Writing Volume Integrators

Introduction

A `RixIntegrator` plugin is responsible for orchestrating the global interoperation of lights, Bxdfs and the ray tracer to produce final pixels. Each integrator implements the required plugin interface and may also offer a custom set of parameters and behaviors. For example, both `PxrPathTracer` and `PxrVCM` implement different techniques to compute global illumination and are built atop the same integrator API that can be used to develop custom integration solutions. The world of hard-surface scattering is pretty well understood but even so, there's an abundance of opportunities in graphics research and production efficiencies that can be exploited with the `RixIntegrator` API.

Integration gets significantly more difficult when we consider participating media, where photon paths can become more complex and the costs of global illumination increase dramatically. Due to the number of different classes of participating media, there is no single preferred integration technique. Rather, it's common to adopt different techniques for different media. Simple cases, like glass, can be treated as simple extensions to a path-tracing integrator. More complex cases like smoke or skin require custom integration algorithms tuned to the properties of the particular medium. Rather than build a multitude of media models into our global integrators, it's helpful to factor out the responsibilities for volume integration from the global integration problem. We distinguish between the global integration domain, managed by the single `RixIntegrator` plugin, and smaller volumetric domains, each of which are is by its own volume integrator, each of which implement the `RixVolumeIntegrator` interface.

Within a volumetric domain, we require a `RixVolumeIntegrator` plugin to respond to the global integrator's request for `GetNearestHits()` and `GetTransmission()` results. In the case of `GetNearestHits()`, a volume integrator typically executes after a `RixBxdf` generates samples for the purposes of indirect rays (i.e. to generate the next hit along the path).

`RixIntegrator` and Volumes

? Unknown Attachment

Let's go into more detail about the interplay between the global `RixIntegrator` plugin and local `RixVolumeIntegrators`. Consider the diagram above. Two rays fired from the camera hit the same surface at points A and B; this surface has a local volume integrator bound to it. Before the global integrator even considers the presence of the volume or not, the Bxdf bound to the surface generates samples (runs `GenerateSamples`) to determine the direction of indirect rays. One of these directions is a reflection. At this point, we introduce the convention that reflection rays are interested in an **incident** volume to the surface, and the global integrator asks the renderer to begin volume integration on the reflection ray via a call to `BeginIncidentVol()` on the shading context.

Because no volumes have yet been encountered, the renderer reports that the reflection ray will not go through any volume, so no volume is returned by `BeginIncidentVol()`. The global `RixIntegrator` therefore passes this ray straight to `IntegratorContext::GetNearestHits()`. So far, this ray has been treated no differently than what would have been generated had the camera rays hit a surface. Upon returning from `IntegratorContext::GetNearestHits()` we have the shading context at C, which is a surface, and we continue with the next bounce in the `RixIntegrator`.

On the other hand, suppose that the ray direction generated by the Bxdf `GenerateSamples()` execution at B is a refraction event, as inferred by the fact that N.V > 0. Several things now need to take place:

1. The Bxdf must set the `lobeSampled` on this ray to have the Transmit bit, to mark that the ray enters the interior of the surface.
2. The global RixIntegrator sees that the Transmit bit is set, and based on this convention asks the renderer for the **opposite** volume bound to the surface. In this case, because the ray is **entering** the surface, the opposite volume has the same factory as the Bxdf bound to the surface.
3. A new instance of a `RixVolumeIntegrator` object is created by a call to `RixBxdf::BeginInterior()`, and the ray is passed to this object's `GetNearestHits()` call.

It is now the primary job of the volume integrator's `GetNearestHits()` call to return D as the next hit point. As far as the primary `RixIntegrator` is concerned, that's all it has to know about. Before it returns D, however, the `RixVolumeIntegrator` has an opportunity to influence the rendered results in two broad ways:

1. It can perform some operation across the interval B-D and ask the main integrator to Splat those results. In the current design it can only do so by using the `IntegratorDelegate` interface. By doing so the volume is expected to perform some estimate of the in-scattered radiance along that interval.
2. It can compute the attenuated transmission across the interval B-D and write that into the transmission field of the shading context associated with the hit point D.

For various special cases (i.e. subsurface scattering) we can add a third item to this list:

1. A volume integrator may choose to override the ray directions computed by the Bxdf and instead shoot rays in some other direction, returning an entirely different D (or perhaps not even return a hit point at all)

Note that in this example, we've only discussed a set of two camera rays hitting a volume. Of course, a camera ray batch size is typically much higher, resulting in a set of rays that reflect and a set of rays that refract at the volume boundary; ideally, we would shade these rays in parallel in only two batches and run the volume integrator only once. To do so, the primary integrator can and should sort the rays into subsets, appropriate for delegation to the correct volume integrator.

? Unknown Attachment

In this second diagram, suppose that the blue square represents a volume and the red sphere is a diffuse solid object inside the volume. The process of running the volume shader on the interval A-B proceeds as described earlier. The new scenario happens at B: the Bxdf which runs there has no notion of volume shading because it will never allow the possibility of rays transmitting into the red sphere.

However, it's also clear from the diagram that the same volume shader that ran on the interval A-B needs to run on the interval B-C. To accommodate this scenario, when the global RixIntegrator asks for the **incident** volume at B (because the ray is a reflection ray), the same BxdfFactory associated with A is used to create a new instance of the local volume integrator . This is possible because the renderer has automatically kept track of the fact that a volume has been **entered**, but has not yet been **exited**.

This leads to the interesting scenario that two completely unrelated closures execute at B: the Bxdf associated with the red sphere, and the volume integrator associated with the blue volume. Moreover, while these executions occur nominally at the same location, the geometries are completely different. Again, the renderer automatically keeps track of most of this on behalf of the global RixIntegrator.

Finally, we can comment on the situation that arises at C: the Bxdf that executes will still set the transmit lobeSampled, signifying a desire for an opposite volume. However, the renderer at this point realizes that there is no opposite volume, because the ray will have exited the volume at this point. Therefore no volume is returned to BeginOppositeVol.

To tie this all together, here's some pseudo code to convey the RixIntegrator requirements for the tracing of volume-aware indirect rays:

```
// ictx is a RixIntegratorContext associated with primary camera rays
// sctx is a RixShadingContext associated with a group of hit points
// requiring shading. We are tracing the indirect rays traced by the
// the sctx. The bxdf GenerateSamples method has already been executed
// on the sctx to determine the directions of these rays as well as
// whether or not the rays transmit into the surface.
if (!sctx->HasVolume(k_AnyVolume))
    ictx->GetNearestHits(...);
else
    // Group the rays into 2 bundles: transmit and reflect.
    // This can be accomplished by checking the lobeSampled.GetTransmit()
    // flag on the ray
    RixVolumeIntegrator *volint;
    if( transRays)
    {
        volint = sctx->BeginOppositeVol(transRays.size(), transRays);
        if(volint)
            volint->GetNearestHits(...);
            sctx->EndVolume(volint);
        else
            ictx->GetNearestHits(...);
     }
     if (reflRays)
     {
        volint = sctx->BeginIncidentVol(reflRays.size(), reflRays);
        if(volint)
            volint->GetNearestHits(...);
            sctx->EndVolume(volint);
        else
            ictx->GetNearestHits(...);
     }
```

Volume Closures

Similar to `RixBxdf`, `RixVolumeIntegrators` are closures that are created by a subclass of `RixBxdfFactory` - in this case, by a call to `RixBxdfFactory::BeginInterior()`. At the time that `BeginInterior()` is called, a shading context is bound to the `RixVolumeIntegrator`. Pattern graph evaluation via `EvalParam()` may occur during `BeginInterior()` to evaluate any needed inputs to the volume integrator that may be used by `GetNearestHits()` or `GetTransmission()`. Hence, any outputs from these pattern graph evaluations should be saved explicitly by the plugin until these routines (`GetNearestHits()/GetTransmission`) are called (hence our use of the term *closure*).

Again, like `BeginScatter()`, `BeginInterior()` is not explicitly called by a `RixIntegrator`; it **may** be called by the renderer automatically upon a call to `RixIntegratorContext::GetNearestHits()`. In such cases, the renderer may also call `BeginScatter()` on the factory **at the same time**, create a `RixBxdf`, and bind the same shading context to both. This detail is mostly irrelevant to authors of either type of plugin. However, as an optimization, when the renderer executes the upstream pattern graphs for the `RixBxdf` and the `RixVolumeIntegrator` via `ShadingContext::EvalParam()`, both plugins will share the same renderer built-in pattern cache. If they pull on the same pattern subgraph, the shared subgraph will only need to be executed once. It's important to note at this juncture that the number of points in this shading context bound at `BeginInterior()` may not be equal to the number of rays requested by `GetNearestHits()`. For example, this would occur in the aforementioned scenario where the renderer, as an optimization, decided to reuse the same shading context for a Bxdf scatter closure and a volume closure, and the Bxdf only set the transmit `lobeSampled` on some subset of the rays - only those points where the Bxdf refracted into the surface would create rays that would be passed to the volume `GetNearestHits()` call. A ray's `shadingCtxIndex` can be used to determine which one of the points on the shading context are associated with the ray.

In certain other cases (such as the situation in the previous section where a ray bounced off a diffuse object inside a volume) the shading contexts bound to the `RixBxdf` and `RixVolumeIntegrator` by necessity must be **different**. The renderer automatically creates these different shading contexts for you. In the case of the volume integrator, the construction of the volume shading context and the call to `BeginInterior` is actually delayed until `BeginIncidentVol()` or `BeginOppositeVol()`. There will be no opportunity for reusing a pattern cache with a Bxdf in this circumstance. The renderer's operating model is that you should always execute all the pattern graph inputs to the volume integrator and not worry about caching as the renderer will try to share inputs with the RixBxdf automatically when possible.

Note that in the discussion so far, we are only talking about read-only shading contexts that are bound to the volume integrator at the time of `BeginInterior` and persist through `GetNearestHits()` and `GetTransmission()`. We will discuss later in more detail a different kind of shading context that will allow for perform shading **inside** the volume.

Transmission and Volumes

Transmission rays (often fired indirectly via lighting services) also require special handling where volumes are concerned; however, nearly all of this detail is handled by the renderer itself and does not require special handling by writers of `RixIntegrators`. Only `RixVolumeIntegrator` writers are required to implement a `GetTransmission()` method which will return the transmission between two points through a volume.

One important detail here: transmission rays are always considered to be **non-refractive**, **non-bending** rays. If a transmission ray goes through glass without any consideration of refraction, this is a departure from physically based rendering. Nonetheless, this effect can be very useful because it can enable direct lighting optimizations for objects behind the glass, which may be preferable to the more expensive use of indirect rays which would be required to actually run the `RixBxdf::GenerateSamples()` method.

By enforcing this constraint of non-bending rays, the renderer is also able to ignore the volume integrator's `GetNearestHits()` method (which actually fires real rays) and instead use the volume integrator's `GetTransmission()` method (which simply runs on an interval decided by the renderer). This distinction is important since it preserves optimizations that are inherent to transmission rays, and we expect RIS to fire many more transmission rays than camera or indirect rays.

We will not go into the full explanation of how transmission is handled internal to the renderer, but as an example of the types of issues that crop up for transmission rays, we will consider this situation of a blue glass sphere lit by a yellow square.

? Unknown Attachment

The shader bound to the glass specifically requests cheap, non-physically based shadows for transmission rays. This means that when we are direct lighting the point A, the transmission ray that goes **through** the glass to the light position at C also intersects the glass at D, but D is not considered to be a refractive event: the transmission ray does not bend. For that ray that went through the glass, the renderer must now consider the **opposite** volume integrator bound to A (which would likely be created by the glass shader itself), and run its `GetTransmission()` method on the interval between A-D to correctly attenuate the transmission of the direct lighting contribution between A and D. Note that if there is an opacity shader bound to the surface, then `GetTransmission()` is run on A-D **in addition** to the opacity shaders at A and D.

On the other hand, there may also be a possibility that we are direct lighting the point at A, but the ray to the light source does not go through the glass at B. If there is an **incident** volume bound to A the renderer must consider that volume and run the transmission method in the interval between B-D.

A Trivial Volume Integrator

As described above, the volume integrator's implementation of `GetNearestHits()` is made by the global integrator in place of a call to `IntegratorContext::GetNearestHits()` which would normally be made. In servicing this request, the volume integrator itself is likely to make use of the exact same integrator context services to sample the scene. Therefore, a volume that does absolutely nothing can simply make a call to `IntegratorContext::GetNearestHits()` in its own implementation of `GetNearestHits()` (i.e it just passes through the hits), and return a trivial transmission of 1.0:

```
class NullInterior : public RixVolumeIntegrator
{
public:
    NullInterior(RixShadingContext const *sc, RixBxdfFactory *f) :
        RixVolumeIntegrator(sc, f) {}
    void GetNearestHits(RtInt nRays, RtRayGeometry const *rays,
                        RixRNG *rng, RixBXLobeTraits const &lobesWanted,
                        RixIntegratorContext &ictx, RixLightingServices* lightingServices,
                        IntegratorDelegate *lcb,
                        // results:
                        RtInt *nGroups, RixShadingContext const **shadeGroups,
                        // optional inputs
                        RtUString const subset, RtUString const excludeSubset, bool isLightPath,
                        RtHitSides hitSides, bool isPrimary)
    {
        ictx.GetNearestHits(nRays, rays, lobesWanted, false,
                            nGroups, shadeGroups,
                            subset, excludeSubset, isLightPath, hitSides, isPrimary);
    }

    void GetTransmission(RtInt numRays, RtRayGeometry const *rays,
                         RixRNG *rng, RixIntegratorContext &ictx,
                         RtColorRGB *trans, RtColorRGB *emission,
                         RtUString const subset, RtUString const excludeSubset)
    {
        RtInt nPts = shadingCtx->numPts;
        for (int i = 0; i < nPts; ++i)
        {
            trans[i] = RtColorRGB(1.0f);
        }
    }
}
```

The corresponding `RixBxdfFactory` returns an instance of this trivial integrator in response to `BeginInterior()`. It is also required to implement the `GetInstanceHints()` method and indicate that a volume exists, via the `k_ComputesInterior` and `k_InteriorTransmission flags`. The hints allow the renderer to trivially determine (without having to run BeginInterior) whether or not a volume interior is bound to a surface.

```
class NullFactory: public RixBxdfFactory
{
    RixVolumeIntegrator *BeginInterior(RixShadingContext const *sCtx,
                                       RixSCShadingMode mode,
                                       RtConstPointer instanceData)
    {
        RixShadingContext::Allocator pool(sCtx);
        void *mem = pool.AllocForVolume<NullInterior>(1);
        return new (mem) NullInterior(sCtx, this);
    }

    int GetInstanceHints(RtConstPointer instanceData) const
    {
        int hints = k_ComputesInterior | k_InteriorTransmission;
        return hints;
    }
}
```

Beer's Law

A slightly less trivial volume that implements absorption but no scattering (to model a purely forward scattering glass interior) would still need to make a call to IntegratorContext::GetNearestHits, but would also compute and return an extinction based on the distance to the next hit after applying Beer's Law. This is a very easy extension to the trivial volume integrator.

```
class GlassInterior : public RixVolumeIntegrator
{
public:
    GlassInterior(RixShadingContext const *sc,
                  RixBxdfFactory *f, RtColorRGB const &absorption) :
    RixVolumeIntegrator(sc, f),
    m_absorption(absorption) {}
    virtual ~GlassInterior() {}

    void GetNearestHits(RtInt nRays, RtRayGeometry const *rays,
                        RixRNG *rng, RixBXLobeTraits const &lobesWanted,
                        RixIntegratorContext &ictx, RixLightingServices* lightingServices,
                        IntegratorDelegate *lcb,
                        // results:
                        RtInt *nGroups, RixShadingContext const **shadeGroups,
                        // optional inputs
                        RtUString const subset, RtUString const excludeSubset, bool isLightPath,
                        RtHitSides hitSides, bool isPrimary)
    {
        ictx.GetNearestHits(nRays, rays, lobesWanted, false,
                            nGroups, shadeGroups,
                            subset, excludeSubset, isLightPath, hitSides, isPrimary);
        for (int sg = 0; sg < *nGroups; sg++)
        {
            RixShadingContext const *hitsctx = shadeGroups[sg];
            RtFloat const* hitVLen;
            hitsctx->GetBuiltinVar(RixShadingContext::k_VLen, &hitVLen);
            for (int i = 0; i < hitsctx->numPts; i++)
            {
                RtFloat negdist = -hitVLen[i];
                RtColorRGB expw;
                expw.r = std::exp(m_absorption.r * negdist);
                expw.g = std::exp(m_absorption.g * negdist);
                expw.b = std::exp(m_absorption.b * negdist);
                hitsctx->transmission[i] = expw;
            }
        }
    }

    void GetTransmission(RtInt numRays, RtRayGeometry const *rays,
                         RixRNG *rng, RixIntegratorContext &ictx,
                         RtColorRGB *trans, RtColorRGB *emission,
                         RtUString const subset, RtUString const excludeSubset)
    {
        RtInt nPts = shadingCtx->numPts;
        for (int i = 0; i < nPts; ++i)
        {
            RtFloat negdist = -rays[i].maxDist;
            RtColorRGB expw;
            expw.r = std::exp(m_absorption.r * negdist);
            expw.g = std::exp(m_absorption.g * negdist);
            expw.b = std::exp(m_absorption.b * negdist);
            trans[i] = expw;
        }
    }
private:
    RtColorRGB m_absorption;
}
```

Shading in a Volume

Things start to become more complicated when we consider *single scattering*, or direct lighting of points inside a volume. The read-only shading context associated with a volume integrator at the beginning of the `GetNearestHits()` call is created by the renderer, and has shading values associated with the beginning of the volume interval. However, volume integration typically wants to perform direct lighting at points somewhere else, in the *inside* of the volume, chosen by some form of importance sampling.

In order to create a new volume shading context populated with values in the interior of a volume, the renderer provides the `BeginVolumeSampling()` and `EndVolumeSampling()` methods. When called upon a parent shading context, this creates a new, mutable shading context. An important point to note is that the shading context associated with the volume integrator after the call to `BeginVolumeSampling` will be replaced with the new mutable shading context that represents the interior of the volume. When `EndVolumeSampling()` is called the original non-mutable shading context will be reinstated. This new shading context has several properties similar to the mutable shading context created by `CreateMutableContext()`:

1. The new shading context inherits the built-in values of its parent shading context, except for u, v, and w, which instead reflect the position of P relative to the bounds of the volume in object space.
2. Built-in values can be updated by a call to `SetBuiltinVar()`.
3. Unlike normal shading contexts, the pattern graph remains open during the entire duration of `BeginVolumeSampling()`/`EndVolumeSampling()`
4. If certain built-in values are set, the built-in pattern graph cache will be flushed, and subsequent calls to `RixShadingContext::EvalParam()` will trigger re-evaluation of the pattern graph and new output values to be generated.

The following code illustrates the basics of creating and prepping a volume shading context for evaluation of scattering.

```
// lens is an array of the length of the volume interval,
// which was computed by a previous call to GetNearestHits.
// newRays is a copy of the rays passed into GetNearestHits.

// Get the positions associated with the beginning of the
// volume intervals.
RtPoint3 const *sP;
shadingCtx->GetBuiltinVar(RixShadingContext::k_P, &sP);

// shadingCtx is a field of the RixVolumeIntegrator.
// Create the child volume shading context
RixShadingContext const *parentCtx = shadingCtx;
RixShadingContext* vCtx = parentCtx->BeginVolumeSampling();

// Get uvw values associated with the beginning of the volume interval.
// Compute the change in uvw over the entire volume interval.
RtFloat const *u, *v, *w;
vCtx->GetBuiltinVar(RixShadingContext::k_u, &u);
vCtx->GetBuiltinVar(RixShadingContext::k_v, &v);
vCtx->GetBuiltinVar(RixShadingContext::k_w, &w);
RtVector3* duvwdRay = vpool.AllocForVolume<RtVector3>(nPts);
memset(duvwdRay, 0, nPts * sizeof(RtVector3));
for (int i = 0; i < numRays; ++i)
{
    int sCtxIndex = newRays[i].shadingCtxIndex;
    duvwdRay[sCtxIndex] = newRays[i].direction * lens[i];
}

// Allocate storage for quantities associated with the builtins for
// the sample location
RtPoint3 *vP = pool.AllocForVolume<RtPoint3>(nPts);
RtVector3 *vVn = pool.AllocForVolume<RtVector3>(nPts);
RtFloat *stepU = vpool.AllocForVolume<RtFloat>(nPts);
RtFloat *stepV = vpool.AllocForVolume<RtFloat>(nPts);
RtFloat *stepW = vpool.AllocForVolume<RtFloat>(nPts);

// Decide a set of per-ray sample locations in the volume,
// and compute P, Vn, and uvws associated with each location
for (i = 0; i < numRays; ++i)
{
    float alpha = (pick sample point in volume);
    int sCtxIndex = newRays[i].shadingCtxIndex;
    vP[sCtxIndex] = sP[sCtxIndex] + alpha * (hitP[i] - sP[sCtxIndex]);
    vVn[sCtxIndex] = -newRays[i].direction;
    for (volIndex = 0; volIndex < nVolumes; ++volIndex)
    {
        stepU[sCtxIndex] = u[sCtxIndex] + alpha * duvwdRay[sCtxIndex].x;
        stepV[sCtxIndex] = v[sCtxIndex] + alpha * duvwdRay[sCtxIndex].y;
        stepW[sCtxIndex] = w[sCtxIndex] + alpha * duvwdRay[sCtxIndex].z;
    }
}

// Update volume shading context builtins with properties of the
// sample locations
vCtx->SetBuiltinVar(RixShadingContext::k_P, vP);
vCtx->SetBuiltinVar(RixShadingContext::k_Vn, vVn);
vCtx->SetBuiltinVar(RixShadingContext::k_u, stepU);
vCtx->SetBuiltinVar(RixShadingContext::k_v, stepV);
vCtx->SetBuiltinVar(RixShadingContext::k_w, stepW);
```

Single Scattering

When delegating a `GetNearestHit()` call to a volume integrator, the primary `RixIntegrator` can choose to pass in a delegate which will provide direct lighting services. (Note that some integrators will not do this; PxrVCM is one such example.) For volume integrators that perform in-scattering computations, the lighting delegate represents one means by which the volume integrator can trigger the global integrator's direct lighting implementation at points within the volume, simply by calling `PerformDirectLighting()`.

```
// Run direct lighting delegate on volume shading context
vCtx->scTraits.shader.bsdf =
    vCtx->scTraits.volume->GetBxdfFactory()->BeginScatter(vCtx, lobesWanted, k_RixSCVolumeScatterQuery, NULL);
vCtx->scTraits.shadingMode = k_RixSCVolumeScatterQuery;
lcb->PerformDirectLighting(*vCtx, m_lobesWanted, nsteps);
```

In the code above, note that there is a new `RixBxdf` constructed for the volume shading context. This `Bxdf` is typically a **phase function**. `PxrDoubleHenyeyGreenstein` is an example of a phase function that we provide that can be used here. The use of the shading mode `k_RixSCVolumeScatterQuery` allows a single `RixBxdfFactory` to distinguish between a Bxdf associated with the surface of an object and a Bxdf associated with the volume inside the object.

Also, note that the Bxdf is constructed after calls to `SetBuiltinVar()` have taken place. This will need to occur if the Bxdf itself makes `EvalParam()` calls to query pattern graph values, and those pattern graph values require re-evaluation due to builtins having changed - in other words, the volume is **heterogeneous**. If the Bxdf does not require updating because the pattern does not change in the volume, then the bsdf can be created only once and reused for multiple invocations (the volume is **homogeneous**), however direct lighting still needs to take place at a different place in the volume.

Finally, the volume shading context must be returned to the system via `EndVolumeSampling()`:

```
// We are done with the volume shading context
parentCtx->EndVolumeSampling(vCtx, NULL);
```

Multiple Scattering

An alternate approach to handling scattering in volumes is to simply return points that are **inside** a volume to the primary `RixIntegrator`. The primary integrator will treat these points just like any points on a surface: it will own the problem of direct lighting. Since the primary integrator can also scatter again from these points, this alternate approach also allows for multiple scattering in volumes. Here, we ignore the lighting delegate, if any; this approach is also the only way to perform volume scattering for integrators that do not provide a lighting delegate (such as PxrVCM).

To do this, a `RixVolumeIntegrator` simply needs to set up a mutable shading context created by `BeginVolumeSampling()` as in the previous section to the desired locations in the volume. However, this mutable shading context cannot be itself passed out as a return value because its memory allocation is normally bounded by the lifetime of the volume integrator itself. Instead, the volume integrator must convert this to a non-mutable shading context by calling `EndVolumeSampling()` with a membership array that indicates which points should be retained.

The following code demonstrates a basic framework by which a volume integrator initially fires some rays with a bounded distance determined by a cdf normalized to an infinite length. The rays that hit create a shading context as usual (via `RixIntegratorContext::GetNearestHits()`). The rays that miss are assume to stay in the volume and it will be these rays that will give rise to a new shading context that will represent points inside the volume.

```
// Create some rays to fire and set their maxDist
// based on e.g. a cdf normalized to infinite distance.
// Rays that miss any object are assume to have stayed
// inside the volume
RtRayGeometry *newRays =
    pool.AllocForVolume<RtRayGeometry>(numRays);
for (int i = 0; i < numRays; ++i)
{
    // compute cdf
    // take xi as a random selector
    ...
    newRays[i].maxDist = -logf(1 - xi) * invDensity;
}
iCtx.GetNearestHits(numRays, newRays, lobesWanted,
    false /*don't create a shading ctx for misses*/,
    numGrps, shadeGrps,  // results
    subset, isLightPath, hitSides, isPrimary);

bool *rayTerminates = pool.AllocForVolume<bool>(numRays);
memset(rayTerminates, 0, numRays * sizeof(bool));

// Process the rays that hit something - these are the
// non-volumetric scattering events.
for (int sg = 0; sg < *numGrps; ++sg)
{
    RixShadingContext const *sCtx = shadeGrps[sg];
    for (int i = 0; i < sCtx->numPts; ++i)
    {
        int rayId = sCtx->rayId[i];
        rayTerminates[rayId] = true;
    }
}

// Create the mutable volume shading context that
// will be used as a template for the volumetric
// scattering shading context that will be passed out
RixShadingContext* vCtx = parentCtx->BeginVolumeSampling();
RixShadingContext::Allocator vpool(vCtx);

// Membership array is used for EndVolumeSampling to
// determine what points get passed out of the interior
// integrator. Set initial zero membership for all points.
RtInt *membership = vpool.AllocForVolume<RtInt>(vCtx->numPts);
memset(membership, 0, vCtx->numPts * sizeof(RtInt));

for (int i = 0; i < numRays; ++i)
{
    if (rayTerminates[i]) continue;
    int sCtxIndex = rays[i].shadingCtxIndex;

    // Indicate point is to be retained by EndVolumeSampling
    membership[sCtxIndex] = true;

    // Set builtins on vCtx
    ...
}

// Create the shading group for the points inside the
// volume
RixShadingContext const *outCtx =
    parentCtx->EndVolumeSampling(vCtx, &lobesWanted, membership);

// Append it onto the output list of shading groups
if (outCtx)
{
    shadeGrps[*numGrps] = outCtx;
    (*numGrps)++;
}
```

Subsurface

The **PxrSubsurface** Bxdf uses a built-in volume integrator called SSDiffusion, which is set up like this in the PxrSubsurface constructor:

```
RixShadingContext::Allocator pool(sCtx);
void *mem = pool.AllocForVolume<RixSSDiffusion::Params>(1);
RixSSDiffusion::Params *ssParams = new (mem) RixSSDiffusion::Params();

evalSSParams(sCtx, *ssParams); // fill in ssParams with param values
RixVolumeIntegrator *v =
    sCtx->GetBuiltinVolume(RixShadingContext::k_SSDiffusion, this);
v->SetParameters((void *) &ssParams);
return v;
```

The SSDiffusion interior integrator is not public, but in broad strokes it works like this:

1. Based on the diffuseMeanFreePath and unitLength material properties, it chooses optimal depths from which to shoot subsurface rays.
2. Feeler rays are then traced in the -N directions to determine the thickness of the object. If the thickness is shorter than the optimal depth, the subsurface rays will be shot from a shallower depth.
3. The integrator then picks a random direction for each subsurface ray and traces them by calling the regularRixIntegratorContext:: GetNearestHits() function.
4. For each ray hit, it then computes the area corresponding to the hit distance and evaluates a BSSRDF for the hit distance. The BSSRDF is a close approximation to a ground-truth reference solution (including single-scattering) as determined by Monte-Carlo simulation.
5. Finally, it assigns areas multiplied by BSSRDF values to the transmission array associated with the shading contexts bound to the ray hits.

Overlapping Volumes

Overlapping volumes represent a challenging integration problem, particularly for a path tracer. In regions of overlap, the approach adopted in RIS is to arbitrarily select a single RixVolumeIntegrator (typically the innermost volume), and run only its GetNearestHits and GetTransmission methods. However, that single integrator is allowed to query properties of all other overlapping volumes via the GetOverlappingVolumes method.

All other volume integrators that are relevant in the overlap region are automatically created by the renderer using the appropriateBeginInterior calls, so they are all bound to individual, separate RixShadingContexts. It is these shading contexts that can be queried:

```
std::vector<const RixShadingContext *> const * oVolumes =
    shadingCtx->GetOverlappingVolumes();
```

A volume integrator can now loop over these other volumes and combine or switch between other volumes as it deems fit.