SMASH:
A Next-Generation API for
Programmable Graphics Accelerators

Technical Report CS-2000-14
Computer Graphics Lab
Department of Computer Science
University of Waterloo

August 1, 2000

API Version 0.2

http://www.cgl.uwaterloo.ca/Projects/rendering/Papers/smash.pdf

http://www.cgl.uwaterloo.ca/Projects/rendering/shaders.html

Michael D. McCool
mmccool@cgl.uwaterloo.ca

Abstract

The SMASH API is a testbed for real-time, low-level graphics concepts. It is being developed to serve as a concrete target for the development of advanced extensions to OpenGL as well as a driver to test hardware architectures to support these extensions. SMASH is syntactically and conceptually similar to OpenGL but supports (along with other experimental features) a programmable shader sub-API that is compatible with both multi-pass and single-pass implementations of shaders.

Arbitrary numbers of shader parameters of various types can be bound to vertices of geometric primitives using a simple immediate-mode mechanism. Run-time specification, manipulation, and compilation of shaders is supported. The intermediate-level shading language includes integrated support for per-vertex and per-fragment shaders under a common programming model.

Implementation of real-time rendering effects using SMASH could be enhanced with metaprogramming toolkits and techniques, up to and including RenderMan-like textual shading languages and C++ toolkits with similar levels of compactness and functionality. We give several examples of how a two-term separable BRDF approximation could be implemented using such higher-level constructs.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture.

Keywords: Hardware acceleration and interactive rendering, graphics application programming interfaces, shading languages.

1 Introduction

Real-time graphics systems have reached a turning point. Performance levels, measured in triangles per second, are so high and growing so rapidly that within a year, commodity systems will be available that can overwrite every single pixel of a 640×480 display with its own individual textured polygon over 50 times every 30th of a second. The XBox game platform will be one such example.

While there are still performance issues with rendering *really* large (trillion-primitive) environments and models, the real-time rendering research focus has shifted from rendering scenes quickly to rendering them well.

Due to this shift in emphasis, some classic computer graphics themes have been revived, but with a new emphasis on real-time implementation. Many papers have recently appeared on real-time physically-based global [21, 31, 52, 57] and local [9, 21, 24, 27, 28, 29] illumination, programmable shading [14, 33, 39, 41, 45], lens simulation and tone mapping [10, 15, 22], and even real-time raytracing [49].[1]

We have come to expect high quality from offline rendering systems. However, in some cases we have known for decades how to simulate certain effects in offline systems that are still infeasible in online systems. Implementing these effects in real time, and all at once as in sophisticated offline systems, will open up a range of new applications and will enhance existing ones.

At the same time, suddenly low-end graphics systems are bandwidth limited. Just a couple of years ago PC rasterization systems were so fast, relative to host-based geometry manipulation and transformation, that a good first approximation in analyzing rendering algorithms was to assume that hardware-accelerated texturing and rasterization speed was infinite. Now, with hardware-based transformation engines, graphics subsystems are bandwidth limited for *both* geometry specification *and* for texture and framebuffer access. This problem will only get worse as the exponentially rising ability to perform computation in the host and in the graphics accelerator continues to grow faster than the ability of memories and buses to provide data at high bandwidths and with low latency.

OpenGL and similar APIs are built around a conceptual model

[1] For a summary of real-time rendering techniques, see the recent book by Möller and Haines [38]—although the field is advancing so rapidly that this book should only be considered an introduction.

of graphics hardware. The conceptual model is a kind of contract between hardware architects and rendering algorithm implementors about what features will be provided by the graphics accelerator. This conceptual model, the common point of reference for both programmers and hardware architects, evolved under considerably different constraints than exist today. Unfortunately, some aspects of this conceptual model have outlived their usefulness.

For instance, it was necessary in the early stages of real-time rendering to use a very simple, hardwired, per-vertex lighting model, in conjunction with linear interpolation of color in screen space, to get adequate performance. However, not only is linear screen-space interpolation of colour inaccurate and artifact-prone, but for local lighting in particular there now exist several per-fragment lighting models based on texturing, which are not only more flexible than the existing OpenGL per-vertex lighting model, but also have greatly superior visual quality, and are not much more expensive to implement. In this context, hardware support for a per-vertex Phong lighting model, or at least for *only* this model, doesn't make a lot of sense.

It is probably a good time, therefore, to reexamine the conceptual architectures of our APIs to see if they are really the optimal solutions for implementing real-time, high-quality rendering algorithms. Of course, it's possible that the existing conceptual architecture *is* appropriate for efficiently implementing advanced rendering algorithms.

For instance, it has been recently demonstrated [45] that with some small changes, namely extended precision and signed arithmetic, the framebuffer and texture operators in the existing conceptual architectures can be used to evaluate RenderMan shading programs using multiple rendering passes. Programmable shaders are a major component of production-quality offline rendering systems, and the computational ability exposed and facilitated by such a system can be used to help implement many other advanced rendering techniques, so this is an important advance.

However, multipass shader evaluation has drawbacks. The number of operations in even a simple shader can be large. A shader that combines a number of rendering effects (shadows, glossy reflection, volumetric effects, etc.) can explode in complexity as features interact. Since a pure multipass implementation ties execution of operations to consumption of bandwidth, a complex shader has the potential to quickly use up all available memory bandwidth, even at the high rasterization speeds noted above.

Many hardware accelerators can be configured to perform additional computation in a single pass, via multitexturing and other extensions. Research work is ongoing to extend shader compiler technology to map onto multitexturing architectures, but graphics accelerators currently have several limitations that make this process difficult.

This begs the following questions:

1. Can we design accelerators so they would be easier to compile to?
2. How can an API be structured so that programs written to it will be portable, yet implementations will be scalable?
3. How should we design accelerators so they can be used to efficiently implement a variety of advanced rendering techniques?
4. For what workloads and feature sets should accelerator architectures be optimized?

These questions are difficult to answer, especially the last one. Hardware designers use traces of "typical" graphics applications to analyze performance [11, 25], but of course these applications have been optimized to map onto the *current* conceptual architecture.

There is a chicken-and-egg problem here: Until there are examples of advanced accelerators, applications cannot be optimized for them. Until there are applications for advanced accelerators, there is no motivation for developing these accelarators, no test data to optimize their performance, and no guidance as to what the appropriate feature sets should be. OpenGL's solution to this problem is to evolve incrementally—but evolutionary optimization can easily get stuck in local minima. The alternative is for hardware designers and algorithm developers to work together to try and reach consensus on a new conceptual architecture.

1.1 Goals and Assumptions

The SMASH project is a next-generation graphics accelerator/API hardware/software codesign project and architectural study. The goal is to develop a conceptual architecture appropriate for future graphics accelerators capable of efficiently supporting advanced rendering and modelling algorithms. SMASH is an acronym that stands for "**S**imple **M**odelling **A**nd **SH**ading"—simplicity and elegance being among our goals.

Revisiting the basic assumptions of real-time rendering, we are designing and analyzing the potential performance of a next-generation, programmable, real-time graphics subsystem. Our design has the following specific goals:

1. High frame rate.
2. High modelling complexity.
3. High rendering quality.
4. Low frame latency.
5. Low cost.
6. Flexibility.
7. Portability.
8. Scalability.
9. Ease of learning and use.

If the last goal doesn't seem important, consider that even now advanced performance features, such as multitexturing or host SIMD instructions, are often not used because they are relatively difficult to program. A system that is easy to use and learn will get used more and will have a wider impact, all other things being equal.

To keep things easy to learn, we have tried to keep the number of basic concepts low, have made SMASH consistent with OpenGL when reasonable, have striven for simplicity and consistency, and have eliminated redundant features. However, we have also implemented "conveniences" where appropriate and where they would improve programmer productivity, for instance to enhance debugging—as long as these features do not interfere with high performance.

Scalability means the ability to scale an implementation from a low-cost, moderate performance system to one with very high performance, ideally with a linear or near-linear increase in cost [11]. Portability means that the same program should run on systems from different manufacturers, with reasonable performance levels and scalability being obtained without extensive programmer intervention. Attaining these goals means devising a conceptual architecture which permits a range of implementations that can effectively exploit massive parallelism.

The flexibility goal is also important. OpenGL has proven to be flexible, but getting it to do what you want sometimes requires the use of operators for tasks for which they were not designed and are often not optimal, extensive testing on different implementations to find fast paths and work around the bugs and limitations of various implementations, application of various tricks to simulate signed numbers or higher precision or range, etc. The result is code that is often brittle and unmaintainable, inefficiently uses resources, and is quickly obsolete.

We want to avoid these problems by generalizing the operators that have proven to be most powerful, providing an explicit portable model of programmability, virtualizing resources, providing a powerful compiler subsystem as part of the driver, and removing unnecessary "legacy" limitations (or in some cases, addressing the lack of appropriate limitations).

Portability is also enhanced by the emphasis in SMASH on programmablity—features supported with specific hardware on one platform can be simulated with appropriate programming on another. The SMASH API has been designed to deliberately hide distinctions between built-in and programmed features.

As for the goal of a low-cost implementation, ideally we want SMASH to be simple enough that a high-performance implementation can be built as a single-chip solution in the near future. To facilitate this goal, the design is in fact somewhat conservative. For instance, the design uses z-buffer hidden surface removal rather than order-independent a-buffer, ray-tracing, or scan-line hidden surface removal. SMASH also uses a relatively standard pipeline order so that existing studies of how to implement such pipelines in a scalable fashion [11, 25] are applicable.

Finally, the design is also based on and constrained by the following assumptions about near-future hardware:

1. Relatively low-bandwidth, high-latency memory systems.
2. A relatively low-bandwidth host port, but with a need for flexible (i.e. immediate-mode) geometry specification.
3. High computational performance and density in individual custom integrated circuits, with relatively high on-chip bandwidth.

To address the memory and host bandwidth issues in particular, the new conceptual model should include features to offload modelling and rendering tasks from the host (for instance, by tesselating higher-order surfaces) and to do as much computation (for instance, shading) in a single pass as feasible.

Because we want to contribute to the evolution of OpenGL and because we want SMASH to be easy to learn, the SMASH API is based loosely on OpenGL. However, large chunks of fixed functionality in OpenGL are replaced in SMASH with programmable subsystems, and parts of OpenGL which have been rendered obsolete, like the fixed lighting model, have been removed.

The SMASH API depends heavily on metaprogramming for optimization. Writing to the SMASH API really involves writing a host-CPU program that writes an accelerator-GPU program adapted to the capabilities of a given graphics system. SMASH drivers will include significant just-in-time compiler support to perform low-level adaption of the intermediate-level specification supplied at the API level to the hardware implementation. The programmer is expected to provide high-level strategic guidance only.

The idea of the SMASH project is not to replace OpenGL, but to provide an example to guide its evolution, hopefully allowing it to avoid getting stuck in a local optimum. Like a concept car, the idea isn't that you can buy one right away (or ever), but at least it will be possible to determine if you want something like it.

1.2 Outline

In this document we focus mainly on the programmable shading API for SMASH, which is currently its most well-developed aspect. It is our intention to make freely available the base software implementation of SMASH for research and evaluation purposes; please visit the website noted on the first page of this document. We are also working on a testbed hardware implementation using a Xilinx FGPA prototyping system and a high-performance software implementation on the Intel architecture. This document will be updated as we develop further aspects of SMASH; please mention the date, the version number, and the URL when citing this document.

This document is structured as follows: In Section 2 we review interesting rendering algorithms that have been implemented using hardware acceleration, and make some general observations about these systems that we have used to guide our design. Then, in Section 3, we review the conceptual architectures of current graphics subsystems, and present a high-level overview of the SMASH conceptual architecture.

The sections following these introductory sections provide detail on specific subsystems within SMASH: geometry specification (Section 4), parameter binding (Section 5), defining texture objects (Section 6), shader specification (Section 7), and fragment operations and rasterization (Section 8).

In Section 9 we consider approaches to high-performance implementation, with a focus on the programmable parts of the revised pipeline. We review potential implementation strategies for accelerators, in particular multithreaded processors and reconfigurable computing.

Section 10 is devoted to programming examples. The SMASH API and the just-in-time compiler subsystem built into its driver is meant to be used in conjunction with metaprogramming, and has specific built-in features to make this relatively easy. We give examples of how higher-level shading languages can be built (using the operator overloading facilities of C++ in particular) and can be layered on top of the base functionality provided.

Capabilities intentionally omitted for various reasons are discussed in Section 11. We conclude with a list of future topics we plan to attack in the development of SMASH in Section 12. Currently several "obvious" things are missing—SMASH is *definitely* a work in progress.

1.3 Conventions and Further Information

SMASH uses many of the naming conventions of OpenGL but with `sm` and `SM` in place of `gl` and `GL`. This renaming is necessary so that OpenGL and SMASH can coexist peacefully—SMASH implementations may in fact be layered on top of OpenGL. The base SMASH API uses a C interface like OpenGL but will be associated with utility libraries written in C++ to provide selective syntactic sugaring.

2 Prior Art

Prior work relevant to the development of SMASH includes work done to map advanced rendering algorithms to existing accelerators, work on the efficient implementation of shading languages, and work on the development and analysis of architectures for graphics accelerators. We cover the first two topics briefly here and cover some topics relevant to the implementation of high-performance graphics accelerators in Section 9.

2.1 Advanced Rendering Effects

Several researchers have developed multipass techniques for generating high-quality images using the operations available in contemporary graphics hardware. The effects covered by these techniques include bump mapping [37], normal mapping [23], and reflections off planar [9] and curved reflectors [16, 21, 40]. The traditional local illumination model used by graphics hardware can also be extended to include shadows (using shadow maps [50], shadow volumes [9, 35], or horizon maps (for precomputed self-shadowing) [24, 51]), arbitrary bidirectional reflectance functions [7, 27, 29, 30, 28, 21], complex light sources [20, 50], and caustics [54]. The abstract operations supported by OpenGL's conceptual model have been summarized and mathematically characterized by Trendall [54].

Other researchers have developed methods for solving the global illumination problem with the help of graphics hardware. Keller [31] uses multipass rendering to generate indirect illumination for diffuse environments. Stürzlinger and Bastos [53] can render indirect illumination in glossy environments by visualizing the solution of a photon map algorithm. Stamminger *et al* [52] and Walter *et al* [57] place OpenGL light sources at virtual positions to simulate indirect illumination reflected by glossy surfaces, effectively using Phong lobes as a set of basis functions. Heidrich and Seidel [21] use multiple rendering passes and environment maps to render global illumination solutions for non-diffuse environments.

Another useful class of techniques uses light fields; these can be either rendered directly [12, 34], used to illuminate other objects [20], or used to simulate realistic cameras and lens systems [22]. Hardware acceleration has been used for decompressing light fields at interactive rates.

In summary, it has been clearly demonstrated that high quality images can be achieved using current graphics hardware, using hand-tuned multipass techniques. Systems have also been demonstrated that combine several techniques [8, 9].

2.2 Shading Languages

Shading languages, such as Pixar's RenderMan shading language, [4, 17, 46, 56] can be used for more than just specifying local lighting models. Since shading programs assign color to a surface with a relatively arbitrary computation, and can use other sources of pretabulated information, they can also be used to render shadows, generate non-photorealistic "sketch" renderings for visualization and artistic purposes, and can be potentially used to integrate the results of global illumination solutions into a rendering.

Several attempts have been made to integrate programmable shading languages into interactive rendering. Some researchers have accelerated the software evaluation of shaders by precomputing parts of them (known as specialization) [14], or by parallelizing shader computation over a number of MIMD or SIMD processors [46].

Explicit hardware and architectural support for procedural shading has appeared in graphics accelerators, although until recently full support was limited to research prototypes. The most prominent example is the PixelFlow architecture [39, 41, 33], which is in the process of being commercialized (PixelFuzion). In this architecture, several rendering units based on both general-purpose microprocessors and specialized processor-enhanced memory run in parallel a program implementing a rendering pipeline on part of the scene database. Shaders are implemented using a SIMD array, with shader programs compiled from a high-level language (pfman) almost identical to Pixar's RenderMan shading language, except for the addition of fixed-point types. However, programmable shaders have not yet appeared in the commercial version of this system.

3 Accelerator Architectures

Diagrams of the current high-level architecture for OpenGL 1.2.1 are shown in Figure 1. There are actually several OpenGL conceptual architectures, depending on which of the various extensions and optional features are supported. Particularly important for the implementation of advanced rendering effects are the imaging extensions (which include convolution as well as other simpler frame-

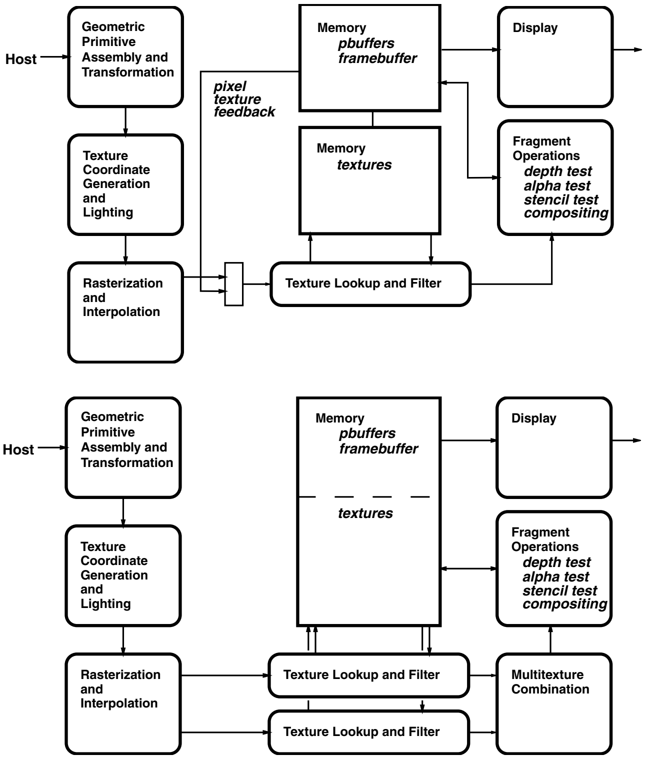
Host
Geometric Primitive Assembly and Transformation
Memory
pbuffers
framebuffer
Display
pixel texture feedback
Texture Coordinate Generation and Lighting
Memory
textures
Fragment Operations
depth test
alpha test
stencil test
compositing
Rasterization and Interpolation
Texture Lookup and Filter
Host
Geometric Primitive Assembly and Transformation
Memory
pbuffers
framebuffer
textures
Display
Texture Coordinate Generation and Lighting
Fragment Operations
depth test
alpha test
stencil test
compositing
Rasterization and Interpolation
Texture Lookup and Filter
Texture Lookup and Filter
Multitexture Combination

Figure 1: *In practice OpenGL 1.2.1 defines two main architectures, diagrammed here. In the first architecture, only a single texture unit is supported. This is typical of high-end SGI systems, some of which also support additional features such as pixel texture feedback, and which usually also have physically separate texture and framebuffer memories. On PC graphics systems, multitexturing units are common, in which multiple texture lookups and a limited amount of computation can be performed before fragments are written to the framebuffer. Also, usually a single unified memory is used.*

buffer manipulation operations), various forms of multitexturing, extended environment map formats such as cube maps, higher-dimensional texture formats, and various forms of dependent texturing (including bump-mapping and pixel-texture feedback).

OpenGL implementations can also vary along other dimensions, such as in the relative performance of rasterization *vs.* transformation. Memory models can also vary between implementations; in particular, copies between texture memory and framebuffer memory may or may not be cheap, an issue for certain algorithms that conceptually render into texture maps [51]. OpenGL's conservative approach of using a split memory system in its conceptual model (because some systems are in fact implemented that way) requires an explicit copy between the framebuffer and texture memory. Unfortunately, this can lead to unnecessary data movement on systems with a unified memory system: such systems could potentially render directly into a texture but this possibility is not currently exposed in OpenGL.

The SMASH high-level conceptual architecture is shown in Figure 2. The units which we are primarily concerned with in this document are the programmable vertex shader and the programmable fragment shader, which replace the lighting and the texturing units, respectively.

Ultimately SMASH will support a "synchronized" buffer model which can be efficiently implemented on both split and unified

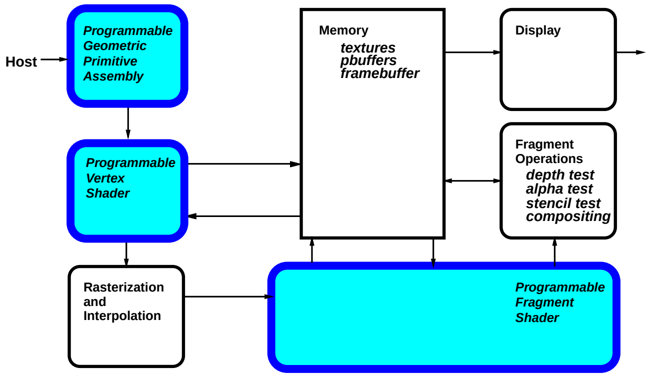
Host
Programmable Geometric Primitive Assembly
Memory
textures
pbuffers
framebuffer
Display
Programmable Vertex Shader
Fragment Operations
depth test
alpha test
stencil test
compositing
Rasterization and Interpolation
Programmable Fragment Shader

Figure 2: *The SMASH conceptual architecture replaces parts of the OpenGL graphics pipeline with programmable units, although the overall structure of the pipeline is retained.*

memory models, and a programmable geometry generation and manipulation system (shown here immediately before the vertex shader unit). We will cover these subsystems in greater detail in later versions of this report.

Rather than the fixed texture coordinate generation and lighting units of OpenGL, the SMASH architecture has a programmable *vertex shading unit*. Vertex shaders can be simulated in the driver on the host system, ideally using high-performance features of the host CPU, such as Intel's SSE or AMD's 3DNow! instructions. Defining them as part of the graphics API, however, permits the transparent migration of vertex shader capabilities into the hardware accelerator.

Fragment shaders apply additional shading operations after the rasterization stage, ultimately generating the color and depth of each fragment. Results of the vertex shaders, as well as additional parameters attached to the vertices of the primitives, are hyperbolically interpolated [5, 43] before being passed down to the fragment shader. For maximum performance, the fragment shader should ideally be supported by a hardware accelerator, since it must efficiently process the large number of fragments generated by the rasterizer.

In the SMASH conceptual model the vertex shader and the fragment shader have identical capabilities, including access to textures. This permits parts of a shader evaluation to be allocated to either the vertex or fragment level, as required. It also permits the use of lookup tables in vertex shaders to evaluate tabulated functions.

4 Specifying Geometry

Geometric primitives in SMASH can currently be specified using only an immediate-mode interface. While an immediate-mode interface puts more of a burden on the host processor, it is more flexible than an array-based interface, use of higher-level primitives (subdivision surfaces, displacement maps) will eventually offset some of its limitations, and multithreaded interfaces can be used to balance host generation and accelerator consumption rates, as proposed for OpenGL [11, 26, 25]. Retained mode is useful for some purposes, so SMASH will eventually support an array-based interface and OpenGL-style display lists.

There are a few differences between the SMASH model of primitive assembly and OpenGL's. SMASH assembles triangles internally from generic streams of vertices and "steering" commands. It is our intention to make the assembly process programmable, with the flexibility to support advanced features such as various forms of mesh compression, subdivision surfaces, and displacement maps. To this end, the identifier passed to the **smBegin** call is a *geometry assembly object identifier*, not just an enumerated constant as in OpenGL.

A geometry assembly object identifies a *geometry assembly program* that converts vertex/command streams into streams of triangles. For compatibility, there are predefined geometry assembly objects that support generalizations of the usual OpenGL primitive types, with names similar to those of OpenGL.

The triangle strip primitive supported by SMASH, using one of these predefined assemblers, is in fact a generalization of OpenGL's triangle strip primitive. SMASH triangle fans are just a special case of the SMASH triangle strip primitive with slightly different initial conditions. Hybrids of strips and fans can also be created.

Like GL, but unlike OpenGL, SMASH can explictly indicate a "vertex swap" operation without actually repeating a vertex. Unlike GL, SMASH uses a generalized mechanism to indicate the swap, which can ultimately be extended to other purposes.

The **smSteer** call issues an *assembly steering command* into the vertex stream. Assembly steering commands can modify the current geometry assembly state, but their effect depends on the current assembler and the current state of the assembler. This is best explained with an example.

For the predefined SM_TRIANGLE_STRIP and SM_TRIANGLE_FAN geometry assembly program, the current assembly state can be one of eight values: FanLoadLeft, FanLoadRight, Left, Right, AltLeft, AltRight, StripLoadLeft, and StripLoadRight. The geometric primitive assembler also has a vertex cache. To implement steerable triangle strips, only two vertices need to be cached: an "old left" vertex L and an "old right" vertex R. These cache slots will be used to hold the vertices to the left and right of the last edge crossed before the current triangle, when that triangle is viewed from its front face.

In both the SM_TRIANGLE_FAN and SM_TRIANGLE_STRIP primitive assemblers the first two vertices given are placed in L and R, in that order. These are the roles of the FanLoadLeft and FanLoadRight states for triangle fans, and the StripLoadLeft and StripLoadRight states for triangle strips. The initial state when assembling a triangle fan is FanLoadLeft. A transition from FanLoadLeft to FanLoadRight is caused by a **smVertex*** call and loads the vertex into L. When another vertex is specified in the FanLoadRight state, the vertex is loaded into R and a transition is made into the Left state. These transistions cannot be initiated or affected by steering commands. A similar sequence holds for SM_TRIANGLE_STRIP but with the state sequence StripLoadLeft, StripLoadRight and AltRight.

Each subsequent vertex V generates triangles and replaces the L and R vertices using the following rules when in each of the four remaining states:

Left: This is the initial state for SM_TRIANGLE_FAN after the first two vertices are loaded. When a new vertex arrives, it is used to generate a new triangle and then the strip turns to the left. Turning to the left means that the new current edge has the same leftmost vertex, but we replace the rightmost vertex in the cache:

- Emit a triangle using LRV.
- Replace R with V.

AltLeft: In this state, a new vertex generates a new triangle, turns to the left, then changes state to turn to the right next time:

- Emit a triangle using LRV.
- Replace R with V.
- Enter the AltRight state.

Right: When a vertex arrives when the assembler is in this state, the assembler generates a new triangle, then turns to the right while remaining in the same state. This state can be used to generate a right-handed triangle fan:

- Emit a triangle using LRV.
- Replace L with V.

`AltRight`: This is the initial state for the `SM_TRIANGLE_STRIP` assembler after loading the first two vertices. When a new vertex arrives while the assembler is in this state, it generates a new triangle by turning to the right, then changes state to turn to the left next time:

- Emit a triangle using LRV.
- Replace L with V.
- Enter the AltLeft state.

The steering commands for this assembler simply force a change of state without changing the vertex cache or emitting a triangle. These commands use predefined constants named after their target states: `SM_LEFT`, `SM_RIGHT`, `SM_ALT_RIGHT`, and `SM_ALT_LEFT`. In summary, the assembly state can be changed explicitly, with an **smSteer** call, or implicitly, with an **smVertex** call.

5 Binding Parameters

In OpenGL, the fixed lighting model is parameterized by a number of multidimensional values, such as normals, light source directions, colors, and texture coordinates. These parameters are bound to vertices when primitives are specified. Some of these parameters are used to compute other values at the vertices that are then interpolated, while others are interpolated directly. Finally, the interpolated parameters and derived values are interpreted by the fragment processing unit.

Multiple texture coordinates are currently supported only in some implementations of OpenGL and with a tight binding between texture units, texture parameters, and texture objects. While they have been used as such, texture coordinates as they currently exist in OpenGL do not really have the right properties to be used as general-purpose shader parameters.

The introduction of per-vertex and per-fragment programmable shaders requires a much looser binding of parameters to primitives, and also access to and application of other important internal state, such as the modelview transformation.

The parameter sub-API of SMASH permits the specification of a list of generalized parameters that can be bound to each primitive and each vertex of a primitive. Parameters are typed: they can be normals, covectors, vectors, tangents, planes, points, texture coordinates, and just plain generic parameters.

The different types of parameter differ in how they are transformed and normalized, and can also be used, during development, to catch mismatches between shader parameters and the parameters expected by a shader (i.e. to perform type checking). However, ultimately all parameters are represented by fixed-length tuples of numbers which are identically and hyperbolically interpolated by the rasterizer, without reference to the type.[2]

Multiple normals, tangents, and any other kind of parameter may be specified in immediate mode with one API call each. Specifying a parameter pushes it onto a stack. The parameter stack actually consists of a stack of numbers and a stack of indexes into the stack for each parameter; in this way we can support parameters of different dimensionality on the stack.

A distinction is made between parameters that vary across a surface and those that are constant over a primitive using a "stack locking" mechanism. Inside a primitive's scope, the parameter stack is reset at each **smVertex** call to the state it had at the **smBegin** call for that primitive. Therefore, parameters pushed onto the stack before opening a primitive are constant for the duration of that primitive. An example is in order before we get into the details of the interface mechanism. Consider Figure 3.

```
smParam2d(0.2, 0.5);
smMatrixMode(SM_MODELVIEW);
smPushMatrix();
    smRotated(30.0, 0.3, 0.4, 0.5);
    smTranslatei(50, -30, 10);
    smPoint3d(100.0, 10.5, 15.6);
smPopMatrix();
smTangent3d(0.0, -0.5, 0.5);
smBegin(SM_TRIANGLES);
    smColor3d(0.3, 0.5, 0.2);
    smTexCoord2d(0.2, 0.1);
    smParam1i(5);
    smTexCoord1d(0.0);
    smNormal3d(0.7, 0.5, 0.5);
    smVertex3d(0.0, 0.0, 0.0);

    smColor3d(0.4, 0.6, 0.1);
    smTexCoord2d(0.6, 0.7);
    smParam1i(7);
    smTexCoord1d(0.4);
    smNormal3d(0.5, 0.7, 0.5);
    smVertex3d(0.0, 1.0, 0.5);

    smColor3d(0.3, 0.5, 0.2);
    smTexCoord2d(0.9, 0.8);
    smParam1i(3);
    smTexCoord1d(0.7);
    smNormal3d(0.5, 0.5, 0.7);
    smVertex3d(1.0, 1.0, 1.5);
smEnd(SM_TRIANGLES);
```

Figure 3: *Example of parameter binding mechanism. This code fragment defines a triangle with three constant parameters (a generic 2D parameter, a transformed point, and a tangent vector) as well as five per-vertex interpolated parameters (an RGB color, a 2D texture coordinate, a 1D parameter, a 1D texture coordinate, and a normal). Parameters are specified before the vertex to which they attach, and constant parameters are specified outside* ***smBegin/smEnd*** *blocks. At a* ***smVertex*** *call the index pointing to the top of the parameter stack, by default, is reset to its value at the time of the* ***smBegin*** *call.*

5.1 Parameter Types and Transformation

Given a transformation matrix A, a vector $\vec{\mathbf{b}}$ is conceptually coordinatized by a tuple of numbers arranged in a column, and is is transformed by $\mathsf{A}\vec{\mathbf{b}}$. In contrast, a *covector* $\vec{\mathbf{c}}$ is conceptually coordinatized by a tuple of numbers arranged in row. A covector is transformed by $\vec{\mathbf{c}}\mathsf{A}^{-1}$. Normals are covectors; tangents are vectors. Covectors and vectors are duals of one another.

Vectors and covectors should be normalized to unit length for certain operations, so they represent a "pure" direction. The notation $\hat{\mathbf{c}}$ will be used for vectors and covectors of unit length.

Points and planes (as coefficients of plane equations) are also duals can both be represented as homogenous tuples, in which case

[2] The common interpolation model may be lifted in the future. For now, SMASH should *at least* use a common hyperbolic interpolator.

they are coordinatized using columns and rows and transform like vectors and covectors, respectively.

SMASH defines nine types of parameters in five categories. The categories differ in how they are transformed.

Generic: Basic (generic) parameters are not transformed at all, just pushed onto the parameter stack.

Color: Colors are not transformed. We use a separate type, however, because color-space transformations may be supported in the future, and (secondarily) to enhance type-checking.

Primal Geometry: Tangents, vectors, and points are transformed by the current modelview matrix. Conceptually they are coordinatized using column tuples. Depending on the state of the `SM_NORMALIZE_TANGENTS` and `SM_RESCALE_TANGENTS` flags, tangents may also be rescaled or normalized to unit length. Points and vectors are never normalized automatically.

The output dimensionality of a transformed point, vector, or tangent is always the same as the input.

Dual Geometry: Normals, covectors, and planes are transformed by the inverse transpose of the current modelview matrix. Conceptually they are coordinatized using row tuples. Depending on the state of the `SM_NORMALIZE_NORMALS` and `SM_RESCALE_NORMALS` flags, normals may also be rescaled or normalized to unit length. Planes and covectors are never normalized automatically.

Coordinates: Texture coordinates are transformed by the current texture matrix and then pushed onto the parameter stack.

Variants of the texture coordinate specification function give the dimensionality of the post-transformed texture coordinate if it is different than the input, for instance to extend a 1D texture coordinate to 3D or vice-versa.

Several familiar operations for which specialized interfaces are used in OpenGL can use the parameter stack (in conjunction with an appropriate shader) instead.

For instance, to specify a light source position, just push a point onto the parameter stack. To specify a light source direction, push a vector. Both will be transformed by the current modelview matrix and end up in the view coordinate system.

You can push multiple light source positions or directions, a set of generic parameters to specify attenuation coefficients, etc. Of course, you need an appropriate shader to interpret these parameters. A utility library for SMASH will implement the Phong lighting model using the shader interface defined in Section 7, among other lighting models. The Phong lighting model is not part of SMASH proper.

5.2 Specifying Parameters

To specify a parameter, simply use one of the following calls to transform the parameter according to its type and push it on the parameter stack:

smParam{**1234**}{**sifd**}(T *p*...)
smColor{**1234**}{**sifd**}(T *p*...)
smTexCoord[{**1234**}**from**]{**1234**}{**sifd**}(T *p*...)
smNormal{**3**}{**sifd**}(T *p*...)
smCovector{**3**}{**sifd**}(T *p*...)
smPlane{**34**}{**sifd**}(T *p*...)
smTangent{**3**}{**sifd**}(T *p*...)
smVector{**23**}{**sifd**}(T *p*...)
smPoint[**4from**]{**234**}{**sifd**}(T *p*...)

When one of **smParam***, **smNormal***, **smColor*** etc. is called (we will refer to these functions generically as "parameter specification functions"), its argument is transformed and pushed onto the parameter stack. The parameter stack consists of an undifferentiated sequence of floating-point numbers (called "parameter stack elements"). A second stack is used internally that indexes into this sequence to identify the start of each parameter tuple (called a "parameter stack item"). The net effect is that the parameter stack behaves as if it can store variable-length parameters.

Transformations are linear or affine depending on the type of the parameter. They may be linear transformations of homogeneous coordinates, but such coordinates are *not* normalized after transformation. Projective normalization, if any, must be performed explicitly by the shader, and of course can be omitted for points if the transformation is affine. This is to permit, for instance, indexing of a 4D lightfield by a 4D texture coordinate, and omission of the projective divide when it is unnecessary.

Each element of every parameter will always be interpolated identically and hyperbolically in a projectively correct manner. Normals and tangents are *not* spherically interpolated or renormalized at each pixel to unit length. If this is desirable, it can be simulated using an appropriate shader program.

Strictly speaking, only **smParam** is really needed, the rest of the types and their associated transformations are just for convenience. The shader program could just apply appropriate matrix transformations as needed. Unfortunately this would require constant insertion of chunks of code into the shader given by the user for what is likely to be an extremely common operation, and delaying transformation may not have the right effect. For instance, we usually want to specify lights in a different coordinate system than our models. The explicit-transformation approach also obfuscates the role and transformation properties of normal covectors and tangent vectors making the resulting code harder to maintain. The current proposal also permits a certain amount of type-checking and manipulation of parameters using transformations set up on the matrix stacks.

SMASH supports various modes for deferring transformations to make the semantics of SMASH easier to implement on hardware expected in the near future—as foreshadowed, for instance, by the proposed modifications to the DirectX interface for supporting programmable vertex shaders.

Deferring transformation of parameters until the **smVertex** call would be better for implementation on hardware with a programmable vertex shader. Unfortunately, this behaviour would be contrary to the current OpenGL semantics and would have the problems noted above with specification of lighting parameters. Therefore, we currently support a mode which can be set with a flag named `SM_DEFER_TRANSFORMATIONS` to defer transformations of parameters constant over a primitive to the **smBegin** call and of interpolated parameters to the **smVertex** call. Depending on the implementation, this can be faster, slower, or the same speed as transforming parameters immediately[3]. However, as testing this flag in parameter API calls takes time that is in short supply during immediate-mode rendering, we may change this in the future. Options include never deferring (best for user flexibility), always deferring (probably best for performance, so parameters can be sent down to the hardware as a group), and supporting disjoint sets of parameter specification calls, one set that defers, another that doesn't.

The parameter specification functions may be called anywhere, inside or outside a **smBegin**/**smEnd** primitive block. Normally, calls outside a primitive block will be used to define constant parameters, those inside will be used to define parameters that vary for each vertex and are interpolated across the primitive. This is done using a set of stack-resetting rules, described in detail in the following section.

[3] Querying performance tradeoffs is another place OpenGL needs some work. . .

5.3 Parameter Stack Control

The call

smPopParam()

explicitly discards the frontmost (last-defined) item of the parameter stack. The call

smDropParams(SMsizei n)

pops the $n \geq 1$ frontmost items off the stack. Finally, the call

smClearParams()

clears or partially clears the stack, depending on whether it is called inside or outside a **smBegin**/**smEnd** primitive block.

Associated with the parameter stack are several externally visible and manipulable pieces of state: a Boolean flag accessed with the enumerated constant

SM_PARAM_AUTORESET

and integers accessed with the enumerated constants

SM_PARAM_CONSTANT_ITEMS
SM_PARAM_ITEMS
SM_PARAM_CONSTANT_ELEMENTS
SM_PARAM_ELEMENTS

The limits on size of the parameter stack can be determined by recalling integers using the enumerated constants

SM_PARAM_MAX_ITEMS
SM_PARAM_MAX_ELEMENTS

In the following we will use these names to refer to the corresponding state and values.

The integer SM_PARAM_ELEMENTS holds the total number of elements (individual scalar numbers) currently on the parameter stack. If the element stack is visualized as an array of scalars, it points to the next free element. It is always less than or equal to the implementation-dependent integer constant SM_PARAM_MAX_ELEMENTS. Its initial value is 0, indicating that the stack is by default empty.

The integer SM_PARAM_ITEMS holds the total number of items currently on the parameter stack. If the element stack is visualized as an array of items, it points to the next free item slot. However, as an item may consist of a variable number of elements, an item slot should be visualized as holding an offset into the element stack for the start of each item pushed onto the item stack. The maximum value of SM_PARAM_ITEMS is given by the value associated with SM_PARAM_MAX_ITEMS. Of course, the number of parameters that can be pushed may be less than this if the total number of elements in all items pushed exceeds SM_PARAM_MAX_ELEMENTS.

Whenever **smParam** (or some other parameter specification function) is called, inside *or* outside a primitive block, the integer SM_PARAM_ITEMS is incremented by 1, and the integer SM_PARAM_ELEMENTS is incremented by the (post-transformation) dimensionality of the parameter. If the resulting number of elements would be greater than SM_MAX_PARAM_ELEMENTS the contents of the stack are undefined. Whenever **smPopParam** is called, inside *or* outside a primitive block, SM_PARAM_ITEMS is decremented by 1, and the value of SM_PARAM_ELEMENTS is decremented by the dimensionality of the item popped from the top of the stack. If more items are popped than had been previously pushed, the contents of the stack will again be undefined.[4]

A **smBegin** call copies the current values of the integers SM_PARAM_ELEMENTS and SM_PARAM_ITEMS to SM_PARAM_CONSTANT_ELEMENTS and SM_PARAM_CONSTANT_ITEMS, respectively. Outside a primitive block, a pop of an empty stack generates an error and the contents of the stack are undefined.

A **smClearParams** call should be made to reset the error state.

Inside a primitive block, if the result of a pop would result in a stack size less than SM_PARAM_CONSTANT_ITEMS an error is generated and the contents of the stack are undefined. Whenever **smClearParam** is called outside a primitive block SM_PARAM_ELEMENTS and SM_PARAM_ITEMS are set to 0. Whenever **smClearParam** is called outside a primitive block these values are set to SM_PARAM_CONSTANT_ELEMENTS and SM_PARAM_CONSTANT_ITEMS. Note that given these rules and in the absence of underflow/overflow errors, SM_PARAM_CONSTANT_ELEMENTS is always less than or equal to SM_PARAM_ELEMENTS and SM_PARAM_CONSTANT_ITEMS is always less than or equal to SM_PARAM_ITEMS and both are greater than or equal to 0.

Both of SM_PARAM_CONSTANT_ELEMENTS and SM_PARAM_CONSTANT_ITEMS are set to 0 outside a primitive block.

The Boolean flag SM_PARAM_AUTORESET controls stack reset behaviour. Its value cannot be changed inside a primitive block. It defines two different modes for defining per-vertex parameters inside a primitive block. If it is true, then a vertex call resets the stack by copying the integers SM_PARAM_CONSTANT_ELEMENTS and SM_PARAM_CONSTANT_ITEMS back into SM_PARAM_ELEMENTS and SM_PARAM_ITEMS. If it is false, a **smVertex** call does *not* affect the stack, and the parameter stack must be explictly managed.

The net effect of these rules is as follows. Constant parameters, i.e. those that do not vary from vertex to vertex of a primitive, should be set up outside a **smBegin**/**smEnd** block. Inside a block, additional per-vertex "varying" parameters may be pushed onto the parameter stack but the constant parameters may not be modified. To share parameters between vertices, turn off the auto-reset and pop the parameter stack explicitly.[5]

Whenever a **smVertex** call is made, a snapshot of the current parameter stack is made and attached to the vertex. Within a single primitive block, the length of the parameter stack and the types of arguments need to be consistent with the types declared by the shader. When a primitive is rasterized, first a vertex shader generates new per-vertex derived parameters and concatenates those with the parameter stacks bound to each vertex. Within the rasterizer all parameter elements are interpolated in a projectively correct manner to each pixel fragment. Finally, all constant parameters and interpolated parameters are passed down the pipeline to the fragment shader.

[4] Trying to detect underflow and overflow errors on stacks can be a potential source of inefficiency. To avoid this problem, particularly for functions used for immediate-mode rendering, our implementation can be compiled in either *paranoid* mode or in *fast* mode. Using the library when in paranoid mode will try to detect and report all errors. When in fast mode, if an error is made the system is guaranteed not to crash or overwrite memory outside the graphics subsystem, but otherwise the results will be undefined.

[5] It is expected that autoreset mode will only be turned off in exceptionally complex cases. In fact, we'd like to invite comment on whether it is needed at all. As with deferral, a small amount of performance is potentially wasted in immediate mode to check the current mode and act accordingly. A higher-performance option would be to define a non-resetting version of the **smVertex** call.

6 Texture Objects

Texture objects in SMASH are in some ways simpler than in OpenGL; specifically, there are no associated texture coordinate generation modes because such functionality is subsumed by shaders. Texture identifiers in SMASH are also opaque and use the type `SMtexture` for texture object identifiers.[6]

7 Shader Specification

The SMASH shader sub-API gives a way for the application program to build a shader program on the fly. On-the-fly compilation is supported so that shaders can be optimized dynamically by the host program.

SMASH supports a stack-and-register expression language. Once a shader definition has been opened, individual API calls are used to add individual instructions to it; when a shader definition is complete, it is closed and then the driver compiles it. Other API calls exist to manage shader definitions, set compilation options, activate and deactivate a current shader, let the system know which shaders will be used in the near future (to optimize performance by pipelining shader loading) and so forth.

It should be emphasized that the abstract model of programming that SMASH uses is meant to be an efficient and flexible way to *specify* programs to an optimizing backend from front ends of various kinds. We most emphatically do *not* mean to imply that implementation of SMASH programs will necessarily use exact stack machine semantics, although a literal implementation is a convenient way to build a software interpretor for SMASH. Possible approaches to implementation and hardware acceleration are discussed in Section 9.

Storage for a shader program takes the form of a stack containing n-tuples of real numbers (with each tuple possibly being of different dimensionality) and a set of registers, also holding n-tuples. Conceptually the stack, the number of registers, and the length of items is unbounded. We do not define limits on registers, stack size, or shader length since the compiler may well optimize the shader and reduce the amount of memory needed. However, once a shader has been closed, the system can be queried to determine if the shader will be implemented using hardware acceleration. As with OpenGL, SMASH will fall back to a software implementation if hardware acceleration is not feasible—although of course this may not result in the desired performance, since software implementation of fragment shaders will also require software rasterization.

Arithmetic operations act on the stack, popping operands off and pushing results on. For each operation simple rules determine the dimensionality of the result given the dimensionality of the arguments; many operations come in several forms that differ only in the dimensionality of the operands expected. For instance, multiplication will multiply elementwise tuples of the same dimensionality, but if given a scalar and an n-tuple as an argument, will multiply the scalar by every element of the n-tuple and generate a new, scaled n-tuple. The dimensionality of the tuples are fixed at compile time and so this overloading will not incur any performance penalty during execution of the shader.

Load and store instructions can pop data off the stack and put it in a register, can just copy the top of stack to a register, and finally can retreive data from a register and push it on the stack. Registers contain tuples of arbitrary lengths, but like stack items these lengths are fixed at compile time. Registers can be allocated and deallocated in a stack-based fashion to facilitate use of nested scopes.

It is assumed that the backend optimizes away unnecessary data movement operations and dead code, so programmers are free (and should be encouraged) to write "readable" code rather than trying to perform these relatively trivial optimizations manually.

The stack machine architecture proposed enables a simple metaprogramming API and in particular permits the semantics of the host language to be used for modularity. Due to the stack-based nature of SMASH's execution and register allocation model, there is little potential for naming conflicts; what remains can be resolved using the namespace and scope mechanisms of the host language.

For instance, shader "macros" can simply be wrapped in host language functions that emit the necessary sequence of shader operations inline into an open shader definition; likewise scoped names can be given to registers by naming their identifiers using scoped variables in the host language.

7.1 Definition

Definitions of shader programs are opened by the following call:

`SMshader` **smBeginShader**`()`

which returns an opaque identfier of type `SMshader` that can be used later to refer to the shader. Shader identifiers can only be allocated by the system, unlike the case in OpenGL with texture object identifiers. Programmers should *not* assume shader identifiers are small numbers.

Open shader definitions are closed, and compilation initiated on the current shader program, by the following call:

smEndShader`()`

At most one shader definition can be open at any one time; calling **smBeginShader** before closing an open definition is an error.

Calls that can be used inside an open shader definition and *only* inside an open shader definition include the word **Shader**.

After calling **smEndShader** the system should be queried to determine if the shader program can be implemented using hardware acceleration and if so, how many resources it will consume. This can be done using the call

`SMdouble` **smShaderResources**`(SMshader` *`s`*`, SMenum` *`r`*`)`

with various values of the resource identifier *`r`*. To determine if a shader will be implemented using hardware acceleration, the resource `SM_SHADER_ACCEL` can be queried; a value greater than or equal to 100 indicates full hardware acceleration, a value less than that indicates that some fraction of the shader had to be performed with software assistance. Other resource tokens can be used to determine determine other metrics of performance, such as the number of passes, using resource `SM_SHADER_PASSES`. Of course, nothing beats benchmarking; the values returned by this call are only meant to be rough guides.

7.2 Activating Shaders

Shaders are made active using the following call:

smShader`(SMshader` *`s`*`)`

The active shader is also called the "current" shader. There can be only one active shader at a time.

Activating a shader also automatically activates and loads any associated texture objects. To maximize performance, shaders that share textures should be used in sequence; the system will detect this and avoid reloading an already loaded texture object.

To permit preloading of shader programs before they are actually used, the calls

[6] If this description seems terse, you're right. Further detail on this part of the API will be provided in a future version of this report.

smNextShader(SMshader *s*)
smNextShaders(SMsizei *n*, SMshader* *ss*)

hint to the system that the indicated shader(s) will be used in the near future and that the system should prepare to switch to them. When multiple shaders are hinted the ones with the lowest index in the array will be used the soonest. Whenever a hint call is made of either type it supercedes all previous hints. These calls should be made immediately after a call to **smShader** to maximize the time available for preparation. Violating a hint or omitting hints is not an error, but may reduce performance. Note that loading shaders is a fairly heavyweight process, since it potentially involves loading several textures.

7.3 Deleting Shaders

Finally, shaders can be deleted using

smDeleteShader(SMshader *s*)
smDeleteShaders(SMsizei *n*, SMshader* *ss*)

Deleting a shader does *not* delete any texture objects it uses. It is not mandatory to delete shader objects, but it may save resources.

7.4 Saving and Restoring Precompiled Shaders

The “shader object” abstraction is used because non-trivial compilation may be needed to map a shader program onto a given implementation, and also to permit fast switching between shaders when drawing a scene. However, while the intention is that compilation should be fast enough to take place during load time, very complex shaders might benefit from compilation during installation. Once a shader has been compiled for a particular graphics accelerator, a platform-dependent, opaque byte stream can be recalled, saved for later use, and restored using the following calls:

SMsizei **smShaderSize**(SMshader *s*)
smGetShaderProg(SMubyte* *prog*, SMshader *s*)
SMbool **smSetShaderProg**(SMint *s*, SMubyte* *prog*)

Storage for the array passed to **smGetShaderProg** must be allocated by the application program. If a bytestream has been previously stored, at load time the program needs only to check if the graphics hardware configuration has changed, since such precompiled bytestreams are not intended to be portable. The **smSetShaderProg** call returns false if the shader could not be loaded—typically because it was compiled for a different platform. In that case, it must be respecified.

When a shader is stored in a binary format all texture objects are saved with it, by default.[7]

7.5 Executing Shaders

Shaders are normally executed during rendering of primitives. However, a shader can also be executed explicitly using the following call:

smExecShader(SMdouble *p*[4])

This executes the active shader, using the parameters on the parameter stack, at position *p* (specified using homogeneous coordinates) in model space—the point *p* is transformed, in the same way a vertex would be, by the current modelview matrix. Results must be read back using the following call:

smGetExecShaderColor(SMdouble *c*[4])

Right now shaders have only one result, a color. However, as we add possible outputs, for instance to the view-space position to support displacement shaders, we can easily add more result functions. Executing shaders explicitly can be used for test purposes, but can also be used to sample points from textures or access other internal state accessible to shader programs.

7.6 Shader Programming Calls

Shader programming calls issue instructions into an open shader definition, and can be divided into declarations and operations.

Declarations control how the shader is implemented, define registers to hold intermediate results, and control how the shader interfaces with the outside world. Operations can be divided into categories: stack manipulation (for instance, pulling items out of the middle of the stack to support shared subexpressions, swapping items in the stack), arithmetic, comparisons (discontinuous functions that return 0 or 1, optionally with smooth transitions for antialiasing), logical operations, component manipulation, and register load/store. Operations cannot act on registers.

7.6.1 Parameter Declaration and Access

Parameters for a shader must be declared in the order they are expected to appear on the parameter stack. While it is not mandatory, for readability parameters should be declared before any other shader operations are specified.

Declaration is done with the following calls:

SMparam **smShaderDeclareParam**(SMuint n)
SMparam **smShaderDeclareColor**(SMuint n)
SMparam **smShaderDeclareTexCoord**(SMuint n)
SMparam **smShaderDeclareNormal**(SMuint n)
SMparam **smShaderDeclareCovector**(SMuint n)
SMparam **smShaderDeclarePlane**(SMuint n)
SMparam **smShaderDeclareTangent**(SMuint n)
SMparam **smShaderDeclareVector**(SMuint n)
SMparam **smShaderDeclarePoint**(SMuint n)

Declaration establishes both the type and the dimensionality n of the parameter. The type is used currently only for type-checking,[8] but the dimensionality determines the size of the tuple pushed on the stack for parameter access operations. The opaque identifier returned by each call should be assigned to a variable with an evocative name for later reference.

Parameters are accessed with the following calls, which push a copy of the parameter onto the stack.

smShaderGetParam(SMparam p)
smShaderGetColor(SMparam p)
smShaderGetTexCoord(SMparam p)
smShaderGetNormal(SMparam p)
smShaderGetCovector(SMparam p)
smShaderGetPlane(SMparam p)
smShaderGetTangent(SMparam p)
smShaderGetVector(SMparam p)
smShaderGetPoint(SMparam p)

[7] We are working on a mechanism to (a) permit sharing of textures and (b) reduce storage costs when textures are shared. In the meantime, this policy is safe, if not efficient.

[8] Type-checking is only done in paranoid mode. Fast mode assumes the parameters pushed and the number expected match. If not, undefined results should be expected.

7.6.2 Register Allocation

Registers hold untyped tuples. They are allocated with the following call; the parameter n is the dimensionality:

SMreg **smShaderAllocReg**(SMuint n)

As with parameters, the opaque identifier returned by this call should be assigned to a host language variable with a useful name. It will be required later to refer to the register.

To facilitate use of subscopes, the call

smShaderBeginBlock()

records the current register allocation state, and the call

smShaderEndBlock()

restores the register allocation state in a last-in, first-out fashion; in other words, it deallocates all registers allocated since the last nested **smShaderBeginBlock** call.

To store a value in a register, one of the following calls should be used:

smShaderStore(SMreg r)
smShaderStoreCopy(SMreg r)

The **smShaderStore** call pops the value from the front of the execution stack and puts the item in the indicated register. If the dimensionalities do not match an error is generated. The **smShaderStoreCopy** call just copies the item from the front of the stack into the register but does not pop it.

To retreive an item from a register, the following call pushes it onto the execution stack.

smShaderLoad(SMreg r)

The value stored in the register is not disturbed.

7.6.3 Stack Manipulation

For some stack manipulation instructions, items on the stack are referred to by integer offsets from the "front" of the stack. The 0th item is the item on the front of the stack, the 1st item is the next item on the stack, and so forth.

The drop operation discards k items from the front of the stack: The items dropped can be of any dimensionality, and need not all have the same dimensionality.

smShaderDrop(SMuint k)

The **smShaderPop** operation is just **smShaderDrop** with an implied argument of 1. It is provided for consistency with "pop" operations defined elsewhere in the API.

The **smShaderDup** operation pulls any item out of the stack and pushes it onto the front. The original is not deleted so the rest of the stack is not disturbed:

smShaderDup(SMuint k)

The **smShaderPush** operation is just **smShaderDup** with an implied operand of 0; it makes a copy of the item on the front of the stack.

7.6.4 Component Manipulation

Each item on the stack is a tuple of elements. Component manipulation instructions permit items to be constructed from and decomposed into elements.

Conceptually, items are pushed onto the stack *highest element first*. This is important to understand since several operations in this section treat the execution stack as a sequence of elements, not items. For instance, if we push items (a_0, a_1, a_2), (b_0, b_1), and (c_0, c_1, c_2, c_3) onto the execution stack, in that order, then considered as a sequence of elements the execution stack will look like

$$a_2, a_1, a_0, b_1, b_0, c_3, c_2, c_1, c_0$$

Elements are numbered starting at the "front" or extreme right. In this example, c_0 would have element index 0, c_1 would have element index 1, b_0 would have element index 4, a_1 would have element index 7, and so forth.

The extraction operation permits direct access to items, and so can be used to assemble a new item from arbitrary elements of other items. The output of an extract operation can have a different dimensionality from any of its inputs. In fact, the dimensionalities of the inputs are basically ignored, but you do have to consider them to compute the element indices. There are several versions of this operation, one general version that requires an array parameter and several others with a fixed number of integer parameters for dimensionality 1 through 4:

smShaderExtract(SMsizei k, SMuint* e)
smShaderExtract1(SMuint ...)
smShaderExtract2(SMuint ...)
smShaderExtract3(SMuint ...)
smShaderExtract4(SMuint ...)

The extract operation does not disturb the existing items on the stack but generates a new item by pulling out the indicated elements and pushing the newly constructed item onto the stack.

The **smShaderSwap** operation exchanges the 0th and 1st item on the stack. Swapping the two bottom elements does *not* change any of the element indices of items higher on the stack, since the total number of elements on the bottom of the stack for the first two items will remain the same.

The **smShaderRev** operation reverses the order of elements in an item. Like the swap operation, it does not change the element indices of elements higher in the stack.

The **smShaderJoin** and **smShaderCat** operations join two items together into one by concatenating their elements. The join operation does this without changing the element indices, but this makes it seem like the second operand (on the front of the stack) is concatenated to the *left* of the first operand. The concatenate operation **smShaderCat** is equivalent to a swap followed by a join and has semantics more consistent with the arithmetic operators.

Finally, the split operation, invoked with

smShaderSplit(SMuint k)

splits all the elements of the item on the front of the stack and makes two new items of lower dimensionality. In pseudocode:

```
POP $(a_0, a_1, \ldots a_{k-1}, a_k, \ldots a_{n-1})$
PUSH $(a_k, a_{k+1}, \ldots a_{n-1})$
PUSH $(a_0, a_1, \ldots a_{k-1})$
```

The results will be of dimensionality k and $n - k$ if the input is of dimensionality n. The parameter k must be greater than or equal to 1 and less than or equal to $n - 1$. Note that if k is 1 then a scalar will be placed onto the front of the stack. If k is equal to $n - 1$ then the second item on the stack will be a scalar. To be consistent with the ordering of elements, the item on the front of the stack will be drawn from the *lower*-indexed elements of the item that was originally on the front of the stack.

Each new item must have at least one element; the parameter k must vary between 1 and n.

This operation does not change the element indices of any element on the stack, it just reclassifies elements into different items. A join operation reverses a split.

7.6.5 Constants

Constants may be pushed onto the execution stack with the following calls:

smShaderParam{**1234**}{**sifd**}(T *p*...)
smShaderColor{**1234**}{**sifd**}(T *p*...)
smShaderTexCoord[{**1234**}**from**]{**1234**}{**sifd**}(T *p*...)
smShaderNormal{**3**}{**sifd**}(T *p*...)
smShaderCovector{**3**}{**sifd**}(T *p*...)
smShaderPlane{**34**}{**sifd**}(T *p*...)
smShaderTangent{**3**}{**sifd**}(T *p*...)
smShaderVector{**23**}{**sifd**}(T *p*...)
smShaderPoint[**4from**]{**234**}{**sifd**}(T *p*...)

These operations work exactly like the corresponding parameter specification functions, except they push items onto the shader execution stack, not the parameter stack. Transformations are also applied as with the parameter specification functions, but these transformations are applied using the transformation matrices in effect at the time the shader is specified, *not* when the shader is run.

If you want to transform something explicitly using the transformations in effect at the time the shader is executed, use the transformation access and matrix multiplication operators.

7.6.6 Environment Access

The shader may depend on information like model-space position, view-space position, view direction, and device-space position. These can be accessed with the following calls:

smShaderGetViewVec()
smShaderGetViewPos()
smShaderGetModelPos()
smShaderGetDevicePos()

The last actually returns a 3D item containing the integer pixel position relative to the origin at the lower-left corner of the output image, and the depth value used in for z-buffering. The depth alone can be accessed with

smShaderGetDepth()

Under perspective using our conventions (see **smFrustum**) the view vector and the view positions are negations of one another, but this is not necessarily the case. In an orthographic view, for instance, the view vector is constant. The **smShaderGetViewVec** call uses the projection matrix to compute the correct view vector, by first back-projecting the eye-point, then (if it is located at a finite position) subtracting the view-space position from it. It does *not* normalize the vector to unit length; the length of the view vector will be the distance to the eye from the point being rendered, in the viewing coordinate system. If the eye point is at infinity, as with an orthographic projection, then the view vector is set up to point in the correct direction but is not guaranteed to be of unit length.

7.6.7 Buffer Access

The current value of the target pixel in the destination buffer can be obtained with the following call:

smShaderGetBuffer()

This pushes value of the destination sample in the destination buffer onto the execution stack.

This permits a simple interface to compositing operations, a fixed set of which are usually implemented at the end of the fragment pipeline on current accelerators. However, use of this feature followed by complex processing of buffer contents may force multipass execution of the shader.

7.6.8 Texture Lookup

Texture accesses can be invoked by the following call:

smShaderLookup(SMtexture t)

This call takes the item off the front of the stack and uses it as a texture coordinate for a texture lookup in texture object t. The dimensionality of the item must match the dimensionality declared for the texture object, and the texture object must have been previously defined.

7.6.9 Arithmetic Operations

For arithmetic operations, the execution stack should be visualized as a list written left-to-right, with the front (top-of-stack) item on the extreme right. Binary operators go between the two items on the right of this list. For non-commutative operators, the item on the front of the stack (item position 0) is the *right* operand and the item at position 1 is the *left* operand.

Arithmetic operators are overloaded on the dimensionality of their arguments. Two-operand arithmetic operators generally operate in one of three modes:

Vector: $(\mathbf{R}^n, \mathbf{R}^n) \mapsto \mathbf{R}^n$.

In this case the two operands have the same dimensionality, and the operation is applied elementwise. Specifically, if $\oplus$ is the operator,

POP $(b_0, b_1, \ldots, b_{n-1})$
POP $(a_0, a_1, \ldots, a_{n-1})$
PUSH $((a_0 \oplus b_0), (a_1 \oplus b_1), \ldots, (a_{n-1} \oplus b_{n-1}))$

The result will have dimensionality n.

Left Scalar: $(\mathbf{R}^1, \mathbf{R}^n) \mapsto \mathbf{R}^n$.

In this case the first operand (the second item on the stack, or equivalently the one just behind the front of the stack) is a scalar, and is applied to all elements of the second vector as the left operand of the operation.

Specifically, if $\oplus$ is the operator,

POP (b_0)
POP $(a_0, a_1, \ldots, a_{n-1})$
PUSH $((a_0 \oplus b_0), (a_1 \oplus b_0), \ldots, (a_{n-1} \oplus b_0))$

The result will have dimensionality n.

Right Scalar: $(\mathbf{R}^n, \mathbf{R}^1) \mapsto \mathbf{R}^n$.

In this case the second operand (the front of the stack) is a scalar, and is applied to all elements of the second vector as the right operand of the operation, resulting in an output of the same dimensionality as the first operand.

Specifically, if $\oplus$ is the operator,

POP $(b_0, b_1, \ldots, b_{n-1})$
POP (a_0)
PUSH $((a_0 \oplus b_0), (a_0 \oplus b_1), \ldots, (a_0 \oplus b_{n-1}))$

The basic two-operand arithmetic operators are invoked by the following calls:

smShaderMult( )
smShaderDiv( )
smShaderAdd( )
smShaderSub( )

The following unary operations are also defined:

smShaderNeg( )
smShaderRecip( )
smShaderSqrt( )
smShaderRecipSqrt( )
smShaderSq( )
smShaderRecipSq( )
smShaderProjDiv( )
smShaderSum( )
smShaderProd( )

The **smShaderNeg** (negation), **smShaderSqrt** (square root), **smShaderRecipSqrt** (reciprocal square root), **smShaderSq** (square), **smShaderRecipSq** (reciprocal square) and **smShaderRecip** (reciprocal) calls operate on every element of a tuple separately, generating a new tuple of the same dimensionality.

The **smShaderProjDiv** call performs homogeneous normalization, multiplying every element of a tuple by the reciprocal of its last element. It generates a new tuple of dimensionality one less than its input tuple.

The **smShaderSum** call sums all elements in a tuple and generates a scalar. Likewise, **smShaderProd** forms the product of all elements in a tuple and outputs a scalar.

7.6.10 Transformations and Matrices

The transformation matrix in effect at the time the shader is executed can be pushed onto the stack with the following call:

smShaderGetMatrix(SMenum *m*)

where *m* is one of SM_MODELVIEW, SM_TEXTURE, or SM_PROJECTION. This call pushes 16 elements onto the stack in row-major order. If for some reason you only want a submatrix, use an **smShaderExtract** operation to extract the elements you want; dead-code removal will eliminate the extra data movement if it is possible.

To push the inverse or adjoint of a standard matrix, use one of the following calls, each of which also pushes 16 elements:

smShaderGetInvMatrix(SMenum *m*)
smShaderGetAdjMatrix(SMenum *m*)

Note that the inverse is the the adjoint divided by the determinant. The determinant can be obtained separately:

smShaderGetDetMatrix(SMenum *m*)

Using these calls will likely be cheaper than computing an inverse explictly. Note as well that these will be the matrices in effect at the time the last vertex in the current triangle was specified.

To multiply a vector by a matrix or a matrix by a vector, use

smShaderMultVecMatrix( )
smShaderMultMatrixVec( )

The first operation treats the left operand as a row vector; the second operation treats the right operand as a column vector. The

smShaderMultMatrix(SMuint *r*, SMuint *c*)

operation multiplies matrices together and outputs a new matrix of the given numbers of rows *r* and columns *c*. The inner dimensions of the factors are inferred from their sizes and the size specified for the output matrices. This works on matrices of any size.

Square matrices can be inverted. This operation uses Cramer's rule to generate a closed-form expression so it should not be used for matrices much larger than four dimensions:

smShaderInvMatrix( )

In some situations, i.e. for projective geometry operations where constant scale factors cancel, the adjoint can be used in place of the inverse:

smShaderAdjMatrix( )

Again, if you only want some elements of the inverse or the adjoint, use an extraction operation and the optimizer will get rid of the extra unused computations.

The determinant of a matrix can also be computed:

smShaderDetMatrix( )

Matrices equivalent to the transformation calls defined for the immediate mode interface are also supported. Like the calls that manipulate the immediate-mode matrix stacks, these calls generate a matrix and postmultiply it onto an existing matrix:

smShaderRotateMatrix( )
smShaderTranslateMatrix( )
smShaderScaleMatrix( )

The other parameters for these calls are located on the execution stack. For **smShaderRotateMatrix**, the stack should contain a 4×4 matrix, a scalar angle, and a 3D unit-length axis vector. These will be popped off and replaced with a transformed matrix. The **smShaderTranslateMatrix** call expects an $n \times n$ matrix and an $(n-1)$D vector, generates a translation matrix, and postmultiplies it by the given matrix. The **smShaderScaleMatrix** call expects an $n \times n$ matrix and an nD vector, generates a diagonal scale matrix, and postmultiplies it by the given matrix. If the scale vector is of length $n-1$, the last scale factor is taken to be 1. Note that **smShaderMult** can be used for uniform scaling.

Finally, the identity and an arbitrary constant matrix can be loaded onto the execution stack. These calls do not replace the item on the bottom of the stack, they push on new items:

smShaderLoadIdentityMatrix(
    SMuint *r*, SMuint *c*)
smShaderLoadMatrixd(
    SMuint *r*, SMuint *c*,
    SMmatrixd *m*)

Obviously matrix operations can potentially use a lot of resources, although in the end they just boil down to a sequence of arithmetic and extraction operations.

7.6.11 Geometric Operations

Geometric operations perform standard operations on vectors and points. While easily definable in terms of basic arithmetic and element manipulation operations defining them explicitly is helpful to both the programmer and possibly the compiler.

They are given as follows:

smShaderDot( )
smShaderCross( )
smShaderLen( )
smShaderSqLen( )
smShaderRecipLen( )
smShaderRecipSqLen( )
smShaderNorm( )
smShaderDist( )
smShaderSqDist( )
smShaderRecipDist( )
smShaderRecipSqDist( )

The **smShaderDot** operation computes a dot (or inner) product. It works on tuples of any dimensionality, and produces a scalar. The **smShaderCross** operation computes a cross product. It takes two three-tuples as input and produces a three-tuple as output.

The **smShaderLen** operation computes the Euclidean length of a tuple; it operates on tuples of any dimension. The **smShaderSqLen** operation computes the squared Euclidean length of a tuple; the **smShaderRecipLen** operation computes the reciprocal of the Euclidean length of a tuple; the **smShaderRecipSqLen** operation computes the reciprocal of the squared Euclidean length of a tuple. Like the **smShaderLen** operation, these work on tuples of any dimensionality.

The **smShaderNorm** operation normalizes a tuple to unit length, and works on tuples of any dimensionality. The **smShaderDist** operation computes the Euclidean distance between two tuples of any dimensionality, and **smShaderSqDist**, **smShaderRecipDist**, and **smShaderRecipSqDist** compute the square, reciprocal, and squared reciprocal of that value.

7.6.12 Standard Mathematical Functions

The SMASH API defines a number of standard mathematical functions. Ultimately, functions equivalent to most builtin RenderMan shading language functions will be defined to make the conversion of RenderMan shading programs to SMASH shaders easier. Note that most of these "built-ins" will probably be implemented as macros and so will not require extra support from the driver. They will be defined, however, in case an accelerator *would* want to provide direct support.

Lowering the input or output precision may be a good idea for some of these functions; performance may be improved. This will be particularly true if an implementation uses iteration or table lookup to evaluate functions.

To evaluate trigonometric functions, use the following calls, which work elementwise on tuples:

smShaderSin( )
smShaderArcSin( )
smShaderCos( )
smShaderArcCos( )
smShaderTan( )
smShaderArcTan( )
smShaderArcTan2( )

The **smShaderArcTan2** call takes two tuples of the same dimensionality as arguments. The elements of the first are interpreted as $r\sin(\theta)$ values, the second as $r\cos(\theta)$ values. Angles are always measured in radians.

7.6.13 Noise

Randomness is a critical component of many shaders. SMASH provides the following noise generation calls:

smShaderHash( )
smShaderVNoise( )
smShaderPNoise( )

The **smShaderHash** function computes a hash value between 0 and 1 based on each element of its input tuple. This value will appear random, but will be repeatable.

The **smShaderVNoise** function computes value noise between 0 and 1. At each point on a grid with unit spacing, a hash value will be computed, and then over the cell linear interpolation will be performed. The input is a multidimensional index into the grid. This noise is inexpensive, but will not look as nice as Perlin noise. Its cost will also grow exponentially with dimensionality at the number of vertices at the corners of each cell grows.

The **smShaderPNoise** function computes derivative-continuous Perlin noise bounded over $[0, 1]$ with a unit cell spacing. This will look nicer than value noise but may be more expensive to compute.

Different platforms may use different hash functions and noise generators, so application developers should not expect features generated by noise functions to be consistent across platforms.

7.6.14 Discontinuities

The following functions results in hard discontinuties, and so should be used with care:

smShaderMax( )
smShaderMin( )
smShaderClamp( )
smShaderAbs( )
smShaderSgn( )
smShaderStep( )
smShaderFloor( )
smShaderCeil( )
smShaderSelect( )

The **smShaderMax** operation compares two tuples elementwise and keeps the larger of each element. If a scalar and a tuple are compared, the scalar is first replicated to the same dimensionality as the tuple. The **smShaderMin** operation is similar, but keeps the smaller of each element. The **smShaderAbs** operation computes the elementwise absolute value of each element of a tuple. The **smShaderClamp** operation limits each element in its input to the range $[0, 1]$. The **smShaderSgn** operation computes the elementwise signum function of each element of a tuple (i.e. the output is -11 if the element was negative, zero if it was zero, and 1 if it was positive). The **smShaderStep** function is a step function, returning zero for negative or zero values and 1 for positive values. The **smShaderFloor** function finds the largest integer less than or equal to each element of its input. The **smShaderCeil** function finds the smallest integer greater than or equal to each element of its input.

Finally, the **smShaderSelect** function is used as a kind of "if statement". It takes three arguments: two items of the same dimensionality and a scalar. If the scalar is negative or zero, the first item is taken, otherwise the second item is taken.

Versions of the previous functions with soft transitions are also provided, and are important for writing antialiased shaders. They all take an additional scalar argument on the stack which should be the width of the transition region. The exact functions used are implementation-dependent but should be derivative continuous:

smShaderSMax( )
smShaderSMin( )
smShaderSClamp( )
smShaderSAbs( )
smShaderSSgn( )
smShaderSStep( )
smShaderSFloor( )
smShaderSCeil( )
smShaderSSelect( )

7.6.15 Discarding Fragments

The following calls consume a single scalar from the execution stack:

smShaderRejectOpen()
smShaderRejectClosed()

For **smShaderRejectOpen**, if that value is positive and greater than or equal to zero, then the fragment is retained. Otherwise, if the value is negative, the fragment is rejected. For **smShaderRejectClosed**, if that value is strictly positive, then the fragment is retained. Otherwise, if the value is negative or zero, the fragment is rejected.

These operations always executes at the highest frequency level; if you use them inside a subshader, they will be "pushed down" to a finer level. Multiple rejections are permitted in a shader. A fragment is rejected if *any* rejection test passes.

7.6.16 Indicating Results

Every shader has to call the function that sets the output color:

smShaderSetColor()

The input tuple can be 1, 2, 3, or 4 dimensional. A 1D item will be interpreted as a luminance and the alpha will be set to 1. A 2D item will be interpreted as a luminance plus alpha. A 3D item will be interpreted as an opaque RGB color (alpha of 1). A 4D item will be interpreted as an RGBA color.

7.7 Subshaders

Often parts of a shader can be evaluated at low resolution and interpolated, and when this is possible, it will generally be a performance win. The simplest example is the evaluation of "smooth" parts of a shader at vertices, such as the diffuse irradiance. The results of these per-vertex evaluations can then be interpolated to fragments, where the shader computation can be completed using the rest of the shader.

The mechanism SMASH uses to specify subshaders permits a shader first to be written at the finest resolution level available and tested. Then, to identify a part of the shader program which can be evaluated at a lower resolution, wrap the desired subexpressions in the following calls:

smBeginSubShader(SMenum *level*)
smEndSubShader()

Currently *level* must be one of

SM_CONSTANT
SM_PRIMITIVE
SM_VERTEX
SM_SUBVERTEX
SM_FRAGMENT
SM_SAMPLE

Constant shaders are defined and evaluated once at shader definition time. Primitive shaders are defined for every **smBegin**/**smEnd** block. Vertex shaders are evaluated at each vertex of a primitive. Subvertex shaders are evaluated at the vertices of primitives that are subdivided internally (i.e. subdivision surfaces, displacement maps). Fragment shaders are evaluated at every pixel fragment. Finally, the finest level of detail may be the sample, in the case of multisampled antialiasing. On systems without multisampling the fragment and sample levels will be identical. By default, shaders are evaluated at the finest possible level.

Subshaders cannot depend on any result computed at a finer level of detail, but otherwise the evaluation semantics are unchanged. Note that to implement SMASH shaders in their full generality, *all* resolution levels need to support identical sets of operations, including, for instance, texture lookups at vertices. While it would be possible to limit the operations that can be used at each level, and such shaders would be forward-compatible with more general systems, such restrictions are undesirable as they could be a source of portability problems and complicate programming.

The syntax given above for subshaders can be extended to other purposes. For instance, part of a shader could be identified for automatic approximation using a particular technique, or part of a shader could be identified for tabulation and storage in an unnamed texture object.

7.8 Precision Management

Precision is currently one of the major limitations in all multipass implementations of shaders. Single-byte-per-component color representations are (barely) adequate for representing the final result of a shader, but are often not sufficient for internal computations, and even the twelve bit color components supported on high-end machines can be limiting.

The SMASH API includes support for specifying the desired numerical precision and dynamic range of shader computations. The mechanism can be considered a way to "annotate" the shade tree specified by the shader program. Precision annotations have the status of hints, not absolute specifications.

The following calls define the current precision mode in an open shader:

smShaderPrecision(SMuint *b*)
smShaderExponentPrecision(SMuint *e*)
smShaderRadix(SMint *r*)
smShaderClamping(SMboolean *c*)
smShaderSigned(SMboolean *s*)

Each of these calls sets state variables which influence the precision modes of operators following them.

The **smShaderPrecision** call specifies the number of bits of precision. This will be the number of bits of precision in the mantissa for a floating-point number (not counting the implied initial 1), and overall for a fixed-point number.

The **smShaderExponentPrecision** call specifies the number of bits of precision in the exponent of a floating-point number; a value of 0 specifies a fixed-point number. An excess $2^{e-1} - 1$ encoding for the exponent should be assumed if the radix position is zero.

The **smShaderRadix** call indicates the position of the radix point for fixed-point numbers. It is treated as an additional exponent bias for floating point numbers. If the radix position is 0, the number can only represent fractions between $[-1, 1)$ if signed, and between $[0, 1)$ if unsigned. If the radix position is equal to the precision, the representation is an integer.

Clamping can be turned on with **smShaderClamping**; by default it is on. If it is turned off computed values exceeding the range of the result representation may give undefined results, except for operations involving reciprocation, which always clamp.

The **smShaderSigned** function specifies whether arithmetic should be signed or unsigned. If clamping is on the output *may* be clamped to zero if negative. Note: turning off signed arithmetic doesn't save any time or resources for the *current* operation, it just guarantees the output of an operation is unsigned, which may make operations that depend on a result simpler.

As each operation is specified the current numerical representation mode is applied to the output of that operator and tracked through the dependency graph indicated by the shader program. Conceptually operations will be performed at "infinite" precision and then mapped to the output representation. For instance, if you multiply two fixed-point 8-bit numbers, the result requires 16 bits to represent exactly. If the output has less precision, it will either be clamped or rounded, depending on the current radix and precision modes.

Precision modes can be changed at any point, on an operator-by-operator basis if necessary. However, continually flipping between floating point and fixed point representations, for instance, will likely not yield the best performance. Generally speaking, although floating-point can be specified, fixed-point may be much more efficient—in fact, specifying floating-point may require software simulation on some platforms.

Precision modes should be considered implementation hints; *an implementation is free to ignore them* and just provide a single representation with “adequate” precision and range, signed numbers, and clamping at large maximum and minimum values (the single-precision IEEE floating point representation and behaviour is to be considered adequate). An implementation can also provide greater range and precision in fixed-point computations than what has been requested.

This means that precision modes should *not* be relied upon. For instance, don’t try to convert numbers to integers by setting the precision equal to the radix position, use **smShaderFloor** instead. To clamp a number to zero, don’t fiddle with **smShaderSigned** and **smShaderClamping**, use **smShaderMax** to force a clamp against a constant zero value.

Specifying precision information is extremely tedious for the programmer, and a programmer is unlikely to discover an optimal arrangement of precision settings in complex situations.

One of our current research goals is the development of techniques for automatic precision and dynamic range selection, and supporting techniques for implementing multiple-precision signed arithemetic in a shader context. Currently, implementing signed multiple-precision arithmetic within certain real-time rendering algorithms is just barely feasible—it has been demonstrated for shadow map comparisons on the GeForce by Mark Kilgard, for example. However, some changes to existing hardware support, for instance a capability to save and take carries from the stencil planes, might make multiple-precision frame-buffer arithmetic a reasonable implementation alternative.

8 Fragments and Rasterization

Once fragments have been shaded, then they must be applied to the destination color and depth buffers, subject to depth and stencil tests. SMASH does not support compositing operators or the alpha test; that functionality is specified using the shader sub-API.

We assume z-buffering is supported as well as the standard OpenGL compositing operations. SMASH also guarantees that fragments will be written into the destination buffers in the same order that the primitives that generate them are specified to the interface.

Values in buffers are limited to the range $[0, 1]$ as in OpenGL and will have a limited set of precisions to conserve bandwidth. When the output is set in the fragment shader it is automatically clamped and quantized to the representable precision of the target buffer.

Despite these similarities, we propose the use of slightly different z-buffer and rasterization coverage conventions than OpenGL. The major design revisions which have been made for fragment generation and depth testing are as follows:

1. The view volume is a semi-infinite pyramidal cone, *not* a frustum.

2. The rasteriser is required to generate fragments for *all* parts of primitives visible in the view volume. By default, *no* hither or yon clipping is performed.

3. By default, the rasterizer saturates depth values rather than clipping fragments that fall outside the representable depth range.

4. A complementary-z device coordinate convention is used which can maximize depth precision over a specified near-far range when a floating-point or z/w rational depth representation is used.

Doing depth testing after fragment shading would permit shaders to manipulate depth values; this would be useful to simulate curved primitives such as spheres [42]. On the other hand, if the depth test is forced to come last in the pipeline we cannot cull shading operations by testing the depth *before* fragment shading, since a shader program might modify the depth. We have chosen not to support depth shaders, since culling fragments before shading is a potentially valuable optimization, depth shaders would probably be a rarely used feature, and displacement maps and programmable geometry assembly (which we do plan to support) will provide an alternative way to define curved geometric “primitives”.

8.1 Device Coordinates

The normalized device coordinate system of SMASH is defined as $[-1, 1] \times [-1, 1] \times [0, 1]$. The standard perspective map in SMASH maps the far plane in the viewing coordinate system to 0.0 in the normalized device coordinate system and maps the near plane in the viewing coordinate system to 1.0 in the normalized device coordinate system. In other words, unlike OpenGL, the convention will be that points closer to the eye will map to *larger* z values in the normalized device coordinate system.

8.2 Near and Far Clips

For compatibility with OpenGL, SMASH supports *optional* clips at $z = 0$ and $z = 1$ in device coordinates, to restrict the infinite view pyramid to a frustum. We specify an infinite viewing volume because the hither and yon clips are a major source of irritation when rendering virtual environments. Yet, it is possible to implement a simple and efficient rasterization algorithm (using hierarchical testing of edge equations) that does not require these clips, does not have a wrap-around-infinity problem, and furthermore eliminates the need to implement a clipper to the window edge [42, 47]. Also, a semi-infinite volume is more compatible with ray-casting, should that ever become part of real-time fragment generation.

The hither and yon clips are off by default. When mapping to older hardware, leaving these clips off will slow things down, since extra screen-aligned polygons will have to be drawn to fill in the clipped part of the polygon to simulate depth saturation. On newer hardware using an edge-equation tiling rasterizer (for instance), leaving the clipping off should ideally speed things up.

With depth saturation, z-buffer based depth testing will become ineffective once objects are closer to the eye than a certain minimum distance; likewise, with a floating-point depth representation, depths beyond the far plane will lose precision rapidly. However, appropriate fragments must still be rasterized, which means that algorithms like shadow volumes [35] should still function properly. Furthermore, if there is only one object close to the eye, probably the most usual case, there is no possibility for a depth-ordering conflict.

To turn hither and yon clips on and off, enable/disable the Boolean flags accessible through the following enumerated constants:

```
SM_NEAR_CLIP
SM_FAR_CLIP
```

The 0 and 1 depth values in the normalized device coordinate system must be representable, but an implementation is permitted to represent additional depth values outside this range, and will saturate fragment depths to the minimum and maximum representable values instead of clipping. A mode can also be enabled that forces saturation at 0 and 1 if strict depth-map consistency is desired between implementations; enable/disable the Boolean flag

```
SM_STRICT_DEPTH_SATURATION
```

which is false by default. Of course, if hither/yon clipping is enabled, the value of this flag is moot.

8.3 Viewport Mapping

After mapping to the normalized device coordinate system, fragments are mapped to pixel coordinates in the current destination buffer with a viewport transformation, set up with the call

smViewport(
    SMint x_bo, SMint y_bo,
    SMuint w, SMuint h)

where x_{bo} and y_{bo} are the lower-left corner of the viewport rectangle, in buffer pixel coordinates, w is the width in pixels, and h is the height in pixels. If x_n and y_n are normalized device coordinates, the viewport mapping is given by

$$\begin{aligned} x_b &= w\left(\frac{x_n+1}{2}\right)+x_{bo} \\ y_b &= h\left(\frac{y_n+1}{2}\right)+y_{bo} \end{aligned}$$

where x_b and y_b are the final buffer coordinates of the fragment.

8.4 Projection and View Volume

To set up perspective, the following call multiplies the current matrix (usually the projection matrix) by a matrix representing a projective transformation:

smFrustum(
    SMdouble ℓ, SMdouble r,
    SMdouble b, SMdouble t,
    SMdouble n, SMdouble f)

The parameters ℓ and r are the positions of the left and right clipping planes, t and b are the positions of the top and bottom clipping planes, and the parameters n and f are the near and far plane distances (down the *negative* z axis in the viewing coordinate system, although these are *positive* distances). Although we will talk about near and far planes in the following, it should be emphasized again that *near and far plane clipping is optional in SMASH.*

The planes defining the viewing pyramid under this transformation will be given by the sets of points

Bottom: $\{[0;0;0],[r;b;-n],[\ell;b;-n]\}$.

Top: $\{[0;0;0],[\ell;t;-n],[r;t;-n]\}$.

Left: $\{[0;0;0],[\ell;b;-n],[\ell;t;-n]\}$.

Right: $\{[0;0;0],[r;t;-n],[r;b;-n]\}$.

Near (optional clip): $\{[\ell;b;-n],[\ell;t;-n],[r;t;-n],[r,b,-n]\}$.

Far (optional clip): $\{[\ell;b;-f],[r;b;-f],[r;t;-f],[\ell,t,-f]\}$.

using the convention that the direction of the interior is given by the right-hand rule. This results in the following plane equations, using the convention that a positive sign denotes the interior of the view volume:

Bottom: $[0,n,-b,0]$.

Top: $[0,-n,-t,0]$.

Left: $[n,0,-\ell,0]$.

Right: $[-n,0,-r,0]$.

Near (optional clip): $[0,0,-1,-n]$.

Far (optional clip): $[0,0,1,f]$.

The perspective matrix $\mathsf{P}=[p_{ij}]$ generated by the **smFrustum** call is given by

$$\mathsf{P} = \begin{bmatrix} \frac{n}{\ell-r} & 0 & \frac{\ell+r}{\ell-r} & 0 \\ 0 & \frac{n}{b-t} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{n}{n-f} & \frac{fn}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

If the view-space position $[x_v;y_v;z_v]$ is given by the homogeneous coordinates $[(w_vx_v);(w_vy_v);(w_vz_v);w_v]$, then this matrix results in the rational linear mapping

$$\begin{aligned} x_n &= \frac{n(w_vx_v)+(\ell+r)(w_vz_v)}{(\ell-r)(w_vz_v)} \\ y_n &= \frac{n(w_vy_v)+(b+t)(w_vz_v)}{(b-t)(w_vz_v)} \\ z_n &= \frac{n(w_vz_v)+nfw_v}{(n-f)(w_vz_v)} \end{aligned}$$

to the normalized device coordinates $[x_n;y_n;z_n]$.

The normalized device coordinate system values z_n are to be used directly as depth values. No mapping is supported to modify the depth range other than what can be accomplished with the projection matrix.

Another way to define the clipping planes above would be to map plane equations for the normalized device coordinate system boundaries at $[-1,1]\times[-1,1]\times[0,1]$ back through the inverse of the current projective matrix.

An implementation can (and probably should) test homogeneous vertex positions against these plane equations to cull triangles that do not intersect the view volume. This can be done in model space as well if we map the plane equations back through MP rather than P. An additional eye-plane ($z_v=0$) test will also be useful if near-plane clipping is disabled; the plane equation for this in model space can be derived from the eye position and view direction.

The call

smPerspective(
    SMdouble θ, SMdouble a,
    SMdouble n, SMdouble f)

where θ is the field of view angle, a is the aspect ratio (width over height), n is the near-plane distance, and f is the far-plane distance, is a special case (axis-aligned symmetric perspective) convenience function which simply calls **smFrustum** with appropriate arguments.

8.5 Depth Representation

Under the complementary-z convention, the eye will be located at positive infinity. Infinitely far away objects in the viewing coordinate system will be mapped towards negative infinity, although under a rational (projective) mapping large view-space depths will actually converge to a finite negative value.

Over the $[0, 1]$ device coordinate range, if floating-point or rational numbers are used to represent depth, this has the effect of compensating for the the loss of precision at the far plane due to the perspective projection, since such representations have greater precision close to zero [32]. Comparisons on floating-point values are inexpensive—if stored in the right way, with the exponent in the high-order bits as a biased number, sign-magnitude integer comparison suffices. This "integer" is in fact a (signed) scaled piecewise linear approximation to the logarithm of the value represented by the floating-point number [6]. Therefore, we can actually do depth comparisons using integer arithmetic and still get the right answer, while still supporting readback of the depth buffer as floating-point numbers without any conversion cost. Comparisons between rational numbers requires two multiplies, but we can potentially avoid a division when rasterizing.

8.6 Depth Tests

To avoid confusion between OpenGL code and SMASH code given the new SMASH device coordinate conventions, new SMASH names are used for depth tests that refers to the geometrical situation instead of the representation, following the complementary z conventions:

```
SM_CLOSER
SM_CLOSER_OR_EQUAL
SM_FARTHER
SM_FARTHER_OR_EQUAL
```

If objects closest to the eye are rendered last, as in Painter's algorithm, depth order will be preserved either with depth testing turned off or with the `CLOSER_OR_EQUAL` depth test.

For most implementations memory bandwidth is better utilized if objects farthest from the eye are rendered last, since more objects will fail the depth test, which will cause more fragments to be discarded before updating the color and depth buffers, and possibly even before performing shading. Occlusion-culling optimizations also depend on an approximate front-to-back ordering. To maximize performance, an application should generally try to render objects closest to the eye first, using the `CLOSER` test.

8.7 User-Defined Clipping

SMASH does *not* have an API to define additional user-defined clipping planes, nor does it have an alpha test. This functionality is not required in the base API since it can be programmed as needed using an appropriate shader and the rejection operator.

The parameter specification sub-API permits the specification of plane equations. These plane equations will be transformed into the viewing coordinate system if they are specified with the modelview matrix stack active. The elimination of fragments on one side of the clipping plane can be performed using an **smShaderDot** shader operation to evaluate the plane equation against the homogenous view-space position and one of **smShaderRejectOpen** or **smShaderRejectClosed** to reject appropriate fragments.

More complex clipping regions, such as spheres and cylinders, or any other region that can be expressed in implicit form, can also easily be programmed.

Clipping in this way does not reduce the workload of the fragment shader if the semantics of the SMASH pipeline are interpreted literally, since the rasterizer would still generate fragments. On the other hand, the driver shader compiler could scan for shader expressions that set up clips according to this idiom and set up clipping equations in the rasterizers as an internal optimization.

9 Implementation

In this section we consider various strategies for the high-performance implementation of the feature set described for SMASH. First, we consider software implementation, and describe how the base software prototype for SMASH is being implemented and how a higher-performance software implementation could be obtained. Then, we survey some possible hardware acceleration architectures and present candidate designs for single-pass shading using multithreading and reconfigurable computing, and the global architectural consequences of these approaches.

9.1 Base Software Implementation

Our base software implementation is meant to be as portable as possible while still being reasonably efficient. It can be used in an interactive mode after setting up an appropriate OpenGL context. Currently our implementation only uses OpenGL for displaying completed images. For maximum portability, it can be compiled so it can run without an OpenGL backend, in which case frame-buffer readback should be used to access the completed images.

After a shader DAG is defined using the shader sub-API, it is optimized, then compiled down to a virtual machine language byte-code, which is interpreted. The interpretor evaluates several shader threads in "parallel" over batches of fragments, i.e. by doing one instruction at a time over multiple shader threads, so that its overhead is low and reasonable CPU utilization is obtained.

To construct the DAG, we actually "execute" the shader symbolically during definition by maintaining pointers to the roots of directed acyclic graphs (DAGs) describing shader expressions rather than actual values in the stacks and registers. This also permits us to determine if the shader program is well-formed and set appropriate error flags immediately after each shader API call.

When the shader definition is closed, we scan the DAGs rooted at the color and depth "results". This automatically results in dead-code removal, since dead code cannot be reached from the results. We then search for and eliminate common subexpressions using hash values computed for the DAGs rooted at each node, propagate constants (effectively evaluating constant parts of the shader), assign register slots for each intermediate value, and perform various other optimizations such as strength reduction.

We also track sharing of shared vertices and edges during primitive assembly to try and reduce recomputation. All primitives in SMASH are first reduced to triangles whose vertices have been transformed by the modelview matrix into the viewing coordinate system. After a reasonably large number of triangles have been assembled that share the same vertex and fragment shaders, a "rasterization packet" is set up. A rasterization packet consists of an array of vertices (whose view-space positions are described in homogeneous coordinates), associated parameter arrays, an array of pairs of vertex indices for each edge, and a triple of indices into the edge array for each triangle. Edge indices use a low-order bit to indicate if the edge orientation should be flipped for that triangle.

Our software rasterization module operates almost entirely in homogeneous coordinates. It performs the following operations on each rasterization packet:

1. Vertices are tested against the plane equations for the viewing volume. Outcodes are used to cull triangles which cannot be visible.

2. Vertices of all potentially visible triangles are marked using an array of bits so blocks of vertices that are unmarked (and so are not part of any potentially visible triangle) are easily identifiable.

3. Edge equations are computed for each potentially visible triangle, sharing computation among shared edges. Computing the edge equations for each edge of a triangle is equivalent to computing the adjoint of the matrix containing the vertex coordinates of each triangle.

4. Triangles in the packet are culled if they are backfacing and the backface cull mode is set. We have to do the test for backfacing triangles even if backface culling is off, since we have to make backfacing polygons visible by reversing the sense of their edge equation tests.[9]

   Triangle orientation is determined by testing the sign of the determinant of the vertex matrix. This requires relatively little additional work as we already have the adjoint of this matrix.

5. Vertex shaders are evaluated on each potentially visible (marked) vertex. A new parameter array is generated by the vertex shader using code generated by the compiler. This new array contains both the results of the vertex shader and parameters copied from the input parameter array for parameters that are simply to be passed down to the fragment shaders. This permits the vertex shader to edit out parameters consumed by the vertex shader.

   During compilation fragment shaders are rewritten to treat the results of vertex shaders as extra parameters by replaceing the code for the subshader with an appropriate parameter access instruction. The results of vertex shaders and passed-through parameters will be interpolated identically by the rasterizer. For simplicity, the rasterizer does not use any information about the internal stucture of the parameter array, it just treats it as a list of numbers to interpolate.

6. For all potentially visible triangles interpolation coefficients are set up for all parameters. This is done by multiplying the three numbers bound to each vertex for each corresponding position in the parameter arrays by the three-dimensional matrix given by the adjoint of the vertex matrix. A projective correction factor is also computed by transforming the reference vector $[1; 1; 1]$ by the same matrix. Note that computing the inverse of the vertex matrix by dividing the adjoint by the determinant is not necessary; the determinants would cancel anyways upon division by the interpolated projective correction factor.

7. Each potentially visible triangle is rasterized and the depth of each fragment is computed.

   We actually perform rasterization by testing potential fragment positions against quantized edge equations expressed in device coordinates. This is done incrementally and in a hierarchical fashion, using only integer arithmetic (mostly additions). This approach is efficient yet completely avoids the need for window-edge and hither and yon clipping having to generate new triangles, and can tolerate primitives crossing the eye plane, or even being semi-infinite.

   If hither and yon clipping are requested, each potential fragment position is tested against two additional plane equations.

8. If a fragment passes the depth test, the elements of the parameter array output by the vertex shading stage are interpolated, normalized by multiplying by the reciprocal of the interpolated projective correction factor, and a fragment with an interpolated parameter array is added to the fragment processing queue.

   Interpolation of each parameter is projectively correct, i.e. it uses rational linear interpolation [43]. We also structure the code so the long latency of computing the reciprocal of the projective correction factor (which we assume happens in parallel with the multiplications and additions required for interpolation) does not affect our throughput. Interpolation of parameters is currently done using floating-point arithmetic.

   Our rasterizer actually generates output fragments in blocks. Incomplete blocks are only generated when we have to flush the pipeline to switch shaders. In particular, fragments from different primitives are combined in the order they are submitted to the rasterizer. Once a block has been filled, it is sent down the pipeline to the fragment shader.

9. The fragment shader implementation *does not assume* that fragments are arranged in a regular grid; each fragment carries its device coordinates with it. However, the fragment shader's performance should be improved if samples are spatially coherent as texture cache coherency will be improved, so the rasterizer orders visits to tiles and samples to improve spatial coherence.

10. Shaded fragments are finally written to the framebuffer if they have not been rejected and if the stencil tests pass. Stenciling can be applied earlier in the pipeline if and only if the shader does not include rejection operators. Note that the depth tests takes place *before* interpolation and fragment shading to cull occluded fragments.

The shader execution module, used for both vertex and fragment shaders, is currently an interpretor. Each operation in a shader program applies to all vertices/fragments in a block at once. This has the advantage that each instruction invokes a reasonably large amount of work so the relative overhead of the interpretor (i.e. the time spent looking up the next instruction) is minimized. The shader interpretor implementation can be considered to be a simulaor for the multithreaded machine architecture discussed later.

The interpretor still suffers the overhead of copying data to/from memory rather than storing it in registers, althogh the data size is small enough that most of these memory accesses should refer to cached data. We also store the front of the stack in a local variable rather than in the stack array to encourage the compiler to keep it in registers.

There are two versions of the interpretor, one with unrolled loops that operates on constant-sized blocks, and one that handles variable-sized blocks. The later is only used when we flush the pipeline to change shaders.

We intend to optimize the implementation of each instruction in our software interpretor by exploiting the Intel SSE and AMD 3DNow! instructions. The rasterizer could also be optimized to use MMX instructions during the evaluation of the edge equations. Finally, we would ultimately like to compile shader programs on the fly to machine language to eliminate the overhead of the interpretor.

9.2 Multipass Fragment Shaders

Fragment shaders can be implemented by compiling them to multiple passes under OpenGL and using OpenGL's compositing operations to perform arithmetic. Basically, extra rendering buffers

[9] SMASH does not support different shaders for front and back surfaces. Two-pass rendering can simulate this if necessary.

and/or textures are used to hold intermediate results of the computation. Compositing and texturing operations can be used to implement each shader operation.

For full generality both color storage formats and the compositing and texturing operations must be extended to support high-precision signed arithmetic, but this is a relatively small change that does not require rearchitecting existing designs. In some contexts extended precision can be implemented even on current graphics accelerators. This implementation possibility, when used as a fall-back, permits a single API to cover machines both with and without single-pass fragment shading capabilities.

The chief disadvantage of the multipass approach is that even after optimization of the shader program, a large number of passes and a great deal of memory may be required, particularly since high precision may be required in the pbuffers, and high-precision operations may themselves require multiple passes. This limits the complexity of the shaders than can be implemented. Furthermore, if a scene has multiple shaders that each cover a small part of the display area, then memory (and bandwidth) utilization will be low. Memory and bandwidth utilization will also be low unless tight bounding boxes around the pixels affected by each shader can be built.

If a processor-enhanced memory is used, as in PixelFlow, similar problems arise, except the memory costs more and so the memory allocation problems are more acute. Optimizing memory usage was one of the primary preoccupations of the PixelFlow shader implementation [42, 41].

Therefore, the multipass approach will be most useful in domains such as industrial design, where the number of shaders in the scene is low and shader coherence is high. The multipass approach is also useful as a fall-back in case shaders get too complex for a single pass even on machines that support single-pass shaders.

9.3 Single-Pass Fragment Shaders

A direct implementation of the conceptual stack machine used in SMASH in hardware would be too slow to be useful. Normally, we would reconstruct the shader expression, optimize it, and then map it onto a different implementation architecture, such as an vector processor, a SIMD or MIMD array, or a multithreaded processor. Shading processors need not be general-purpose units; in particular, branches conditioned on data values are not necessary, and this can simplify implementation.

Since without branches each invocation of a shader takes the same amount of time, for certain architectures we can guarantee that shaders will complete in the order they are started, and so we do not have to reorder fragments to perform front-to-back or back-to-front rendering correctly, at least at the level of a single shading unit.

9.4 General Observations

Some general observations and assumptions can be made:

1. Shader temporal coherence can be assumed. This means that once we activate a single shader, we will probably use it for a while. Even on current systems it makes sense to reorder rendering so that all primitives that use a particular texture, for instance, are rendered together. Shader temporal coherence permits multiple invocations of the same shader program on different fragments to be combined.[10]

2. Shader spatial coherence can be arranged. There are some caveats here. Shader spatial coherence will be broken occasionally, and so this effect will not be as large as that for temporal coherence. However, we should be able to break shader evaluations into small “blocks” each with a reasonable amount of spatial coherence.

3. Shaders can vary widely in complexity and memory requirements. Therefore, an architectural approach that gracefully degrades in performance with increased shader complexity (as opposed to hitting a wall at a certain level of complexity) is desirable.

4. Texture access can assumed to be cached. While it will be reasonably fast most of the time, memory access may require a long multiple-cycle pipeline, and may occasionally require a delay to reload a cache line. Therefore the architecture has to be able to stall while waiting for a texture access to complete, and/or should be able to tolerate large memory access latencies (64 to 256 cycles) so a prefetch cache architecture can be used to hide the variable latency.

5. Fragments must be delivered to each pixel of the frame buffer in the same order that their source primitives are rendered. Therefore, any delays due to cache misses, etc. cannot have the effect of reordering the output of the fragment shader. In the future, techniques for order-independent transparency may make this unnecessary, but for now this constraint is a necessity.

9.5 Multithreaded SIMD Processor

In the multithreaded SIMD approach, shown in Figure 4, we basically implement the strategy described in Section 9.1 for the base software implementation, i.e. using an inner loop over threads rather than instructions. This approach leads to the per-instruction multithreaded processor [1, 2, 3].

In this architecture, instructions are fetched one at a time but are invoked n times for each thread, each time using a different subset of the register file. This is not really a general-purpose multithreaded processor in that we do not execute completely separate instruction streams, but instead use the same instruction stream over different data, as with a SIMD parallel processor.

A multithreaded SIMD processor like this maximizes utilization of functional units and can tolerate high latency, so processors can be relatively small but we can use deeply pipelined, high performance floating-point functional units. To scale up performance, we would then need to have many such processors running in parallel, possibly running different shader programs (which raises the issue of resynchronization of shaded fragments).

The register mapper unit shown in Figure 4 indexes the register file using both the instruction register and a *thread counter*. The register file is multi-banked and multi-ported to allow the external system to read out results and load new parameters in parallel with the fetches and writebacks required for computation. The register file doubles as a FIFO to buffer incoming parameters, so if we are executing small shader programs the “extra” space in the register file will not go to waste: it can be used to improve the elasticity of the rasterizer-to-shader communications network.

While one shader program is executing, we could also load the next shader program indicated by the **smNextShader** hint into the shader program memory. Since we only have to read instructions from the shader program memory once every n cycles there is plenty of bandwidth for this—in fact we could share a single instruction memory among a cluster of processors. Therefore, it should rarely be necessary to wait for a new shader program to load. However, as in the software implementation, we do need enough temporal coherency to be able to assemble an adequate number of fragments using each shader program.

[10] Unfortunately, reordering primitives by the shaders they use conflicts with front-to-back reordering to improve occlusion culling. . .

The advantage of this architecture is that pipeline latency can be hidden. If the number of threads exceeds the maximum pipeline latency, once we arrive back at the first thread, the previous instruction for that thread has completed and the result has been written back, ready to be used in the next instruction. For “small shaders” that can use a large number of threads latency can be *completely* hidden, eliminating the problem of scheduling instructions and simplifying compilation.

For functional units, we can expect latencies of up to 30 cycles for a high-precision pipelined divider/square root unit. Unfortunately, texture lookup units can potentially have much higher latencies, from 64 to 256 cycles. Such large latencies are required in texture lookup units to provide adequate elasticity in a prefeteched texture caching architecture and get performance levels consistent with current graphics accelerators [25]. The larger value of 256 would be expected in a large-scale NUMA distributed-memory architecture or for AGP texturing. In a single-chip solution with a single local memory 64 cycles is (currently) adequate, but this result was derived under the assumption that memory latency would be relatively constant. In the case we are contemplating, a large number of parallel shader units would be competing for the memory access port and so even with a single external memory latencies might be highly variable, as in the NUMA case. To be pessimistic we should take the larger value when evaluating our design.

Under this assumption, 256 threads would be required to completely hide latency. This would require a large register file, so we can expect shader units to be dominated by the space requirements of the registers and the texture caches, not by the functional units.

For complex shaders, we may have larger memory requirements and so may only be able to run a small number of threads. Fortunately, with more work to do we will have an easier time scheduling instructions even if some pipeline latency is exposed. Even with a smaller number of threads the apparent latency can be reduced. For instance, with a texture lookup latency of 256 cycles and 64 threads, each texture lookup would have an apparent latency of 4 instructions, an amount manageable with scheduling.

If it is desired at some time in the future this architecture could be extended to support data-dependent branching and loops simply by using multiple PC’s indexed by the thread counter, thereby extending it to a true simultaneous multithreaded architecture. In this case a natural barrier for ordering would be the “thread block”. By ensuring that fragments in a block can not overwrite each other and ordering arrival of thread blocks at the frame buffer, we can guarantee a framebuffer write order consistent with the primitive ordering. Even if data-dependent loops are present, spatial coherence may result in all the threads in a block taking a similar number of cycles to complete their execution, and so utilization should still be reasonably high. Dynamic scheduling in the processor could be used to improve performance [55] if execution times of threads are expected to vary greatly.

Another issue breaking the “deterministic execution time” property of shaders will be memory access cache misses. If the rasterizer takes care to group spatially close shader threads together as proposed, and if temporal coherence is strongly associated with spatial coherence, then a stall on one thread will likely mean others would also stall. Under these conditions, all threads can share a single texture cache and a miss can stall *all* threads to maintain synchronicity. The whole point of the large lookup latencies noted above is to reduce the miss rate to under 1%, so stalls should not be frequent.

In summary, a simplified multithreaded architecture could potentially run at full CPU utilization for both large and small shader programs, with graceful degradation for larger shaders. However, each thread would run slower so we would have to have many shader processors running in parallel to get good performance. The size of the register file gives an upper bound on shader complexity but can be made large, as it will double as a FIFO to buffer fragment packets.

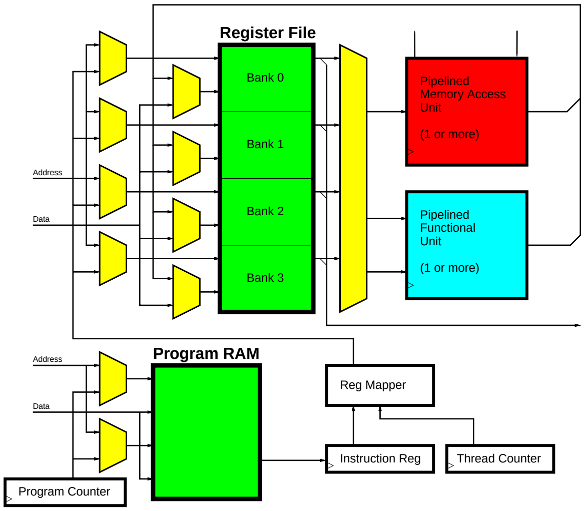

Figure 4: *The multithreaded processor implementation approach uses deeply pipelined functional units (which may be able to do multiple operations in parallel as well) and a single large register file. Only a single program counter and instruction memory is necessary since all threads are executing the same program in lockstep. However, different threads need to access different subsets of the register file.*

9.6 Reconfigurable Computing

A pipelined expression evaluator can also be implemented using an array of reconfigurable logic [13]. Such an implementation could potentially have very high performance and would be completely insensitive to texture lookup latency.

This architecture is diagrammed in Figures 5, 6, and 7.

A simple reconfigurable logic element (the internals of which are shown in Figure 6) is replicated many times to make a large mesh of processing elements. In the example shown here, a very simple element is used that maps three inputs to three outputs using lookup tables to implement three arbitrary Boolean functions. Each element also contains three edge-triggered registers so the entire mesh forms a large configurable pipeline.

With appropriate configuration data each element can implement a single-bit adder, a single-bit conditional adder (useful for implementing multipliers and dividers), a shifter (useful for implementing floating-point operations), or any other necessary logic. Elements can also be used to route data from one place in the mesh to another, to encode constants, and to implement small lookup tables (which can possibly be used in place of small textures, or to help implement functions such as square root).

To implement a pipelined expression evaluator, first macro cells for larger units of functionality (for instance, pipelined adders and multipliers) are designed, then appropriate combinations are concatenated with any necessary routing. This is a relatively simple process for the design shown here and so would satisfy our desire for reasonably efficient run-time metaprogramming.

To complete the design, the expression evaluator would be combined with pipelined texture lookup units, as shown in Figure 7. In this diagram, the expression pipeline flows data from left to right,

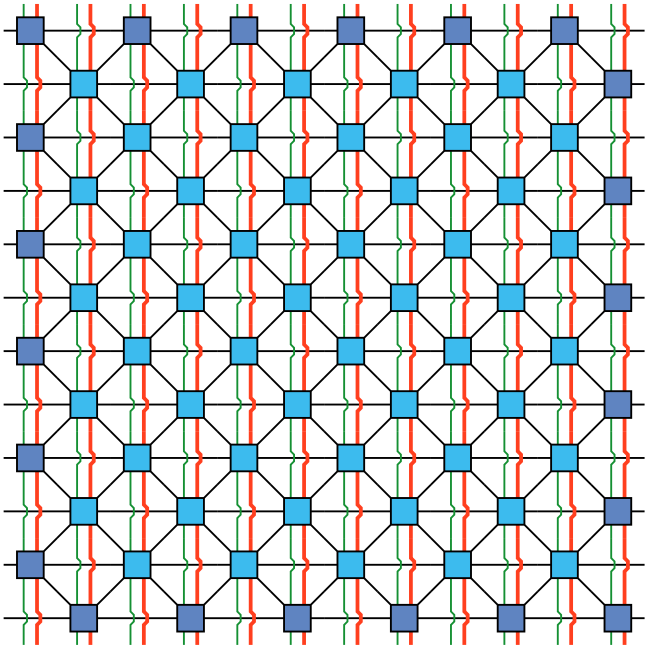

Figure 5: *A reconfigurable mesh pipelined expression evaluator. Data flows from left to right, reconfiguration information flows from top to bottom. The internals of each unit in the mesh are shown in Figure 6.*

while data in the texture lookup units flow from right to left.

To compile a shader expression, it would be first broken into basic "evaluation" blocks between texture lookups. Each such subexpression would be compiled into a separate pipelined evaluator and packed into the evaluation block. To execute a shader, parameters would be pipelined in from the left, would go through the expression evaluator, back through the texture units from right to left, through the expression evaluator again, etc. as many times as necessary to completely evaluate the expression. The result would be a completely pipelined shader evaluator that could generate a shader evaluation on every clock, as long as data (shader parameters) could be fed to it at that rate. The system can also process and accept data of arbitrary precision.

While easy to understand, there are several problems with the simple architecture shown here:

- Single-bit mesh elements require a large amount of configuration data for a small amount of functionality. It would probably be best to use larger, more complex mesh elements, such as a multibit ALU (perhaps even including a multiplier) combined with shifting and routing elements [18, 19]. Providing more flexible routing, so data would not always have to flow in one direction, could also lead to better utilization of the array and the memory access units.

- For implementing shaders in a graphics system, high-bandwidth reconfiguration is required. The system shown here would require a large number of cycles to reload reconfiguration data, which would be a problem if shader temporal coherency was low. Many reconfigurable systems are not designed to permit rapid, run-time reconfiguration.

  One option would be to reuse the data lines as ports over which reconfiguration data could also be streamed, using a pair of shared mode lines to switch elements between **Execution**, **Forward**, **Load**, and **Reset** states. The **Execution** state would use the loaded configuration data; the **Forward** state would send streaming configuration data onto the next element using a fixed communications arrangement (such as left-to-right), the **Load** state would place the incoming configuration data into memory, and finally the **Reset** state would clear the configuration data and set up the element to simply forward data.

  If the number of input and output bits per element could be matched to the amount of reconfiguration data in each element, it should be possible to pipeline in reconfiguration data between shader evaluations, and to only reprogram as many columns as needed to implement a shader, using resets on the other columns, making reconfiguration time proportional to shader complexity.

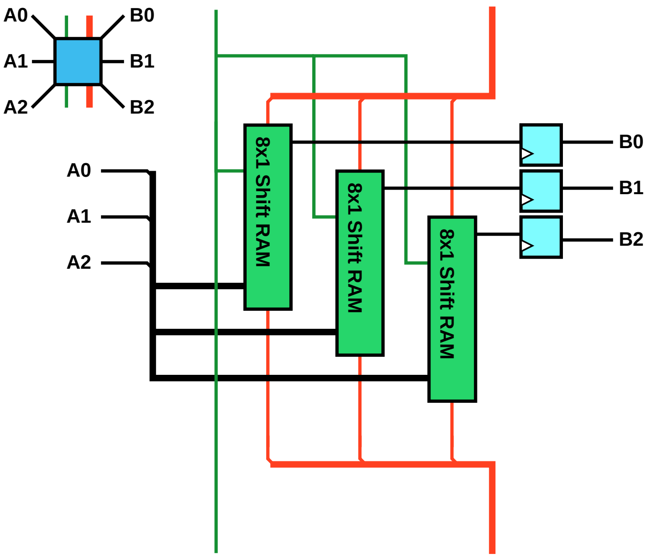

Figure 6: *The internals of a single unit in the mesh shown in Figure 5. Each unit is basically a lookup table that can implement three arbitrary boolean functions with three inputs each.*

- Another design challenge would be figuring out how to feed the shader unit with data at an adequate rate. Unfortunately, shader expressions will vary in the number of parameters they require, and so the input bandwidth required to feed the shading unit will vary. If not enough bandwidth is available on the input to the shader unit to get all the parameters to the shader unit in one cycle, they will have to be buffered over multiple cycles and bubbles will have to be inserted into the pipeline.

- The evaluator has a limited capacity; there will also be a limited number of texture lookup pipelines. Once these limits are exceeded there will be no option but to break the shader evaluation into multiple passes, although with each accomplishing a relatively large amount of work. Note that with a processor, we can use more cycles to perform more complex tasks, but there will be other resource limits (like the number of registers).

In a way this architecture is similar to the systems currently available for multitexturing (i.e. register combiners), but with a finer granularity.

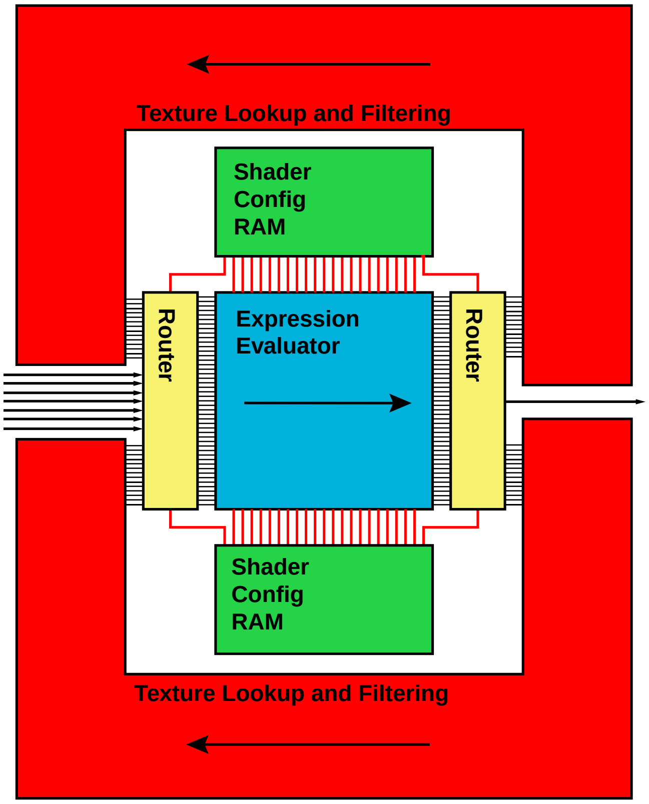

Figure 7: *A complete reconfigurable shader unit. The reconfigurable mesh is combined with pipelined texture lookup units which feed data from the output of the expression evaluator back to its input. Routers at the inputs and outputs of the expression evaluator provide flexibility in configuring the evaluator.*

9.7 Stream Processor

Another possibility for the implementation of a programmable graphics accelerator is the stream processor [44]. A stream processor is optimized for operating on homogeneous streams of data, which in the case of SMASH would be vertex/mode/parameter streams for geometry and fragment parameter streams for fragment shading.

The system described by Owen *et al.* consists of multiple pipelined functional units loosely bound by a high-speed communications network, and coordinated with a sequencer. It can be programmed in C++ using overloaded operators to describe connections between streams. The system has only been simulated but hardware is expected at the end of 2000.

This architecture would actually be capable of a completely reconfigurable graphics pipeline, and could also be used for video processing, simulation, and scientific computing. With the addition of support for fast dynamic reconfigurability, it would certainly be capable of implementing the conceptual architecture described here, perhaps with the addition of a specialized rasterization unit and a load-balancing network between parallel pipeline stages as in the Pomegranate architecture [11].

To summarize, there appear to be several ways to implement programmable shading with performance levels comparable to current graphics accelerators. In fact, if memory access becomes the chief bottleneck (as expected based on current trends), such systems could outperform traditional accelerator architectures if rendering algorithms can be devised that replace memory accesses with computation. However, much more work needs to be done to validate these architectures and adapt them to the needs of real-time graphics.

10 Metaprogramming Techniques

This section explores the use of the SMASH API as a backend to higher-level shading languages and APIs. It should be emphasized again that SMASH *is* a low-level API, and so our initial examples will look suspiciously like assembly language programs, although with virtualized stack and register resources. However, unlike string-based shader language interfaces (see the DX8 vertex shader proposal, for instance), the base SMASH API is type and syntax-checked at the time of compilation of the host language, and features of the host language can be used to manipulate shader programs conveniently and directly. In particular, modularity and control constructs in the host language can be "lifted" into the shader language, and this rapidly lets us build higher-level abstractions.

We will use as our running example an implementation of a two-term, single light source, separable reflectance model [28, 36]. We will assume that four hemicube maps are defined to hold the factors of the two-term separable BRDF approximation. This gives, for example, a good approximation of brushed metal using the Poulin/Fournier reflectance model [28, 48]. The variables `a`, `b`, `c`, and `d` will be predefined to hold the appropriate texture object identifiers. The values stored in `a`, `b` will be unsigned fixed-point values over $[0, 1]$, but will require a common scaling 3-color `aAB` to represent values over a wider range. The values stored in `c`, and `d` will also be over $[0, 1]$ but will represent *signed* factors over a potentially wide range, and so will require biases of `bC` and `bD` and another common scaling 3-color of `aCD`. The pointers `aAB`, `bC`, `bD`, and `aCD` will refer to appropriate predefined 3-element arrays. Because of this biasing and scaling, several operations will be required in the shader simply to "unpack" these textures.[11]

In the following sections we will implement the separable BRDF shader expression for a single point source. We will start with the simplest and lowest-level programming technique, the macro, and then work up to higher-level shader programming techniques.

10.1 Macros

Macros are implemented using host language functions that emit shader operations as side effects; they are lifts of the procedure modularity concept into the shader language from the host language. This lets us use the naming scope mechanism of the host language, with a little bit of help from the API, to strongly encapsulate shader modules.

The following conventions should be used for writing macros:

- Pass operand(s) and result(s) on the stack.
- Consume all parameters.
- Leave undisturbed any values higher on the stack than the macro's parameters.
- Use **smShaderBeginBlock** before issuing any other shader operations and issue **smShaderEndBlock** afterwards. The **smShaderBeginBlock** call pushes the current number of registers allocated onto a stack; calling **smShaderEndBlock** at the end of the macro definition restores the count of the number of registers allocated by popping it from the stack. This encapsulates register allocation and usage.

[11] We may modify the texture API in the future to include scaling and biasing as part of the definition of texture objects. For now, and for the purposes of this example, we will state these operations explicitly.

> We suggest using registers for storing intermediate results to enhance readability. In the example we also show how to name registers; by wrapping shader definitions in host language scope blocks that match the register allocation and deallocation blocks, register names can be limited in scope and so isolated from the rest of the program.

With these conventions, macros are syntactically indistinguishable from built-in API calls. This is an important and useful feature, as it permits macros to be used to enhance portability while providing a growth path for the API.

If a particular hardware vendor wants to provide a hook to some special feature of their hardware (say, bump-mapped environment maps, a new noise function, whatever), they should first write a "compatibility macro" using the standard operators guaranteed to be available on all implementations. Then, in the driver for their particular system, the standard instructions would be replaced with an appropriate hook into the special feature. User-level programs would not change, and would not even have to be recompiled if a dynamically-linked library is used for the required macro package. If precompiled shaders are used, this still works if compilation takes place upon installation; of course, every time the program runs, it should make sure it is using the same graphics subsystem for which it was installed.

On the other hand, if some set of utility macros comes into wide use, a hardware vendor can add explicit support for these macros to their hardware. In any case, all shaders can be portable across all hardware that supports the base shader instruction set.

Certain macro packages may have initializers that declare and initialize texture objects or other information. For instance, "math" macros to implement special functions via table lookup might be useful. SMASH will support a set of useful, "built-in" macros of this nature (as well as other useful, more specific macros, such as the orthonormalization macro used in this example). Because of the need to allocate hidden texture objects, the use of "specified" texture object identifiers (and shader identifiers) is *not* supported. A programmer *must* use system-allocated identifiers, and their internal structure is opaque. While shader and texture identifiers are guaranteed to take the same amount of space as a pointer, they are not to be considered equivalent to pointers (or integers) and an implementation should not depend on their exact value, range, or internal structure.

A shader defined using macros is shown in Figures 8 through 11. The function `FrameVector` defined in Figure 8 outputs shader instructions that generate texture coordinates for a separable BRDF approximation, given a unit-length light source vector, a unit-length normal, and a tangent on the stack. The normal and a tangent are *not* assumed orthonormal. This permits the use of a constant "global" tangent along the axis of a surface or near-surface of revolution, a handy technique to add tangents to objects that don't have them as part of their definition.[12]

The **smShaderDup** and **smShaderExtract** operations used in this example, as well as register store and load operations, will usually be zero cost after compilation and optimization, but they are convenient in this case to specify the dataflow. To reiterate, the SMASH API just specifies an expression tree which will be rebuilt inside the driver, then internal mapping onto the implementation architecture will take place, with its own internal optimization algorithms. For instance, for processor based shaders, register allocation will be performed to minimize the total storage required to implement the specified expression, and scheduling will reorder the operations to maximize usage of the processor's functional unit(s).

```
/** Compute coordinates of normalized vector relative to frame.
 * in:
 *   â: frame vector3 1; normalized to unit length
 *   b̂: frame vector3 2; normalized to unit length
 *   ĉ: frame vector3 3; normalized to unit length
 *   v⃗: source vector3
 * out:
 *   p⃗: surface texcoord3 (3D hemisphere map index)
 * assumes: frame vectors are orthonormal
 */
void
FrameVector ()  // â, b̂, ĉ, v⃗
{
  // Save register allocation state
  smShaderBeginBlock(); {

    // Allocate new registers
    SMreg v = smShaderAllocReg(3);
    SMreg a = smShaderAllocReg(3);
    SMreg b = smShaderAllocReg(3);
    SMreg c = smShaderAllocReg(3);

    // Put arguments into registers
    smShaderStore(v);       // â, b̂, ĉ
    smShaderStore(c);       // â, b̂
    smShaderStore(b);       // â
    smShaderStore(a);       // <empty>

    // Compute coordinates of v⃗ relative to â, b̂, ĉ
    smShaderLoad(c);        // ĉ
    smShaderLoad(v);        // ĉ, v⃗
    smShaderDot();          // (ĉ·v⃗)
    smShaderLoad(b);        // (ĉ·v⃗), b̂
    smShaderLoad(v);        // (ĉ·v⃗), b̂, v⃗
    smShaderDot();          // (ĉ·v⃗), (b̂·v⃗)
    smShaderLoad(a);        // (ĉ·v⃗), (b̂·v⃗), â
    smShaderLoad(v)         // (ĉ·v⃗), (b̂·v⃗), â, v⃗
    smShaderDot();          // (ĉ·v⃗), (b̂·v⃗), (â·v⃗)
    smShaderJoin();         // (ĉ·v⃗), ((â·v⃗), (b̂·v⃗))
    smShaderJoin();         // p⃗ = ((â·v⃗), (b̂·v⃗), (ĉ·v⃗))

  // Release registers
  } smShaderEndBlock();
} // p⃗
```

Figure 8: *Definition of a macro to project a vector against a frame. This is basically a* 3×3 *by* 3 *matrix-vector product, but with the matrix assembled from three vectors. Note that the driver can easily ignore any extraneous data movements, i.e. the movement of the arguments to and from registers.*

10.2 Textual Infix Expressions

A stack-based shading language can sometimes be inconvenient, particularly if complex dataflow requirements must be specified. The addition of registers to the programming model makes it unnecessary to use the stack for intermediate results. However, it is also relatively easy to define some conventions for a simple textual language for shader expressions that can be compiled on-the-fly to shader operations. By implementing some functions in a utility library to give string names to parameters, texture objects, and registers this can be made convenient, yet we can freely intermix such string-based infix expressions and macro or base API calls.

Here are the conventions and functions we propose for the `smu` utility library:

1. The **smuBeginTexture** function wraps **smBeginTexture** but, in addition, inserts a string name into a symbol table. Square brackets will be used after a texture object name in a string-based shader expression to indicate texture lookup.

[12] This is sadly a problem currently. Most modelling packages can't export per-vertex tangents, mostly because most 3D modelling languages have no way of specifying them.

```
/** Orthonormalize one vector against a normalized vector
 * in:
 *  $\vec{t}$: target vector; unnormalized
 *  $\hat{b}$: base vector; normalized to unit length
 * out:
 *  $\hat{v}$: orthonormalized target vector
 */
void
Orthonormalize ()  // $\vec{t}$, $\hat{b}$
{
  // Save register allocation state
  smShaderBeginBlock(); {

    // Allocate and name registers
    SMreg b = smShaderAllocReg(3);

    // Store base vector
    smShaderStore(b);       // $\vec{t}$

    // Orthonormalize
    smShaderDup(0);         // $\vec{t}$, $\vec{t}$
    smShaderLoad(b);        // $\vec{t}$, $\vec{t}$, $\hat{b}$
    smShaderDot();          // $\vec{t}$, $(\vec{t} \cdot \hat{b})$
    smShaderLoad(b);        // $\vec{t}$, $(\vec{t} \cdot \hat{b})$, $\hat{b}$
    smShaderMult();         // $\vec{t}$, $(\vec{t} \cdot \hat{b})\hat{b}$
    smShaderSub();          // $\vec{v} = \vec{t} - (\vec{t} \cdot \hat{b})\hat{b}$
    smShaderNorm();         // $\hat{v} = \vec{v}/|\vec{v}|$

  // Release registers
  } smShaderEndBlock();
} // $\hat{v}$
```

Figure 9: *Definition of a macro to orthonormalize one vector against another.*

2. The special names `%i` refer to the `i`th item on the stack.

3. The special names `#i` refer to the `i`th element on the stack.

4. The comma operator can be used to concatenate items. In particular, this permits the formation of items out of individual elements.

5. Parameters can be referred to by names defined in **smuShaderDeclare*** calls. These calls differ from their **smShaderDeclare*** namesakes only in that a string name is associated with the parameter.

6. Registers can also be referred to by names defined in a **smuShaderAllocReg** call, which differs from **smShaderAllocReg** only in that a string name is associated with the register. Assignments can also be made to registers using the "`=`" operator at the beginning of an expression; this executes a **smShaderStore** operation, so the result is *not* left on the stack after an assignment.

7. Arithmetic infix operators act on n-tuples using the same dimensionality rules as the associated operators **smShaderMult**, **smShaderDiv**, **smShaderAdd**, **smShaderSub**, and **smShaderNeg**. Other operators have the same name as those in the shader language proper but take extra arguments for their operands, use all lower case, and drop the `smShader` prefix. The vertical bar is used to indicate a dot product and the circumflex is used to indicate a cross product. Operator precedence is the same as in C for compatibility with operator-overloaded utility libraries in C++; for dot and cross product to work as expected, it is suggested that they always be enclosed in parentheses.

8. All names (i.e. for texture objects, parameters, and registers) share a common namespace. However, a name defined by the user will take preference over those of built-in functions. More precisely, built-in functions are defined in an outermost "system" scope, and names defined by the user are automatically in a name scope nested inside the system scope. The functions **smuShaderBeginBlock** and **smuShaderEndBlock** push and pop register allocation and environment frames and so create scopes and subscopes for names.

9. The functions **smuShaderAllocReg** and **smuShaderDeclare*** return identifiers just like **smShaderAllocReg** and **smShaderDeclare***. If you use *just* the infix operators you can ignore these return values. If they are retained they should be assigned to `SMreg` and `SMparam` variables with the same names as the strings used in the infix expressions.

Finally, the utility function

smuShaderExpr(`char*` *`string`*)

converts an expression given in the string argument into an appropriate sequence of base API calls to execute the operations specified in the string. An example is shown in Figure 12, using our running example of a separable BRDF reconstruction for two terms and one light source.

10.3 Textual Shading Languages

The above approach uses only a small part of a compiler, so macros and infix expressions can be freely mixed. A complete textual shading language could also be defined. Because of the common back-end compiler and extensive support for optimization expected in the driver, however, this can be a simple syntactic transformation. The transformation can be done on the fly, or using a separate program. Syntax analysis and transformation can be easily implemented using the yacc/lex or the bison/flex compiler-compiler systems, for example.

Following the initiative of the group at Stanford, a type system can be used to distinguish quantities to be computed at different levels of detail (i.e. vertices vs. fragments). The textual language should also have the capability to explicitly set precision requirements.

10.4 Object-oriented Toolkits

You can also lift object-oriented modularity constructs into the shader language. For instance, you can build a shader expression graph from object instances, and then provide a function to "walk" the graph in postfix order and emit appropriate base API calls. Garbage collection is essential in this case to avoid insanity—fortunately, since only DAGs are required, this can be implemented using a reference-counting "smart pointer" or "handle" class in C++.

The running example implementing the separable BRDF is repeated in Figure 13 using a very simple set of object constructors; the `SmExpr` classes are in fact reference-counting "smart pointer" classes that once initialized with a pointer to an object, start tracking references to those objects. Whenever the last reference is removed from an object, the smart pointer "releasing" the object automatically deletes it.

By wrapping object constructors in functions that take handles (smart pointers) and dynamically allocate new nodes but return such handles rather than pointers, we can simplify the syntax for defining each new expression node. Reference-counted garbage collection also permits the use of nested expressions, so we can avoid having to declare and name all the intermediate expression nodes. We can also use a namespace to simplify the names for the constructor functions and other parts of the SMASH API. Consider the example given in Figure 14.

Finally, if we use operator overloading for these constructor functions, we can use simple expressions to specify shading expressions. An example is given in Figure 15. Although it looks similar to the RenderMan shading language, this "shading language" differs from the use of a textual shading language since operator overloading is fully type and syntax checked in the host language at compile time, while we retain all the power of metaprogramming (i.e. programmer-controlled specialization). The only things really missing are specialized constructs such as RenderMan's `illuminate`, but we can easily use alternative approaches to support equivalent functionality. In the case of `illuminate`, a utility library function could be implemented that executes a (host language) `for` loop over light sources and that calls previously registered function pointers to invoke shader macros for each light, then issues shader instructions to accumulate the results. If C++ is used instead of C, such an approach is to be preferred over the use of the **smuShaderExpr** function.

The statements where we "wrap" constants and texture objects (using smart pointer subclasses that automatically invoke the appropriate constructor functions) could be omitted if we did this at the point of declaration of the constants and texture objects. Wrapping in the case of texture objects is necessary for the use of operator overloading of the square bracket operator for the `lookup` operation. We also add wrapping functionality to other subclasses of the smart pointer type to help declare parameters. Likewise, subshader and type attributes can be set using template parameters to the smart pointer classes. Registers are not required; the host language provides equivalent functionality through the use of variables, and the compiler can detect sharing and allocate registers internally to describe this sharing to the API.

The main point of all these programming examples is that several syntaxes and approaches to programming shaders are possible, but do *not* need to be built into the base API. Syntax is a type of user interface: a linguistic one. Several such user interfaces are possible for specifying shaders, and are appropriate for different kinds of users at different times. In fact, although we gave no examples, visual interfaces to programming shader specifications are also possible and for many users desirable.

By providing a basic, low-level API, SMASH does not force use of a particular high-level shader language but hopefully enables the straightforward implementation of a variety suited to different purposes and users. At the same time, a standard C++ utility library can be provided that gives nearly all the expressiveness of the RenderMan shading language.

11 Omitted Features

The SMASH API does not include many features included in, for instance, the RenderMan geometry format or shading language, or in OpenGL. There are two cases: features may have been omitted intentionally to improve performance, or features may be missing just because we haven't gotten around to implementing or specifying them yet. The later case is handled in Section 12; here we deal with intentional omissions.

Each of the major omissions is detailed below, along with the reason for leaving it out. In most cases the "missing" feature was left out intentionally to enhance performance in a hardware implementation. In some cases the feature can be partially simulated.

Built-in Lighting Model: Unlike OpenGL, SMASH does not have a built-in lighting model. However, a variety of lighting models, including the Phong lighting model, can be implemented in a utility library using the functionality provided by the shader sub-API.

Separate Light and Surface Shaders: SMASH does not explicitly support separate specification and compilation of surface and light shaders, it only supports "surface" shaders. The reason for this is that to get the best performance, surface and light shaders have to be combined into a single shader for compilation anyways. To get the same effect, use separate macros for surface and light shaders, then compile the appropriate combinations as you need them.

Looping and Selection Control Constructs: Selection must be implemented using a step function between two expressions, both of which will be, at least conceptually, evaluated. While this may seem wasteful, it is required anyways in SIMD and multipass implementations. A compiler can recognize such constructs and optimize out such evaluations if it is possible.

Looping within the shader is not supported at all, although one can write a loop in the host language to "unroll" parts of a shader. However, this only supports loops whose iteration limits are known at compile time—data-dependent iteration is *not* supported. For instance, you can't iterate a Newton recurrence until some convergence test passes, you must use a fixed number of iterations.

This is the most severe restriction in SMASH shaders—they are not Turing-complete. This limitation is intentional, to permit a wider range of efficient implementation possibilities. Shader expressions as defined in SMASH execute in a fixed finite amount of time, always use the same resources in the same order, can be statically scheduled, can be pipelined, etc. It should be noted that existing implementations of real-time shaders, i.e. SIMD or multipass approaches, don't take good advantage of data-dependent loops anyways: they must do as many iterations as the worst-case shader invocation.

Looping over Light Sources: This limitation is related to the two limitations above. If you want a shader that can handle four light sources, set up a compile-time loop to expand the appropriate part(s) of your shader program four times. You may have to expand both the parameter definition part and the lighting evaluation parts of your shaders.

This means you need a different version of the compiled shader for different numbers of light sources, but you'd want to "specialize" your shaders in this manner anyways for efficiency.[13]

Named Parameters: The parameter binding mechanism in SMASH only permits the definition of which parameters are constant over a primitive to vary by the position of the break-point in the sequence of parameters, i.e. by which are stated before the **smBegin** call and which after. This seems limiting, but consider that you can recompile shaders at any time to use different orderings of parameters. The fixed ordering of parameters permits a fixed schedule and ordering to be established for delivering parameters from the interpolator/rasterizer to the shading unit in a hardware implementation.

Default Values for Parameters: Default parameter values, if permitted and if used, would be constants that can be exploited during optimization. If you really aren't going to use a parameter, you should replace access to that parameter with a

[13] The only problem with this approach is code bloating due to unrolling. We may add some kind of "repeat" construct to SMASH later, but it's likely that such a construct would still be limited to a fixed number of iterations.

constant in your shader program and recompile it to get better performance. With a little metaprogramming you can easily compile versions of a shader with different parameters held constant as needed.

Arrays: Texture maps can be used as read-only arrays when necessary, with integer indexing possible using appropriate clamping and interpolation modes. However, textures cannot be (directly) written by a shader program at run-time, and register and stack addresses can only be specified as constants at compile time. There is no equivalent structure for writing into an array. This is so the resulting shader has a fixed dataflow graph, again so it can map onto a wider range of implementations.

You can use multiple passes to simulate writing into an array: render into a buffer, load the result into a texture, and render again with another shader.

Subfunctions: There is no mechanism for separate compilation of shader subfunctions. However, the modularity constructs of the host language can be lifted into the shader language, and at any rate, to get maximum performance, a shader compiler would normally want to inline subfunctions and analyse the shader as a whole.[14]

In general, run-time flexibility has been given up for performance in the design of SMASH, but you can often make up the difference with compile-time metaprogramming.

Some further limitations include lack of volume shaders, raytracing capabilities, etc. These limitations and the ones noted above may be overcome in time, and in the meantime you can simulate the effect, but you may have to come up with an approach more suited to the capabilities provided by SMASH.

12 Future Work

Currently, SMASH does not address some crucial issues:

1. SMASH currently only supports a limited subset of the full functionality of OpenGL. Several important features in OpenGL do not yet appear in SMASH. We hope to address these issues over time.

2. It is possible but unlikely that in the short term floating-point arithmetic will be widely supported for per-pixel computations.

   Automating the derivation of precision settings for intermediate values is currently one of our highest priorities. While setting these by hand is possible, it is extremely tedious and the chance that non-optimal settings will be chosen is high.

3. Shaders do not necessarily have to be implemented exactly as specified. What is important is that the results “look” right. As an adjunct to precision analysis we plan to look at ways to automatically or semi-automatically simplify shaders by generating approximations for parts of shaders that are only needed to a low precision, driving the entire process with perceptual metrics applied to the output of the shader.

   Prime candidates for this are antialiasing threshold generators, the derivation of which can again be automated with automatic differentiation.

4. A single-pass shader could in theory be used for image processing operations. However, the current conceptual model of SMASH cannot be used to specify, for instance, convolution-type operators unless the frame buffer is first copied to a texture map. An image-processing mode in which a shader could be applied to the frame buffer, with support for access to neighboring pixels, would be interesting and useful.

5. Support for procedural geometry would be a useful adjunct to shader support. Vertex shaders can be used to implement vertex blending and limited displacement mapping, but structural manipulation or dynamic refinement of meshes is not yet possible. We plan to add support for programmable geometry, starting with displacement shaders, but also need to further develop the API for programmable triangle assembly. Standard assemblers for advanced geometry processing, such as compression and view-dependent tesselation, would be very interesting.

13 Conclusions

SMASH, when completed, will hopefully provide a useful target for the extension of graphics APIs to programmable graphics subsystems, or at least will stimulate discussion about the forms such an API should take.

Our main focus so far has been the provision of a useful and powerful programmable shading sub-API. In addition to an API for specifying shaders, we have also presented a simple but flexible immediate-mode mechanism for binding shader parameters to primitives and vertices.

Possible hardware implementations of an accelerator for SMASH have been sketched, including a multithreaded processor and a reconfigurable computing architecture. We have also shown how metaprogramming can enhance the usability of the relatively simple API outlined here.

Interactive computer graphics has been very fortunate in having an API, OpenGL, which mapped onto a wide range of implementations. However, programmability provides an opportunity to make a radical change in the structure of graphics APIs and conceptual architectures, and such a change may be necessary if we are to take full advantage of the potential of programmability. This is an opportunity to rearchitect graphics accelerator interfaces with the goal of making them substantially more powerful yet simpler and easier to use.

Acknowledgements

We would like to acknowledge the help and advice of members of the Computer Graphics Lab at the University of Waterloo, the Computer Graphics Group at Universität Erlangen, and the Computer Graphics Group of the Max-Planck-Institut für Informatik in Saarbrücken, Germany, especially Jan Kautz and Wolfgang Heidrich.

Vincent Ma helped to implement version 0.1 of SMASH.

Several useful discussions with Mark Kilgard and David Kirk of NVIDIA, John C. Hart of Washington State University, Roger Allen of 3dfx, Mark Peercy, Marc Olano, Kurt Akeley and Jon Leech of SGI, Michael Deering of Sun Microsystems, Juan Guardado of Matrox, Pat Hanrahhan, Bill Mark, and Kekoa Proudfoot of Stanford, and Steve Morein of ATI contributed to my thoughts on the matter of programmable graphics APIs.

This research was sponsored by research grants from NSERC, the National Science and Engineering Research Council of Canada, and by a grant from CITO, the Centre for Information Technology

[14] Again, this can lead to code bloat, so we may add a mechanism for separate compilation in the future.

of Ontario. We also gratefully acknowledge the donation of graphics boards from NVIDIA.

References

[1] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *International Symposium on Computer Architecture*, pages 104–114, 1990.

[2] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *International Conference on Supercomputing*, pages 226–236, 1998.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, 1990.

[4] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan-Kaufmann, 2000.

[5] James Blinn. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. Morgan-Kaufmann, 1996.

[6] James Blinn. Jim blinn's corner: Floating-point tricks. *IEEE Computer Graphics & Applications*, 17(4), July–August 1997.

[7] B. Cabral, M. Olano, and P. Nemec. Reflection Space Image Based Rendering. In *Proc. ACM SIGGRAPH*, pages 165–170, 1999.

[8] P. Diefenbach. *Pipeline Rendering: Interaction and Realism throught Hardware-Based Multi-pass Rendering*. PhD thesis, Department of Computer and Information Science, 1996.

[9] P. Diefenbach and N. Balder. Multi-Pass Pipeline Rendering: Realism for Dynamic Environments. In *SIGGRAPH Symp. on Interactive 3D Graphics*, pages 59–70, April 1997.

[10] Frédo Durand and Julie Dorsey. Interactive Tone Mapping. In *Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)*, pages 219–230. Springer, 2000.

[11] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: A Fully Scalable Graphics Architecture. In *Proc. ACM SIGGRAPH*, page to appear, 2000.

[12] S. Gortler, R. Grzeszczuk, R. Szelinski, and M. Cohen. The Lumigraph. In *Proc. ACM SIGGRAPH*, pages 43–54, August 1996.

[13] Steven Guccione. *Programming Fine-Grained Reconfigurable Architectures*. PhD thesis, 1995.

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

[15] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proc. ACM SIGGRAPH*, pages 309–318, August 1990.

[16] Paul Haeberli and M. Segal. Texture mapping as A fundamental drawing primitive. In *Rendering Techniques '93 (Proc. Eurographics Workshop on Rendering)*, pages 259–266. Springer, June 1993.

[17] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Proc. ACM SIGGRAPH*, pages 289–298, August 1990.

[18] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Mapping Applications onto Reconfigurable Kress Arrays. In *Proc. Conference on Field Programmable Logic and Applications*, 1999.

[19] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Using the kressarray for configurable computing. In *Proceedings of SPIE, Conference on Configurable Computing: Technology and Applications*, pages 39–45, November 2–3 1998.

[20] W. Heidrich, J. Kautz, Ph. Slusallek, and H.-P. Seidel. Canned lightsources. In *Rendering Techniques '98 (Proc. Eurographics Workshop on Rendering)*. Springer, 1998.

[21] W. Heidrich and H.-P. Seidel. Realistic, hardware-accelerated shading and lighting. In *Proc. ACM SIGGRAPH*, August 1999. Accepted for publication. Preprint soon available from http://www9.informatik.uni-erlangen.de/Persons/Heidrich.

[22] W. Heidrich, Ph. Slusallek, and H.-P. Seidel. An image-based model for realistic lens systems in interactive computer graphics. In *Graphics Interface '97*, pages 68–75, 1997.

[23] W. Heidrich, R. Westermann, H.-P. Seidel, and Th. Ertl. Applications of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics*, 1999. Accepted for publication.

[24] Wolfgang Heidrich, Katya Daubert, Jan Kautz, and Hans-Peter Seidel. Illuminating Micro Geometry Based on Precomputed Visibility. In *Proc. ACM SIGGRAPH*, page to appear, July 2000.

[25] Homan Igehy. *Scalable Graphics Architectures: Interface & Texture*. PhD thesis, Computer Science Department, 2000.

[26] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The Design of a Parallel Graphics Interface. In *Proc. ACM SIGGRAPH*, pages 141–150, 1998.

[27] Jan Kautz. Hardware Rendering with Bidirectional Reflectances. Technical Report TR-99-02, Dept. Comp. Sci., U. of Waterloo, 1999.

[28] Jan Kautz and Michael D. McCool. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *Eurographics Rendering Workshop*, June 1999.

[29] Jan Kautz and Michael D. McCool. Approximation of Glossy Reflection with Prefiltered Environment Maps. In *Proc. Graphics Interface*, pages 119–126, May 2000.

[30] Jan Kautz, Pere-Pau Vázquez, Wolfgang Heidrich, and Hans-Peter Seidel. A Unified Approach to Prefiltered Environment Maps. In *Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)*, pages 185–196. Springer, 2000.

[31] A. Keller. Instant radiosity. In *Proc. ACM SIGGRAPH*, pages 49–56, August 1997.

[32] Eugene Lapidous and Guofang Jiao. Optimal depth buffer for low-cost graphics hardware. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 67–73, 1999.

[33] Anselmo Lastra, Steven Molnar, Marc Olano, and Yulan Wang. Real-Time Programmable Shading. In *Symposium on Interactive 3D Techniques*, pages 59–66, 207, 1995.

[34] M. Levoy and P. Hanrahan. Light field rendering. In *Proc. ACM SIGGRAPH*, pages 31–42, August 1996.

[35] M. McCool. Shadow Volume Reconstruction from Depth Maps. *ACM Trans. on Graphics*, 2000.

[36] Michael D. McCool and Wolfgang Heidrich. Texture Shaders. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 117–126, 1999.

[37] T. McReynolds, D. Blythe, B. Grantham, and S. Nelson. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 1998 Course Notes*, July 1998.

[38] Tomas Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters, 1999.

[39] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. In *Proc. ACM SIGGRAPH*, pages 231–240, July 1992.

[40] E. Ofek and A. Rappoport. Interactive reflections on curved objects. In *Proc. ACM SIGGRAPH*, pages 333–342, July 1998.

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

[42] Marc Olano. *A Programmable Pipeline for Graphics Hardware*. PhD thesis, Department of Computer Science, 1998.

[43] Marc Olano and Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Proceedings 1997 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 89–95, 1997.

[44] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon Rendering on a Stream Architecture. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware*, page to appear, 2000.

[45] Mark Peercy, Mark Olano, John Airey, and Jeff Ungar. Interactive Multi-Pass Programmable Shading. In *Proc. ACM SIGGRAPH*, July 2000. To appear.

[46] K. Perlin. An image synthesizer. In *Proc. ACM SIGGRAPH*, pages 287–296, July 1985.

[47] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proc. ACM SIGGRAPH*, pages 17–20, August 1988.

[48] P. Poulin and A. Fournier. A Model for Anisotropic Reflection. In *Proc. ACM SIGGRAPH*, pages 273–282, August 1990.

[49] Erik Reinhard, Brian Smits, and Charles Hansen. Dynamic Accleration Structures for Interactive Ray Tracing. In *Rendering Techniques ’00 (Proc. Eurographics Workshop on Rendering)*, pages 299–306. Springer, 2000.

[50] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast Shadows and Lighting Effects using Texture Mapping. In *Proc. ACM SIGGRAPH*, volume 26, pages 249–252, July 1992.

[51] Peter-Pike J. Sloan and Michael F. Cohen. Interactive Horizon Mapping. In *Rendering Techniques ’00 (Proc. Eurographics Workshop on Rendering)*, pages 281–286. Springer, 2000.

[52] M. Stamminger, Ph. Slusallek, and H.-P. Seidel. Interactive walkthroughs and higher order global illumination. In *Modeling, Virtual Worlds, Distributed Graphics*, pages 121–128, November 1995.

[53] W. Stürzlinger and R. Bastos. Interactive rendering of globally illuminated glossy scenes. In *Rendering Techniques ’97 (Proc. Eurographics Workshop on Rendering)*, pages 93–102. Springer, 1997.

[54] Chris Trendall and A. James Stewart. General Calculations using Graphics Hardware, with Applications to Interactive Caustics. In *Rendering Techniques ’00 (Proc. Eurographics Workshop on Rendering)*, pages 287–298. Springer, 2000.

[55] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture*, pages 392–403, 1995.

[56] Steve Upstill. *The RenderMan Companion*. Addison-Wesley, 1990.

[57] B. Walter, G. Alppay, E. LaFortune, S. Fernandez, and D. Greenberg. Fitting virtual lights for non-diffuse walkthroughs. In *Proc. ACM SIGGRAPH*, pages 45–48, August 1997.

```
/** Compute coordinates of halfvector and difference vector relative
 * to local surface frame.
 * in:
 *   $\hat{\mathbf{n}}$: normal covector; normalized to unit length
 *   $\vec{\mathbf{u}}$: tangent vector; may be unnormalized, non-orthonormal
 *   $\hat{\mathbf{l}}$: light vector; normalized to unit length
 * out:
 *   $\vec{\mathbf{p}}$: half vector surface coords (3D hemisphere map index)
 *   $\vec{\mathbf{q}}$: difference vector surface coords (3D hemisphere map index)
 */
void
HalfDiffSurfaceCoords () // $\hat{\mathbf{n}}$, $\vec{\mathbf{u}}$, $\hat{\mathbf{l}}$
{
  // Save register allocation state
  smShaderBeginBlock(); {

    // Allocate and name registers
    SMreg h = smShaderAllocReg(3);
    SMreg t = smShaderAllocReg(3);
    SMreg s = smShaderAllocReg(3);
    SMreg n = smShaderAllocReg(3);
    SMreg tp = smShaderAllocReg(3);

    // Compute normalized half vector $\hat{\mathbf{h}}$
    smShaderGetViewVec(); // $\hat{\mathbf{n}}$, $\vec{\mathbf{u}}$, $\hat{\mathbf{l}}$, $\vec{\mathbf{v}}$
    smShaderNorm();       // $\hat{\mathbf{n}}$, $\vec{\mathbf{u}}$, $\hat{\mathbf{l}}$, $\hat{\mathbf{v}}$
    smShaderAdd();        // $\hat{\mathbf{n}}$, $\vec{\mathbf{u}}$, $\vec{\mathbf{h}} = \hat{\mathbf{l}} + \hat{\mathbf{v}}$
    smShaderNorm();       // $\hat{\mathbf{n}}$, $\vec{\mathbf{u}}$, $\hat{\mathbf{h}}$
    smShaderStore(h);     // $\hat{\mathbf{n}}$, $\vec{\mathbf{u}}$

    // Generate full surface frame from $\hat{\mathbf{n}}$ and $\vec{\mathbf{u}}$
    smShaderSwap();       // $\vec{\mathbf{u}}$, $\hat{\mathbf{n}}$
    smShaderStoreCopy(n); // $\vec{\mathbf{u}}$, $\hat{\mathbf{n}}$
    Orthonormalize();     // $\hat{\mathbf{t}}$
    smShaderStoreCopy(t); // $\hat{\mathbf{t}}$
    smShaderLoad(n);      // $\hat{\mathbf{t}}$, $\hat{\mathbf{n}}$
    smShaderSwap();       // $\hat{\mathbf{n}}$, $\hat{\mathbf{t}}$
    smShaderCross();      // $\hat{\mathbf{s}} = (\hat{\mathbf{n}} \times \hat{\mathbf{t}})$
    smShaderStore(s);     // <empty>

    // Orthonormalize $\hat{\mathbf{t}}$ against $\hat{\mathbf{h}}$
    smShaderLoad(t);      // $\hat{\mathbf{t}}$
    smShaderLoad(h);      // $\hat{\mathbf{t}}$, $\hat{\mathbf{h}}$
    Orthonormalize();     // $\hat{\mathbf{t}}'$
    smShaderStore(tp);    // <empty>

    // Coordinates of $\hat{\mathbf{h}}$ relative to $(\hat{\mathbf{t}}, \hat{\mathbf{s}}, \hat{\mathbf{n}})$
    smShaderLoad(t);      // $\hat{\mathbf{t}}$
    smShaderLoad(s);      // $\hat{\mathbf{t}}$, $\hat{\mathbf{s}}$
    smShaderLoad(n);      // $\hat{\mathbf{t}}$, $\hat{\mathbf{s}}$, $\hat{\mathbf{n}}$
    smShaderLoad(h);      // $\hat{\mathbf{t}}$, $\hat{\mathbf{s}}$, $\hat{\mathbf{n}}$, $\hat{\mathbf{h}}$
    FrameVector();        // $\hat{\mathbf{p}} = ((\hat{\mathbf{t}} \cdot \hat{\mathbf{h}}), (\hat{\mathbf{s}} \cdot \hat{\mathbf{h}}), (\hat{\mathbf{n}} \cdot \hat{\mathbf{h}}))$

    // Coordinates of $\vec{\mathbf{v}}$ relative to $(\hat{\mathbf{t}}', \hat{\mathbf{s}}', \hat{\mathbf{h}})$
    smShaderLoad(tp);     // $\hat{\mathbf{p}}$, $\hat{\mathbf{t}}'$
    smShaderLoad(h);      // $\hat{\mathbf{p}}$, $\hat{\mathbf{t}}'$, $\hat{\mathbf{h}}$
    smShaderDup(1);       // $\hat{\mathbf{p}}$, $\hat{\mathbf{t}}'$, $\hat{\mathbf{h}}$, $\hat{\mathbf{t}}'$
    smShaderCross();      // $\hat{\mathbf{p}}$, $\hat{\mathbf{t}}'$, $\hat{\mathbf{s}}' = (\hat{\mathbf{h}} \times \hat{\mathbf{t}}'))$
    smShaderLoad(h);      // $\hat{\mathbf{p}}$, $\hat{\mathbf{t}}'$, $\hat{\mathbf{s}}'$, $\hat{\mathbf{h}}$
    smShaderGetViewVec(); // $\hat{\mathbf{p}}$, $\hat{\mathbf{t}}'$, $\hat{\mathbf{s}}'$, $\hat{\mathbf{h}}$, $\vec{\mathbf{v}}$
    FrameVector();        // $\hat{\mathbf{p}}$, $\vec{\mathbf{q}} = ((\vec{\mathbf{v}} \cdot \hat{\mathbf{t}}'), (\vec{\mathbf{v}} \cdot \hat{\mathbf{s}}'), (\vec{\mathbf{v}} \cdot \hat{\mathbf{h}}))$

  // Release registers
  } smShaderEndBlock();
} // $\hat{\mathbf{p}}$, $\vec{\mathbf{q}}$
```

Figure 10: *Definition of a macro to compute texture coordinates for a separable BRDF approximation using the reparameterization proposed by Kautz and McCool [28].*

```
/** Local reflectance model using two-term separable BRDF approximation.
 */
SMshader sepbrdf = smBeginShader();
{
    // allocate and name parameters
    SMparam C = smShaderDeclareColor(3);
    SMparam L = smShaderDeclareVector(3);
    SMparam T = smShaderDeclareTangent(3);
    SMparam N = smShaderDeclareNormal(3);

    // Allocate and name registers
    SMreg p = smShaderAllocReg(3);
    SMreg q = smShaderAllocReg(3);

    // Compute surface coordinates (using vertex shader)
    smBeginSubShader(SM_VERTEX);
      smShaderGetNormal(N);    // $\hat{\mathbf{n}}$
      smShaderGetTangent(T);   // $\hat{\mathbf{n}}, \vec{\mathbf{t}}$
      smShaderGetVector(L);    // $\hat{\mathbf{n}}, \vec{\mathbf{t}}, \hat{\mathbf{l}}$
      HalfDiffSurfaceCoords(); // $\hat{\mathbf{p}}, \vec{\mathbf{q}}$
    smEndSubShader();

    // Put interpolated surface coordinates in registers
    smShaderStore(q);            // $\hat{\mathbf{p}}$
    smShaderStoreCopy(p);        // $\hat{\mathbf{p}}$

    // Compute BRDF
    smShaderLookup(a);           // $\mathbf{a}[\hat{\mathbf{p}}]$
    smShaderLoad(q);             // $\mathbf{a}[\hat{\mathbf{p}}], \vec{\mathbf{q}}$
    smShaderLookup(b);           // $\mathbf{a}[\hat{\mathbf{p}}], \mathbf{b}[\vec{\mathbf{q}}]$
    smShaderMult();              // $\mathbf{ab} = \mathbf{a}[\hat{\mathbf{p}}] * \mathbf{b}[\vec{\mathbf{q}}]$
    smShaderColor3dv(aAB);       // $\mathbf{ab}, \alpha$
    smShaderMult();              // $\mathbf{AB} = \mathbf{ab} * \alpha$
    smShaderLoad(p);             // $\mathbf{AB}, \hat{\mathbf{p}}$
    smShaderLookup(c);           // $\mathbf{AB}, \mathbf{c}[\hat{\mathbf{p}}]$
    smShaderColor3dv(bC);        // $\mathbf{AB}, \mathbf{c}[\hat{\mathbf{p}}], \beta 1$
    smShaderAdd();               // $\mathbf{AB}, b\mathbf{c} = \mathbf{c}[\hat{\mathbf{p}}] + \beta 1$
    smShaderLoad(q);             // $\mathbf{AB}, b\mathbf{c}, \vec{\mathbf{q}}$
    smShaderLookup(d);           // $\mathbf{AB}, b\mathbf{c}, \mathbf{d}[\vec{\mathbf{q}}]$
    smShaderColor3dv(bD);        // $\mathbf{AB}, b\mathbf{c}, \mathbf{d}[\vec{\mathbf{q}}], \beta 2$
    smShaderAdd();               // $\mathbf{AB}, b\mathbf{c}, b\mathbf{d} = \mathbf{d}[\vec{\mathbf{q}}] + \beta 2$
    smShaderMult();              // $\mathbf{AB}, b\mathbf{cd} = b\mathbf{c} * b\mathbf{d}$
    smShaderColor3dv(aCD);       // $\mathbf{AB}, b\mathbf{cd}, \gamma$
    smShaderMult();              // $\mathbf{AB}, \mathbf{CD} = b\mathbf{cd} * \gamma$
    smShaderAdd();               // $\mathbf{f} \ = \ \mathbf{AB} + \mathbf{CD}$

    // Compute irradiance and multiply by BRDF
    smBeginSubShader(SM_VERTEX);
      smShaderGetVector(L);      // $\mathbf{f}, \hat{\mathbf{l}}$
      smShaderGetNormal(N);      // $\mathbf{f}, \hat{\mathbf{l}}, \hat{\mathbf{n}}$
      smShaderDot(N);            // $\mathbf{f}, (\hat{\mathbf{l}} \cdot \hat{\mathbf{n}})$
      smShaderParam1d(0);        // $\mathbf{f}, (\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}), 0$
      smShaderMax();             // $\mathbf{f}, s = \max((\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}), 0)$
      smShaderGetColor(C);       // $\mathbf{f}, s, \mathbf{c}$
      smShaderMult();            // $\mathbf{f}, \mathbf{e} = s * \mathbf{c}$
    smEndSubShader();

    smShaderMult();              // $\mathbf{f} * \mathbf{e}$

    // Set output fragment color
    smSetColor();                // <empty>
} smEndShader();
```

Figure 11: *Top level of the definition of a shader that implements a two-term separable approximation to a BRDF assuming a single directional light source.*

```
/** Local reflectance model using two-term separable BRDF approximation.
 * Uses infix expression compiler to simplify expression of the shader.
 */
SMshader sepbrdf = smBeginShader();
{
    // allocate and name parameters
    SMparam C = smuShaderDeclareColor(3,"C");
    SMparam L = smuShaderDeclareVector(3,"L");
    SMparam T = smuShaderDeclareTangent(3,"T");
    SMparam N = smuShaderDeclareNormal(3,"N");

    // Allocate and name registers
    SMreg p = smuShaderAllocReg(3,"p");
    SMreg q = smuShaderAllocReg(3,"q");

    // Compute surface coordinates using macro
    smBeginSubShader(SM_VERTEX);
        smShaderGetNormal(N);
        smShaderGetTangent(T);
        smShaderGetVector(L);
        HalfDiffSurfaceCoords();
    smEndSubShader();

    // Put interpolated surface coordinates in registers
    smShaderStore(q);

    // Another (silly) way of doing the above
    smuShaderExpr("p = %0");

    // Compute BRDF
    smShaderColor3dv(aAB);
    smuShaderExpr("%0*a[p]*b[q]");
    smShaderColor3dv(bD);
    smShaderColor3dv(bC);
    smShaderColor3dv(aCD);
    smuShaderExpr("%0*(c[p]+%1)*(d[q]+%2)");

    // Compute irradiance and multiply by BRDF
    smBeginSubShader(SM_VERTEX);
        smuShaderExpr("C*max((L|N),0)");
    smEndSubShader();

    smShaderMult();

    // Set output fragment color
    smSetColor();
} smEndShader();
```

Figure 12: *A rewrite of the example shader to use the textual infix expression compiler defined in the utility library and named parameters and registers. We still show only the main shader here and still use the macros defined earlier.*

```
/** Local reflectance model using two-term separable BRDF approximation.
 * Builds up DAG as a C++ data structure.
 */
SMshader sepbrdf = SmBeginShader();
{
    // allocate and name parameters
    SmExpr nC = new SmShaderDeclareColor(3,"C");
    SmExpr nL = new SmShaderDeclareVector(3,"L");
    SmExpr nT = new SmShaderDeclareTangent(3,"T");
    SmExpr nN = new SmShaderDeclareNormal(3,"N");

    // Compute surface coordinates using macro
    SmExpr nseq = HalfDiffSurfaceCoords(nN,nT,nL);
    SmExpr nvsc = new SmSubShader(SM_VERTEX,nseq);
    SmExpr nvp = new SmIndex(nseq,0);
    SmExpr nvq = new SmIndex(nseq,1);

    // Compute BRDF
    SmExpr na = new SmShaderLookup(a,nvp);
    SmExpr nb = new SmShaderLookup(b,nvq);
    SmExpr nalpha = new SmShaderColor3dv(aAB);
    SmExpr nab = new SmShaderMult(na,nb);
    SmExpr nAB = new SmShaderMult(nalpha,nab);
    SmExpr nbeta1 = new SmShaderColor3dv(bC);
    SmExpr nbeta2 = new SmShaderColor3dv(bD);
    SmExpr ngamma = new SmShaderColor3dv(aCD);
    SmExpr nc = new SmShaderLookup(c,nvp);
    SmExpr nd = new SmShaderLookup(d,nvq);
    SmExpr nalpha = new SmShaderColor3dv(aAB);
    SmExpr nbeta1c = new SmShaderAdd(nc,nbeta1);
    SmExpr nbeta1d = new SmShaderAdd(nd,nbeta2);
    SmExpr ncd = new SmShaderMult(nbeta1c,nbeta1d);
    SmExpr nCD = new SmShaderMult(ncd,ngamma);
    SmExpr nf = new SmShaderAdd(nAB,nCD);

    // Compute irradiance and multiply by BRDF
    SmExpr nz = new SmShaderParam1d(0);
    SmExpr ndot = new SmShaderDot(nL,nN);
    SmExpr nmax = new SmShaderMax(ndot,nz);
    SmExpr nC = new SmShaderGetColor(C);
    SmExpr ne = new SmShaderMult(nC,nmax);
    SmExpr nve = new SmSubShader(SM_VERTEX,ne);

    SmExpr nfe = new SmShaderMult(nf,nve);

    // Traverse and compile DAG
    SmShaderCompile(nfe);

    // Set color
    smSetColor();
} SmEndShader();
```

Figure 13: *The shader DAG can be expressed using a data structure built up object by object. The* ***SmShaderCompile*** *function propogates subshader information, then traverses the DAG and emits the appropriate shader instructions for each node. We assume that submacros have been redefined to take and return DAGs.*

```
/** Local reflectance model using two-term separable BRDF approximation.
 * Builds up DAG as a C++ data structure, but uses smart pointers and
 * constructor functions.   Also assumes a namespace to reduce the
 * length of names.
 */
SMshader sepbrdf = beginshader();
{
    // allocate and name parameters
    Expr C = declarecolor(3);
    Expr L = declarevector(3);
    Expr T = declaretangent(3);
    Expr N = declarenormal(3);

    // Compute surface coordinates using macro
    Expr sc = subshader(VERTEX,
      HalfDiffSurfaceCoords(N, L, T)
    );

    // Access elements of returned sequence
    Expr nvp = index(sc,0);
    Expr nvq = index(sc,1);

    // Compute BRDF
    Expr f = add(
      mult(
        mult(lookup(a,nvp),lookup(b,nvq)),
        color3dv(aBC)
      ),
      mult(
        mult(
          add(lookup(c,nvp),color3dv(bC)),
          add(lookup(d,nvq),color3dv(bD))
        ),
        color3dv(aCD)
      )
    );

    // Compute irradiance and multiply by BRDF
    Expr e = subshader(VERTEX,
      mult(
        C,
        max(
          dot(L,N),
          param1d(0.0)
        )
      )
    );

    // Multiply irradiance by BRDF
    Expr fe = mult(f,e);

    // Traverse and compile DAG
    fe->compile();

    // Set color
    setcolor();
} endshader();
```

Figure 14: *Functional specification of shader using constructor functions. A C++ namespace can be used to make the names of the constructor functions less verbose without polluting the global namespace.*

```
// wrap constants and texture identifiers
// (would really do when defined)
Color<CONST> alpha(3,aBC), beta1(3,bC), beta2(3,bD), gamma(3,aCD);
Texture A(a), B(b), C(c), D(d);

/** Local reflectance model using two-term separable BRDF approximation.
 * Builds up DAG as a C++ data structure, but uses smart pointers,
 * constructor functions, and operator overloading.
 */
Shader sepbrdf = beginshader();
{
    // allocate and name parameters
    Color C(3);
    Vector L;
    Tangent T;
    Normal N;

    // Compute surface coordinates using macro
    Expr<VERTEX> p, q;
    HalfDiffSurfaceCoords(p,q,N,L,T);

    // Compute BRDF
    Expr f = alpha*A[p]*B[q]
           + gamma*(C[p]+beta1)*(D[q]+beta2);

    // Compute irradiance and multiply by BRDF
    Expr<VERTEX> e = C*max((L|N),0.0));
    Expr fe = f*e;

    // traverse and compile DAG
    fe->compile();

    // Set output fragment color
    setcolor();
} endshader();
```

Figure 15: *Functional specification of shader using constructor functions that are bound to overloaded operators. Operators act on operator trees (shader expressions), not data. There are several templated subclasses of smart pointers to handle declarations, subshaders, etc. The declarations shown at the top would usually be handled at the point of definition, i.e. when you define a texture object you would also use the C++ library and so would get a smart pointer rather than an identifier anyways.*