Chapter 7

API Design

Michael McCool

SMASH Metaprogramming Shader API

Michael D. McCool, Qin Zheng and Tiberiu Popa

Computer Graphics Lab
Department of Computer Science
University of Waterloo

5th April 2002

1 Introduction

Modern graphics accelerators have embedded programmable components in the form of vertex and fragment shading units. Current APIs permit specification of the programs for these components using an assembly-language level interface. Shader compilers are available [5, 7] but these read in an external string specification, which can be inconvenient. Both the DX9 and OpenGL 2.0 proposals also use an external high-level shading language separate from the host language, with actual shader programs specified either in strings or in separate files.

It is possible, using C++, to define a high-level shading language directly in the API. This would permit more direct interaction with the specification of textures and parameters, simplifies implementation (an LR parser is not required), and permits on-the-fly generation and manipulation of shader programs to specialize them [2] as needed. Precompiled shader programs could still be used simply by compiling and running a C++ program defining an appropriate shader and dumping a precompiled binary representation. However, parameter naming and binding are simplified if the application program and the shader program are compiled together, since objects defining named parameters and textures can be accessed by the shader definition directly.

SMASH supports two levels of API: an OpenGL-like low-level C-compatible API, whose calls are indicated with the prefix `sm`, and a high-level C++ API whose calls and types are indicated with the prefix `Sm`. In this document, we focus on the C++ shader API. The low-level shader API, which the high-level shader API compiles to, is based on the DX9 assembly language but with a function call-based API in the style of ATI's OpenGL vertex shader extensions.

This whitepaper describes a work in progress and the detailed syntax of the final system may differ from what is shown below. Please access our website at

`http://www.cgl.uwaterloo.ca/Projects/rendering/`

for more up to date information. The syntax described here also differs from that documented for earlier versions of SMASH.

2 Metaprogramming Parser

String based shading languages need a separate parsing step, usually based on an LR grammar parser-compiler such as YACC or Bison, to convert the syntax of the shader program to a parse tree. However, using a metaprogramming API, the shader program is specified using a

sequence of function calls originating directly in the application program. The API then interprets this sequence of calls as a set of "tokens" to be used to generate a parse tree which can in turn be compiled by an on-the-fly compiler backend in the API driver library. Expressions in a shading language can be parsed and type-checked at the application program's compile time using operator overloading. To do this, overloaded operator functions are defined that construct symbolic parse trees for the expressions rather than executing computations directly. The "variables" in the shader are in fact smart reference-counting pointers to nodes in directed acyclic graphs representing expressions, and each operator allocates a new node and uses smart pointers to refer to its children. The reference-counting smart pointers implement a simple garbage collection scheme which in this case is adequate to avoid memory leaks. Compiling expressions in this way eliminates a large chunk of the grammar for the shading language; the API gets passed a complete parse tree for expressions directly, and does not have to build it itself by parsing a flat sequence of tokens. Each assignment in sequence is recorded as a statement in the shader program and buffered until the entire sequence of commands has been received. When the shader program is complete, code generation and optimization is performed by the driver, resulting internally in machine language which is prepared for downloading to the specified shader unit when the shader program is bound.

Eventually, when shading units support control constructs, the shading language can be extended with API calls that embed tokens for control keywords in the shader statement sequence: `SmIF(`*`cond`*`)`, `SmWHILE(`*`cond`*`)`, `SmENDIF()`, etc. Complex statements are received by the API as a sequence of such calls/tokens. For instance, a WHILE statement would be presented to the API as a WHILE token (represented by an `SmWHILE(`*`cond`*`)` function call; note the parameter, which refers to an expression parse tree for the condition), a sequence of other statements, and a matching ENDWHILE token. Use of these constructs can be wrapped in macros to make the syntax slightly cleaner (i.e. to hide semicolons and function call parenthesis):

```
#define SM_IF(cond)              SmIF(cond);
#define SM_ELSEIF(cond)          SmELSEIF(cond);
#define SM_ELSE                  SmELSE();
#define SM_ENDIF                 SmENDIF();
#define SM_WHILE(cond)           SmWHILE(cond);
#define SM_ENDWHILE              SmENDWHILE();
#define SM_DO                    SmDO();
#define SM_UNTIL(cond)           SmUNTIL(cond);
#define SM_FOR(init,cond,inc)    SmFOR(init,cond,inc);
#define SM_ENDFOR                SmENDFOR();
```

Since expression parsing (and type checking) is done by C++ at the compile time of the host language, all that is needed to parse structured control constructs is a straightforward recursive-descent parser. This parser will traverse the buffered token sequence when the shader program is complete, generating a full parse tree internally. Code generation can then take place in the usual way.

Although true conditional execution and looping are not yet available in any commercial real-time shading system, such control constructs can theoretically be implemented efficiently in the context of a long texture lookup latency with either a recirculating pipeline or a multithreaded shading processor.

3 Testbed

Our high-level shader API is built on top of SMASH, a testbed we have developed to experiment with possible next-generation graphics hardware features and their implementation.

This system is modular. Pipelines can be built with any number of shading processors or other types of modules (such as rasterizers or displacement units) chained together in sequence or in parallel. The API has to deal with the fact that any given SMASH system might have a variable number of shading units, and that different shading units might have slightly different capabilities (for instance, vertex shaders might not have texture units, and fragment shaders may have a limited number of registers and operations). These restrictions are noted when a system is built and the shader compiler adapts to them.

The API currently identifies shaders by pipeline depth. In the usual case of a vertex shader and a fragment shader, the vertex shader has depth 0 and the fragment shader has depth 1. When a shader program is downloaded, the packet carrying the program information has a counter. If this counter is non-zero, it is decremented and the packet is forwarded to the next unit in the pipeline. Otherwise, the program is loaded and the packet absorbed. Modules in the pipeline that do not understand a certain packet type are also supposed to forward such packets without change. A flag in each packet indicates whether or not packets should be broadcast over parallel streams or not; shader programs are typically broadcast. In this fashion shader programs can be sent to any shader unit in the pipe. Sequences of tokens defining a shader program are defined using a sequence of API calls inside a matched pair of `SmBeginShader(`*`shaderlevel`*`)` and `SmEndShader()` calls. Once defined, a shader can be loaded using the `SmBindShader(`*`shaderobject`*`)` call.

When a program is running on a shader unit, vertex and fragment packets are rewritten by that unit. The system supports packets of length up to 255 words, not counting a header which gives the type and length of each packet. Each word is 32 bits in length, so shaders can have up to 255 single-precision inputs and outputs.[1] Type declarations in shader parameter declaration can be used to implicitly define packing and unpacking of shorter parameters to conserve bandwidth when this full precision is not necessary. Other units, such as the rasterizer and compositing module, also need to have packets formatted in a certain way to be meaningful; in particular, the rasterizer needs the position of a vertex in a certain place in the packet (at the end, consisent with the order of parameter and vertex calls). These units also operate by packet rewriting; for instance, a rasterizer parses sequences of vertices according to the current geometry mode, reconstructs triangles from them, and converts them into streams of fragments.

4 Parameter Binding

It is convenient to support two different techniques for passing parameters to shaders. For semi-constant parameters, the use of named parameters whose values can be changed at any time and in any order is convenient. We will give these parameters the special name of *attributes* and will reserve the word *parameters* for values specified per-vertex. A named attribute is created simply by constructing an object of an appropriate type:

```
// create named transformation attributes
SmAttributeAffXform3x4f modelview;
SmAttributeProjXform4x4f perspective;
// create named light attributes
SmAttributeColor3f light_color;
SmAttributePoint3f light_position;
```

The constructor of these classes makes appropriate calls into the API to allocate state for these attributes, and the destructor makes calls to deallocate this state. Operators overloaded on

[1] In practice, to support antialiasing at the fragment level, extra overhead may also be required to transmit differentials. Using half precision for two differential values per parameter doubles the amount of space required for each parameter and cuts the maximum number of parameters in half.

these classes are used in the shader definition to access these values. When a shader definition uses such an attribute the compiler notes this fact and arranges for the current value of each such attribute to be bound to a constant register in the shader unit when the shader program is loaded. Attributes of all types can be associated with stacks for save/restore. For convenience, operators are overloaded on both the classes themselves and pointers to them so either implicit or explicit allocation and deallocation can be used. Arrays of attributes are supported with special classes as well.

For parameters whose values change at every vertex, for efficiency we have chosen to make the order of specification of these parameters important. In immediate mode, a sequence of generic multidimensional parameter calls simply adds parameters to a packet, which is sent off as a vertex packet when the vertex call is made (after adding the last few parameters given in the vertex call itself). This is actually supported directly in the low-level API. For instance, suppose we want to pass a tangent vector, a normal, and a texture coordinate to a vertex shader at the vertices of a single triangle. In immediate mode we would use calls of the form

```
smBegin(SM_TRIANGLES);
    smVector3fv(tangent[0]);
    smNormal3fv(normal[0]);
    smTexCoord2fv(texcoord[0]);
    smVertex3fv(position[0]);

    smVector3fv(tangent[1]);
    smNormal3fv(normal[1]);
    smTexCoord2fv(texcoord[1]);
    smVertex3fv(position[1]);

    smVector3fv(tangent[2]);
    smNormal3fv(normal[2]);
    smTexCoord2fv(texcoord[2]);
    smVertex3fv(position[2]);
smEnd();
```

The types given above are optional, and are checked at runtime only in a special “test mode”. High-performance runtime mode, which is invoked by linking to a different version of the API library, simply assumes the types match. The generic parameter call `smParam*` can be used in place of `smVector*`, `smNormal*`, etc. Vertex and parameter arrays are of course also supported for greater efficiency.

Declarations inside every shader definition provide the necessary information to enable the system to make sure the necessary named parameters are loaded into the shader and that unnamed parameters are unpacked in the right order for vertices and fragments. The API must also ensure that when a shader is bound that any texture objects it uses are also bound. The C++ API itself also uses classes to wrap low-level texture objects so that within a shader definition a texture lookup can be specified as if it were an array access. As with attributes, operators are overloaded on pointers to texture objects as well as on texture objects themselves. To avoid confusion with the `[]` array-access operator, a special class is defined for arrays of texture objects.

In the rest of the paper we will give a sequence of examples to demonstrate the high-level C++ shader API.

5 Modified Phong Lighting Model

Consider the modified (reciprocal) Blinn-Phong lighting model:

$$L_o \quad = \quad \Big(k_d[\mathbf{u}] + k_s[\mathbf{u}](\hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^q\Big) \max(0, (\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}))I_\ell / r_\ell^2$$

where $\hat{\mathbf{v}}$ is the normalized view vector, $\hat{\mathbf{l}}$ is the normalized light vector, $\hat{\mathbf{h}} = \mathrm{norm}(\hat{\mathbf{v}} + \hat{\mathbf{l}})$ is the normalized half vector, $\hat{\mathbf{n}}$ is the normalized surface normal, I_ℓ is the light source intensity, r_ℓ is the distance to light source, $k_d[\mathbf{u}]$, $k_s[\mathbf{u}]$, and q are parameters of the lighting model, and $\mathbf{u}$ is a 2D surface texture coordinate.

We will implement this using per-pixel computation of the specular lobe and texture mapping of k_d and k_s. In general, the notation $t[\mathbf{u}]$ indicates a filtered and interpolated texture lookup, not just a simple array access (although, if the texture object access modes are set up appropriately, it can be made equivalent to a simple array access).

5.1 Vertex Shader

This shader computes the model-view transformation of position and normal, the projective transformation of view-space position into device space, the halfvector, and the irradiance. These values will be ratiolinearly interpolated by the rasterizer and the interpolated values will be assigned to the fragments it generates. The rasterizer expects the last parameter in each packet to be a device-space 4-component homogeneous point.

```
SmShader phong0 = SmBeginShader(0); {
  // declare vertex parameters, in order given
  SmInputTexCoord2f ui;    // texture coords
  SmInputNormal3f nm;      // normal vector (MCS)
  SmInputPoint3f pm;       // position (MCS)

  // declare outputs, in order sent
  SmOutputVector3f hv;     // half-vector (VCS)
  SmOutputTexCoord2f uo;   // texture coords
  SmOutputNormal3f nv;     // normal (VCS)
  SmOutputColor3f ec;      // irradiance
  SmOutputPoint4f pd;      // position (HDCS)

  // compute VCS position
  SmRegPoint3f pv = modelview * pm;
  // compute DCS position
  pd = perspective * pv;
  // compute normalized VCS normal
  nv = normalize(nm * inverse(modelview));
  // compute normalized VCS light vector
  SmRegVector3f lvv = light_position - pv;
  SmRegParam1f rsq = 1.0/(lvv|lvv);
  lvv *= sqrt(rsq);
  // compute irradiance
  SmRegParam1f ct = max(0,(nv|lvv));
  ec = light_color * rsq * ct;
  // compute normalized VCS view vector
  SmRegVector3f vv = -normalize(pv);
  // compute normalized VCS half vector
```

```
  hv = normalize(lvv + vv);
  // pass through texture coordinates
  uo = ui;
} SmEndShader();
```

We do not need to provide prefixes for the utility functions `normalize`, `sqrt`, etc. since they are distinguished by the type of their arguments. In our examples we will also highlight, using boldface, the use of externally declared attribute and texture objects.

The types `SmInput*` and `SmOutput*` are classes whose constructors call allocation functions in the API. The order in which these constructors are called provides the necessary information to the API on the order in which these values should be unpacked from input packets and packed into output packets. Temporary registers can also be declared explictly as shown, although of course the compiler will declare more internally in order to implement expression evaluation, and will optimize register allocation as well. These "register" declarations, therefore, are really just smart pointers to expression parse trees.

SMASH permits allocation of named transformation matrices in the same manner as other attributes. Matrices come in two varieties, representing affine transformations and projective transformations. When accessing a matrix value, the matrix can be bound either as a transpose, inverse, transpose inverse, adjoint, or transpose adjoint. The adjoint is useful as it is equivalent to the inverse within a scale factor. However, we do not need to declare these bindings explicitly since simply using a object representing a named attribute or matrix stack is enough to bind it to the shader and for the API to arrange for that parameter to be sent to the shader processor when updated. The symbolic functions `transpose`, `inverse`, `adjoint`, etc. cause the appropriate version of the matrix to be bound to the shader.

5.2 Fragment Shader

This shader completes the Blinn-Phong lighting model example by computing the specular lobe and adding it to the diffuse lobe. Both reflection modes are modulated by specular and diffuse colors that come from texture maps using the previously declared texture objects `phong_kd` and `phong_ks`. The rasterizer automatically converts 4D homogenous device space points (specifying the positions of vertices) to normalized 3D device space points (specifying the position of each fragment). The 32-bit floating-point fragment depth z comes first in the output packet to automatically result in the correct packing and alignment for x and y.

The Phong exponent is specified here as a named attribute. Ideally, we would antialias this lighting model by clamping the exponent as a function of distance and curvature [1], but we have not implemented this functionality.

```
SmShader phong1 = SmBeginShader(1); {
  // declare inputs, in order given
  SmInputVector3f hv;      // half-vector (VCS)
  SmInputTexCoord2f u;     // texture coordinates
  SmInputNormal3f nv;      // normal (VCS)
  SmInputColor3f ec;       // irradiance
  SmInputParam1f pdz;      // fragment depth (DCS)
  SmInputParam2us pdxy;    // fragment 2D position (DCS)

  // declare outputs, in order sent
  SmOutputColor3f fc;      // final fragment color
  SmOutputParam1f fpdz;    // final fragment depth
  SmOutputParam2us fpdxy; // final fragment 2D position

  // compute texture-mapped diffuse lobe
```

```
  fc = phong_kd[u];
  // compute texture-mapped specular lobe
  fc += phong_ks[u]
        * pow((normalize(hv)|normalize(nv)),phong_exp);
  // multiply lighting model by irradiance
  fc *= ec;
  // pass through depth and position
  fpdz = pdz;
  fpdxy = pdxy;
} SmEndShader();
```

Since it is not needed for bit manipulation, we use the operator "`|`" to indicate the inner (dot) product between vectors rather than bitwise OR. We also use the operator "`&`" for the cross product, which has the advantage that the triple product can be easily defined. However, parentheses should be always be used around dot and cross products when they are used in other expressions due to the low precendence of these operators.

Matrix multiplications are indicated with the "`*`" operator. In matrix-vector multiplications if the vector appears on the right it is interpreted as a column and if on the left as a row. For the most part this eliminates the need to explicitly specify transposes. Since we have chosen to use "`*`" to represent matrix multiplication and not the more abstract operation of typed transformation application, to transform a normal you have to explicitly specify the use of the inverse and use the normal as a row vector. Use of the "`*`" operator on a pair of tuples of any type results in pairwise multiplication. Use of "`*`" between a 1D scalar value and any nD tuple results in scalar multiplication.

6 Separable BRDFs and Material Mapping

A bidirectional reflection distribution function f is in general a 4D function that relates the differential incoming irradiance to the differential outgoing radiance.

$$L_o(\mathbf{x}, \hat{\omega}_o) \quad = \quad \int_\Omega f(\hat{\omega}_o, \mathbf{x}, \hat{\omega}_i)\ \max(0, \hat{\mathbf{n}} \cdot \hat{\omega}_i)\, L_i(\mathbf{x}, \hat{\omega}_i)\, d\hat{\omega}_i.$$

Relative to a point source, which would appear as an impulse function in the above integral, the BRDF can be used as a lighting model:

$$L_o(\mathbf{x}, \hat{\omega}_o) \quad = \quad f(\hat{\omega}_o, \mathbf{x}, \hat{\omega}_i)\ \max(0, \hat{\mathbf{n}} \cdot \hat{\omega}_i)\, I_\ell / r_\ell^2.$$

In general, it is impractical to tabulate a general BRDF. A 4D texture lookup would be required. Fortunately, it is possible to approximate BRDFs by factorization. A numerical technique called homomorphic factorization [4] can be used to find a separable approximation to any shift-invariant BRDF:

$$f_m(\hat{\omega}_o, \hat{\omega}_i) \quad \approx \quad p_m(\hat{\omega}_o)\, q_m(\hat{\mathbf{h}})\, p_m(\hat{\omega}_i)$$

In this factorization, we have chosen to factor the BRDF into terms dependent directly on incoming direction, outgoing direction, and half vector direction, all expressed relative to the local surface frame. Other parameterizations are possible but this one seems to work well in many circumstances and is easy to compute.

To model the dependence of the reflectance on surface position, we can sum over several BRDFs, using a texture map to modulate each BRDF. We call this *material mapping*:

$$f(\hat{\omega}_o, \mathbf{u}, \hat{\omega}_i) \quad = \quad \sum_{m=0}^{M-1} t_m(\mathbf{u})\, f_m(\hat{\omega}_o, \hat{\omega}_i)$$

$$= \sum_{m=0}^{M-1} t_m(\mathbf{u})\, p_m(\hat{\omega}_o)\, q_m(\hat{\mathbf{h}})\, p_m(\hat{\omega}_i).$$

When storing them in a fixed-point format, we also rescale the texture maps to maximize precision:

$$f(\hat{\omega}_o, \mathbf{u}, \hat{\omega}_i) = \sum_{m=0}^{M-1} \alpha_m t'_m(\mathbf{u}) p'_m(\hat{\omega}_o) q'_m(\hat{\mathbf{h}}) p'_m(\hat{\omega}_i).$$

6.1 Vertex Shader

Here is a vertex shader to set up material mapping using a separable BRDF decomposition for each material.

```
SmShader hf0 = SmBeginShader(0); {
  // declare vertex parameters, in order given
  SmInputTexCoord2f ui;  // texture coords
  SmInputVector3f t1;    // primary tangent
  SmInputVector3f t2;    // secondary tangent
  SmInputPoint3f pm;     // position (MCS)

  // declare output parameters, in order given
  SmOutputVector3f vvs;  // view-vector (SCS)
  SmOutputVector3f hvs;  // half-vector (SCS)
  SmOutputVector3f lvs;  // light-vector (SCS)
  SmOutputTexCoord2f uo; // texture coords
  SmOutputColor3f ec;    // irradiance
  SmOutputPoint4f pd;    // position (HDCS)

  // compute VCS position
  SmRegPoint3f pv = modelview * pm;
  // compute DCS position
  pd = perspective * pv;
  // transform and normalize tangents
  t1 = normalize(modelview * t1);
  t2 = normalize(modelview * t2);
  // compute normal via a cross product
  SmRegNormal3f nv = normalize(t1 & t2);
  // compute normalized VCS light vector
  SmRegVector3f lvv = light_position - pv;
  SmRegParam1f rsq = 1.0/(lvv|lvv);
  lvv *= sqrt(rsq);
  // compute irradiance
  SmRegParam1f ct = max(0,(nv|lvv));
  ec = light_color * rsq * ct;
  // compute normalized VCS view vector
  SmRegVector3f vv = -normalize(pv);
  // compute normalized VCS half vector
  SmRegVector3f hv = norm(lvv + vv);
  // project BRDF parameters onto SCS
  vvs = SmRegVector3f((vvv|t1),(vvv|t2),(vvv|nv));
  hvs = SmRegVector3f((hvv|t1),(hvv|t2),(hvv|nv));
```

```
  lvs = SmRegVector3f((lvv|t1),(lvv|t2),(lvv|nv));
  // pass through texture coordinates
  uo = ui;
} SmEndShader();
```

6.2 Fragment Shader

The fragment shader completes the material mapping shader by using an application program loop (running on the host) to generate an unrolled shader program. Note that a looping construct is not required in the shader program to implement this. In fact, the API does not even see the loop, only the calls it generates.

```
SmShader hf1 = SmBeginShader(1); {
  // declare parameters, in order given
  SmInputVector3f vv;      // view-vector (SCS)
  SmInputVector3f hv;      // half-vector (SCS)
  SmInputVector3f lv;      // light-vector (SCS)
  SmInputTexCoord2f u;     // texture coordinates
  SmInputColor3f ec;       // irradiance
  SmInputParam1f pdz;      // fragment depth (DCS)
  SmInputParam2us pdxy;    // fragment position (DCS)

  // declare outputs, in order sent
  SmOutputColor3f fc;      // fragment color
  SmOutputParam1f fpdz;    // fragment depth
  SmOutputParam2us fpdxy; // fragment position

  // intialize total reflectance
  fc = SmColor3f(0.0,0.0,0.0);
  // sum up contribution from each material
  for (int m = 0; m < M; m++) {
     fc += hf_mat[m][u] * hf_alpha[m]
           * hf_p[m][vv] * hf_q[m][hv] * hf_p[m][lv];
  }
  // multiply by irradiance
  fc *= ec;
  // pass through fragment position and depth
  fpdz = pdz;
  fpdxy = pdxy;
} SmEndShader();
```

Here the texture array objects `hf_mat`, `hf_p`, and `hf_q` should have been previously defined, along with the normalization factor attribute array `hf_alpha`. The texture objects `hf_p` and `hf_q` should have been set up as cube maps so unnormalized direction vectors can be used directly as texture parameters.

7 Marble and Wood

To implement marble, wood, and similar materials, we have used the simple parameterized model for such materials proposed by John C. Hart et al. [3]. This model is given by

$$t(\mathbf{x}) \quad = \quad \sum_{i=0}^{N-1} \alpha_i |n(2^i \mathbf{x})|,$$

$$\begin{aligned} \mathbf{u} &= \mathbf{x}^T \mathsf{A}\mathbf{x} + t(\mathbf{x}), \\ k_d(\mathbf{x}) &= c_d[\mathbf{u}], \\ k_s(\mathbf{x}) &= c_s[\mathbf{u}]. \end{aligned}$$

where n is a bandlimited noise function such as Perlin noise [6], t is the "turbulence" noise function synthesized from it, A is a 4×4 symmetric matrix giving the coefficients of the quadric function $\mathbf{x}^T\mathsf{A}\mathbf{x}$, c_d and c_s are a 1D MIP-mapped texture maps functioning as filtered color lookup tables, and $\mathbf{x}$ is the model-space (normalized homogeneous) position of a surface point. The outputs need to be combined with a lighting model, so we will combine them with the Phong lighting model (we could just as easily have used separable BRDFs and material maps, with one color lookup table for each).

Generally speaking we would use fractal turbulence and would have $\alpha_i = 2^i$; however, for the purposes of this example we will permit the α_i values to vary to permit further per-material noise shaping and will bind them to named attributes. Likewise, various simplifications would be possible if we fixed A (marble requires only a linear term, wood only a cylinder) but we have chosen to give an implementation of the more general model and will bind A to a named attribute.

The low-level SMASH API happens to have support for Perlin noise, generalized fractal noise, and generalized turbulence built in, so we do not have to do anything special to evaluate these noise functions. If we had to compile to a system without noise hardware, we would store a periodic noise function in a texture map and then could synthesize aperiodic fractal noise by including appropriate rotations among octaves in the noise summation.

7.1 Vertex Shader

The vertex shader sets up the Phong lighting model, but also computes half of the quadric as a linear transformation of the model space position. Note that this can be correctly ratiolinearly interpolated.

```
SmShader pnm0 = SmBeginShader(0); {
  // declare vertex parameters, in order given
  SmInputNormal3f nm;  // normal vector (MCS)
  SmInputPoint3f pm;   // position (MCS)

  // declare outputs, in order sent
  SmOutputPoint4f ax;  // coeffs x MCS position
  SmOutputPoint4f x;   // position (MCS)
  SmOutputVector3f hv; // half-vector (VCS)
  SmOutputVector3f nv; // normal (VCS)
  SmOutputColor3f ec;  // irradiance
  SmOutputPoint4f pd;  // position (HDCS)

  // transform position
  SmRegPoint3f pv = modelview * pm;
  pd = perspective * pv;
  // transform normal
  nv = normalize(nm * inverse(modelview));
  // compute normalized VCS light vector
  SmRegVector3f lvv = light_position - pv;
  SmRegParam1f rsq = 1.0/(lvv|lvv);
  lvv *= sqrt(rsq);
  // compute irradiance
```

```
  SmRegParam1f ct = max(0,(nv|lvv));
  ec = light_color * rsq * ct;
  // compute normalized VCS view vector
  SmRegVector3f vv = -normalize(pv);
  // compute normalized VCS half vector
  hv = norm(lvv + vv);
  // pass through texture coordinates
  uo = ui;

  // projectively normalize position
  x = projnorm(pm);
  // compute half of quadric
  ax = quadric_coefficients * x;
} SmEndShader();
```

7.2 Fragment Shader

The fragment shader completes the computation of the quadric and the turbulence function and passes their sum through the color lookup table. Two different lookup tables are used to modulate the specular and diffuse parts of the lighting model, which will permit, for example, dense dark wood to be shinier than light wood (with the appropriate entries in the lookup tables).

```
SmShader pnm1 = SmBeginShader(1); {
  // declare parameters, in order given
  SmInputPoint4f ax;      // coeffs x MCS position
  SmInputPoint4f x;       // position (MCS)
  SmInputVector3f hv;     // half-vector (VCS)
  SmInputVector3f nv;     // normal (VCS)
  SmInputColor3f ec;      // irradiance
  SmInputParam1f pdz;     // fragment depth (DCS)
  SmInputParam2us pdxy;   // fragment 2D position (DCS)

  // declare outputs, in order sent
  SmOutputColor3f fc;     // fragment color
  SmOutputParam1f fpdz;   // fragment depth
  SmOutputParam2us fpdxy; // fragment 2D position

  // compute texture coordinates
  SmRegTexCoord1f u = (x|ax) + turbulence(pnm_alpha,x);
  // compute diffuse and specular colors
  SmRegColor3f kd = pnm_cd[u];
  SmRegColor3f ks = pnm_cs[u];
  // compute Blinn-Phong lighting model
  fc += kd;
  fc += ks * pow((normalize(hv)|normalize(nv)),phong_exp);
  // multiply by irradiance
  fc *= ec;
  // pass through fragment depth and position
  fpdz = pdz;
  fpdxy = pdxy;
} SmEndShader();
```

Acknowledgements

This research was funded by grants from the National Science and Engineering Research Council of Canada (NSERC), the Centre for Information Technology of Ontario (CITO), the Canadian Foundation for Innovation (CFI), the Ontario Innovation Trust (OIT), and finally the Bell University Labs initiative.

References

[1] John Amanatides. Algorithms for the detection and elimination of specular aliasing. In *Proc. Graphics Interface*, pages 86–93, May 1992.

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

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

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

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

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

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