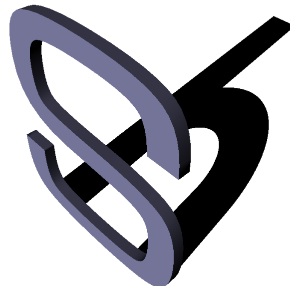
Chapter 7. Shader Metaprogramming with Sh

Michael McCool

A Short Introduction to



Michael McCool

President, Serious Hack, Inc.
Associate Professor, University of Waterloo

mmccool@serioushack.com
mmccool@cgl.uwaterloo.ca

http://serioushack.com/
http://libsh.org/
http://www.cgl.uwaterloo.ca/

25th April 2005

Figure 1: Some example shaders.

1 Introduction

Sh is a high-level GPU programming "language" whose "parser" is implemented using C++ operator overloading. Therefore, Sh is not really a language, but an advanced C++ API. It looks at first like a standard graphics library, with matrices, points, and vectors, and you can use it that way if you want. But you can also capture sequences of operations in a "retained mode", much like a display list, and compile these operations for later execution on the GPU. Sh has its own optimizer and supports a modular backend system. The runtime engine of Sh tries to make best use of the available hardware and graphics API features but without burdening the programmer with low-level details. It manages buffers and textures as well as shaders, making it much more convenient to program multipass algorithms and to encapsulate data representations.

Sh programs can run on the GPU but act like extensions of the host application. In particular, the semantics of Sh have been set up so that Sh programs act like C++ functions, and textures and buffers act like arrays. For instance, uniform parameters are simply variables external to Sh pro-

gram definitions, and textures are simply an array-valued parameter type. The C++ scope rules then control which parameters and textures get bound to which shaders, while the Sh runtime system manages updates of uniforms, buffers, and binding of texture units. This means that C++ modularity constructs can be used to organize Sh programs: namespaces, templates, and classes can be used to organize and parameterize shaders; you can define your own types and operators; and you can define libraries of functions and shaders. C++ control constructs can also be used to manipulate and construct Sh programs on the fly. Such metaprogramming can be used to adapt implementation complexity and performance to the target platform, generate variants of shaders for different levels of detail, and generate shaders from data files read in at runtime. Since C++ scope rules are used to control binding, no glue code is needed.

Sh can be used for single shaders, to implement complex multipass algorithms, or for general-purpose computation on GPUs. For general-purpose computation, compiled Sh program objects can be applied as functions to streams of data. These programs will execute on the GPU without it ever being necessary for the user to make a graphics API call. The same programs can be compiled (on the fly, using just-in-time compilation, and with full support for metaprogramming) to the host CPU as well, so the decision to execute an algorithm on the CPU or GPU can be deferred — until runtime if necessary. Program objects can also be manipulated in various ways: specialization, conversion of uniform variables to inputs, conversion of inputs to uniform variables, conversion of inputs to texture lookups, currying, functional composition, concatenation, and other operations are available.

Sh comes with an extensive standard library that includes matrix and geometry functions, lighting models, noise functions (cellnoise, perlin, turbulence, worley, ...), advanced texture representations, standard shader kernels, and other functionality.

In the following sections we will introduce Sh by working through a number of examples. First, in Section 2, we will present the basic datatypes of Sh, the tuples and matrices, and show how they can be used in immediate mode. Section 3 shows how Sh programs are created by capturing sequences of operations on these types using a retained-mode mechanism. We also show how the interface between Sh programs and the rest of the application is defined. In Section 4 we discuss textures, and show how Sh can be used to build data abstractions. In Section 5 we present the stream processing model of Sh, which provides support for general-purpose programming. We

also discuss capabilities for manipulating program objects. In Section 6 we discuss the internals of the Sh implementation and the development tools available for it. Finally, in Section 7 we explain the licensing and history of Sh and how to obtain it. The core of Sh is free, open-source, and vendor and API neutral. However, the language is also being commercialized and a backward-compatible, extended professional version is being developed and will be supported by Serious Hack Inc., a company created for this purpose.

2 Tuples, Matrices, Operators, and Functions

The core types in Sh are n-tuples of numbers. Tuples have a semantic type (such as point, vector, normal, plane, and so forth), a length (which must be a compile time constant, but may otherwise be arbitrary; in particular, tuples in Sh are not limited to length 4), and a storage type. For instance, ***ShPoint3f*** is a three-dimensional point stored as a triple of single-precision floating-point numbers, whereas ***ShColor4ub*** is a four-component color whose components are stored as unsigned bytes. Generic tuples are given the semantic type of ***ShAttrib***, for instance ***ShAttrib3f***. These names are actually predefined `typedef`s wrapping a more general template mechanism for specifying tuples.

Operators are overloaded on these types. Arithmetic operators usually act componentwise, except for multiplication and division on certain types in which an alternative definition makes sense (matrices in particular). Scalar promotion works on most operators; for instance, you can multiply a vector by a scalar (and that scalar can be a C++ floating-point value or an Sh 1-tuple).

Sh is not overly picky about type checking; for instance, it is possible to add two points, even though this is geometrically questionable. However, if we made this operation illegal, common operations such as barycentric combinations would be annoying to specify. Most of the standard library functions will also silently accept most semantic types. The semantic types still provide useful documentation and metainformation, and in some cases control the meaning of operators and functions (multiplication of matrices, for instance).

The "`()`" notation is used for swizzling, using integers to index components. If `c` is an ***ShColor3f*** representing an RGB color, then `c(2,1,0)` gives that color in BGR order. Swizzles can also change the length of a tuple or

repeat elements. If we are given a 2-component LA color `b` and want to convert it to a 3-component RGB color, we could use `b(0,0,0)`. The swizzle notation with one argument can be used to select elements of a tuple, although the "`[]`" operator can also be used in this case.

The "`|`" and "`^`" operators are used for dot product and cross product respectively, although `dot` and `cross` functions are also supported. Sh supports a large number of other built-in functions, including noise functions of various kinds, tuple sorting, trigonometric, exponential, logarithmic, and geometric operations, smoothed discontinuities, and spline evaluators. All functions in the standard library are designed to work on graphics hardware without branches or loops, and when possible the names are similar to those in other shading languages and the C++ standard library.

Comparison operations return tuples whose components are either 0 or 1, using the same storage types as their inputs. The "`&&`" and "`||`" operators are defined to mean `min` and `max` respectively (which is consistent with Boolean operations). The `any` and `all` operations can be used to reduce Boolean tuples to a single decision. The `cond` function supports conditional assignment (unfortunately, "`?:`" cannot be overloaded in C++) with both Boolean tuples and scalars.

Suppose you would like to simulate the appearance of glass. You will need a function to compute a reflection vector, a refraction vector, and the Fresnel coefficient to determine the ratio between reflection and refraction. These are actually already in the standard library, but if you wanted to define a reflection function yourself, you could do it as shown in Listing 1. This has the obvious semantics. In fact, you can use function pointers, pass by reference to return values in arguments, separate compilation, and all the usual machinery of C++.

```
ShVector3f
reflect (ShVector3f v, ShNormal3f n) {
  return ShVector3f(2.0*(n|v)*n - v);
}
```

Listing 1: Reflection function.

Sh supports small fixed-size matrices to represent transformations of points and vectors. Matrix types have names like *ShMatrix3x4f* and can be square or rectangular. There is a standard library that supports operations on these matrices, and includes functions to compute the determinant, adjoint, trace, inverse, and transpose, and to build matrices from tuples by row

or column. Matrices also support swizzling and slicing. If `M` is a 4×4 matrix, then `M(2,1,0)(2,1,0)` extracts the 3×3 upper-left submatrix and transposes it. Both row and column swizzles must be supplied, but the empty swizzle "`()`" is the identity. Therefore, `M()(3,2,1,0)` reverses the columns of `M` and `M(3,2,1,0)()` reverses the rows. Matrices also support the "`[]`" operator. Application of this operator extracts a tuple representing a row of the matrix. A second application of "`[]`" then selects a component of this tuple, so `M[2]` selects row 2 of the matrix `M` as an *ShAttrib4f* and `M[2][3]` selects the scalar at row 2, column 3.

On the left hand side of an assignment, the swizzling operators can be used to selectively write to elements of both tuples and matrices. This can be interpreted as swizzling references to elements. The only restriction is that on the left hand side, elements cannot be repeated (since this would imply assigning two values to one element).

Transformations of tuples by matrices is supported by the "`*`" or "`|`" operators, which both represent matrix/tuple multiplication. If the matrix is on the left and the tuple on the right, the tuple is interpreted as a column vector. If the tuple is on the left and the matrix is on the right, the tuple is interpreted as a row vector. This rule avoids a lot of transpose operators, and is consistent with the use of "`|`" for dot product. If a tuple is one element too small for the matrix being applied to it, it is automatically extended with a homogeneous coordinate relative to its type. For instance, if you try to transform an *ShPoint3f* by an *ShMatrix4x4f*, the point will be automatically extended with a homogeneous coordinate of 1. If an *ShVector3f* is transformed the same way, it will be extended with a homogeneous coordinate of 0. This is one of the only places where the semantic type of a tuple matters—and note that the alternative would be to declare a size mismatch error.

3 Programs, Parameters, and Attributes

Tuple and matrix types can be used in immediate mode as if Sh were a simple graphics utility library. However, in Sh sequences of operations on these types can also be recorded in a retained mode and dynamically cross compiled to another target using the *SH_BEGIN_PROGRAM* and *SH_END* keywords. The *SH_BEGIN_PROGRAM* keyword takes a string parameter that specifies the compilation target. A value of `"gpu:vertex"` indicates compilation to the

vertex shading unit of the currently installed GPU, while "gpu:fragment" indicates compilation to the fragment unit of the currently installed GPU. Compilation to the CPU is also supported for stream processing targets, which we will discuss later. The *SH_BEGIN_PROGRAM* keyword returns an object of type *ShProgram* which represents the compiled shader.

Consider Listing 2, which implements vertex and fragment programs that support point and normal transformation as well as a modified Blinn-Phong lighting model for point light sources. We use C++ metaprogramming to unroll a loop over the light sources and build different versions of the shader supporting different numbers of point sources. A typical rendering (with one source) is given on the left hand side of Figure 2.

```
vertex_shader = SH_BEGIN_PROGRAM("gpu:vertex") {
  // declare input vertex attributes (unpacked in order given)
  ShInputNormal3f nm;          // normal vector (MCS)
  ShInputPosition3f pm;        // position (MCS)
  // declare outputs vertex attributes (packed in order given)
  ShOutputNormal3f nv;         // normal (VCS)
  ShOutputPoint3f pv;          // position (VCS)
  ShOutputPosition4f pd;       // position (DCS)
  // specify computations
  pv = MV | pm;                           // VCS position
  pd = VD | pv;                           // DCS position
  nv = normalize(nm | inverse_MV);        // VCS normal
} SH_END;
fragment_shader = SH_BEGIN_PROGRAM("gpu:fragment") {
  // declare input fragment attributes (unpacked in order given)
  ShInputNormal3f nv;       // normal (VCS)
  ShInputPoint3f pv;        // position (VCS)
  ShInputPosition3f pd;     // fragment position (DCS)
  // declare output fragment attributes (packed in order given)
  ShOutputColor3f fc;       // fragment color
  // compute unit normal and view vector
  nv = normalize(nv);
  vv = -normalize(ShVector3f(pv));
  // process each light source
  for (int i=0; i<NLIGHTS; i++) {
    // compute per-light normalized vectors
    ShVector3f lv = normalize(light_position[i] - pv);
    ShVector3f hv = normalize(lv + vv);
```

```
    ShColor3f ec = light_color[i] * max(0.0,(nv|lv));
    // sum up contribution of light source
    fc += ec*(kd + ks*pow(pos(hv|nv),exp));
  }
} SH_END;
```

Listing 2: Vertex and fragment shaders for the Blinn-Phong lighting model using multiple point light sources.

Note the use of *Input* and *Output* binding type modifiers to modify the declaration of some of the types inside each shader. Conceptually, *ShProgram*s represent functions that are applied in parallel to streams of records, each record containing k objects of various types, and produces another stream of records, with each output record containing m objects of various types. For instance, a vertex shader takes the vertex attributes bound to each input vertex by the user and produces another set of attributes that will be interpolated by the rasterizer, and this computation is repeated for each vertex. Likewise, the fragment shader takes as input interpolated values bound to fragments and computes output values (in this case only one color tuple per fragment, but multiple output values are possible in Sh, even if the hardware does not support them directly). In general, we refer to values that are presented as inputs and outputs to *ShProgram* objects as *attributes*.

Tuples and matrices declared without qualifiers are temporaries local to the shader. They are initialized to zero at the start of every invocation of a shader, which in this case simplifies the accumulation of contributions from multiple light sources. Sh also permits reading from output tuples and writing to input tuples (the latter does not change the real input data; shaders are pass by value). If the target platform does not support this, then Sh automatically introduces an appropriate temporary.

Other values are used in this computation. These are highlighted in italics in the listings: *MV*, *light_position*, *kd*, and so forth. These are Sh matrix and tuple types declared external to a shader. Using an externally declared Sh object in an *ShProgram* definition means that the values of that object should be made available for use by the shader as a "global variable". These objects cannot be assigned to inside the shader, but immediate-mode assignments to tuples and matrices referenced by a program will update these values for the next evaluation of the program. We call Sh objects used in this manner *parameters*.

The scope rules and modularity constructs of C++ can be used to con-

trol which parameters get bound to which shaders. Normally we would define shaders inside a framework that encapsulates parameters and program objects. A simple example is given in Listing 3. Here we declare transformation parameters in the *BaseShader* abstract class, point light parameters for `NLIGHT` light sources in the *PointLightShader* templated subclass, and finally the Blinn-Phong specific parameters in the *BlinnPhong* subclass. The constructor *BaseShader* calls the initialization method, eventually defined in the *BlinnPhong* concrete class, which constructs the *ShProgram* objects `vertex_shader` and `fragment_shader`. The `bind` member function loads these shaders onto the GPU when called. Such encapsulation is not mandatory. On the other hand, more sophisticated frameworks are certainly possible. Sh provides some optional frameworks that can provide various shader management facilities. Many other programming and encapsulation techniques are enabled by the close binding between C++ and Sh and the semantic similarity of *ShProgram* definitions to dynamic function definitions with static binding to parameters.

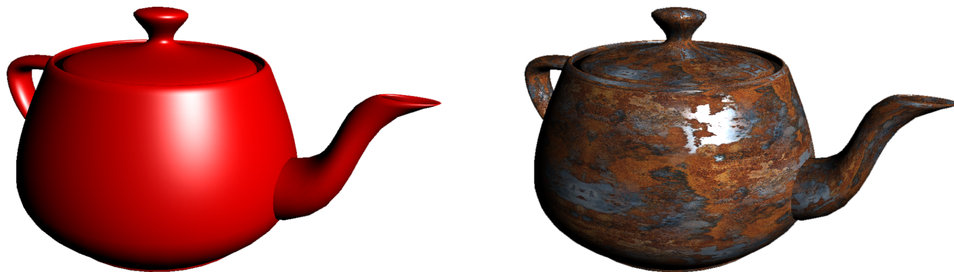


Figure 2: The Blinn-Phong lighting model, with specular and diffuse textures on the right.

```
class BaseShader {
  public:
    static ShMatrix4x4f VD;              // VCS to DCS
    static ShMatrix4x4f MV;              // MCS to VCS
    static ShMatrix4x4f MD;              // MCS to DCS
    static ShMatrix4x4f inverse_MV;      // MCS from VCS
    ShProgram vertex_shader;
```

```
    ShProgram fragment_shader;
    BaseShader();
    void bind();
    virtual void init() = 0;
};
template <int NLIGHTS>
class PointLightShader: public BaseShader {
  public:
    static ShPoint3f light_position[NLIGHTS];
    static ShColor3f light_color[NLIGHTS];
    PointLightShader();
};
template <int NLIGHTS>
class BlinnPhongShader: public PointLightShader<NLIGHTS> {
  public:
    ShColor3f ks;      // specular color
    ShColor3f kd;      // diffuse color
    ShAttrib1f exp;    // exponent
    BlinnPhongShader();
    virtual void init();
};
```

Listing 3: Framework classes for managing parameters.

Introspection is also supported on *ShProgram* objects. Member functions are supported for iterating over inputs, outputs, and parameters, and recalling metadata for each. Sh supports built-in metadata such as type, name, and description, but also provides a mechanism for the application to bind arbitrary metadata to tuples, matrices, and other Sh types, then retrieve this data from *ShProgram*s that use them.

4 Arrays and Textures

Texture maps in Sh are supported as template classes which take the type they store as a template argument. For instance, if we want to create a texture that stores 3-channel unsigned byte color data in a 3D grid, we would declare an *ShTexture3D<ShColor3ub>*. If we wanted a cube map of floating-point vectors, we would declare an *ShTextureCube<ShVector3f>*.

Lookups on textures are supported with the "()" and "[]" operators.

These are slightly different. The "`()`" operator treats the lookup as if the texture were a tabulated function, and the function were resolution independent. It therefore uses a normalized texture coordinate range of $[0,1] \times [0,1]$. However, if "`[]`" is used, Sh centers texels at the integers, although interpolation is still performed.

An example shader using texture lookup is given in Listing 4; this is a small modification of the shader given in Listing 2. All we have done is passed through texture coordinates in the vertex shader (demonstrating the ***InOut*** binding type, which declares both an input and an output) and converted ***ks*** and ***ks*** to textures. An example rendering is given on the right of Figure 2.

```
template <int NLIGHTS>
class TexturedBlinnPhongShader
  : public PointLightShader<NLIGHTS> {
public:
  ShTexture2D<ShColor3f> ks;  // specular texture
  ShTexture2D<ShColor3f> kd;  // diffuse texture
  ShAttrib1f q;               // exponent
  TexturedBlinnPhongShader();
  void init() {
   vsh = SH_BEGIN_PROGRAM("gpu:vertex") {
    // declare input vertex attributes
    ShInOutTexCoord2f u;        // texture coordinate
    ShInputNormal3f nm;         // normal vector (MCS)
    ShInputPosition3f pm;       // position (MCS)
    // declare output vertex attributes
    ShOutputNormal3f nv;        // normal (VCS)
    ShOutputPoint3f pv;         // position (VCS)
    ShOutputPosition4f pd;      // position (DCS)
    // specify computations
    pv = (MV|pm)(0,1,2);        // VCS position
    pd = VD|pv;                 // DCS position
    nv = normalize((nm|VM)(0,1,2));    // VCS normal
   } SH_END;
   fsh = SH_BEGIN_PROGRAM("gpu:fragment") {
    // declare input fragment attributes
    ShInputTexCoord2f u;     // texture coordinate
    ShInputNormal3f nv;      // normal (VCS)
    ShInputPoint3f pv;       // position (VCS)
    ShInputPosition3f pd;    // fragment position (DCS)
```

```
      // declare output fragment attributes
      ShOutputColor3f c;         // fragment color
      // compute unit normal and view vector
      nv = normalize(nv);
      vv = normalize(-pv);
      // process each light source
      for (int i=0; i<NLIGHTS; i++) {
        // compute per-light normalized vectors
        ShVector3f lv = normalize(lights[i].pv - pv);
        ShVector3f hv = normalize(lv + vv);
        ShColor3f ec = lights[i].c * pos(nv|lv);
        // sum up contribution of light source
        c += ec*kd(u) + ks(u)*pow(pos(hv|nv),q);
      }
    } SH_END;
  }
};
```

Listing 4: Textured Blinn-Phong shader.

Texture types are supported for 1D, 2D (both square and rectangular), 3D, and cube textures. Three major classes of texture types are supported. The *ShArray* types (*ShArray1D*, *ShArray2D*, *ShArrayRect*, etc.) only support nearest-neighbor lookup and no filtering. The *ShTable* types support bilinear interpolation but not filtering. Finally, *ShTexture* types support MIP-map filtering with trilinear interpolation. Other minor modes or variations on these modes are supported with template trait modifiers. Suppose you want a MIP-filtered *ShArray2D*. Then you could use the type *ShMIPFilter<ShArray2D<T> >*. Other trait modifiers can be used to set wrap and clamping modes and detailed interpolation modes.

You can define your own data abstractions by encapsulating texture types and providing your own access functions. For intance, consider Listing 5, which implements homomorphically factorized reflectance models [3], using a parabolic representation of the hemispherically parameterized functions involved. Figure 3 gives some renderings using homomorphically factorized materials.

```
class HfShader: public PointLightShader<1> {
public:
  ShTexture2D<ShColor3f> p;
  ShTexture2D<ShColor3f> q;
```

```
ShColor3f a;
HfShader ();
void init() {
 vsh = SH_BEGIN_PROGRAM("gpu:vertex") {
  // declare input vertex attributes (unpacked in order given)
  ShInputVector3f tm;           // primary tangent (MCS)
  ShInputVector3f sm;           // secondary tangent (MCS)
  ShInputPosition3f pm;         // position (MCS)
  // declare output vertex attributes (packed in order given)
  ShOutputVector3f vs;          // view vector (SCS)
  ShOutputVector3f ls;          // light vector (SCS)
  ShOutputColor3f ec;           // irradiance
  ShOutputPosition4f pd;        // position (HDCS)
  // compute transformations
  ShPoint3f pv = (MV|pm)(0,1,2);      // VCS position
  pd = MD|pv;                         // DCS position
  // find surface frame
  ShVector3f tv = normalize((MV|tm)(0,1,2));
  ShVector3f sv = normalize((MV|sm)(0,1,2));
  ShNormal3f nv = normalize(tv^sv);   // compute normal
  // compute irradiance
  ShVector3f lv = normalize(light.pv - pv);
  ec = light.c * pos(nv|lv);
  // compute SCS view and light vectors
  ShVector3f vv = normalize(-pv);
  ls = ShVector3f(tv|lv,sv|lv,nv|lv);
  vs = ShVector3f(tv|vv,sv|vv,nv|vv);
 } SH_END;
 fsh = SH_BEGIN_PROGRAM("gpu:fragment") {
  // declare input fragment attributes (unpacked in order given)
  ShInputVector3f vs;         // view vector (SCS)
  ShInputVector3f ls;         // light vector (SCS)
  ShInputColor3f ec;          // irradiance
  ShInputPosition3f pd;       // fragment position (DCS)
  // declare output fragment attributes (packed in order given)
  ShOutputColor3f c;          // final color
  // compute normalized vectors
  ls = normalize(ls);
  vs = normalize(vs);
  ShVector3f hs = normalize(ls + vs);
```

```
    c = a * ec * p(parabolic(vs))
               * q(parabolic(hs))
               * p(parabolic(ls));
  } SH_END;
 }
};
```

Listing 5: Vertex and fragment shaders for homomorphic factorization.

We can encapsulate the representation of the BRDF in a class, as shown in Listing 6. We could easily define other classes with the same base class to represent reflectance models in different ways; for instance, we might define classes that use spherical harmonics, Lafortune lobes [1], or parameterized lighting models. It should be noted, however, that a real data abstraction would provide stronger data hiding by providing copy constructors and access methods rather than making data members public. Similar techniques can be used to build new texture types, for instance sparse textures, cubically interpolated textures, or silhouette map textures.

```
// Texture abstraction: radially symmetric 2D texture
template <typename ELEM>
class RadialTexture2D {
 ShTexture1D<ELEM> tex;
 ELEM operator() (ShTexCoord2f u) const {
   ShTexCoord1f r = 2.0*length(u-ShTexCoord2f(0.5,0.5));
   return tex(r);
 }
};
// BRDF abstraction: HF representation
template <typename TEXTURE>
class HfBRDF: public BRDF {
public:
 TEXTURE p, q;
 ShColor3f a;
 HfBRDF();
 ShColor3f operator() (ShVector3f vs, ShVector3f ls) const {
  ls = normalize(ls);
  vs = normalize(vs);
  ShVector3f hs = normalize(ls + vs);
  ShColor3f c = p(parabolic(vs))
               * q(parabolic(hs))
```

```
              * p(parabolic(ls));
  return c * a;
 }
};
template <typename F>
class BRDFShader: public PointLightShader<1> {
public:
 BRDFShader();
 void init() {
  vsh = SH_BEGIN_PROGRAM("gpu:vertex") {
   ...  // as before
  } SH_END;
  fsh = SH_BEGIN_PROGRAM("gpu:fragment") {
   // declare input fragment attributes (unpacked in order given)
   ShInputVector3f vs;        // view vector (SCS)
   ShInputVector3f ls;        // light vector (SCS)
   ShInputColor3f ec;         // irradiance
   ShInputPosition3f pd;      // fragment position (DCS)
   // declare output fragment attributes (packed in order given)
   ShOutputColor3f fc;        // fragment color
   // multiply BRDF by irradiance
   fc = ec * F(vs,ls);
  } SH_END;
 }
};
```

Listing 6: Data abstraction for the homomorphically factorized BRDF representation.

5 Channels and Streams

To extend Sh to stream computation and provide facilities for manipulating programs dynamically, two extra operators are defined that act on program objects: the connection operator "<<", defined as functional composition or application, and the combination operator "&", which is equivalent to the concatenation of the source code of programs. These operators can also be used with stream abstractions based on the *ShStream* and *ShChannel* types, but we will describe their effect on program objects first.

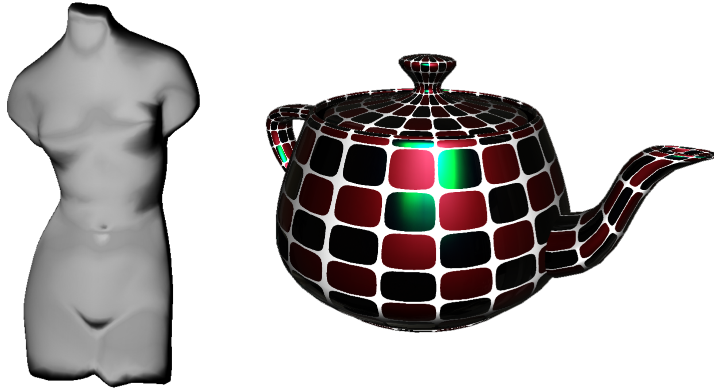


Figure 3: Homomorphically factorized materials. On the left, satin, on the right, a combination of three materials (garnet red, satin, and mystique).

Suppose we have a program `q1` with n inputs and k outputs and another program `p1` with k inputs and m outputs. The "`<<`" operator creates a new program object with n inputs and m outputs by taking the outputs of `q1` and feeding them into the inputs of `p1`.

Suppose we are given two more programs `p2` and `q2`. Let `p2` have n inputs and m outputs, and let `q2` have k inputs and ℓ outputs. Applying the "`&`" operator to `p2` and `q2` results in a new program with $n+k$ inputs and $m+\ell$ outputs. This new program has all the inputs, outputs, and computations of the original programs.

Because of the way Sh is defined, the operator "`&`" is in fact equivalent to the concatenation of the source code of the input programs, using two separate scopes. Such a concatenation would ensure that the inputs and outputs of `p2` are declared before `q2`, and so would give the same result as defined above.

The use of these operators to combine Sh programs can result in redundant computation. However, the "`<<`" operator, in conjunction with the optimizer in the Sh compiler (particularly dead code removal) and the definition of some simple "glue" programs, can be used to eliminate such redundant computations. Both operators actually operate on the internal representation of Sh programs to build a completely new program, which is ultimately run through the full suite of optimizations and virtualizations supported by the Sh backend. This has important implications.

For instance, suppose we combine two programs with "&" and the resulting program computes the same value twice (in two different ways, so we cannot discover this fact using common subexpression elimination). We can define a simple program that copies its inputs to its outputs *except* for one of the redundant results. This "glue" program can be connected to the output of the combined program and the Sh dead code eliminator will remove the redundant computation.

Unfortunately, to satisfy the type rules for connecting Sh programs, we need to define the interface of each such glue program to match the particular interface types of the given base program. This is annoying if all we want to do is rearrange the inputs and outputs.

Sh provides some shortcuts, similar to manipulators in the C++ `iostream` library, for manipulating the input and output channels of Sh programs: deleting outputs, reordering inputs and outputs, replacing inputs with texture lookups, and so forth. These manipulators are really functions that return instances of either program objects or instances of special manipulator classes. Manipulator classes store information about the particular manipulation required. When combined with a program object in an expression, the appropriate glue program is automatically generated, using introspection over the program objects the manipulator is combined with, to perform the desired manipulation. This approach can automatically resolve type issues.

Sh supports a stream computation model for general purpose computation based on the extension of these two operators to data. Stream objects are represented in Sh using the `ShChannel` template class and the `ShStream` class. A channel is a sequence of elements of the type given as its template argument. Streams are containers for several channels of data, and are specified by combining channels (or other streams) with the "&" operator.

Streams only *refer* to channels, they do not create copies. A channel can still be referenced as a separate object, and can also be referenced by more than one stream at once. For convenience, an `ShChannel` of any type can also be used directly as a single-channel stream. Streams may not refer to themselves as components.

In addition to being viewed as a sequence of channels, a stream can also be seen as a sequence of homogeneous records, each record being a sequence of elements from each component channel. Stream programs conceptually map an input record type to an output record type, and are applied in parallel to all records in the stream. If an `ShProgram` is compiled with the `"gpu:stream"` or `"cpu:stream"` profile, it can be applied to streams.

The "<<" operator is overloaded to permit the application of stream programs to streams. For instance, a program `p` can be applied to an input stream `a` and its output directed to an output stream `b` as follows:

```
b = p << a;
```

When specified, the above stream operation will execute immediately.

Use of "`p << a`" alone creates an unevaluated program, which is given the type *ShProgram* (and can be assigned to a variable of this type, if the user does not want execution to happen immediately). What actually happens is that input attributes are replaced with fetch operators in the intermediate language representation of the program. These fetch operators are initialized to refer to the given stream's channels. Such program objects can also be interpreted as "procedural streams". Only when an unevaluated procedural stream is assigned to a concrete output stream will the computation actually be executed.

The implementation of the << operator permits partial evaluation (*currying*). You do not have to supply all the inputs to a program at one time. If a stream program is applied to a stream with an insufficient number of channels, an unevaluated program with fewer inputs is returned. This program requires the remainder of its inputs before it can execute.

Currying is a concept borrowed from functional languages. In a functional language, currying is usually implemented with deferred execution. Since in a pure functional language values in variables cannot be changed after they are set, this is equivalent to using the value in effect at the point of the partial evaluation. However, in an imperative language, we are free to modify the value provided to the partially evaluated expression before final evaluation is performed. We *could* copy the value at the point of the partial evaluation, but this would be expensive for stream data. Instead, we use deferred read semantics: later execution of the program will use the value of the stream in effect at the point of actual execution, *not* the value in effect at the point of the partial evaluation. This is useful in practice, as we can create (and optimize) a network of kernels and streams in advance and then execute them iteratively.

The "<<" operator can also be used to apply programs to Sh tuples. A mixture of tuple and stream inputs may be used. In this case, the tuple is interpreted as a stream all of whose elements are the same value. The same by-reference semantics are applied for consistency. In fact, what happens is that an input "varying" attribute is converted into a "uniform" parameter,

a useful operation.

Since we provide an operator for turning a varying attribute into a uniform parameter, we also provide an inverse operator for turning a parameter into an attribute. Given program `p` and parameter `x`, the following removes the dependence of `p` on `x`, creating a new program object `q`:

```
ShProgram q = p >> x;
```

The parameter is replaced by a new attribute of the same type, pushed onto the *end* of the input attribute list.

The "`&`" operator can also be applied to streams, channels, or tuples on the left hand side of an assignment. This can be used to split apart the output of a stream program. For instance, let `a`, `b`, and `c` be channels or streams, and let `x`, `y`, and `z` be streams, channels, or tuples. Then the following binds a program `p` to some inputs, executes it, and extracts the individual channels of the output:

```
(a & b & c) = p << x << y << z;
```

This syntax also permits Sh programs to be used as subroutines (let all of `a`, `b`, `c`, `x`, `y`, and `z` be tuples).

Listing 7 defines a stream program to update the state of a particle system in parallel [8]. This kernel implements simple Newtonian physics and can handle collisions with both planes and spheres. The particles are then rendered as point sprites by feeding the positions of the particles back through the GPU as a vertex array (code for this is not shown). Screenshots are shown in Figure 4.

This example demonstrates the use of deferred read semantics for currying. The `state` stream is defined and bound to the `particle` program along with some uniform parameters. The result is assigned to the `update` program object, which triggers compilation and optimization. The `update` object now has compiled-in access to the channels pointed to by the `state` stream. The inner loop is very simple and fast. In particular, all shader compilation is done during setup.

```
// SETUP
particle = SH_BEGIN_PROGRAM("gpu:stream") {
  ShInOutPoint3f Ph;   // head position
  ShInOutPoint3f Pt;   // tail position
  ShInOutVector3f V;   // velocity
  ShInputVector3f A;   // acceleration
```

```
  ShInputAttrib1f delta;  // timestep
  Pt = Ph; // Physical state update
  A = cond(abs(Ph(1)) < 0.05, ShVector3f(0.,0.,0.), A);
  V += A * delta;
  V = cond((V|V) < 1.0, ShVector3f(0.0, 0.0, 0.0), V);
  Ph += (V - 0.5 * A) * delta;
  ShAttrib1f mu(0.1), eps(0.3);
  for (int i = 0; i < num_spheres; i++) { // Sphere collisions
    ShPoint3f C = spheres[i].center;
    ShAttrib1f r = spheres[i].radius;
    ShVector3f PhC = Ph - C;
    ShVector3f N = normalize(PhC);
    ShPoint3f S = C + N * r;
    ShAttrib1f collide = ((PhC|PhC) < r*r) * ((V|N) < 0);
    Ph = cond(collide, Ph - 2.0 * ((Ph - S)|N) * N, Ph);
    ShVector3f Vn = (V|N) * N;
    ShVector3f Vt = V - Vn;
    V = cond(collide, (1.0 - mu) * Vt - eps * Vn, V);
  }
  ShAttrib1f under = Ph(1) < 0.0; // Collide with ground
  Ph = cond(under, Ph * ShAttrib3f(1.0, 0.0, 1.0), Ph);
  ShVector3f Vn = V * ShAttrib3f(0.0, 1.0, 0.0);
  ShVector3f Vt = V - Vn;
  V = cond(under, (1.0 - mu) * Vt - eps * Vn, V);
  Ph(1) = cond(min(under,
                (V|V) < 0.1), ShPoint1f(0.0f), Ph(1));
  ShVector3f dt = Pt - Ph; // Avoid lines disappearing
  Pt = cond((dt|dt) < 0.02,
             Pt + ShVector3f(0.0, 0.02, 0.0), Pt);
} SH_END;
// define stream specifying current state
ShStream state = (pos & pos_tail & vel);
// define update operator, bind to inputs
ShProgram update = particle << state << gravity << delta;
// IN INNER LOOP
// execute state update (input to update is compiled in)
state = update;
```

Listing 7: Particle system simulation.

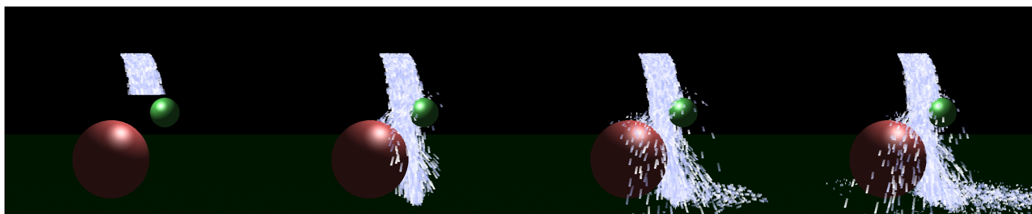


Figure 4: Frames from the particle system animation corresponding to Listing 7.

6 Tools and Engineering

The Sh implementation includes three major components: the front end, the optimizer, and the backend.

The front end supports the API described here, and generates an intermediate representation of programs. Our intermediate representation is similar to the ARB assembly language, but is somewhat higher-level. When *SH_BEGIN_PROGRAM* is called, a new intermediate representation is initialized, and a global flag is set so that Sh classes know they are inside a program definition. When operations are specified inside a program definition, an appropriate instruction is added to the intermediate representation. We do not build a parse tree, since operations are already called in postfix order. This process builds a correct but inefficient program. In particular, there are a lot of extraneous moves due to constructors and function parameter passing. However, these extra operations, as well as many other sources of inefficiency, will be cleaned up later by the optimizer.

When an externally declared parameter or texture is used in a program, a dependency is set up. The visible Sh classes are really only smart pointers; the real data is kept elsewhere, and is reference counted. Therefore, even if a program refers to a parameter that is later destroyed, the value of that parameter will still be available. When a shader is loaded, it goes over its list of dependencies and notifies them that it is active. Whenever a parameter is modified in immediate mode, it checks for active dependencies, and updates the appropriate constant registers as necessary. Updates are done lazily. When coordinating Sh with a graphics API, an update function must be called once all parameters have been modified. Lazy update is especially important for large parameters that are expensive to download, such as tex-

tures. Sh also shadows textures and streams on both the host and the GPU (or other target processor) and keeps track of which is the most recently modified version.

The optimizer is a crucial component of Sh. As of the current release, it supports forward substitution, dead code removal, constant propagation, and uniform lifting. These are not just nice features, they are essential. For instance, dead code elimination is used for virtualization and specialization. Uniform lifting finds computations in shaders that depend only on uniform parameters and moves them to the host, creating a set of hidden, dependent uniform parameters.

The backends map the intermediate representation to a particular target platform, and provide runtime support. In order to support stream processing and data-dependent control flow, we may have to decompose shaders into multiple parts and schedule these parts over multiple passes. The backends also do detailed buffer management, automatically using appropriate graphics API extensions for this purpose when available. Multiple GPU backends are available that support different runtime strategies or programming interfaces. Currently we support ARB assembly, NV assembly, or the OpenGL Shading Language to program a GPU internally. In this tutorial, we have only used the default GPU and CPU compilation targets, but a more detailed syntax is available for selecting alternatives. A CPU backend is also available that supports the dynamic compilation and linking of vectorized host code.

Some development and test tools are available for use with Sh. One example is `shrike`, shown in Figure 5. The `shrike` application provides a class framework for shaders and a browser interface to compare multiple shaders. It uses introspection to determine the parameters used for each shader and uses the metadata for each parameter to build a user interface for those parameters.

7 License, History, Status, and Access

The core of Sh is a free Open Source project, distributed under a license which reads as follows:

> *Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:*

Figure 5: Glass and silhouetted Gooch shaders, demonstrated inside `shrike`.

1. *The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.*
2. *Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.*
3. *This notice may not be removed or altered from any source distribution.*

However, the source is copyright Serious Hack Inc., and all rights to the source are owned by this company. Anyone wishing to contribute to the official code base must transfer IP rights to the company. This is necessary for Sh to be usable in commercial applications, and protects the IP for other open-source uses. Although the core is free, the company is committed to providing continuing and ongoing support for the language, and is also developing a professional version with added (but backward-compatible) extensions.

The name of the language does conflict with the Bourne Shell, but this was an unfortunate historical accident. Sh was developed in the Computer Graphics Lab at the School of Computer Science at the University of Waterloo. The SMASH API [2, 7] was a conceptual predecessor to Sh, but targetted a simulated GPU, since no GPUs were available at the time that

supported the necessary features. This system was subsequently factored into two projects: a GPU simulator component called Sm (which Sh can still target, but which it does not depend on) and a high-level programmable API/language component called Sh. The prototype of the current design of Sh was implemented as part of Zheng Qin's M.Math thesis. The current version of Sh was significantly reengineered by Stefanus Du Toit, now a graduate student at the University of Waterloo, and co-founder of Serious Hack Inc. Other contributors include Kevin Moule (just-in-time CPU backend), Bryan Chan (kernel library and utility functions), Jack Wang (matrix and quaternion library), and Zaid Mian (noise functions). Michael McCool is the team leader and President of Serious Hack Inc., and is responsible for the design of the language (he even occasionally gets to write some code).

Sh is under active development. The core of the language is stable (and there is an extensive regression test suite in place to make sure it stays that way) but we are working on improving performance and stream programming functionality. It will continue to improve in performance and capabilities, but it is the intent of the developers that all further extensions to the language will be backward compatible with the current version, as much as possible, and that the core of the implementation will always be platform and graphics API independent, vendor neutral, and available under Open Source. Sh requires a floating-point capable GPU such as an NVIDIA GeForceFX 5200 (or better) or an ATI Radeon 9600 (or better).

The most recent version of Sh may be downloaded from `http://libsh.org/`. Mailing lists for news related to Sh and documentation may also be accessed at this site. Online documentation includes an up to date version of this tutorial, an HTML version of the reference manual, links to papers on Sh [4, 6], internal documentation generated from structured comments, and various other propaganda. A book on Sh called *Metaprogramming GPUs with Sh* [5] is also available from AK Peters via `http://www.akpeters.com`.

References

[1] E. Lafortune, S.-C. Foo, K. Torrance, and D. Greenberg. Non-linear approximation of reflectance functions. *Computer Graphics (Proc. SIGGRAPH)*, pages 117–126, August 1997. 14

[2] Michael McCool. SMASH: A next-generation API for programmable graphics accelerators. Technical Report CS-2000-14, University of Waterloo, April 2001. API Version 0.2. Presented at SIGGRAPH 2001 Course #25, *Real-Time Shading*. 23

[3] Michael McCool, Jason Ang, and Anis Ahmad. Homomorphic factorization of BRDFs for high-performance rendering. *ACM Trans. on Graphics (Proc. SIGGRAPH)*, pages 171–178, August 2001. 12

[4] Michael McCool, Zheng Qin, and Tiberiu Popa. Shader metaprogramming. In *Proc. Graphics Hardware*, pages 57–68, September 2002. 24

[5] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters, 2004. 24

[6] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. on Graphics (Proc. SIGGRAPH)*, August 2004. 24

[7] Marc Olano, John C. Hart, Wolfgang Heidrich, and Michael McCool. *Real-Time Shading*. AK Peters, 2002. 23

[8] Karl Sims. Particle animation and rendering using data parallel computation. *Computer Graphics (Proc. SIGGRAPH)*, 24(4):405–413, August 1990. 19