http://www.rapidmind.net

Writing Shaders and General Purpose GPU Programs Using the RapidMind™ Development Platform

Table Of Contents

Introduction 2

Tuples, Matrices, Operators and Functions 4

Programs, Parameters and Attributes 6

Arrays, Tables and Textures 9

Streams 13

Engineering 17

References 18

Abstract

The RapidMind Development Platform is a commercial interface for high-level data-parallel programming embedded in C++. It can target a number of massively parallel processors and computer systems, including GPUs and the Cell BE processor. The RapidMind platform provides a uniform and portable interface vocabulary for a range of consumer, embedded and scientific applications requiring high performance. On the GPU, this interface vocabulary can be used for specifying and manipulating both shaders and general-purpose programs.

This tutorial presents the basic types supported by the interface, tuples and matrices, and shows how they can be used in immediate mode to perform computations on the host. We will then show how program objects are created by capturing sequences of operations on these types using a retained mode "recording" mechanism. Program objects can be executed on the GPU as either shaders or stream programs.

A number of examples will be given to demonstrate how the interface can be used to program both shaders and general-purpose stream computations.

Introduction

The RapidMind Development Platform is an interface for high-level data-parallel programming, consisting of a vocabulary of types and operations embedded in C++. It can target a number of massively parallel processors and computer systems, including but not limited to GPUs and the Cell BE processor. The RapidMind platform provides a uniform and portable interface for a range of consumer, embedded, and scientific applications requiring high-performance computation. This document is aimed at developers targeting graphics and general-purpose applications specifically on GPUs. It is assumed that the reader has a background in both C++ and graphics. Examples will be drawn from interactive graphics that highlight the advanced programming capabilities of the RapidMind platform.

On GPUs, the RapidMind platform can be used for *both* shaders and general-purpose programs. We use a stream processing model of programming that is well-suited to efficient execution on GPU architectures. As a shader language, the RapidMind platform is both complete and more expressive than any other shading language. It provides the full power of C++ for object-oriented data abstraction while completely eliminating the need for "glue" code. In its general purpose mode, the RapidMind platform enables the use of the GPU as a high-performance numerical co-processor. Shaders are in fact just a special case of the more general purpose concept of a stream program.

Figure 1: Some procedural shaders implemented with our platform.

The RapidMind platform has a unique interface for specifying computations. It is packaged as library, but with the functionality and ease of use of a specialized programming language. It includes a dynamic just-in-time compiler. Unlike other libraries its functionality is completely open-ended. It is easiest to think about the RapidMind platform as being a *vocabulary* for data-parallel programming. Rather than designing a separate language to specify parallel computations, we have just added a small number of nouns (types) and verbs (operations) to an existing language, C++. We have used the standard features of C++ to do this, so the RapidMind platform can be used with a variety of ISO-compliant compilers and your existing development

tool chain. The RapidMind platform can be seen either as an advanced API for specifying high-performance computations or a just-in-time compiler whose "parser" is implemented using C++ operator overloading.

The structure of our interface has major advantages in practice. Custom languages that are compiled separately from the host language require extra "glue" code to interface the computation to the host program. The RapidMind system requires no glue code at all: the interface *is* the language.

Our system is also unique in that it targets multiple computational resources with a common interface, including GPUs, the Cell BE processor, and multi-core CPUs. You can use it to write shaders in a graphics application or pure numerical computations in a scientific application, or both, as in a game or a visualization application. In the following the focus is on the GPU as a target, but it should be emphasized that most of the code and programming techniques presented will run on any of the supported targets.

The RapidMind system looks at first like a standard operator overloaded numerical/graphics library, with matrices, points and vectors, and you can use it that way if you want. We intentionally gave it this interface to make it intuitive to use. However, the RapidMind platform supports an additional feature with deep implications: you can also *record* sequences of operations, and compile these recorded operations into a "program object". Program objects can then be run on a variety of target processors, including the GPU and the host CPU. In essence, we have added the ability to create new functions at runtime to standard C++.

The RapidMind compiler subsystem is dynamic, and can take the (arbitrary) computations specified through the interface and generate optimized code on the fly. It supports a modular backend system that can compile the same computation to multiple co-processors, even from the same host program. In addition to the compiler, the RapidMind platform also includes a runtime engine. The runtime engine is designed to make best use of the available hardware features but without burdening the programmer with low-level details. It manages memory as well as code, making it possible to program complex multipass algorithms and to encapsulate data representations in an object-oriented fashion. It should be emphasized, however, that our compiler records *only* the numerical computations specified by the host program, not the object-oriented overhead associated with C++. A programmer using our system can use object-oriented code structuring mechanisms without suffering a performance penalty in time-critical applications. The object-oriented structure serves only as scaffolding. For actual execution, this scaffolding is torn completely away by our compiler. You can structure your code with C++, but get the runtime performance of FORTRAN, an advantage even for CPU targets.

Programs written using the RapidMind platform can run on the GPU but act like extensions of the host application. The semantics of our vocabulary have been set up so that program objects act like C++ functions which follow the normal scope rules of C++. For instance, to communicate a value from the host to a co-processor, the code used in a program object simply has to refer to a non-local variable outside its definition, and the host only has to assign a new value to that variable whenever necessary. The system will automatically track dependencies between variables and when the variable is updated, will ensure that the code running on the co-processor can access the new value. We also manage remote memory buffers (such as textures on the GPU) so they act like arrays. The visibility of these arrays follows the same scope rules as other variables. In summary, the scope rules of C++ control which parameters and data arrays are visible to which program objects. Since all the modularity constructs of C++ are based on scope rules, our approach enables the use of *all* C++ modularity constructs for structuring computations: namespaces, templates, and classes can be used to organize and parameterize

program objects; you can define your own types and operators; and you can define libraries of functions and program objects.

The C++ control constructs can also be used to manipulate and construct program objects 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, to build custom "fast paths" based on runtime options and to generate program objects from data files read in at runtime or specified at runtime via a user interface.

For general-purpose computation, compiled program objects can be applied as functions to streams of data. These programs will execute on the designated co-processor without it ever being necessary for the user to make a co-processor specific API call. In the case of the GPU, it is unnecessary to use a graphics API if you are only interested in general-purpose computation. 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 on a co-processor 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 array and texture lookups, currying, functional composition, concatenation and other operations. These dynamic code manipulation capabilities are extremely powerful but are built upon a small number of simple operations we call a *program algebra*.

In the following we will introduce the RapidMind platform by working through a number of examples. First, we will present the basic types, tuples and matrices, and show how they can be used in immediate mode to perform computations on the host. Then we will show how program objects are created by capturing sequences of operations on these types using a retained-mode "recording" mechanism. We also show how the interface between program objects and the rest of the application is defined. Then we discuss how memory objects are managed and how our platform can be used to build data abstractions. For general-purpose computation, we support a stream processing model. Our stream processing abstraction also includes a program algebra which is useful for manipulating and transforming program objects, and a stride and offset algebra on arrays which is useful for expressing structured data access patterns.

Tuples, Matrices, Operators and Functions

The core types in the RapidMind platform are short sequences of numbers, or n-tuples. 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 are not limited in length), and a storage type (how the actual numbers are stored). For instance, **`Point3f`** represents a three-dimensional point stored as a triple of single precision floating-point numbers, whereas **`Color4ub`** is a four-component color whose components are stored as unsigned bytes. Floating point and integer types at a variety of precisions are available. All RapidMind types are declared in their own namespace to avoid name conflicts with user-defined types.

Generic tuples are given the semantic type of **`Value`**, for instance **`Value3f`**. These names are actually predefined type definitions wrapping a more general template mechanism for specifying tuples. For general purpose, non-graphics computations, the existing semantic types can be ignored and new, application-specific types can be derived. As much as possible, the type **`Value1f`** has the same semantics and follows the same syntactic rules as a `float`. For instance, to implement complex numbers, you can just use `std::complex<`**`Value1f`**`>`.

Operators are overloaded on these types. Arithmetic operators usually act component by component, 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 example, you can multiply a vector by a scalar (and that scalar can be a C++ floating-point literal scalar or a 1-tuple).

Swizzling is a useful operation that permits the flexible extraction and rearrangement of the components of tuples. The "`()`" notation is used for swizzling, using integers to index components. If `c` is a **`Color3f`** 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 (luminance plus transparency) 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. The RapidMind platform supports a large and growing number of other built-in functions, including noise and hashing functions of various kinds, tuple sorting, trigonometric, exponential, logarithmic, and geometric operations, smoothed discontinuities, and spline evaluators. To perform better on SIMD target machines, all functions in the standard library are designed to work without branches or loops. When possible the names of our standard library functions are similar to those in the C++ standard library or in other major shading languages (for graphics-oriented functions).

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 be equivalent to `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, the ternary operator "?:" cannot be overloaded in C++; `cond` is equivalent semantically) with both Boolean tuples and scalars.

Normal C++ syntax can be used to specify functions. For example, suppose you would like to simulate the appearance of glass. You will need a function to compute a reflection vector, a refraction vector and to determine the ratio between reflection and refraction (known as the Fresnel coefficient). Functions to compute these quantities 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 function definition 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++.

```
Vector3f
reflect (Vector3f v, Normal3f n) {
  return Vector3f(2.0*(n|v)*n - v);
}
```

Listing 1: A function to compute a reflection vector.

We also support small fixed-size matrices. In geometric applications like vision and graphics, these can be used to represent transformations of points and vectors. Matrix types have names like **`Matrix3x4f`** and can be square or rectangular. There is a set of standard library functions that support 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. For example, 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, reversing both the rows and columns of the result. 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 a **`Value4f`** and `M[2][3]` selects the scalar at row 2, column 3.

On the left hand side of an assignment, the same indexing operators can be used to selectively write to elements of both tuples and matrices. This can be interpreted as swizzling references to elements, and is commonly called *writemasking*. 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. Some semantic types support some additional special rules. In particular, if a point or vector 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 a **`Point3f`** by a **`Matrix4x4f`**, the point will be automatically extended with a homogeneous coordinate of 1. If a **`Vector3f`** is transformed the same way, it will be extended with a homogeneous coordinate of 0. This is one of the few places where the semantic type of a tuple matters—and note that the alternative would be to declare a size mismatch error.

Programs, Parameters and Attributes

The tuple and matrix types of our platform can be used in immediate mode as if they were a simple matrix/vector utility library with a relatively sophisticated syntax. However, sequences of operations on these types can also be *recorded* in a retained mode and dynamically cross compiled to another target using the **`BEGIN_PROGRAM`** and **`END`** keywords. The **`BEGIN_PROGRAM`** keyword takes a string parameter that specifies the compilation target. For example, 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. The **`BEGIN_PROGRAM`** keyword returns an object of type **`Program`** which represents the compiled program. We are using the short form of the keywords here; longer forms with prefixes are also available to avoid name conflicts in production code.

For another example, consider Listing 2. This example implements vertex and fragment programs that support point and normal transformation as well as a modified Blinn-Phong lighting model for point light sources. Shaders are just special cases of program objects. They are applied in parallel to streams of vertices or pixels as they are drawn to the screen in an interactive application on a GPU. We will show how our platform can apply program objects to arbitrary streams of data in a few pages.

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 light source) is given on the left hand side of Figure 2.

```
vertex_shader = BEGIN_PROGRAM("gpu:vertex") {
  // declare input vertex attributes (unpacked in order given)
  In<Normal3f> nm;      // normal vector (MCS)
  In<Position3f> pm;    // position (MCS)
```

```
    // declare outputs vertex attributes (packed in order given)
    Out<Normal3f> nv;      // normal (VCS)
    Out<Point3f> pv;       // position (VCS)
    Out<Position4f> pd;    // position (DCS)
    // specify computations
    pv = MV|pm;                    // VCS position
    pd = VD|pv;                    // DCS position
    nv = normalize(nm|inverse_MV); // VCS normal
} END;

fragment_shader = BEGIN_PROGRAM("gpu:fragment") {
    // declare input fragment attributes (unpacked in order given)
    In<Normal3f> nv;       // normal (VCS)
    In<Point3f> pv;        // position (VCS)
    In<Position3f> pd;     // fragment position (DCS)
    // declare output fragment attributes (packed in order given)
    Out<Color3f> fc;       // fragment color
    // compute unit normal and view vector
    nv = normalize(nv);
    vv = -normalize(Vector3f(pv));
    // process each light source (note that this is a metaprogrammed loop)
    for (int i =0; i < NLIGHTS; i++) {
      // compute per-light normalized vectors
      Vector3f lv = normalize(light_position[i] - pv);
      Vector3f hv = normalize(lv + vv);
      // access the light color (array access is metaprogrammed)
      Color3f ec = light_color[i] * max(0.0,(nv|lv));
      // sum up contribution of light source
      fc += ec *(kd + ks*pow(pos(hv|nv),q));
    }
} END;
```

Listing 2: Vertex and fragment shaders for the Blinn-Phong lighting model.

Note the use of **In** and **Out** binding type modifiers to modify the declaration of some of the types inside each shader. Conceptually, **Program** objects represent functions that are applied in parallel to streams of records, each record containing *k* objects of various types. This 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. 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 general, we refer to values that are presented as inputs and outputs to **Program** 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. We also support reading from output tuples and writing to input tuples (the latter does not change the real input data;

program objects always use pass by value). If the target platform does not support these semantics, our compiler automatically introduces an appropriate temporary.

Other values are used in this computation. These are highlighted in italics in the listings: the modelview matrix *MV*, the point source position *light_position*, the diffuse coefficient *kd*, the Phong exponent *q*, and so forth. These are instances of matrix and tuple types declared external to a shader. Using an externally declared object in a **Program** definition means that the values of that object should be made available for use by the shader as a "non-local 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 shader. We call objects used in this manner *parameters*.

The scope rules and modularity constructs of C++ can be used to control 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 **Program** 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. A wide variety of programming and encapsulation techniques are enabled by the close binding between C++ and the RapidMind platform and the semantic similarity of **Program** definitions to dynamic function definitions with static binding to parameters.

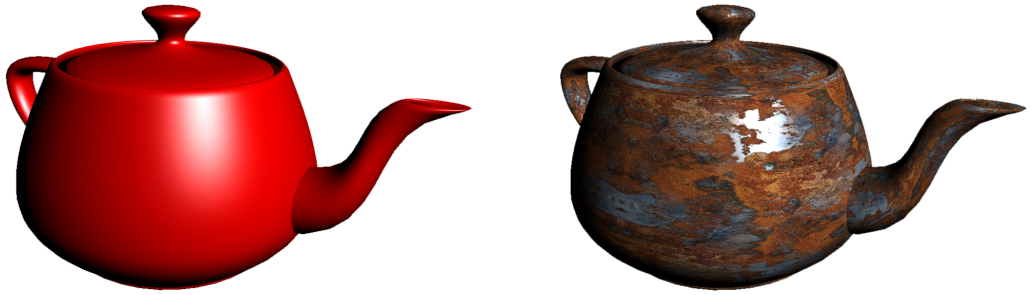

Figure 2: The Blinn-Phong lighting model, using both specular and diffuse textures on the right.

```
class BaseShader {
  public:
    static Matrix4x4f VD; // VCS to DCS
    static Matrix4x4f MV; // MCS to VCS
    static Matrix4x4f MD; // MCS to DCS
    static Matrix4x4f inverse_MV; // MCS from VCS
    Program vertex_shader;
    Program fragment_shader;
    void bind ();
    virtual void init () = 0;
    BaseShader ();
```

```
};
template <int NLIGHTS>
class PointLightShader: public BaseShader {
  public:
    static Point3f light_position[NLIGHTS];
    static Color3f light_color[NLIGHTS];
    PointLightShader ();
};
template <int NLIGHTS>
class BlinnPhongShader: public PointLightShader<NLIGHTS> {
  public:
    Color3f ks;     // specular color
    Color3f kd;     // diffuse color
    Value1f q;      // exponent
    BlinnPhongShader ();
    virtual void init ();
};
```

Listing 3: Framework classes for managing parameters.

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

Arrays, Tables and Textures

Texture maps 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 a **Texture3D**<**Color3ub**>. If we wanted a cube map of floating-point vectors, we would declare a **TextureCube**<**Vector3f**>. 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]×[0,1]. However, if "`[]`" is used, texels are centered 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 modifier, which marks a variable that is both an input and an output) and converted *ks* and *kd* to textures. An example rendering is given on the right of Figure 2.

```
template <int NLIGHTS>
class TexturedBlinnPhongShader: public PointLightShader<NLIGHTS> {
  public:
    Texture2D<Color3f> ks;  // specular texture
    Texture2D<Color3f> kd;  // diffuse texture
    Value1f q;  // exponent
    TexturedBlinnPhongShader ();
    void init () {
      vertex_shader = BEGIN_PROGRAM("gpu:vertex") {
        // declare input vertex attributes
```

```
        InOut<TexCoord2f> u;    // texture coordinate
        In<Normal3f> nm;        // normal vector (MCS)
        In<Position3f> pm;      // position (MCS)
        // declare output vertex attributes
        Out<Normal3f> nv;       // normal (VCS)
        Out<Point3f> pv;        // position (VCS)
        Out<Position4f> 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
      } END;

      fragment_shader = BEGIN_PROGRAM("gpu:fragment") {
        // declare input fragment attributes
        In<TexCoord2f> u;    // texture coordinate
        In<Normal3f> nv;     // normal (VCS)
        In<Point3f> pv;      // position (VCS)
        In<Position3f> pd;  // fragment position (DCS)
        // declare output fragment attributes
        Out<Color3f> 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
          Vector3f lv = normalize(light_position[i].pv - pv);
          Vector3f hv = normalize(lv + vv);
          Color3f ec = light_color[i].c * pos(nv|lv );
          // sum up contribution of light source
          c += ec*kd(u) + ks(u)* pow(pos(hv|nv),q);
        }
      } 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 **`Array`** types (**`Array1D`**, **`Array2D`**, **`ArrayRect`**, etc.) only support nearest-neighbor lookup and no filtering. The **`Table`** types support bilinear interpolation but not filtering. Finally, **`Texture`** types support MIP-map filtering with trilinear interpolation. Other minor modes or variations on these modes are supported with template trait modifiers.

You can define your own data abstractions by encapsulating texture types and providing your own access functions. For instance, 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 :
    Texture2D<Color3f> a;
    Texture2D<Color3f> b;
    Color3f k;
    HfShader ();
    void init () {
      vertex_shader = BEGIN_PROGRAM("gpu:vertex") {
        // declare input vertex attributes (unpacked in order given)
        In<Vector3f> tm;      // primary tangent (MCS)
        In<Vector3f> sm;      // secondary tangent (MCS)
        In<Position3f> pm;    // position (MCS)
        // declare output vertex attributes (packed in order given)
        Out<Vector3f> vs;     // view vector (SCS)
        Out<Vector3f> ls;     // light vector (SCS)
        Out<Color3f> ec;      // irradiance
        Out<Position4f> pd;   // position (HDCS)
        // compute transformations
        Point3f pv = (MV|pm)(0,1,2);  // VCS position
        pd = MD|pv ;  // DCS position
        // find surface frame
        Vector3f tv = normalize((MV|tm)(0,1,2));
        Vector3f sv = normalize((MV|sm)(0,1,2));
        Normal3f nv = normalize(tv^sv);  // compute normal
        // compute irradiance
        Vector3f lv = normalize (light.pv - pv);
        ec = light .c * pos(nv|lv);
        // compute SCS view and light vectors
        Vector3f vv = normalize(-pv);
        ls = Vector3f(tv|lv ,sv|lv ,nv|lv );
        vs = Vector3f(tv|vv ,sv|vv ,nv|vv );
      } END;
      fragment_shader = BEGIN_PROGRAM("gpu:fragment") {
        // declare input fragment attributes (unpacked in order given)
        In<Vector3f> vs;  // view vector (SCS)
        In<Vector3f> ls;  // light vector (SCS)
        In<Color3f> ec;  // irradiance
        In<Position3f> pd;  // fragment position (DCS)
        // declare output fragment attributes (packed in order given)
        Out<Color3f> c;  // final color
        // compute normalized vectors
        ls = normalize(ls);
        vs = normalize(vs);
        Vector3f hs = normalize(ls + vs);
        c = k * ec
          * a(parabolic(vs))
          * b(parabolic(hs))
          * a(parabolic(ls));
      } 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. A more complete implementation could 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 {
  protected:
    Texture1D<ELEM> tex;
  public:
    ELEM operator() (TexCoord2f u) const {
      TexCoord1f r = 2.0 * length(u - TexCoord2f(0.5, 0.5));
      return tex(r);
    }
};
// BRDF abstraction: HF representation
template <typename TEXTURE>
class HfBRDF: public BRDF {
  public:
    TEXTURE a, b;
    Color3f k;
    HfBRDF ();
    Color3f operator () (Vector3f vs, Vector3f ls) const {
      ls = normalize(ls);
      vs = normalize(vs);
      Vector3f hs = normalize(ls + vs);
      Color3f c = a(parabolic(vs))
                * b(parabolic(hs))
                * a(parabolic(ls));
      return c * k;
    }
};
template <typename F>
class BRDFShader: public PointLightShader<1> {
  public:
    BRDFShader ();
    void init () {
    vertex_shader = BEGIN_PROGRAM("gpu:vertex") {
      ... // as before
    } END ;
    Fragment_shader = BEGIN_PROGRAM("gpu:fragment") {
      // declare input fragment attributes (unpacked in order given)
      In<Vector3f> vs;     // view vector (SCS)
      In<Vector3f> ls;     // light vector (SCS)
      In<Color3f> ec;      // irradiance
      In<Position3f> pd;   // fragment position (DCS)
      // declare output fragment attributes (packed in order given)
      Out<Color3f> fc;     // fragment color
```

```
        // multiply BRDF by irradiance
        fc = ec * F(vs,ls);
      } END;
    }
};
```

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

Streams

To provide facilities for manipulating programs dynamically, two extra operators are defined that act on program objects and create new program objects: the connection operator "<<", which is 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 **`Stream`** and **`Array`** types, but we will describe their effect on program objects first.

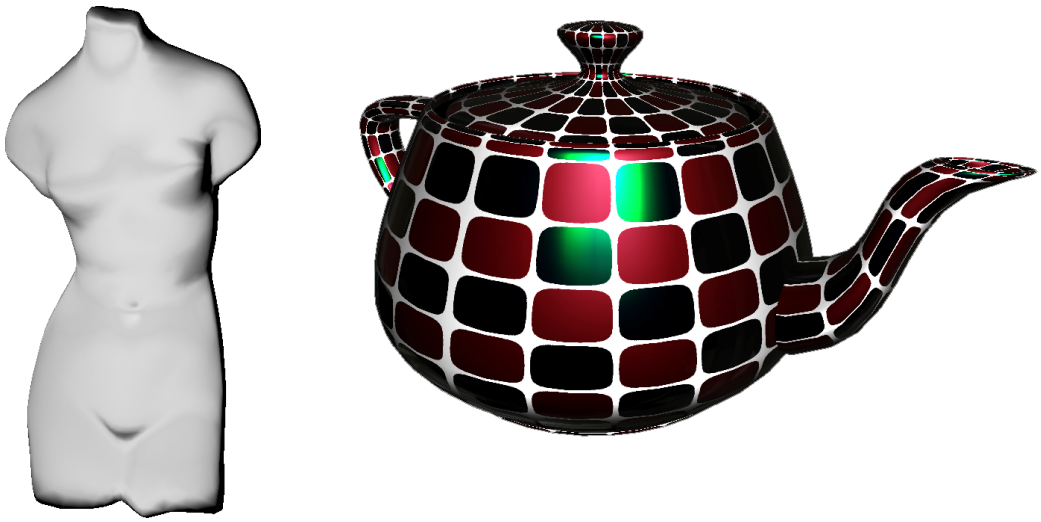

Figure 3: Homomorphically factorized materials. On the left, satin, on the right, a combination of three materials (garnet red, satin, and mystique).

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`.

Now 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+l* outputs. This new program has all the inputs, outputs, and computations of the original programs.

Because of the way our interface 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 program objects can result in redundant computation. However, the "<<" operator, in conjunction with the optimizer in our dynamic compiler

(particularly dead code removal) and the definition of some simple "connector" programs, can be used to eliminate such redundant computations. Both operators actually operate on the internal representation of program objects to build a completely new program objects, which are ultimately run through the full suite of optimizations and virtualizations supported by our platform's dynamic compiler. 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 "connector" program can be attached to the output of the combined program and our dead code eliminator will remove the redundant computation.

To satisfy the type rules for connecting programs, we need to define the interface of each such glue program to match the particular interface types of the given base program. To make this easier, we provide some shortcuts, similar to manipulators in the C++ iostream library, for manipulating the input and output channels of RapidMind 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 connector 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.

The RapidMind Development Platform supports a stream computation model for general purpose computation, based on the extension of these two operators to data. Recall that arrays are containers for a number of elements of the specific type given as its template argument. Stream objects are represented using the **`Stream`** class, and are containers that hold a sequence of arrays of potentially different types.

Streams are specified by combining arrays (or other streams) with the "`&`" operator. Streams only refer to arrays, they do not create copies. An array can still be referenced as a separate object, and can also be referenced by more than one stream at once. For convenience, an **`Array`** of any type and dimensionality 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 a **`Program`** is compiled with the "`stream`" profile, it can be applied to streams. This is the default mode; an empty argument to the **`BEGIN_PROGRAM`** keyword also specifies a stream program.

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 **`Program`** (and can be assigned to a variable of this type, if the user does not want execution to happen immediately). Such pre-bound 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 via 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 set them up rapidly for execution in the inner loop of our program.

The "<<" operator can also be used to apply programs to 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. This is an extremely useful operation in practice.

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`:

```
Program 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 arrays, and let `x`, `y`, and `z` be streams, arrays, or tuples. Then the following binds a program `p` to some inputs and executes it:

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

This syntax also permits program objects 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. The update object now has compiled-in access to the state arrays referenced by the state stream. The inner loop is very simple and fast. In particular, all program object compilation is done during setup, not in the inner loop.

```
// SETUP
particle = BEGIN_PROGRAM("gpu:stream") {
  InOut<Point3f> Ph; // head position
  InOut<Point3f> Pt; // tail position
  InOut<Vector3f> V; // velocity
  In<Vector3f> A;    // acceleration
  In<Value1f> delta; // timestep
  Pt = Ph; // Physical state update
  A = cond(abs(Ph(1)) < 0.05, Vector3f(0.,0.,0.), A);
  V += A * delta;
  V = cond((V|V) < 1.0, Vector3f(0.,0.,0.), V);
  Ph += (V - 0.5 * A) * delta ;
  Value1f mu(0.1), eps(0.3);
  for (int i = 0; i < num_spheres; i++) { // Sphere collisions
    Point3f C = spheres[i].center ;
    Value1f r = spheres[i].radius ;
    Vector3f PhC = Ph - C;
    Vector3f N = normalize(PhC);
    Point3f S = C + N * r;
    Value1f collide = ((PhC|PhC) < r*r) * ((V|N) < 0);
    Ph = cond(collide, Ph - 2.0 * ((Ph - S)|N) * N, Ph );
    Vector3f Vn = (V|N) * N;
    Vector3f Vt = V - Vn;
    V = cond(collide, (1.0 - mu) * Vt - eps * Vn, V);
  }
  Value1f under = Ph (1) < 0.0; // Collide with ground
  Ph = cond(under, Ph * Value3f(1., 0., 1.), Ph);
  Vector3f Vn = V * Value3f(0., 1., 0.);
  Vector3f Vt = V - Vn;
  V = cond(under, (1.0 - mu) * Vt - eps * Vn, V);
  Ph(1) = cond(min(under,(V|V) < 0.1), Point1f(0.0f), Ph(1));
  Vector3f dt = Pt - Ph; // Avoid lines disappearing
  Pt = cond((dt|dt) < 0.02, Pt + Vector3f(0.0, 0.02, 0.0), Pt);
} END;

// define stream specifying current state
Stream state = (pos & pos_tail & vel);
// define update operator, bind to inputs
Program update = particle << state << gravity << delta ;

// IN INNER LOOP
// execute state update (input to update is compiled in)
state = update;
```

Listing 7: Particle system simulation.

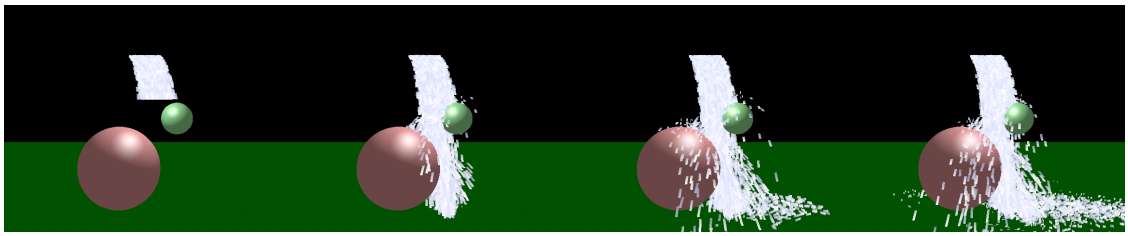

Figure 4: Frames from the particle system animation corresponding to Listing 7.

Engineering

The implementation of the RapidMind platform includes three major level of modularity: the front end(s), the optimizer, and the backend(s). Our main front end supports the C++ API described here, and generates an intermediate representation of programs. Our intermediate representation is similar to an assembly language, but higher-level. When **`BEGIN_PROGRAM`** is called, a new intermediate representation is initialized, and a mode flag is set so that our types know they are declared inside a program definition. When operations are specified inside a program definition, an appropriate instruction for each 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, are cleaned up completely by the optimizer.

When an externally declared parameter or texture is used in a program, a dependency is set up. The visible classes in our interface 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 to the program. When a program object is loaded, the runtime system goes over its list of dependencies and notifies them that an active program object depends on them. Whenever a parameter is modified in immediate mode, it checks for active dependencies, and updates the appropriate values on the target co-processor (the constant registers on the GPU, in this case) as necessary. Updates are done lazily. When coordinating 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 textures. We also shadow textures and streams on both the host and the GPU (or other target processor) and keep track of which is the most recently modified version.

The optimizer is a crucial component of the RapidMind platform. Among other optimizations, 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 program objects that depend only on uniform parameters and arranges to compute them once only, 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 programs into multiple parts and schedule these parts over multiple passes, but this is done completely transparently to the user. 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 for GPUs we support ARB assembly, ARB with

NV assembly extensions, or the OpenGL Shading Language. In this tutorial, we have only used the default GPU and CPU compilation targets, but a more detailed syntax is available for selecting alternatives and passing parameters to the backend in order to control execution and coordinate with graphics APIs and other applications running on the same co-processor when necessary. A CPU backend is also available that supports the dynamic compilation and linking of host code. Finally, as mentioned earlier, a backend that supports the multi-core Cell BE processor is also available.

References

For more information about the RapidMind Development Platform and other products, please visit our website at `http://www.rapidmind.net/`.

An earlier version of the platform interface described here was documented in the book *Metaprogramming GPUs with Sh* [5]. This book contains many additional examples which are still relevant. The RapidMind Development Platform is based on many years of research in the Computer Graphics Lab at the University of Waterloo. The original concept for the interface is described in a research paper presented at SIGGRAPH/Eurographics Graphics Hardware 2002 entitled *Shader Metaprogramming* [4]. The program algebra operations were presented at SIGGRAPH 2004 in a paper entitled *Shader Algebra* [6]. A chapter discussing the derivation of metaprogrammed co-processor APIs, which presents the evolutionary path to our current interface, can be found in the book *Real-Time Shading* [7]. The factored representation of reflectance models used as an example here was first presented at SIGGRAPH 2001 in *Homomorphic Factorization of BRDFs for High-Performance Rendering* [3].

[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.

[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*.

[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.

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

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

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

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

[8] Karl Sims. *Particle Animation and Rendering using Data Parallel Computation*. Computer Graphics (Proc. SIGGRAPH), 24(4):405–413, August 1990.

This document was written by Dr. Michael McCool, co-founder and Chief Scientist of RapidMind Inc. He is also an Associate Professor at the University of Waterloo.