TARTU UNIVERSITY
FACULTY OF MATHEMATICS
Institute of Computer Science

Design and Applications of a Real-Time Shading System

Master's Thesis

Mark Tehver

Supervisor: Eero Vainikko

Author: ..... ". . . " May 2004
Supervisor: ..... ". . . " May 2004
Head of Chair: ..... ". . . " May 2004

Tartu 2004

Contents

1 Introduction **4**
- 1.1 Prolog . . . 4
- 1.2 Overview of the Thesis . . . 4

2 OpenGL Pipeline and Shading Extensions **6**
- 2.1 Overview of OpenGL . . . 6
- 2.2 OpenGL Pipeline . . . 6
- 2.3 Shading Extensions . . . 7
  - 2.3.1 Standard transformation and lighting model . . . 7
  - 2.3.2 Issues with the standard lighting model . . . 9
- 2.4 Vertex Programs . . . 9
  - 2.4.1 Parallelism . . . 10
- 2.5 ARB Vertex Program Extension . . . 11
  - 2.5.1 Registers . . . 11
  - 2.5.2 Texture coordinates . . . 12
  - 2.5.3 Instructions . . . 13
  - 2.5.4 Operand modifiers . . . 14
  - 2.5.5 Resource limits . . . 14
  - 2.5.6 Other vertex programming extensions . . . 15
- 2.6 Fragment Programs . . . 15
- 2.7 ARB Fragment Program Extension . . . 16
  - 2.7.1 Registers . . . 17
  - 2.7.2 Instructions . . . 17
  - 2.7.3 Dependent texturing . . . 18
  - 2.7.4 Operand modifiers, limits . . . 18
  - 2.7.5 Other fragment programming extensions . . . 18

3 High-Level Shading Languages **20**
- 3.1 Overview of Illumination Models . . . 20
  - 3.1.1 Bidirectional Reflectance Distribution Function . . . 21
- 3.2 Shading Languages . . . 22
  - 3.2.1 Shade trees . . . 23
  - 3.2.2 Pixel Stream Editor . . . 24
- 3.3 RenderMan . . . 24
  - 3.3.1 Types . . . 25
  - 3.3.2 Built-in functions . . . 26
  - 3.3.3 Execution environment . . . 26
- 3.4 Real-Time High-Level Shading Languages . . . 27
  - 3.4.1 Interactive Shading Language . . . 28
  - 3.4.2 SMASH and Sh . . . 29
  - 3.4.3 HLSL and Cg . . . 30

3.4.4 The OpenGL Shading Language . . . 31
3.4.5 Stanford Real-Time Shading Language . . . 31
3.4.6 ASHLI . . . 33

4 Experimental Shading System 34
4.1 Overview . . . 34
4.2 System Description . . . 34
4.3 Shading Language Description . . . 36
4.3.1 Basic type system . . . 36
4.3.2 Computation frequency . . . 37
4.3.3 Contexts . . . 38
4.3.4 Language constructs . . . 39
4.3.5 Primitive functions . . . 40
4.3.6 Function lookup scheme . . . 41
4.3.7 Built-in variables . . . 42
4.3.8 Swizzles . . . 43
4.3.9 Metaprogramming support . . . 43
4.3.10 Shader output . . . 45
4.4 Compiler Pipeline . . . 46
4.4.1 Preprocessor . . . 46
4.4.2 Parsing, type checking and inlining . . . 46
4.4.3 High-level optimizations . . . 47
4.4.4 Multiple pass generation . . . 48
4.4.5 Fragment-level code generation . . . 49
4.4.6 Vertex-level code generation . . . 50
4.4.7 Primitive group code generation . . . 50
4.5 Runtime . . . 50
4.5.1 API . . . 51
4.5.2 Shader states . . . 51
4.5.3 Hardware limits, custom configuration . . . 52
4.5.4 Shader caching . . . 52
4.6 Issues . . . 53
4.6.1 Multipass shaders . . . 53
4.6.2 Floating point model . . . 54
4.6.3 Extensions . . . 54

5 Performance and Applications of Shading System 56
5.1 Performance Compared to Fixed Function OpenGL Pipeline . . . 56
5.1.1 Implementation . . . 57
5.1.2 Surface shader . . . 57
5.1.3 Light shaders . . . 58
5.1.4 Testing procedure . . . 59
5.1.5 Performance . . . 60

5.1.6 Implementing lighting model in fragment level 61
5.2 Ray-Tracing a Simple Scene 64
5.2.1 Ray and sphere intersection using high-level shading language 64
5.3 Implementing High-Quality Scene Preview Modes for 3D Authoring Packages 70
5.3.1 XSI shader classes 70
5.3.2 Shader connection points 71
5.3.3 Example 72
5.3.4 Translating XSI shader nodes 74
5.4 Future Work 76
5.4.1 Advancements in hardware 76
5.4.2 Extending the shading system 77

6 Conclusions **79**

A Experimental Shading Language Syntax **81**

B Shading Language Prelude File **86**

C C Language API for Shading System **98**

1 Introduction

1.1 Prolog

Computer graphics hardware with 3D rendering support was introduced to personal computers in the middle of 1990s. From since, the advances have been so rapid that the current consumer level products already overshadow any specialized graphics hardware at any price point ever produced. The term Graphics Processing Unit (GPU) was introduced in the late 90s and the analogy with Central Processing Unit (CPU) can be justified – graphics processors have become flexible and autonomous, and can perform complex computations that were commonly reserved for CPUs alone. The purpose of 3D graphics processors is to accelerate commonly used graphics algorithms to provide real time frame rate. There is a large class of applications in the field of simulation, visualization and games for which real time frame rate is essential.

One of the goals for programmable graphics hardware is to provide different shading models for 3D objects. Shading models are used to customize surface color, texture and material properties. Conventional graphics pipeline (as defined by OpenGL or DirectX APIs) provides only a fixed shading model. The shading model implemented by conventional OpenGL is an empirical local illumination model. There are many other empirical illumination models with different features and they can be preferable in different cases. It is possible to bypass the shading model completely but this requires the application to calculate shading data for vertices which may require lots of computational resources.

1.2 Overview of the Thesis

The primary goal of this thesis is to discuss the design and implementation of a high-level real-time shading system working on modern graphics hardware. This includes an overview of the problems that are specific to graphics hardware interfaces and problems that exist in low-level shading interfaces. This goal is achieved by describing a design of a specfic high-level shading system implemented by the author. The secondary goal is to present possible applications of shading systems and discuss the performance of the implemented high-level shading system.

This thesis is divided into five chapters. After the current introduction, the second chapter gives a short overview of OpenGL graphics standard and discusses recently introduced vertex and fragment programming extensions. The third chapter gives an overview of high-level shading languages, focusing on interactive and real-time shading languages. The fourth chapter discusses design and implementation of a high-level real-time shading language and it unifies concepts introduced in previous chapters. This chapter is based on the original work of the author and presents the problems that must be solved when designing real-time shading systems.

An important part of that chapter is the discussion of the shading system features and optimizations that allow users to write simple, yet efficient applications and build complex shaders from simple shader nodes at runtime. Several of these features are currently unique to the described shading system but could be added to other shading systems also.

The last chapter provides examples of possible applications of the implemented shading system. It also gives information about performance of the system. Like the fourth chapter, this chapter is based on the author's work. Several test cases were implemented using both the described system and standard OpenGL. Thus, the chapter should provide insight about performance penalty when moving from a fixed function transform and lighting model to a programmable shader model.

2 OpenGL Pipeline and Shading Extensions

2.1 Overview of OpenGL

The OpenGL API (Application Programming Interface) is the most wide-spread standard for developing multi-platform, interactive 3D graphics applications. It provides a low-level software interface and can be used in the broadest application markets such as CAD, content creation, entertainment, game development and manufacturing. OpenGL has been designed using the client-server paradigm. This means the client applications and the graphics server can work on the same or separate machines. Network communication, if required, is handled transparently.

As a low-level API, OpenGL routines work on simple graphic primitives like points, lines and polygons. OpenGL provides also a middle-level interface for creating more complex, lighted and texture mapped surfaces. This gives software developers access to geometric and image primitives, display lists, modelling transformations, lighting and texturing, anti-aliasing, blending and many other features. OpenGL is mostly considered a procedural graphics interface as the scenes are rendered through commands describing how to render the scene objects. Still, some declarative features like material description and light objects are also provided for higher-level scene description.

Originally based on IRIS GL (a proprietary API from Silicon Graphics), OpenGL 1.0 was introduced in 1992. At the time of writing, the latest OpenGL version is 1.5 [19], which this thesis is based on. Although being a widely accepted standard, the OpenGL is still constantly evolving. Formal revisions are made at periodic intervals, and extensions are being developed that allow application developers to access the latest hardware features. The most important of these extensions are approved by OpenGL Architecture Review Board (short for ARB) and included in the core of new OpenGL versions.

2.2 OpenGL Pipeline

Figure 1 shows a schematic pipeline diagram of the OpenGL. The OpenGL architecture is structured as a state-based pipeline. Applications control the pipeline by entering commands from the left, while final rendering is stored in the frame buffer. Display lists are used to store a sequence of commands which can be later reused and issued together.

This thesis focuses on per-vertex operations, rasterization and per-fragment operations. These stages are discussed in the next sections. A detailed description of all stages is given in the OpenGL manual [19].

The vertex stage operates on geometric primitives described by vertices: points, line segments and polygons. In this stage vertices are transformed, lit and clipped against a viewing volume.

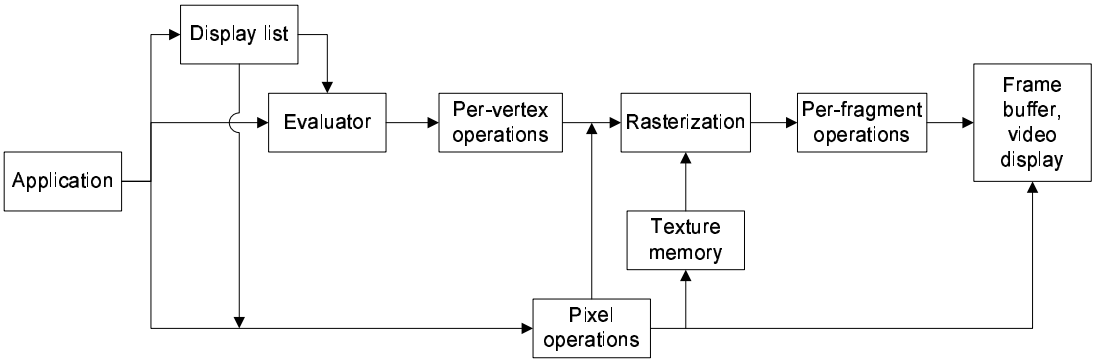


Figure 1: High-level view of the OpenGL pipeline.

The rasterization stage produces a series of fragments by sampling primitives at pixel centers. Fragments consist of frame buffer address and sampled attribute values (colors and texture coordinates).

Per-fragment operations use fragment data to calculate fragment color and possibly other data like depth information. After fragment color has been calculated, additional steps like depth buffering, blending, masking and other operations are performed to calculate final fragment color that is stored in frame buffer.

2.3 Shading Extensions

2.3.1 Standard transformation and lighting model

Conventional OpenGL provides fixed function transformation and lighting (T&L) pipeline that is often implemented in hardware. Its function is shown in figure 2 (note that the presented diagram is simplified, multiple texture units, edge attributes and vertex blending are missing).

The vertex and normal transformation stages apply a linear transformation to homogenous vertex coordinates, providing little flexibility and no support for more general transformations. Texture coordinate generation (texgen) units can compute texture coordinates based on vertex coordinates and normal vector. Only a few hardcoded functions are provided [19].

In fact, the standard transformation pipeline (with texgen) can be described with the following equations:

$$\mathbf{V}_o = v(\mathbf{V}_i) \tag{1}$$

$$\mathbf{N}_o = n(\mathbf{N}_i) \tag{2}$$

$$\mathbf{T}_o^k = t^k(\mathbf{V}_i, \mathbf{N}_i, \mathbf{T}_i^k) \tag{3}$$

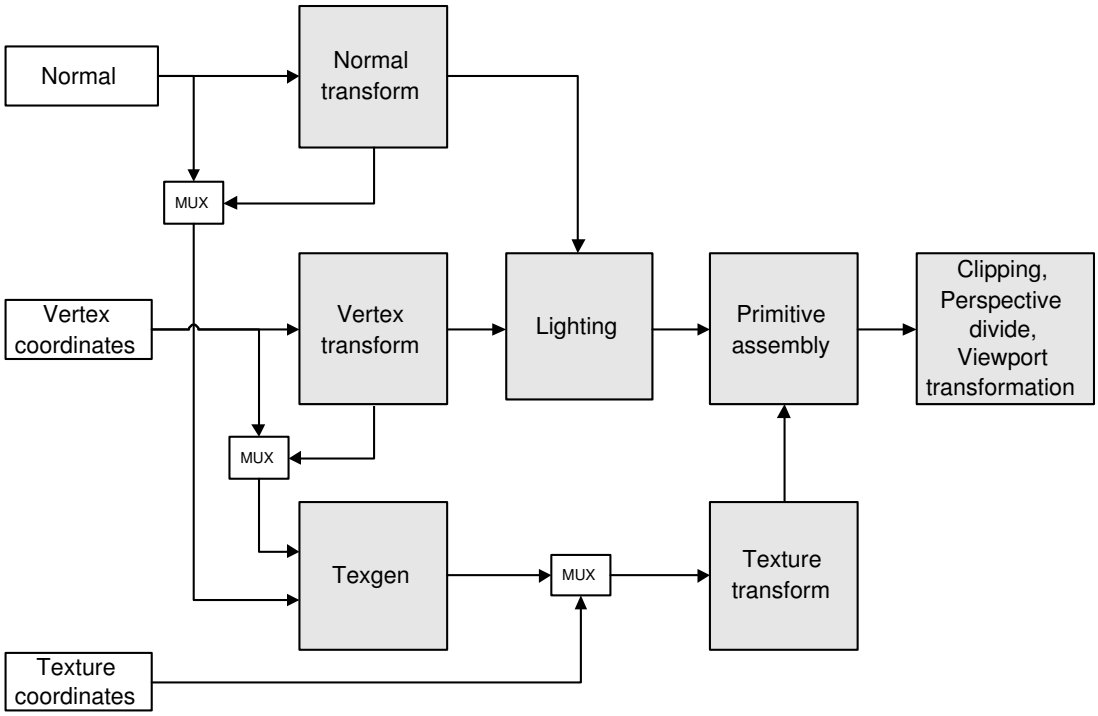


Figure 2: Standard OpenGL Transformation and Lighting pipeline.

where v, n are linear functions and t^k (k denotes texture unit number) can be chosen only from a small set of predefined functions (note: the functions also depend on current modelview matrix and texture matrices, but this state does not vary inside primitive groups). $\mathbf{V}_i$, $\mathbf{N}_i$ denote input vertex coordinates and normal, $\mathbf{T}_i^k$ texture unit k coordinates. $\mathbf{V}_o$, $\mathbf{N}_o$, $\mathbf{T}_o^k$ denote output values respectively, that are sent to the next pipeline stage (rasterization).

OpenGL uses a slightly generalized form of the following lighting model ($\otimes$ denotes componentwise multiplication):

$$\mathbf{i}_{tot} = \mathbf{m}_{emi} + \mathbf{a}_{glob} \otimes \mathbf{m}_{amb} + \sum_{k=1}^{n} c_{spot}^k d^k (\mathbf{i}_{amb}^k + \mathbf{i}_{diff}^k + \mathbf{i}_{spec}^k) \tag{4}$$

where

$\mathbf{i}_{tot}$ is the total calculated lighting intensity.
$\mathbf{a}_{glob}$ is the ambient intensity of scene.
$\mathbf{m}_{amb}$ and $\mathbf{m}_{emi}$ denote material ambient color and emissiveness, respectively.
d^k denotes distance attenuation factor for light k.
c_{spot}^k is used for spotlight k to limit the light application to a specified cone.
$\mathbf{i}_{amb}^k$, $\mathbf{i}_{diff}^k$ and $\mathbf{i}_{spec}^k$ specify light k ambient, diffuse and specular contributions.

The number of supported lights is limited, OpenGL specification requires support for at least eight lights, although implementations may support more.

2.3.2 Issues with the standard lighting model

Perhaps the most important problem with the OpenGL lighting model arises from the fact that all calculations are performed once per-vertex. With vertex level lighting, undersampling effects are common unless objects are highly tessellated. This is especially visible when specular lighting term is used ($\mathbf{i}_{spec}$ in the lighting formula) as the highlight may fall between vertices and may not be visible at all. Also, when a highly tessellated object is viewed from a distance, vertex level lighting calculations are still performed for each vertex. In this case, the object will be oversampled with no visual benefit. Dynamic Level-of-Detail techniques may help selecting the right number of vertices, but current hardware does not directly support this.

Another problem is that the OpenGL lighting model produces adequate visuals only for a limited class of surfaces. Plastic surfaces may look realistic enough, but good-looking metallic materials or human skin are not possible at all.

2.4 Vertex Programs

Vertex programs are powerful generalizations of the fixed function T&L functionality. Conceptually, vertex programs can implement any transformation for vertices, normals, texture coordinates and other per-vertex attributes. Vertex programs can be used to implement the functions of a conventional T&L pipeline, thus being a true superset of its capabilities.

Vertex programs implement the following model:

$$\mathbf{i}_{tot} = i(\mathbf{V}_i, \mathbf{N}_i, \mathbf{T}_i^1, \mathbf{T}_i^2, ..., \mathbf{T}_i^m) \tag{5}$$
$$\mathbf{V}_o = v(\mathbf{V}_i, \mathbf{N}_i, \mathbf{T}_i^1, \mathbf{T}_i^2, ..., \mathbf{T}_i^m) \tag{6}$$
$$\mathbf{N}_o = n(\mathbf{V}_i, \mathbf{N}_i, \mathbf{T}_i^1, \mathbf{T}_i^2, ..., \mathbf{T}_i^m) \tag{7}$$
$$\mathbf{T}_o^k = t^k(\mathbf{V}_i, \mathbf{N}_i, \mathbf{T}_i^1, \mathbf{T}_i^2, ..., \mathbf{T}_i^m) \tag{8}$$

Here i, v, n, t^k are not limited to linear functions. In case of vertex programs, there is no need to differentiate between vertex position, normal, texture coordinates. Thus, in general all parameters are referred to as *vertex attributes*. Although not shown explicitly in the equations, vertex programs may also use a set of environment parameters (called *program parameters*) that can be defined per primitive group (outside OpenGL `Begin` and `End` scope). Sometimes vertex parameters are referred to as *vertex constants* as their value never changes during the execution of a vertex program.

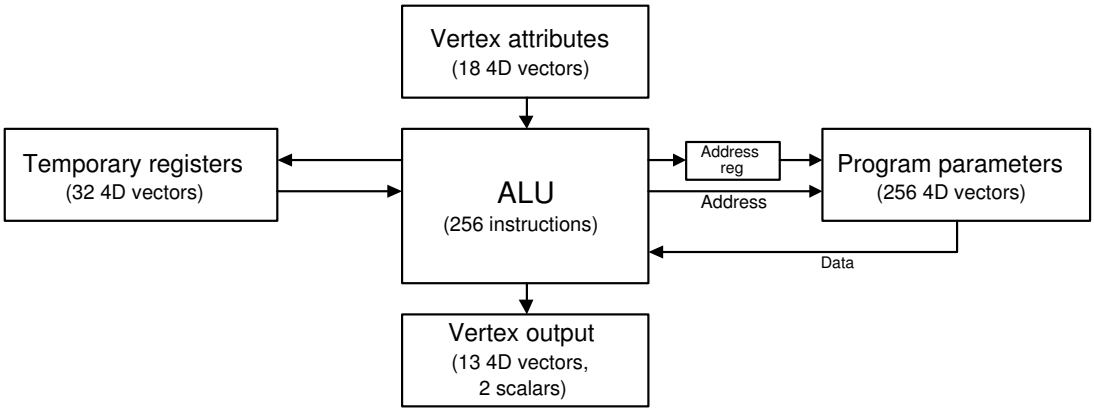


Figure 3: ARB vertex program extension architecture. Presented limits apply to Radeon 9500/9700 GPUs.

Note that vertex programs do not replace all stages of vertex processing. The frustum clipping, perspective divide, viewport scaling are performed as in the fixed-function pipeline.

2.4.1 Parallelism

Vertex programs process all vertices independently. Thus, information about the primitive that the vertex is part of, is not provided to the program. This is a fundamental limit of the approach, but improves parallelism and performance (as vertices may be shared between several primitives). When such information is required, it can still be passed to the vertex program using vertex attributes (all vertices of a primitive have same values for such attributes).

The fundamental data type used in vertex transformations is a single precision floating point type. Vertices and vertex attributes are usually encoded as 3- or 4-component vectors. Instructions on these components are typically performed in parallel, thus parallelism also exists on instruction level.

The real benefit of vertex programs is present only if they are executed on graphics hardware (Graphics Processing Units or GPUs), not on CPUs. Otherwise all their functionality can be easily implemented by general purpose processors and some complexity is simply lifted from an application to the API. In such case the CPU must transform each vertex and its attributes and then send the transformed vertices and attributes to GPU. For large vertex sets, this becomes computationally and bandwidth-wise very expensive.

Attribute number	OpenGL vertex parameter
0	vertex position
1	vertex weights 0-3
2	normal
3	primary color
4	secondary color
5	fog coordinate
8+n	texture coordinate set n

Table 1: Input registers (attribute registers) of vertex programs. User-supplied vertex attributes (like normal vector) can be accessed through input registers.

2.5 ARB Vertex Program Extension

The scheme of `ARB_vertex_program` extension [31] architecture is presented in figure 3. At first sight the architecture seems quite similar to a regular CPU architecture, but there are several important differences. The most important one is that this model is inherently more limited than general purpose CPU models – vertex programs have no access to main or local (graphics) memory, only a limited register file is available. Also general forms of branching are not available (although conditional execution is possible). Unlike CPU architectures, there is no native support for integer operations.

Other differences will be discussed in the following subsections. Note that we do not provide formal overview of the extension – only the most important concepts are discussed that are essential in understanding high-level shading language implementations. Also, we do not discuss the extension at the OpenGL API level at all. These details can be found in the extension specification [31].

2.5.1 Registers

There are several distinct register files: input file (contains vertex attributes), output file, temporary and constant register files (program parameters). All registers (except address register) contain four-component vectors, each component being a floating point number. Address register contains a single floating point scalar value.

Temporary register file is only used to store temporary results and is both readable and writable. The extension allows to define user-specified names for temporary registers, but for clarity this thesis uses only names `r0`, `r1`, ...

Input registers are aliased to conventional per-vertex parameters like vertex coordinates, normal vector and texture coordinates (shown in table 1). Input register file is read-only, registers are specified as `vertex.attrib[N]` (for current implementations, N is between 0 and 15).

Output register file contains the final, transformed vertex parameters and

Output register name	Components	Description
`result.position`	(x, y, z, w)	Clip-space coord. of vertex
`result.color.front.primary`	(r, g, b, a)	Front-facing primary color
`result.color.front.secondary`	(r, g, b, a)	Front-facing secondary color
`result.color.back.primary`	(r, g, b, a)	Back-facing primary color
`result.color.back.secondary`	(r, g, b, a)	Back-facing secondary color
`result.fogcoord`	$(f, *, *, *)$	Fog coordinate
`result.pointsize`	$(s, *, *, *)$	Point size
`result.texcoord[n]`	(s, t, r, q)	Texture coordinate for unit n

Table 2: Output registers of vertex programs. Note that fog coordinate and point size registers are vector registers, although only a single component of them is used.

is only writable. Transformed vertex parameters are sent to the rasterization stage. Output registers are specified in table 2. Vertex program must write to `result.position` register (otherwise rasterization would not be possible), while writing to the other output registers is optional.

Constant register file is read-only inside vertex program, but can be changed by application outside the `Begin` and `End` scope. Constant register file is intended to store parameters that do not depend on vertices. Extension divides constant register file into two classes, called *program environment parameters* and *program local parameters*. Program environment parameters (specified as `program.env[N]`, where N is between zero and implementation-specific constant) are associated with OpenGL context and are common to all programs of the same context. Program local parameters are associated with vertex programs (specified as `program.local[N]`, N is between zero and implementation-specific value).

2.5.2 Texture coordinates

In case of triangle rasterization, fragment texture coordinates and colors are interpolated using the following formula:

$$\mathbf{V}_f = \frac{(a_f/w_a)\mathbf{V}_a + (b_f/w_b)\mathbf{V}_b + (c_f/w_c)\mathbf{V}_c}{(a_f/w_a) + (b_f/w_b) + (c_f/w_c)} \tag{9}$$

Here w_a, w_b and w_c denote w-coordinates of three vertices, $\mathbf{V}_a$, $\mathbf{V}_b$, $\mathbf{V}_c$ denote the interpolated quantities (usually four-component vectors) for three vertices and a_f, b_f and c_f are barycentric coordinates for the fragment (by definition, $a_f + b_f + c_f = 1$). OpenGL allows implementations to simplify this formula by using an approximation:

$$\mathbf{V}_f = a_f\mathbf{V}_a + b_f\mathbf{V}_b + c_f\mathbf{V}_c \tag{10}$$

Instructions	Output	Inputs	Description
MAD	v	v,v,v	Multiply and add
ADD SUB MUL	v	v,v	Componentwise add, subtract, multiply
MIN MAX	v	v,v	Componentwise minimum, maximum
SGE SLT	v	v,v	Componentwise >= and < relations
DST LIT XPD	v	v,v	Distance vector, lighting terms, cross product
MOV ABS SWZ	v	v	Componentwise move, absolute value, swizzle
FLR FRC	v	v	Componentwise floor, fraction
DP3 DP4 DPH	s	v,v	Dot products for 3D, 4D, 2D vectors
EX2 LG2	s	s	Exponential and logarithm base 2
EXP LOG	s	v	Exponential and logarithm base 2 (approx.)
RCP RSQ	s	s	Reciprocal and reciprocal square root
POW	s	s,s	Exponentiate
ARL	a	v	Address register load

Table 3: Vertex program instruction set. In the inputs column letter 's' denotes scalar operand, 'v' vector operand. In the output column 's' denotes scalar output (result is replicated across all components), 'v' denotes vector output and 'a' denotes address register.

For texture coordinates, this produces very noticeable artifacts [9], and in practice this is only applicable to color interpolation. Fortunately, current implementations perform interpolation with true perspective correction even for colors.

2.5.3 Instructions

Vertex program instruction set is presented in table 3. The number of operations is small and seems quite limited. Branch instructions are missing, although simple conditional execution can be still achieved using a combination of `SGE` (or `SLT`), `MUL` and `MAD` (although this has some limitations that are discussed in the fourth chapter). Some instructions (`MUL`, `SUB`, `MOV`) resemble ordinary CPU instructions, while other instructions like `DP3`, `DP4` are not included in CPU instruction sets. In computer graphics 3x3 and 4x4 matrix multiplications are quite common and dot product instructions allow to perform these more efficiently. `DST` and `LIT` instructions are very specific to computer graphics – both encode common instruction sequences for lighting models. `DST` instruction is used in an instruction sequence for calculating vector $(1, d, d^2, 1/d)$ where d denotes vertex distance from a light source. `LIT` instruction is used to accelerate the computation of ambient, diffuse and specular lighting terms.

Few instructions like `EX2`, `LG2`, `RCP`, `RSQ` belong to the class of *scalar instructions*. These instructions are unary instructions and take a single scalar as an argument. As all registers contains vectors, scalar operations must specify which

source component of a vector has to be used. The results of scalar operations are replicated across all components. For example, the instruction

```
RCP r0,r1.x;
```

calculates $(1/\texttt{r1.x}, 1/\texttt{r1.x}, 1/\texttt{r1.x}, 1/\texttt{r1.x})$ and stores the result in `r0`.

2.5.4 Operand modifiers

In order to better support data rearranging in vectors, optional *swizzle* operations are provided as argument modifiers for all instructions. For example, the following instruction:

```
ADD r0, r1.xxzw, r2.yzxw;
```

calculates $(\texttt{r1.x} + \texttt{r2.y}, \texttt{r1.x} + \texttt{r2.z}, \texttt{r1.z} + \texttt{r2.x}, \texttt{r1.w} + \texttt{r2.w})$ and stores the result in register `r0`. Additionally, all source operands can be negated:

```
ADD r0, r1, -r2;
```

Note that although instruction set is small, several instructions are redundant in the sense that they can be expressed through other instructions. For example, `SUB` instruction is actually not required as it can be expressed using an `ADD` with the second operand negated. Likewise, `XPD` instruction can be emulated using a `MUL` and an `ADD` instruction.

The extension also provides support for partial register updates, this functionality is called *masking*. For example,

```
MOV r0.xy, r1;
```

only copies x- and y-components from register `r1` to `r0`, z- and w-components are left unchanged. When the mask is omitted, all components will be updated.

Although vertex programs do not support random memory access, limited support is still provided for small "lookup"tables. This is provided using a single *address register* – instructions can offset constant register file access using this register. `ARL` instruction is provided for setting this register.

2.5.5 Resource limits

The extension specifies queriable resource limits on register counts, parameter counts and program length. There are strict minimum requirements for each limit. Every implementation must accept programs which use 12 or less temporary registers, 96 program parameters and 128 instructions. In case when a program exceeds hardware resources, but can be emulated within software limits, the extension provides means to detect this. Mixing software and hardware implementations may produce different results. This is important as rasterization

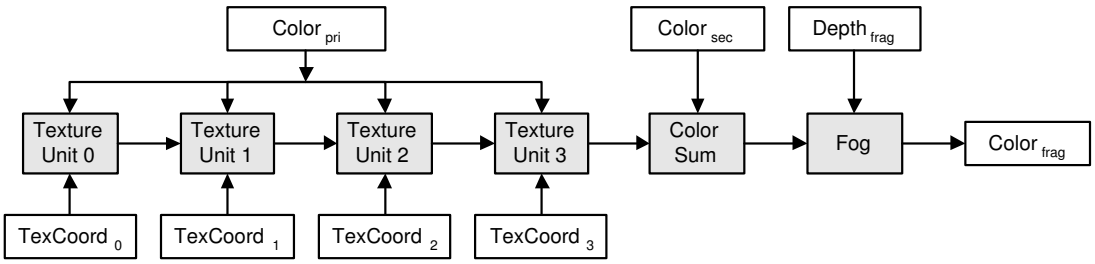


Figure 4: The OpenGL multitexturing pipeline. The number of texture units (here four) may vary with the implementations.

may generate different fragments depending on whether the vertex program was executed on CPU or GPU, which can produce artifacts in multipass renderings. Even very subtle differences in calculations are magnified by point sampling at polygon edges.

2.5.6 Other vertex programming extensions

Currently, there are two other extensions for OpenGL that define similar functionality to `ARB_vertex_program`: `NV_vertex_program` and `EXT_vertex_shader`. `NV_vertex_program` was the first vertex programming extension and closely follows the vertex unit design of GeForce 3 GPU [16]. For example, the limited number of constant register file ports are exposed to user; a single instruction may not refer to more than one constant. `EXT_vertex_shader` is more suitable for constructing shaders at runtime and provides larger set of instructions, some of which are quite complex. In general, both extensions (plus `ARB_vertex_program`) can be used to implement any other vertex programming extension and are all designed after DirectX 8 Vertex Shader 1.1 model.

DirectX 9 defines several new vertex shader models. Vertex Shader 2.0 model adds support for static looping – it provides branching instructions which may depend only on program parameters. Vertex shader specification 3.0 is more advanced, provides support for dynamic branching and instruction masking and supports texture operations for vertices. There is currently one vendor-specific OpenGL extension that provides support for dynamic branching and instruction masking – `NV_vertex_program2`. Though, it does not support vertex texturing.

2.6 Fragment Programs

Although all discussed vertex programming extensions are relatively similar, there is much more diversity in fragment programming models. The basic model that is supported by most graphics processors is the OpenGL *multitexturing pipeline* model with color sum and fog stages. It is presented in figure 4. Each multi-

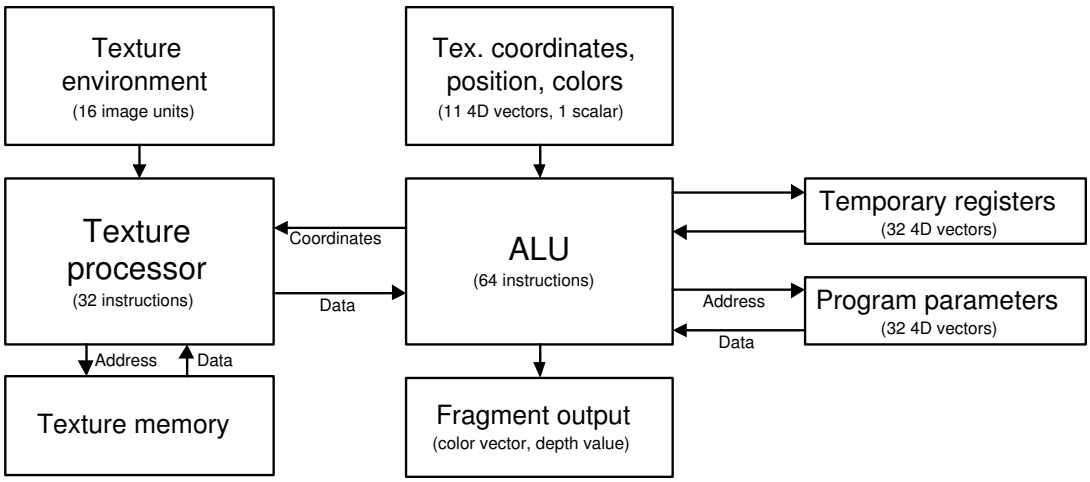


Figure 5: ARB fragment program extension architecture. Presented limits apply to Radeon 9500/9700 GPUs.

texturing stage of the pipeline (called *texture unit*) performs a simple arithmetic operation (like `ADD`, `MUL`, `DOT3`) taking operands from previous stages (if available), interpolated fragment color or texture color. The number of units is dependent on implementation, typically it is two, four or eight. This is a rather limited model and utilizes fixed point arithmetic using range [0..1], typically with 8-12 bit precision per color component.

Multitexturing stages are followed by color sum and fog stages. Color sum stage adds secondary color to the calculated fragment color from the last multitexturing stage. Fog stage, if enabled, blends the fog color with the calculated color depending on the fragment depth value.

Although the multitexturing model seems as limited as the fixed texture coordinate generation modes, in reality it can be generalized by using *multipass techniques*. Using multipass rendering, same primitive is rendered multiple times and blended with previously rendered fragments. This allows a large variety of effects [4] although it quickly becomes expensive as the number of passes grows. This approach conflicts with general trends in hardware design where storage/bandwidth improvements are considered expensive and computational resources relatively inexpensive [12].

2.7 ARB Fragment Program Extension

The ARB fragment program model is similar to the ARB vertex program model described earlier. The ARB vertex program allowed to define an algorithm for each vertex specified via OpenGL to generate custom transformations, vertex colors and texture coordinates, while ARB fragment program allows to use similar

model to generate custom fragment colors and overwrite fragment depth.

Like the ARB vertex program model, this model is inherently parallel – all fragments can be processed in parallel.

`ARB_fragment_program` extension [31] replaces texturing, color sum and fog stages in the OpenGL pipeline. The stages after fog, like pixel ownership test, scissor test and following stages are identical in both modes.

Like vertex programs are superset of conventional transform and lighting, fragment programs are true superset of conventional multitexturing. The most significant change besides the register file model, is the usage of floating point numbers as primary data types instead of fixed point types. Although fixed point has advantage of requiring less resources on hardware side, low dynamic range and need to track underflow and overflow conditions makes this hard to use in practice.

2.7.1 Registers

ARB fragment program register files are similar to vertex program register files, although vertex program output registers correspond directly to fragment program input registers. Fragment programs have similar temporary and constant register files, while output register file contains only two members: `result.color` (which is four-component vector) and `result.depth` (which is a scalar register). By writing into `result.depth`, fragment program can overwrite the default depth value generated by rasterization stage.

2.7.2 Instructions

`ARB_fragment_program` keeps the basic architecture close to the ARB vertex programming extension. All operations of vertex programs except `ARL`, `LOG` and `EXP` are provided in fragment programs also. Several new instructions are also included: `SIN`, `COS`, `SCS` (calculates `SIN` and `COS` in parallel), `LRP` (linear interpolation between two vectors) and `CMP` instruction for conditional execution. Additionally, the extension includes three texturing instructions and `KIL` instruction. Texturing operations take three input parameters: texture image object, sampling coordinates and texture sampler type (for example, 1D, 2D, 3D, cube map texture). For example, the following instruction

```
TEX r0, texture[1], fragment.texcoord[2], 2D;
```

reads texture unit 1 (which must be associated with a 2D texture) using texture coordinate set 2. The result is stored in register `r0`.

`KIL` instruction can be used to remove current fragment from further processing. It takes a single argument (four-component vector) and if any of its components are less than zero, then the fragment is discarded from further processing.

2.7.3 Dependent texturing

One feature available in fragment programs that is not available in multitexturing pipeline, is *dependent texturing*. Unlike multitexturing pipeline, texture object and texture coordinate sets are decoupled in `ARB_fragment_program` extension and texture instructions can refer to samplers and coordinate sets independently.

The ARB fragment programming extension places a limit on dependent texturing operations. When a texture instruction uses a result computed by earlier instruction as texture coordinate (thus, depending on the earlier instruction), *texture indirection* occurs. ARB fragment programming extension places a maximum limit on the number of possible texture indirections. Minimum requirement for an implementations is to support chains of up to depth four. Thus, some implementations may not support the following code sequence:

```
TEX r1, texture[0], fragment.texcoord[0], 2D;
TEX r2, texture[0], r1, 2D;
TEX r3, texture[0], r2, 2D;
TEX r4, texture[0], r3, 2D;
TEX r5, texture[0], r4, 2D;
```

as each instruction starting from second requires texture indirection – thus, the number of nodes in instruction dependency chain is five.

2.7.4 Operand modifiers, limits

Same operand modifiers that are available in vertex programs, are also available in the fragment programming extension. The ARB fragment program provides one *instruction modifier*. When instruction mnemonic has `_SAT` suffix, then instruction result is *saturated* before writing to the output register. Saturation simply clamps all result components to [0..1] range.

Similar to vertex programs, limits on maximum number of instructions, temporary registers and constants can be queried during runtime. Fragment programs introduce new limits which are related to texture operations – the total number of allowed texture instructions may be smaller than allowed ALU instructions and dependent texturing may be limited. Some limits for R300 graphics processor are shown in figure 5.

2.7.5 Other fragment programming extensions

The ARB fragment programming extension is derived from DirectX 9 Pixel Shader 2.0 specification [23]. The ARB extension provides more orthogonal support in terms of operand and output modifiers (DirectX restricts usage of swizzling and does not allow specifying output mask in texturing operations), but lacks *multiple render target* support. Multiple render target support means that pixel shader may write to additional color buffers in addition to the main

frame buffer. Support for this is provided by one vendor specific extension in OpenGL.

There are several extensions that provide support for similar features available in DirectX 8 Pixel Shader versions 1.1-1.4. `NV_register_combiners2` and `NV_texture_shader` extensions provide similar features as Pixel Shader version 1.1 [31]. A more powerful Pixel Shader 1.4 like support is provided by a single extension `ATI_fragment_shader`. All these extensions use fixed-point arithmetic and are supported only on vendors specific hardware.

There is also a more powerful fragment programming extension: `NV_fragment_program`. It has fixed instruction count limits and it provides several new instructions.

3 High-Level Shading Languages

3.1 Overview of Illumination Models

In computer graphics surface appearance is controlled by illumination models. For interactive computer graphics, various *local illumination models* can be used, while *global illumination models* are computationally more demanding and are better suited for offline rendering. The difference between these two classes lies in the way how light sources are described and interact with surfaces. In case of local illumination models, only a finite number of discrete light sources determine the surface color, while in global illumination models the light that is reflected and transmitted from other surfaces is also taken into account.

Illumination models can be divided into three components: reflection, transmission and emission of light. In case of local illumination models, only reflection and emission components are typically used. Thus, most of the illumination models discussed are really reflection models that can be combined with additional models describing light transmission.

The simplest local illumination model is constant coloring of a surface - by ignoring all light sources and other surfaces, we can use a predefined constant color for all surface points:

$$\mathbf{i}_{amb} = \mathbf{s}_{amb} \tag{11}$$

This kind of illumination is rarely used alone (although commonly used as a component in more complex models), as it provides very little visual information. The simplest practical illumination is based on a Lambert's law which describes ideally diffuse surfaces (dot denotes a scalar product operation):

$$\mathbf{i}_{diff} = \mathbf{s}_{diff} \cdot \max(\mathrm{dot}(\mathbf{n}, \mathbf{l}), 0) \tag{12}$$

Here $\mathbf{n}$ (normal vector) and $\mathbf{l}$ (light vector) are assumed to be of unit length, for illustration look at figure 6.

For shiny surfaces (like plastics or metals), Lambertian model itself does not produce realistic results. Usually such surfaces should have a viewer-dependent specular highlight. One of the simplest terms that can produce such effect is Phong specular term:

$$\mathbf{i}_{spec} = \mathbf{s}_{spec} \cdot \max(\mathrm{dot}(\mathbf{r}, \mathbf{v}), 0)^{s_{shi}} \tag{13}$$

A model that includes diffuse, specular and ambient (constant) terms, as defined above, is usually called Phong model:

$$\mathbf{i}_{Phong} = \mathbf{s}_{amb} + \mathbf{s}_{diff} + \mathbf{s}_{spec} \tag{14}$$

While this model can provide good approximation to matte and plastic surface, it does not produce particularly good results for other surfaces. Thus, new

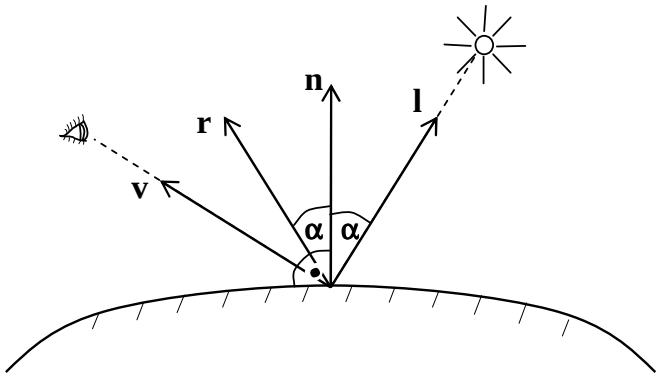


Figure 6: Vectors used in local illumination models. All are assumed to be of unit length.

terms are required for such surfaces. Besides Phong model, there exist several other widely used models like Blinn, Cook-Torrance [28].

3.1.1 Bidirectional Reflectance Distribution Function

The Phong illumination model is an example of an empirical model - the specular component of the model is constructed by *ad-hoc* method and is not physically plausible [15].

Physically plausible models require several additional variables like incoming and outgoing wavelength, polarization, position (which might differ due to subsurface scattering). For most applications, a model that takes all this into account would be simply too complex and computationally expensive. When subsurface scattering and polarization are not required, we can use a simplified function called *Bidirectional Reflectance Distribution Function* (BRDF). BRDF concept is central in *Rendering Equation* [13] which provides a good framework for analyzing reflectance models. Although several formalizations exist, BRDF is typically presented as a function of five scalar variables: $\rho(\theta_i, \phi_i, \theta_r, \phi_r, \lambda)$. It is defined to be the ratio of outgoing intensity to the incoming energy:

$$\rho(\theta_i, \phi_i, \theta_r, \phi_r, \lambda) = \frac{L_r(\theta_i, \phi_i, \theta_r, \phi_r, \lambda)}{E_i(\theta_i, \phi_i, \lambda)} \tag{15}$$

Here θ_i, ϕ_i are spherical coordinates of incoming light (see figure 7). Likewise, θ_r, ϕ_r are spherical coordinates of outgoing light. λ represents the wavelength of light.

L_r is the reflected intensity (*radiance*) in the outgoing direction, E_i is the incoming energy (*irradiance*). BRDF is measured in inverse steradians and can be thought of as a ratio of light reflected per unit solid area. The properties and definitions of formal quantities used in BRDF are presented in [30].

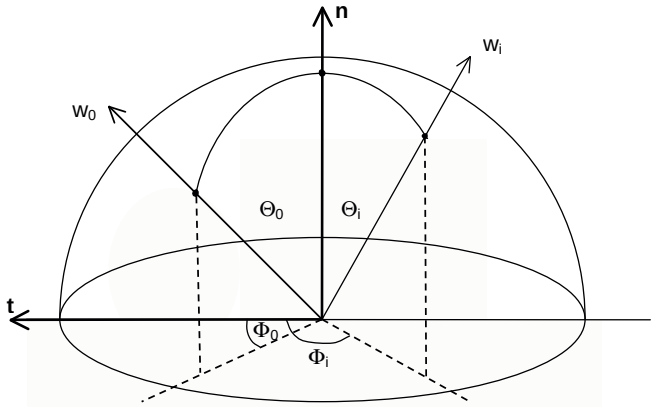


Figure 7: Spherical coordinates for incoming and outcoming light

An importance of BRDF relies on the fact that it can be measured from physical surfaces using a special device [33]. In case when BRDF has been constructed (measured) for a surface, it can be utilized as follows for calculating reflected intensity:

$$L_r(\theta_r, \phi_r) = \sum_{k=1}^{n} \rho(\theta_i^k, \phi_i^k, \theta_r, \phi_r) L_i^k \cos(\theta_i^k) \tag{16}$$

Here n is the number of light sources, L_i^k denotes the incoming intensity (radiance) of light source k at surface while θ_i^k and ϕ_i^k denote spherical coordinates of the light source (light direction from surface).

Although the commonly used RGB color model needs only three wavelength samples, the function still depends on four other variables. When represented as a table (for example, produced by physical measurement), the storage cost can be huge. Several techniques like using spherical harmonics [11], spherical wavelets or spline patches can be used to compactly represent the BRDF of a surface.

An important class of BRDFs are *isotropic* BRDFs - the BRDF of such surfaces does not depend on both ϕ_i and ϕ_r but instead on $\phi_i - \phi_r$. An equation 15 can be simplified for such surfaces. BRDFs that can not be simplified are called *anisotropic* BRDFs.

3.2 Shading Languages

The first shading languages arise from the recognition that as more features were added to local illumination models, their usage became more complex and less intuitive. By replacing fixed illumination model with multiple models for different surface types, most irrelevant parameters can be hidden from the user and usage of relevant parameters can be simplified. The general solution to this problem was given first by Cook [2].

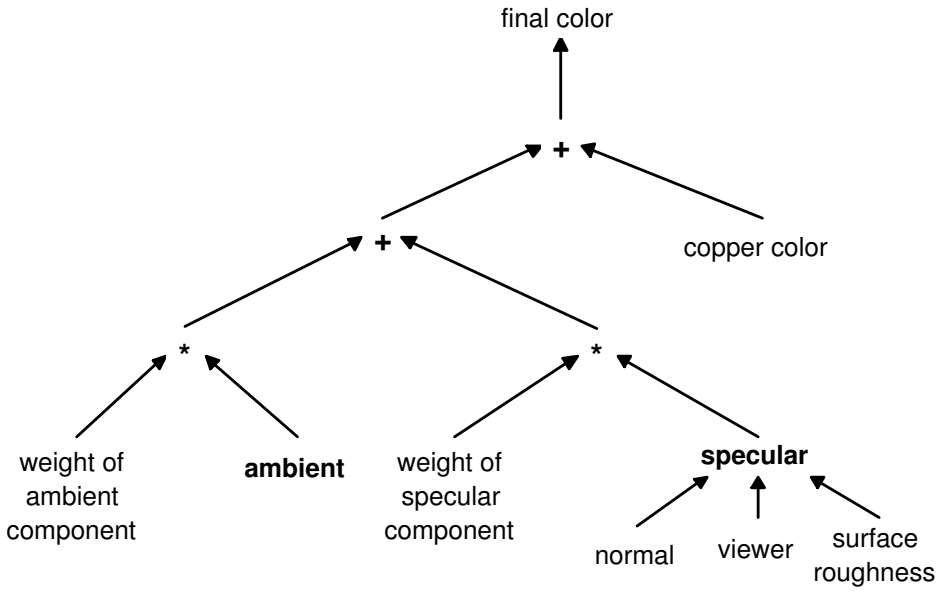


Figure 8: An example of Copper shade tree (presented in [2]).

3.2.1 Shade trees

Cook eliminated fixed reflection equations like Phong illumination model and replaced them with an expression tree that allowed the user to specify relevant components of shading and operations on these components. The result was called *shade tree*. An example shade tree is presented in figure 8.

Each operation was a node of the tree. The input parameters for a 'diffuse' node were surface color $\mathbf{C}$ and surface normal vector $\mathbf{N}$. The node performs an operation $\mathbf{C} \cdot \mathrm{dot}(\mathrm{unit}(\mathbf{N}), \mathrm{unit}(\mathbf{L}))$, where $\mathbf{L}$ is a vector from the surface point to the light (unit denotes a vector normalization operation).

This tree is traversed in postorder. The output of the root node is the final color. The surface normal, object location and various other parameters are leaves of the tree.

Cook also introduced the concept of *light trees* and *atmosphere trees*. Shade trees needed information like incoming light direction and color. Such information was also presented using separate trees. This allowed to reuse same light trees with several shade trees. Atmosphere trees were used to calculate the final displayed intensity which could be different from the light leaving the surface due to various atmospheric effects such as haze.

A high-level, interactive graphical implementation of Cook's shade trees is given by Abram and Whitted [7]. Their implementation has some differences compared to Cook's original concept. Unlike Cook's shade trees where nodes were evaluated from bottom to top, their implementation allowed the user to

```
surface
plastic (float Ka = 1; float Kd = .5; float Ks = .5;
         float roughness = .1; color specularcolor = 1;)
{
    normal Nf = faceforward (normalize(N),I);
    Oi = Os;
    Ci = Os * (Cs * (Ka*ambient() + Kd*diffuse(Nf))
       + specularcolor * Ks*specular(Nf,-normalize(I),roughness));
}
```

Figure 9: A plastic surface shader (taken from [32] library) implemented in RenderMan sl.

define different evaluation order for child nodes ('before', 'during', 'after').

3.2.2 Pixel Stream Editor

Cook's idea was extended by Perlin [25] by allowing more general flow control than was possible in original shade tree. Perlin's language supported conditional and looping constructs, function definitions and logical operations. The language was constructed for an environment called Pixel Stream Editor (PSE). The shaders were split into a series of passes through PSE. PSE acted as a powerful filter on an array of pixels.

Perlin did not divide shaders into classes like Cook. All shader types had to be combined into a single 'surface' shader. Thus, this approach was conceptually on lower level than Cook's for shading calculations. This approach had a serious problem – shaders were executed after visibility calculations and all data that was provided to the shader programs consisted of a triple $(Point, Normal, SurfaceId)$ for each pixel. Global illumination techniques (like radiosity simulations) or ray tracing require more information than surface visibility and can not be incorporated in post processing stage.

3.3 RenderMan

The RenderMan shading language is part of the RenderMan Interface (RI), which was developed by Pixar to standardize 3D scene descriptions and to set several quality standards for photorealistic image rendering. The RenderMan Interface has achieved the status of *de facto* industry standard. Besides shading language, RI also defines The RenderMan Interface Bytestream archive format (RIB), C API and requirements with optional capabilities for a RIB renderer.

The history of RI is tied with the development of REYES rendering system. RI was built on experience that was gathered during experimentation with REYES. The final version 3.1 of RenderMan Interface was presented in 1989 and currently there exist several conforming implementations [32, 8].

RenderMan shading language (sl) resembles C language. The language provides mathematical functions, flow control and several high level features specific to computer graphics. Like in Cook approach, RenderMan shaders are divided into several classes:

- *Light shaders* calculate the incident light intensity and color for given surface point and point of the light source.
- *Surface shaders* compute the reflected light intensity and color at given direction. Typically this involves integrating (in practice, summing) over all light sources.
- *Volume shaders* modulate the light intensity and color when light travels through solid object volume. A special case of the volume shader is called *atmosphere shader* - this encompasses all scene objects.
- *Displacement shaders* are used to modify the geometry of objects. Given a surface point, they compute a displaced surface point.
- *Imager shaders* are used for postprocessing images.

RenderMan includes at least one built-in shader for each class. Figure 10 shows the dataflow between shader classes: surface shader has central part and connects other shaders together.

3.3.1 Types

The sl has only four built-in types: floats, colors, points and strings. The number of components in the color type can be larger than three (depending on implementation), thus larger spectra can be represented than possible using RGB color space. Several color spaces like RGB, HSV, HSL can be used for defining colors. Like colors, points can be defined in different coordinate systems (like 'camera', 'world', 'object').

RenderMan introduces a concept that is not present in most programming languages: variables are divided into *uniform* and *varying* storage classes. Uniform variables are used to specify values that are constant across surface. Unlike uniform parameters (which are specified once per surface), varying parameters are defined at surface vertices. The values for interior surface points are produced using bilinear interpolation of vertex parameters. The shading language allows to convert uniform values into varying values but the reverse operation is not valid. Using uniform values whenever possible can provide faster shading. The final result produced by shaders is almost always varying as uniform expressions can only provide flat shading. Variables and shader parameters can be declared using `varying` or `uniform` modifier.

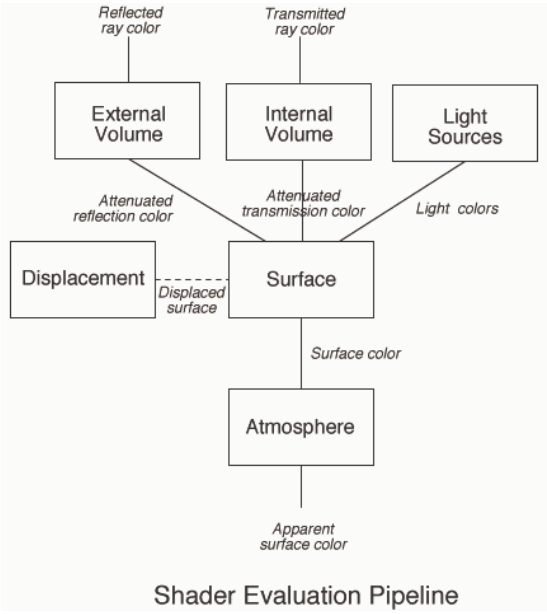


Figure 10: RenderMan shader evaluation pipeline. Central to the pipeline is the surface shader that can query and use results produced by other shaders.

3.3.2 Built-in functions

The set of built-in functions in RenderMan is quite rich. Besides common trigonometrical and algebraic functions, sl also provides *noise functions*, various geometric functions for normalizing vectors, calculating reflection and refraction vectors. To simplify color calculations, functions are provided for calculating diffuse, specular and ambient color contributions, shadowing, texturing, *etc.*

3.3.3 Execution environment

RenderMan provides execution environment with various input and output variables. All input variables are set before the shader is called; shader result is also stored in the environment variables. Table 4 lists some of the surface shader environment variables. Light shaders and other shaders have a similar set of built-in variables.

Surface shaders should overwrite `Ci` and `Oi` variables to provide custom color for material. Variables that contain differential values like `dPdu`, `dPdv`, `du` and `dv` are typically used for prefiltering procedural textures containing high frequencies. RenderMan provides two normal vectors: `N` and `Ng`. `Ng` is always defined as a cross product of `dPdu` and `dPdv`. Shading normal can be provided explicitly together with vertices, otherwise geometric normal is also used as a shading normal.

Variable name	Type and storage class	Description
`Cs`	color, varying/uniform	Surface color
`Os`	color, varying/uniform	Surface opacity
`P`	point, varying	Surface position
`dPdu,dPdv`	point, varying	Derivative of surface pos. along u,v
`N`	point, varying	Surface shading normal
`Ng`	point, varying/uniform	Surface geometric normal
`u,v`	float, varying	Surface parameters
`du,dv`	float, varying/uniform	Change in surface parameters
`s,t`	float, varying	Surface texture coordinates
`Ci`	color, varying	Incident ray color
`Oi`	color, varying	Incident ray opacity

Table 4: Subset of RenderMan surface shader environment variables.

3.4 Real-Time High-Level Shading Languages

RenderMan-like features and quality is the goal for real-time shading languages. Unfortunately, current hardware is not capable of executing general shaders with real-time speed. Although this is the primary reason, there exist several other reasons why specialized real-time languages are useful:

1. Some features of RenderMan are not useful or are too general. For example, global illumination models can be represented in RenderMan, but with lower efficiency compared to specialized systems.

2. RenderMan API is quite different from widespread immediate mode APIs. It is not straightforward to port existing OpenGL or DirectX applications. Thus, shading APIs closer to existing real-time APIs are desirable.

3. Current hardware must support all legacy programs that use fixed function pipeline instead of shaders. Shading languages must interact with fixed function pipeline in a well defined manner allowing applications to gradually replace fixed function programming model with shading model.

From a general architectural viewpoint, OpenGL can be viewed as a complex virtual machine. One of the most surprising results is that by utilizing two additional extensions (plus a few already existing ones), OpenGL can be turned into a general virtual machine capable of executing RenderMan shaders. This approach is discussed by Peercy *et al* in [18]. Their approach is based on an observation that a single rendering pass is basically a SIMD instruction, executing on all fragments of a primitive. Instruction set is formed from OpenGL pipeline states and can be represented by a N-component tuple. Blending mode, alpha

test mode, stencil function and other state parameters are components of this tuple.

Basic arithmetic operations like addition or multiplication can be done in two passes: in the first pass the geometry is rendered to frame buffer without any blending, while in the second pass blending mode is set using `BlendFunc(ONE, ONE)` (for add operation) or `BlendFunc(DEST, ZERO)` (for multiply operation). There are two complications for more general cases: RenderMan assumes that all arithmetic is performed using floating point values, while standard OpenGL provides only values in range [0..1]. The *color range extension* is used to overcome this limitation and provides full floating point range for all operations. Second extension is required to support arbitrary arithmetic functions (by utilizing blending, only a few operations can be provided). Although OpenGL textures can be used as function look up tables, all texture coordinates are associated with vertices and it is not possible to do dependent texturing by reading texture coordinates from another texture. The second extension (*pixel texture*) makes this possible.

Stenciling can be used to perform conditional execution. Stenciling operations mask pixels that do not satisfy given condition and should not be updated. ARB imaging extension is used to implement `while`-loops by executing the body of the loop multiple times and testing for the termination condition using *min-max* function. Simple swizzle operations are done by setting the *color matrix*. Color matrix extension allows to calculate any linear combination of RGBA color components in the OpenGL pixel transfer pipeline.

3.4.1 Interactive Shading Language

Besides implementing a RenderMan compiler for OpenGL, Peercy and others have also implemented a simplified shading language called Interactive SL [24]. Unlike general RenderMan compiler, this language does not require any OpenGL extensions beyond color matrix transformation. Some restrictions were put on language (language does not allow to use varying nested expressions) to make worst-case temporary storage requirements explicit. Also, as dependent texturing extension is not needed, language supports only texture coordinates that are associated with vertices (or derived from vertices using texgen functionality).

Paper [18] discusses performance of both approaches: RenderMan shaders and shaders implemented in Interactive SL. RenderMan shaders with moderate complexity required at least 100 OpenGL passes. Even if both required extensions were supported by hardware, the renderings would probably not be produced in real time. Simple Interactive SL shaders required around 10 passes and were rendered on Silicon Graphics hardware at interactive rate (around 10 frames/sec).

In general, although OpenGL with the two described extensions can be used as a general virtual machine capable of executing general shading languages, the resource requirements for real-time rendering are huge if shaders are complex and

require many passes. A lot of redundant processing is needed as all geometry has to be transformed for each pass. As each pass uses blending operations, bandwidth requirements are huge, likewise. The extensions required (like ARB imaging subset) can be considered expensive and are more general than really needed.

3.4.2 SMASH and Sh

SMASH (Simple Modelling And SHading) is a project to study codesign of next-generation graphics accelerators and APIs [21]. SMASH API is loosely based on OpenGL with large parts of fixed functionality replaced with programmable subsystems. The idea behind SMASH was not to create a replacement for OpenGL but to guide its evolution. SMASH can be layered on top of OpenGL.

At the basic level, SMASH can be thought as another vertex and fragment programming extension - SMASH provides unified shader model. This means that both vertex and fragment shaders have identical capabilities. SMASH also includes support for programmable geometric primitive assembly, which is not present in OpenGL. SMASH divides shader parameters into five categories: generic, color, primal geometry (points, tangents), dual geometry (normals, planes) and texture coordinates. Each of these categories has different automatic transformations. For example, for a primal geometry vector, modelview transformation is used, while for dual geometry a transpose inverse modelview matrix is used instead. This classification simplifies handling of different entities and handles common operations automatically for user. Transformations are always linear or affine (depending on the entity category).

SMASH uses stack-based shader programming model. There are no temporary registers visible to user. All operands are read from the stack and results pushed to the stack. Vectors of arbitrary size are used as primary types (although vector sizes are fixed at compile time). Support for conditional execution is provided, but looping constructs are not supported.

Due to the stack based programming model, it is possible to use features of standard C++ to define a high-level shading language directly in the API, without once having to resort to string manipulation of embedded shader scripts or use external files. The use of syntactic sugaring (like operator overloading) provided by C++ allows automatic parsing of expressions during application program compilation [22]. This embedded language can provide more direct interaction with the specification of textures, attributes and parameters. This is the primary reason why SMASH can be classified as a high level shading language. Authors call the embedding technique *shader metaprogramming*. The term metaprogramming means that one program is used to generate and manipulate another (shader in this case).

Sh [20] is based on the same metaprogramming principles as SMASH - but it can be layered directly on top of OpenGL shading extensions (like OpenGL ARB

```
ShShader vsh = SH_BEGIN_VERTEX_PROGRAM {
  ShInputNormal3f normal; // input normal
  ShInputPosition4f p;    // input position
  ShOutputPoint4f ov;     // output viewer-vector
  ShOutputNormal3f on;    // output normal
  ShOutputVector3f lvv;   // output light-viewer vector
  ShOutputPosition4f opd; // output transformed position

  opd = Globals::mvp | p;
  on  = normalize(Globals::mv | normal);
  ov  = -normalize(Globals::mv | p);
  lvv = normalize(Globals::lightPos - (Globals::mv | p)(0,1,2));
} SH_END_PROGRAM;

ShShader fsh = SH_BEGIN_FRAGMENT_PROGRAM {
  ShInputVector4f v;      // input viewer-vector
  ShInputNormal3f n;      // input normal
  ShInputVector3f lvv;    // input light-viewer vector
  ShInputPosition4f p;    // input transformed position
  ShOutputColor3f out;    // fragment output color

  out(0,1,2) = Globals::color * dot(normalize(n), normalize(lvv));
} SH_END_PROGRAM;
```

Figure 11: Lambertian shader for a single light source. Written in C++ using Sh metaprogramming library.

vertex and fragment programming extensions).

Note that SMASH nor Sh do not support shader types like light shaders or deformation shaders. All shaders must be explicitly combined into a single surface shader. Also, when Sh is layered on top of ARB vertex and fragment programming extensions, application must provide separate vertex and fragment programs. An example Sh shader is presented in figure 11. This example uses global variables `mv`, `mvp`, `lightPos`. These denote modelview matrix, concatenation of projection, modelview matrix and light position vector and are part of OpenGL state. An important relation should be noted – vertex program output declarations must match fragment program input declarations. This way, output registers of the generated vertex program are allocated in the same order as input registers of the generated fragment program.

3.4.3 HLSL and Cg

HLSL [23] (High-Level Shader Language) is a high level shading language designed for Microsoft DirectX 9 API. Cg ("C for Graphics") is its multiplatform extension that works also on top of OpenGL. As both languages are syntactically

and semantically very close, only Cg is discussed here. Name Cg captures the concept of the language more closely, due to strong resemblance to C and by providing features beyond simple shading. Although languages themselves are similar, shading APIs are quite different.

Cg does not divide shaders into classes like RenderMan, only combined surface shaders are supported. In fact, surface shaders are typically presented using two Cg shaders: vertex shaders and fragment shaders. Both correspond directly to the vertex and fragment programming extensions described in chapter two.

Cg can be combined with CgFX extension, to declare multipass effects, samplers with texture and filtering attributes and various graphics pipeline states like polygon *culling mode* [19].

All this makes Cg closer to hardware level and less abstract than RenderMan. The Cg compiler by NVIDIA supports different hardware profiles with differing capabilities. There are minimum requirements that all profiles must follow while some features are made optional for the profiles. For example, some profiles may not support `while`-loops in vertex programs. The data types in Cg correspond roughly to C data types. An interesting addition is the `half` data type. This provides 'half'-precision (16-bit) floating point calculations, which are usually sufficient for color calculations.

3.4.4 The OpenGL Shading Language

The OpenGL Shading Language (usually abbreviated as *glslang*) is part of the OpenGL 2 specification [3]. At the time of writing, there are no known hardware implementations supporting all required features of this language. In general, the language is similar to HLSL and Cg and does not provide higher level features of RenderMan shading language. One of the most noteworthy concepts of the OpenGL 2 is the omission of lower level (assembly level) shading language. The reason for this is not to constrain the graphics architectures and provide more options, as low-level compatibility may hinder innovation in graphics hardware implementations.

3.4.5 Stanford Real-Time Shading Language

Stanford Real-Time Shading Language (Stanford RTSL) was one of the first real-time shading languages that was designed for consumer-level graphics hardware [14]. The project started with a goal of creating an intermediate layer between OpenGL hardware and applications, with an intent of supporting multiple hardware targets through multipass techniques. The language has evolved through several iterations, the first versions of the language were Lisp-like and provided support for combined surface shaders only. The current version of the language resembles more C and RenderMan sl and provides support for surface and light shaders.

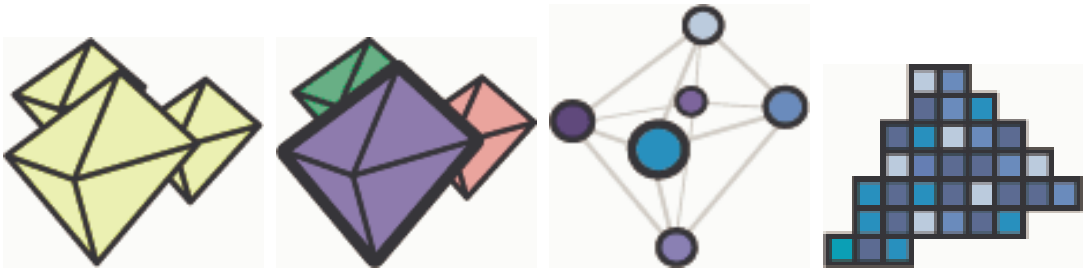


Figure 12: Constant, per-primitive group, per-vertex, per-fragment computation frequencies

The most notable idea of the Stanford RTSL is the concept of *computation frequency*. This is a generalization of the RenderMan *varying* and *uniform* parameters. Four different frequencies are supported in RTSL: constant, primitive group, vertex and fragment. Figure 12 illustrates these frequencies. Note that *per-primitive* frequency is missing. As OpenGL API and shading extensions do not provide support for such frequency, it is omitted. RenderMan varying parameters correspond rougly to vertex frequency, while uniform parameters correspond to primitive group frequency (ideally they should correspond to per-primitive frequency). For operations, frequency specifies how often the computation is performed. Expressions involving varying parameters may correspond to both fragment and vertex frequency (in general, linear combinations of varying parameters can be represented at vertex frequency, while other expressions of varying parameters should be expressed at fragment level). The language allows the trading of some efficiency for correctness. If user wants to perform less computations per-fragment; such computations can be approximated at vertex level. For example, diffuse lighting calculations can be represented at vertex level with sufficient quality in many cases, while specular calculations at vertex level produce typically poor results.

Language supports natively three- and four-component vectors, 3-by-3 and 4-by-4 matrices, `float` type, texture object references (`texref` type), clamped values in range [0..1] (`clampf` type). Unlike RenderMan, there is no distinction between geometrical points and colors, both can be encoded using either three- or four-component vectors. Likewise, the user must explicitly transform points and vectors if multiple coordinate systems are needed.

Not all operations are available at all frequencies. In general, at each higher frequency level, the number of operations decreases. There are some exceptions, for example, texturing is available only at fragment level, but not at lower frequencies. Matrix calculations are not supported at fragment level, while limited support is provided for such operations at vertex level.

Stanford RTSL system has modular design and supports several targets for

each computational frequencies. Primitive group and vertex expressions can be compiled into C or Intel x86 machine code. Additionally, `NV_vertex_program` extension is supported for vertex calculations. At fragment level, compiler supports standard OpenGL multitexturing (with limited functionality), `NV_register_combiners` extension and `NV_fragment_program`. Targets at different frequencies can be combined with each other.

3.4.6 ASHLI

ASHLI (Advanced Shading Language Interface) is a real-time rendering system supporting multiple shading languages including HLSL, glsl and RenderMan [1]. ASHLI supports only a subset of these languages and is really intended to work as a bridge between digital content creation software (like Maya, 3D Studio Max or Softimage) and hardware. Compared to other shading languages, ASHLI architecture resembles Stanford RTSL – both languages are specialized for shading, while glsl and HLSL are more like hardware model abstractions.

A unique feature of ASHLI, compared to glsl or HLSL, is the ability to generate multiple passes. This is essential considering the targets of the language – as shaders created by artists inside content creation software can have arbitrary complexity and only a very limited subset could run within a single rendering pass.

ASHLI provides API for initializing, specifying and compiling shaders, but not for directly rendering shaded primitives or objects. After the shader has been compiled, application can query shader assembly code (ASHLI supports DirectX 9 shader models and OpenGL ARB vertex and fragment programming extensions) and *formals*. Formals describe how to bind textures and set program constants. In the case when shader requires multiple passes, formals describe how intermediate pass results should be stored.

Although ASHLI provides compiler front-end for RenderMan shaders, hardware shader models do not support features like branching and subroutine calls. Thus, support for `for`-loops is currently limited to loops with constant loop count and general `while`-loop support is missing. Also, not all data types are fully supported (array support is limited) and several function primitives are not currently implemented. This means that using existing RenderMan shaders can be problematic, but when such limitations are known at shader design time, most of the issues could be resolved.

4 Experimental Shading System

4.1 Overview

This section gives an overview of an experimental shading system designed and implemented by the author of this thesis and describes important issues that are common to most real-time shading languages. The described shading system is based on Stanford RTSL but includes several important extensions and changes. At the same time, it provides support for only a single hardware target and works at the time of writing only on DirectX 9 compatible consumer graphics processors.

An example shader written for this system is presented in figure 13. It shows a Blinn-Phong lighting model implementation. The shading language is based on the concept of four computing frequencies like Stanford RTSL and supports three classes of shaders: surface shaders (as shown in figure), light shaders and deformation shaders.

The syntax diagrams of the shading language are presented in appendix A. Like Stanford RTSL, the language has been revised and the presented syntax is used for the second version of the language. The revision was made to support newer hardware targets and to provide more functionality in the language core and less in its external library. The syntax is similar to both C and RenderMan. For example, the language supports similar syntax for comments and preprocessor directives (with a few deprecated features and extensions). All identifiers are case-sensitive.

The current version targets ARB-approved fragment and vertex program extensions. The first revision supported actually more targets like OpenGL 1.1 multitexturing and `ATI_fragment_shader` fragment targets, but these were removed later in development process. Although these targets had special optimizers, only very limited fragment computations were possible. It was very easy to produce some unintended artifacts due to the reason that all calculations were performed in fixed-point arithmetic and it was very easy to loose precision in intermediate calculations. Supporting multiple targets in quite rapidly evolving system required too much technical work and was not in line in creating an experimental system.

4.2 System Description

The shading system (see figure 14) is divided into two main modules: compiler and runtime. Both can work without another, the compiler module is completely separated from the runtime and acts as a server in the client-server paradigm. Compiler provides a relatively high-level interface for the runtime for loading shader programs, binding light shaders to surfaces and compiling shaders. Nevertheless, it is possible to use the runtime without the compiler part. In this case,

```
surface shader blinn(float Ks, float4 Cd, float4 Cs)
{
    float4 Csurf = integrate(float4 Cl) {
        float3 c = lit(dot(L, N), dot(H, N), Ks);
        return (c[1] * Cd + c[2] * Cs) * Cl;
    };
    return Ca + Csurf;
}
```

Figure 13: A simplified Blinn-Phong surface shader implemented using shading language.

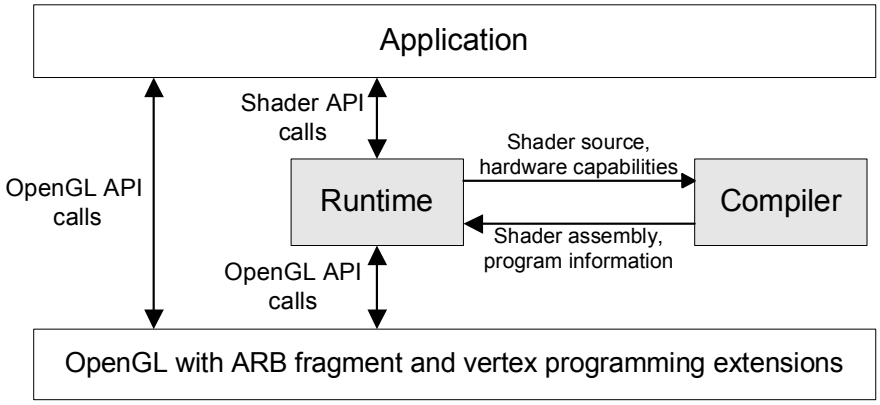


Figure 14: High-level view of shading system architecture.

it is the user's responsibility to supply system compiled fragment and vertex programs via *shader cache* file (it is described in the following sections).

The runtime is exposed to the user as OpenGL like API, syntax and basic immediate mode functions have same signatures as in Stanford RTSL API. The runtime API is used for shader loading, shader compilation, shader parameter setting, *etc* (C language bindings are given in appendix C). A typical application performs the following steps for setting up a proper environment (actual procedure names are shown in parenthesis, asterisk denotes a procedure from a procedure group):

```
Initialize OpenGL
Initialize Shading System (Init)

Load secondary (light and deformation) shaders (LoadShader)
Query parameter handles for input variables (ParameterHandle)
Set constant frequency parameters (Parameter*)
Compile secondary shaders (CompileShader)

Load surface shaders (LoadShader)
```

```
Query parameter handles for input variables (ParameterHandle)
Set constant frequency parameters (Parameter*)
Bind light and deformation shaders as needed (UseLight, UseDeformation)
Compile surface shaders (CompileShader)
```

After these steps the application rendering loop is run, consisting of the following steps for each frame:

```
Application specific handling of shader input parameters
For each primitive group
  Bind shader (BindShader)
  Set primitive group frequency parameters (Parameter*)
  For each primitive in group
    Start primitive group (Begin)
    For each vertex in primitive
      Set per-vertex parameters (Parameter*)
      Specify tangent-space vectors (Tangent*, Binormal*, Normal*)
      Specify vertex position (Vertex*)
    End
    Finish primitive group (End)
  End
End
```

The pseudocode shown here uses immediate-mode API. For higher performance, shading system provides vertex object API that greatly reduces the number of API calls needed for rendering.

4.3 Shading Language Description

This section gives an overview of different aspects of the shading language design. Most emphasis is put on concepts specific to shading languages and features that differentiate our experimental shading language from other shading languages.

4.3.1 Basic type system

As the real-time shading system's primary goal is efficiency and speed, the field specific abstraction level of type system is lower than in RenderMan type system. No distinction is made between color values, vectors, points. The user must select appropriate types for these. If multiple coordinate systems are used, it is the user's responsibility to perform any coordinate system transformations needed. This leaves more options for the user to optimize in special cases, but may require more coding for common cases.

Basic types are formed using five fundamental and two compound types, listed in table 5. All types have constructors, except `texref`. This omission is intentional, as hardcoding texture handles in shader code is bad practice as this reduces code modularity and creates implicit dependencies on texture handles. Texture handles must be specified by the user, through the runtime API.

Keyword	Description	Example of usage
`bool`	boolean type	`bool b = true`
`int`	integral value	`int x = 2`
`texref`	texture object reference	`texref t`
`clampf`	floating point value in range [0..1]	`clampf f = 0.5`
`float`	full range floating point value	`float f = 1.2`
`[ ]`	tuple, a list of basic types	`float[2] v = { 1.2, 1.3 }`
`struct`	compound structure type	`struct { float a; bool b; } s = struct { a = 0.1, b = false }`

Table 5: Shading language built-in basic types.

Tuple type and structure type provide projection operators. Note that tuple type is different from array type present in most languages; tuple size can not be changed once it is specified and projection operator requires constant index.

Note that there is no *void* type as in C. This is because language does not allow any side effects like assigning global variables or writing to a memory location.

The following types are defined for shorthand: `float2`, `float3`, `float4`, `clampf2`, `clampf3`, `clampf4`, `matrix2`, `matrix3`, `matrix4`, `deformation_t`, `fragment_t`. The last two types are used for *deformation shaders* and extended surface shaders.

The language provides two kinds of type conversions: implicit and explicit. In case of implicit conversions, types are converted automatically when needed. Precision is never lost in this case. Implicit conversions are provided from `int` to `float` and `clampf` to `float`. Compound types can be implicitly converted if all their components can be implicitly converted.

Explicit conversions are provided currently only from `float` to `clampf`. The user must explicitly specify this conversion using the cast operator (casting syntax is similar to C). Conversion from `float` to `int` is not available, mostly due to a very limited support for integer types.

4.3.2 Computation frequency

Full, qualified types within the language include two components: basic type and computation frequency (listed in table 6). Frequency can be specified together with a basic type by prepending the basic type with a frequency specifier (e.g. `vertex float k`). Computation frequency does not have to be explicitly specified, as it can be deduced from frequencies of variables and expressions. For example, assuming expressions `e1` and `e2` have fragment frequency, we can deduce that `e1 + e2` also has fragment frequency, assuming `+` operator is available at fragment frequency for types of `e1` and `e2`. In general, explicit computation

Keyword	Description, typical usage
`constant`	value can be specified before code generation and does not change afterwards
`primitive group`	value can be specified outside `Begin`/`End` and does not depend on every primitive scope
`vertex`	value varies per-vertex or per-primitive
`fragment`	value has to be calculated per-fragment

Table 6: Shading language computation frequencies.

Keyword	Description
`surface`	surface-specific context
`light`	light-specific context
`deformation`	deformation-specific context

Table 7: Supported shader contexts.

frequency should be avoided if it is not necessary, as the compiler automatically selects operations at the lowest possible frequency. When frequency is omitted from shader parameter declarations, primitive group frequency is assumed by default. Shader parameters can have any frequency except fragment frequency – if fragment frequency inputs are required, then texture operations can be used.

In most cases when operation can be done at a lower frequency, then it should be done so – as performing the operation at a higher frequency provides same results. An exception to this are fragment level computations, for which results may be very different if operations are performed at vertex level instead of fragment level. For example, unit normal vectors are often used in shading calculations. When normals are specified per-vertex, then by normalizing vectors at vertex level and then using normals in fragment level calculations may produce unexpected results. The reason for this is that at fragment level, interpolated normals are used – but when unit normal vectors of same primitive are not collinear, then interpolated normal will be shorter than one unit. This problem can be simply fixed by renormalizing vectors at fragment level instead.

4.3.3 Contexts

Language includes three contexts (see table 7) that correspond to the shader classes supported by the language. Each shader object (instance) has its own context state and can access only the variables of given context type (for example, light shaders can not use variables declared with surface context and vice versa).

In most cases, users must deal with contexts only when declaring shaders – then the shader class must be explicitly given (by prepending the shader output

type with `surface`, `light` or `deformation`). In some cases, users may need to explicitly specify contexts when new variables or functions are declared. This is needed when the variable or function refers to a context not available in the current scope. This helps the user to avoid the propagation of unwanted contexts far from the actual error.

4.3.4 Language constructs

The language supports a subset of C programming language expressions and statements (diagrams of language syntax are given in appendix A).

The language supports similar loop and conditional execution statements as does C language. Two minor differences should be noted here: first, statement blocks in `while` and `if` clauses must always be inside braces. Second, `return` statement must be always the last statement of a function or a shader - early exit is not supported.

The language supports subset of operators available in C language. Most C language binary operators are available, but only simple assignment operator is supported (complex assignment operators like `+=` or `--` are currently not supported). All arithmetic and logical operators have same relative precedence as in C. Assignment operator is treated differently compared to C language - assignment is treated as a statement, not expression.

But due to limited capabilities of the ARB vertex and fragment extensions, several limitations apply to usage of language constructs. First, `while` and `for` loop support is limited - conditional expressions in these constructs must have constant frequency. Simple conditional execution is provided using `?:` operator (condition can have any frequency), while `if` construct requires the condition to have constant frequency.

For same reasons, recursion in function calls is not supported (neither direct nor indirect). Such restrictions are necessary as intended targets for the language do not support general function calls, which means all program code must be inlined.

It is clear that language does not belong to the class of programming languages equivalent to Turing machines or partially recursive functions, when computations need to be performed at vertex or fragment level. The simple execution model guarantees that programs written in the language always terminate after they have been compiled (although it is possible to create non-terminating program at constant frequency). In fact, the number of executed instructions always remains constant and does not depend on input data in vertex and fragment programs.

Perhaps with such limitations one may assume that language is very limited. Fortunately most local illumination models have the following form:

Primitive	Supported frequency	Description
`rcp`	all frequencies	Reciprocal (approximation) of scalar
`invert`	prim. group	Inverted 2x2, 3x3, 4x4 matrix
`lit`	fragment/vertex	Phong lighting model coefficients
`dist`	fragment/vertex	Distance attenuation factors
`tex1D`	fragment	1D texture read
`lerp`	fragment	Linear interpolation

Table 8: Small subset of our shading language primitives.

$$\mathrm{s}(x) = \mathrm{f}(x) + \sum_{k=1}^{n} \mathrm{l}_k(x) \tag{17}$$

Such construct is supported within the language by `integrate` operator with a restriction that n and l_k must be specified before compilation (thus, light sources can be considered as constant frequency parameters for surface shaders). A typical usage of `integrate` operator is shown in figure 13. The operator requires user to specify a variable that will be bound to light shader output. The output values of operator code block are added together (by applying a `+` operator).

Note that `integrate` operator is the only way how surface shaders can query information about lights. All other queries result in dependence on `light` context, which can not be removed without applying `integrate` operator.

4.3.5 Primitive functions

All built-in primitive functions are shown in *Prelude file* (this file includes declarations of primitive functions and definitions of built-in variables), which is listed in appendix B. For reference, a subset of the primitives is listed in table 8. Primitives for vertex and fragment frequency were selected to be as close to instruction level as possible. Most primitives correspond to single instructions, while more complex operations are defined using composition of such primitives in Prelude file.

Most simple arithmetic operations like addition, componentwise multiplication are defined for scalars, tuples of size two, three and four. Although it is technically fairly straightforward to implement support for finite tuples of arbitrary length, this has proven to be unimportant in practice.

Note that most operations are defined within single frequency, they take arguments of certain frequency and produce output with same frequency. The notable exceptions are texture instructions (instructions with `tex`-prefix). They require the texture reference argument to be at constant or primitive group frequency. This reflects the restriction in OpenGL that textures can be bound only

outside `Begin-End` scope. As the primary intention for `texref` type is to pass texture handles to texture operations, no support is provided for `texref` type above primitive group frequency. It is possible to create examples that seemingly use `texref` variables at higher frequencies, but the results are never affected by `texref` values in such cases.

In case of texture operations, two versions are provided for most operations: non-projective and projective. For cube maps, only a single version is provided (projective and non-projective versions are equivalent). Projective texturing is not strictly required - it can be implemented with an ordinary texture instruction and a multiplication/division, but direct support provides higher performance. In addition, the language provides support for checking whether a `texref` expression is valid using `isnil` primitive (`texref` handle must be non-zero). Texture border color, filtering and other information comes from the texture object, referenced via `texref` parameter. It is not possible to access or update this information inside shaders directly - controlling this is application's responsibility.

4.3.6 Function lookup scheme

The shading language supports *function overloading* in the same sense as C++ does. For example, it is possible to define two functions with the same name and same number of arguments, but with different argument types and frequencies. Overloading support is important for built-in arithmetic operators like `+`, `*` and allows the user to write more generic code.

Compared to th C++ type system, our shading language also uses the computation frequency concept. This complicates function resolving scheme. For example, assuming the following two operators are available:

```
vertex   float operator + (vertex   float, vertex   float);
fragment float operator + (fragment float, fragment float);
```

Which one should the compiler use? Intuitively, the answer is simple: if at least one of the arguments has fragment frequency, the second function must be used. While in case when arguments have vertex or lower frequency, it is better to use the first one. Better in this context means that is likely more efficient - as typically the number of fragments generated during rasterization is higher than the number of vertices used. An issue that further complicates resolving is that language allows user-defined functions with no explicitly specified parameter frequencies:

```
float operator + (float x, float y) { ... }
```

The lookup scheme is a generalization of the example above with the goal of trying to minimize frequency casts. Resolving consists of four consecutive steps, assuming we have a function application f($A_1, A_2, ..., A_n$):

1. Consider only the functions with the given name and arity n. If there are no such functions, report an error. Otherwise proceed to the next step with the selected functions.

2. Consider a possible matching function f($T_1, ..., T_n$) only when arguments A_1, ..., A_n can be converted to T_1, ..., T_n, respectively. This means that T_k must have frequency that is higher or equal to the frequency of argument A_k, for all $1 \leq k \leq n$. Also, conversion from A_k to T_k must not reduce argument precision, otherwise it is not considered valid. For example, conversion from `clampf` to `float` is valid, while conversion from `float` to `int` is not. If none of the functions pass this test, report an error. Otherwise proceed to the next step with the functions that satisfy described conditions.

3. Order remaining functions by number of frequency conversions necessary. Functions are associated with vectors of the form (C_3, C_2, C_1, G), where C_3 counts frequency conversions from constant to fragment frequency, C_2 counts conversions from constant to vertex frequency and from primitive group to fragment frequency, C_1 counts frequency casts from constant to primitive group frequency and so on. G counts generic parameters in function signature (generic parameter is defined as parameter that has not been given explicit frequency). Functions with minimum cost are only considered in the next step (there can be multiple such functions).

4. If there are two or more candidates left, then if one of the remaining functions has base types $T_1, ..., T_n$ that matches exactly base types of arguments $A_1, ..., A_n$, then this function is preferred and the other candidates are discarded. If there are two or more candiates left at this point, an ambiguity error is reported. Otherwise the remaining function will be used.

This function lookup scheme works well in practice - with primitive functions and generic functions implemented in Prelude file, most efficient function is almost always selected. It is possible to construct cases, when ambiguity errors are reported, but user can always add explicit type casts to promote arguments to expected input types (at least when function declaration includes explicit parameter frequencies).

4.3.7 Built-in variables

Besides shader input parameters, it is possible to declare variables at program scope. Such variables can be used in all shaders within the program. Such variables are used to make a subset of OpenGL state available to shaders. For example, `Mproj` variable is automatically set by the runtime when a new primitive group is rendered. Runtime reads OpenGL projection matrix and sets `Mproj` variable. The value can then be used in all shaders, as this variable is defined in Prelude file.

Variable	Context	Frequency	Description
`P`	surface	vertex	Vertex position in eye-space
`E`	surface	vertex	Unit length vector from eye to vertex
`T, B, N`	surface	vertex	Vertex tangent-space vectors (eye-space)
`Cprev`	surface	fragment	Frame buffer color at current fragment
`Ca`	surface	prim. group	Global ambient color
`L`	light	vertex	Surface-light vector
`Mview, Mproj`	none	prim. group	Modelview and projection matrices

Table 9: Subset of shading language built-in variables.

Light state is passed to shaders in a similar way. As light state must be associated with each light separately, `light` context is used for light variable declarations.

The language also provides a special variable called `Cprev`. This variable has `surface` context and can be used to create semi-transparent materials. This variable can be read to get the color of previously rendered pixel at the fragment's position. A subset of built-in variables is given in table 9.

4.3.8 Swizzles

Special syntactic sugar is provided by the language to support permutations and projections of tuple types. For example, `pos.yxzw` expression (assuming `pos` is declared as a four-component tuple) returns a tuple where x- and y-components are swapped. Similarly, `pos.z` extracts z-component (and is equivalent to `pos[2]`) of the variable. Only x-, y-, z- and w-components can be projected/swizzled, but this is not really a limitation considering the application field of the language.

Language supports also component assignments. For example, `pos[3] = 1` (or `pos.w = 1`) is a valid statement - other components are left unchanged.

4.3.9 Metaprogramming support

Shading system provides support for metaprogramming, although at a different level compared to SMASH or Sh shading languages described in the previous chapter. Two features are very helpful when shaders must be created dynamically at runtime: `UseShader` API procedure and constant frequency parameters.

`UseShader` API procedure is useful when complex shaders can be built from simple shaders. By using this procedure, the user can link one shader output to another shader input. This way it is possible to create a small number of simple shaders and combine these at runtime. Any custom directed acyclic graph can be created where each node is a user-defined shader.

Constant frequency parameters, on the other hand, are useful for creating more complex shaders which are automatically simplified at the compile time. For example, consider the following shader:

```
surface shader float4 tex_surface(constant texref tex, constant float4 color)
{
  float4 result = color;
  if (!isnil(tex)) {
    result = result * tex2D(tex, P.xy);
  }
  return result;
}
```

When a texture reference `tex` passed to the shader is valid, then the shader returns surface color multiplied by texture color, as expected. When the texture reference is not valid (handle is equal to 0), then surface color is simply returned and no texturing occurs. Note that this is not an optimization made by the compiler, but semantics of constant frequency evaluation. The compiler does not generate code for the statement inside the `if`-clause when the condition is false. This means that the user can be sure about which parts of the code are left in/out, without having to wonder whether an optimization rule is applied or not. Thus, the user can write more general versions of shaders without worrying about performance.

Besides constant frequency parameters, preprocessor can be used to customize shaders at program level (preprocessing is done once, when a shader program is loaded). Preprocessor can be used to select between multiple code paths depending on which OpenGL extensions are supported. For example, if the OpenGL implementation supports `ARB_fragment_program_shadow` extension, a preprocessor token `ARB_fragment_program_shadow` is set. An application that utilizes *shadow maps* [29] can be optimized for this extension while providing also a fallback code path for non-supporting targets:

```
  ...
#ifdef GL_ARB_fragment_program_shadow
  float visible = shadow2D(shadowmap, shadowcoord).w;
#else
  float4 depth = tex2D(shadowmap, shadowcoord.xyw);
  float visible = depth >= shadowcoord.z / shadowcoord.w ? 1 : 0;
#endif
  ...
```

Some preprocessor tokens are used to pass hints to the compiler to generate more optimal code in cases where generic solution is not necessary. For example, if the application can guarantee that it always loads orthonormal matrices for modelview transformations, the same transformation can be used for normals also, instead of its transposed inverse. This reduces the amount of work

needed to be done by primitive group or vertex programs. At the time of writing, three such preprocessor variables are exposed: `__HINT_UNIT`, `__HINT_PROJECTED`, `__HINT_RIGID`. The first, if defined, assumes that normals, binormals and tangents are always defined as unit vectors. The second variable assumes that vertices in `Vertex*` calls are always projected to $w = 1$ plane and that modelview transformation does not modify w-coordinate. The last variable assumes that modelview matrix expresses a *rigid transformation* (matrix is orthonormal). The application can pass information if the system is allowed to make such assumptions by calling `Properties` API call. The users can define their own tokens and implement application-specific optimizations in a similar way.

4.3.10 Shader output

Language places different requirements to shader output types depending on shader class. Light shaders can return any type, but surface shaders must be customized for these output types (the type used in `integrate` operator must be compatible with light shader output type). Typically `float4` type is used to pass the light color and intensity information reaching a surface point.

Deformation shaders must return an expression with `deformation_t` type - this is defined as a structure:

```
typedef struct {
  float4 position; float3 normal; float3 binormal; float3 tangent;
} deformation_t;
```

All fields in the output structure must be assigned; all values are expected to be in object-space coordinate system. The default deformation shader (which is applied automatically, but can be overridden by user) simply assigns the user-specified vertex position, normal, binormal and tangent vectors to the structure fields.

Two different return types are supported for surface shaders. First, `float4` type is supported, which is used by most surface shaders. The return value specifies the color of the fragment and its alpha channel. In rare cases, the shader may need to overwrite the default fragment depth or discard the fragment from further processing. For such cases, `fragment_t` type is also supported. This type is defined as:

```
typedef struct {
  float4 color; float depth; bool kill;
} fragment_t;
```

Surface shader does not have to assign all `fragment_t` fields - all fields are optional. When `depth` field is updated, it is expected to contain a new z-coordinate of fragment in window coordinates.

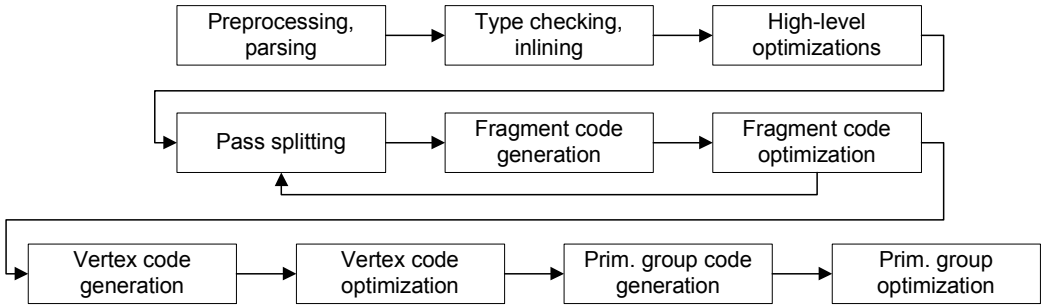


Figure 15: Shading compiler pipeline.

4.4 Compiler Pipeline

A high-level view of the compiler pipeline is presented in figure 15. In the following subsections we describe these stages, most of which contain several substages. The emphasis is put on issues that are specific to shading compilers.

4.4.1 Preprocessor

The preprocessor stage has two common uses: first, it is used in Prelude file for expanding code blocks with simple substitutions and second, it allows the user to specify hints about application behaviour and query supported extensions. Preprocessor syntax is very similar to C preprocessor syntax, although support for macros is not provided.

4.4.2 Parsing, type checking and inlining

Prelude file is used by the compiler front-end to define commonly needed functions and to pass information to front-end about back-end capabilities. It is automatically loaded before shader programs. Prelude file is exposed to the user, like header files in C language.

The central part of the front-end is the type checking and transformation part, which builds a single type checked directed acyclic graph (DAG) representation from the parse tree. Constant frequency variables are also assigned at this stage and all constant-frequency expressions/statements are evaluated. In case of non-surface shaders, this process is straightforward. In case of surface shaders, the system also needs a list of bound light shaders and a deformation shader. These shaders must be compiled beforehand. Light shaders are substituted and their results are added together during transformation. Deformation shader is substituted into code at the post transformation stage. The compiler uses a special primitive `__DEFORM()` to decouple actual the deformation shader from light and surface shaders. All references to `__DEFORM()` are substituted with the actual deformation in the post transformation stage.

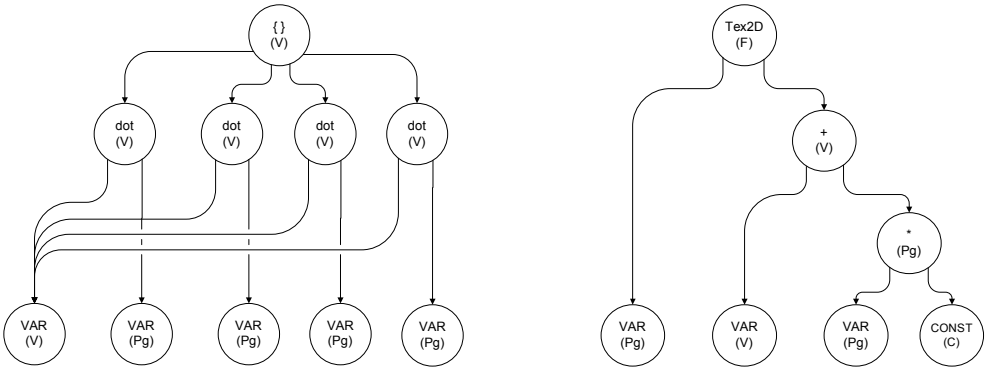


Figure 16: Inlined shader DAG. Each node represents either primitive function application, variable, constant or frequency conversion (not shown in figure). Each node has specific frequency, shown in parenthesis (F - fragment, V - vertex, Pg - primitive group, C - constant).

The generated DAG forms a *core language* for the later stages. Nodes in this DAG have one of four forms: a primitive function application, a frequency cast, a free variable reference or a constant. There are no updates, no temporary variables, no loops, no support for recursion of any kind. As the generated graph is always acyclic, it can be traversed without checking for loops. The rule that is always forced by the type checker, is that subexpressions of an expression always have same or lower frequency than the expression itself. All contexts are resolved in the core, which means that context dependent variables are always tagged with the information describing their source (for example, light shader input parameters are tagged with light identificators).

Figure 16 shows a simplified example of the generated DAG. The DAG consists of two subgraphs and two root nodes, one of these nodes is used for calculating vertex positions (leftmost in the diagram), while the other is used for fragment color. Note that type checking ensures that for each directed edge from node A to node B, frequency of node A is always as high or higher than frequency of node B. Also, as constants are evaluated during inlining, the nodes with constant frequency are always the leaves of DAG. Frequency casts are not shown in this figure.

4.4.3 High-level optimizations

The most important optimization done after the inlining phase is the common subexpression elimination. This step reduces the number of nodes in the DAG generated during inlining.

Other optimizations done at this stage include simple algebraic simplifications, like replacing expression multiplications with units with expressions, *etc.*

4.4.4 Multiple pass generation

This pass is probably unique to shading compilers, thus the problem will be given more emphasis compared to the other code generation passes. Multiple pass generation is supported by only a few shading systems. For example, HLSL, Cg, Sh or glsl do not support this.

ARB fragment program implementations may support only 64 instructions, 16 temporary registers and 24 constant registers. Thus, it is very easy to create a high-level shader that exceeds those limits. Fortunately it is possible to virtualize hardware in this regard - by storing intermediate calculations in floating point frame buffers and textures when necessary. This problem resembles register allocation/spilling problem, but it has some specific features.

Thus, very often the DAG that has to be translated into a fragment program is too complex (contains too many nodes, requires too many constants or exceeds some other resource). We can solve this problem by simplifying the graph – by calculating some subgraphs separately and storing the results in textures, we can replace these subgraphs with texture read instructions.

An interesting feature of this problem is that graph partitions do not have to be disjoint – it is often reasonable to recalculate some subexpressions that are used in different passes. The reason is that creating a separate pass requires redrawing primitives to a separate frame buffer (*p-buffer*), which can be slower than performing same calculations several times.

Unfortunately, deciding whether the recalculation is cheaper than using a separate pass is not easy – it depends on hardware configuration, on the number of primitives sent to the shading system, complexity of the vertex programs and other factors. Thus, the only practical solution is to apply some kind of heuristic in the decision. Our compiler uses a simple cost model where each instruction is assigned a constant cost and a new pass generation is assigned larger cost. When decision has to be made between recalculation and creating a new pass, then the compiler chooses the solution with the smallest cost.

Our shading compiler uses Recursive Dominator Split (RDS) algorithm [6] and lower quality, but faster version called RDSh for splitting fragment programs into multiple passes. Neither algorithm is optimal (optimal solution is a NP-optimization problem). RDSh algorithm, which is used by default, requires $O(n \cdot \mathrm{g}(n))$ steps where $\mathrm{g}(n)$ is the cost of testing a pass with n-nodes. Currently $\mathrm{g}(n)$ evaluation requires fragment level code generation, optimization and register allocation – as a result RDSh algorithm typically requires $O(n^2 \cdot log(n))$ steps. RDSh algorithm can be also implemented as a post-optimization step (but before register allocation), in that case the complexity could be reduced to $O(n^2)$.

4.4.5 Fragment-level code generation

Fragment-level code generation is straightforward – most primitives correspond directly to hardware instructions. Code generation is a simple process – each instruction has its own code template, templates are appended by following data dependencies in the shader DAG.

Typically the output of code generation is highly non-optimal. A common issue is that generated instruction sequence contains lots of `MOV` instructions and scalar arithmetic instructions. Following low-level optimization steps are used to improve code quality:

- Dead code removal. This step removes instructions that produce results not consumed by other instructions (and that do not write to output registers like `result.color` or `result.depth`). This stage is useful as other steps may remove some instructions from the execution path.
- `MOV` instruction propagation forwards. This transformation tries to substitute instructions following a `MOV` instruction and using the `MOV` output with the `MOV` input. This transformation is not possible with all `MOV` instructions. For example:

  ```
  MOV r1.x,r0;
  ADD r2,r1,r3;
  ```

  In this case `ADD` instruction reads components of `r1` that are not assigned in the preceding `MOV` instruction, thus `r1` can not be substituted with `r0`.
- `MOV` instruction propagation backwards. This is similar to the forward propagation. Only the preceding instructions are considered – transformation tries to reassign output registers.
- `MUL` and `ADD` to `MAD` collapsing. This transformation tries to combine `MUL` and `ADD` instructions into a single `MAD` instruction.
- Instruction vectorizing. This optimization joins multiple instructions that write to same output register and use same input register into single vector instructions.

All steps are performed multiple times – process is stopped only when instruction count can not be reduced anymore. This is achieved typically within two to four optimization passes. Results are good, especially when considering the simplicity of the steps performed. Results are discussed in the next chapter.

The next step after the low-level optimization is the register allocation. Our compiler currently uses simple linear scan register allocation algorithm [26] for

assigning temporary registers, which provides adequate allocation. Vertex programs with less than a hundred instructions usually require ten or less temporary registers. Besides allocating temporary registers, this step also allocates constant registers and texture coordinates (in case of fragment programs).

4.4.6 Vertex-level code generation

As ARB fragment and vertex programming models are similar, our compiler utilizes same codepath for both. Some features are not available in vertex programs – like texture instructions, instruction `SAT` modifiers. For example, in case of `SAT` modifiers, instructions utilizing them are replaced with same instruction without `SAT` modifier and a `MIN` and `MAX` sequence.

Compared to the fragment code generation, vertex-level programs have another constraint – it is currently not possible to split them into multiple passes. At the time of writing, it is not possible to direct vertex program output to vertex arrays. If such functionality were available, vertex programs could be split into multiple passes like fragment programs. This would allow user to create vertex level expressions in the language with arbitrary complexity.

4.4.7 Primitive group code generation

Primitive group expressions are translated into special virtual machine instructions. Virtual machine uses register-file model, although register file is not limited – programs are free to use as many registers as needed. The number of instructions is also unlimited. Instruction set is designed with efficient matrix-matrix and matrix-vector operations in mind. For example, matrix multiplication and matrix inversion primitives are included in the instruction set.

Primitive group code generation is straightforward, and is based on instruction templates, like fragment and vertex code generation. But unlike fragment and vertex programs, optimizations performed after the code generation are much simpler – only simple copy propagation and dead code removal steps are used. Register allocation after the optimization is based on linear scan register allocation.

4.5 Runtime

The runtime is layered on top of the OpenGL API. It gives the application access to a higher-level API for handling shader objects, compiling shaders, binding parameters and rendering primitives. As shown in figure 14, the runtime API does not hide the OpenGL API, and, in fact, the user is expected to mix OpenGL calls with shading system calls. It is not reasonable nor practical to hide OpenGL completely from the user, as OpenGL is evolving and the user may need some OpenGL extensions not provided by the shading system.

Unfortunately, this organization has its limitations too. If the set of extensions supported by implementation is explicitly stated, it is possible to define exact rules which OpenGL calls are available to the application and which calls are reserved. The runtime does not restore the state of extensions (like `ARB_fragment_program`) after API calls, this may generate unintended results if the application uses this extension explicitly. Likewise, some future extensions may have priority over ARB fragment and vertex programming extensions, thus if such extensions are enabled, the runtime may not operate correctly at all.

4.5.1 API

The runtime API consists of around 20 constants and 60 procedures (see appendix C). API calls are used for setting compiler flags, loading shaders, compiling shaders, setting shader parameters, binding lights to surfaces and few other tasks. The runtime provides a simple immediate mode API where the user must feed every single vertex manually (which is relatively slow) and a high-performance vertex array object API. Unlike the Stanford API, which is modelled closely after OpenGL vertex arrays, the discussed system provides a higher-level API hiding more details from the user. This approach was taken because of two main reasons: as experimental shading system provides more powerful type system, the intention was to hide type encodings from user. Also, by providing higher-level API, it is possible to utilize more modern extensions like vertex array object extensions to provide higher geometry throughput by allowing geometry data to be held in the OpenGL server memory (uploaded into local graphics memory). The drawback of this approach is that it is inefficient for dynamic geometry, when geometry data sets are generated at runtime for each frame.

4.5.2 Shader states

The first user calls made to shader API after initialization are typically `LoadProgram` and `LoadShader` calls. `LoadProgram` creates a new global program object, while `LoadShader` call creates a new shader object under current program object. After loading a program or a shader, all input parameters can be set (via `Parameter*` calls), other shaders (which must be in (pre-)compiled) can be attached to the parameters via `UseShader` calls. Shaders are in the *uncompiled state* at this point. For surface shaders, deformation shader and light shaders can be attached at this point (via `UseDeformation` and `UseLight` calls).

Shaders can be compiled by calling either `CompileShader` or `PrecompileShader` API procedures. Both procedures are identical for non-surface shaders. In case of surface shaders, `CompileShader` generates fragment, vertex and primitive group programs and the shader can be used in the subsequent drawing calls. `PrecompileShader` only performs type-checking and inlining and leaves surface shader in *precompiled state*. Shaders in that

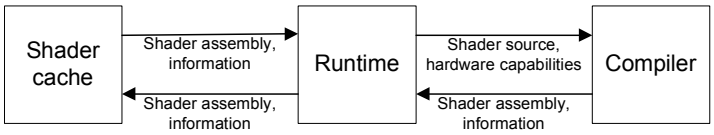


Figure 17: Runtime shader caching.

state can not be used in the drawing calls, but can be attached to input parameters of other shaders. After the shader has been compiled or precompiled, constant frequency parameters can not be changed. All `Parameter*` calls to such parameters result in a runtime error.

Another state is the *drawing state*, which shaders enter by issuing `Begin` calls. Like in the OpenGL API, only a subset of API calls are available in this state. `Parameter*` calls are available, but only for vertex frequency parameters. After calling `End`, shader will be back in compiled state.

4.5.3 Hardware limits, custom configuration

The shader API provides `Properties` procedure for overriding default configuration values and for providing custom parameters for code generation. When omitted, queries about hardware limits are performed automatically by the runtime. This way, the runtime tries to help the compiler to create the best performing code for current target. By overriding hardware limits, the user can check how the shader performs on a less capable target.

`Properties` call can be used to define preprocessor tokens, including token described in the compiler section earlier. When the user can assure that all modelview matrices do not change vertex w-coordinate and that all vertices sent through API have $w = 1$, then by calling `sglProperties("PreludeDefs: \"__HINT_PROJECTED\"")` (using C API), the compiler can eliminate some instructions from the vertex program.

4.5.4 Shader caching

The runtime utilizes shader caching subsystem, shown in figure 17. This cache is used to reduce traffic between the runtime and compiler parts, and to speed up applications that create a large number of shader objects. The runtime API provides two procedures for cache management: `LoadCache` and `SaveCache`. The first procedure is used to load previously stored cache state from a file, while the second is used to store the cache state to a file. When the application does not create shaders dynamically, its shader cache can be stored in a file and the compiler part is not needed afterwards. If the application must work on different targets, then the cache file can be created for each such target. Using a single target can be suboptimal in cases when shaders are split into multiple passes.

4.6 Issues

This section describes the issues that are specific to shading language implementations. Some of the issues are common to all shading languages (precision, semantics-preserving optimizations). Multipass issues are only important to the shading systems that try to virtualize hardware, by using multiple passes when necessary.

4.6.1 Multipass shaders

Due to the relatively low ARB programming extension instruction and register limits, hardware virtualization is essential for complex shaders. Splitting shaders into multiple passes is relatively straightforward using RDS or RDSh algorithms, unfortunately complications exist in the runtime part. First, in order for multipass rendering to work, the runtime must be able to render same primitives in each separate pass – which means the user-specified data must be cached so that the runtime can redraw primitives as many times as needed.

Currently the only option to store the results of temporary calculations is via offscreen *p-buffers*. Support for floating-point p-buffers is provided only via vendor-specific extensions at the time of writing, and requires separate *OpenGL context*. This means that OpenGL state that affects pipeline stages occuring before per-fragment operations must be cloned for the new contexts. Currently, the runtime clones common state variables for p-buffer rendering contexts: this includes *culling mode* and *automatic normal calculation* [19].

ARB programming extensions provide two sets of limits – so-called native limits and absolute limits. When the native limits are exceeded, the program may be accepted, but it will not run in hardware. This may result in slightly different results, which in case of vertex programs may lead to generation of different fragments compared to hardware implementation. This can be a potential problem for multipass shaders if some passes exceed the limits and others do not (some pixels may be touched by fewer passes). This can produce visual artifacts.

There is also one fundamental problem with multipass rendering, which can not be solved without sacrificing performance. The problem occurs when primitive batches (triangle strips, for example) contain overlapping primitives (triangles). In this case some pixels are rendered twice and can be overwritten depending on several conditions like depth test, stencil test or alpha test outcome. In case of alpha test, the alpha component of shader result is used to determine whether the final color of pixel will be overwritten or not. But for multipass shaders, this alpha value has not yet been calculated – it will be available only after the last pass. Solution exists: such batches must be split into non-overlapping subbatches and rendered separately. But this is not practical for real-time system (batches can contain thousands of primitives, sorting them requires at least $O(n \cdot \log(n))$ steps). For this reason, solving such pathological

cases is left for the user. Fortunately, if a shader does not use alpha testing and does not generate custom depth value as a shader result, simple depth testing works with multipass shaders. A similar problem also exists with `Cprev` variable – when primitives overlap within a single rendering batch (and the shader uses `Cprev` variable) then visual artifacts are very likely. A special data structure called *F-buffer* [17] has been proposed. It solves the overlapping primitive problem by assigning a unique location to each rasterized fragment (not pixel, as in p-buffer), while subsequent passes can the use F-buffer results as textures. Hardware-based F-buffer support is not currently available via extensions.

4.6.2 Floating point model

One of the largest issues with ARB vertex programming extension is the omission of `CMP` instruction (it is provided for fragment programs). The compiler uses a sequence of `ADD, MUL` and `MAD` instructions to emulate this instruction: assuming condition `c` is encoded as 0 or 1 and we must choose between expressions `e1` and `e2`, then we can calculate this using $c * e1 + (1 - c) * e2$ (here $*$ denotes floating-point multiplication). Unfortunately, when either `e1` or `e2` is $+\infty$, $-\infty$ or NaN, then at least by IEEE floating point model, the result should be NaN [12]. Early hardware implementations forced $0 * x = 0$ for all x values [16], but `ARB_vertex_program` extension does not require this if x is NaN or $\pm\infty$.

Another issue with the shader floating point model is that vertex and fragment level computation precision is not guaranteed. For constant and primitive group frequency computations `float` type corresponds to double-precision floating point number, but vertex and fragment level `float` precision is hardware dependent. Current hardware generation uses single precision, 32-bit floating point model for all vertex-level operations. For fragment-level operations, ATI Radeon hardware currently uses a 24-bit floating point model (floating point values are encoded with 7-bit exponent and 16-bit mantissa) while NVIDIA GeForce FX series implements a full single precision floating point model.

As results of floating point calculations are not rigidly defined for ARB programming models, more agressive optimizations (compared to strict compilers) can be justified in shading compilers. For example, collapsing a `MUL` and a `ADD` instructions into a `MAD` should be valid even when results are slightly different.

4.6.3 Extensions

At the time of writing, OpenGL core functionality does not present all the features required for implementing the shading system discussed here. As a result, implementation is dependent on multivendor OpenGL extensions (like `GL_ARB_vertex_program`), vendor-specific extensions (like `GL_NV_float_buffer` and `GL_ATI_texture_float` for floating point textures) and windowing-system extensions like `WGL_ARB_pbuffer`. The runtime uses more than 15 different ex-

tensions, most of which are optional (if unavailable, the shading system can still run without them, although with reduced performance).

Perhaps the largest problem with OpenGL is the lack of an extension that would allow to send vertex program outputs into vertex array objects (it is possible to emulate this by transferring data through CPU, but this would be likely even slower than a pure software implementation of the vertex programming extension). Similar support is fortunately available for fragment programs via textures and p-buffers and it allows to support shaders that do not fit into a single hardware pass. Unfortunately, vertex programs can not be split into multiple passes when resource limits are exceeded - this can result in a code generation failure reported by the shading compiler.

There is also another problem with the first generation DirectX 9 hardware – the ARB fragment programming extension is supported, but its specification is not fully followed by all vendors. The problem exists with Radeon 9700 GPUs – not all instructions take an equal number of instruction slots. For example, even though hardware reports that 64 ALU instructions are supported, in reality the number of instructions available can be a lot less when `LIT` instructions are used. As a result, the compiler must know about internal architectures of GPUs and be conservative when dividing shaders into passes, as it should never generate fragment programs that fail to execute due to limited resources.

5 Performance and Applications of Shading System

5.1 Performance Compared to Fixed Function OpenGL Pipeline

In this section we implement a large subset of the OpenGL lighting model within our shading language. We have two goals: first, to show that it is easy to implement a similar lighting model using the shading language. Our second goal is to evaluate the performance of such implementation (and of the shading system in general).

OpenGL provides a unified lighting model allowing to represent directional lights, point lights and spotlights within a single framework. To give a compact representation, we use $*$ operator to denote componentwise vector multiplication and $\odot$ operator, which is defined as $\mathbf{u} \odot \mathbf{v} = \max\{\mathrm{dot}(\mathbf{u}, \mathbf{v}), 0\}$.

We use a lighting equation where all lighting components are combined into the primary color $\mathbf{c}_{pri}$, while the secondary color $\mathbf{c}_{sec}$ is always zero:

$$\begin{aligned} \mathbf{c}_{pri} &= \mathbf{e}_{cm} \\ &+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\ &+ \sum_{i=0}^{n-1} (att_i)(spot_i)[(\mathbf{amb}_i) + (\mathbf{diff}_i) + (f_i)(\mathbf{spec}_i)] \qquad (18) \\ \mathbf{c}_{sec} &= 0, \qquad (19) \end{aligned}$$

where

$$\begin{aligned} \mathbf{amb}_i &= \mathbf{a}_{cm} * \mathbf{a}_{cli} \\ \mathbf{diff}_i &= (\mathbf{n} \odot \mathbf{l}_i)\mathbf{d}_{cm} * \mathbf{d}_{cli} \\ \mathbf{spec}_i &= (\mathbf{n} \odot \mathbf{h}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}. \end{aligned} \qquad (20)$$

In these formulas $\mathbf{e}_{cm}$ describes material emittance, while $\mathbf{a}_{cm}$, $\mathbf{d}_{cm}$ and $\mathbf{s}_{cm}$ describe material ambient, diffuse and specular reflectance (all are RGBA colors). Similarly, $\mathbf{a}_{cli}$, $\mathbf{d}_{cli}$ and $\mathbf{s}_{cli}$ are light i ambient, diffuse and specular intensity vectors (colors). $\mathbf{a}_{cs}$ is the global ambient intensity vector, which is applied to all surfaces. s_{rm} is the specular exponent and corresponds to the sharpness of the viewer-dependent highlight. Vector $\mathbf{h}_i$ (half-angle vector) is defined as $\frac{\mathbf{l}_i+\mathbf{v}}{|\mathbf{l}_i+\mathbf{v}|}$. Other vectors are shown in figure 6 – all geometric vectors are assumed to be normalized (having unit length).

The attenuation coefficient att_i, spotlight coefficient $spot_i$ and specular coefficient f_i (the term is used to cancel specular component when the angle between light vector and normal is greater than 90 degrees) are defined as follows:

$$\begin{aligned} att_i &= \begin{cases} \frac{1}{k_{0i}+k_{1i}d+k_{2i}d^2}, & \mathbf{P}_{pli}(w) \neq 0 \\ 1.0, & \mathbf{P}_{pli}(w) = 0 \end{cases} \\ spot_i &= \begin{cases} (\mathbf{e} \odot \mathbf{s}_{dli})^{s_{rli}}, & c_{rli} \neq 180, \mathbf{e} \odot \mathbf{s}_{dli} \geq \cos(c_{rli}) \\ 0.0, & c_{rli} \neq 180, \mathbf{e} \odot \mathbf{s}_{dli} < \cos(c_{rli}) \\ 1.0, & c_{rli} = 180 \end{cases} \\ f_i &= \text{sign}(\mathbf{n} \odot \mathbf{l}_i) \end{aligned} \tag{21}$$

Here k_{0i}, k_{1i}, k_{2i} are user-defined attenuation coefficients for each light, $\mathbf{e}$ (the eye vector) is defined as $\mathbf{e} = -\mathbf{v}$. $\mathbf{s}_{dli}$ is the spotlight direction vector, it is used when c_{rli} (the spotlight cone angle) is not equal to 180 degrees.

5.1.1 Implementation

In the shading language implementation we declare all material color and light intensity parameters as having primitive group frequency. Another option is to use constant frequency for most parameters – this would provide more optimization opportunities for the compiler in cases when some of the parameters are zero. This could give shading system version a performance edge (assuming the OpenGL implementation does not have such optimizations).

Lighting model parameters are generated randomly and we use same parameter values for both the shading system and the OpenGL implementation. Visually, the results of both implementations are indistinguishable.

We classify lights as directional lights, point lights, spotlights and use separate shader for each light type. Although the OpenGL lighting equation is presented as a single unified formula, in practice implementations classify lights in a similar way – this can be seen from the performance results presented below. Following graphic cards and drivers were used: ATI Radeon 9700 PRO (Catalyst 4.3), NVIDIA GeForce FX 5200 (ForceWare 52.43). These graphics cards represent the high-end and low-end of first generation DirectX 9 hardware. All tests were run under Windows 2000 SP2, using 2GHz Athlon XP processor.

5.1.2 Surface shader

Before we can write a surface shader for OpenGL lighting model, we must define a type that can capture output of all light classes and enables high performance implementation. A straightforward approach is to define a compound structure for all light-specific parameters that are common to all light classes:

```
typedef struct {
  float4 a_cl;
  float4 d_cl;
  float4 s_cl;
  float3 dir;
  float  att;
  float  spot;
} light_t;
```

Using this type we can translate our simplified OpenGL lighting model into shading language (for better readability, variable names correspond directly to the parameters given in the lighting equation above):

```
surface shader float4 gl_surface(
  float4 e_cm, float4 a_cm, float4 d_cm, float4 s_cm, float s_rm
)
{
  float4 c_pri =
    e_cm +
    a_cm * Ca +
    integrate(light_t l) {
      float3 h = normalize(l.dir - E);
      float3 v = lit(dot(N, l.dir), dot(N, h), s_rm);
      float4 c =
        a_cm * l.a_cl +
        v[1] * (d_cm * l.d_cl) +
        v[2] * (s_cm * l.s_cl);
      return l.spot * l.att * c;
    };
  return c_pri;
}
```

Here, `lit` primitive is used to calculate both diffuse and specular terms. This primitive takes three arguments: the dot product $\mathbf{n} \odot \mathbf{l}_i$, dot product $\mathbf{n} \odot \mathbf{h}_i$ and s_{rm}. It returns a three-component vector $\mathbf{v}$, where $\mathbf{v}[0] = 1$, $\mathbf{v}[1] = \mathbf{n} \odot \mathbf{l}_i$ and $\mathbf{v}[2] = f_i * (\mathbf{n} \odot \mathbf{h}_i)^{s_{rm}}$ (here f_i is defined as $\text{sign}(\mathbf{n} \odot \mathbf{l}_i)$). This primitive maps directly into `ARB_vertex_program` `LIT` instruction and is designed for efficient implementation of OpenGL standard lighting model.

5.1.3 Light shaders

Directional lights are specified by the light direction and light intensity parameters. In OpenGL such lights are given position $(x, y, z, 0)$ in homogenous coordinates – thus such lights are assumed to be infinitely far away. As a result, only constant attenuation term is required, as distance related attenuation coefficients are always zero. When only directional lights are used, the OpenGL lighting equation presented above can be simplified:

$$\begin{aligned}\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\ &+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\ &+ \sum_{i=0}^{n-1} \mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \mathbf{l}_{dir})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \mathbf{h})^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli} \end{aligned} \quad (22)$$

Implementation of directional light is very straightforward (light-specific variable `L_pos` holds the light position in homogenous coordinates):

```
light shader light_t dir_light(
  float4 a_cl, float4 d_cl, float4 s_cl
)
{
  float3 dir = normalize(direction(L_pos));
  return struct {
    a_cl = a_cl, d_cl = d_cl, s_cl = s_cl, dir = dir, att = 1, spot = 1
  };
}
```

We do not present point light shader, which has complexity between the directional light shader and the spotlight shader. Spotlight is the most complex of the given light classes and its implementation is given below:

```
light shader light_t spot_light(
  float4 a_cl, float4 d_cl, float4 s_cl, float3 k, float c_rl, float s_rl
)
{
  float3 dir = direction(L_pos - P);
  float3 vec = -normalize(dir);
  float  att = rcp(dot(direction(dist(dir)), k));
  bool   cone = dot(vec, L_dir) < cos(c_rl);
  float  spot = cone ? 0 : pow((clampf) dot(vec, L_dir), s_rl);
  return struct {
    a_cl = a_cl, d_cl = d_cl, s_cl = s_cl,
    dir = normalize(dir), att = att, spot = spot
  };
}
```

Here `L_dir` holds the normalized spotlight direction vector (and is equal to the $\mathbf{s}_{dli}$ vector in the OpenGL lighting equation). `dist` primitive is used to calculate the distance vector $(1, |\mathbf{v}|, |\mathbf{v}|^2)$ for a vector $\mathbf{v}$.

5.1.4 Testing procedure

For performance measurement, we use a single sphere tessellated into 1,000,000 triangles. The triangles are rendered using a single triangle strip call in a 512x512

window. Although we do not measure "pure"transformation and lighting performance (as rendering involves other steps like triangle setup and rasterization), transformation and lighting with setup stage are clearly limiting the throughput of the GL pipeline as increasing the window resolution to 1024x1024 gives almost no performance penalty.

Most OpenGL lighting parameters are chosen in a way that they can not be optimized away. For example, light ambient, diffuse, specular intensities are set to non-zero values so that they affect the final result. All attenuation factors are set to random positive values for point lights and spotlights, while only constant attenuation term (which is set to 1) is used for directional lights. This should represent the common way how lights are used. The viewer is assumed to be at $(0, 0, 0)$ in eye coordinates (OpenGL also allows the viewer to be placed at $(0, 0, \infty)$, which simplifies per-vertex calculations).

In order not to limit GPU by the CPU-GPU interface constraints (limited bandwidth), all geometry data (vertices, normals) is uploaded to the graphics processor local memory. Subsequent rendering calls utilize vertex buffer objects (via `ARB_vertex_buffer_object` extension). This extension is used for both the shading language implementation and the OpenGL implementation.

Note that the shading system tests performed are a measurement of generated code quality. As the number of rendering commands sent to the GPU is small in our tests, compared to the work that GPU itself has to perform, vertex and fragment program quality is essential for achieving good results. When tests with high number of simple rendering commands were performed, it is likely that the shading system would perform much slower compared to OpenGL – as the amount of work done by the shading system would be proportionally much larger.

5.1.5 Performance

We perform six tests for each light type – by varying the number of lights. Results are listed in table 10. For each test we give performance data (number of vertices or triangles per second) when lighting model is implemented in the shading language. We also list the number of vertex instructions generated for each test. Generated fragment program consists of a single instruction for all these tests: `MOV result.color,fragment.texcoord[0]` – vertex-level shading result (stored in texture coordinate set 0) is simply assigned to the final fragment color.

Results for pure vertex transformation performance are not given in table 10. In case of pure transformation, vertices are only transformed and no lighting calculations are done. This requires four `DP4` instructions in the vertex program. Performance is almost identical in the OpenGL and shading language implementations – Radeon 9700 PRO GPU can transform 140 million vertices per second.

Looking at the results, we can see that the OpenGL implementation beats the shading language in most cases on Radeon 9700 PRO. But the gap is quite small – the largest difference is in the test with 8 directional lights – the fixed

function version works 19% faster. In point light tests, our shader version is faster in half of the tests, while in spotlight tests, the shader version is faster in two tests. Strangely, shader version is most competitive in cases of 1 and 8 lights. Note that in the test with 8 spotlights, the shading compiler version is over two times faster than the fixed function version! The OpenGL result for this test is remarkably low compared to the OpenGL result with 6 spotlights. Thus, it is possible that software solution is used by OpenGL drivers in this case.

It is hard to give full analysis of the performance – in case of simple shaders, generated shader code seems optimal (a careful inspection did not reveal any misoptimizations or possible improvements). The likely reasons for the performance delta are following (OpenGL implementations are proprietary, thus the following can be considered only speculation):

- Hardware may include additional instructions that are not exposed in the ARB vertex programming extension. Likewise, some instruction modifiers may be supported (like `SAT` modifier, which is present in the ARB fragment programming extension).
- Our shading language compiler makes inaccurate assumptions about lower level microarchitectural details. The current shading compiler makes a rather simplistic assumption that all instructions have latency and throughput of 1 instruction/cycle and that all instructions are atomic (not split into multiple instructions). Also, the compiler does not optimize for superscalar vertex processors where issuing multiple instructions per clock is possible if certain conditions are met.

Note that the second point above is satisfied in case of Radeon 9700 PRO GPU – it includes four symmatrical vertex processing units, each of which consists of a vector ALU (capable of executing 4D vector operations) and a scalar unit. Both units work in parallel.

5.1.6 Implementing lighting model in fragment level

It is very straightforward to implement the described lighting model using per-fragment calculations. This can improve visual quality in areas where lighting function changes very rapidly – such as spotlight border areas where $spot_i$ (see above) is discontinous.

There are very few necessary changes to be made to the previously described surface and light shaders: per-vertex variables like `P`, `N` and `E` must be replaced with their per-fragment counterparts. For example, `N` could be replaced with `normalize((fragment float3) N)`. Resulting shaders could be further optimized, for example, $spot_i$ calculation could be replaced with a texture lookup, but for sake of simplicity, such optimizations are omitted.

1 light			
Characteristic	Dir. light	Point light	Spotlight
Vertices/sec (GL)	42.0M	32.3M	30.2M
Vertices/sec (SH)	41.9M	32.6M	31.3M
VP instructions	26	37	43
2 lights			
Characteristic	Dir. light	Point light	Spotlight
Vertices/sec (GL)	33.9M	22.3M	19.5M
Vertices/sec (SH)	32.6M	24.5M	19.3M
VP instructions	38	57	69
3 lights			
Characteristic	Dir. light	Point light	Spotlight
Vertices/sec (GL)	28.2M	18.4M	14.6M
Vertices/sec (SH)	25.0M	17.0M	13.7M
VP instructions	49	76	94
4 lights			
Characteristic	Dir. light	Point light	Spotlight
Vertices/sec (GL)	23.2M	14.4M	11.6M
Vertices/sec (SH)	21.1M	13.6M	10.5M
VP instructions	60	95	119
6 lights			
Characteristic	Dir. light	Point light	Spotlight
Vertices/sec (GL)	17.7M	10.5M	8.2M
Vertices/sec (SH)	15.3M	10.1M	7.7M
VP instructions	82	133	169
8 lights			
Characteristic	Dir. light	Point light	Spotlight
Vertices/sec (GL)	14.8M	6.9M	2.9M
Vertices/sec (SH)	12.0M	7.3M	6.5M
VP instructions	104	171	194

Table 10: Performance of the OpenGL lighting model implementations (Radeon 9700 PRO GPU). GL denotes the OpenGL implementation, SH denotes the shading language implementation. We also list the number of vertex program instructions generated by the shading compiler for each test.

All tests are performed using same hardware and software configuration as the vertex-level tests. The sphere model is simplified – it consists of 10,000 polygons (instead of 1,000,000 as in case of vertex level tests). This should move the bottleneck from vertex programs to fragment programs. Backface culling is also used, thus only visible fragments are rendered (polygons and fragment not facing towards the viewer are ignored). Viewport size of 1024x1024 is used for testing. Despite these differences, the output of fragment-level lighting model is visually very close to the vertex-level lighting model (despite the fragment-level test case containing 100 times less polygons).

Results are shown in table 11. Besides performance data, this table also lists the number of generated fragment and vertex program instructions. In cases when multiple point or spotlights are used, shaders must be split into several passes as instruction counts in fragment programs exceed the fragment program limits on Radeon 9700 GPU. In such cases, instruction counts are given for each pass separately.

Few conclusions can be made by looking at the results. First, performance drops considerably when a shader must be split into multiple passes. For example, the test with 3 directional lights has rougly two times higher performance than the test with 4 directional lights – the latter requires two passes instead of one. In case of multiple passes, the number of passes largely determines the shader throughput. The complexity of individual fragment or vertex programs of these passes has secondary effect.

By looking how the shaders are split into multiple passes, two interesting details are revealed. First, the shaders in spotlight tests are almost perfectly split into multiple passes – instruction counts in individual passes are almost equal and each light source requires a single additional pass. But point lights are split in relatively strange way – in test cases with 2, 4, 6 and 8 lights, first passes consist of only six instructions. At the same time, instruction sequences in final passes contain over 40 instructions.

Another point to notice is that point light tests with 6 and 8 light sources are actually slower than the tests with same number of spotlights. The reason is that these point light tests require 5 and 8 textures for temporary storage. Tests were performed with a viewport of size 1024x1024, each such texture requires 16MB of space. When we include the size of the offscreen p-buffer and additional onscreen and depth buffers, then all these buffers require more than 128MB, which is the amount of local memory on our Radeon 9700 testing board. The shading system allows the user to manually reduce the precision of temporary textures (by default, textures with 32-bit floating point precision are used). When we force 16-bit floating point precision, then tests with 6 and 8 point lights can produce 8.9 and 6.6 million fragments per second, respectively. This is the expected result and is higher than the results of spotlight tests (even when 16-bit precision is forced for these tests).

For comparison, we have listed results for GeForce FX 5200 GPU in table

11. Compared to Radeon 9700 GPU, GeForce FX 5200 has much weaker implementation of fragment and vertex programs, but it supports fragment programs with up to 1024 instructions. Thus all fragment programs generated in our tests fit into a single pass. Performance results are much lower, even in cases when Radeon 9700 GPU requires multiple passes.

In case of single pass shaders, Radeon 9700 GPU is roughly 8-10 times faster. This is reasonable, when we take into account GPU clock frequencies (325MHz in case of Radeon 9700 PRO GPU versus 250MHz in case of GeForceFX 5200) and the number of parallel rendering pipelines (8 in case of Radeon 9700 PRO and 1 in case of GeForceFX 5200) for both GPUs. Even when shaders are split into multiple passes, Radeon 9700 GPU is still approximately 5 times faster. Two results are unexpected – in tests with 6 and 8 lights, the results of point light tests are lower compared to the spotlight tests (although instruction counts are lower for point light tests). It is likely that the code generated for point light tests is not optimally scheduled for the GeForce FX hardware and the hardware fails to operate with its full potential.

5.2 Ray-Tracing a Simple Scene

OpenGL rendering model uses polygons as primitives. Higher-order surfaces like NURBS patches or quadrics must be decomposed into triangles (polygons) before rendering. Nevertheless, the power of programmable shading units of todays' GPUs make it possible to emulate other rendering techniques like ray-tracing [27] or radiosity calculations. In this section we show that ray-tracing a simple scene can be practical and very efficient while using a high-level shading language.

5.2.1 Ray and sphere intersection using high-level shading language

As an example, let's use a simple scene containing a single sphere with radius R and origin at $(0, 0, 0)$. Our goal is to calculate the intersection point $\mathbf{x}$ of a ray originating from a point $\mathbf{p}$, intersecting with the screen grid at a given fragment. We can formulate this using two equations:

$$\mathbf{x} = \mathbf{p} + \mathbf{d}t$$
$$R^2 = \mathrm{dot}(\mathbf{x}, \mathbf{x})$$

Here $\mathbf{d}$ is direction vector from the ray origin to the current fragment in the screen grid and t is the distance from the origin to the intersection point (also unknown). We can find t by solving the following quadratic equation:

$$at^2 + 2bt + c = 0$$

1 light			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	88.7M	66.0M	52.6M
FP instructions	16	24	30
VP instructions	10	16	16
2 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	57.2M	24.0M	20.0M
FP instructions	28	6,41	30,32
VP instructions	10	10,21	16,21
3 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	42.3M	14.7M	12.5M
FP instructions	39	6,40,26	30,31,32
VP instructions	10	10,21,21	16,21,21
4 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	21.4M	10.7M	9.1M
FP instructions	15,41	6,6,40,42	30,31,31,32
VP instructions	10,15	10,10,21,21	16,21,21,21
6 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	11.0M	4.4M	5.9M
FP instructions	15,15,15,45	6,6,40,6,41,42	30,31,31,31,31,32
VP instructions	10,10,10,15	10,10,21,10,21,21	16,21,21,21,21,21
8 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	7.4M	2.7M	4.3M
FP instructions	15,15,15,15,46,18	6,6,40,6,41,6,41,42	30,31,31,31,31,31,31,32
VP instructions	10,10,10,10,15,15	10,10,21,10,21,10,21,21	16,21,21,21,21,21,21,21

Table 11: Performance of the OpenGL lighting model implemented at fragment level (Radeon 9700 PRO GPU). Tests are performed using a scene consisting of 10,000 triangles. Besides performance, we list also the number of fragment and vertex program instructions generated for each test (when shaders are split into multiple passes then instruction counts are given for pass separately).

1 light			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	9.4M	7.8M	6.7M
FP instructions	16	24	30
VP instructions	10	16	16
2 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	6.7M	4.3M	3.5M
FP instructions	28	44	56
VP instructions	10	16	16
3 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	4.7M	3.1M	2.5M
FP instructions	39	63	81
VP instructions	10	16	16
4 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	3.7M	2.3M	1.9M
FP instructions	50	82	106
VP instructions	10	16	16
6 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	2.7M	1.3M	1.3M
FP instructions	72	120	156
VP instructions	10	16	16
8 lights			
Characteristic	Dir. light	Point light	Spotlight
Fragments/sec	2.1M	0.68M	0.81M
FP instructions	94	158	206
VP instructions	10	16	16

Table 12: Performance of OpenGL lighting model implemented at fragment level (GeForce FX 5200 GPU). Tests are performed using a scene consisting of 10,000 triangles. Besides performance, we list also number of fragment and vertex program instructions generated for each test.

where $a = \mathrm{dot}(\mathbf{d}, \mathbf{d})$, $b = \mathrm{dot}(\mathbf{d}, \mathbf{p})$ and $c = \mathrm{dot}(\mathbf{p}, \mathbf{p}) - R^2$.

Note that with this formulation, $\mathbf{x}$ can be also used as a normal vector. In general, there can be zero, one or two solutions to the equation – in case when the ray misses the sphere, there is no solution, while when the ray hits the sphere, there can be one or two intersection points. We must select intersection point with smaller t value – which is nearest to the viewer. At the same time, we must ignore negative t values – such points are not visible.

The ray direction vector $\mathbf{d}$ used above can be calculated from the fragment coordinates by applying inverse viewport and modelview transformations.

Listing at figure 18 shows a function written in our shading language that calculates the nearest intersection point of a ray for the current fragment and a sphere. Two things should be noted here: `__SCREENPOS()` is an internal function that returns the current fragment position in window coordinates (it returns a triple (x, y, w), the z-coordinate can be derived in case of perspective transformation from the w-coordinate as shown in the code). The conversion from vertex frequency to fragment frequency when calculating `ray_odir` is necessary, as interpolation of normalized vertex vectors does not yield normalized vectors in fragment frequency – thus vectors must be (re)normalized in fragment program.

Using the intersection routine, we can quickly combine the sphere with any shading model. The shader is executed by drawing a single quad covering whole viewport (by issuing four `Vertex*` calls). Figure 19 shows how the intersection code can be combined with a diffuse lighting model, bump mapping and simple texture mapping.

Output of the resulting shader is shown in figure 20. Table 13 lists characteristics of the compiled shader code and its performance (note that the performance of the shader does not depend on the size of the sphere – it depends rather on the size of the viewport). As can be seen from this table, most of the work is done within the fragment program – the fragment program consists of 31 instructions, 3 of which are texture instructions. Also, by multiplying the fragment program instruction count with the number of fragments processed per second, we get roughly 2.8 billions instructions per second. This number is higher than the product of Radeon 9700 PRO GPU frequency (325MHz) and the number of parallel rendering pipelines (8) – which means that each GPU fragment pipeline is capable of executing multiple instructions per clock.

As a side note, ray-traced spheres can be combined seamlessly with traditional polygon objects when we assign `depth` member in addition to `color` and `kill` members in the shader return statement. By providing a custom depth value for each sphere fragment, we can render polygonal objects and spheres in arbitrary order and get correct results where the objects are overlapping.

```
typedef struct {
  bool hit;
  float3 opos;
} intersection_t;

intersection_t intersect(float r)
{
  // World-space ray direction and position
  matrix4 inv_proj = invert(Mvport * Mproj);
  float4  ray_sdir = __SCREENPOS().xyzz * { 1, 1, -Mproj[2][2], 1 };
  float4  ray_wdir = inv_proj * ray_sdir;
  float4  ray_wpos = { 0, 0, 0, 1 };

  // Object-space ray direction and position
  matrix4 inv_view = invert(Mview);
  float3  ray_odir = normalize((fragment float3) direction(inv_view * ray_wdir));
  float3  ray_opos = direction(inv_view * ray_wpos);

  // Quadratic equation coefficients (note that a==1 as ray_odir is normalized)
  float b = dot(ray_odir, ray_opos);
  float c = dot(ray_opos, ray_opos);

  // Solve intersection equation. If d < 0, then ray misses sphere.
  float d  = b * b - (c - r * r);
  float t1 = -b + sqrt(d);
  float t2 = -b - sqrt(d);
  float tt = min(t1 > 0 ? t1 : t2, t2 > 0 ? t2 : t1);

  return struct {
    hit  = d > 0 && tt > 0,
    opos = ray_opos + ray_odir * tt
  };
}
```

Figure 18: Ray and sphere intersection calculation code.

Characteristic	**Value**
Fragments per second	90.5M
Frame rate (1024x768 window)	115 fps
Fragment program instruction count	31
Vertex program instruction count	27
Total prim. group instruction count	29

Table 13: Performance and characteristics of the sphere shader (on Radeon 9700 PRO).

```
typedef struct {
  float4 color;
  float3 direction;
} light_t;

surface shader fragment_t sphere(texref texmap, texref normalmap)
{
  // Calculate intersection info
  intersection_t is = intersect(5);

  // Simple Lambertian reflectance model
  float4 illum =
    integrate(light_t lt) {
      float3 n_o = direction(texCUBE(normalmap, is.opos)) * 2 - { 1, 1, 1 };
      float3 n_w = transpose(invert(affine(Mview))) * n_o;
      return (clampf) dot(lt.direction, n_w) * lt.color;
    } + Ca;

  // Find final color
  float4 color = illum * texCUBE(texmap, is.opos);
  return struct { color = color, kill = !is.hit };
}

light shader light_t directional(float4 color)
{
  return struct { color = color, direction = L };
}
```

Figure 19: Shaders for ray-traced sphere.

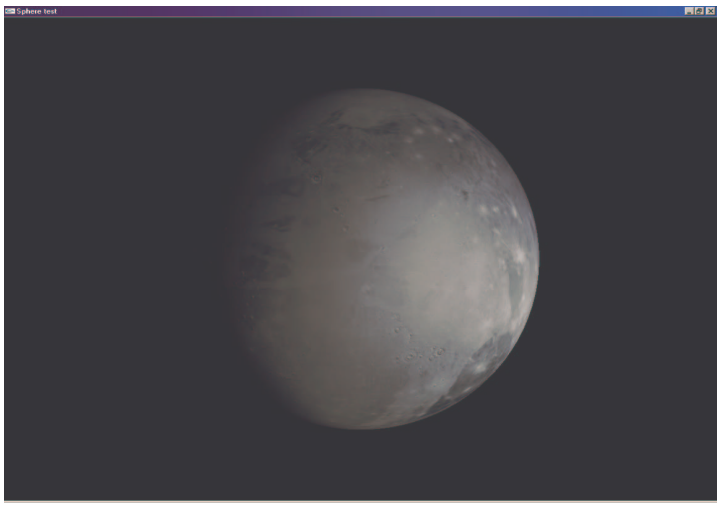


Figure 20: Screenshot of a ray-traced sphere.

5.3 Implementing High-Quality Scene Preview Modes for 3D Authoring Packages

In this section we discuss how the shading system features can be used to a render a large class of custom materials defined in the *Softimage|XSI* authoring package. Softimage|XSI is built around a flexible shading system of the *mental ray* rendering software [5]. Complex materials can be built from a set of simpler shading nodes which are connected together. This is similar to *shade tree* approach discussed in chapter three, but in mental ray the concept is generalized – shaders are divided into more classes.

The possibility to combine simple shaders to create complex effects gives artists very powerful control over object appearance. Unfortunately, Softimage|XSI 3.5 and other modelling packages let an artist preview their work only in a special *preview window*. Rendering in this preview window may take from several to tens of seconds when complex materials are used. At the same time, the default viewing mode based on OpenGL works in real time, but supports only a few simple nodes and basic texturing.

Mostly, due to the flexibility of fragment programs in modern GPUs, it is possible to provide higher quality real-time renderings of customized materials than in standard OpenGL. The shading system described in this thesis provides quite straightforward way to map materials with any complexity into graphics hardware. In the following subsections we will give a sketch of a general framework of the implementation. We will first give an overview of Softimage|XSI shading capabilities and then focus on shading system API and the compiler. Thus, this section provides a real-world application example of the shading system and brings out its benefits.

As the Softimage|XSI approach to materials and shading is more complex than provided by other packages like Lightwave or 3D Studio MAX, the discussed methods for implementing real-time preview modes are applicable to these packages also.

5.3.1 XSI shader classes

Figure 21 shows an example material shader graph in Softimage|XSI 3.5. This material is used to provide a "rough metallic"look of the head object in figures 22, 23.

The example consists of seven shading nodes that are connected together and form a single material. Two leftmost nodes are used for two textures: the upper texture is used to modulate surface colors, while the lower provides custom normal vectors for the head surface. These nodes connect to two image nodes, which associate textures with *texture spaces* (custom texture coordinate sets that are associated with the head object vertices). 'Phong' node is the central shading node in this graph – this node provides real shading calculations and uses light sources

associated with the head model. 'mul_color' node modulates the calculated Phong shading results with a custom lightmap. Last node, 'ospace_normalmap' transforms the normal vectors provided by 'NormalMap_image' into a format that is usable by shading nodes like 'Phong'.

Softimage|XSI materials provide several connection points – two of which ('Surface' and 'Bump map') are shown in the figure. But this is only a small subset of shader classes that can be used to customize material. The following list includes all available shader classes:

- *Surface shader* determines the basic color of an object as well as reflection, refraction and transparency.
- *Volume shader* modifies rays as they pass through an object or scene.
- *Environment shader* is used instead of a surface shader when light rays "escape"scene (do not hit any object).
- *Contour shader* is used to customize appearance of object edges.
- *Displacement shader* can move surface points geometrically and can be used to add roughness to objects.
- *Shadow shader* determines how the light is altered when the object obstructs light.
- *Photon and photon volume shaders* are used for global illumination effects.
- *Bump map shaders* are used to provide roughness illusion by perturbing the surface normal.

All these shaders can be combined into materials. Besides customizing materials, shaders can be used to customize lights and image postprocessing. Light shaders and postprocessing shaders are not discussed here, as they are not directly associated with material graphs.

Shader classes that can be efficiently implemented using our shading system include surface, environment and bump map shaders. Displacement shaders could be implemented efficiently on future hardware, when texturing support becomes available in vertex programs also. Volume shader support is also possible, but the shading system does not provide straightforward support for this.

5.3.2 Shader connection points

Note that directed edges of the shader graph in figure 21 have different colors. Edge colors are used to denote data types. Red color denotes RGBA color type, yellow denotes geometric 3D vector and blue denotes texture object. In addition, scalars, integers and booleans are also supported. Each shader connection point

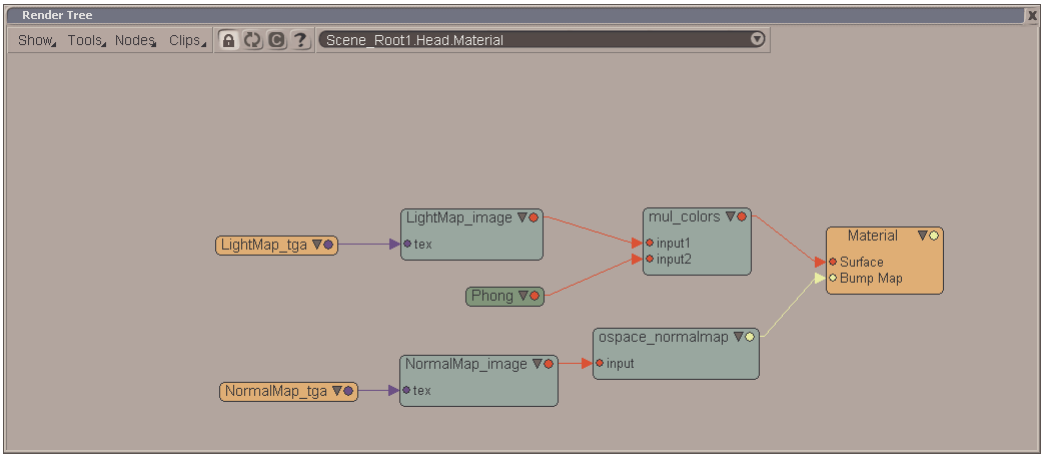


Figure 21: An example shader graph in Softimage|XSI environment

has specific type, likewise shader outputs a value of specific type. Only shaders with compatible types can be connected together.

Besides other shaders, constants and cubic splines can be connected to most shader connection points (such connection points are not shown in figure 21). For example, Phong shading node supports around 20 different parameters that describe diffuse, specular components of illumination model, reflection, refraction and transparency. In case of the head model, ambient, diffuse and specular parameters are used, while reflection, refraction and transparency effects are disabled.

5.3.3 Example

Figures 22, 23 and 24 give example of a simple scene, rendered using mental ray renderer, our shading language scene viewer and Softimage internal OpenGL renderer, respectively. First two images have very similar visual attributes and look much better than the third one. The greatest difference between the first two images is the omission of shadow behind the statue, while head material and spotlight area are visually very close. The third image has very poor quality at the spotlight area and head material looks drastically different.

Note that the first image takes roughly ten seconds to render at 1280x900 resolution, while the second image is renderered in real-time, 35 frames per second at same resolution on a Radeon 9700 PRO GPU. The missing shadow could be included in the real-time version (although with lower quality).

The scene consists of three objects (head, statue and floor), each with its own material, and three light sources (spotlights). The floor is affected only by one light source, while both the head and the statue are affected by all three spotlights. Table 14 gives details about compiled shaders. Performance is mostly

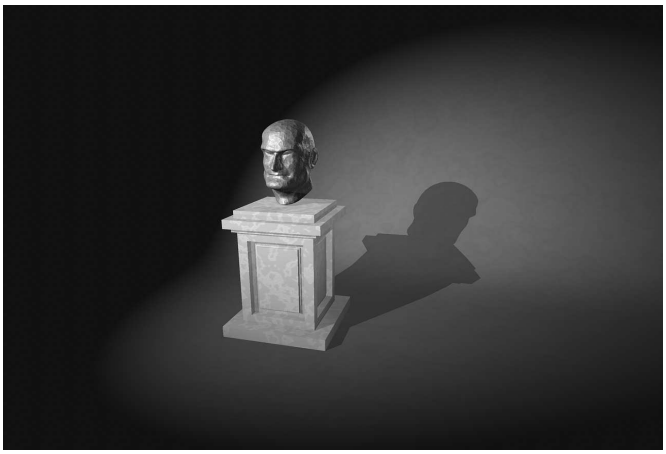


Figure 22: Test scene rendered using *mental ray* renderer.

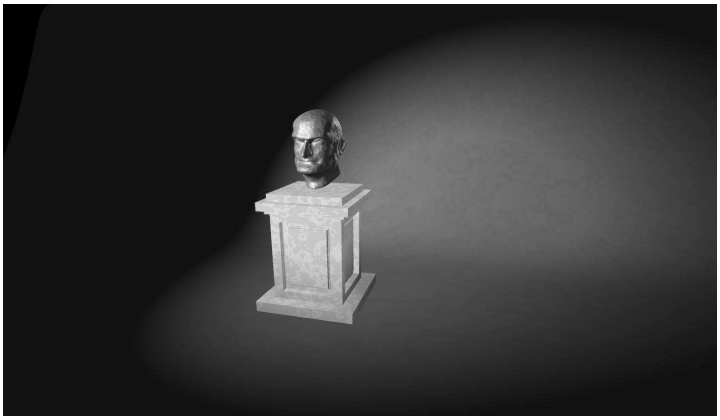


Figure 23: Test scene rendered using real-time shading system.

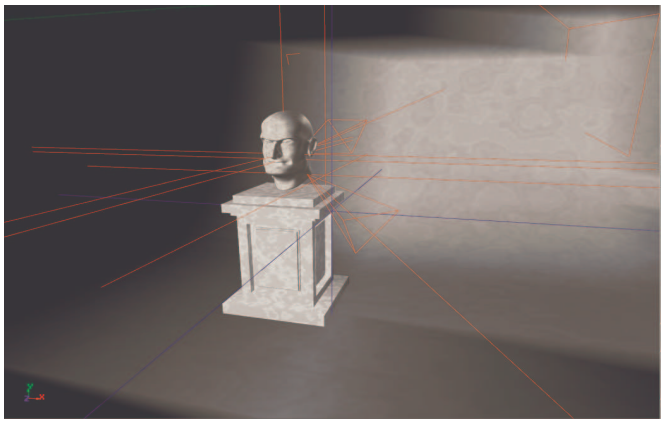


Figure 24: Softimage|XSI OpenGL preview mode of the scene.

Characteristic	Head	Statue	Floor
Passes	4	1	1
FP instructions	12,46,33,51	49	20
VP instructions	11,17,17,18	18	18

Table 14: Instruction counts in compiled shaders of example scene. FP denotes fragment program and VP vertex program.

limited by head shader which requires four rendering passes. Statue material, although affected by three spotlights, does not have any specular term contribution, thus its compiled shader is much simpler.

5.3.4 Translating XSI shader nodes

As an example, we describe how to automatically translate XSI surface shader graphs into our shading system. First, we must have the following shaders implemented in the shading language:

1. We must have basic pass-through shader with constant and primitive group input frequency for each type used in the XSI shader graphs. For example, we need the following shaders for RGBA color type:

```
surface shader float4 const_color(constant float4 input)
{
  return input;
}

surface shader float4 uniform_color(primitive group float4 input)
{
  return input;
}
```

2. Shader nodes should be implemented in the shading language. Mental ray supports dozens of shader nodes, but most scenes require a small subset only. As an example, 'mul_color' shader used in the example above can be implemented as follows:

```
surface shader float4 mul_color(float4 color1, float4 color2)
{
  return color1 * color2;
}
```

Translation scheme is straightforward. The following pseudo-C listing shows how this can be done (note that shader API calls are prefixed with 'sgl' string):

```
RealtimeShader translateShader(XSIShader xsi_shader)
{
  RealtimeShader shader = sglLoadShader(xsi_shader.name);
  foreach (XSIParam xsi_param in xsi_shaders)
  {
    RealtimeShader param_shader;
    switch (xsi_param.connection_type)
    {
    case Constant: param_shader = translateConst(xsi_param.constant); break;
    case FCurve:   param_shader = translateFCurve(xsi_param.fcurve);  break;
    case Shader:   param_shader = translateShader(xsi_param.shader);  break;
    }
    sglBindShader(shader);
    sglUseShader(sglParameterHandle(xsi_param.name), param_shader);
  }
  sglBindShader(shader);
  // ... call sglUseLight procedure to attach required light shaders
  sglPrecompileShader();
  return shader;
}
```

In this listing, we first the load real-time shader using `LoadShader` API procedure, then scan all parameters of the original shader. When a parameter has constant value or is connected to a cubic curve (the term *f-curve* is used in the code), then another translation procedure is called (look below). When the parameter is connected to another shader, then `translateShader` is called recursively. Parameter translation creates a new shader that is attached to the current shader using `UseShader` API procedure.

After all parameters have been translated, we can attach light sources to the surface using `UseLight` API procedure. After that, the surface shader can be precompiled. Note that the final surface shader, that is connected to the material directly, must be compiled using `CompileShader` procedure instead of `PrecompileShader`. This is not shown in this listing.

Translation of constants and cubic curves is simpler, we load shader corresponding to the given constant or cubic curve type and then set its only parameter:

```
RealtimeShader translateConst(XSIConst xsi_const)
{
  RealtimeShader shader;
  switch (xsi_const.type)
  {
  case Scalar:
    param_shader = sglLoadShader("const_scalar");
    sglParameter1f(sglParameterHandle("input"), xsi_const.value.scalar);
    break;
  // ... similar cases for other types: booleans, integers, vectors, colors
  }
```

```
  return shader;
}

RealtimeShader translateFCurve(XSIFCurve xsi_fcurve)
{
  // ... similar to translateConst, but uniform shaders must be used instead
  // ... uniform variables can be later updated for each frame as needed
}
```

Although this translation is straightforward, the generated shaders are very efficient, even when shader nodes themselves are complicated. For example, Phong node may have around 20 connection points – but usually only a few of them are used and the others are left to default state. In default state such parameters are usually zero and have no influence upon final shading result. The translation scheme uses constant frequency pass-through shaders (`const_color`, for example) for such parameters. As the values of these parameters are known to the compiler, our shading compiler can often remove such terms, thus creating special optimized versions of the shaders.

New shader nodes can be added to this system without changing the described translation code. This translation scheme can be applied to light shaders and deformation shaders also.

5.4 Future Work

Advancements in real-time shading systems are driven mostly by advancements of graphics hardware. So, to conclude this chapter, we give a short overview how GPUs have advanced during the last three years and discuss how these developments can be utilized in the presented shading system.

5.4.1 Advancements in hardware

When the work with the experimental shading system started, floating point model was not yet available for fragment programs. Table 15 gives a short overview how hardware support for shading has advanced in terms of performance and features. Note that besides higher performance and larger instruction count limits, new hardware provides dynamic branching and full floating point model that allows implementing shading models that were not even possible before.

Capabilities of vertex and fragment units are converging – floating point model is used in both, while limited texture support is starting to appear in vertex pipeline also. Unified model for vertex and fragment program could theoretically provide better load-balancing – central scheduler could use unified pool of vertex and fragment processors and allocate these to perform vertex and fragment calculations as needed.

Characteristic	GeForce 3	GeForce 6800U
Operating frequency	200MHz	400MHz
Pixel throughput (billions pixels/sec)	0.8	6.4
Max. fragment instructions	12	65536
Primary datatype used in fragment pipeline	9-bit FX	32-bit FP
Vertex throughput (millions vertices/sec)	50	600
Max. vertex instructions	128	65536
Primary datatype used in vertex pipeline	32-bit FP	32-bit FP
Introduction date	February 2001	April 2004

Table 15: Comparison of hardware advancements during the development of the experimental real-time shading system (FP means floating-point, while FX means fixed-point).

Up to this point, the goal of graphics hardware has been to provide movie-quality graphics in real-time. Although this is a moving target (quality of offline rendering is advancing constantly), the gap between offline rendering and real-time rendering is becoming smaller. So it is likely that graphics hardware will be utilized for offline rendering in the future to provide shorter rendering times.

5.4.2 Extending the shading system

Due to advancements of graphics hardware, several new features could be implemented in our shading system. First, general support for control-flow constructs has become possible. This includes simple `if`, `for` and `while` statements, but support for subroutines and recursion is still very limited – return address stack depth is limited. Another feature that could be implemented, is support for vertex textures. Vertex textures could be utilized as large look-up tables – this can speed up calculations of complex trigonometric functions. Besides, vertex textures can be used to displace geometry in deformation shaders.

Shading compiler could use glsl as a target language instead of the ARB fragment and vertex programming extensions. Similar approach is used by many high-level compilers, where C language is used as a portable target language (so existing C compilers could be utilized). This approach provides real benefits as the discussed shading language is higher-level than glsl – it divides shaders into three classes, provides specialized `integrate` construct (enabling the user to decouple surface and light descriptions) and allows the user to write shaders that are automatically translated into vertex and fragment programs. By utilizing glsl language as a target language, better code quality and performance could be achieved.

Another possible development would be integration of shading API with a *scene-graph* interface. With the scene-graph approach (where full description of

scene is available), shading system could be more efficient and provide support for other shader classes like atmosphere shaders. Unlike standardized lower-level graphics APIs like OpenGL and DirectX, there exist tens of scene-graph APIs with different features [10, 34]. Thus, scene graph approach is also more risky – it is difficult to create a scene-graph level API that is compatible with multiple existing scene graph implementations.

As a final note, by looking at the relatively high instruction count limits in table 15, one may assume that splitting shaders into multiple passes is not necessary with latest hardware. Unfortunately, this is not true. Although instruction count limits have become large, the number of supported temporary registers is relatively low – 32. Even worse, the number of texture coordinate sets is still eight. Thus, it is quite easy to create high-level shader exceeding these resources and high-level shading systems must still split complex shaders into multiple passes.

6 Conclusions

This thesis has given an overview of existing real-time shading systems and discussed the design of an experimental shading system. The discussed system is based on concepts introduced by Stanford RTSL, most notably it uses similar computation-frequency concept. This allows the user to write a single high-level shader that is divided by the compiler into primitive group, vertex and fragment programs in a well-defined manner. This is different from the approach of other high-level shading languages like HLSL, Cg or glsl – in these languages, the user must specify separate shaders for each programmable unit of graphics processor – and must connect these shaders together manually. Unlike Stanford RTSL, the discussed shading system provides extended metaprogramming support. This allows the user to create complex shaders by combining together simple shaders.

In addition, we compare the performance of the discussed shading system to the fixed-function OpenGL pipeline. Although our system is little bit slower in most tests (the largest difference is less than 20%), we believe that this penalty is justified as it is very easy to extend the implemented lighting model or convert the per-vertex lighting model to per-fragment model.

We provide examples of two possible applications of real-time shading systems. First, we implement a simple ray-tracer using the shading language. Then, we demonstrate a possbility to reduce the gap between offline rendering systems and real-time systems by implementing an important subset of Softimage|XSI shader graphs using our shading system. The discussed implementation is efficient and simple, largely due to the metaprogramming features present within our shading system.

Reaalaja varjutussüsteemi disain ja rakendused

Mark Tehver

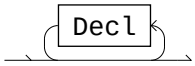
Kokkuvõte

Viimaste aastate jooksul on graafikaprotsessorite puhul asendatud paljud piiratud funktsionaalsusega osad programmeeritavate osade vastu. Peamiseks põhjuseks on vajadus võimaldada graafikaprotsessoritel kasutada erinevaid *varjutusmeetodeid*, mis määravad kolmemõõtmeliste objektide visuaalsed omadused. Klassikalised meetodid nagu *Phongi varjutusmudel* annavad häid tulemusi vaid väheste materjalide puhul. Samas pole võimalik luua ühtset praktilist mudelit, mis sobiks kõikide materjalide jaoks ja mis oleks samas ka efektiivselt realiseeritav. Seetõttu on antud võimalus kirjeldada erinevaid *varjutusprotseduure*, mis täidetakse graafikaprotsessorite poolt. Varjutusprotseduuride kirjeldamine graafikaprotsessori jaoks toimub aga masinkäskude tasemel ja nõuab kasutajalt detailseid teadmisi graafikaprotsessorite arhitektuuri kohta. Seetõttu on viimase paari aasta jooksul loodud mitmed varjutussüsteemid, mis peidavad madalama taseme detailid kasutaja eest ja lubavad keskenduda paremini varjutusmudelite realiseerimisele.

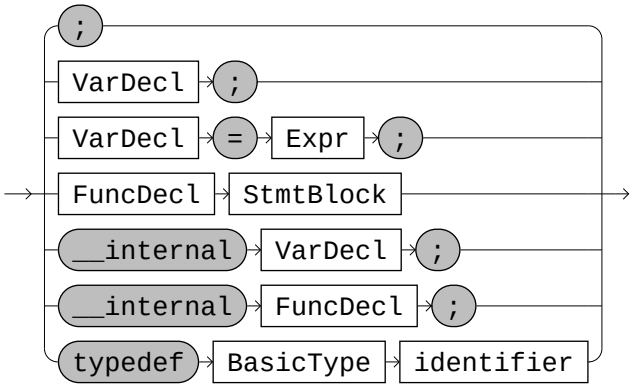
Käesolev töö kirjeldab ühe varjutussüsteemi loomist, annab ülevaate spetsiaalsest *varjutuskeelest* ja probleemidest, mis tuleb lahendada sarnaste süsteemide disainimisel. Lisaks annab töö ka ülevaate loodud süsteemi efektiivsusest, töös võrreldakse OpenGL graafikaliidese standardvarjutusmudeli kiirust varjutussüsteemi abil loodud mudeliga. Lisaks näitab töö, kuidas on võimalik kasutada varjutussüsteemi ka graafikaprotsessorite jaoks mittestandardsete joonistusmeetodite realiseerimiseks, nagu näiteks piiratud *ray-tracing* meetodi jaoks. Lõpuks tuuakse näide ka varjutussüsteemi peamise rakendusvaldkonna kohta – kirjeldatakse, kuidas lähendada reaalaja graafikat ja nn. *offline* graafikat. Näitena on realiseeritud *Softimage|XSI* varjutusgraafide automaatne transleerimine varjutussüsteemi abil.

A Experimental Shading Language Syntax

Prog

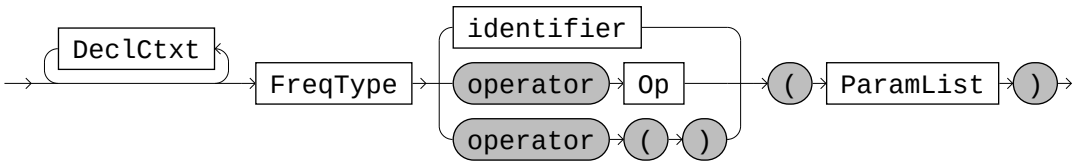


Decl

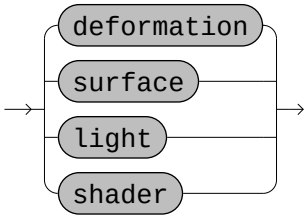


VarDecl

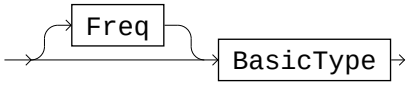
FuncDecl



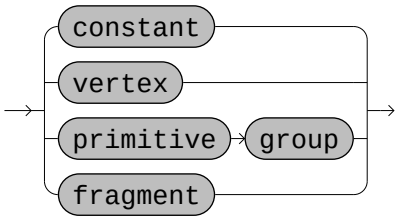
DeclCtxt



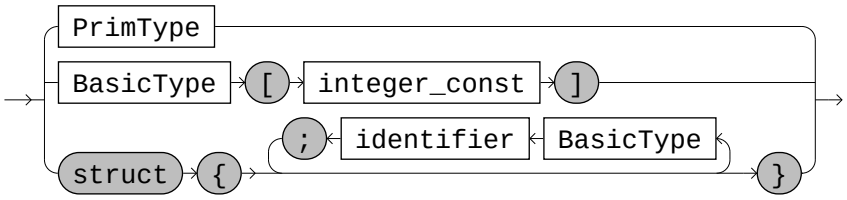
FreqType



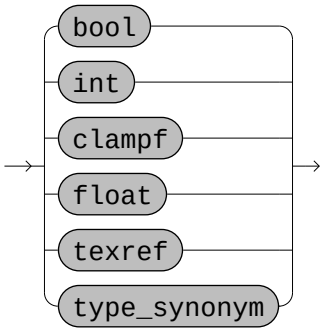
Freq



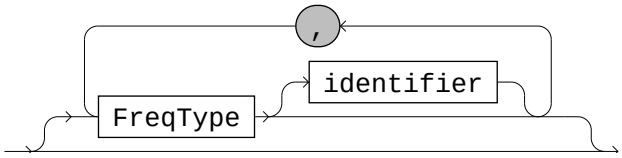
BasicType



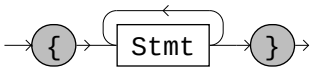
PrimType



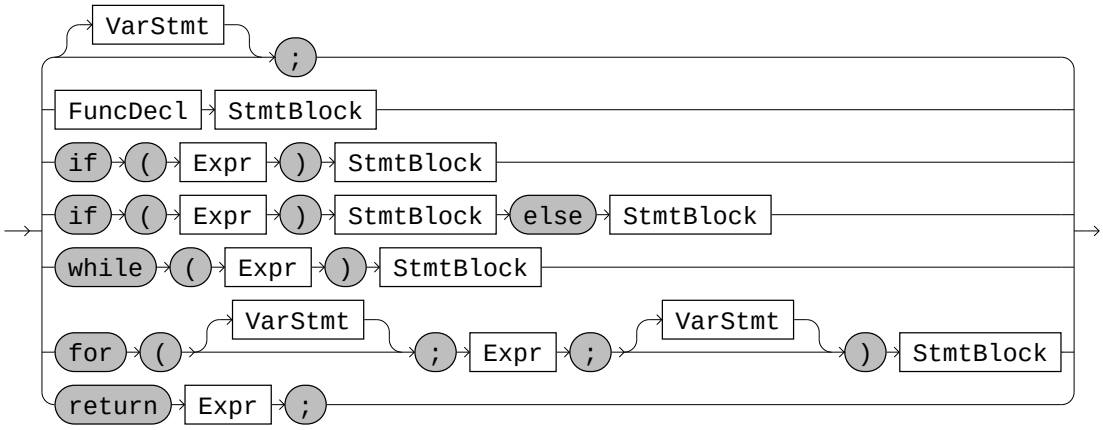
ParamList



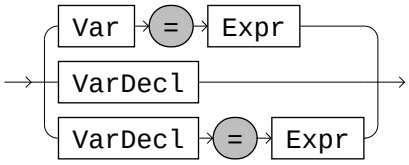
StmtBlock



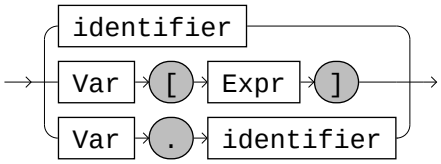
Stmt



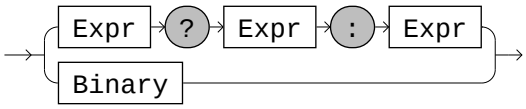
VarStmt



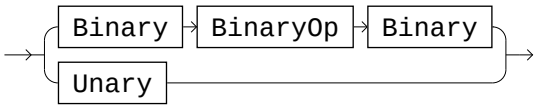
Var



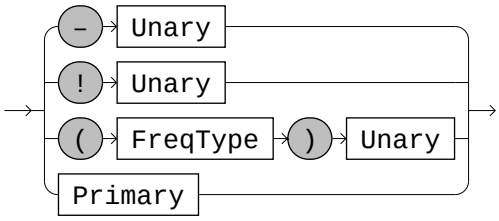
Expr



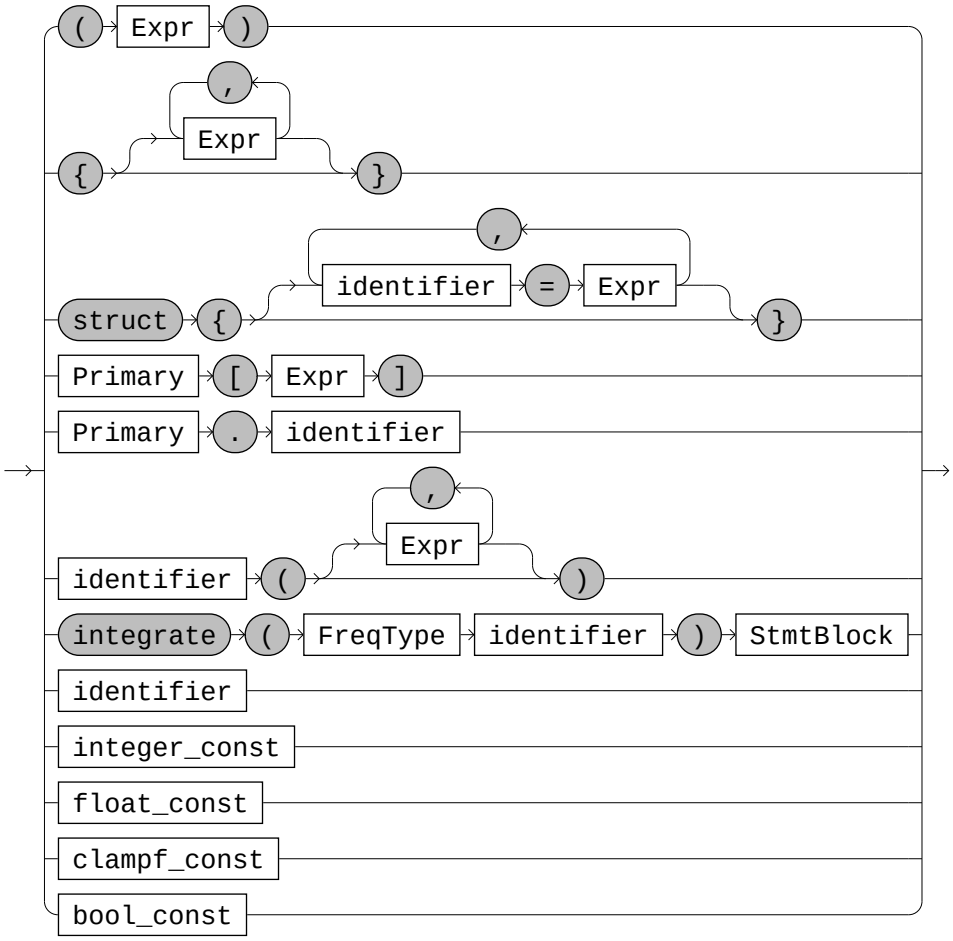
Binary



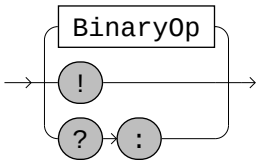
Unary



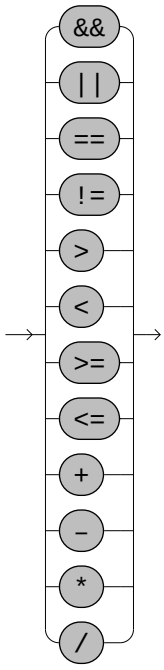
Primary



Op



BinaryOp



B Shading Language Prelude File

```
/*
 * Shading language built-in types, constants, operators and functions
 *
 * (c) 2001 Mark Tehver
 */

typedef float[2]    float2;
typedef float[3]    float3;
typedef float[4]    float4;
typedef clampf[2]   clampf2;
typedef clampf[3]   clampf3;
typedef clampf[4]   clampf4;
typedef float[2][2] matrix2;
typedef float[3][3] matrix3;
typedef float[4][4] matrix4;

// Expected output type of deformation shaders
typedef struct {
  float4 position; float3 normal; float3 binormal; float3 tangent;
} deformation_t;

// Return type for advanced surface shaders
// (provides support for overwriting default depth value and killing fragment)
typedef struct {
  float4 color; float depth; bool kill;
} fragment_t;

/*
 * All common operations generated through macro expansion
 */

#foreach Freq (constant, primitive group, vertex, fragment)
  __internal Freq float  operator () (Freq int);
  __internal Freq float  operator () (Freq clampf);
  __internal Freq clampf operator () (Freq float);

  __internal Freq bool operator >= (Freq float, Freq float);
  __internal Freq bool operator <  (Freq float, Freq float);
  __internal Freq bool operator !  (Freq bool);
  __internal Freq bool operator && (Freq bool, Freq bool);
  __internal Freq bool operator || (Freq bool, Freq bool);

  #foreach Type (float, float2, float3, float4)
    __internal Freq Type operator - (Freq Type);
    __internal Freq Type operator + (Freq Type, Freq Type);
    __internal Freq Type operator - (Freq Type, Freq Type);
    __internal Freq Type operator * (Freq Type, Freq Type);

    __internal Freq Type abs(Freq Type);
```

```
    __internal Freq Type floor(Freq Type);
    __internal Freq Type frac(Freq Type);
    __internal Freq Type min(Freq Type, Freq Type);
    __internal Freq Type max(Freq Type, Freq Type);
    __internal Freq float dot(Freq Type, Freq Type);
  #endfor

  __internal Freq float rcp(Freq float);   // rcp(x)=1/x
  __internal Freq float rsqrt(Freq float); // rsqrt(x)=1/sqrt(x)
  __internal Freq float exp2(Freq float);  // exp2(x)=pow(2, x)
  __internal Freq float log2(Freq float);  // log2(x)=log(2, x)
  __internal Freq float pow(Freq float, Freq float);
#endfor

/*
 * Texref operations
 */

#foreach Freq (constant, primitive group)
  __internal Freq bool isnil(Freq texref); // return true if texref is 0
#endfor

/*
 * Trigonometrical, approximation functions
 */

#foreach Freq (constant, primitive group, fragment)
  __internal Freq float sin(Freq float);
  __internal Freq float cos(Freq float);
#endfor

#foreach Freq (primitive group, vertex, fragment)
  __internal Freq float3 cross(Freq float3, Freq float3);
#endfor

__internal constant int   operator () (constant float);
__internal constant int   operator - (constant int);
__internal constant int   operator + (constant int, constant int);
__internal constant int   operator - (constant int, constant int);
__internal constant int   operator * (constant int, constant int);
__internal constant float operator / (constant float, constant float);

__internal constant int   abs(constant int);
__internal constant int   min(constant int, constant int);
__internal constant int   max(constant int, constant int);
__internal constant float sqrt(constant float);

/*
 * Lighting operations
 */
```

```
// See ARB vertex and fragment program spec. for definitions
#foreach Freq (vertex, fragment)
  __internal Freq float3 lit(Freq float, Freq float, Freq float);
  __internal Freq float4 dist(Freq float, Freq float);
#endfor

/*
 * Fragment level operations
 */

__internal fragment float4 tex1D(primitive group texref, fragment float);
__internal fragment float4 tex1D(primitive group texref, fragment float2);
__internal fragment float4 tex2D(primitive group texref, fragment float2);
__internal fragment float4 tex2D(primitive group texref, fragment float3);
__internal fragment float4 tex3D(primitive group texref, fragment float3);
__internal fragment float4 tex3D(primitive group texref, fragment float4);
__internal fragment float4 texRECT(primitive group texref, fragment float2);
__internal fragment float4 texRECT(primitive group texref, fragment float3);
__internal fragment float4 texCUBE(primitive group texref, fragment float3);

// Shadow functions work as GL_ARB_shadow extension specifies
#ifdef GL_ARB_fragment_program_shadow
  __internal fragment float4 shadow1D(primitive group texref, fragment float2);
  __internal fragment float4 shadow1D(primitive group texref, fragment float3);
  __internal fragment float4 shadow2D(primitive group texref, fragment float3);
  __internal fragment float4 shadow2D(primitive group texref, fragment float4);
  __internal fragment float4 shadowRECT(
    primitive group texref, fragment float3
  );
  __internal fragment float4 shadowRECT(
    primitive group texref, fragment float4
  );
#endif

// Linear interpolation: lerp(x,y,c)=(1-c)*x + c*y
#foreach Type (float, float2, float3, float4)
  __internal fragment Type lerp(fragment Type, fragment Type, fragment Type);
#endfor

#foreach Type (float2, float3, float4)
  __internal fragment Type lerp(fragment Type, fragment Type, fragment float);
#endfor

/*
 * Matrix operations
 */

#foreach Type (matrix2, matrix3, matrix4)
  __internal primitive group Type operator - (primitive group Type);
  __internal primitive group Type
             operator + (primitive group Type, primitive group Type);
```

```
  __internal primitive group Type
             operator - (primitive group Type, primitive group Type);
  __internal primitive group Type
             operator * (primitive group Type, primitive group Type);
  __internal primitive group Type invert(primitive group Type);
  __internal primitive group Type transpose(primitive group Type);
#endfor

__internal primitive group matrix3 affine(primitive group matrix4);

/*
 * Derived arithmetic operations
 */

float2 operator * (float2 v, float s) { return v * { s, s }; }
float2 operator * (float s, float2 v) { return { s, s } * v; }
float3 operator * (float3 v, float s) { return v * { s, s, s }; }
float3 operator * (float s, float3 v) { return { s, s, s } * v; }
float4 operator * (float4 v, float s) { return v * { s, s, s, s }; }
float4 operator * (float s, float4 v) { return { s, s, s, s } * v; }

float operator / (float x, float y)
{
  return x * rcp(y);
}

bool operator <= (float x, float y)
{
  return y >= x;
}

bool operator > (float x, float y)
{
  return y < x;
}

bool operator == (float x, float y)
{
  return x >= y && y >= x;
}

bool operator != (float x, float y)
{
  return x < y || y < x;
}

float sqrt(float x)
{
  return rsqrt(x) * x;
}
```

```
float ceil(float x)
{
  return -floor(-x);
}

float2 sincos(float x)
{
  // First 4 members of Taylor series
  float pi = 3.14159265358979;
  float x1 = frac(x * (0.5 / pi) + 0.5) * (2 * pi) - pi;
  float x2 = x1 * x1;
  float x4 = x2 * x2;
  float x6 = x4 * x2;
  float x8 = x4 * x4;
  float4 u = { x2, x4, x6, x8 };
  float s = x1 * (1 + dot(u, { -1.0/6.0, 1.0/120.0, -1/5040.0, 1/362880.0 }));
  float c = 1 + dot(u, { -1.0/2.0, 1.0/24.0, -1/720.0, 1/40320.0 });
  return { s, c };
}

float sin(float x)
{
  return sincos(x)[0];
}

float cos(float x)
{
  return sincos(x)[1];
}

float sign(float x)
{
  return x < 0 ? -1 : (x > 0 ? 1 : 0);
}

float clamp(float x, float x0, float x1)
{
  return min(max(x, x0), x1);
}

float step(float x0, float x)
{
  return x < x0 ? 0 : 1;
}

float smoothstep(float x0, float x1, float x)
{
  clampf sx = (clampf) ((x - x0) / (x1 - x0));
  return (-2 * sx + 3) * sx * sx;
}
```

```
float mod(float x, float y)
{
  return frac(x / y) * y;
}

/*
 * Derived vector operations
 */

float length(float2 v) { return sqrt(dot(v, v)); }
float length(float3 v) { return sqrt(dot(v, v)); }
float length(float4 v) { return sqrt(dot(v, v)); }

float3 rgb(float4 v)
{
  return { v[0], v[1], v[2] };
}

float alpha(float4 v)
{
  return v[3];
}

float3 direction(float4 v)
{
  return rgb(v);
}

float3 normalize(float3 v)
{
  float s = rsqrt(dot(v, v));
  return v * s;
}

float4 project(float4 v)
{
  float s = rcp(v[3]);
  return v * s;
}

float3 cross(float3 v1, float3 v2)
{
  float x = v1[1] * v2[2] - v1[2] * v2[1];
  float y = v1[2] * v2[0] - v1[0] * v2[2];
  float z = v1[0] * v2[1] - v1[1] * v2[0];
  return { x, y, z };
}

float3 faceforward(float3 n, float3 i)
{
  return dot(n, i) < 0 ? -n : n;
```

```
}

float3 reflect(float3 v, float3 n)
{
  float3 n_unit = normalize(n);
  return (2 * dot(n_unit, v)) * n_unit - v;
}

#foreach Type (float, float2, float3, float4)
  Type lerp(Type a, Type b, float t)
  {
    return a * t + b * (1 - t);
  }
#endfor

float2 lerp(float2 a, float2 b, float2 t)
{
  return a * t + b * ({ 1, 1 } - t);
}

float3 lerp(float3 a, float3 b, float3 t)
{
  return a * t + b * ({ 1, 1, 1 } - t);
}

float4 lerp(float4 a, float4 b, float4 t)
{
  return a * t + b * ({ 1, 1, 1, 1 } - t);
}

/*
 * Matrix operations
 */

matrix2 identity2()
{
  return { { 1, 0 }, { 0, 1 } };
}

matrix3 identity3()
{
  return { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };
}

matrix4 identity4()
{
  return { { 1, 0, 0, 0 }, { 0, 1, 0, 0 }, { 0, 0, 1, 0 }, { 0, 0, 0, 1 } };
}

matrix3 affine(matrix4 m)
{
```

```
  return {
    { m[0][0], m[0][1], m[0][2] },
    { m[1][0], m[1][1], m[1][2] },
    { m[2][0], m[2][1], m[2][2] }
  };
}

matrix2 transpose(matrix2 m)
{
  return
  {
    { m[0][0], m[1][0] },
    { m[0][1], m[1][1] }
  };
}

matrix3 transpose(matrix3 m)
{
  return {
    { m[0][0], m[1][0], m[2][0] },
    { m[0][1], m[1][1], m[2][1] },
    { m[0][2], m[1][2], m[2][2] }
  };
}

matrix4 transpose(matrix4 m)
{
  return {
    { m[0][0], m[1][0], m[2][0], m[3][0] },
    { m[0][1], m[1][1], m[2][1], m[3][1] },
    { m[0][2], m[1][2], m[2][2], m[3][2] },
    { m[0][3], m[1][3], m[2][3], m[3][3] }
  };
}

float2 operator * (matrix2 m, float2 v)
{
  return { dot(m[0], v), dot(m[1], v) };
}

float2 operator * (float2 v, matrix2 m)
{
  matrix2 mt = transpose(m);
  return { dot(mt[0], { v[0] }.xx), dot(mt[1], { v[1] }.xx) };
}

float3 operator * (matrix3 m, float3 v)
{
  return { dot(m[0], v), dot(m[1], v), dot(m[2], v) };
}
```

```
float3 operator * (float3 v, matrix3 m)
{
  matrix3 mt = transpose(m);
  return { dot(mt[0], { v[0] }.xxx),
           dot(mt[1], { v[1] }.xxx),
           dot(mt[2], { v[2] }.xxx) };
}

float4 operator * (matrix4 m, float4 v)
{
  return { dot(m[0], v), dot(m[1], v), dot(m[2], v), dot(m[3], v) };
}

float4 operator * (float4 v, matrix4 m)
{
  matrix4 mt = transpose(m);
  return { dot(mt[0], { v[0] }.xxxx), dot(mt[1], { v[1] }.xxxx),
           dot(mt[2], { v[2] }.xxxx), dot(mt[3], { v[3] }.xxxx) };
}

matrix2 operator + (matrix2 m1, matrix2 m2)
{
  return { m1[0] + m2[0], m1[1] + m2[1] };
}

matrix3 operator + (matrix3 m1, matrix2 m2)
{
  return { m1[0] + m2[0], m1[1] + m2[1], m1[2] + m2[2] };
}

matrix4 operator + (matrix4 m1, matrix4 m2)
{
  return { m1[0] + m2[0], m1[1] + m2[1], m1[2] + m2[2], m1[3] + m2[3] };
}

matrix2 operator * (matrix2 m1, matrix2 m2)
{
  matrix2 m2t = transpose(m2);
  return { m2t * m1[0], m2t * m1[1] };
}

matrix3 operator * (matrix3 m1, matrix3 m2)
{
  matrix3 m2t = transpose(m2);
  return { m2t * m1[0], m2t * m1[1], m2t * m1[2] };
}

matrix4 operator * (matrix4 m1, matrix4 m2)
{
  matrix4 m2t = transpose(m2);
  return { m2t * m1[0], m2t * m1[1], m2t * m1[2], m2t * m1[3] };
```

```
}

/*
 * Built-in variables
 */

__internal primitive group float4 __viewportscale;
__internal primitive group matrix4 __modelview;
__internal primitive group matrix4 __projection;
__internal primitive group float4 __ambient;

__internal light primitive group float4 __lightpos;
__internal light primitive group float3 __lightdir;
__internal light primitive group float3 __lightup;

/*
 * Special built-in internal functions
 */

// User-specified object-space vertex position and tangent-space
__internal vertex float4 __POSITION();
__internal vertex float3 __NORMAL();
__internal vertex float3 __TANGENT();
__internal vertex float3 __BINORMAL();

// Texture coordinates for mapping texture 1-1 to viewport
__internal vertex float3 __SCREENPOS();

// Compiler replaces this with deformation shader output
__internal vertex deformation_t __DEFORM();

// Framebuffer color of current fragment
__internal fragment float4 __SCREENCOLOR();

/*
 * Context-dependent variables for shaders
 */

#ifdef __HINT_UNIT
  float3 __normalize(float3 v) { return v; }
#else
  float3 __normalize(float3 v) { return normalize(v); }
#endif

#ifdef __HINT_PROJECTED
  float4 __project(float4 v) { return v; }
#else
  float4 __project(float4 v) { return project(v); }
#endif

#ifdef __HINT_RIGID
```

```
  vertex float3 __eyenormal =
    __normalize(affine(__modelview) * __DEFORM().normal);
#else
  vertex float3 __eyenormal =
    __normalize(transpose(invert(affine(__modelview))) * __DEFORM().normal);
#endif

vertex float4 __eyeposition =
  __project(__modelview * __DEFORM().position);
vertex float3 __eyebinormal =
  __normalize(affine(__modelview) * __DEFORM().binormal);
vertex float3 __eyetangent  =
  __normalize(affine(__modelview) * __DEFORM().tangent);

// NOTE: ARB vp does not have a "proper" ?: operator
// (it is emulated using multiplication),
// thus, we must be careful not to introduce infinities inside
// conditional expression
// (NV3x does not seem to enforce 0*inf=0, while R300 does)
light vertex float3 __eyelight =
  __lightpos[3] == 0
    ? -direction(__lightpos)
    :  direction(__eyeposition - (
         __lightpos[3] == 0 ? { 0, 0, 0, 0 } : project(__lightpos))
       );

/*
 * Public matrix variables
 */

// Modelview, projection and viewport transformation matrices
primitive group matrix4 Mview = __modelview;
primitive group matrix4 Mproj = __projection;
primitive group matrix4 Mvport = {
    { __viewportscale[0], 0, 0, __viewportscale[0] },
    { 0, __viewportscale[1], 0, __viewportscale[1] },
    { 0, 0, 1, 0 },
    { 0, 0, 0, 1 }
  };

/*
 * Public vertex-level variables
 */

// Surface positions, tangent-space vectors, eye vector, object-space position
surface vertex float4 P = __eyeposition;
surface vertex float3 N = __eyenormal;
surface vertex float3 T = __eyetangent;
surface vertex float3 B = __eyebinormal;
surface vertex float3 E =
  normalize(direction(__modelview * __DEFORM().position));
```

```
surface vertex float4 Pobj = __project(__DEFORM().position);

// Eye-space light vector, half-angle vector
light vertex float3 L = normalize(-__eyelight);
light surface vertex float3 H = normalize(L - E);

// Light-space surface vector, surface distance
light vertex float3 S =
  { cross(__lightdir, __lightup), __lightup, __lightdir } * __eyelight;
light vertex float Sdist = length(__eyelight);

/*
 * Deformation shader input variables
 */

// Object-space position, tangnent-space vectors
deformation vertex float4 PP = __POSITION();
deformation vertex float3 NN = __NORMAL();
deformation vertex float3 TT = __TANGENT();
deformation vertex float3 BB = __BINORMAL();

/*
 * Predefined colors
 */

// Ambient color, previous fragment color
surface primitive group float4 Ca = __ambient;
surface fragment float4 Cprev = __SCREENCOLOR();

/*
 * Default deformation shader (used when user does not specify any)
 */

float4 __position_transform(float4 pos)
{
  return (__projection * __modelview) * pos;
}

deformation shader deformation_t __default_deformation()
{
  return struct { position = PP, normal = NN, binormal = BB, tangent = TT };
}
```

C C Language API for Shading System

```
/*
 * This file is part of shading system runtime
 *
 * (c) 2001 Mark Tehver
 * Permission to copy, use, modify, sell and distribute this software
 * is granted provided this copyright notice appears in all copies.
 * This software is provided "as is" without express or implied
 * warranty, and with no claim as to its suitability for any purpose.
 */

#ifndef sgl_API_h

#define sgl_API_h

#ifdef _WIN32
#include <windows.h>
#endif
#include <GL/gl.h>

#ifdef _WIN32
#ifdef SGL_DLL_EXPORTS
#define SGL_API __declspec(dllexport)
#else
#define SGL_API __declspec(dllimport)
#endif
#define SGL_APIENTRY APIENTRY
#else
#define SGL_API
#define SGL_APIENTRY
#endif

#ifdef __cplusplus
extern "C" {
#endif

/*
 * Constants
 */

/* Vertex array usage types */

#define SGL_DYNAMIC_USAGE   0x1000
#define SGL_STATIC_USAGE    0x1001
#define SGL_STREAM_USAGE    0x1002

/* Vertex array types */

#define SGL_VERTEX_ARRAY    0x1100
#define SGL_NORMAL_ARRAY    0x1101
```

```
#define SGL_TANGENT_ARRAY   0x1102
#define SGL_BINORMAL_ARRAY  0x1103

/* Parameter frequencies */

#define SGL_CONSTANT_FREQ   0x2000
#define SGL_PRIMGROUP_FREQ  0x2001
#define SGL_VERTEX_FREQ     0x2002

/* Parameter types */

#define SGL_BOOL_TYPE       0x3000
#define SGL_INT_TYPE        0x3001
#define SGL_TEXREF_TYPE     0x3002
#define SGL_CLAMPF_TYPE     0x3003
#define SGL_FLOAT_TYPE      0x3004
#define SGL_TUPLE_TYPE      0x3005
#define SGL_STRUCT_TYPE     0x3006

/* Error reporting */

#define SGL_MSG_NONE        0x4000
#define SGL_MSG_WARN_ONCE   0x4001
#define SGL_MSG_WARN        0x4002
#define SGL_MSG_ABORT       0x4003

/* Light parameters */

#define SGL_LIGHT_POSITION  0x5000
#define SGL_LIGHT_DIRECTION 0x5001
#define SGL_LIGHT_UPAXIS    0x5002

/*
 * Routines
 */

/* Initialization, error checking, misc. */

SGL_API void   SGL_APIENTRY sglProperties(const char * propstr);
SGL_API void   SGL_APIENTRY sglInit(void);
SGL_API void   SGL_APIENTRY sglStop(void);
SGL_API void   SGL_APIENTRY sglClear(GLbitfield mask);
SGL_API void   SGL_APIENTRY sglViewport(
       GLint x, GLint y, GLsizei width, GLsizei height
);
SGL_API void   SGL_APIENTRY sglDebugLevel(GLenum minor, GLenum major);
SGL_API GLenum SGL_APIENTRY sglGetError(void);
SGL_API const GLubyte * SGL_APIENTRY sglErrorString(GLenum errcode);
SGL_API void   SGL_APIENTRY sglLoadCache(const char * fname);
SGL_API void   SGL_APIENTRY sglSaveCache(const char * fname);
```

```
/* Program loading and binding */

SGL_API GLuint SGL_APIENTRY sglCreateProgram(const char * progstr);
SGL_API GLuint SGL_APIENTRY sglLoadProgram(const char * fname);
SGL_API void   SGL_APIENTRY sglFreeProgram(GLuint progid);
SGL_API void   SGL_APIENTRY sglBindProgram(GLuint progid);

/* Shader loading, binding and compiling */

SGL_API GLuint SGL_APIENTRY sglLoadShader(const char * sname);
SGL_API void   SGL_APIENTRY sglFreeShader(GLuint shaderid);
SGL_API void   SGL_APIENTRY sglBindShader(GLuint shaderid);
SGL_API void   SGL_APIENTRY sglPrecompileShader(void);
SGL_API void   SGL_APIENTRY sglCompileShader(void);

/* Shader parameter information */

SGL_API GLuint SGL_APIENTRY sglParameterHandle(const char * pname);
SGL_API const char * SGL_APIENTRY sglGetParameterName(GLuint phandle);
SGL_API GLenum SGL_APIENTRY sglGetParameterType(GLuint phandle);
SGL_API GLenum SGL_APIENTRY sglGetParameterFrequency(GLuint phandle);
SGL_API GLuint SGL_APIENTRY sglGetParameterSize(GLuint phandle);
SGL_API GLuint SGL_APIENTRY sglGetParameterHandle(GLuint phandle, GLuint idx);
SGL_API GLuint SGL_APIENTRY sglGetParameterOffset(GLuint phandle);

/* Light control and shader connection */

SGL_API void SGL_APIENTRY sglAmbient4fv(const GLfloat * p);
SGL_API void SGL_APIENTRY sglLightPosefv(
       GLuint lightid, GLenum pname, const GLfloat * p
);
SGL_API void SGL_APIENTRY sglUseLight(GLuint lightid);
SGL_API void SGL_APIENTRY sglUseDeformation(GLuint deformationid);
SGL_API void SGL_APIENTRY sglUseShader(GLuint phandle, GLuint shaderid);

/* Shader parameter setting */

SGL_API void SGL_APIENTRY sglShaderParameter1i(
       GLuint shaderid, GLuint handle, GLint p
);
SGL_API void SGL_APIENTRY sglShaderParameter1ui(
       GLuint shaderid, GLuint handle, GLuint p
);
SGL_API void SGL_APIENTRY sglShaderParameter1f(
       GLuint shaderid, GLuint handle, GLfloat p
);
SGL_API void SGL_APIENTRY sglShaderParameter2f(
       GLuint shaderid, GLuint handle, GLfloat p1, GLfloat p2
);
SGL_API void SGL_APIENTRY sglShaderParameter2fv(
       GLuint shaderid, GLuint handle, const GLfloat * p
```

```
);
SGL_API void SGL_APIENTRY sglShaderParameter3f(
       GLuint shaderid, GLuint handle, GLfloat p1, GLfloat p2, GLfloat p3
);
SGL_API void SGL_APIENTRY sglShaderParameter3fv(
       GLuint shaderid, GLuint handle, const GLfloat * p
);
SGL_API void SGL_APIENTRY sglShaderParameter4f(
       GLuint shaderid, GLuint handle,
       GLfloat p1, GLfloat p2, GLfloat p3, GLfloat p4
);
SGL_API void SGL_APIENTRY sglShaderParameter4fv(
       GLuint shaderid, GLuint handle, const GLfloat * p
);
SGL_API void SGL_APIENTRY sglShaderParameter9fv(
       GLuint shaderid, GLuint handle, const GLfloat * p
);
SGL_API void SGL_APIENTRY sglShaderParameter16fv(
       GLuint shaderid, GLuint handle, const GLfloat * p
);

/* Bound shader parameter setting */

SGL_API void SGL_APIENTRY sglParameter1i(GLuint handle, GLint p);
SGL_API void SGL_APIENTRY sglParameter1ui(GLuint handle, GLuint p);
SGL_API void SGL_APIENTRY sglParameter1f(GLuint handle, GLfloat p);
SGL_API void SGL_APIENTRY sglParameter2f(
  GLuint handle, GLfloat p1, GLfloat p2
);
SGL_API void SGL_APIENTRY sglParameter2fv(GLuint handle, const GLfloat * p);
SGL_API void SGL_APIENTRY sglParameter3f(
       GLuint handle, GLfloat p1, GLfloat p2, GLfloat p3
);
SGL_API void SGL_APIENTRY sglParameter3fv(GLuint handle, const GLfloat * p);
SGL_API void SGL_APIENTRY sglParameter4f(
       GLuint handle, GLfloat p1, GLfloat p2, GLfloat p3, GLfloat p4
);
SGL_API void SGL_APIENTRY sglParameter4fv(GLuint handle, const GLfloat * p);
SGL_API void SGL_APIENTRY sglParameter9fv(GLuint handle, const GLfloat * p);
SGL_API void SGL_APIENTRY sglParameter16fv(GLuint handle, const GLfloat * p);

/* Vertex and tangent-space setting */

SGL_API void SGL_APIENTRY sglVertex3f(GLfloat p1, GLfloat p2, GLfloat p3);
SGL_API void SGL_APIENTRY sglVertex3fv(const GLfloat * p);
SGL_API void SGL_APIENTRY sglVertex4f(
  GLfloat p1, GLfloat p2, GLfloat p3, GLfloat p4
);
SGL_API void SGL_APIENTRY sglVertex4fv(const GLfloat * p);
SGL_API void SGL_APIENTRY sglNormal3f(GLfloat p1, GLfloat p2, GLfloat p3);
SGL_API void SGL_APIENTRY sglNormal3fv(const GLfloat * p);
```

```
SGL_API void SGL_APIENTRY sglTangent3f(GLfloat p1, GLfloat p2, GLfloat p3);
SGL_API void SGL_APIENTRY sglTangent3fv(const GLfloat * p);
SGL_API void SGL_APIENTRY sglBinormal3f(GLfloat p1, GLfloat p2, GLfloat p3);
SGL_API void SGL_APIENTRY sglBinormal3fv(const GLfloat * p);

/* Scope and flushing */

SGL_API void SGL_APIENTRY sglBegin(GLenum mode);
SGL_API void SGL_APIENTRY sglEnd(void);
SGL_API void SGL_APIENTRY sglFlush(void);
SGL_API void SGL_APIENTRY sglFinish(void);

/* Vertex array object API */

SGL_API GLuint SGL_APIENTRY sglNewArray(
  GLenum type, GLuint size, GLenum usage
);
SGL_API GLuint SGL_APIENTRY sglNewParamArray(
  GLuint phandle, GLuint size, GLenum usage
);
SGL_API void   SGL_APIENTRY sglResizeArray(GLuint array, GLuint size);
SGL_API void   SGL_APIENTRY sglUpdateArray(
  GLuint array, GLuint eloffset, GLuint elcount,
  GLenum type, GLsizei stride, GLuint size, GLuint offset, const void * data
);
SGL_API void   SGL_APIENTRY sglBindArray(GLuint array, GLuint offset);
SGL_API void   SGL_APIENTRY sglUnbindArray(GLuint array);
SGL_API void   SGL_APIENTRY sglDrawArrays(
  GLenum mode, GLuint first, GLuint count
);
SGL_API void   SGL_APIENTRY sglDrawElements(
  GLenum mode, GLuint count, GLenum type, const GLvoid * indices
);
SGL_API void   SGL_APIENTRY sglFreeArray(GLuint array);

#ifdef __cplusplus
}
#endif

#endif
```

References

[1] Arcot Preetham Avi Bleiweiss. Ashli - advanced shading language interface. In *Siggraph 2003 Notes*, 2003. Available from `http://www.ati.com/developer/SIGGRAPH03/AshliNotes.pdf` (30.04.2004).

[2] R. L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, number Vol.18, pages 223–231, January 1984.

[3] John Kessenich Dave Baldwin, Randi Rost. *The OpenGL Shading Language. Version 1.0.*, June 12, 2002. Available from `http://www.3dlabs.com/support/developer/ogl2/specs/glslangspecv1.0.pdf` (30.04.2004).

[4] Paul J. Diefenbach. *Pipeline Rendering: Interactive and Realism Through Hardware-based Multi-pass Rendering*. PhD thesis, University of Pennsylvania, 1996.

[5] Thomas Driemeyer. *Rendering with mental ray*, volume 1. Springer-Verlag New York, 2000.

[6] Pradeep Sen Kekoa Proudfoot Eric Chan, Ren Ng and Pat Hanrahan. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 1–11, 2002.

[7] T. Whitted G.D. Abram. Building block shaders. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 283–288, 1990.

[8] Larry Gritz and James K. Hahn. Bmrt: a global illumination implementation of the renderman standard. *J. Graph. Tools*, 1(3):29–48, 1996.

[9] Paul Heckebert. Fundamentals of texture mapping and image warping. Master's thesis, UCB/CSD 89/516, CS Division, U.C. Berkeley, June 1989.

[10] K.Hinrichs J. Dollner. A generic rendering system. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):99–118, April–June 2002.

[11] Peter-Pike Sloan Jan Kautz and John Snyder. Fast, arbitrary brdf shading for low-frequency lighting using spherical harmonics. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 291–296. Eurographics Association, 2002.

[12] David A. Patterson John L. Hennessy. *Computer Architecture: A Quantitative Approach; second edition.* Morgan Kaufmann, 1996.

[13] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150. ACM Press, 1986.

[14] Svetoslav Tzvetkov Pat Hanrahan Kekoa Proudfoot, William R. Mark. A real-time procedural shading system for programmable graphics hardware. In *28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 159–170. ACM Press, 2001.

[15] Robert R. Lewis. Making Shaders More Physically Plausible. In *Fourth Eurographics Workshop on Rendering*, number Series EG 93 RW, pages 47–62, Paris, France, 1993.

[16] Erik Lindholm, Mark J. Kligard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM Press, 2001.

[17] William R. Mark and Kekoa Proudfoot. The f-buffer: a rasterization-order fifo buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 57–64. ACM Press, 2001.

[18] John Airey P. Jeffrey Ungar Mark S. Peercy, Mark Olano. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000 (New Orleans, Louisiana, July 23-28, 2000). In Computer Graphics, Annual Conference Series.* ACM SIGGRAPH, 2000.

[19] Kurt Akeley Mark Segal. *The OpenGL Graphics System: A Specification (Version 1.5).*

[20] Michael D. McCool. *Sh - Embedded Metaprogramming Language.* Available from `http://libsh.sourceforge.net/` (30.04.2004).

[21] Michael D. McCool. Smash: A next-generation api for programmable graphics accelerators. In *SIGGRAPH 2000 Course on Real-Time Programmable Shading, New Orleans*, 2000.

[22] T.S. Popa Michael D. McCool, Z. Qin. Shader metaprogramming graphics hardware. In *Conference on Graphics hardware*, pages 1–12, 2002.

[23] Microsoft. *DirectX Graphics Programmers Guide. Microsoft Developers Network Library, DirectX 9 edition, 2002.*

[24] Mark Olano. *Interactive Shading Language (ISL) Language Description Version 2.4 March 26, 2002.* Available from `http://lust.u-strasbg.fr/Documentations/Visualisation/Shader/islspec.html` (30.04.2004).

[25] K. Perlin. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, 1985.

[26] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[27] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[28] K. E. Torrance R. L. Cook. A reflectance model for computer graphics. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, number Vol.15, No.3, pages 301–316, July 1981.

[29] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 283–291. ACM Press, 1987.

[30] Christophe Schlick. A survey of shading and reflectance models for computer graphics. In *Computer Graphics Forum, v13, n2*, pages 121–132, June 1994.

[31] SGI. *OpenGL Extension Registry.* Extension specifications available from `http://oss.sgi.com/projects/ogl-sample/registry/` (30.04.2004).

[32] The Aqsis Team. *Aqsis Renderer.* Homepage `http://www.aqsis.com` (30.04.2004).

[33] G. J. Ward. Measuring and modeling anisotropic reflection. In *Computer Graphics (SIGGRAPH'92 Proceedings)*, number Vol. 26, No.3, pages 265–272, July 1992.

[34] Josie Wernecke. *The Inventor Mentor : Programming Object-Oriented 3D Graphics with Open Inventor, Release 2.* Addison Wesley, 1994.