Light Path Expressions (LPEs)

Introduction

Production renderers often output multiple display channels. For example, one output image of the renderer may contain all of the illumination that appears in the final rendered image, while separate output images may consist of just the skin subsurface scattering illumination, direct specular illumination, indirect diffuse illumination, the illumination from a particular group of lights, the illumination on a specific set of geometry, and so forth.

Light Path Expressions (LPEs) in RenderMan are used to specify what light transport paths to output to various display channel outputs. The Rix Light Path Expression (`RixLPE`) interface allows a person writing a custom integrator to communicate information about light scattering events that is then used to route specific light transport paths to various output display channels, as indicated by the user-supplied Light Path Expressions.

An advantage of Light Path Expressions is that by allowing the user to write a short expression that describes the set of light transport paths that should be included in a particular display channel output, there is no need to modify the shader or BxDF source code if the routing of light transport paths to output display channels is later changed. Instead the `RixLPE` system tracks the light scattering events along each light transport path internally, and is able to generate new output display channels for an arbitrary set of user-supplied Light Path Expressions.

The source code of the `PxrDirectLighting` and `PxrPathTracer` integrators provides an example of how to use the `RixLPE` interface, and the documentation below shows the specific API calls one would make to add tracking of light scattering events to an integrator in order to support Light Path Expression display channel outputs.

Initialization

In order to use the `RixLPE` interface, first obtain a `RixLPE` instance:

```
RixLPE *rixLPE = integratorCtx->GetRixLPE();
```

Then allocate `RixLPEState` instances (typically one per light transport path):

```
RixLPEState *states = rixLPE->AllocateStates(maxShadingCtxSize);
```

The `AllocateStates()` method takes a parameter that indicates the number of `RixLPEState` instances to allocate. In the example here, "`maxShadingCtxSize`" is used (the maximum number of shading samples per shading context that will be passed into the integrator per batch), which means that we will be able to track scattering events separately for each shading sample in the shading context.

In general, a separate `RixLPEState` should be allocated for each light transport path that will be processed per batch of shading samples in the shading context.

State Transitions

Next, we can track the light scattering events along each light transport path. At each scattering event along the light path, call `MoveCamera()` and then `MoveVertex()` on the `RixLPEState` instance corresponding to that light path in order to track relevant scattering events. For example:

```
states[sCtxIndex].MoveCamera(sCtx, sCtxIndex);
states[sCtxIndex].MoveVertex(sCtx, sCtxIndex, scatterEvent1); // primary hit
states[sCtxIndex].MoveVertex(sCtx, sCtxIndex, scatterEvent2); // secondary hit
```

In the calls to `MoveCamera()` and `MoveVertex()` above, the shading context sample is identified using the shading context index variable ("`sCtxIndex`"). The `RixShadingContext` instance (the "`sCtx`" variable in the example code above) provides the shading context to the API calls so that any needed context state is accessible. In this example, we have just one light transport path per sample in the shading context batch; a different indexing scheme into the array of `RixLPEState` instances could be used if there were more than one light transport path per shading sample in the batch.

The `MoveVertex()` call takes an additional `RixLPEScatterEvent` parameter (the "`scatterEvent1`" and "`scatterEvent2`" variables above) that provides information about the type of light scattering event.

Recording Results

After the light scattering events have been recorded using `MoveCamera()` and `MoveVertex()` along the light transport path, use the `RixLPE::SplatHelper` class to aid with writing results to the LPE AOV display channels:

```
RixLPE::SplatHelper aovs(...);
aovs.SplatPerLobe(lobeWeights, weightIndex, thruput, isFinite, clamp, isHoldout);
```

Note that `SplatHelper::SplatPerLobe()` is a utility routine that performs the final direct lighting (or emissive object) light path transitions and will accumulate the per-lobe contributions into the beauty and LPE AOVs. (See the implementation of `SplatPerLobe()` in `RixLPEInline.h` for details.)

The parameters to `SplatHelper::SplatPerLobe()` supply information about the per-lobe contributions (the "`lobeWeights`" argument in the example above) and provide an index into the per-lobe contributions array ("`weightIndex`"). The method also takes as input the path throughput ("`thruput`"), whether the contribution is finite ("`isFinite`"), a clamping factor ("`clamp`"), and whether this is a holdout contribution ("`isHoldout`"). If any of the geometry along the light path is a holdout piece of geometry, then the contribution for that light path is a "holdout" contribution, and "true" should be passed in for the "`isHoldout`" parameter.

Alternatively, some integrators may wish to manually perform the final state transitions that reach a light or emissive object, and then make separate per-lobe invocations of the `SplatHelper::SplatValue()` method to accumulate contributions for each lobe. E.g.,:

```
states[sCtxIndex].MoveVertex(sCtx, sCtxIndex, scatterEvent3);
states[sCtxIndex].MoveLight(sCtx, sCtxIndex, ...);
RixLPE::SplatHelper aovs(...);
aovs.SplatValue(lobe1Contribution, ...);
aovs.SplatValue(lobe2Contribution, ...);
```

For emissive objects, the `SplatHelper::SplatEmission()` method is used to record the illumination scattering event information:

```
aovs.SplatEmission(emission, thruput, isFinite, clamp, isHoldout);
```

Similar to the `SplatHelper::SplatPerLobe()` method above, the `SplatHelper::SplatEmission()` method takes as input information about the contribution for this light path (the "`emission`" argument), the path throughput ("`thruput`"), whether the contribution is finite ("`isFinite`"), a clamping factor ("`clamp`"), and whether this is a holdout contribution ("`isHoldout`").

Cleaning Up

When the integrator is finished with the `RixLPEState` instances, then use `RixLPE::FreeStates()` to free the memory of the `RixLPEState` instances:

```
rixLPE->FreeStates(maxShadingCtxSize, states);
```

Checking for LPE Existence

It is possible to check for the existence of any user-supplied Light Path Expressions in the scene. If there are no Light Path Expressions in the scene, integrators can avoid performing some work as an optimization.

Use the `RixLPE::AnyLPEs()` method to determine whether there are any light path expression (LPE) AOV display channels at all.