Writing Light Filters

Introduction

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

Light filters change the color of a light after the light has been sampled.

Binding and linking

Light filters are bound to lights like bxdfs are bound to geometry: they obey attribute scoping and the last one specified wins.

```
LightFilter "PxrGobo" "gobo1"  "string coordSys" ["Gobo1"] "string map" ["ratGrid.tx"]
Light "PxrPrectLight" "light1"
```

Like lights, light filters are enabled by default. Light filters may be selectively enabled using a group membership similar to light linking and ray tracing subsets. By default, a light filter belongs to a global group and is enabled everywhere. If a filter expresses that it belongs to a named group, it will instead be disabled by default and enabled for only those geometric primitives that subscribe to the group. Light filter groups are specified through a parameter to the light filter and reported to the renderer through the light filter `GetProperty()` method (see below). For example:

```
LightFilter "PxrBlocker" "blocker" "string linkingGroups" ["blockers"]
...
Attribute "lightfilter" "string subset" ["blockers"]
Sphere ...
```

Will turn on all light filters in group "blockers" for all lights shining on the sphere. By convention, all of the filters shipped with RenderMan use "linkingGroups" as the name of the parameter. Both the "linkingGroups" and "lightfilter:subset" strings may be comma-separated lists of group names. Group name matching is not scoped to a light. Any filter in a matching group on any light active on a given geometric primitive will be enabled.

Implementing the RixLightFilter interface

Light filters implement the RixLightFilter interface found in RixLightFilter.h. RixLightFilter 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 ownlLight filter, you can #include "RixLightFilter.h" and make sure your light filter class implements the required methods inherited from the RixShadingPlugin interface: Init(), Finalize(), Synchronize(), GetParamTable(), and CreateInstanceData(). Additionally, your light filter subclass must implement the three methods of the RixLightFilter interface, Filter(), GetProperty(), and GetRadianceModifier().

Generally, there is one shading plugin instance of a RixLightFilter per bound RiLightFilter (RIB) request. This instance may be active in multiple threads simultaneously.

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

```
RIX_LIGHTFILTERPLUGINCREATE
{
    return new MyLightFilter();
}

RIX_LIGHTFILTERPLUGINDESTROY
{
    delete ((MyLightFilter*)filter);
}
```

RixLightFilter::Filter()

Filter is where the work is done.

```
virtual void Filter(
        RixLightFilterContext const *lfCtx,
    RtConstPointer instanceData,
    int const numSamples,
    int const * shadingCtxIndex,
    RtVector3 const * toLight,
    float const * dist,
    float const * lightPdfIllum,
    RixBXLobeWeights *contribution);
```

Like pattern plugins, light filters are supplied a context. The `RixLightFilterContext` is a subset of the `RixShadingContext` and does not support full pattern generation (`EvalParam()` is not supported). That capability is under consideration for future releases.

Light filtering happens after lighting services has picked which lights will be sampled for each point in a shading context. The `RixLightFilterContext` member variable `numPts` is the number of points in the underlying shading context. Light filters are called on a light-by-light basis, and not all of the points in a shading context will be getting a sample from a given light. Some may get zero samples, others may get multiple samples. For this reason, the Filter call is sample-centric: it operates over the samples generated for a given light.

The `shadingCtxIndex` maps a particular sample back to the point on the shading context for which it was generated.

The three lighting arrays `toLight`, `dist`, and `lightPdfIllum`, are `numSamples` in length. These are the vector to the light, the distance from the point on the shading context to the sample point on the light, and the pdf of the sample on the light.

The input/output array contribution contains the lighting contribution distributed into some number of diffuse and specular lobes. For each lobe there are `numSamples` entries. For instance, to modulate the lighting by a filter that passes only the red channel:

```
for(int j = 0; j < contribution->GetNumSpecularLobes(); ++j) {
    for(int i = 0; i < numSamples; ++i) {
       contribution->GetSpecularLobe(j)[i] *= RtColorRGB(1.0,0.0f,0.0f);
    }
}

for(int j = 0; j < contribution->GetNumDiffuseLobes(); ++j) {
    for(int i = 0; i < numSamples; ++i) {
       contribution->GetDiffuseLobe(j)[i] *= RtColorRGB(1.0,0.0f,0.0f);
    }
}
```

RixLightFilter::GetProperty()

```
RixSCDetail PxrGobo::GetProperty(
    RtConstPointer instanceData, LightFilterProperty property, void const** result) const
{
    myData* data = (myData*)instanceData;

    if (RixLightFilter::k_LinkingGroups == property)
    {
        (*result) = &data->m_linkingGroups;
        return k_RixSCUniform;
    }
    return k_RixSCInvalidDetail;
}
```

`GetProperty()` is an extensible API through which the renderer may query the plugin. The list of valid queries are enumerated in `LightFilterProperty`. If a plugin can answer the query, it fills in `result` and returns `k_RixSCUniform`. Otherwise `GetProperty` should return `k_RixSCInvalidDetail`. The example given shows how the plugin returns the unique string for the linking groups to which the light filter belongs.

RixLightFilter::GetRadianceModifier()

```
bool PxrMyLightFilter::GetRadianceModifier(
    FilterRadianceModifierProperty property,
    RixLightFilterContext const* lfCtx,
    RtConstPointer instanceData,
    float* result) const
{
    *result = m_attenuation;
    return true;
}
```

`GetRadianceModifier()` returns a single float representing how much the filter modulates the light emitted from a light source. Not all filters attenuate or amplify the light signal; those plugins would return false and not modify `result`.

Instance Data

For a detailed discussion of instance data, see lightweight instancing services. When the CreateInstanceData routine is called, the plugin has access to the parameter list of the light filter and can create arbitrary data that is stored by the renderer and supplied as the instanceData pointer during filtering. This should be uniform data that can be accessed by multiple threads simultaneously. Pre-processing the parameter list is an important performance optimization. A light filter will be invoked for every surface-to-light interaction, and in a production-level shot there can be billions of these events.

As an example, below is the CreateInstanceData method for PxrLightFilterCombiner. Like bxdfs, light filters can be referenced in arguments to other light filters. To run multiple light filters, call EvalParam to get pointers to the upstream light filter and it's instance. These are stored and used later during filtering.

```
struct myData
{
    int arrayLen;
    RixLightFilter** filters;
    RtConstPointer* instances;
};

static void releaseInstanceData(RtPointer data)
{
    myData* md = (myData*) data;
    delete[] md->filters;
    delete[] md->instances;
    delete md;
}

int PxrLightFilterCombiner::CreateInstanceData(
    RixContext &ctx,
    char const *handle,
    RixParameterList const *plist,
    RixShadingPlugin::InstanceData *idata)
{
    RixSCType typ;
    bool isconnected;
    int arraylen;
    plist->GetParamInfo(k_mult, &typ, &isconnected, &arraylen);

    myData* mydata = new myData;
    mydata->arrayLen = arraylen;
    mydata->filters = new RixLightFilter* [arraylen];
    mydata->instances = new RtConstPointer [arraylen];

    for (int i=0; i<arraylen; ++i)
        plist->EvalParam(k_mult, i, &mydata->filters[i], &mydata->instances[i]);

    idata->data = (void *) mydata;
    idata->freefunc = releaseInstanceData;
    return 0;
}
```

Running multiple filters

Like bxdfs, light filters can be parameters to other light filters. The last filter given before an Light call (the root filter in a tree of filters) is responsible for delegating to the filters in its parameter list. Furthermore, light filters can be disabled on a gprim basis. To respect this, it is the responsibility of the filter to call `RixLightFilterContext::IsEnabled()`. If the upstream filter is not enabled, don't run it. `IsEnabled()` also returns a pointer to the upstream filter's `instanceData`, which should be passed to its `Filter()` method.

```
void PxrLightFilterCombiner::Filter(
    RixLightFilterContext const *lfCtx,
    RtConstPointer instanceData,
    int const numSamples,
    int const * shadingCtxIndex,
    RtVector3 const * toLight,
    float const * dist,
    float const * lightPdfIllum,
    RixBXLobeWeights *contribution)
{
  myData const * mydata = (myData const *) instanceData;

  for (int i=0; i<mydata->arrayLen; ++i) {
      RtConstPointer nextInstanceData;
      if (lfCtx->IsEnabled(mydata->instances[i], &nextInstanceData))
      {
          mydata->filters[i]->Filter(
              lfCtx, nextInstanceData,
              numSamples, shadingCtxIndex,
              toLight, dist, lightPdfIllum, contribution);
      }
  }
}
```