Writing Integrators

- Introduction
- Implementing the RixIntegrator Interface
- Integration Context
    - Primary Rays
    - Ray Tracing
- Integration
    - Direct Lighting
        - Generating light samples, evaluating bxdf response
        - Generating bxdf samples, evaluating light contribution
    - Writing To The Display
    - Random Number Generation
    - Indirect Rays
        - Generating Bxdf Samples
        - Creating Indirect Rays
        - Trace Indirect Rays
    - Light Transport
    - Ray Differentials & Ray Spreads
        - Camera Ray Spread
        - Reflected Ray Spread
- Ray Property Queries

Introduction

This documentation is intended to instruct developers in the authoring of custom *integrators*. Developers should also consult the `RixIntegrator.h` header file for complete details.

An integrator plugin is used to model the integration of camera rays. These plugins are responsible for taking primary camera rays as input from the renderer and performing some work with these rays. Usually this work involves tracing the rays through the scene, computing the lighting on the hit points, and sending integrated results to the display services.

Implementing the RixIntegrator Interface

`RixIntegrator.h` describes the interface that integrators must implement. `RixIntegrator` is a subclass of `RixShadingPlugin`, and therefore shares the same initialization, synchronization, and parameter table logic as other shading plugins. Integrators do not support lightweight instances, and therefore `CreateInstanceData()` should not be overridden as any created instance data will not be ever returned to the `RixIntegrator`. Therefore to start developing your own integrator, you can `#include "RixIntegrator.h"` and make sure your integrator class implements the required methods inherited from the `RixShadingPlugin` interface: `Init()`, `Finalize()`, `Synchronize()`, and `GetParamTable()`.

The `RIX_INTEGRATORCREATE()` macro defines the `CreateRixIntegrator()` method, which is called by the renderer to create an instance of the integrator plugin. Generally, the implementation of this method should simply return a `new` allocated copy of your integrator class. Similarly, the `RIX_INTEGRATORDESTROY()` macro defines the `DestroyRixIntegrator()` method called by the renderer to delete an instance of the integrator plugin; a typical implementation of this method is to `delete` the passed in integrator pointer:

```
RIX_INTEGRATORCREATE
{
    return new MyIntegrator();
}
RIX_INTEGRATORDESTROY
{
    delete ((MyIntegrator*)integrator);
}
```

Integration Context

To facilitate the job of an integrator plugin, the renderer provides an *integration context* of type `RixIntegratorContext` which contains information about the primary rays, pointers to implementations of display services and lighting services, and routines to trace rays against the renderer's geometric database.

Primary Rays

Information about the primary camera rays are supplied via the `numRays`, `numActiveRays`, and `primaryRays` fields of the `RixIntegratorContext`.

The `RixShadingContext` member `int* integratorCtxIndex` links a given shading point with it associated primary ray: the shading point with index `i` is associated with the ray `primaryRays[integratorCtxIndex[i]]`.

Ray Tracing

Ray tracing services are provided by the renderer via the `GetNearestHits()` and `GetTransmission()` methods provided on the `RixIntegratorContext`.

`GetNearestHits()` is invoked by an integrator to send rays against the geometric database and return the list of nearest ray hits. Two versions of this routine are provided.

- The first version returns its ray hits in the form of a list of `RixShadingContext`. These shading contexts represent a collection of points which have had their associated Bxdfs fully executed and set up for sample evaluation and generation. Since a Bxdf evaluation may trigger an upstream evaluation of all input patterns, this call is considered to be very expensive as it invokes full shading.
- The second version of this call returns its ray hits in the form of a list of `RtHitGeometry`. No shading contexts are set up in this routine, and only information about the geometric locale is returned. This version of the call is preferred if no shading needs to be performed, such as in the case of an occlusion-only integrator.

All the shading contexts that are returned by `GetNearestHits()` must be explicitly released back to the renderer with the `ReleaseShadingContexts()` method. However:

- shading contexts that are provided by the renderer to the integrator (through `RixIntegrator::Integrate()`)
- shading contexts returned to the renderer (through `RixIntegrator::IntegrateRays()`

do not need to be released. The renderer will take care of this when appropriate.

Shading contexts associated with a bxdf closure can use a consequent amount of memory, so it is recommended to release them as soon as they are not needed anymore. This is usually possible after the integrator is done evaluating or generating samples for these bxdfs.

`GetTransmittance()` can be invoked by an integrator to compute the transmittance between two points in space. This is of use for bidirectional path tracing applications where the transmittance between vertex connections needs to be computed. It may also be used for computing shadow rays if the lighting services cannot be used for this purpose for some reason.

Both methods above need to be provided with an array of `RtRayGeometry` that have been properly initialized.

Integration

The `IntegrateRays()` method is the primary entry point for this class invoked by the renderer. The implementation of this routine is expected to fire the list of active primary rays delivered via the `RixIntegratorContext& ictx` parameter. This method has output parameters: the `numShadingCtxs` and `shadingCtxs` parameters are expected to be filled in with a list of primary shading contexts that are associated with the firing of the active primary rays. These primary shading contexts should not be released by the integrator.

> ⚠ An implementation of `IntegrateRays()` may choose to ignore the camera rays supplied in the `RixIntegratorContext` entirely, and shoot an entirely different set of rays. If it chooses to do so, it should be extremely careful about splatting with the display services, as those routines are set up to be indexed by the `integratorCtxIndex` field of the original primary rays.

The default implementation supplied for `IntegrateRays()` simply calls `RixIntegratorContext::GetNearestHits` to trace the primary rays, and passes the associated shading context results to `Integrate()`, which is the secondary entry point for this class. `Integrate()` is never directly invoked by the renderer; it is provided as an override convenience for implementors that are content with the default behavior of `IntegrateRays`. Following a call to `IntegrateRays()` (or `Integrate()`), integrators are expected to provide results to the renderer via the display services.

The type of results depends on the integrator. Most integrators will at least trace camera rays and generate results that depend on the scene geometry, but this is not mandatory. Non-physically-based integrators may generate results that do not depend on object materials or lights (e.g. `PxrVisualizer`), while physically-based integrators will usually simulate light transport, taking into account materials and light properties.

A physically-based integrator is expected to compute the amount of light coming from the camera ray hit toward the camera ray origin. This usually involves:

- Tracing camera rays provided to `IntegrateRays()` (hits are returned as `RixShadingContext` objects).
- Computing direct lighting (i.e. light coming directly from light sources), which involves:
  - Initializing lighting services for a given `RixShadingContext`
  - Generating light samples and evaluating the bxdf contribution for these
  - Generating bxdf samples and evaluating the light contribution for these
- Computing indirect lighting (i.e. light coming from non-light sources part of the scene), which involves:
  - Generating bxdf samples and tracing indirect rays for each of these

Direct Lighting

Computing direct lighting for a given batch of points (encapsulated by a `RixShadingContext` object) first requires initializing the lighting services.

```
RixLightingServices* lightingServices = integratorContext.GetLightingServices();
RixBXEvaluateDomain evalDomain = k_RixBXBoth;
RixLightingServices::Mode lsvcMode = RixLightingServices::k_IgnoreFixedSampleCount;
int fixedSampleCount = 0;
int indirectSamples = 1;
lightingServices->Begin(&shadingContext, &rixRNG, evalDomain,
                  RixLightingServices::k_MaterialAndLightSamples,
                  lsvcMode,
                  RixLightingServices::SampleMode(), // defaults
                  &fixedSampleCount,
                  totalDepth,
                  indirectSamples);

//
// computeDirectLighting(...)
//

lightingServices->End();
```

Once lighting services have been initialized, it is possible to ask for light sample generation and evaluation. Note that bxdf sample generation and evaluation is available as soon as `RixShadingContext` objects have been returned by `GetNearestHits()`.

In a standard Multiple Importance Sampling computation, we need to

- generate bxdf samples and evaluate light contribution for each of them
- generate light samples and evaluate bxdf response for each of them

Note that because the bxdf API returns multiple-lobe results, we need to setup `RixBXLobeWeights` objects beforehand (instead of dealing with a simple `RtColorRGB` per sample). This requires setting up `RtColorRGB` buffers of appropriate size.

Generating light samples, evaluating bxdf response

```
// numLightSamples is the number of light samples generated for *each* shading point.
// Currently, this value is used for all points in the current shading context.

// RixLightingServices::GenerateLightSamples() fills array parameters with the first
// sample for all shading points first, then the second sample, and so on...

// m_ClDiffuse, m_ClSpecular, m_ClUser are arrays of RtColorRGB buffers. Each buffer is of size
// numLightSamples * numPoints.  They will store the generated light samples.

RixBXLobeWeights lightContributions(
    numLightSamples * numPoints,
    m_numPotentialDiffuseLobes,
    m_numPotentialSpecularLobes,
    m_numPotentialUserLobes,
    m_ClDiffuse,
    m_ClSpecular,
    m_ClUser);

// m_diffuse, m_specular, m_user are arrays of RtColorRGB buffers. Each buffer is of size
// numLightSamples * numPoints. They will store the bxdf contribution for each light sample.

RixBXLobeWeights evaluatedMaterialContributions(
    numLightSamples * numPoints,
    m_numPotentialDiffuseLobes,
    m_numPotentialSpecularLobes,
    m_numPotentialUserLobes,
    m_diffuse,
    m_specular,
    m_user);

// For additional description of the call parameters, see RixLightingServices API.
lightingSvc->GenerateSamples(
    numLightSamples, &rixRNG, lightGroupIds, lightLpeTokens, directions, lightNormals, distance,
    &lightContributions, transmission, nullptr, lightPdf,
    lobesWanted, &evaluatedMaterialContributions, evaluatedMaterialFPdf, evaluatedMaterialRPdf,
    lobesEvaluated, nullptr, throughput);


// We don't need to make an explicit call to the bxdf's EvaluateSamples(), because the lighting
// services have done it for us, since we provided them with 'evaluatedMaterialContributions'.
```

Generating bxdf samples, evaluating light contribution

```
// numBxdfSamples is the number of bxdf samples generated for *each* shading point.
// Currently, this value is used for all points in the current shading context.

RixBXLobeWeights bxdfContributions(

    numBxdfSamples* numPoints,
    m_numPotentialDiffuseLobes,
    m_numPotentialSpecularLobes,
    m_numPotentialUserLobes,
    m_diffuse,
    m_specular,
    m_user);

// The RixBxdf GenerateSample API is single-sample (per shading point), so when dealing with
// multiple bxdf samples, we need to wrap it inside a loop.
for (int bs = 0; bs < numBxdfSamples; bs++) {
    int offset = bs * numPoints;

    // Changing the offset of the lobe weights will write into the lobe weights at the appropriate
    // offset for this set of bxdf samples.
    bxdfContribution.SetOffset(offset);

    bxdf.GenerateSample(k_RixBXDirectLighting, lobesWanted, &rixRNG,
                        lobeSampled + offset, directions + offset,
                        bxdfContributions, materialFPdf + offset,
                        materialRPdf + offset, nullptr);

    for (int i = 0; i < numPoints; i++) distances[offset + i] = 1e20f;

    incRNG(shadingContext);
}

// Reset the offset of the lobe weights back to zero for the code below.
bxdfContributions.SetOffset(0);

RixBXLobeWeights lightContributions(
    numBxdfSamples * numPoints,
    m_numPotentialDiffuseLobes,
    m_numPotentialSpecularLobes,
    m_numPotentialUserLobes,
    m_ClDiffuse,
    m_ClSpecular,
    m_ClUser);

lightingSvc->EvaluateSamples(
    // inputs
    numBxdfSamples, &rixRNG, directions, distances, materialFPdf, &bxdfContributions, lobeSampled,
    // outputs
    lightGroupIds, lightLpeTokens, &lightContributions, transmission, nullptr, lightPdf,
    nullptr, nullptr, throughput);
```

Writing To The Display

Once the final contribution for a given shading point has been computed, the `RixDisplayServices` API can be used to splat this contribution to the appropriate pixel. The integrators do not have direct access to the pixels, instead they have to provide the display services with the appropriate integrator context index (which can be found in `RixShadingContext::integratorCtxIndex`).

```
// Writing to display services. 'ciChannelId' is the id associated with the 'Ci' channel.
RixDisplayServices* displayServices = integratorContext.GetDisplayServices();

// These point to the final contribution and alpha values we want to splat to the pixels.
RtColorRGB* finalContributions = ...;  // of size shadingContext->numPts
RtColorRGB* finalAlpha = ...;          // of size shadingContext->numPts

for (int i = 0; i < shadingContext.numPts; i++)
{
    displaySvc->Splat(ciChannelId, shadingContext.integratorCtxIndex[i], finalContributions[i]);
    displaySvc->WriteOpacity(ciChannelId, shadingContext.integratorCtxIndex[i], finalAlpha[i]);
}
```

Random Number Generation

You can find more about using RixRNG here. This document will help you understand how to improve sampling strategies.

Indirect Rays

In addition to compute direct lighting (as described above), physically-based integrators also need to deal with indirect lighting. This is done by casting secondary rays from the camera hits, and performing a full lighting computing on the secondary hit points. Since this involves both computing direct and indirect lighting, this is a recursive process.

The integrator is responsible for creating secondary rays (usually using the bxdf to do so), and trace them by calling `RixIntegrator::GetNearestHits()`. The integrator will then use the returned `RixShadingContext` objects to compute direct and indirect lighting, similarly to what was done in `RixIntegrator::Integrate()`.

In order to get the directions and weights of the indirect rays, integrators should use the bxdf `GenerateSamples()` method. Tracing indirect rays can be split into 3 steps.

Generating Bxdf Samples

```
RixBXLobeWeights lw(
    numIndirectSamples * numPoints,
    m_numPotentialDiffuseLobes,
    m_numPotentialSpecularLobes,
    m_numPotentialUserLobes,
    m_diffuse,
    m_specular,
    m_user);

// Generate the indirect ray directions based on the bxdf.
for (int bs = 0; bs < numIndirectSamples; bs++)
{
    int offset = bs * numPoints;

    // Changing the offset of the lobe weights will write into the lobe weights at the appropriate
    // offset for this set of bxdf samples.
    lw.SetOffset(offset);

    bxdf.GenerateSample(
        k_RixBXIndirectLighting,
        m_lobesWanted,
        &rng,
        m_lobeSampled + offset,
        m_directions + offset,
        lw,
        m_FPdf + offset,
        m_RPdf + offset,
        nullptr);

    for (int i = 0; i < npoints; i++) m_distances[offset + i] = 1e20;
}

// Resets the offset of the lobe weights back to zero for the code below.
lw.SetOffset(0);
```

Creating Indirect Rays

```
// Initializes rays to be traced. We may not have to trace as many rays as bxdf samples were
// generated, since some of the bxdf weights may be zero, or we may use russian roulette, so we keep
// a count of the rays to process.
int currentRay = 0;
for (int bs = 0; bs < numIndirectSamples; bs++)
{
    for (int i = 0; i < numPoints; i++)
    {
        int sampleIndex = bs * numPoints + i;

        int rayId = sCtx.rayId[i];

        RtRayGeometry& ray = m_rays[currentRay];
        ray.origin = bias(P[i], Ngn[i], m_directions[sampleIndex], biasValue);
        ray.maxDist = m_distances[sampleIndex];

        ray.rayId = currentRay;
        ray.originRadius = iradius[i];
        ray.lobeSampled = lobeSampled;
        ray.wavelength = wavelength ? RtRayGeometry::EncodeWavelength(wavelength[i]) : 0;
        // Compute ray spread for the lobe
        ray.SetRaySpread(lobeSampled, iradius[i], ispread[i], curvature[i], m_FPdf[sampleIndex]);
        ray.InitOrigination(&sCtx, Ngn, i);

        currentRay++;
    }
}
int numRays = currentRay;
```

Trace Indirect Rays

```
// Let's trace the rays
int* numShadingCtxs;
RixShadingContext const** shadingCtxs;

iCtx.GetNearestHits(numRays, m_rays, lobesWanted, false, numShadingCtxs, shadingCtxs);
```

Light Transport

The final pseudo-code for computing light transport is the following:

```
RixIntegrator::Integrate(numSCtxs, sCtxs)
        ComputeLightTransport(numSCtxs, sCtxs)
        Splat results to display services

ComputeLightTransport(numSCtxs, sCtxs)
        For each shading context sCtx:
                ComputeDirectLighting(sCtx)
                ComputeIndirectLighting(sCtx)

ComputeDirectLighting(sCtx)
        InitializeLightingServices()
        GenerateLightSamples()
        EvaluateBxdfSamples()
        Compute MIS weights
        GenerateBxdfSamples()
        EvaluateLightSamples()
        Compute MIS weights

ComputeIndirectLighting(sCtx)
        iRays = CreateIndirectRays(sCtx)
        (numSCtxs, sCtxs) = TraceIndirectRays(iRays)
        ComputeLightTransport(numSCtxs, sCtxs)
```

Ray Differentials & Ray Spreads

Ray differentials determine texture filter sizes and hence texture mipmap levels (and texture cache pressure in scenes with many textures).

In RIS the goal for ray differential computation was improved efficiency (over REYES), even if it's not going to give quite as accurate ray differentials in all cases. Auxiliary ray-hit shading points are no longer created, and we only compute an isotropic ray "spread" - not a full anisotropic set of ray differentials. The ray spread expresses how much the ray gets wider for every unit of distance it travels.

Camera Ray Spread

By default, the spread of camera rays is set up such that the *radius* of a camera ray is 1/4 pixel. The *width* of the camera ray is two times its radius, ie. 1/2 pixel. Footprints are constructed at ray hit points such that a camera ray hit footprint is 1/2 pixel wide. Equivalently, the *area* of a camera ray footprint is 1/4 pixel. (This is true independent of image resolution and perspective/orthographic projection.)

This choice of default camera ray spread has both a theoretical and a practical foundation. *Theory:* footprints that are 1/2 pixel wide match the Nyquist sampling limit. *Practice:* our experiments indicate that footprints smaller than 1/2 pixel wide do not sharpen the final image, but footprints wider than that do soften the final image. Moving to smaller than 1/2 pixel width is all pain (finer mipmap levels, more texture cache pressure), no gain (no image quality improvement). Moving to wider than 1/2 pixel is more subjective: some people prefer the sharp look, some prefer the softer look.

Reflected Ray Spread

For reflection we compute the reflected ray spread using two approaches:

1. Ray spread based on surface curvature. The ray spread for reflection from a curved smooth surface is simple to compute accurately using Igehy's differentiation formula:

   ```
   spread' = spread + 2*curvature*PRadius
   ```

2. Ray spread based on roughness (pdf). The ray spread from a flat rough surface depends on roughness: the higher the roughness the lower the pdf in a given direction; here we map the pdf to a ray spread using a heuristic mapping:

   ```
   spread' = c * 1/sqrt(pdf) -- with c = 1/8
   ```

We set the overall ray spread to the max of these two.

This ray spread computation is done in the SetRaySpread() function (see RixIntegrator.h), which is called from the various RIS integrators. Integrator writers can easily make their own version of SetRaySpread() using other techniques and heuristics and call that from their integrators.

Ray Property Queries

Implementors of `RixBxdf` or other shading plugins may want to query ray properties such as the ray depth or eye throughput, in order to allow for artistic control or optimization. For instance, as an optimization a `RixBxdf` may want to skip the evaluation of a particularly expensive lobe, if the current ray depth of the hit point is beyond some arbitrary threshold.

Since it is the integrator that is best suited for tracking such ray properties, we require that user-authored integrators that would like to participate in such ray property queries to override the `GetProperty()` routine and provide the necessary information as requested by a `RixBxdf` or other shading plugin. Integrators that do not implement ray property queries should return `false` from `GetProperty()`, and the caller that is attempting to ask the integrator for the property must recover gracefully by not implementing the optimization.

The definition of `enum RayProperty` is in `RixShading.h`, and matches the `GetProperty()` call from `RixShadingContext`; in fact, the implementation of `RixShadingContext::GetProperty()` simply turns around and calls `RixIntegrator::GetProperty()`. Implementors should expect that some rays may be invalid, as signalled by a `rayId` value less than zero. It is the caller's responsibility to allocate the correct amount of storage (i.e. the implementor of the callback in `RixIntegrator` does not need to allocate the memory). The expected output return values in `result` for each value of `RayProperty` are as follows:

- `k_RayDepth`: `result` is expected to be an `int *`, and should be filled in with the current depth of the ray with matching `rayId` associated with the current `IntegrateRays` invocation. The implementor must check for `rayId` < 0 and return a -1 depth if such a ray ID is encountered.
- `k_RayRngSampleCtx`: `result` is expected to be `RixRNG::SampleCtx*`, and should be filled in with a copy of the appropriate `RixRNG::SampleCtx` that ensures decent stratification results for the ray with matching `rayId`.
- `k_RayThruput`: `result` is expected to be RtColorRGB `*`, and should be filled in with the current thruput to the eye of the ray with matching `rayId` associated with the current `IntegrateRays` invocation.
- `k_RayVolumeScatterCount`: result is expected to be an int *, and should be filled in with the number of times a volume direct light scattering event has occurred during the current `IntegrateRays` invocation for the given ray with matching `rayId`.
- `k_RayVolumeSampleCount`: result is expected to be an int *, and should be filled in with the number of times a volume sample was taken during the current `IntegrateRays` invocation for the given ray with matching `rayId`.