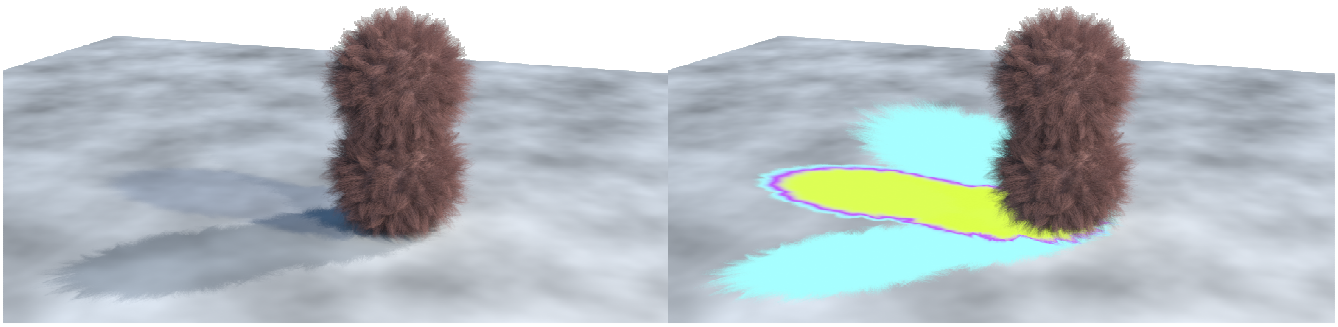
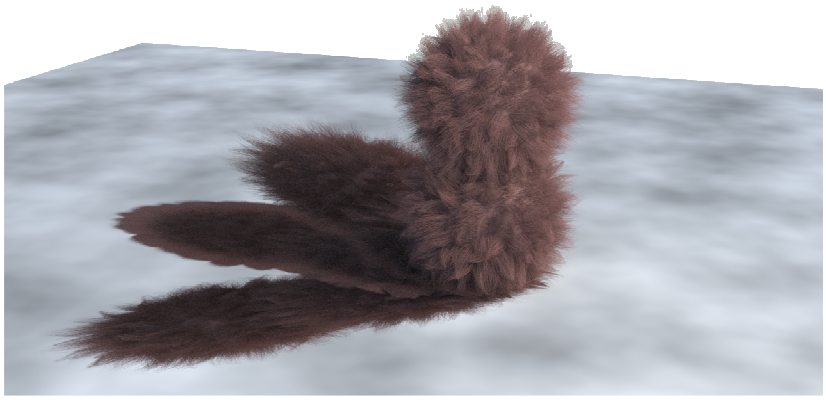
Light / Ray-traced Plug-in

This short tutorial will cover the minimum steps required to use main ray-traced tools. We will highlight use of direct light and gradient interface.

The purpose of this Ray-traced plug-in is to modify direct shadows color of the scene.





Headers

The first part of your plug-in source file will include the required headers. There are two types of headers you will typically want to use:

```
#include <lxsdk/lx_shade.hpp>
#include <lxsdk/lx_vector.hpp>
#include <lxsdk/lx_package.hpp>
#include <lxsdk/lx_action.hpp>
#include <lxsdk/lx_log.hpp>
#include <lxsdk/lx_raycast.hpp>
#include <lxsdk/lxu_math.hpp>
#include <lxsdk/lx_envelope.hpp>
```

The server Class

The heart of any plug-in is a C++ class that you write to implement a server interface. The interface is the set of standard methods that nexus uses to integrate your server into the application. This is done by inheriting from multiple *implementation* classes. In this case we're going to inherit from the *CustomMaterial* implementation.

```
class TutoRay : public CLxImpl_CustomMaterial
{

	public:
	…

};
```

The CustomMaterial super-class defines many methods. The cmt prefix is unique to the CustomMaterial implementation class, and allows multiple implementations with the same or similar methods to be inherited by the same server.

```
int		cmt_Flags() LXx_OVERRIDE;
LxResult	cmt_SetupChannels(ILxUnknownID addChan) LXx_OVERRIDE;
LxResult	cmt_LinkChannels(ILxUnknownID eval, ILxUnknownID item) LXx_OVERRIDE;
LxResult	cmt_ReadChannels(ILxUnknownID attr, void **ppvData) LXx_OVERRIDE;
void		cmt_Cleanup(void *data) LXx_OVERRIDE;
void		cmt_MaterialEvaluate(ILxUnknownID vector, void *data) LXx_OVERRIDE;
void		cmt_ShaderEvaluate(ILxUnknownID vector, ILxUnknownID rayObj,
LXpShadeComponents		*sCmp, LXpShadeOutput *sOut, void *data) LXx_OVERRIDE;
```

The *override* keyword is optional but very useful. It declares to the compiler that you intend for your method to be derived from an identical method in a super-class. That means that if the method in the implementation class changes the compiler will throw an error. If you don't use the override keyword (or if your compiler doesn't support it -- it's not standard) then you get no error, but your method will never be called

Server Methods

Like a package, a server must define its channels. The *SetupChannels()* method is identical to the one used by packages. In this tutorial, we want a gradient color channel. It's better to define all channels names to reuse in other channels methods.

```
#define CHANs_GRADIENTCOLOR		"gradientColor"

LxResult
TutoRay::cmt_SetupChannels(
ILxUnknownID		addChan)
{
	CLxUser_AddChannel	ac(addChan);
	LXtVector		one = {1, 1, 1};

	ac.NewChannel(CHANs_GRADIENTCOLOR, LXsTYPE_COLOR1);
	ac.SetVector(LXsCHANVEC_RGB);
	ac.SetDefaultVec(one);
	ac.SetGradient(LXsTYPE_COLOR1);

	return LXe_OK;
}
```

The plug-in read its channel values in a modifier context. That means that it has to declare the channels it wants to read using the Evaluation Interface. This is also a good place to cache the offsets for any vector packets that it wants to read. Offsets are previously declared in server class.

```
LxResult
TutoRay::cmt_LinkChannels(
ILxUnknownID          eval,
ILxUnknownID          item)
{
       CLxUser_Evaluation    ev(eval);

       int                   i = 0;
       idx[i++] = ev.AddChan(item, CHANs_GRADIENTCOLOR".R");
       idx[i++] = ev.AddChan(item, CHANs_GRADIENTCOLOR".G");
       idx[i++] = ev.AddChan(item, CHANs_GRADIENTCOLOR".B");

       ray_offset = pkt_service.GetOffset(LXsCATEGORY_SAMPLE, LXsP_SAMPLE_RAY);
       pos_offset = pkt_service.GetOffset(LXsCATEGORY_SAMPLE, LXsP_SAMPLE_POSITION);
       lgt_offset = pkt_service.GetOffset(LXsCATEGORY_SAMPLE, LXsP_LIGHTING);

       return LXe_OK;
}
```

Linking channels occurs once as items are added or their relationships are changed. Reading channels then happens any time channel values change. For example changing time may cause channels to be read again if they are animated. Values are read from an Attributes Interface using the index cached when linking channels. The values are then stored in an allocated data structure which is returned from the method.

```
LxResult
TutoRay::cmt_ReadChannels(
ILxUnknownID          attr,
void                 **ppvData)
{
       CLxUser_Attributes    at(attr);
       RendData*                rd = new RendData;

       int                   i = 0;
       for (int j = 0; j <3; ++j)
              at.ObjectRO(idx[i++], grad_filt[j]);

       ppvData[0] = rd;
       return LXe_OK;
}
```

ShaderEvaluate's the main method of this plug-in, is called for each ray who intersect the CustomMaterial, so it's intended to be as fast as possible.

The private data struct containing channel values is passed as data. The plug-in can read common input packets from the vector given their offset, and the results are written to the texture output packet, already extracted from the vector as an argument. Values may be written as scalar or vector.

Note also that this is a *void* function. Anything that can fail, such as allocations, should already have been done as part of reading channels. The texture evaluation is assumed to succeed and there is no allowance for failure.

```
void
TutoRay::cmt_ShaderEvaluate(
ILxUnknownID            vector,
ILxUnknownID            rayObj,
LXpShadeComponents      *sCmp,
LXpShadeOutput          *sOut,
void                    *data)
{
        const RendData          *rd = (RendData *)data;
        const LXpSamplePosition *sPos = (LXpSamplePosition*)pkt_service.FastPacket
        (vector, pos_offset);

        CLxLoc_Raycast          raycast;
        CLxLoc_Lighting         lighting;
        LXpLightSource          lSrc;
        LXtFVector              col; LXx_VCLR(col);
        int                     lightSamples;
        const int               flags = LXfRAY_SCOPE_POLYGONS | LXfRAY_TYPE_CAMERA;

        raycast.set(rayObj);
        pkt_service.PacketObject(vector, lgt_offset, lighting);
        lighting.LightSourceCount(vector, &lightSamples);

        …
}
```

The ID rayObj is the raycast object, ShadeComponents has the information from the old shader.
ShadeOuput contains the color and many information of the output shade.
When lighting's initialized we can start the round of all lights. For each ray, we try to see if the light is visible.

```
        double t = 0;
        for (int j = 0; j < lightSamples; ++j)
        {
                lighting.LightSourceByIndex(vector, j, &lSrc);
                …
        }
```

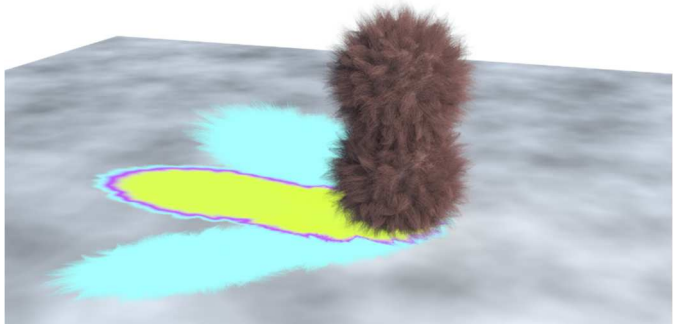
LSrc takes the information of current light source, so with the position of first ray and the direction of light we can check if we are in a shadow. We start by creating a new vector ray.

```
                ILxUnknownID vector1 = raycast.RayPush(vector);
                if (vector1)
                {
                        LXtFVector dir;
                        LXx_VCPY(dir, lSrc.dir);
                        LXx_VNEG(dir);
                        if (raycast.Raytrace(vector1, sPos->wPos, dir, flags) >= 0.)
                                t += 1 / (double)lightSamples;
                }
                raycast.RayPop(vector);
```

Raytrace function give the distance if there is an intersection otherwise it returns negative value.
When we are in the first case, we are in shadow so we increment the value t to give a normalize value dependent on the number of shadow.

```
        for (int j = 0; j < lightSamples; ++j)
        {
                …
        }
        if (t)
        {
                sOut->color[0] = (float)grad_filt[0].Evaluate(t);
                sOut->color[1] = (float)grad_filt[1].Evaluate(t);
                sOut->color[2] = (float)grad_filt[2].Evaluate(t);
        }
}
```

grad_filt is our gradient color, you can define a color in user interface directly. If t is null no color is assigned and the CustomMaterial keep the old shader value.



Now we want to change the shadows by the color of object who is interested, it's create a projection.
So when he has an intersection, we collect the color of this object.

```
if (raycast.Raytrace(vector1, sPos->wPos, dir, flags) >= 0.)
{
    LXpSampleRay     *sRay1 = (LXpSampleRay*)pkt_service.FastPacket(vector1,
ray_offset);
    LXx_VCPY(sOut->color, sRay1->color);
}
```

We ask values of the new SampleRay for get object color and it's all! In our example two are recover are covered with fur so we need to change the flag of rayTrace. We add shading evaluation but be careful this flag also adds considerable time, 2 minutes without and 22 minutes with for 3 lights.

```
const int                   flags = LXfRAY_SCOPE_POLYGONS | LXfRAY_TYPE_SHADOW |
LXfRAY_EVAL_SHADING;
```

Server Tags

Server objects also support a tags interface. Server Tags are string/string pairs that define the attributes of servers. In the case of savers this is the type of object that's saved, the file extension of the format, and the user name. These can be added easily by defining a descInfo array in the server class:

```
static LXtTagInfoDesc		descInfo[];
```

And declaring the contents of the array statically in the module. The array is terminated with a null name:

```
LXtTagInfoDesc		TutoRay::descInfo[] = {
	{ LXsSRV_USERNAME, "TutoRay Material" },
	{ LXsSRV_LOGSUBSYSTEM, "comp-shader" },
	{ 0 }
};
```

Initialization

Plug-ins need to declare the servers they export, and the interfaces those servers support. This is done with the *initialize()* function which is called from the *common* lib code as the plug-in is loaded. This is largely boilerplate, but with the classes and interfaces varying depending on the plug-in. The first interface defined is also the type of the server. Note that the name of the server is an internal name string, so it should start with lowercase and cannot contain spaces.

```
void
initialize()
{
	CLxGenericPolymorph*	srv1 = new CLxPolymorph<TutoRay>;

	srv1->AddInterface(new CLxIfc_CustomMaterial<TutoRay>);
	srv1->AddInterface(new CLxIfc_StaticDesc<TutoRay>);
	lx::AddServer(SRVs_TutoRay, srv1);
}
```

Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

  <atom type="CommandHelp">
    <hash type="Item" key="material.TutoRay@en_US">
      <atom type="UserName">TutoRay Material</atom>
    </hash>
  </atom>

  <atom type="Filters">
    <!-- Filter Categories -->
    <hash type="Preset" key="TutoRay:filterPreset">
      <atom type="Name">TutoRay</atom>
      <atom type="Description"></atom>
      <atom type="Category">custMaterials:filterCat</atom>
      <atom type="Enable">1</atom>
      <list type="Node">1 .group 0 &quot;&quot;</list>
      <list type="Node">1 itemtype 0 1 &quot;material.TutoRay&quot;</list>
      <list type="Node">-1 .endgroup </list>
    </hash>
  </atom>

  <atom type="Attributes">
    <hash type="Sheet" key="TutoRay:sheet">
      <atom type="Label">TutoRay</atom>
      <atom type="Filter">TutoRay:filterPreset</atom>
      <atom type="Group">itemprops/general</atom>

      <list type="Control" val="cmd layout.createOrClose cookie:GradientEditor
layout:{Gradient           Editor Palette_layout} title:{@frame.gradient@@@@}
width:1080 height:330 persistent:1 open:?">
        <atom type="Label">Edit Gradient</atom>
      </list>
      <list type="Control" val="vt geembed">
        <atom type="ViewportAlwaysUsesMinSize">1</atom>
        <hash type="ViewportArgument"
key="channel">material.TutoRay$gradientColor</hash>
        <atom type="Label">Gradient H</atom>
      </list>

      <hash type="InCategory" key="itemprops:general#head">
        <atom type="Ordinal">39.3.5</atom>
      </hash>
    </hash>
  </atom>

  <!-- Texture Layer Categories -->
  <atom type="Categories">
    <hash type="Category" key="itemtype:textureLayer">
      <hash type="C" key="material.TutoRay">custMaterials</hash>
    </hash>
  </atom>

  <!-- Texture Layer Category Message Tables -->
  <atom type="Messages">
    <hash type="Table" key="material.TutoRay.en_US">
      <hash type="T" key="1">TutoRay Material</hash>
    </hash>
```

```
  </atom>

</configuration>
```

Testing

To test your new plug-in you can just copy-paste the *.lx in extra folder and *.cfg in resrc folder. You may to restart.