Writing Lights

Introduction

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

The `RixLightFactory` interface is a subclass of `RixShadingPlugin`, and defines a shading plugin responsible for creating a `RixLight` object.

The `RixLight` interface characterizes the light emitting from an analytic light source - a light source that can be described programmatically or by a formula.

RixLightFactory

RixLightFactory 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 Light, you can `#include "RixLight.h"` and make sure your light 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 `RixLightFactory` per bound `RiLight` (RIB) request. This instance may be active in multiple threads simultaneously.

The RIX_LIGHTFACTORYCREATE() macro defines the CreateRixLightFactory() function, which is called by the renderer to create an instance of the light factory plugin. Generally, the implementation of this method should simply return a new allocated copy of your light factory class. Similarly, the RIX_LIGHTFACTORYDESTROY() macro defines the DestroyRixLightFactory() function called by the renderer to delete an instance of the light factory plugin; a typical implementation of this method is to delete the passed in light factory pointer:

```
RIX_LIGHTFACTORYCREATE
{
    return new MyLightFactory();
}

RIX_LIGHTFACTORYDESTROY
{
    delete ((MyLightFactory*)factory);
}
```

RixLight

`RixLight` is the abstract base class from which you can derive your own light implementations. To illustrate the API, we have provided PxrSimpleRectLight.cpp, which implements a simple non-textured single-sided light of rectangular shape. Note that the PxrRectLight that ships with RenderMan offers more features than illustrated here, and uses more sophisticated sampling strategies. It also supports bidirectional sampling and photon emission, which the example does not.

The light's constructor is called by the corresponding `PxrSimpleRectLightFactory`.

We have two methods used to communicate geometric properties to the ray tracer. `GetBounds()` returns a sequence of points describing the bounding shape of the light. The bounds should be expressed in the local space of the light. For our rect light example, there are four points in the range +/- 0.5 in x and y. The rect light lies on the z=0 plane. Intersect, the second method, will compute an intersection between the light and an incoming ray. The intersection is computed in the local space of the light. A consequence of this is that the ray direction will not be normalized if the light's transform contains a scale. It's important, therefore, not to make use of any optimisations in your intersection function that does assume a unit length direction.

Light selection methods

There are three methods that act as helpers for the renderer's light selection scheme. Light selection is a stochastic process whereby, according to integrator settings, one or more lights are assigned to a shade point in a rendering iteration. The lights that are selected have samples generated for them (see below). The purpose of selection is to attempt to choose the lights liable to contribute most to the shade point in question, thereby keeping variance low.

RixLight::GetIncidentRadianceEstimate()

```
    virtual RtFloat GetIncidentRadianceEstimate(
        RtPoint3 const& P,
        RtMatrix4x4 const& lightToCurrent,
        RtMatrix4x4 const& currentToLight) const = 0;
```

To help with this calculation, the renderer will call `GetIncidentRadianceEstimate()` on the light, providing both the position of the shade point (in 'current' space) and a pair of transforms. In our RectLight example, we check to see if the shade point lies to the front of the light. If it does, we multiply its intensity by its area (which may be non-unity in the event of a scale transform) and the cosine of the angle between its normal and the vector between shade point and light center. We then divide by the squared distance to the light center and return the result.

RixLight::GetIncidentRadianceEstimate()

```
    virtual RtFloat GetIncidentRadianceEstimate(
        RtPoint3 const& segmentOrigin,
        RtVector3 const& segmentDir,
        RtFloat segmentLen,
        RtMatrix4x4 const& lightToCurrent,
        RtMatrix4x4 const& currentToLight,
        RtFloat& minT,
        RtFloat& maxT) const = 0;
```

A second overload of `GetIncidentRadianceEstimate()` is used to compute estimates for ray segments rather than individual points. This is used exclusively for equiangular sampling of volumes. In our example, we find the nearest point on the incoming line segment to the light and then treat that just as the shade point in the simpler case. Note that this overload has minT and maxT as return values. These can be used to 'clip' the line segment, providing a subset over which the light provides non-zero illumination. For example, since the rect light is single-sided, we could clip the segment against the light's plane. Similarly, if the light was a spot light, we could clip the segment against the cone's frustum.

RixLight::GetPowerEstimate()

```
    virtual float GetPowerEstimate(RtMatrix4x4 const& xform) const = 0;
```

`GetPowerEstimate()` should return the light's intensity by its area. This is a crude estimate given independent of any shade point.

RixLight::GenerateSamples()

```
struct GenerateSamplesResults
{
public:
    int& patchIndex; // only set by mesh lights
    RtFloat3& UVW;
    RtVector3& direction;
    float& distance;
    float& pdfDirect;
    bool const isBidirectional;
    float& pdfEmit;
    float& pdfEmitDirection;
    float& solidAngleToArea;
    RtColorRGB diffuseColor;
    RtColorRGB specularColor;
    RtNormal3& normal;
};
virtual void GenerateSamples(
    RixLightContext const& lCtx,
    RixScatterPoint const& scatter,
    GenerateSamplesResults& results) const = 0;
```

`GenerateSamples()` is the function used to create a sample on the light and put it in the `GenerateSamplesResult` structure, defined in RixLight.h. `UVW` indicates the position of the sample in the light's parametric space; direction is the normalized vector from the shade point to the light sample position in 'current' space; distance is the distance between the two points; and pdf is the pdf of the chosen point in solid angle measure. In the example case, we have a uniform probability of sampling across the light's surface, so the area pdf is 1/area. This is then converted to solid angle measure by multiplying by the cosine of the angle between light and outgoing direction, and dividing by the squared distance. The light returns both radiance in both `diffuseColor` and `specularColor`. These will be interpreted separately by a bxdf's diffuse and specular lobes, and allows for a light to contribute different radiances for each. The light should also return the local-space normal at the sampled point on the light. (The normal is constant in the example rect light.) Note that the input `RixLightContext` grants the function access to the sample's time in normalized shutter time (ie 0 at shutter open and 1 at shutter close); a function `GetLightToCurrentTransform()` will return a matrix at the appropriate time, and gives access to a random-number pair in a well-stratified sequence. A flag on the `GenerateSamplesResult` indicates whether the light is being used in a bidirectional setting. If so, it expected to provide three further return values (not covered by the example). `solidAngleToArea` is a conversion factor to convert between the two pdf measures. For a rect light, this would be the cosine of the angle between light normal and the direction vector divided by the squared distance. `pdfEmit` is the probability of emitting a photon from the selected sample position on the light, again expressed in a solid angle measure. (For a rect light with a uniform sampling scheme, `pdfEmit` would be 1 /area.) `pdfEmitDirection` is the probability of emitting a photon in the selected direction given the selected sample position. (For a rect light with cosine emission distribution, this would be cos(theta) / PI.)

RixLight::EvaluateSamples()

```
struct EvaluateSamplesResults
{
    float& pdfDirect;
    bool const isBidirectional;
    float& pdfEmit;
    float& pdfEmitDirection;
    float& solidAngleToArea;
    RtColorRGB diffuseColor;
    RtColorRGB specularColor;
    RtNormal3& normal;
};
virtual void EvaluateSamples(
    RixLightContext const& lCtx,
    RixSamplePoint const& sample,
    RixScatterPoint const& scatter,
    EvaluateSamplesResults& results) const = 0;
```

`EvaluateSamples()` is called so that the light can compute intensity and angular-measure pdf for an incoming ray direction (typically generated by sampling a Bxdf). `EvaluateSamples()` will only be called for a ray if a previous Intersect call returned true for the same ray. Results are returned in the `EvaluateSamplesResult` structure, definied in RixLight.h. '`pdfDirect`' is the solid-angle-measure pdf for the ray; `diffuseColor` and `specularColor` are the light's contribution for diffuse and specular lobes respectively, and 'normal' is the light's surface normal at the point of intersection. The bidirectional result quantities are the same as described above for `GenerateSamples()`.

RixLight::GenerateEmission()

```
struct GenerateEmissionResults
{
    int& patchIndex; // only set by mesh lights
    RtFloat3& UVW;
    RtPoint3& position;
    RtNormal3& normal;
    RtVector3& direction;
    float& distance;
    float& pdfEmit; // area measure
    float& pdfEmitDirection;
};
virtual void GenerateEmission(
    RixLightContext const& lCtx,
    GenerateEmissionResults& results) const = 0;
```

`GenerateEmission()` is the function used to create photons from the light, used in a bidirectional pathtracing context. Note that it requires four random numbers: two for picking a point on the surface (with uniform probability in our example) and two for picking a direction (with a cosine distribution). Note that in this special case, since we don't at this stage in the process of a shade point, the pdfs are **not** in the solid angle measure. We return `pdfEmit` and `pdfEmitDirection` (see above) and the renderer will employ a solid-angle-measure conversion once the emitted photon has struck a surface internally.

RixLight::EvaluateEmissionForCamera()

```
struct EvaluateEmissionForCameraResults
{
    RtColorRGB cameraColor;
};
virtual void EvaluateEmissionForCamera(
    RixLightContext const& lCtx,
    RixSamplePoint const& sample,
    RixScatterPoint const& scatter,
    EvaluateEmissionForCameraResults& results) const = 0;
```

`EvaluateEmissionForCamera()` will be called if a light is marked as camera-visible and is intersected by a camera ray. Its result is returned in the `EvaluateEmissionForCameraResults` structure, which contains the single color field `cameraColor`.

RixLight::Edit()

```
 virtual RixLight* Edit(
    RixContext& ctx,
    RtUString const name,
    RixParameterList const* pList,
    RtPointer instanceData) = 0;
```

`Edit()` is the function that will be called after any changes are made to the light properties. It is expected to update the class members for any subsequent sampling. Note that in more sophisticated lighting examples, this could involve such things as computing a new CDF table for a textured light.