RTSL: a Ray Tracing Shading Language

Steven G. Parker[†] Solomon Boulos[◇] James Bigler[†] Austin Robison[◇]

[†]SCI Institute, University of Utah [◇]School of Computing, University of Utah

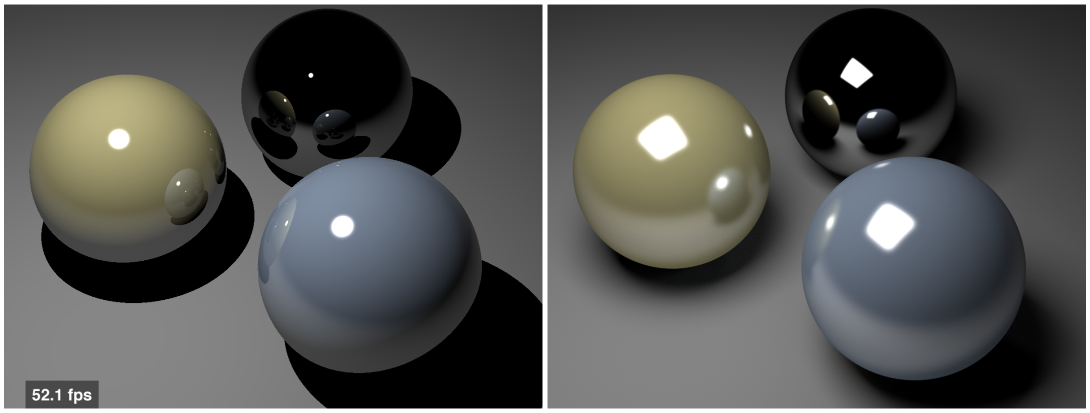

Figure 1: Three spheres with an RTSL Ward material applied to them. All other materials, primitives and lights are also RTSL shaders. The images were renderered with the Manta interactive ray tracer (left) and the batch Monte Carlo renderer Galileo (right).

Abstract

We present a new domain-specific programming language suitable for extending both interactive and non-interactive ray tracing systems. This language, called "ray tracing shading language" (RTSL), builds on the GLSL language that is a part of the OpenGL specification and familiar to GPU programmers. This language allows a programmer to implement new cameras, primitives, textures, lights, and materials that can be used in multiple rendering systems. RTSL presents a single-ray interface that is easy to program for novice programmers. Through an advanced compiler, packet-based SIMD-optimized code can be generated that is performance competitive with hand-optimized code. This language and compiler combination allows sophisticated primitives, materials and textures to realize the performance gains possible by SIMD and ray packets without the low-level programming burden. In addition to the packet-based Manta system, the compiler targets two additional rendering systems to exercise this flexibility: the PBRT system and the batch Monte Carlo renderer Galileo.

1 Introduction

Since its inception [24], ray tracing has lent itself to a modular design. There are a handful of natural modules in almost any renderer, including cameras, geometric primitives, textures, materials, and lights. Each of these modules is easily described either as a set of equations (e.g., primitive intersection) or as a simple procedure (e.g., solid texturing [15]). Despite nearly all renderers having these common notions, there is no common language for describing them as there is for materials in the Renderman Shading Language [8].

Interactive ray tracing has been made possible through improved single-thread performance and expanding multiple core performance. Specifically, the widespread availability of SIMD units such as Intel's SSE and Motorola's Altivec has proved useful for tracing multiple rays in parallel for nearly no additional cost [22]. The majority of these recent interactive systems, however, have only supported triangles and use very simple shading (e.g., a diffuse material with point light shadows) . This is due to the extreme difficulty required in efficiently and correctly writing more complicated primitive intersection routines, light source descriptions, and materials.

We designed RTSL to address these issues based on our own experiences with writing high performance code throughout our renderers. The primary goals of RTSL are:

- **Simplicity** is imperative for widespread adoption. To keep the language clean, it may be necessary to avoid the ability to express complicated data structures such as acceleration structures. Similarly, "implement once" features like image based textures are probably best handled outside of RTSL.
- **Flexibility** is required to ensure that developers do not simply return to C++ for all their needs. In this regard, we would like to allow for function calls, virtual method calls, etc. In addition, developers should be able to implement a majority of rendering components (i.e. cameras, primitives, lights, etc.).
- **Portability** is an important goal for any language. In particular, due to the growing nature of the ray tracing community, we feel it is important to allow the same description to work across renderers or even throughout the course of evolution of a single renderer.
- **Packets** are hard for the majority of users to understand but can provide performance improvements over single ray code. Ideally, a compiler should be able to automatically generate packet code from a single-ray description.
- **SIMD** extensions can provide large performance gains on recent processors, but are notoriously difficult to use. Our goal is to automatically generate high performance SSE code without manual intervention. This becomes more important as the core of a renderer is highly tuned but shaders, textures, etc. are either not tuned or restricted to be very simple.
- **Special cases** can be exploited to improve the performance of a system, and are normally implemented manually or through C++ templates. Given an appropriate description, we hope to generate specialized code for some of these special cases.

2 Previous Work

The origin of flexible shading was first formalized with Shade Trees [6]. Perlin extended this to a more complete language that emphasized procedural textures using solid noise functions [15]. These works were the basis for the Renderman Shading Language (RSL) [8]. A particular difference, however, is that RSL does not support the full generality of shade trees. For example, since a shader in RSL cannot recursively call another shader producing a simple "mix" requires manual copying of source code.

One of the first flexible graphics architectures implemented in the spirit of RSL was the PixelFlow shading language [12]. More recently, graphics hardware has adopted RSL-like languages such as Cg [11], HLSL [13], and GLSL [17]. Each of these is naturally similar to RSL in syntax, but quite different in its object model. For example, Cg, HLSL and GLSL are all designed around recent graphics hardware and the notions of Vertex Shaders and Pixel/Fragment Shaders. None of these approaches fit well with the object oriented nature of ray tracing, but their syntax is fairly natural as it is based on simple vector math.

Because of the high performance of graphics hardware, it has become desirable to harness GPUs for providing interactive previewing of high quality renderings. In order to be useful, the previews need to be similar to the "final render" and yet still be interactive. In this spirit, the LPICS system [14] uses deep framebuffers combined with hardware language descriptions for Renderman shaders and lights. The translation to Cg was done manually. The Lightspeed system provides an improved solution which automatically transforms Renderman descriptions [16].

RSL was used as input to the Vision global illumination system [19, 20], the BMRT ray tracer [7] took RSL descriptions to material descriptions in a ray tracer and researchers in Saarland have used an extended version of RSL to create efficient code for a modern interactive ray tracer, the work closest to our own[1].

The majority of these solutions have focused on material description. This is partially due to implementation concerns such as designing a language for programming GPUs. With software based ray tracers, these restrictions are unnecessary.

3 RTSL: a Ray Tracing Shading Language

GLSL is a high-level shading language that was developed by 3DLabs for programming the GPU. It was created to give developers control over the rendering process in the GPU pipeline without exposing the details of the hardware or requiring assembly language programming. In the same vein as GLSL, the primary goal of RTSL is to make programming an interactive ray tracing system more accessible. Ray tracing enables a rich variety of primitives, materials and textures to be used together. However, recent *interactive* ray tracing work has been largely limited to simple shading models and even simpler primitives, such as triangles. The lesson that can be learned from the GPU community is that a language that allows relative novices to express complex computation will lead to dramatic results.

RTSL is a simple language capable of describing the majority of modules for any ray tracing based renderer. We have made every attempt to design the language to be renderer-neutral by imagining how a compiler could target the many different systems that we have experience. We believe this will allow for wider adoption and utility similar to the way that GLSL allows portability between GPU implementations from different vendors. In addition, as compute power increases interactive ray tracing will be able to provide higher visual quality, so we believe that having an identical description of materials for both interactive and offline systems is a must. We have based our syntax on GLSL, but this was based on a matter of taste and other languages may have been just as well suited.

[1]P. Slusallek, personal communication

3.1 GLSL overview

For the reader interested in learning more about GLSL, we refer you to the OpenGL Orange Book [18] or the language specification [1]. Here we will describe the main features of GLSL and in the following section we highlight the modifications that we have made to more readily support interactive ray tracing.

GLSL is a special-purpose, procedural, imperative programming language that is syntactically similar to C. It uses the same operators as C with the exception pointers (i.e., no address or dereference operators). GLSL does allow fixed-sized arrays and structures, and also inherits C's increment (++) and decrement (−−) operators.

GLSL has three basic types: float, int and bool. The precision and range of the floating point and integer type are not defined by the language. GLSL also defines first-class types that are fixed-length vectors of the above primitive types, called vec2, vec3 and vec4 for the float type, and ivec2,3,4 and bvec2,3,4 for int and bool, respectively. In addition, GLSL defines matrix types for 6 different sizes from 2x2 to 4x4, including non-square matrices. GLSL also defines first-class operators between these matrix and vector types.

GLSL has a very powerful mechanism for selecting components in the built-in vector primitives. Each component can be accessed with a single letter, usually x,y,z or w[2], and an also be indexed like a C array. Letters can be combined to extract multiple components in the same expression, such as "pos.xy", which will produce the first and second components in a vec2, vec3 or vec4. This can be used as either an lvalue or rvalue in programming language parlance.

3.2 Differences between GLSL and RTSL

GLSL provides a syntax that is very convenient to concisely express low-level computation. However, it was designed for use on a GPU: it contains features that an interactive ray tracing system does not need, and is missing features that are needed for such a system. We attempted to preserve as much of the core language as possible both to provide a transition path for GPU shader programs, and also to capitalize on familiarity to GPU programmers by keeping the language consistent with GLSL wherever possible.

Object model: GLSL provides the ability to write two different functions, called a fragment program and a vertex program. They are distinguished only by file extension on the filesystem. With RTSL, we needed a more flexible mechanism since we wanted to allow programs for multiple extension points, including cameras, primitives, textures, lights, and materials. Furthermore, we wanted to allow the creation of a single object that could fill more than one role – such as a primitive that also acted as a material.

Consequently, we added a simple object model to the language. This object model provides single inheritance, and multiple interface inheritance (similar to Java). Built-in interfaces for the above extension points act as a base for all of the classes in the system. Syntactically, an object is specified by adding:

```
class classname : base_interface1;
```

to the beginning of the file. Subsequently defined functions are inserted into the class as methods until the end of the file or a subsequent class declaration. A constructor is implemented by creating a function called "constructor". Namespaces, nested classes and multiple object inheritance are not supported. A backend compiler may choose to facilitate namespaces through a different mechanism. Additionally, all interface functions return void and take no arguments for flexibility and extensibility.

In addition to the class model, GLSL adds support for a reference type. Again, similar to Java, a reference is analogous to a pointer without pointer arithmetic. References to other RTSL objects provide a powerful mechanism for combining classes without the cut/paste reuse that is common in many GPU-based programs

[2]r,g,b,a and s,t,p,q are also allowed – see the GLSL specification for details

and even other shading languages. For texturing, these references are used to implement full shade trees [6], which is not possible in RSL or GLSL. This feature will require the target machine and backend renderer to support function calls or to utilize a more sophisticated linking phase that combines instructions from multiple RTSL classes.

Texture sampling: GLSL provides a number of first class objects that represent textures on the GPU. Through a host-based programming language (usually C or C++), texture images are bound to these objects. A series of 26 builtin functions for texturing are used to lookup values in these textures.

Instead, RTSL uses the aforementioned class and reference capability to access texture values. The code querying a texture does not need to know whether the texture is procedural or image-based. We anticipate that RTSL would *not* be used for implementing image-base texture lookup and filtering, since the renderer may want to utilize sophisticated storage mechanisms such as tiling or caching. Therefore, the target renderer should provide RTSL compatible objects that provide image texture functionality. In this respect, it is similar to the operation of a GPU: the programmer has only basic control over the precise storage and lookup of textures that are owned by the system. RTSL would need dynamically sized arrays to be able to accomplish more of this task, which was outside the scope of our initial effort.

Lighting constructs: GLSL contains support for OpenGL-style lighting. A fixed number of lights exist, each of which can be enabled or disabled. The GLSL language exposes this functionality through a set of global variables that reflect the parameters for each light source. Shader programs must explicitly compute lighting, and many of them assume a limited number of lights to avoid checking that each light is enabled.

To provide more flexibility in a ray tracing system where lighting and shadows interplay, we have adopted the illuminance loop construct from RSL. The *illuminance* keyword operates like a for loop, iterating over each of the light sources and executes the loop body for those that are not occluded. A Kajiya-style path tracer may elect to choose one light source at random rather than iterating through all light sources. This is a portion of the Phong material illustrating this concept:

```
illuminance(rt_HitPoint, rt_GeometricNormal,
       rt_ShadingNormal, PI/2) {
     vec3 Ln = rt_LightDirection;
     result += DiffuseColor * rt_LightColor * max(0., dot(
      Ln, rt_ShadingNormal));

     vec3 H_lum = normalize(Ln + V);
     float NDotH_lum = max(0., dot(H_lum, rt_ShadingNormal
      ));
     float phong_term_lum = pow(NDotH_lum,
      SpecularExponent);

     result += phong_term_lum * SpecularColor *
      rt_LightColor;
   }
```

Listing 1: An example *illuminance* statement.

Similarly, the *ambient* keyword acts as an if statement that executes the body of the code only if ambient lighting is necessary given the hit position and normal. The target renderer may elect to eliminate the body of this statement entirely, as in a path-tracing based renderer. An example taken from a Lambertian shader is:

```
ambient(rt_HitPoint, rt_ShadingNormal)
     result += DiffuseColor*rt_LightColor;
```

Listing 2: An example *ambient* statement.

First class color type: RTSL adds a first-class type called "color" that abstractly represents color in the underlying renderer without requiring a 3 channel color space. Most computations that use colors can use multiplication and addition operators without specifically requesting the rgb components. This is valuable for interfacing with renderers that have a more sophisticated color representation – for example, the Monte Carlo renderer Galileo has both an rgb and spectral mode. Additionally, this could be used to support a renderer that uses a fixed-point color rather than a floating point color. However, the color type will always have floating point semantics. Colors are not required to be in the range $[0, 1]$.

Built-in variables: For GLSL, the primary mechanism for communicating data between the shader and the rest of the GPU is a set of globally scoped variables that are either read and/or written by the vertex program. For example, gl_FragColor represents the color that will be assigned to a pixel after the result of a fragment program, so the fragment program must assign a value to gl_FragColor before returning. In computing this color, the fragment program can read a number of variables that describe both the fragment and the general state in the system, such as the coordinates of the fragment and various transformation matrices.

RTSL adopts a similar convention, but the set of variables is completely different than GLSL. Furthermore, since RTSL fills multiple roles, the set of variables available to each function can vary. For example, a shading method will read the `HitPosition` `ShadingNormal` variables to will produce a color for the sample. A primitive intersection method will read the `RayOrigin` and `RayDirection` and will set the new hit distance (usually through the hit function described below). Note that exposing this state as a variable in RTSL does not imply that the state is stored in a variable in the target renderer. For example, Galileo always stores the inverse ray direction (`InverseRayDirection`) while PBRT only stores the ray direction requiring computation of the reciprocal when accessed. Manta defers the computation of the inverse direction until it is requested, then stores it for subsequent use.

We consider this state to be one of the primary advantages of adopting GLSL as the basis for this shading language. It allows the set of state to be extended without breaking existing code, and avoids requiring repetitive typing of function prototypes. Similar to GLSL, RTSL prefixes the built-in variables with rt_ to avoid naming conflicts with user code. A more detailed description of the built-in state designed for RTSL will be presented in Section 3.4.

Variable qualifiers: GLSL follows the Renderman convention for annotating variables with keywords to allow data to be passed from one rendering stage to another. In addition to locally scoped variables declared within a function, GLSL allows variables to be declared at the global scope with the following modifiers:

- **attribute:** variables passed from the application to the shader that may change per vertex.
- **uniform:** variables passed from the application to the shader that may change per primitive.
- **varying:** variables passed from the vertex shader to the fragment shader, and may change per pixel fragment.
- **const:** a compile-time constant.

In a non-rasterization system, most of these modifiers do not have significance. Therefore, RTSL allows the following modifiers for variables at class scope:

- **const:** a compile-time constant.
- **public, private:** variables specific to an instance of this class. The renderer can decide if and how to distinguish public from private. The constructor, or functions called from the constructor, are only allowed to write to these data members. A per-ray method (such as intersect or shade) is not allowed to modify class state.

- **scratch:** variables that are used to pass state within a particular object instance between multiple methods. This is conceptually stored per ray, but the backend may utilize stack-based variables or a special section of the raypacket to implement it.

Additionally, a vector variable can be annotated with the **unit** keyword to indicate to the compiler that the vector will always have a length of 1.

New built-in functions and constants: Most of the functions defined by the GLSL specification are also valuable for ray tracing. GLSL even defines builtin functions for noise textures, reflection and refraction. In addition to those defined by the GLSL standard, RTSL adds the following builtin constant and functions:

- **INT_MIN, INT_MAX, FLT_MIN, and FLT_MAX** are similar to the C standard library definitions for determining the range of builtin types.
- **PI** is defined to support appropriate scaling for BRDF integration and other geometric operations. Although GLSL provides builtin functions for computing degrees to/from radians, this is not sufficient for a ray tracing based system.
- **matrixType inverse(matrixType m)** will invert any square or affine transformation (i.e. 2x3 or 3x4) matrix. The results are undefined if the matrix is singular.
- **bvec3 inside(vec3 x, vec3 low, vec3 high)** is a convenience function that returns true if the first argument is bracketed by the second and third arguments. This function is overloaded for float, vec2, vec3 and vec4, and the range can either be a single scalar (used for all components) or a vector value of the same rank as the first argument. When used with vector arguments, the return value is a boolean vector type. This allows simple constructs such as:
  ```
  if (all(inside(u, 0.0, 1.0))){
   // All components of u are in the range [0,1]
  }
  ```
  This style of code is not only quite readable but also is generally more efficient for a SIMD implementation than a series of cascading if statements.
- **vec3 perpendicularTo(vec3)** returns a vector that is perpendicular to the given vector. This function is overloaded for any vec type. The vector chosen and the algorithm for selecting that vector are implementation dependent.
- **int dominantAxis(vec3)** returns the index of the axis that contains the largest component (absolute value). This function is overloaded for vec2, vec3 and vec4.
- **color trace(vec3 O, vec3 D)** spawns a secondary ray with the given origin and direction. This returns the color. For simplicity in the common case, an overloaded version is available that only takes a direction and assumes the origin as the current hitpoint.
- **bool hit(float thit)** is used for primitive intersection to report a ray-object intersection. The thit value is not required to be larger than 0, as the renderer must automatically prune values outside of the useful range. Hit returns true if the intersection is now the new closest intersection for the ray.
- **float luminance(color c)** computes the luminance value from the underlying color type. This is useful for ray-tree pruning.
- **float rand()** generates a uniform random variable in the range $[0,1)$.
- **pow** was extended to support additional overloads where the exponent is an integer value, to allow optimizations based on a successive power generation algorithm.
- **min and max** were extended to support a "horizontal" max, which returns the largest (or smallest) component of a single vector argument.

3.3 Standard class hierarchies

Using RTSL, we define a handful of common interfaces found in most renderers: **Cameras**, **Primitives**, **Textures**, **Materials**, and **Lights**. While some renderers may have more distinct interfaces than these and some have less, we feel these components are universal enough to be easily translated by a backend. Implementing a primitive involves creating a class derived from the built-in **Primitive** interface, and defining methods for intersection, normal computation and bounding box computation. A simple sphere primitive class in RTSL is shown in Listing 3. Each of these base classes will be discussed.

```
class Sphere : rt_Primitive;

public vec3 center;
public float radius;

void constructor(vec3 newcenter,
                 float newradius) {
  center = newcenter;
  radius = newradius;
}

void intersect() {
  vec3 O = rt_RayOrigin - center;
  vec3 D = rt_RayDirection;
  float A = dot(D, D);
  float B = dot(O, D);
  float C = dot(O, O) - radius*radius;
  float disc = B*B-A*C;
  if(disc > 0.0){
    float r = sqrt(disc);
    float t0 = -(r+B)/A;
    if(t0 > rt_Epsilon){
      hit(t0);
    } else {
      float t1 = (r-B)/A;
      hit(t1);
    }
  }
}

void computeNormal() {
  rt_GeometricNormal = (rt_HitPoint - center)/radius;
}

void computeBounds() {
  rt_BoundMin = center - vec3(radius);
  rt_BoundMax = center + vec3(radius);
}
```

Listing 3: A Sphere in RTSL

3.3.1 Cameras

A **Camera** class must provide only one function: *generateRay*. In this function, the user may access the 2D `ScreenCoord` variable and must write to the `RayOrigin` and `RayDirection` variables. To allow for depth of field, **Cameras** additionally have access to the 2D `LensCoord` but are not required to use it. Listing 4 shows a simple camera example.

```
class PinholeCamera : rt_Camera;

public vec3 position;
public vec3 lower_left_corner;
public vec3 uvw_u;
public vec3 uvw_v;
public vec3 uvw_w;

void constructor(vec3 eye, vec3 up, vec3 gaze,
                 vec2 uv0, vec2 uv1, float d)
```

```
{
  position = eye;
  uvw_v = up;
  uvw_w = gaze;
  uvw_u = normalize( cross( uvw_v, uvw_w ) );
  uvw_v = normalize( cross( uvw_w, uvw_u ) );
  uvw_w = normalize( gaze );
  lower_left_corner = uv0.x * uvw_u + uv0.y * uvw_v + d *
      uvw_w;
  uvw_u = uvw_u * (uv1.x - uv0.x);
  uvw_v = uvw_v * (uv1.y - uv0.y);
}

void generateRay()
{
  rt_RayOrigin = position;
  rt_RayDirection = lower_left_corner + rt_ScreenCoord.x*
     uvw_u + rt_ScreenCoord.y*uvw_v;
}
```

Listing 4: A Pinhole Camera in RTSL

3.3.2 Primitives

Primitives in RTSL are more complicated than the majority of other classes. Most renderers support a variety of methods for primitive intersection, and many choose to defer the computation of values including texture coordinates, normals, etc. To address this issue, **Primitives** provide the following functions: *intersect*, *computeNormal*, *computeBounds*, *computeTextureCoordinates*, and *computeDerivatives*.

The *intersect* function uses various properties of the ray including `RayOrigin`, `RayDirection`, `HitDistance`, `Epsilon` to determine whether a ray intersects the primitive. A special *hit* function is provided that reports a potential intersection and then returns true if the primitive is now the closest object. This can be used to avoid unnecessary computation in the event of a miss.

The *computeNormal* function provides access to `HitPoint` and any other scratch variables that were declared in the class. This function is provided to facilitate systems that used deferred normal computation.

The world bounds of the object are computed by *computeBounds*, which is typically used for acceleration structure creation.

Many primitives have a natural parametrization that can be directly used for texturing. RTSL exposes this in *computeTextureCoordinates* that has access to all the hit information and scratch variables.

The pair of functions, *generateSample* and *samplePDF* are used to generate points on the surface of the **Primitive** according to the specified probability density function, writing to the values `HitPoint` and `PDF`, respectively. These methods are optional, but need to be implemented if a primitive is to be used as an area light in some renderers.

Finally, *computeDerivatives* is provided to allow renderers to support knowledge of surface derivatives *dPdU* and *dPdV* and other derivatives. For example, the PBRT backend will fill in a differential geometry structure. For further discussion on the purpose and semantics of these derivatives, see Section 3.4.

3.3.3 Materials

Materials in RTSL can provide both arbitrary shading through the *shade* function or physically-based rendering through *BSDF*, *sampleBSDF*, and *evaluatePDF*. Some backends may use only one or the other, and other backends may use both.

In *shade*, the user must simply compute and set `SampleColor`. To do so, they have access to shading information (e.g., `HitPoint`, `GeometricNormal`, `ShadingNormal`) as well as incident lighting through the *illuminance* and *ambient* statements.

The physically-based rendering functions provide users methods to define physical scattering BSDFs. Within *BSDF* a user needs to compute the total amount of radiance transferred between a pair of directions: `RayDirection` and `LightDirection` and store that in `BSDFValue`. As with *shade* the user has access to all relevant shading information.

An example material is shown in Listing 5. The section comments will be explained in more detail in Section 4.3.

```
class SchlickDielectric : rt_Material;

public float eta;
public float f0;
public color absorption;

void constructor(float inside, float outside, color
     absorb) {
  eta = inside/outside;
  f0 = pow((outside-inside)/(outside+inside), 2);
  absorption = absorb;
}

void shade() {
  // Section 0
  vec3 I = normalize(rt_RayDirection);
  vec3 P = rt_HitPoint;
  vec3 NN = rt_ShadingNormal;

  float eta_temp = eta;
  color atten = color(1.);
  if (dot(rt_RayDirection, rt_ShadingNormal) > 0.) {
    eta_temp = 1./eta;
    NN = -NN;
    atten = pow(absorption, rt_HitDistance);
  }

  float negNdotV = dot(I, NN);
  float k = 1.0 - eta_temp * eta_temp * (1. - negNdotV *
     negNdotV);
  vec3 R = reflect(I, NN);
  vec3 T = R;
  if (k >= 0.) {
    T = eta_temp * I - (eta_temp * negNdotV + sqrt(k)) *
     NN;
  }

  float Fr = f0 + (1.-f0)*pow(1.+negNdotV, 5);
  float Ft = 1.-Fr;
  // Section 1
  color refl_result = trace(P, R);
  // Section 2
  color refr_result = trace(P, T);

  // Section 3
  rt_SampleColor = atten * (Fr * refl_result + Ft *
     refr_result);
}
```

Listing 5: A Dielectric Material in RTSL

To enable importance sampling, a class can implement the sampling function pair *sampleBSDF* and *evaluatePDF*. The *evaluatePDF* function must be able to evaluate the probability density function of the sampling method, while *sampleBSDF* should generate a sample according to the same density function. The 2D random seed `BSDFSeed` is available to choose sample directions, however, more are always available through the *rand* function. The generated direction should be stored to `LightDirection`, and similarly *evaluatePDF* should store its result in `PDF`.

Materials may also optionally provide an *emission* function to describe the emissive component of a **Material**. This allows easy support for area lights and does not require the renderer to store two

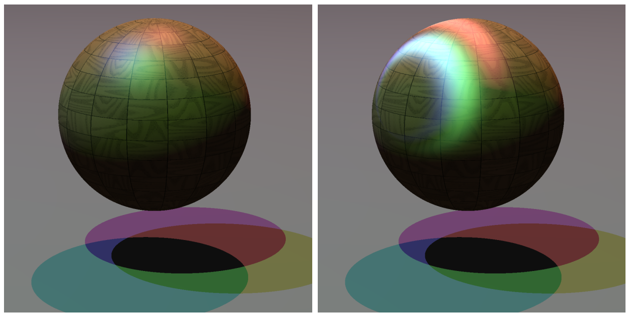

Figure 2: Two spheres using the same RTSL texture with two separate RTSL materials, a Phong shader (left) and a Ward shader (right).

pointers for both a material and a light source shader. However, renderers that require this distinction (as Renderman does) may require the backend to split the RTSL **Material** class into separate classes in the target language.

3.3.4 Textures

While **Materials** can produce acceptable images using constant color parameters, varying surface detail is better described using **Textures**. **Textures** must provide a *lookup* function which stores its result in `TextureColor` or `FloatTextureValue` for scalar textures. To determine the texture value at a particular shading point, the *lookup* function has access to all the shading info a **Material** might have. **Textures** are assumed to contain colors, though in the future RTSL may be extended to handle textures containing other types, such as float or vec2.

The separation of **Material** and **Texture** is quite useful and one of the key missing features in both the Renderman Shading Language and GLSL. For example, a procedural wood texture can be used as the diffuse term in both a dusty material or a varnished material. Figure 2 shows an example separation.

3.3.5 Lights

In RTSL, the **Light** class is specifically for singular lights. By this we mean lights that would not be intersected by a reflected ray in a path tracer. This includes point lights, directional lights, spot lights, and so forth. The **Light** class provides an *illumination* function that determines the direction and distance to the light from the point being shaded as well as the amount of light that would reach that point if it were unoccluded. Occlusion testing and/or shadow ray attenuation is left to the renderer.

3.4 State Variables

These classes operate by manipulating a suite of built-in variables that reflect the state of the renderer. As discussed previously, these variables are globally scoped (lexically) in RTSL but are not truly global. They merely replace the input and output arguments of the function in a convenient way.

Several of these variables were mentioned in Section 3.3, but it is worth considering more of the global state here. For space considerations, we do not discuss all of it in detail.

An overview of these variables are in Table 1. Each method (and its callees) may refer only to parts of the global state that are defined, and may only write to an even smaller subset.

Epsilon is a small constant used to avoid numerical noise. It should be larger for floating point representations with lower precision. It is set by the target renderer and is available for all methods and classes.

TimeSeed is used for a Monte-Carlo sample to support animation and motion blur. It is available in all of the ray-based methods of all classes. Depending on the underlying implementation, it may vary per ray, may be constant over a packet or over an entire frame. The RTSL code is insulated from these differences.

RayOrigin and RayDirection define the parameters of the ray currently being processed. The *Camera.generateRay* method must write to both of these variables. Any of the **Primitive** and **Material** methods (except *Primitive.computeBounds*) can read it. A `RayDirection` may or may not be unit length.

HitDistance is the t parameter of the current closest intersection. It is optionally set by the *Camera.generateRay* method, and can be read by and of the *Primitive* methods.

ScreenCoord is the coordinates of the sample center in screen space, in the range $[0,1]$. It is used only in the *Camera.generateRay* method, but some renderers may choose to expose it to other methods through a non-standard interface.

LensCoord is the coordinates of the sample on the lens. It is used only in the *Camera.generateRay* method.

GeometricNormal and ShadingNormal reflect the normal computation at various points of the computation. The geometric normal is computed by the *Primitive.computeNormal* function, and is available for the *Primitive.computeDerivatives* and *Light.illumination* methods and the **Material** methods. Alternatively, a primitive may set the `GeometricNormal` in the intersect method if it is the new closest hit, in which case the *computeNormal* function is not needed. Setting the geometric normal also sets the shading normal, and instancing objects will only modify the shading normal. Bump mapping would be implemented as a Material that modified the geometric normal and then called a child material object.

TextureUV and TextureUVW are the 2D and 3D texture coordinates. They are created by the primitive and are used by texture classes. They may be lazily evaluated, i.e. an implementation may elide calls to *Primitive.computeTextureCoordinates* if the texture coordinates are not used. If the primitive does not implement *computeTextureCoordinates*, the texture coordinates default to world space coordinates.

Derivative information is tracked in a fashion similar to RSL but we chose to expose only the most commonly used values. Surface parametrization derivatives, `dPdu` and `dPdv`, are computed in **Primitive**'s *computeDerivatives* method and read by other shaders. Additionally, *computeTextureCoordinates* can optionally write to the quartet of derivative variables `dsdu`, `dsdv`, `dtdu` and `dtdv` that represent the texture coordinate derivatives with respect to the surface parametrization.

These derivatives can be used, for example, to implement bump mapping and to band-limit procedural textures with the aid of two additional variables, `du` and `dv`. These variables represent the change in surface parametrization at a given point on the shaded object, analogous to the RSL variables of the same name. It is completely up to the backend renderer to determine how the values of these variables are computed. If a particular renderer employs a finite-differencing scheme for computing derivatives, `du` and `dv` would be the delta values used in the differencing, while a renderer that analytically computes derivatives may simply set these variables to zero.

4 RTSL Compiler

We have implemented a compiler for the language discussed above. It utilizes a yacc/lex-based parser with a symbol table, and generates an abstract syntax tree (AST) in memory. The remainder of the compiler is implemented as a series of passes over this AST to perform analysis and create transformations of the tree. A static check phase ensures that the input is valid through type analysis.

Camera	Primitive	Texture	Material	Light
vec3 RayOrigin	vec3 RayOrigin	vec2 TextureUV	vec3 RayOrigin	vec3 HitPoint
vec3 RayDirection	vec3 RayDirection	vec3 TextureUVW	vec3 RayDirection	vec3 GeometricNormal
vec3 InverseRayDirection	vec3 InverseRayDirection	color TextureColor	vec3 InverseRayDirection	vec3 ShadingNormal
float Epsilon	float Epsilon	float FloatTextureValue	vec3 HitPoint	vec3 LightDirection
float HitDistance	float HitDistance	float du	vec3 dPdu	float TimeSeed
vec2 ScreenCoord	vec3 BoundMin	float dv	vec3 dPdv	
vec2 LensCoord	vec3 BoundMax	float dsdu	vec3 LightDirection	
float du	vec3 GeometricNormal	float dtdu	float LightDistance	
float dv	vec3 dPdu	float dsdv	color LightColor	
float TimeSeed	vec3 dPdv	float dtdv	color EmissionColor	
	vec3 ShadingNormal	vec3 dPdu	vec2 BSDFSeed	
	vec2 TextureUV	vec3 dPdv	float TimeSeed	
	vec3 TextureUVW	float TimeSeed	float PDF	
	vec2 dsdu		color SampleColor	
	vec2 dsdv		color BSDFValue	
	float PDF		float du	
	float TimeSeed		float dv	
void constructor()	void constructor()	void constructor()	void constructor()	void constructor()
void generateRay()	void intersect()	void lookup()	void shade()	void illumination()
	void computeBounds()		void BSDF()	
	void computeNormal()		void sampleBSDF()	
	void computeTextureCoordinates()		void evaluatePDF()	
	void computeDerivatives()		void emission()	
	void generateSample()			
	void samplePDF()			

Table 1: RTSL state variables and interface methods. In code, all state variables are prefixed with rt_

To demonstrate the flexibility of RTSL, we have written three backends for our compiler corresponding to three very different renderers: Galileo, PBRT, and Manta. Each of these three backends emit C++ code that is then compiled and linked into the individual renderers. Galileo and PBRT are Monte Carlo path tracers, whereas Manta is an interactive packet based ray tracer that takes advantage of SSE optimizations when possible. We begin with the description of our Galileo backend to help explain the basic compiler issues. We then explain the work necessary to create a similar backend for PBRT. Finally, we describe what was required to automatically generate packet and SSE code for Manta.

4.1 Galileo backend

For Galileo, the backend was simply designed to output working code. There is no requirement for optimization beyond that provided by the standard C++ compiler. In many cases, the RTSL classes mapped easily to Galileo (e.g. Cameras, Textures, and Lights). The majority of the backend code simply handles the general output of typed variables, operators, function calls, etc. In a few cases, however, more work was required to map RTSL to Galileo and we discuss these cases here.

4.1.1 Primitive

In Galileo, primitive intersection returns a boolean result indicating whether or not a new hit point has been found. This is a fairly common design for renderers, but RTSL is designed to be more implementation neutral and so each standard interface method does not take any arguments nor return any results. Galileo also does not defer any computation during intersection as RTSL allows. To resolve this difference, we simply have to inline the various deferred functions wherever a new hit point is computed. Scratch variables are simply local variables in C++.

4.1.2 Material

Naturally, most interesting extensions of nearly any renderer will be to its material library. Galileo supports both standard arbitrary shading and physically based rendering, providing two different methods: *kernel* and *radiance*. The *kernel* function represents the kernel of the rendering equation [10], which is expressed as a Fredholm Integral Equation [23]. When a non-singular kernel is not available (e.g. for perfect dielectrics), the user may override the more general *radiance* function that simply returns a color.

Transforming an RTSL *BSDF* into a kernel is fairly straightforward: a simple cosine correction term is multipled by the BSDF result. The "anything goes" shading in RTSL's *shade* function naturally maps to Galileo's *radiance*. Supporting the *illuminance*, *ambient*, *trace*, and *emission* statements required more care.

The *ambient* statement in RTSL provides for an "if-then-else" structure precisely for global illumination renderers that lack a notion of "ambient light". In this case, Galileo always takes the "else" path as intended, thereby ignoring ambient lighting. If Galileo added final gather methods, an option might be provided to use one of those methods as the `LightColor` value for the more traditional ambient evaluation.

The *illuminance* statement is intended to loop over incident singular lighting and apply the body of the statement for each light. As Galileo is usually run as a more traditional path tracer, we chose not to loop over all lights but to take some number of directional samples and evaluate those instead. The number of samples is currently a parameter for our compiler, but could simply be an optional parameter to *illuminance* as in RSL.

Galileo only stores a single Ray in its rendering context, so for a *trace* call the necessary state must be stored off into temporaries, the color result computed, and the original state returned. This is straightforward, but we include this description for completeness.

RTSL represents area light emission by using an *emission* function for **Materials**. This departs from the RSL tradition of separate "LightShaders" and "SurfaceShaders" that Galileo has followed. As we realized, however, other than scene graph issues there is no reason an object cannot simply emit light as long as it implements the *radiance* function. The only case where this would be a problem is an area light source with both a *kernel* and an emission term. We do not believe this would happen in practice and may be better handled outside of RTSL if it is rare.

4.2 PBRT backend

The single-ray architecture of PBRT enables its compiler backend to share code with the Galileo backend. Any RTSL class not mentioned here is assumed to behave in the same manner as the Galileo implementation. Minor modifications to the backend were necessary to change generated type names and class interfaces; however, due to architectural differences between the renderers, additional code is inserted into the generated methods to adapt RTSL models to PBRT's codebase. PBRT is also missing certain functionality expected by RTSL, such as a vector division operator or linear interpolation of colors that is provided via injected code in each generated object.

4.2.1 Primitive

Like Galileo, PBRT does not defer certain computation and requires the inlining of RTSL methods. For example, PBRT uses differential geometry information to imply the surface normal at intersection time. **Primitives** are therefore required to implement the *computeDerivatives* function such that the surface normal can be calculated from the surface parameterization derivatives, ignoring the *computeNormal* function entirely. PBRT's system of ray transforms (essentially making every **Primitive** an instance) also requires that intersection code is wrapped by a correct transformation of the incident ray.

4.2.2 Material

Due to the strict physically based nature of PBRT, several restrictions are placed on RTSL **Materials** compiled for this renderer. PBRT describes materials as collections of BSDFs, therefore only supporting *BSDF* based RTSL **Materials**; *shade* and associated functions such as *trace* and *illuminance* are unsupported. *BSDF*, *sampleBSDF* and *evaluatePDF*, on the other hand, map directly to PBRT's class functions.

4.3 Manta Backend

The Manta Interactive Ray Tracer allows us to take advantage of both packets and SIMD extensions. Automatically generating efficient packet and SIMD code from an RTSL description presents both challenges and opportunities.

4.3.1 Automatic Packetization

For primitive intersections and some other methods, the Manta backend can simply loop over all of the rays in the packet and perform the single-ray RTSL computation for each ray. However, *Material* methods require additional analysis. This is due to what we refer to as *packet synchronization points*. A packet synchronization point is any statement that operates on an entire packet of rays in the Manta architecture, such as *trace*, *ambient* and *illuminance*. For example, with the *illuminance* statement, an entire set of shadow rays are created for all of the rays in the packet. To accomplish this, all of the computation preceding the *illuminance* statement must be completed for all of the rays in the packet before the shadow rays are created and traced.

To achieve this, we implemented a "section analyzer" pass that separates RTSL statements into an ordered set of sections. These sections are a group of code that works over each ray individually (in a loop) or over all rays (such as casting shadow rays). These sections are each analyzed and generated independently. Listing 5 shows the packet synchronization points that are discovered by the Manta back end. Section 0 operates over each ray in the packet and completes before section 1 and 2 are executed with a single call for each trace function. Section 3 is again a loop over all rays. Variables that cross section boundaries (such as atten, Fr and Ft) are saved in packet-sized arrays on the stack.

To implement scratch variables, the Manta backend allocates storage in the ray packet for those variables to be stored. Manta allows these variables to be overwritten if another primitive eventually reports as the closest intersection. This is consistent with the RTSL semantics.

4.3.2 Packet Optimizations

Beyond the simple benefits for amortizing memory references and virtual function calls, packets also offer algorithmic benefits [4]. In the case of primitive intersection, there are a number of special cases that allow the number of operations required for an entire packet to be reduced from the more general case. The example in Listing 3 uses up to 32 floating pointing operations, including three "expensive" operations (divides/square roots). Knowing that the rays all share a common origin will reduce the total number of flops to 23. Knowing that the rays are all unit-length reduces the flops to 24 but also reduces the expensive operations to only one. Combining these two operations uses only 15 operations with a single square root. To accomplish these operations, the RTSL compiler employs a number of optimization passes.

For a packet with a common origin (e.g. as for primary rays coming from a pinhole camera), the vector from the sphere center to the ray origin is invariant for the packet. For large packets, computing a "packet invariant" term once can result in considerable performance improvements. To achieve this, the compiler implements loop hoisting to factor out operations that are common to all rays in a packet. It also factors out other operations that are common over the ray packet, such as the computation of $radius * radius$ in 3.

To exploit unit directions, the compiler tags variables with a unit-length attribute. This starts with the `RayDirection` variable and is propogated through the dependency graph. Dot products of a vector with itself are reduced to 1, and calls to the normalize function are eliminated. This is followed by a constant propagation pass to eliminate identity operations. This optimization could also be performed for target systems that always assume that ray directions are normalized. The programmer can also annotate variables with the unit attribute to provide additional hints to the compiler.

The optimizer accurately locates these optimizations in the primitives that we have implemented, making it performance-competitive with hand-generated code.

4.3.3 SIMD

Many modern microprocessors offer SIMD instruction sets to operate on multiple (usually four) operands simultaneously under certain constraints. Manta ensures that data is stored and aligned in a manner that is friendly to these architectures. However, Manta employs a splitting algorithm for accomodating divergent ray packets, so functions must operate on a range of rays. These rays may begin and end on boundaries that are not a multiple of four. Consequently, similar to standard loop unrolling, it is necessary to handle these cases outside of the main loop body that operates on four rays at a time. Doing this by hand is tedious and very error prone. Our RTSL compiler handles this quite naturally and allows us to easily tweak how we generate these cases. We have found that using SSE with a masking operation is more efficient than the short scalar loops that we had originally implemented.

Translating complicated control flow is perhaps one of the largest strengths of RTSL. In traditional SIMD processing, a mask is generated whenever a conditional statement is evaluated. If the mask indicates that no further processing is required, a particular code section may be skipped. Otherwise, that section must be entered but all state changes must obey the current mask. Religiously tracking these masks is also quite error prone and is probably a reason for the lack of "advanced code" in current interactive ray tracers. We avoid the use of a boolean complement instruction by reversing the sense of masks in else cases. In the first if statement, active rays have a mask of all 1's, and for the else statement active rays have a

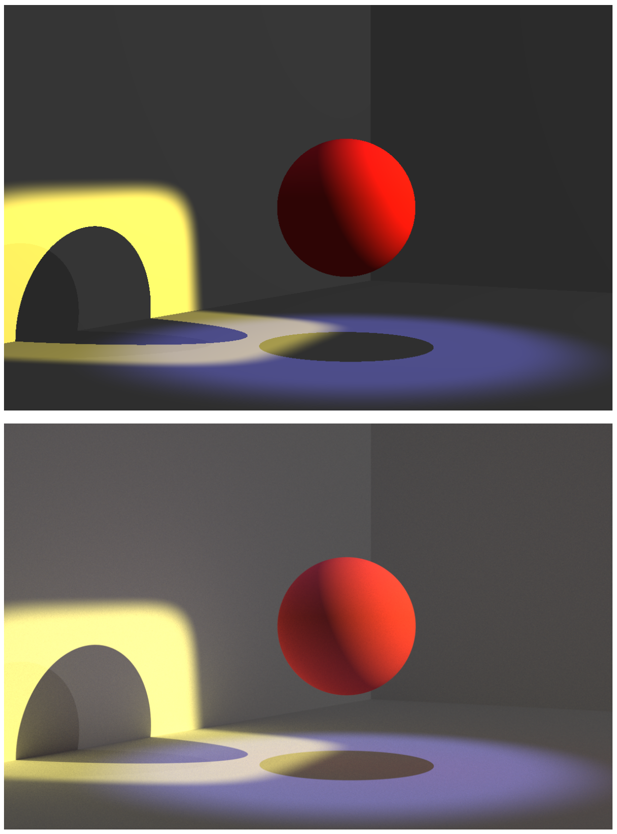

Figure 4: RTSL translation of Pixar's uberlight [3] rendered by Manta (top) and Galileo (bottom).

mask of all 0's. Using DeMorgan's law, we can eliminate the need to complement the masks.

A final complication for utilizing SIMD is that some features are simply not available in SIMD instruction sets. For example, SSE does not provide many standard math library functions (e.g. pow, exp, log, sin, cos) that are so commonly used in rendering software. For some of these functions, we have chosen to implement our own versions in SSE; however, for others (such as Perlin noise) we have simply chosen an "unpack-scalar-repack" solution. Any section of code containing such a statement will obviously not reap benefits of SIMD, but the remaining code may still produce a benefit. In the future, we hope to employ more native SIMD functions wherever possible.

5 Results

We translated a number of RSL shaders from renderman.org [2] in addition to existing C++ shaders from Manta and Galileo to RTSL. For example, Figure 4 shows a translation of Pixar's uberlight [3] running in both Manta and Galileo. PBRT generates identical results to Galileo using its path tracer plugin; we will therefore only refer to Galileo results for single-ray comparisons. Additionally, Figure 5 shows two procedural textures translated from RSL shaders.

One of the goals of RTSL was to allow users to extend renderers without explicit knowledge of packets or SSE. In Table 2, we demonstrate our results for automatic scalar and SSE packet code for a variety of primitives that can be seen in Figure 6. Numbers are seconds per frame for shading using a single core of a 3.0GHz MacPro for 1M rays (a 1024×1024 frame). Intersection time was calculated by taking the difference of time to render a single primitive with no other geometry in the scene with a scene containing no geometry. The primitive occupied a large percentage of the frame. The compiler-generated code results in a factor of 2.15 to 5.67 over code that does not exploit the SIMD instructions. The **SuperEllipsoid** implementation achieves even more than the theoretical factor of 4 improvement because the complexity of the scalar code is not handled by the C++ compiler, so an alternative compilation strategy may narrow this gap somewhat.

While Manta has an hand-optimized sphere intersection, it does not have a bilinear patch nor an SSE version of superellipsoid. Manta's superellipsoid does not take advantage of ray packet special cases. We believe that this is due to the extreme amount of effort required to carefully generate these versions by hand, which our compiler does automatically. Table 4 shows a comparison between code lengths of RTSL and scalar and SSE compiler output from the Manta backend. Galileo and PBRT code is of similar length to the scalar Manta code. The length of the SSE versions of the compiled output helps drive home the point that it is exceedingly unlikely that these SSE primitive intersection routines could have been written by hand correctly, if at all. We believe that the SSE versions of the bilinear patch and superellipsoid represent the first working vectorized implementations of these primitives.

Shown in Listing 6 is an example block of code written in RTSL and its corresponding conversions to both Manta scalar and Manta SSE. The single scalar code looks much like the hand written code found in Manta, while the SSE code shows the complexity involved in a handful of expressive lines of RTSL.

```
// RTSL code
if (all(inside(vec2(u.x, v.x), 0.f, 1.f))){
  vec3 p0 = mix(mix(p00, p01, v.x),
                mix(p10, p11, v.x),
                u.x);
```

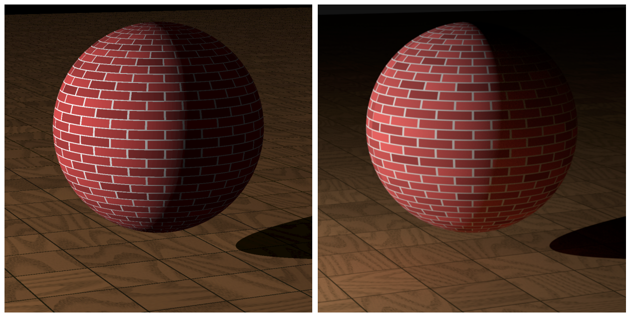

Figure 5: Procedural RTSL textures translated from RSL rendered by Manta (left) and PBRT (right). The brick and wood patterns are slightly different between images due to differences in renderer implementations of noise functions.

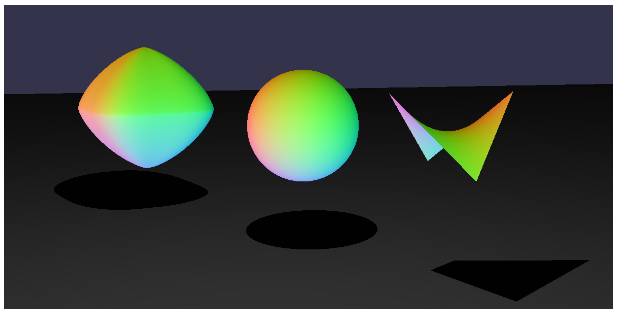

Figure 6: A superellipsoid, sphere and bilinear patch rendered by Manta with a normal visualization shader.

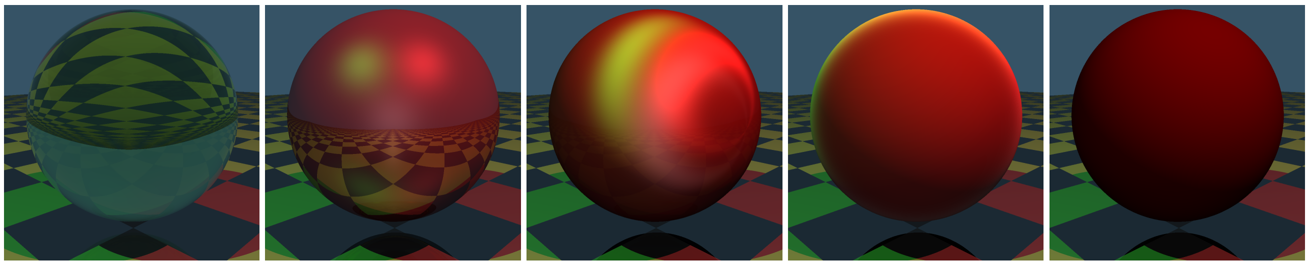

Figure 3: Our shading examples. From left to right: Dielectric, Phong, Ward, Velvet and Lambertian.

	Seconds per frame			Speedup Over	
Primitive	Scalar	SSE	Manta	Scalar	Manta
Sphere	.0048	.0022	.0023	2.15x	1.03x
Bilinear	.0544	.0161	NA	3.37x	NA
SuperEllipsoid	2.0064	.3537	2.2931	5.67x	6.48x

Table 2: Comparisons of different primitives for both SSE and Scalar output from our compiler to versions in the Manta interactive ray tracer when they are available. Numbers are seconds per frame for primitive intersection.

	Seconds per frame			Speedup Over	
Material	Scalar	SSE	Manta	Scalar	Manta
Dielectric	.4988	.2091	.7409	2.38x	3.54x
Lambertian	.0316	.0221	.0221	1.43x	1.00x
Phong	.0987	.0585	.0528	1.68x	.90x
Velvet	.1410	.0594	NA	2.37x	NA
Ward	.1369	.0654	NA	2.09x	NA

Table 3: Comparisons of different materials for both SSE and Scalar output from our compiler to versions in the Manta interactive ray tracer when possible. Numbers are seconds per frame for shading.

Shader	RTSL	Scalar	SSE
Sphere	45	119	1661
Bilinear	113	192	6152
SuperEllipsoid	181	224	3732
Brick	98	131	1074
ParquetPlank	144	190	1753
UberLight	117	178	500
Dielectric	43	229	1173
Lambertian	27	174	632
Phong	49	307	1423
Velvet	67	256	1339
Ward	72	249	1298

Table 4: Comparisons of length of code of various RTSL shaders and their compiled output from the Manta backend. Numbers are lines of code.

```
  float t0 = dot(D, p0 - O) / dot(D, D);
  hit(t0);
}

// Manta Single Scalar
VectorT<float, 2> insidearg1(VectorT<float, 2>(rtsl_u.x()
     , rtsl_v.x()));
VectorT<bool, 2> allarg2(VectorT<int, 2>(insidearg1[0] >
     (0.f) && insidearg1[0] < (1.f), insidearg1[1] > (0.f
     ) && insidearg1[1] < (1.f)));
if((allarg2[0] && allarg2[1])){
  Vector rtsl_p0 = Interpolate(Interpolate(rtsl_p00,
     rtsl_p01, rtsl_v.x()), Interpolate(rtsl_p10,
     rtsl_p11, rtsl_v.x()), rtsl_u.x());
  float rtsl_t0 = (Dot(rtsl_D, (rtsl_p0 - rtsl_O)) / Dot(
     rtsl_D, rtsl_D));
  rays.hit(i, rtsl_t0, getMaterial(), this,
     getTexCoordMapper());
}

// Manta SSE
__m128 rtsl_u_x = _mm_div_ps(_mm_xor_ps(_mm_add_ps(
     _mm_mul_ps(rtsl_Av1_, rtsl_v_x), rtsl_A1_),
     _mm_castsi128_ps(_mm_set1_epi32(0x80000000))),
     _mm_add_ps(_mm_mul_ps(rtsl_Auv1_, rtsl_v_x),
     rtsl_Au1_));
__m128 rtsl_u_y = _mm_div_ps(_mm_xor_ps(_mm_add_ps(
     _mm_mul_ps(rtsl_Av1_, rtsl_v_y), rtsl_A1_),
     _mm_castsi128_ps(_mm_set1_epi32(0x80000000))),
     _mm_add_ps(_mm_mul_ps(rtsl_Auv1_, rtsl_v_y),
     rtsl_Au1_));
__m128 insidearg13 = _mm_set1_ps(0);
__m128 insidearg15 = _mm_set1_ps(1);
__m128 insidearg16 = rtsl_u_x;
__m128 insidearg17 = rtsl_v_x;
__m128 condmask18 = _mm_and_ps(_mm_and_ps(_mm_cmpgt_ps(
     insidearg16, insidearg13), _mm_cmplt_ps(insidearg16,
      insidearg15)), _mm_and_ps(_mm_cmpgt_ps(insidearg17,
      insidearg13), _mm_cmplt_ps(insidearg17, insidearg15
     )));
int condintmask19 = _mm_movemask_ps(condmask18);
__m128 ifmask20 = _mm_and_ps(ifmask11, condmask18);
int ifintmask21 = _mm_movemask_ps(ifmask20);
if(ifintmask21 != 0){
  __m128 mixarg22 = rtsl_v_x;
  __m128 mixarg23 = _mm_sub_ps(_mm_set1_ps(1.0), mixarg22
     );
  __m128 mixarg24 = rtsl_v_x;
  __m128 mixarg25 = _mm_sub_ps(_mm_set1_ps(1.0), mixarg24
     );
  __m128 mixarg26 = rtsl_u_x;
  __m128 mixarg27 = _mm_sub_ps(_mm_set1_ps(1.0), mixarg26
     );
  __m128 rtsl_p0_x = _mm_add_ps(_mm_mul_ps(_mm_add_ps(
     _mm_mul_ps(_mm_set1_ps(rtsl_p00.x()), mixarg23),
     _mm_mul_ps(_mm_set1_ps(rtsl_p01.x()), mixarg22)),
     mixarg27), _mm_mul_ps(_mm_add_ps(_mm_mul_ps(
     _mm_set1_ps(rtsl_p10.x()), mixarg25), _mm_mul_ps(
      _mm_set1_ps(rtsl_p11.x()), mixarg24)), mixarg26));
  __m128 rtsl_p0_y = _mm_add_ps(_mm_mul_ps(_mm_add_ps(
      _mm_mul_ps(_mm_set1_ps(rtsl_p00.y()), mixarg23),
      _mm_mul_ps(_mm_set1_ps(rtsl_p01.y()), mixarg22)),
     mixarg27), _mm_mul_ps(_mm_add_ps(_mm_mul_ps(
      _mm_set1_ps(rtsl_p10.y()), mixarg25), _mm_mul_ps(
      _mm_set1_ps(rtsl_p11.y()), mixarg24)), mixarg26));
  __m128 rtsl_p0_z = _mm_add_ps(_mm_mul_ps(_mm_add_ps(
      _mm_mul_ps(_mm_set1_ps(rtsl_p00.z()), mixarg23),
      _mm_mul_ps(_mm_set1_ps(rtsl_p01.z()), mixarg22)),
     mixarg27), _mm_mul_ps(_mm_add_ps(_mm_mul_ps(
      _mm_set1_ps(rtsl_p10.z()), mixarg25), _mm_mul_ps(
      _mm_set1_ps(rtsl_p11.z()), mixarg24)), mixarg26));
  __m128 rtsl_t0_ = _mm_div_ps(_mm_add_ps(_mm_mul_ps(
     rtsl_D_x, _mm_sub_ps(rtsl_p0_x, rtsl_O_x)),
     _mm_add_ps(_mm_mul_ps(rtsl_D_y, _mm_sub_ps(rtsl_p0_y
     , rtsl_O_y)), _mm_mul_ps(rtsl_D_z, _mm_sub_ps(
     rtsl_p0_z, rtsl_O_z)))), _mm_add_ps(_mm_mul_ps(
     rtsl_D_x, rtsl_D_x), _mm_add_ps(_mm_mul_ps(rtsl_D_y,
      rtsl_D_y), _mm_mul_ps(rtsl_D_z, rtsl_D_z))));
  rays.hitWithMask(i, ifmask20, rtsl_t0_, getMaterial(),
     this, getTexCoordMapper());
}
```

Listing 6: A section of code of a bilinear patch primitive intersection in RTSL, Manta single scalar, and Manta SSE

Table 2 demonstrates that the performance of our compiled results can be competitive with manually generated code for primitives. We also examined how close we are to hand written SSE materials (see Table 3). These numbers are seconds per frame for shading using a single core of a 3.0GHz MacPro for 1M rays (a 1024×1024 frame). Time to perform the shading was calculated by taking the difference of time to compute a frame similar to the one as in Figure 3 and again with the scene with the sphere using no shading. This was done in an effort to isolate the shaders' performance from the rendering system. For this test, we compared our Phong and Dielectric implementations to those in Manta. In particular, the Phong material in Manta has been extensively hand tuned and takes advantage of a number of optimizations we do not believe a compiler would automatically generate. We also compare a few shaders that are not available in Manta, that we believe are complex enough to stress our system.

6 Discussion

Any effort to produce a new programming language involves many trade-offs, practical considerations, and matters of personal taste. Ideally, the process also incorporates knowledge from a community of stake holders. This document represents the beginning of such a process for interactive ray tracing. We now address some of the considerations that we put into the work presented here.

The Renderman Shading Language was also considered as a possibility for the base language, and if we just wanted to support materials, textures, and lights, then that may have been a good choice. However, we felt that supporting cameras, primitives, and possibly other classes in the future was important. The RTSL object model enables shade trees to be built in a more flexible manner than with RSL. Furthermore, it was attractive to build on something that is familiar to the large number of GPU programmers.

We intentionally avoided the temptation to create a language that would allow an entire ray tracing application to be built. In particular, we consciously chose to not address acceleration structures and scene definition because we felt it would complicate the language to an intractable level. Full applications are typically written by a few highly experienced architects and then extended by many, often less experienced, users. It is these extensions that we wanted to enable.

In demonstrating the effectiveness of RTSL, we targeted multiple rendering systems. Undoubtedly some design decisions may be a burden to other target systems. However, we think that Manta and Galileo are probably as different as two ray tracers could be, so we are confident that the bulk of the system could be targeted to most rendering systems. We also added a PBRT target to help ensure we covered a variety of renderers and to show how RTSL can be adapted to various systems.

RTSL provides significant advantages in developing algorithms that exploit SSE and ray packets. The advantage may not be as significant for only single-ray renderers, but we have still found it to be preferable to writing C++ code due to its simplicity. Furthermore, RTSL enables code to be used in more than one renderer.

One of our main concerns as we embarked on this project was whether the compiler could generate SIMD and packet code that was competitive with human-generated code. We were pleased that in most cases the compiler was competitive with or even outperformed hand-generated code. However, a main advantage is that the level of programming experience needed to extend a ray tracer is lowered, and the productivity for experienced programmers is raised. As in any programming, bottlenecks revealed by profiling should be examined for potential hand-tuning.

Our current compiler lowers to C++ code instead of to assembly or machine code. Furthermore, it is designed to be "readable" C++ code – generating full expressions instead of the more common but less readable single-assignment form. This design choice provides both advantages and disadvantages. It provided for a much simpler mechanism to incorporate RTSL code into existing systems, and could potentially provide a more suitable path for subsequent manual tuning. However, control-flow constructs such as loops were a bit of a challenge.

Using C++ also leverages the optimization and scheduling efforts of existing compilers. We rely on the target compiler to perform common subexpression elimination, instruction scheduling and register allocation. However, none of the currently available compilers can automatically generate packet-based SIMD code from single-ray code. Reflecting on the analysis performed in Section 4.3, a general-purpose compiler would not have the application-level knowledge to support such analysis. We anticipate that our RTSL compiler could generate assembly language for some systems such as a hardware ray tracer if the ISA of that hardware was expressive enough to include features such as procedural textures.

7 Conclusions

We have presented a shading language to allow cross-platform extensions to ray tracing programs. This language allows a ray tracing system to be extended with a simple but flexible syntax. We have also demonstrated a compiler for this language that produces code that is performance-competitive with hand-generated code, including packet-based intersection, shading and texturing. We have demonstrated this functionality on three open source ray tracers that span a large space of architectural styles. For an interactive ray tracing program we have shown that it is feasible to automatically generate both ray packet and SSE code for shading, camera ray generation, and ray-object intersection.

8 Future Work

The creation of a new language has historically been as much a result of sociological and commercial forces as of technical concerns, both foreseeable and unforeseen. We have demonstrated that RTSL can successfully achieve these goals but acknowledge that improvements are certainly possible.

One of the current limitations of RTSL is its assumption that rendering begins by tracing rays from the camera. Algorithms such as Metropolis Light Transport [21] and Photon Mapping [9] require additional functionality from RTSL classes.

To support such bidirectional transport in the future, a **Camera** may provide a *getFilmPosition* function. The purpose of this func-

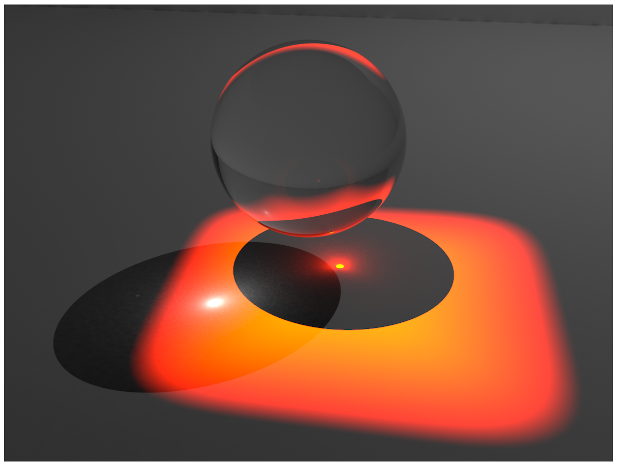

Figure 7: A scene rendered with PBRT's photon mapping plugin. This uses hand-modified output from an RTSL **Light** shader to enable bidirectional sampling.

tion is to determine where an incoming ray would intersect the film plane, if at all. Additionally, a bidirectional RTSL would require the ability to generate directional samples from light sources through an interface similar to **Material**'s BSDF sampling functions. Figure 7 shows an RTSL light being used by PBRT's photon mapping plugin. The output from this shader was hand-modified to support generating these samples.

RTSL is also an extensible system; future additions of new classes to support modules such as backgrounds, environments and ambient lights or tonemapping operators will not require modifying any current RTSL code. Volume rendering could be supported through Renderman-style volume shaders. Additional future extensions to the language could include default arguments to class constructors and support for additional or arbitrary derivative calculations within shaders. RTSL already supports conditional compilation for various renderers through the C preprocessor; a natural extension of this is renderer-specific state variables for supporting renderer specific features.

We are exploring just-in-time compilation with late binding to allow more sophisticated optimization of complex shade trees similar to Renderman. Texture mapping currently operates only on colors and scalars but perhaps a simple generic type mechanism could extend that to arbitrary types. We are examining support for additional systems, including a nascent hardware ray tracing system. We are also open to the possibility of adding support for some types of acceleration structures if it can be achieved without incurring unwanted complexity in the language and compiler. We are also interested in simplifying primitive intersection further by implementing the analysis to automatically segment the normal and texture coordinate computations, which would allow the user to implement just a single function.

We believe that the compiler could be modified to generate interval arithmetic code to allow more sophisticated packet-based culling [5]. Additional vector-based optimizations may be also be possible.

After translating several RSL shaders to RTSL by hand, we plan to create a mechanical translator from the Renderman format to our own. This would allow renderers supporting RTSL to function as Renderman previewers and allow for LPICS [14] and Lightspeed [16] style lighting design.

Acknowledgments

This work was supported by the State of Utah Center of Excellence Program, the University of Utah Brown Fellowship and the Center for Simulation of Accidental Fires and Explosions (under grant W-7405-ENG-48). We would also like to thank Peter Shirley and the Utah Render Group for helpful discussions in the design of our language.

References

[1] OpenGL(R) Shading Language. `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`.

[2] RMR Shaders. `http://www.renderman.org/RMR/RMRShaders.html`.

[3] R. Barzel. Lighting controls for computer cinematography. *J. Graph. Tools*, 2(1):1–20, 1997.

[4] J. Bigler, A. Stephens, and S. Parker. Design for parallel interactive ray tracing systems. In *the Proceedings of IEEE Symposium on Interactive Ray Tracing*, pages 187–196, 2006.

[5] S. Boulos, I. Wald, and P. Shirley. Geometric and arithmetic culling methods for entire ray packets. Technical Report UUCS-06-10, University of Utah, 2006.

[6] R. L. Cook. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH '84)*, pages 223–231. ACM Press, 1984.

[7] L. Gritz and J. K. Hahn. BMRT: a global illumination implementation of the RenderMan standard. *J. Graph. Tools*, 1(3):29–48, 1996.

[8] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH '90)*, pages 289–298. ACM Press, 1990.

[9] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30. Springer-Verlag, 1996.

[10] J. T. Kajiya. The rendering equation. In *Computer Graphics (Proceedings of SIGGRAPH '96)*, pages 143–150. ACM Press, 1986.

[11] W. Mark, R. Glanville, K. Akeley, and M. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2003)*, pages 896–907, 2003.

[12] M. Olano and A. Lastra. A shading language on graphics hardware: the Pixelflow shading system. In *Computer Graphics (Proceedings of SIGGRAPH '98)*, pages 159–168, 1998.

[13] C. Peeper and J. Mitchell. Introduction to the DirectX® 9 high level shading language, 2003.

[14] F. Pellacini, K. Vidimce, A. Lefohn, A. Mohr, M. Leone, and J. Warren. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. In *ACM Trans. on Graph. (Proceedings of SIGGRAPH '05)*, pages 464–470. ACM Press, 2005.

[15] K. Perlin. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH '85)*, pages 287–296, 1985.

[16] J. Ragan-Kelley, C. Kilpatrick, B. W. Smith, D. Epps, P. Green, C. Hery, and F. Durand. The Lightspeed automatic interactive lighting preview system. In *ACM Trans. on Graph. (Proceedings of SIGGRAPH 2007, to appear)*, 2007.

[17] R. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, 2004.

[18] R. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[19] P. Slusallek, T. Pflaum, and H. Seidel. Implementing RenderMan - practice, problems and enhancements. *Computer Graphics Forum*, 13(3):443–454, 1994.

[20] P. Slusallek, T. Pflaum, and H. Seidel. Using procedural RenderMan shaders for global illumination. *Computer Graphics Forum*, 14(3):311–324, 1995.

[21] E. Veach and L. J. Guibas. Metropolis light transport. In *Computer Graphics (Proceedings of SIGGRAPH '97)*, pages 65–76. ACM Press/Addison-Wesley Publishing Co., 1997.

[22] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):153–164, 2001.

[23] E. W. Weisstein. Fredholm integral equation of the second kind. `http://mathworld.wolfram.com/FredholmIntegralEquationoftheSecondKind.html`.

[24] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.