case is that a 4 by 1 matrix will take four constant registers using only one component in each register. To save registers, Sh stores the transpose of such matrices. Access to the transposed matrix has to be adjusted accordingly.

In the Sh prototype, 3 by 2 matrices are still stored row by row. The reason is that if the transpose is stored, it only saves one register, but may make the matrix computation more complicated, therefore generating longer code.

Chapter 5

Sh Language Features

Because Sh is embedded in C++, it basically follows the syntax of C++ so that users of Sh won't feel a big difference when they switch between the two different languages. Some features Sh inherits from C++ directly; in others, Sh tries to follow the C++ syntax. Things that are the same in both C++ and Sh include variable declaration, function definition and function calls, basic operators, using "`;`" to end a statement and using curly braces "{" and "}" for scope.

There are also some differences between Sh and the host language C++ either because we added new features for Sh which don't exist in C++ or because C++ restrictions prevent Sh from having the exact syntax of C++. For instance, Sh has new data types and new operators, and also has swizzling and write masking which C++ does not have. Semantically Sh has the same control constructs as C++. Syntactically, Sh control constructs are different from those of C++ because we cannot overload C++ keywords such as `for`, `while`, `switch` or "`;`" etc. Sh has defined `SH_FOR`, `SH_WHILE`, and `SH_SWITCH`, etc. instead. Sh also does not support

pointers, although C++ pointers can be used to manipulate Sh shaders.

In this chapter, we are going to introduce the Sh language. First, a complete shader example will be shown to give the general idea of an Sh shader. Then the main features of Sh will be introduced.

```
ShShader julia0 = SH_BEGIN_SHADER(0) {

    // declare input vertex parameters
    ShInputTexCoord2f ui;
    ShInputPoint4f pdi;

    // declare outputs vertex parameters
    ShOutputTexCoord2f uo(ui);
    ShOutputPoint4f pdo(pdi);

}
SH_END_SHADER
```

Figure 5.1: Sh Vertex shader for Julia set

Figure 5.1 is an example of an Sh shader. It is the vertex shader of a pair of shaders that computes the Julia set as a procedural texture. The variable `julia0` is the shader object. The parameter `0` of `SH_BEGIN_SHADER` indicates this is a vertex shader. This shader does not compute the Julia set itself, but simply copies the input texture coordinates and point position to the output texture coordinates and point position.

5.1 Data Types

Sh supports basic data types, such as `int`, `float` and `double`. Internally, these data types will be promoted respectively to Sh data type `ShAttrib1i`, `ShAttrib1f` and `ShAttrib1d`. The `boolean` data type is not yet supported in Sh. The Sh data types include:

```
ShAttrib[1234]*

ShVector[1234]*
ShNormal[1234]*
ShPoint[1234]*
ShPlane[1234]*
ShColor[1234]*
ShTexCoord[1234]*
ShTexture[123]D*
ShTextureCube*

ShMatrix[1234]x[1234]*

ShInput*
ShOutput*
```

`ShAttrib[1234]*` is the basic data type in Sh. The numbers 1, 2, 3 and 4 indicate whether the variable is a scalar, a 2-tuple, a 3-tuple or a 4-tuple variable.

The `*` is replaced with the storage type of the variable. It can be `i` for integers, `f` for single precision floating-point or `d` for doubles, etc. as in OpenGL. For example, `ShAttrib2f` is a 2-tuple single precision floating-point variable. In the Sh prototype, only single precision floating-point numbers were implemented because they are the only values supported by Sm.

In the data type design of Sh, the `ShAttrib[1234]*` are the super classes of all other classes. Most of the features of `ShAttrib[1234]*` are inherited by other classes.

`ShVector[1234]*` are the data types for vectors.

`ShNormal[1234]*` are the data types for normals.

`ShPoint[1234]*` are the data types for points.

`ShPlane[1234]*` are the data types for planes.

`ShColor[1234]*` are the data types for color.

`ShTexCoord[1234]*` are the data types for texture coordinates.

`ShTexture[123]D*` are the data types for texture maps.

`ShTextureCube*` are the data types for cubic texture maps.

Other than the basic texture data types, we also derived data types for sparse textures and for textures that support cubic interpolations. More detail will be given in Chapter 6.

`ShMatrix[1234]x[1234]*` are the data types for matrices. The first number indicates the number of rows and the second number indicates the number of columns. For example, `ShMatrix2x3f` means a matrix with 2 rows and 3 columns of floats.

The results of operations between variables of the same data type has the the same data type. The results of operations between variables of different data types has data type `ShAttrib[1234]*`. We chose this rule to keep the language simple and easy to learn. We also do not restrict operations based on type. For instance, two points can be added although this has no geometric meaning.

Some data types have special rules for certain operations. For example, point and vector data types are promoted to corresponding homogeneous data of higher dimensions automatically when homogeneous data are required in matrix operations. However, these exceptions are only applied when otherwise there will be an error under the usual rules. In the case of homogeneous promotion, transformation of a `ShVector3f` by a `ShMatrix4x4f` would otherwise result in a type mismatch.

Since `float` and `double` are converted to `ShAttrib1*` implicitly, any operations on `ShAttrib1*` can be applied with one of the operands being a constant number.

Variables declared with the above data types are temporary variables that will be given temporary registers in the GPU. The `Input` and `Output` modifiers in the data types indicate the variables are input variables or output variables. Input variables are allocated input registers and output variables are allocated output registers. For instance, `ShInputVector3f` declares a variable as input vector that has 3 components.

5.2 Parameters and Local Temporary Variables

Uniform parameters and varying local temporary variables use the same data types, but work differently. Parameters are constant inside shaders. They are declared

outside shaders and can only be modified outside shaders. Any attempts to modify parameters inside shaders will cause error messages. When assembly code is generated, parameters are allocated constant registers. Local temporary variables are first regarded as local constant variables until they are modified by non-constant variables (any expression depending on an input variable is non-constant). Then they are regarded as temporary variables. The registers for local temporary variables are allocated accordingly.

5.3 Operators

There are two kinds of operators in Sh: general operators borrowed from C++ and Sh operators. General operators include arithmetic operators, compound assignment operators, increment and decrement operators, relational operators, logic operators and explicit type casting operators. Additional Sh operators include linear algebra operators, texture lookup operators, swizzling operators, and writemasking operators.

5.3.1 General Operators

Arithmetic Operators

Arithmetic operators such as $+$, $-$, $*$, $/$, etc. are component-wise operators for all types, including matrices. Note that $*$ does *not* mean matrix multiplication. We also do not restrict operators based on type. The $\hat{}$ operator is used for exponentiation, since this is a common operation in lighting models.

Normally, these operators only apply between tuples of the same size. However,

a 1-tuple is promoted by duplication to match tuples of any size on the other side of an arithmetic operator. Since floats and doubles are converted to 1-tuples, these operators can also be used with numeric constants.

operator	left operand l	right operand r	function
$+$	n tuple $n \times m$ matrix	n tuple $n \times m$ matrix	component-wise $l + r$
$*$	n tuple or scalar $n \times m$ matrix or scalar	n tuple or scalar $n \times m$ matrix or scalar	component-wise $l * r$
$/$	n tuple $n \times m$ matrix	n tuple or scalar $n \times m$ matrix or scalar	component-wise l/r
$-$	n tuple $n \times m$ matrix	n tuple $n \times m$ matrix	component-wise $l - r$
unary $-$		n tuple $n \times m$ matrix	component-wise $-r$
^	scalar	scalar	l^r

Increment and Decrement Operators

These operators increment or decrement a 1-tuple by 1.

operator	left operand l	right operand r	function
prefix $++$		scalar	$r = r + 1$
postfix $++$	scalar		$l = l + 1$
prefix $--$		scalar	$r = r - 1$
postfix $--$	scalar		$l = l - 1$

Compound Assignment Operators

Compound assignment operators are component-wise operators to modify variable in place.

operator	left operand l	right operand r	function
$+=$	n tuple $n \times m$ matrix	n tuple $n \times m$ matrix	component-wise $l = l + r$
$-=$	n tuple $n \times m$ matrix	n tuple $n \times m$ matrix	component-wise $l = l - r$
$*=$	n tuple $n \times m$ matrix	n tuple $n \times m$ matrix	component-wise $l = l * r$
$/=$	n tuple $n \times m$ matrix	n tuple $n \times m$ matrix	component-wise $l = l/r$

Relational Operators

Relational operators are component-wise operators for evaluating the relationship between two operands. Because we do not yet support booleans or integers, these operators return a `ShAttrib1f` containing either the value 1.0 or 0.0.

operator	left operand l	right operand r	function
$<$	n tuple $1 \times n$ matrix $n \times 1$ matrix	n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise $l < r$
$<=$	n tuple $1 \times n$ matrix $n \times 1$ matrix	n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise $l <= r$
$>$	n tuple $1 \times n$ matrix $n \times 1$ matrix	n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise $l > r$
$>=$	n tuple $1 \times n$ matrix $n \times 1$ matrix	n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise $l >= r$
$==$	n tuple $1 \times n$ matrix $n \times 1$ matrix	n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise $l == r$
$!=$	n tuple $1 \times n$ matrix $n \times 1$ matrix	n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise $l != r$

Logic Operators

Logic operators are component-wise boolean operators. They act correctly on tuples defined such that each component holds either a 1.0 or 0.0. However, "&&"

can also be used to compute the minimum and "||" can also be used to compute the maximum in other contexts.

operator	left operand l	right operand r	function
!		n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise $1 - r$
&&	n tuple $1 \times n$ matrix $n \times 1$ matrix	n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise "min" $\min(l, r)$
||	n tuple $1 \times n$ matrix $n \times 1$ matrix	n tuple $1 \times n$ matrix $n \times 1$ matrix	component-wise "max" $\max(l, r)$

5.3.2 Sh Operators

Linear Algebra Operators

In Sh, linear algebra is supported by two operators: the matrix/vector product operator "|" and the cross product operator "&". Their functions are listed in the following table. If operator "|" is applied to a matrix and a tuple that don't have matching dimensions, special tuple promotion rules are applied to handle homogeneous coordinates. For example, for `ShMatrix4x4f | ShPoint3f`, the 3 tuple will be promoted to a 4 tuple by assigning 1 to the forth component, since it is a point; while in example `ShMatrix4x4f | ShVector3f`, the 3 tuple will be promoted to a 4 tuple by assigning 0 to the forth component, since it is a vector.

If a tuple appears on the left of a | operator, it will be interpreted as a row

vector. If a tuple appears on the right of a | operator, it will be interpreted as a column vector. This rule eliminates the need for many transposes. Let `x` and `y` be tuples, and let `M` be a matrix. Then `x|y` is the dot product of `x` and `y`, `M|y` is the transformation of `y` as a column, and `x|M` is the transformation of `x` as a row. Quadratic forms can be expressed with `x|M|x`.

operator	left operand l	right operand r	function
|	n tuple	n tuple	dot product of l and r
	$n \times m$ matrix	m tuple	tuple r is column vector
	n tuple	$n \times m$ matrix	tuple l is row vector
	$n \times m$ matrix	$m \times k$ matrix	matrix multiplication
&	3 tuple	3 tuple	cross product of l and r

Texture Lookup Operators

The texture lookup operator "`[ ]`" can only be applied to texture data types such as `ShTexture[123]D*`, `ShTextureCube*`, `ShSparseTexture2D*` and `ShCubicInterpTex2D*`. It takes one parameter of type `ShTexCoord[1234]*`. The following example illustrates how the operator "`[ ]`" is used.

```
1    ShTexture3Df tex;
2    ShTexCoord4f coord;
3    ShColor3f color = tex[coord];
```

In this example, a special rule is applied for homogeneous coordinate calculation. The variable `tex` is a 3 dimensional texture map, and `coord` is 4-tuple texture

coordinates. We divide the first 3 components of `coord` by its forth component and use the results to do the texture lookup. If the dimensions of the coordinates and texture match, this division is not performed.

Swizzle and Writemask Operators

The "`( )`" operator is used as the swizzling and writemasking operator. It can be applied to both tuples and matrices. It is used in single pairs with tuples and double pairs with matrices. When it is used with matrices, the first pair of "`( )`" is used for selecting rows of the matrices and second pair of "`( )`" is used for selecting columns. An empty "`( )`" with no argument means the whole row or whole column is selected. The operator "`( )`" is a swizzling operator if it is used on the right hand side of the assignment operator, and a writemasking operator if on the left hand side. Pairs of "`( )`" can also be concatenated together. They are applied from left to right. The final result is the combined result of them. The following examples illustrate how the operator "`( )`" works.

```
1    ShVector4f c(0.1, 2, 3.4, 0.5), v;
2    c(1,2,3) = c(0,1,2);
        copies the first, second and third components of
vector c to its second, third and forth components.
3    ShMatrix4x4f m(array), d, f;
4    ShMatrix3x3f a = m(0,1,2)(0,1,2);
        assigns the first, second, third rows and first, second,
third columns of matrix m to matrix a.
```

```
5    d()(0) = a()(2);
        assigns the third column of a to the first
column of d.
6    f(2)() = a(1)();
        assigns the second row of a to the third row
of f.
7    v(0) = c(1,2)(0);
        assigns the second component of c to the first
component of v.
8    a = m()(1,2,3)(0,1,2)(0,1,2);
```

5.4 Control Constructs

Semantically, Sh supports the same control constructs as those in C++, and also supports arbitrary nesting of the control constructs. The Sh control construct keywords are listed in the following table.

Sh Control Construct	C/C++ Control Construct
SH_IF(c) SH_ELSE SH_ENDIF	if(c) {...}else{...}
SH_WHILE(c) SH_ENDWHILE	while(c){...}
SH_DO SH_UNTIL(c)	do{...}until(c)
SH_FOR(a,b,c) SH_ENDFOR	for(a; b; c){...}
SH_BREAK SH_CONTINUE	break; continue;

SH_SWITCH(c)
SH_ENDSWITCH
SH_CASE(c)
SH_DEFAULT
	switch(c){
	case(c): {...}
	default: {...}
	}

5.5 Compile-time Checking

In Sh, data types are C++ classes defined with templates. There are some restrictions on operations on different data types. For instance, additions of variables with different dimensions are not allowed (except for scalar promotion), the row

and column of matrices must match in matrix operations (except for homogeneous promotion) and so on. So we have to define operations on different data types. Since the data types are defined with C++ templates, it is possible to define general operations for a group of data types that work similarly, and define special operations for certain data types as well. When undefined operations are invoked in shaders, the C++ compiler will figure out that the operations are illegal and give error messages. This is a way to make the error detection happen at compile-time instead of run-time.

The Sh shading language doesn't have its own modularity constructs. It borrows C++ modularity constructs such as function calls, classes and templates to organize shaders and reuse code.

5.6 Functions

Sh functions must be declared outside shaders. The return types and formal parameter types can be basic data types like `int`, `double`, etc., or Sh data types. The actual parameters can be passed either by value or by reference.

The copy constructors of all Sh data type classes are defined to have the same effect as an assignment operator. So when the actual parameters are passed as values to the formal parameters, and when the return data are passed to the temporary variables, the constructors are called to generate assignment parse trees. Eventually, these extra assignments will be optimized away although they are not

in our prototype. Instructions inside the function calls work the same way as instructions in a normal shader. They are parsed and the expression and control keyword tokens are put into a token list.

So function calls in C++ generate inline code in Sh shaders. When a function is called, instead of creating the overhead of a function call, a copy of the function definition is placed at the point of the call. For now, this is the only function call mechanism that Sh supports. Later on, we plan for Sh to support operator overloading of "`( )`" on shader objects to support real function calls in the generated code, in anticipation of GPUs that support subroutines.

5.7 Classes

The definitions of vertex shader and fragment shader can be wrapped in a C++ class as in the following example. This example defines a **Phong** class. The vertex shader and the fragment shader are defined in the constructor of the class.

When an object of the class is declared, the constructor is called to initialize the object and the assembly code for vertex shader and fragment shader are generated on the fly. With a different constructor parameter `exp`, each instance of the **Phong** class defines its specialized shaders. The member function `Bind()` loads the shaders into the GPU. The shaders do not necessarily have to be defined in the constructor. They can be defined separately in different member functions.

The Sh parameters (global variables) are declared as public member variables so that the parameters can be accessed and modified from outside of the class. It is therefore possible for application programs to define shaders, modify parameters,

and define shaders again with the new parameters.

Garbage collection here consists of two steps. First, at the end of the shader, the shader variables are removed by C++. The destructors of these variables reclaim the memory that is no longer referenced. Second, when the lifetime of an object of the class ends, its private variables **phong0** and **phong1** are removed automatically. The destructors of these shader variables clear all the memories they are using, including the memory used by the parse tree. No explicit memory deallocation is needed.

```
ShColor3f
phong (
    ShVector3f hv;
    ShNormal3f nv;
    ShColor3f kd;
    ShColor3f ks;
    ShAttrib1f exp;
) {
    ShAttrib1f hn = (normalize(hv) | normalize(nv));
    return kd + ks * pow(hn, exp);
}

class Phong {
    private:
        ShShader phong0, phong1;
```

```
public:
    ShTexture2DColor3f kd;
    ShColor3f ks;

Phong (
    double exp;
) {
    ShShader phong0 = SH_BEGIN_SHADER(0) {
        ShInputTexColor2f ui;
        ShInputNormal3f nm;
        ShInputPoint3f pm;

        ShOutputVector3f hv;
        ShOutputTexColord2f uo(ui);
        ShOutputNormal3f nv;
        ShOutputColor3f ec;
        ShOutputPoint4f pd;

        ShPoint3f pv = modelview | pm;
        pd = perspective | pv;
        nv = normalize(nm | adjoint(modelview));
        ShVector3f lvv = light_position - pv;
```

```
        ShAttrib1f rsq = 1.0 / (lvv | lvv);
        lvv *= sqrt(rsq);
        ShAttrib1f ct = max(0.0, (nv | lvv));
        ec = light_color * rsq * ct;
        ShVector3f vv = -normalize(ShVector3f(pv));
        hv = normalize(lvv + v);
    } SH_END_SHADER;

    phong1 = SH_BEGIN_SHADER(1) {
        ShInputVector3f hv;
        ShInputTexCoord2f u;
        ShInputNormal3f nv;
        ShInputColor3f ec;
        ShInputAttrib1f pdz;
        ShInputAtrib2f pdxy;

        ShOutputColor3f fc;
        ShOutputAttrib1f fpdz(pdz);
        ShOutputAttrib2f fpdxy(pdxy);

        fc = ec * phong(hv, nv, kd[u], ks, exp);
    } SH_END_SHADER;
}
```

```
    void
    bind() {
        ShBindShader(phong0);
        ShBindShader(phong1);
    }
};
```

5.8 I/O Templates and Structs

Data types in Sh are defined in a class hierarchy. The type `ShAttrib[1234]*` is the superclass of other classes. All classes are actually defined with C++ template arguments. Each class takes two template parameters: tuple dimension and I/O type. For instance, `ShAttrib[1234]f` is defined using a `typedef` of `ShAttribf<int D, int IO>`. Parameter `D` is the tuple dimension, and parameter `IO` is the I/O type. Parameter `IO` can be `SH_INPUT`, `SH_OUTPUT` or `SH_LOCAL`.

One advantage of defining Sh data types with C++ templates is that Sh can use templated C++ structs. A typical case is that the outputs of a vertex shader are always the inputs of a fragment shader. If a struct is defined for this data set, the struct can be used in both shaders. The I/O parameter should be set to `SH_INPUT` in vertex shader and `SH_OUTPUT` in fragment shader. The following is an example that uses an I/O struct in Sh.

```
    template< int IO>
    struct COMMON_DATA {
```

```
    ShNormalf<3, IO>
    ShColorf<3, IO>
    ShVector<3, IO>;
};
```

Suppose these data types are the outputs in a vertex shader and the inputs in a fragment shader, then in the vertex shader, the output variables can be declared as `COMMON_DATA<SH_OUTPUT>outv`, whereas in the fragment shader, the input variables can be declared as `COMMON_DATA<SH_INPUT>inv`

Since the Sh data types are defined as template classes, users can add new classes as subclasses of the Sh classes. Templates are suggested in the definition of new classes, so that the new classes can still be used with structs.

But normally when we declare variables, we don't use the template formats. We use `typedef` to define convenient formats for data types. For example, instead of using `ShNormalf<3,SH_INPUT>` to declare a variable, we can use `ShInputNormal3f`.

5.9 Assembly Language

The Sh shading language is a high-level shading language, but it also supports the embedding of assembly-like instructions. Each assembly-like instruction in Sh corresponds to one instruction in the GPU. Sometimes we need to mix the high-level language with the low-level language, especially when we already have the old assembly code and we want to port it into Sh shaders directly. There are also times when we want to hand optimize the shaders.

For example, the expression $c = a*p+b$ will generate a parse tree which has one "$*$" node, one "+" node, and one "=" node because of the operator overloading. Three assembly instructions are generated for this expression. But if we use the multiply-add-copy instruction **ShMAC(c, a, p, b)**, only one assembly instruction is generated which does exactly the same thing.

Each Sh assembly-like instruction is a function call. In the Sh prototype, the number of parameters in the functions is at most 4. The function generates a parse tree that has a root node with four child nodes pointing to the parameters. The root node also records the type of the operation. An assembly token pointing to this root node is generated and pushed into the token list.

In the assembly-like function, Sh also checks to see if one temporary variable (currently marked as a constant register) has been modified by non-constant variables. If so, this variable will be marked as temporary variable.

5.10 Library

Sh has a library of support functions similar to those in Cg, the OpenGL shading language, and RenderMan. For example, since the "`?:`" operator in C++ cannot be overloaded, we provided a function `selection` that does the same thing. We also provide functions for evaluating noise, performing linear interpolation, evaluating trigometric functions, and transposing and inverting matrices.

Chapter 6

Applications

Other than the basic 2D and 3D texture data types, as a test of Sh, we also implemented new texture data types such as sparse textures and textures that support cubic interpolation. The purpose of this work is not just to add a few useful data types, but to experiment with abstraction mechanisms that can give users great benefits and convenience. For instance, when users declare a sparse texture, the texture will work the same way as normal textures, except it tries to do some data compression to save memory resources. When users declare cubic interpolation textures, the textures will check if the hardware supports cubic interpolation, if not, the cubic interpolation is done in software. We could also check if the hardware does linear interpolation and if so use a simpler cubic interpolation algorithm. There are also automatic conversions between different data types so that users can do different operations on the same data.

6.1 Sparse Texture

Sparse textures have a relatively small number of foreground pixels on a uniform background. To store a large sparse texture directly is not efficient. So if only the background color and the few foreground pixels are stored, we could save a fair amount of memory. To know the location of the foreground pixels, we also need an index to record the location of each foreground area.

The Sh sparse texture data type uses two textures internally as shown in Figure 6.1. One is for data, the other is for index to the foreground pixels. In the first texture, data are colours stored in blocks. The blocks start from the origin (0,0). The first block is for the background color. Other blocks are for colours in the foreground. In the index texture, each element is a pair of coordinates (for the 2D data texture).

The input sparse texture is first split into small square blocks. Each block corresponds to one element in the index array. The sparse texture is scanned block by block. If there is no foreground in the current block, the coordinates in the index texture is set to (0,0) indicating that the color of this block is the color of the background stored in the first block. If the current block has foreground pixels, the block should be put into the next block in the data texture. The coordinates for this block in the index texture are set to the offset of the block in the data texture.

In Figure 6.1, the input is a 16×16 sparse texture. The background is a white color. The foreground is a black patch. The gray grid is the texture pixel. The texture is split into 4×4 blocks as indicated by the black grid. Of these 16 blocks, there are 6 blocks that have foregrounds indicated by number 1 to 6. The data

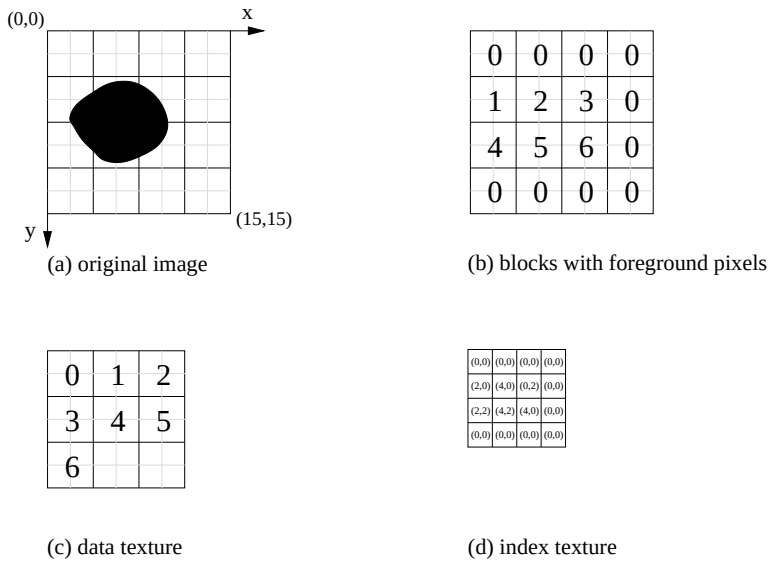

Figure 6.1: Sparse texture

texture has to have at least 7 blocks (one for the background color). The index texture is 4×4. The blocks that have foregrounds are stored in the data texture as shown in Figure 6.1 (c). The coordinates in the index texture are the offsets of the blocks in the data texture.

The texture lookup includes three steps. Given the texture coordinates, first compute which block the pixel falls in and the offset of this pixel in the block. Second, perform a texture lookup in the index texture to find the origin of the block in the data texture. Third, perform a texture lookup in the data texture using the origin and offset to find the color of the pixel.

In Figure 6.1, the original texture has 64 pixels. In the compressed format, we have 52 pixels. The compression ratio is 81.25%. However, this is a fairly trivial example. The compression ratio is determined by how many foregrounds there are

and by how the sparse texture is split.

Figure 6.2 shows how a more realistic sparse texture is compressed. The left image is the original image. The right image shows how the foreground is compressed. The red square at the top-left corner is the background color. The blue part of the image is used to pad the texture's dimensions to powers of two (because we used square textures which is the only supported texture type in Sm when the Sh prototype was implemented).

Since the square texture in GPU has to be powers of 2, the compression ratio is decreased by a certain amount. In the worst case, the texture may actually increase its memory requirement because of the extra index data. We can also pack the index into one texture with the data. Retangular textures can also be used, which will get rid of the powers of 2 requirement. On current GPUs, retangular textures have to be used for floating point data anyway. Such textures do not support either filtering or bilinear interpolation. This does not matter, however, because we have to compute bilinear interpolation and filtering *after* the sparse lookup. This idea could be extended to variable resolution textures and vector quantization.

6.2 Cubic Interpolation

Normally GPU only supports nearest-neighbor and linear-interpolation texture lookup. We developed a texture map type in Sh for cubic interpolation.

Given a texture map, we hope to find a cubic B-Spline surface that each pixel on the texture falls exactly on the surface, and other points on the surface between the pixels are the B-Spline cubic interpolation of the adjacent pixels. Once the

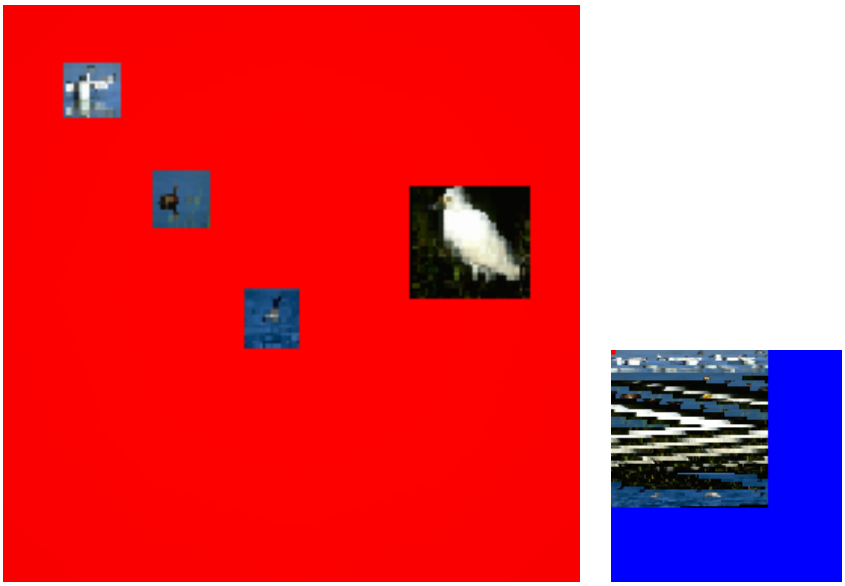

Figure 6.2: The compression of sparse texture
The left image is the original sparse texture. The right image shows how the foreground data are stored.

surface is found, we can get the cubic-interpolated color for any given coordinates in the texture.

The Sh cubic-interpolated texture data type was based on this idea. A spline surface can be represented relative to either the Cardinal spline bases or the uniform B-Spline basis. For the texture that is going to be interpolated, the original pixel colors are the Cardinal spline coefficients. We can easily get the B-Spline coefficients from the Cardinal spline coefficients. Then we store the B-Spline coefficients instead of the Cardinal spline coefficients. We do this, because over all C^2 cubic spline basis, the uniform B-spline basis has minimal support. This minimizes the number of data points we have to access to perform an interpolation.

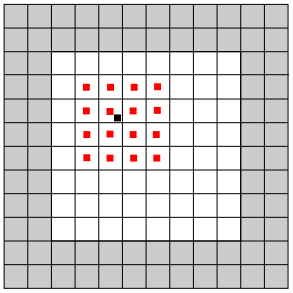

Figure 6.3: Texture with reflection borders

To get the cubic-interpolated color at an arbitrary position in the texture, we need the adjacent 16 B-Spline coefficients as is shown in Figure 6.3. For the points along the edges, a reflective border condition is used. To speed up the computation, we store the border pixels together with the texture in a bigger texture, so we avoid computing the border pixels.

Figure 6.3 is the texture that is actually stored in GPU. The inner square

represents the original B-spline data, and the outer square represents the data with reflective borders.

Figure 6.4 and Figure 6.5 are the results of bi-cubic interpolation texture lookup. Small images are mapped onto a bigger plane. The left column is the result of nearest-neighbor texture lookup. The middle column is the result of bilinear interpolation texture lookup. The right column is the result of bi-cubic interpolation texture lookup.

From the results we see that when the images get smaller and smaller (8×8 and 4×4), bilinear interpolation starts to exhibit obvious artifacts. In Figure 6.4, when the texture map is 8×8, the bilinear interpolation has artifacts and the shape of the flower is not kept as well as the one using bi-cubic interpolation. In Figure 6.5, when the texture map is 8×8, the red circle starts to change to a square when using bilinear interpolation, whereas the red circle is quite well preserved when using cubic interpolation.

Cubic interpolation can be used in many applications. It can be used for image texture compression. In image pyramids [8], when smaller images are used to approximate bigger images, cubic interpolation can get lower approximation error for the same amount of data. Cubic interpolation can be used in BRDF data compression (as in Figure 6.5) as well. Since the compressed BRDF data can be well approximated with smaller errors by cubic interpolation, we can increase the compression ratio. The disadvantage with cubic interpolation based on the B-spline basis is that the data needs to be preprocessed. If the original pixels are used as B-spline coefficients, the image will be over blurred. Conversion from the original

pixels to the necessary B-spline coefficients is equivalent to a sharpening of the input image. This can be accomplished with a pair of causal and anti-causal recursive filters [44].

6.3 Conversions Between Texture Data Types

The new texture types can convert to each other automatically. The conversion happens at the declaration of new texture variables. When the user declares a new texture variable and initializes it with an old texture variable, the new texture variable will copy texture data from the old texture variable and store these data in current texture format. So the users can use the features of the new textures. Note that this is where we do the preprocessing to convert from pixel values to B-spline coefficients, for instance. All new texture types also support the [ ] texture lookup operator so they can be used interchangeably in shaders.

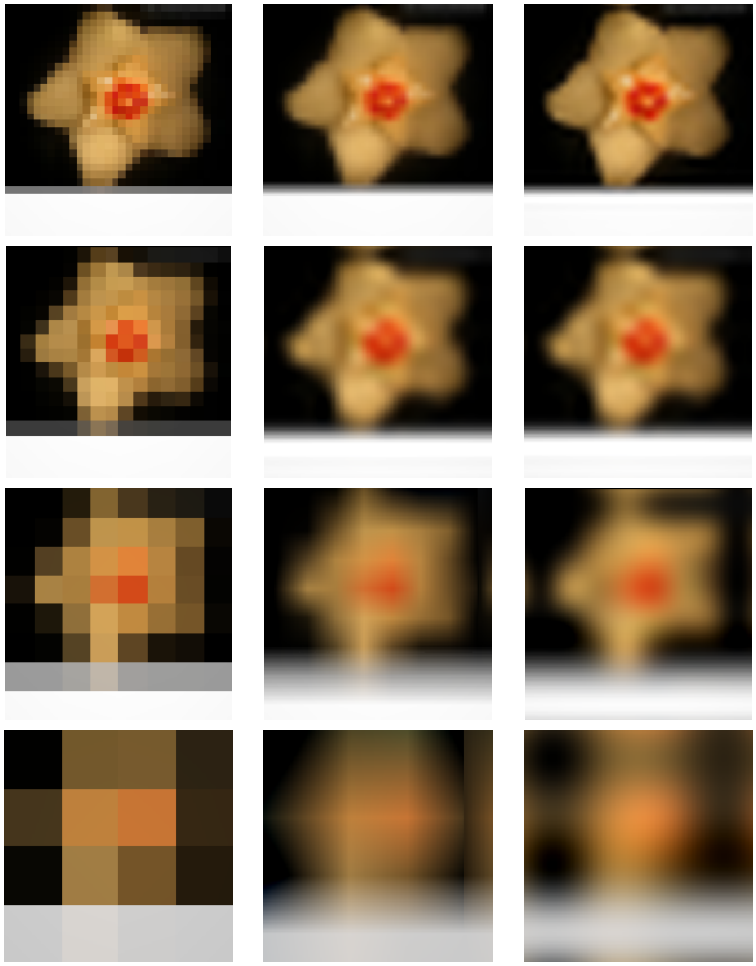

Figure 6.4: Results of cubic interpolation of hoya flower. Images of 32×32 (first row), 16×16 (second row), 8×8 (third row) and 4×4 (forth row) are mapped onto a plane. The left column is the result of nearest-neighbor texture lookup. The middle column is the result of bilinear interpolation texture lookup. The right column is the result of bi-cubic interpolation texture lookup.

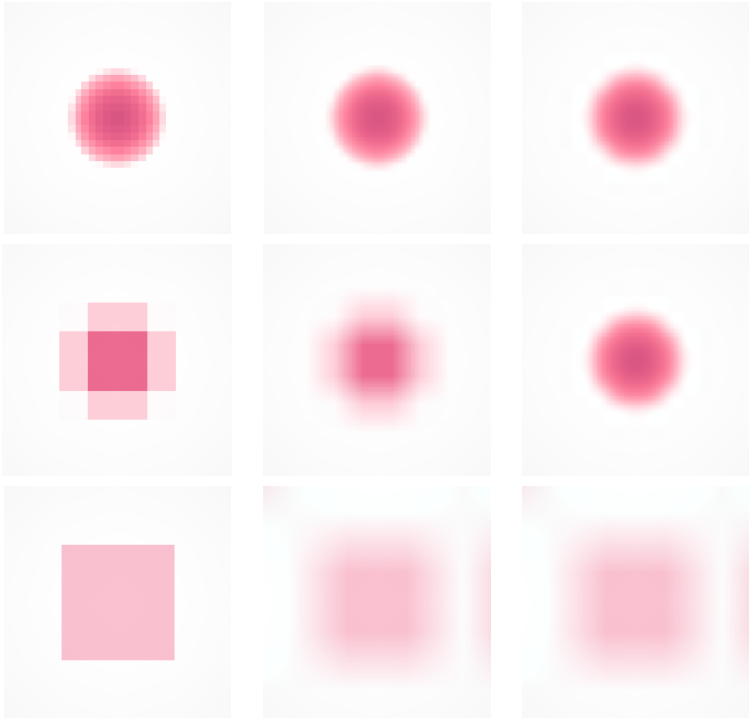

Figure 6.5: Results of cubic interpolation of BRDF data. Images of 32×32 (first row), 8×8 (second row) and 4×4 (third row) are mapped onto a plane. The left column is the result of nearest-neighbor texture lookup. The middle column is the result of bilinear interpolation texture lookup. The right column is the result of bi-cubic interpolation texture lookup.

Chapter 7

Results

This chapter presents some more example shaders that demonstrate some useful aspects of the Sh shading language. The examples are:

- Modified Phong Lighting Model
- Separable BRDFs and Material Mapping
- Parameterized Noise
- Julia Set

The first example is a simple shader implementing the Blinn-Phong lighting model. The second example shows an alternative method for building lighting methods, homomorphic factorization, but also combines several materials using material mapping. The third example demonstrates the use of a noise function to implement wood and marble shaders. Noise can be either provided by the underlying shading system or implemented by the compiler using precomputed textures,

without change to the high-level shader (although implementing noise using textures will, of course, use up texture units). The fourth example demonstrates a complex computation using a loop: the Julia set fractal.

7.1 Modified Phong Lighting Model

This example implements the modified Blinn-Phong lighting Model. An example image is shown in Figure 7.1. The vertex shader is shown in Figure 7.2 and the fragment shader is shown in Figure 7.3.

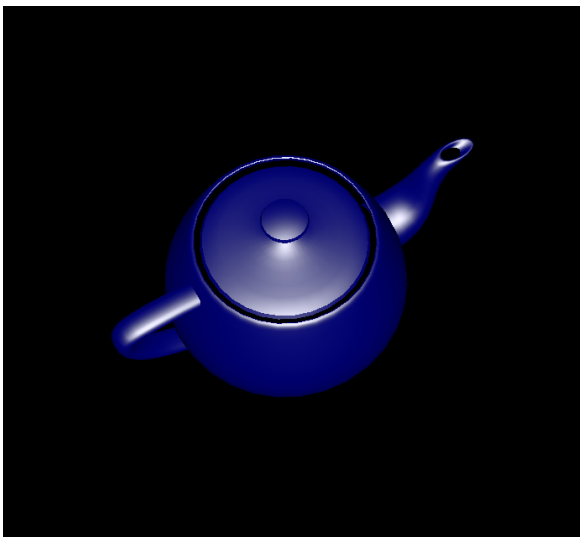

Figure 7.1: Sh example of Blinn-Phong shading

```
ShShader phong0 = SH_BEGIN_SHADER(0) {
    // declare input vertex parameters
    // unpacked in order given
    ShInputTexCoord2d ui;            // texture coords
    ShInputNormal3f nm;              // normal vector (MCS)
    ShInputPoint3f pm;               // position (MCS)

    // declare outputs vertex parameters
    // packed in order given
    ShOutputVector3f hv;             // half-vector (VCS)
    ShOutputTexCoord2f uo(ui);       // texture coords
    ShOutputNormal3f nv;             // normal (VCS)
    ShOutputColor3f ec;              // irradiance
    ShOutputPoint4f pd;              // position (HDCS)

    // compute VCS position
    ShPoint3f pv = modelview | pm;
    // compute DCS position
    pd = perspective | pv;
    // compute normalized VCS normal
    nv = normalize(nm | adjoint(modelview));
    // compute normalized VCS light vector
    ShVector3f lvv = light_position - pv;
    ShAttrib1f rsq = 1.0 / (lvv | lvv);
    lvv *= sqrt(rsq);
    // compute irradiance
    ShAttrib1f ct = max(0, (nv|lvv));
    ec = light_color * rsq * ct;
    // compute normalized VCS view vector
    ShVector3f vvv = -normalize(pv);
    // compute normalized VCS half vector
    hv = normalize(lvv + vvv);
} SH_END_SHADER;
```

Figure 7.2: Sh vertex shader for the Blinn-Phong lighting model.

```
ShShader phong1 = SH_BEGIN_SHADER(1) {
    // declare input fragment parameters
    // unpacked in order given
    ShInputVector3f hv;              // half-vector(VCS)
    ShInputTexCoord2f u;             // texture coordinates
    ShInputNormal3f nv;              // normal (VCS)
    ShInputColor3f ec;               // irradiance
    ShInputAttrib1f pdz;              // fragment depth (DCS)
    ShInputAttrib2f pdxy;             // fragment 2D position (DCS)

    // declare output fragment parameters
    // packed in order given
    ShOutputColor3f fc;              // fragment color
    ShOutputAttrib1f fpdz(pdz);       // fragment depth
    ShOutputAttrib2f fpdxy(pdxy);     // fragment 2D position

    // compute Blinn-Phong lighting model
    fc = phong_cd[u] + phong_cs[u]
       * pow((normalize(hv)|normalize(nv), phong_exp);
    // multiply by irradiance
    fc *= ec;
} SH_END_SHADER;
```

Figure 7.3: Sh fragment shader for the Blinn-Phong lighting model.

7.2 Separable BRDFs and Material Mapping

Separable BRDFs [30] approximate BRDFs by factorization. In particular, a numerical technique called homomorphic factorization is used to find a separable approximation to any shift-invariant BRDF. BRDFs are factorized into terms dependent directly on incoming light direction, outgoing view direction and half vector direction, all expressed relative to the local surface frame. To model the dependence of the reflectance on surface position, we can sum over several BRDFs, using a texture map to modulate each BRDF. This is called material mapping.

An example image is shown in Figure 7.4. The vertex shader is shown in Figure 7.5 and the fragment shader is shown in Figure 7.6.

Figure 7.4: Sh example of separable BRDF

```
ShShader hf0 = SH_BEGIN_SHADER(0) {
    // declare input vertex parameters, unpacked in order given
    ShInputTexCoord2f ui;            // texture coords
    ShInputVector3f t1;              // primary tangent
    ShInputVector3f t2;              // secondary tangent
    ShInputPoint3f pm;               // position (MCS)
    // declare output vertex parameters, packed in order given
    ShOutputVector3f vvs;            // view-vector (SCS)
    ShOutputVector3f hvs;            // half-vector (SCS)
    ShOutputVector3f lvs;            // light-vector (SCS)
    ShOutputTexCoord2f uo(ui);       // texture coords
    ShOutputColor3f ec;              // irradiatestingnce
    ShOutputPoint4f pd;              // position (HDCS)
    // compute VCS position
    ShPoint3f pv = modelview | pm;
    // compute DCS position
    pd = perspective | pv;
    // transform and normalize tangents
    t1 = normalize(modelview | t1);
    t2 = normalize(modelview | t2);
    // compute normal via a cross product
    ShNormal3f nv = normalize(t1 & t2);
    // compute normalized VCS light vector
    ShVector3f lvv = light_position - pv;
    ShAttrib1f rsq = 1.0 / (lvv|lvv);
    lvv *= sqrt(rsq);
    // compute irradiance
    ShAttrib1f ct = max(0, nv|lvv);
    ec = light_color * rsq * ct;
    // compute normalized VCS view vector
    ShVector3f vvv = -normalize(pv);
    // compute normalized VCS half vector
    ShVector3f hv = normalize(lvv + vvv);
    // project BRDF parameters onto SCS
    vvs = ShVector3f((vvv|t1), (vvv|t2), (vvv|nv));
    hvs = ShVector3f((hvv|t1), (hvv|t2), (hvv|nv));
    lvs = ShVector3f((lvv|t1), (lvv|t2), (lvv|nv));
} SH_END_SHADER;
```

Figure 7.5: Sh vertex shader for homomorphic factorization.

```
ShTexCoord3f parabolic (ShVector3f v) {
    ShTexCoord3f u;
    ShAttrib3f nv = normalize(v);
    u(0) = (7.0/8.0) * nv(0) + nv(2) + 1;
    u(1) = (7.0/8.0) * nv(1) + nv(2) + 1;
    u(2) = 2.0 * (nv(2) + 1);
    return u;
}
ShShader hf1 = SH_BEGIN_SHADER(1) {
    // declare input fragment parameters, unpacked in order given
    ShInputVector3f vv;              // view-vector (SCS)
    ShInputVector3f hv;              // half-vector (SCS)
    ShInputVector3f lv;              // light-vector (SCS)
    ShInputTexCoord2f u;             // texture coordinates
    ShInputColor3f ec;               // irradiance
    ShInputAttrib1f pdz;             // fragment depth (DCS)
    ShInputAttrib2f pdxy;            // fragment position (DCS)

    // declare output fragment parameters, packed in order given
    ShOutputColor3f fc;              // fragment color
    ShOutputAttrib1f fpdz(pdz);       // fragment depth
    ShOutputAttrib2f fpdxy(pdxy);     // fragment position

    // initialize total reflectance
    fc = ShColor3f(0.0, 0.0, 0.0);
    // sum up contribution from each material
    for(int m = 0; m < M; m++) {
        // sum up weighted reflectance
        fc += hf_mat[m][u] * hf_alpha[m]
            * hf_p[m][parabolic(vv)] * hf_p[m][parabolic(lv)]
            * hf_q[m][parabolic(hv)];
    }
    // multiply by irradiance
    fc *= ec;
} SH_END_SHADER;
```

Figure 7.6: Sh fragment shader for homomorphic factorization.

7.3 Parameterized Noise

In this example, the simple parameterized noise model proposed by John C. Hart *et al.* [16] is implemented to simulate wood. It can also be used to simulate marble and similar materials.

An example image is shown in Figure 7.7. The vertex shader is shown in Figure 7.8 and the fragment shader is shown in Figure 7.9.

Figure 7.7: Sh example of wood

```
ShShader pnm0 = SH_BEGIN_SHADER(0) {
    // declare input vertex parameters, unpacked in order given
    ShInputNormal3f nm;              // normal vector (MCS)
    ShInputPoint3f pm;               // position (MCS)

    // declare output vertex parameters, packed in order given
    ShOutputPoint4f ax;              // coeffs x MCS position
    ShOutputPoint4f x;               // position (MCS)
    ShOutputVector3f hv;             // half-vector (VCS)
    ShOutputNormal3f nv;             // normal (VCS)
    ShOutputColor3f ec;              // irradiance
    ShOutputPoint4f pd;              // position (HDCS)

    // transform position
    ShPoint3f pv = modelview | pm;
    pd = perspective | pv;
    // transform normal
    nv = normalize(nm | adjoint(modelview));
    // compute normalized VCS light vector
    ShVector3f lvv = light_position - pv;
    ShAttrib1f rsq = 1.0/(lvv|lvv);
    lvv *= sqrt(rsq);
    // compute irradiance
    ShAttrib1f ct = max(0,(nv|lvv));
    ec = light_color * rsq * ct;
    // compute normalized VCS view vector
    ShVector3f vvv = -normalize(pv);
    // compute normalized VCS half vector
    hv = norma(lvv + vvv);
    // projectively normalized position
    x = projnorm(pm);
    // compute half of quadric
    ax = pnm_quadric | x;
} SH_END_SHADER;
```

Figure 7.8: Sh vertex shader for wood.

```
ShShader pnm1 = SH_BEGIN_SHADER(1) {
    // declare input fragment parameters, unpacked in order given
    ShInputPoint4f ax;               // coeffs x MCS position
    ShInputPoint4f x;                // position (MCS)
    ShInputVector3f hv;              // half-vector (VCS)
    ShInputVector3f nv;              // normal (VCS)
    ShInputColor3f ec;               // irradiance
    ShInputAttrib1f pdz;             // fragment depth (DCS)
    ShInputAttrib2f pdxy;            // fragment 2D position (DCS)

    // declare output fragment parameters, packed in order given
    ShOutputColor3f fc;              // fragment color
    ShOutputAttrib1f fpdz(pdz);      // fragment depth
    ShOutputAttrib2f fpdxy(pdxy);    // fragment 2D position

    // compute texture coordinates
    ShTexCoord1f u = (x|ax) + turbulence(pnm_alpha, x);
    // compute Blinn-Phong lighting model
    fc = phong_cd[u] + phong_cs[u]
       * pow((normalize(hv)|normalize(nv)), phong_exp);
    // multiply by irradiance
    fc *= ec;
} SH_END_SHADER;
```

Figure 7.9: Sh fragment shader for wood.

7.4 Julia Set

This example implements the Julia set fractal. The example demonstrates the use of a conditional loop. This is not really a practical example, it is just meant to show the syntax for the definition of loops.

An example image is shown in Figure 7.10. The vertex shader is shown in Figure 7.11 and the fragment shader is shown in Figure 7.12.

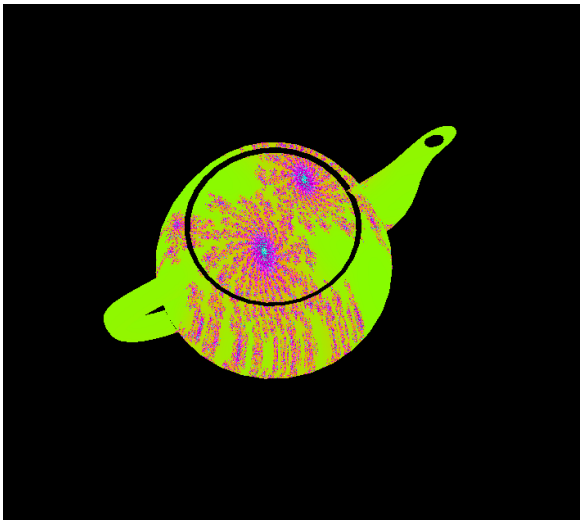

Figure 7.10: Sh example of Julia set

```
ShShader julia0 = SH_BEGIN_SHADER {
    // declare input vertex parameters, unpacked in order given
    ShInputTexCoord2f ui;            // texture coords
    ShInputPoint3f pm;               // position (MCS)

    // declare outputs vertex parameters, packed in order given
    ShOutputTexCoord2f uo(ui);       // texture coords
    ShOutputPoint4f pd;              // position (HDCS)

    // compute DCS position
    pd = (perspective | modelview) | pm;
} SH_END_SHADER;
```

Figure 7.11: Sh vertex shader for Julia set.

```
ShShader julia1 = SH_BEGIN_SHADER(1) {
    // declare input fragment parameters, unpacked in order given
    ShInputTexCoord2f u;              // texture coordinates
    ShInputAttrib1f pdz;              // fragment depth (DCS)
    ShInputAttrib2f pdxy;             // fragment 2D position (DCS)

    // declare output fragment parameters, packed in order given
    ShOutputColor3f fc;               // fragment color
    ShOutputAttrib1f fpdz(pdz);       // fragment depth
    ShOutputAttrib2f fpdxy(pdxy);     // fragment 2D position

    ShAttrib1f i = 0;
    ShAttrib2f v = u;
    SH_WHILE((v|v) < 2.0 && i < julia_max_iter) {
        v(0) = u(0) * u(0) - u(1) * u(1) + julia_c(0);
        v(1) = 2 * u(0) * u(1) + julia_c(1);
        u = v;
        i++;
    } SH_ENDWHILE;

    // send increment through lookup table
    fc = julia_map[julia_scale * i];
} SH_END_SHADER;
```

Figure 7.12: Sh fragment shader for Julia set.

Chapter 8

Conclusion

In this thesis, the Sh shading language prototype was described. This language is an embedded shading language targeting GPUs. It is built on top of C++. The compiler was constructed using C++ operator overloading for expressions, and a recursive descent parser for control constructs. C++ classes and templates are used to construct data types. Smart pointers are used for garbage collection.

8.1 The Sh Pros and Cons

Compared with other shading languages, the main differences with Sh are how it is implemented and how it is bound to the main application. Conventional shading languages are built just like normal programming languages. The development of those languages has to start from the scratch. Since our language is built on top of C++, its development avoided lots of the work that has to be done in conventional compilers. For example, the parsing of the expressions in Sh is done by

C++ automatically, the parsing of control constructs is simply done by a recursive descent parser, and the definition of data types is implemented by C++ classes and templates. Although the construction of our compiler is simple, we get a relatively complete language.

The reason we say our language is relatively complete is due to several aspects. First, we have relatively complete data types. The Sh data types include `Vector`, `Normal`, `Point`, `Texture`, `Matrix`, etc., while in other shading languages, for instance Cg, the data types are simply groups of floats, such as `float3`, `float4`. We distinguish different data types, because we can add special features, or restrictions to different data types. The Sh language also supports arrays. The arrays are declared just like in C++. Since the pointer is not supported in Sh, arrays must be defined using array syntax. C++ also gives the users the ability to define new types as needed.

The Sh language has a complete set of operators. The operators include arithmetic operators, increment and decrement operators, compound assignment operators, rational operators, logic operators, type casting operators, linear algebra operators, texture lookup operators, swizzle and writemask operators. These operators cover all the C++ operators and extend them with specific graphics operators.

The control constructs in Sh are complete. Compared with C++, semantically, the Sh language has every control construct that C++ has, although some of them cannot be used for now, such as `SWITCH... CASE...` because of the limitations of current GPU.

Although Sh does not have a subprogram capability, it can use the modularity

constructs of C++ to better organize the shaders. In Chapter 5, we have already introduced how C++ functions, classes and templates are used to organize the Sh code. Some of the other shading languages also support functions. But only Sh can use `class` and `template` because it takes advantage of C++. Note that the fact that Sh only inlines functions is not a disadvantage relative to other existing shading language implementation. All other shading languages also implement functions by inlining.

The Sh language tries to provide a complete library to users. We can never say that our library is complete, but the Sh developers will do their best to satisfy users. However, since Sh is based on C++, users can define their own libraries and use C++ linking tools to provide separate compilation. Separate compilation is not supported in many other shading languages.

There are some advantages of Sh over other shading languages. First, the Sh language is easy to extend. The Sh data types are C++ classes, and the Sh operators are operators defined for these data types, so users can define their own data types and operators easily. Second, Sh can use C++ features directly, the C++ modularity constructs are used in Sh, and the C++ control constructs are lifted into Sh. Third, binding of Sh shaders with application programs is extremely easy. The Sh language is in fact a C++ library, so the binding of Sh shaders with application programs only involves the matching of input and output data of shaders. The shaders can be put anywhere, either in separate files or mixed with other application programs. In other shading languages, the binding of shaders with application programs is a nuisance, and the shaders have to be put in separate files, because the

shading languages are completely different languages from the host languages, and the compilation has to be done separately, normally containing two steps, whereas the compilation of Sh shaders is a one-step process using C++ compiler. Fourth, the Sh language is similar to C++, so it is convenient for C/C++ users to learn without putting in too much effort. Finally, with the future development of C++, Sh shaders can always use new C++ features.

On the other hand, since Sh is embedded in C++, the restrictions in C++ pose some challenges to the Sh language. Syntactically, the Sh language cannot follow the C++ rules accurately. A simple example is the Sh control constructs. Since we cannot overload "`;`", the arguments in Sh `FOR` loop are separated by "`,`" instead of "`;`". In error checking, some errors that should be checked at compile time are delayed to run time. An example for this is the `SH_BREAK` and `SH_CONTINUE`. The illegal use of them are only checked at run time, because the parsing of Sh control constructs is in fact performed at run time. Although we did lift the checking of matching control construct keywords to compile time by using pairs of "{" and "}", the error messages we get from the C++ compiler are not precisely what we want.

Debugging of shaders is also a problem. GPUs don't provide APIs for accessing intermediate values in registers, so it is hard to provide a convenient debugging tool as in other languages. This is also a problem with other GPU shading languages though. In fact, Sh can be used in immediate mode to simulate the computation on the GPU and can be debugged using normal C++ tools.

8.2 Improvement and Future Work

Despite all the problems we have met in the development of Sh, the work described in this thesis demonstrates that building an embedded shading language on top of C++ is not only possible but also efficient. This language prototype has all the basic functionality a language needs, and its expressiveness has been tested by several application shaders. Yet improvements to our language are still necessary. Since the prototype was implemented, a new version of Sh was built with an improved parser and optimizer.

Through the optimizations performed in this thesis, we found that a better data structure was needed to represent the internal data so that optimizations can be powerful and efficient. The static single assignment (SSA) form and control dependency graph is now being used to represent data flow and control flow properties of programs. An efficient algorithm that computes these data structures for arbitrary control flow graphs is available [9]. In this algorithm, the intermediate code is put into SSA form, optimizations such as code motion, elimination of partial redundancies and constant propagation are performed on the SSA form representation, then the optimized SSA form is put back into intermediate code.

In the Sh prototype, the parse tree for each expression was relatively independent. Each expression has one assignment and one assignment token is pushed into the token list. This representation has been changed in the new version of Sh. Now, each operator generates a new value that is assigned to a temporary register, and this assignment token is put into the token list. So in the token list, all the expression tokens are simple assignments. This intermediate code is closer to SSA

form and therefore is easier to put into SSA form than before. This approach to parsing generates more data motion but the more powerful optimizer can clean this up. Also, we do not need to depend on a garbage collector to recover parse trees for expressions as they are never built.

Future work will focus on the following areas. Optimization needs to be extended to handle writemasking. SSA form has difficulty representing partial updates. A better debugging tool needs to be developed. Methods should be found to lift as much error checking as possible to C++ compile time. Real function calls should be added by overloading the "`( )`" operator when GPUs support subroutines. The Sh library needs to be extended by adding more useful functions, and more example shaders should be provided to users. Finally, it would be interesting to explore operators that recombine shaders into new shaders, providing new forms of code reuse and modularity.

Bibliography

[1] *The RenderMan Interface*. 2000. version 3.2.

[2] 3DLabs. *OpenGL 2.0 Shading Language White Paper*. December 2002. 1.1 edition.

[3] Inc. Ltd. 3Dlabs. *OpenGL 2.0 Shading Language White Paper*. 2002. available from http://www.3dlabs.com/support/developer/ogl2/whitepapers/OGL2-_Shading_Language_1.2.pdf.

[4] Inc. Ltd. 3Dlabs. *3Dlabs OpenGL 2.0 Specifications*. 2003. available from http://www.3dlabs.com/support/developer/ogl2/index.htm.

[5] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[6] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for motion pictures*. Morgan Kaufmann, 2002.

[7] ATI. *Pixel Shader Extension*. 2000. Specification document, available from http://www.ati.com/online/sdk.

[8] Patrick Brigger, Frank Muller, Klaus Illgner, and Michael Unser. Centered Pyramids. In *IEEE Trans. Image Processing*, Sept. 1999. vol. 8, NO. 9.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In *ACM Transactions on Programming Languages and Systems*, 1991. available from http://citeseer.nj.nec.com/cytron91efficiently.html.

[10] Scott Draves. Compiler Generation for Interactive Graphics Using Intermediate Code. In *Dagstuhl Seminar on Partial Evaluation*, pages 95–114, 1996.

[11] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, second edition, 1998.

[12] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling Embedded Languages. In *SAI/PLI*, pages 9–27, 2000.

[13] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proc. ACM SIGPLAN*, pages 160–170, 1996.

[14] B. Guenter, T. Knoblock, and E. Ruf. Specializing Shaders. In *Proc. ACM SIGGRAPH*, pages 343–350, August 1995.

[15] Pat Hanrahan and Jim Lawson. A Language for Shading and Lighting Calculations. In *Computer Graphics (SIGGRAPH'90 Proceedings)*, pages 289–298, August 1990.

[16] John C. Hart, Nate Carr, Masaki Kameya, Stephen A. Tibbitts, and Terrance J. Coleman. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 45–53. ACM Press, August 1999.

[17] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39 – 45, 1998.

[18] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Computer Graphics (SIGGRAPH'90 Proceedings)*, August 1999.

[19] Jack Herrington. *Code Generation in Action*. Manning, 2003.

[20] Jan Kautz and Michael D. McCool. Approximation of glossy reflection with prefiltered environment maps. In *Proc. Graphics (SIGGRAPH'99 Proceedings)*, pages 119–126, May 2000.

[21] Jan Kautz, Pere-Pau Vazquez, Wolfgang Heidrich, and Hans-Peter Seidel. Unified approach to prefiltered environment maps. In *Rendering Techniques (Proc. Eurographics Workshop on Rendering)*, 2000.

[22] B. W. Kernighan. A language for typesetting graphics. In *Software - Pract. and Exper. (GB)*, pages 12:1 – 21, January 1982.

[23] E. Lafortune and Y. Willems. *Using the modified Phong reflectance model for*

physical based rendering. 1994. Technical Report CW197. Comp. Sci., K.U. Leuven.

[24] Anselmo Lastra, Steven Molnar, Marc Olano, and Ylan Wan. Real-time programmable shading. In *1995 Symposium on Interactive 3D Graphics*, pages 59–66, April 1995. ACM SIGGRAPH.

[25] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. In *Design and Implementation*, pages 137–148, 1996.

[26] Jon Leech. *OpenGL extensions and restrictions for PixelFlow*. 1998. Department of Computer Science, University of North Carolina.

[27] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proc. SIGGRAPH*, pages 149 – 158, August 2001.

[28] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *ACM SIGGRAPH*, 2003.

[29] William R. Mark and Kekoa Proudfoot. Compiling to a VLIW fragment pipeline. In *Graphics Hardware 2001*, April 2001. SIGGRAPH/Eurographics.

[30] M. D. McCool, J. Ang, and A. Ahmad. Homomorphic Factorization of brdfs for high-performance rendering. In *Proc. SIGGRAPH*, pages 171 – 178, August 2001.

[31] Michael D. McCool. *A next-generation API for programmable graphics accelerators.* API Version 0.2 Presented at SIGGRAPH 2001 Course 25, Real-time Shading.

[32] Michael D. McCool, Chris Wales, and Kevin Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware*, 2001.

[33] Microsoft. *DirectX 9.* 2001. available from http://www.microsoft.com/-mmscorp/corpevents/meltdown2001/ppt/DXG9.ppt.

[34] Steven S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[35] M. Olano and A. Lastra. A shading language on graphics hardware: The pixelflow shading system. In *Proc. SIGGRAPH*, pages 159 – 168, July 1998.

[36] Marc Olano. A programmable pipeline for graphics hardware. In *PhD thesis, Univeristy of North Carolina at Chapel Hill*, 1999.

[37] Marc Olano, John C. Hart, Wolfgang Heidrich, and Michael D. McCool. *Real-Time Shading.* A K Peters, Ltd., 2002.

[38] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proc. SIGGRAPH*, pages 425 – 432, July 2000.

[39] Ken Perlin. An image synthesizer. In *Proc. SIGGRAPH*, pages 287 – 296, July 1985.

[40] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'c and tcc: A language and compiler for dynamic code generation. In *ACM Transactions on Programming Languages and Systems*, pages 21(2):324 – 369, 1999.

[41] K. Proudfoot, W. R. Mark, P. Hanrahan, and S. Tzvetkov. A real-time procedural shading system for programmable graphics hardware. In *Proc. SIGGRAPH*, August 2001.

[42] Kekoa Proudfoot. *Version 5 Real-Time Shading Language Description*. October 2000. available from http://graphics.stanford.edu/projects/shading/docs/lang5.txt.

[43] SGI. *Interactive Shading Language(ISL) Language Description*. March 2002. available from http://lust.u-strasbg.fr/Documentations/Visualisation/Shader/islspec.html.

[44] Michael Unser, A. Aldroubi, and M. Eden. B-spline signal processing. Part I: Theory. In *IEEE Trans. Signal Processing*, Feb 1993. vol. 41.

[45] Steve Upstill. *The RenderMan Companion: A programmer's guide to realistic computer graphics*. Addison-Wesley, 1999.

[46] Todd L. Veldhuizen. C++ template as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1999.

[47] T. Whitted and D. M. Weimer. A Software Testbed for the Development of 3D Raster Graphics System. In *ACM SIGGRAPH*, 1981.

[48] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley Publishing Company, 1995.