Writing Patterns

Introduction

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

A `RixPattern` plugin is used to connect textures and procedurally generated patterns to `RiBxdf` parameters, or to other patterns to create a shading graph. There are numerous pattern plugins included with the RenderMan software, but if none of the included plugins generate the pattern you want, then this guide will help you write your own pattern plugin. Source code for many of the RenderMan pattern plugins can be found in the `PixarRenderMan-Examples-VERSION/plugins/pattern/` directory which is installed as part of the separate examples package.

For pattern generation using the Open Shading Language (OSL), see the **PxrOSL** plugin documentation on Working with PxrOSL.

Implementing the RixPattern Interface

`RixPattern.h` defines the interface that all pattern plugins must implement. `RixPattern` is a subclass of `RixShadingPlugin`, and therefore shares the same initialization, synchronization, and parameter table logic as other shading plugins. Because a `RixPattern` is expected to be a lightweight object that may be created many times over the course of the render, `RixPattern` is expected to take advantage of the lightweight instancing services p rovided by `RixShadingPlugin` . Therefore to start developing your own pattern, you can `#include "RixPattern.h"` and make sure your pattern class implements the required methods inherited from the `RixShadingPlugin` interface: `Init()`, `Finalize()`, `Synchronize()`, `GetParamTable()`, and `CreateInstanceData()`.

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

```
RIX_PATTERNCREATE
{
    return new MyPattern();
}
RIX_PATTERNDESTROY
{
    delete ((MyPattern*)pattern);
}
```

Computing Pattern Output

`ComputeOutputParams()` is the heart of a pattern plugin: it evaluates the input parameters, and computes the pattern output. It is called once per graph execution, and all outputs must be computed during this single invocation. The number and type of outputs should match the number and type of outputs declared in the parameter table. The domain of evaluation of this function is a shading context, which is of type `RixShadingContext`, defined in `RixShading.h`.

To read an input value, use the `RixShadingContext::EvalParam()` method. The desired input parameter to the pattern is selected by an integer `paramId`, which is the ordinal position of the parameter in the parameter table. Patterns are expected to know the `paramId`, the type of the associated parameter, and are expected to pass a pointer to a pointer of the appropriate type. As such, it is suggested that a private parameter enumeration is used to keep track of the order that the parameters are created in the parameter table. For more information, please consult the documentation for `RixShadingContext::EvalParam()` and `RixShadingPlugin::GetParamTable()`.

After reading input values, output values need to be set up. First, memory buffers for the requested outputs should be allocated using the `RixShadingContext` memory allocation services. These buffers should then be bound to the requested `OutputSpec outputs` parameter passed to `ComputeOutputParams()`, and the type and detail information about those outputs filled in as well. This information should match the declarations from the parameter table. The following code is boilerplate that can be used: it reads the plugin's parameter table, loops through and allocates the appropriate buffers, and sets the detail and type assuming that the output is always a varying color or float (typical of most patterns).

```
    // Find the number of outputs
    RixSCParamInfo const* paramTable = GetParamTable();
    int numOutputs = -1;
    while (paramTable[++numOutputs].access == k_RixSCOutput) {}

    // Allocate and bind our outputs
    RixShadingContext::Allocator pool(sctx);
    OutputSpec* out = pool.AllocForPattern<OutputSpec>(numOutputs);
    *outputs = out;
    *noutputs = numOutputs;

    // looping through the different output ids
    for (int i = 0; i < numOutputs; ++i)
    {
        out[i].paramId = i;
        out[i].detail = k_RixSCInvalidDetail;
        out[i].value = NULL;
        type = paramTable[i].type; // we know this

        sctx->GetParamInfo(i, &type, &cinfo);
        if(cinfo == k_RixSCNetworkValue)
        {
            if( type == k_RixSCColor )
            {
                out[i].detail = k_RixSCVarying;
                out[i].value = pool.AllocForPattern<RtColorRGB>(sctx->numPts);
            }
            else if( type == k_RixSCFloat )
            {
                out[i].detail = k_RixSCVarying;
                out[i].value = pool.AllocForPattern<RtFloat>(sctx->numPts);
            }
        }
    }
```

Finally, the pattern can now actually compute the values that go into the output buffers. This is typically done by using the inputs and looping through the number of shaded points `RixShadingContext::numPts` to compute some values that are stored in the allocated output buffers.

```
RtColorRGB* outColor = (RtColorRGB*) out[k_resultRGB].value;
for (int i=0; i<sctx->numPts; i++)
{
    // Compute some output values based on your input. Here we assume
    // outColor is the memory buffer allocated for an output parameter,
    // and inputColor and inputFloat are two inputs that were returned from
    // EvalParam.
    if (style == 1)
    {
        outColor[i] = inputColor[i] * inputFloat[i];
    }
}
```

In the simple example above, outColor is assigned the buffer that was allocated corresponding to the private enumeration value `k_resultRGB`, which matches the position of that output in the parameter table. (So long as the output parameters are at the **beginning** of the parameter table, reuse of this enumeration is valid for this purpose.) We assume the `style` variable was a uniform `RtInt` input value, so there is only one value for all the points in the shading context. Meanwhile, the `inputColor` and `inputFloat` variable were varying instead of uniform, so they are pointers to an array of `RtColorRGB` values and array of `RtFloat` values respectively, one for each shaded point in the shading context.

The `ComputeOutputParams()` method should return 0 if no error occurred while calculating the output, otherwise it should return a non-zero integer value.

Testing Your Pattern Plugin

After you have implemented the code for your pattern plugin, you can build it using the commands listed in the Compiling Plugins page. The next step is to test your plugin. To test it, you'll need to make sure prman can find your plugin in the **standardrixpluginpath** list of directories, which is defined in $RMANTREE/etc/rendermn.ini as:

```
/standardrixpluginpath          ${RMANTREE}/lib/RIS/pattern:${RMANTREE}/lib/RIS/bxdf:${RMANTREE}/lib/RIS
/integrator:${RMANTREE}/lib/RIS/projection
```

You can add a rendermn.ini file to your HOME directory and modify the **standardrixpluginpath** value to contain the directory where your pattern plugin is located.

Then you can try to render this RIB file after you have replaced **"PxrCustomPattern"** with the name of your pattern plugin and connect your pattern's output parameter to one of the input parameters of the **PxrDiffuse** Bxdf:

```
Display "patternTest" "framebuffer" "rgba"
Quantize "rgba" 255 0 255 0
Format 128 128 1
Projection "perspective" "fov" [45]
Hider "raytrace" "string integrationmode" ["path"]
Integrator "PxrPathTracer" "integrator"
WorldBegin
    AttributeBegin
        Attribute "identifier" "name" ["sphere1"]
        Translate 0 0 2.75
        Pattern "PxrCustomPattern" "customPattern"
        Bxdf "PxrDiffuse" "smooth"
            "reference color diffuseColor" "customPattern:outColor"
        Sphere 1.0 -1.0 1.0 360.0
    AttributeEnd
WorldEnd
```

Texture Baking

RenderMan can optionally bake pattern outputs to 2D or 3D textures by evaluating those patterns over an output manifold. Pattern plug-ins that wish to bake outputs should provide custom implementations of the RixPattern::Bake2dOutput or RixPattern::Bake3dOutput methods that return true. When in bake mode, RenderMan queries these methods to describe the output manifold and to initialize display drivers. For 2d atlas/UDIM outputs that set RixPattern::Bake2dSpec::atlas to true, RenderMan will query RixPattern::Bake2dOutput once for each UV tile.

It is possible to write a generalized baking node that bakes the output of arbitrary upstream pattern graphs. For example, see PxrBakeTexture and PxrBakePointCloud pattern plug-ins:

```
Hider "bake"
Format 512 512 1
Display "render.exr" "openexr" "rgba"
Projection "perspective" "fov" [30]
Translate 0 0 5
WorldBegin
    AttributeBegin
        Pattern "PxrFractal" "pattern"
        Pattern "PxrBakeTexture" "baked" "reference color inputRGB" ["pattern:resultRGB"]
          "string filename" ["bake.tif"] "string display" ["tiff"]
          "string primVar" ["st"] "int resolutionX" [512] "int resolutionY" [512]
        Bxdf "PxrDiffuse" "default" "reference color diffuseColor" ["baked:resultRGB"]
        Sphere 1 -1 1 360 "varying float[2] st" [0 0 1 0 0 1 1 1]
    AttributeEnd
WorldEnd
```