Writing Bxdfs

Introduction

This documentation is intended to instruct developers in the authoring of custom *bxdfs* (previously referred to as *surface shaders*). Developers should also consult the `RixBxdf.h` header file for complete details.

The `RixBxdfFactory` interface is a subclass of `RixShadingPlugin`, and defines a shading plugin responsible for creating a `RixBxdf` object from a shading context (`RixShadingContext`) and the set of connected patterns (`RixPattern`).

The `RixBxdf` interface characterizes the *light-scattering behavior* (sometimes referred to as *material response*) for a given point on the surface of an object.

`RixBxdfFactory`

Implementing the RixBxdfFactory Interface

`RixBxdfFactory` is a subclass of `RixShadingPlugin`, and therefore shares the same initialization, synchronization, and parameter table logic as other shading plugins. Therefore to start developing your own Bxdf, you can `#include "RixBxdf.h"` and make sure your bxdf factory class implements the required methods inherited from the `RixShadingPlugin` interface: `Init()`, `Finalize()`, `Synchronize()`, `GetParamTable()`, and `CreateInstanceData()`. Generally, there is one shading plugin instance of a RixBxdfFactory per bound `RiBxdf` (RIB) request. This instance may be active in multiple threads simultaneously.

Integrators (`RixIntegrator`) use `RixBxdfFactory` objects by invoking `RixBxdfFactory::BeginScatter()` to obtain a `RixBxdf.` Because a `RixBxdf` is expected to be a lightweight object that may be created many times over the course of the render, `RixBxdfFactory` is expected to take advantage of the lightweight instancing services provided by `RixShadingPlugin`. In particular, `BeginScatter()` is provided a pointer to an instance data that is created by `RixBxdfFactory::CreateInstanceData`(), which is called once per *shading plugin instance*, as defined by the unique set of parameters supplied to the material description. It is expected that the instance data will point to a private cached representation of any expensive setup which depends on the parameters, and `BeginScatter()` will reuse this cached representation many times over the course of the render to create `RixBxdf` objects.

The `RIX_BXDFPLUGINCREATE()` macro defines the `CreateRixBxdfFactory()` method, which is called by the renderer to create an instance of the bxdf plugin. Generally, the implementation of this method should simply return a new allocated copy of your bxdf factory class. Similarly, the `RIX_BXDFPLUGINDESTROY()` macro defines the `DestroyRixBxdfFactory()` function called by the renderer to delete an instance of the bxdf plugin; a typical implementation of this function is to `delete` the passed in bxdf pointer:

```
RIX_BXDFPLUGINCREATE
{
    return new MyBxdfFactory();
}
RIX_BXDFPLUGINDESTROY
{
    delete ((MyBxdfFactory*)bxdf);
}
```

`RixBxdfFactory::BeginScatter()`

As mentioned above, integrators invoke `RixBxdfFactory::BeginScatter()` to obtain a `RixBxdf`. The renderer's operating model is that the Bxdf that is obtained this way is a *closure*, with the closure functions being `GenerateSample`, `EvaluateSample, EvaluateSamplesAtIndex()`, and `EmitLocal`. The `RixBxdfFactory` should stash state in the `RixBxdf` object and consider that the `RixBxdf` lifetime is under control of the integrator. Generally integrators will attempt to minimize the number of live `RixBxdf` objects but may nonetheless require a large number. For this reason, the `RixBxdf` instances should attempt to minimize memory consumption and construction / deconstruction costs.

Any computations that the Bxdf need in order to efficient evaluate its closure functions should be computed once inside `RixBxdfFactory::BeginScatter()`, and then *saved* in the overridden RixBxdf class. Critically, these computations include upstream evaluation of any pattern networks. Therefore, it is typical for `BeginScatter()` to invoke `RixShadingContext::EvalParam()` in order to evaluate the relevant bxdf input parameters, and then pass the pointers returned from `EvalParam` to the Bxdf constructor. Since Bxdfs also generally require geometric data, or built-in variables, such as the shading normal (Nn) and viewing direction (Vn), either `BeginScatter()` or the Bxdf constructor itself will need to call `RixShadingContext::GetBuiltinVar()` function for each such built-in variable.

The following code demonstrates a trivial constant Bxdf's implementation of `BeginScatter()`. Here, the parameter id for the emission color is passed to `EvalParam()` in order to get a pointer `emitColor` that is passed to the `ConstantBxdf`() constructor.

```
RixBxdf *
PxrConstant::BeginScatter(RixShadingContext const *sCtx,
                          RixBXLobeTraits const &lobesWanted,
                          RixSCShadingMode sm,
                          RtPointer instanceData)
{
    // Get all input data
    RtColorRGB const* emitColor;
    sCtx->EvalParam(k_emitColor, -1, &emitColor, &m_emitColor, true);
    RixShadingContext::Allocator pool(sCtx);
    void *mem = pool.AllocForBxdf<ConstantBxdf>(1);
    ConstantBxdf *eval = new (mem) ConstantBxdf(sCtx, this, lobesWanted, emitColor);
    return eval;
}
```

In the following code from `PxrDiffuse`, we demonstrate how its constructor sets up required geometric information that is later on used for sample generation and evaluation.

```
    PxrDiffuse(RixShadingContext const *sc, RixBxdfFactory *bx,
               RixBXLobeTraits const &lobesWanted,
               RtColorRGB const *emit,
               RtColorRGB const *diff,
               RtColorRGB const *trans,
               RtNormal3 const *bumpNormal) :
        RixBxdf(sc, bx),
        m_lobesWanted(lobesWanted),
        m_emit(emit),
        m_diffuse(diff),
        m_transmission(trans),
        m_bumpNormal(bumpNormal)
    {
        RixBXLobeTraits lobes = s_reflDiffuseLobeTraits | s_albedoLobeTraits;
        if (m_transmission)
            lobes |= s_tranDiffuseLobeTraits;

        m_lobesWanted &= lobes;

        sc->GetBuiltinVar(RixShadingContext::k_P, &m_P);
        if(m_bumpNormal)
            m_Nn = bumpNormal;
        else
            sc->GetBuiltinVar(RixShadingContext::k_Nn, &m_Nn);
        sc->GetBuiltinVar(RixShadingContext::k_Ngn, &m_Ngn);
        sc->GetBuiltinVar(RixShadingContext::k_Tn, &m_Tn);
        sc->GetBuiltinVar(RixShadingContext::k_Vn, &m_Vn);
    }
```

`BeginScatter()` is passed an instance data pointer created via `CreateInstanceData` that can be used to cache and reuse certain calculations common amongst all factory instances of the same Bxdf; for more information, please consult the lightweight instancing discussion in `RixShadingPlugin`.

`BeginScatter()` is also passed two parameters that can be used as *hints* to optimize the calculation. `RixBXLobeTraits const &lobesWanted` is a description of the Bxdf lobes that the renderer expects to generate or evaluate; this parameter can be used to avoid any computations not necessary for the requested lobes. `RixSCShadingMode` will take either the value `k_RixSCScatterQuery`, indicating that the factory should construct a Bxdf for scattering on the surface, or `k_RixSCVolumeScatterQuery`, indicating that a Bxdf should be constructed for scattering on the inside of a volume.

`RixBxdfFactory::BeginOpacity()`

In certain cases, integrators may also call `RixBxdfFactory::BeginOpacity()` to retrieve a `RixOpacity` object. `BeginOpacity` should be implemented in a similar fashion to `BeginScatter()`, except that will be only be invoked by the renderer in narrower constraints: either for presence and opacity. As such, any inputs to the factory that do not affect presence nor opacity need not be evaluated. Furthermore, the `RixSCShadingMode` can be examined to further narrow down the inputs; it will take either the value `k_RixSCPresenceQuery` or `k_RixSCOpacityQuery`.

`RixBxdfFactory::BeginInterior()`

Bxdfs that have interesting volumetric interior properties need to implement the `BeginInterior()` method, as well as an associated `RixVolumeIntegrator`. Please consult Writing Volume Integrators for more information.

Instance Hints

Bxdfs that require special opacity handling or support interior shading need to indicate their support for these capabilities via an *instance hint*. Most Bxdfs do not require such, and should simply implement in their factory a trivial implementation of `GetInstanceHints()` which simply returns `k_TriviallyOpaque`. Bxdfs that do modulate opacity and/or require interior shading are required to override the `GetInstanceHints()` method and return the appropriate bits in `InstanceHints` to the renderer in order to trigger calls to `BeginOpacity` and `BeginInterior`.

As a further optimization, Bxdfs that deal with opacity or interiors may choose to change their behavior based upon their instance parameters. For example, they may opt out of opacity entirely if they can prove via inspection of the parameters that the intended result is equivalent to opaque. Bxdfs that choose to do so should "bake" this awareness into their instance data at the time of `CreateInstanceData()` and inspect this instance data within the `GetInstanceHints()` implementation.

Note that at the time of `CreateInstanceData`, like other shading plugins, Bxdfs are unable to inspect the values of pattern network inputs; therefore, in cases these inputs are provided (i.e: `RixParameterList::GetParamInfo()` returns `k_RixSCNetworkValue`) the Bxdf may have no choice but to defer inspection of the inputs until `BeginOpacity()` or `BeginInterior()`. At that time, those methods may then choose to return NULL instead.

`RixBxdf`

Once a `RixBxdf` object is obtained, the integrator may invoke the following methods:

- `RixBxdf::GetEvaluateDomain()` to figure out the domain over which the Bxdf evaluate samples;
- `RixBxdf::GenerateSample()` to generate samples of the bxdf function, *one sample* for *each point* of the shading context;
- `RixBxdf::EvaluateSample()` to evaluate the bxdf function, *one direction* for *each point* of the shading context;
- `RixBxdf::EvaluateSamplesAtIndex()` to evaluate the bxdf function, *one-or-many directions* for a *given point* of the shading context;
- `RixBxdf::EmitLocal()` to retrieve the bxdf's local emission.

The primary `RixBxdf` entry points operate on a collection of shading points (`RixShadingContext`) in order to reasonably maximize shading coherency and support SIMD computation. Integrators rely on the `RixBxdf`'s ability to generate and evaluate samples across the entire collection of points. Sample evaluation may be performed in an *all-points-one-sample* variant using `EvaluateSample()`, and a *1-point-n-samples* variant via `EvaluateSamplesAtIndex()`. Generation, however, is constrained to *all-points-one-sample*. Evaluation typically has different requirements (e.g. for making connections in a bidirectional integrator), whereas generation typically benefits from being performed all points at once.

The `RixBxdf` methods above are expected to return quantities for each point of the shading context originally given to `RixBxdfFactory::BeginScatter()`, excepting `RixBxdf::EvaluateSamplesAtIndex()`, that operates on a single point of the shading context, evaluating the bxdf function for one or many directions.

In order to maintain physical correctness, bxdfs are expected to conserve energy and obey the Helmholtz reciprocity principle. Care should be taken so that `RixBxdf::GenerateSample()`, `RixBxdf::EvaluateSample()` and `RixBxdf::EvaluateSamplesAtIndex()` return consistent results. This allows bxdf plugins to be compatible with different rendering techniques such as unidirectional path tracing, bidirectional path tracing, photon mapping and vertex connection merging (VCM).

Sample Generation

The `GenerateSample()` function has the following input parameters: `transportTrait`, `lobesWanted`, and random number generator `rng`.

- The `transportTrait` tells the Bxdf the subset of light transport to consider: direct illumination, indirect illumination, or both.
- `lobesWanted` specifies what lobes are requested, for example specular reflection, diffuse transmission, etc.
- `rng` should be called to generate well-stratified samples; such samples typically reduce noise and improve convergence compared to using uniform random samples.

The `GenerateSample()` function has the following output parameters (results): `lobeSampled`, `directions`, `weights`, `forwardPdfs`, `reversePdfs`, and `compTrans`. All results are arrays with one value per sample point.

- `lobeSampled` is similar to the input `lobesWanted`, and specifies which lobe was actually sampled. Bxdfs must respect the `transportTrait` and `lobesWanted` request and indicate which class of lobe is associated with each sample by setting `lobesSampled` for each generated sample.

If and only if the bxdf is unable to generate a requested sample, then `lobesSampled` should be marked as invalid by calling `SetValid(false)`; for example, if the `lobesWanted` argument requests a specific lobe (e.g., diffuse reflection) that the Bxdf does not support because the Bxdf only supports glossy reflections, then `lobesSampled[i].SetValid(false)` should be called. However, if it is possible to generate the requested samples, then `lobesSampled` should not be marked invalid and should instead be set to the sampled lobe for each point in the shading context.

When generating a Bxdf sample, you can also run into corner cases (often legitimate cases) where a valid sample cannot be generated. For instance, if the camera ray hits the backside of a single-sided object, a valid sample cannot be generated. Therefore, the `lobeSampled` for this camera ray should also be marked as invalid in this case by calling `SetValid(false)`.

If an invalid sample is returned to the integrator, the integrator will terminate the ray and avoid any further computation. Bxdfs that do not scatter light (e.g. `PxrConstant`) should also mark all samples as invalid.

> ⚠ There is a subtle difference between an invalid sample and a black sample. If a camera ray hits a diffuse black surface, the sample will have zero contribution to the final image despite being valid. It would make sense for the integrator to terminate the ray because further bounces will not contribute to the final image either, but it is important to splat the black contribution to both the beauty and alpha channel. Failing to do so might result in missing geometry in the rendered image. An invalid sample on the other hand, should not only be terminated but also ignored for splatting purposes.

- `directions` is the generated ray direction vectors; these directions **must** be unit length.
- `weights` is a color per sample indicating that sample's weight.
- `forwardPdfs` should account for light moving from the L to V direction, whereas `reversePdfs` account for the opposite (from V to L). Bxdfs should always provide both pdf values for integrators to use. Bxdfs that do not scatter light (e.g. PxrConstant) should set both pdfs to zero.
- `compTrans` is an optional result which can be used to indicate transmission color; this will be used as alpha in compositing. A bxdf should check that `compTrans` is not NULL before assigning to it.

As an example, a purely Lambertian diffuse bxdf should loop over the sample points and for each sample point set the result values as follows: set `lobeSampled` to diffuse reflection, set the reflection direction to a randomly generated direction chosen with a cosine distribution on the hemisphere defined by the surface normal (using a well-stratified 2D "random" sample value generated by calling the provided random number generator), set the weight to the diffuse surface albedo color times dot(Nn, Ln) divided by pi, set the forward pdf to dot(Nn, Ln) divided by pi, and set the reverse pdf to dot(Nn, Vn). More details can be found in the source code for the PxrDiffuse bxdf.

Sample Evaluation

The parameters to the `EvaluateSample()` function are very similar to `GenerateSample()`: `transportTrait`, `lobesWanted`, `rng`, `lobesEvaluatec`, `directions`, `weights`, `forwardPdfs` and `reversePdfs`. The main difference is that `directions` is an array of **inputs** that the function should compute weights and pdfs for.

The `EvaluateSamplesAtIndex()` function is very similar to `EvaluateSample()`, but is used to evaluate multiple samples at the same surface position and normal, but with different illumination directions (Ln). The inputs are the same as for `EvaluateSample()`, except that it has two additional inputs: `index` and number of samples `numSamples`. `index` is used to index into built-in variables such as the normal Nn and viewing direction Vn, and `numSamples` is the number of directions (Ln) to evaluate the bxdf for. The functionality of `EvaluateSamplesAtIndex()` otherwise is similar to `EvaluateSample()`, and exists in order to make sample evaluation more efficient in certain light transport settings.

Evaluation Domain

Bxdfs can help integrators converge more quickly by providing hints about the domain over which they need to be integrated (the full domain being the entire sphere). This is done by the bxdf implementing a `RixBxdf::GetEvaluateDomain()` function that returns the appropriate `RixBXEvaluateDomai` value. For more information, please see Bxdf Evaluation Domain.

`RixOpacity`

The renderer will invoke the following methods on `RixOpacity`:

- `RixBxdf::GetPresence()` to evaluate the *geometry presence*.
- `RixBxdf::GetOpacity()` to evaluate the *opacity color*.

Additional Considerations

Bxdf Evaluation Domain

Bxdf Lobes

Non-Opaque Surfaces

Alpha for Compositing