RixShadingPlugin

- Introduction
  - Plugin vs. Plugin instance vs. Closures
  - Plugin Instance C++ representation
    - void RixShadingPlugin::CreateInstanceData(InstanceData*)
    - XXX* RixYYYPlugin()::CreateXXX()
  - Closures
  - Parameter Table
    - Dynamic Parameters
- Initialization and Synchronization
  - Plugin initialization
  - Plugin instance initialization
  - Plugin Synchronization
  - Plugin instance synchronization
  - Closures synchronization
  - Interactive rendering sessions
    - Subtleties
    - Known limitations
- Misc
  - Overview
  - Examples
    - PxrDiffuse
    - PxrDirt
  - Installation
  - Creating an .Args File

Introduction

`RixShadingPlugin` is a base class characterizing the requirements of a RenderMan renderer from shading plugins: `RixBxdfFactory`, `RixDisplacementFactory`, `RixDisplayFilter`, `RixIntegratorFactory`, `RixLightFactory`, `RixLightFilter`, `RixPattern`, `RixProjectionFactory`, and `RixSampleFilter`. These are plugins that implement services for the renderer.

This class provides entry points that are executed with a mix of various frequencies:

- once per rendering session
- once per render (in an interactive session, there can be multiple renders)
- once per batch of points to be shaded (i.e. `RixShadingContext`)
- multiple times per batch of points to be shaded

and various granularity:

- once per plugin
- once per *instance* of the shading plugin (as defined in the next section)
- multiple times per *instance* of the shading plugin

Plugin vs. Plugin instance vs. Closures

Here, the term *instance* is unfortunately overloaded, and it is important to differentiate between: an *instance of a shading plugin* as defined by the unique set of parameters given to the material definition, and a *C++ instance* which involves an actual memory allocation, construction, and destruction of an object.

Each use of a shading plugin with a different set of parameters will be considered as separate *plugin instance*. A render using two `PxrTexture` patterns (each using a different texture file parameter), will trigger the initialization of the (unique) `PxrTexture` plugin, and the initialization of two *instances* of the `PxrTexture` plugin (independently of the internal C++ representation chosen).

Finally, some plugin types will create short-lived objects referred to as *closures*. These transient objects are created for a given plugin instance, for a given batch of points. They allow the pre-computation of data that can be re-used during multiple computations associated with the the given batch of points.

Plugin Instance C++ representation

Depending on the plugin type, *plugin instances* may correspond to different internal C++ structures.

void RixShadingPlugin::CreateInstanceData(InstanceData*)

Some shading plugin types can create private instance data using the `CreateInstanceData()` method. Instance data would typically be computed from the unique evocation parameters, supplied to `CreateInstanceData()` via the `RixParameterList` class. This occurs when the shading plugin instance is created for the first time from those parameters (e.g. at the time a material definition is created (i.e. very early on in a render)).

Using these parameters, plugins may *bake* a cached understanding of their behavior, requirements, or even precomputed results into a private representation that the renderer will automatically track with the instance and supply back to the plugin methods. This allows the plugin to cache and therefore avoid repeated computations when processing each batch of points. In this case, the instance data is essentially a C++ structure representing one instance of the plugin.

If the shading plugin does create instance data, it should be stored in `InstanceData::data`. If the data requires non-trivial deletion, `InstanceData::freefunc` should be set to a function that the renderer will invoke when the plugin instance will no longer be needed. A trivial implementation of `CreateInstanceData()` produces no instance data and returns without modifying any member of the provided `InstanceData` structure.

Any instance data that is created will be automatically returned to the shading plugin methods by the renderer when the compute methods are invoked. The implementation of the compute methods is now free to use this instance data to reduce the cost of processing the current batch of points.

For example, the `RixPattern` plugin type uses this mechanism:

- `RixPattern::CreateInstanceData(..., RixParameterList const*, InstanceData*)`
- `RixPattern::ComputeOutputParams(RixShadingContext const*, ..., RtPointer data)`

The other plugin types using this representation are: `RixBxdf`, `RixDisplacement`, `RixDisplayFilter`, `RixLightFilter` and `RixSampleFilter`. `RixBxdf` and `RixDisplacement` additionally make use of closures.

> ⚠ `RixShadingPlugin::CreateInstanceData()` may be called in multiple threads, and so its implementation should be re-entrant and thread-safe.

XXX* RixYYYPlugin()::CreateXXX()

Some other shading plugin types use a different mechanism instead of `CreateInstanceData()`. Generally, the `RixShadingPlugin` sub-class associated with the plugin type will expose a `CreateXXX()` method, used to build a new C++ object. In this case, the C++ object is a direct representation of the *plugin instance*. Generally, the render will then call methods on the newly minted C++ object, instead of using `RixShadingPlugin` methods and passing an instance data pointer.

For example, the RixProjection plugin type uses this mechanism:

- `RixProjection* RixProjectionFactory::CreateProjection(..., RixParameterList const*)`
- `RixProjection::Project(RixProjectionContext&)`

The other plugin types using this representation are: `RixIntegrator` and `RixLightFactory`.

Closures

Generally, the `RixShadingPlugin` subclass (e.g. `RixPattern`) will expose only a few compute methods (e.g. `RixPattern::ComputeOutputParams()`) that is executed once per batch of points (`RixShadingContext`), per plugin instance. When a specific class is used to represent the plugin instance (e.g. `RixProjection`), it will usually expose the compute methods itself (e.g. `RixProjection::Project()`).

However some plugins types require particular computations to happen multiple times for a given plugin instance, for a given batch of points. In this case, a closure object will be created. Note that the closures are built after the plugin instance has been created. Since any instance data (created by `CreateInstanceData()`) is be automatically returned to the shading plugin methods by the renderer, the plugin is free to use this instance data to reduce the cost of creating the associated closure.

Bxdf plugins are the best example of this:

- `RixBxdfFactory::CreateInstanceData()` fill the `InstanceData::data` with a representation of the bxdf instance, and this will provided to the closure creation method below.
- `RixBxdf RixBxdfFactory::BeginScatter(RixShadingContext*, ..., RtPointer data)` returns a closure for the provided batch of points and plugin instance
- `RixBxdf::GenerateSamples()` and `RixBxdf::EvaluateSamples()` may be called repeatedly on this closure. These methods are able to share and re-use any precomputation that may have happened in the `RixBxdf` closure constructor.

Parameter Table

All shading plugins are expected to return a description of their input and output parameters via the `GetParamTable()` method. This returns a pointer to an array of `RixSCParamInfo`, containing one entry for each input and output parameter, as well as an extra empty entry to mark the end of the table. This parameter table is used by the renderer to ensure proper type checking and validate the connections of upstream and downstream nodes. As such, each entry in the table should set a name, a type (`RixSCType` enumeration), detail (varying vs uniform, `RixSCDetail` enumeration), and access (input vs output, `RixSCAccess` enumeration). These declarations also need to be kept in sync with the associated .args file.

For an example of usage, consider a pattern plugin which returns a color. The *resultC* output parameter is a color, so it is defined in the parameter table as:

```
RixSCParamInfo("resultC", k_RixSCColor, k_RixSCOutput)
```

A float input parameter named *density* can be defined as:

```
RixSCParamInfo("density", k_RixSCFloat)
```

While a float[16] input parameter named *placementMatrix* can be defined as:

```
    RixSCParamInfo("placementMatrix", k_RixSCFloat, k_RixSCInput, 16)
```

The full implementation of `GetParamTable()` for this plugin would look something like this:

```
RixSCParamInfo const *
MyPattern::GetParamTable()
{
    static RixSCParamInfo s_ptable[] =
    {
        // outputs
        RixSCParamInfo("resultC", k_RixSCColor, k_RixSCOutput),
        // inputs
        RixSCParamInfo("density", k_RixSCFloat),
        RixSCParamInfo("placementMatrix", k_RixSCFloat, k_RixSCInput, 16),
        RixSCParamInfo(), // end of table
    };
    return &s_ptable[0];
}
```

The ordinal position of a parameter in the parameter table is the integer `paramId` used to evaluate parameter inputs using the `RixShadingContext::EvalParam` method. Because these need to be kept in sync, it is recommended that you create a parameter enumeration (a private `enum` type) to keep track of the order that your parameters were created in the table. The enumeration can be used later on when calling `RixShadingContext::EvalParam` in the body of the shader. Following the three parameter table entries above:

```
enum paramId
{
    k_resultC=0, // output
    k_density,
    k_placementMatrix,
    k_numParams
};
```


In order to facilitate the reuse of the same parameter enumeration for pattern output computation, it is highly recommended that all outputs be placed at the beginning of the parameter table.

Dynamic Parameters

A plugin can create its parameter table dynamically based on the parameters provided to each instance of the plugin. This dynamically created table is created using the `CreateInstanceData()` method, and should be saved in the `paramtable` member of the `InstanceData`. If the associated memory need to be freed, if should be taken care of in the `freefunc()` routine. Generally, static interfaces should be preferred over dynamic interfaces due to their extra memory expense. If the `paramtable` member remains null, all instances will share the parameter table returned by `GetParamTable()`. In order to prevent the renderer from filtering out dynamic parameters as bad inputs, a plugin that is using a dynamically created table should have a `k_RixSCAnyType` entry in its plugin parameter table.

Initialization and Synchronization

Plugin initialization

In order to initialize the plugin, the renderer will call `Init()` once. Even if the plugin is evoked multiple times during the render with different arguments, `Init()` will still be called only once during the lifetime of the render. The `RixContext` parameter can be used by the plugin to request any `RixInterfaces` services provided by the renderer. Any expensive memory allocations or operations that can be reused during the lifetime of the plugin can be performed in this routine. Upon successful initialization, this routine should return a zero status.

`Finalize()` is the companion to `Init()`, called at the end of rendering with the expectation that any data allocated within the `Init()` implementation will be released.

The `Init()` method will only ever be called once per plugin.

Plugin instance initialization

Depending on the paradigm used by the plugin type, `CreateInstanceData()` or `CreateXXX()` will be called once per plugin instance:

- CreateInstanceData() allows the plugin instance to create private data (stored in `InstanceData::data`), that will then be provided back to the various plugin methods the renderer will call during rendering.
- CreateXXX() returns a C++ object that should store its own representation of the plugin instance. The renderer will call methods on this object during rendering (instead of calling plugin methods).

The renderer provides the unique evocation parameters that triggered the creation of the plugin instance through the the `RixParameterList` class.

`RixParameterList` allows for the evaluation of the plugin instance parameters via the `EvalParam()` method. To aid in this, it allows for the querying via `RixParameterList::GetParamInfo()` of whether the parameters have been unset (and are therefore at their default value), set as a uniform value, or are part of a *network connection*, i.e. the parameter is computed by an upstream node in the shading graph. A network connection is understood to be a varying quantity, and its value cannot be evaluated at the time that `CreateInstanceData` is evoked; this is why `EvalParam()` will return `k_RixSCInvalidDetail` if the parameter is a network connection. Otherwise, `EvalParam()` can be used to get an understanding of the uniform, non-varying parameters that are passed to the shading instance, and these can be used to perform any precomputations as needed.

The `CreateInstanceData()` method will only ever be called once per plugin instance (a unique set of parameters).

Plugin Synchronization

The `Synchronize()` routine allows the plugin to respond to synchronization signals delivered by the renderer. This call may happen multiple times during a render session and/or during a given render.

The renderer may provide additional information to the plugin via the input parameter `RixParameterList`. These signals include:

- `k_RixSCRenderBegin`: The renderer is being initialized.
- `k_RixSCRenderEnd`: The renderer is about to end.
- `k_RixSCInstanceEdit`: Currently unused.
- `k_RixSCCancel`: Currently unused.
- `k_RixSCCheckpointRecover`: A signal that the renderer is about to restart rendering from a checkpoint. The parameter list will contain a single constant integer "increment" which contains the increment value from which the renderer will restart.
- `k_RixSCCheckpointWrite`: A signal that the renderer is about to write a checkpoint. The parameter list will contain two values: a constant integer "increment" indicating the increment value the renderer will write, and a constant string "reason" which contains one of three values: "checkpoint", "exiting", or "finished", indicating why the renderer is writing the checkpoint.
- `k_RixSCIncrementBarrier`: A signal that the rendering of an new increment is about to begin. This signal will only be received if the integrator has set `wantsIncrementBarrier` to true in the `RixIntegratorEnvironment`. The parameter list will contain a single constant integer "increment" which contains the increment value the renderer is about to render.

The `Synchronize()` method may be called multiples times during a rendering session, and in some cases during a render (for the 'increment barrier' message).

Plugin instance synchronization

Similarly to `Synchronize()`, the `SynchronizeInstanceData()` method allows the plugin instance to update its state when a render starts.

There are some subtle differences with `Synchronize()` though:

- there is no synchronization message, it is always assumed to be k_RixSCRenderBegin
- the plugin instance must explicitly subscribe to this mechanism, by appropriately setting InstanceData::synchronizeHints duringCreateInstanceData()

If the plugin type doesn't use `CreateInstanceData()` but `CreateXXX()`, the created objects will sometimes expose an `Edit()` or `Synchronize()` method.

The `SynchronizeInstancedata()` method will always be called before a new render starts.

Closures synchronization

Closures are transient objects with a very short lifetime, and are re-created with such a frequency that there is no need for synchronization mechanism.

Interactive rendering sessions

Because `Init()` and `CreateInstanceData()` are only called once during a rendering session, they are unable to capture edits that may have happened after the plugin instance has been initialized. In some cases, this includes edits that may have happened before the render starts, yielding counter-intuitive behaviors.

It is therefore strongly recommended for these two methods to only rely on data that was explicitly provided (e.g. the plugin instance parameter list). In particular, special care should be taken not to query options or anything related to the render state (displays, integrator environment, lpe-related quantities, etc...) for the following reasons:

- options may be edited after these methods have been called
- displays and render state will most likely be undefined when these methods are called, and may be subsequently edited

In order to support general edits, and to be future-proof, the `Synchronize()` and `SynchronizeInstanceData()` should be used instead to query options and render state.

Subtleties

- `RixProjection` (the type of objects created by `RixProjectionFactory::CreateProjection()` to represent projection plugin instances) doesn't expose a `Edit()` or `Synchronize()` method. Instead, `RixProjection::RenderBegin()` will be called, allowing for the plugin to set the `RixIntegratorEnvironment::deepMetric`. There is currently no way `RixProjection` plugins to opt-in / opt-out of the `RixProjection::RenderBegin()` call.
- `RixLight` (the type of objects created by `RixLightFactory::CreateLight()` to represent light plugin instances) currently uses a mix of `CreateInstanceData()` and `CreateLight()` paradigms. It is strongly recommended to keep `CreateInstanceData()` empty and put the majority of the implementation in `CreateLight()`.
- RixIntegrator ((the type of objects created by `RixIntegratorFactory::CreateIntegrator()` to represent integrator plugin instances) exposes `Synchronize()`. This method uses a slightly different signature, including a synchronization message, and it is currently not possible to opt-in/opt-out of the synchronization calls.

Known limitations

- `RixBxdf::GetInstanceHints()` is called after `CreateInstanceData()` but before `SynchronizeInstanceData()`. As a consequence, if a bxdf instance's hint depends on a global option or on the render state, edits will have no effect. This will lead to counter-intuitive behaviors, and it is strongly recommended for bxdf instance hints to only depend on the shader parameters provided to `CreateInstanceData()`
- `RixPattern::Bake2dOutput()` and `RixPattern::Bake3dOutput()` are called after `CreateInstanceData()` but before `SynchronizeInstanceData()`. The same limitations as the one described in the item above apply.
- `RixLight::Edit()` and `RixLight::SynchronizeInstanceData()` are currently not called.

Misc

Overview

The following table summarizes the structure of the various RixShadingPlugin, in relation to initialization and synchronization. All plugins use the `RixShadingPlugin::Init()` and `RixShadingPlugin::Finalize()` calls.

Plugin	RixShadingPlugin subclass	Instance representation	Generates a closure (per RixShadingContext)
bxdf	RixBxdfFactory	InstanceData::data	Yes, RixBxdf
displacement	RixDisplacementFactory	InstanceData::data	Yes, RixDisplacement
display filter	RixDisplayFilter	InstanceData::data	No
integrator	RixIntegratorFactory	RixIntegrator	No
light	RixLightFactory	RixLight	No
light filter	RixLightFilter	InstanceData::data	No
pattern	RixPattern	InstanceData::data	No
projection	RixProjectionFactory	RixProjection	No
sample filter	RixSampleFilter	InstanceData::data	No

Examples

PxrDiffuse

First, note that while the bxdf plugin is published and exposed as 'PxrDiffuse', it is comprised of two classes:

- `PxrDiffuseFactory`, that inherits from `RixBxdfFactory`, a subclass of `RixShadingPlugin`. This is the class that manages instance data and associated closures.
- `PxrDiffuse`, that inherits from `RixBxdf`. This is the type used by the bxdf closures.

As an example of usage of instance data, consider the `PxrDiffuse` bxdf. Although it is a fairly trivial bxdf, it does handle presence and opacity, and the renderer passes instance data to the `RixBxdfFactory::GetInstanceHints()` interface in order to get an understanding of the requirements for presence and opacity. In the following code, `PxrDiffuseFactory` checks its own presence parameter to see if it is a connection, knowing that its `Args` file only allows presence to be set to a default value (and therefore is trivially fully opaque) or is connected (and therefore requires the renderer to perform presence calculations). If it is connected, then it sets the instance data to be the same `InstanceHints` bitfield that is requested by the renderer from `RixBxdf::GetInstanceHints`.

```
plist->GetParamInfo(k_presence, &typ, &cnx1);
if(cnx1 == k_RixSCNetworkValue)
{
    if (cachePresence == 0)
    {
        req |= k_ComputesPresence;
    }
    else
    {
        req |= k_ComputesPresence | k_PresenceCanBeCached;
    }
}
```

PxrDirt

For a more complicated example of instance data usage, consider the `PxrDirt` pattern. Its instance data routine caches the values of many uniform parameters and reuses them in `PxrDirt::ComputeOutputParams()`, knowing that its .args file prohibits those parameters from being set to network connections.

```
Data *data = static_cast<Data*>(instanceData->data);

data->numSamples = 4;
data->distribution = k_distributionCosine;
data->cosineSpread = 1.0f;
data->falloff = 0.0f;
data->maxDistance = 0.0f;
data->direction = k_directionOutside;
data->raySpread = 1.0f;

params->EvalParam(k_numSamples, 0, &data->numSamples);
data->numSamples = RixMax(1, data->numSamples);
params->EvalParam(k_distribution, 0, &data->distribution);
params->EvalParam(k_cosineSpread, 0, &data->cosineSpread);
params->EvalParam(k_falloff, 0, &data->falloff);
params->EvalParam(k_maxDistance, 0, &data->maxDistance);
params->EvalParam(k_direction, 0, &data->direction);
```

Installation

RenderMan will search for shading plugins on demand, under the `rixplugin` searchpath. Custom shading plugins can be installed in a directory that can either be appended to the `/rixpluginpath` settings in Rendermn.ini; or the directory can be appended to the `rixplugin` search path which is emitted by the bridge.

Creating an .Args File

If you would like RenderMan for Maya or RenderMan for Katana to recognize your plugin and provide a user interface for changing input parameters and connecting output parameters to other nodes, then you will need to create an args file for your shading plugin. The args file defines the input and output parameters in XML so that tools like RMS or Katana can easily read them, discover their type, default values, and other information used while creating the user interface for the pattern node. Please consult the Args File Reference for more information.