RixShadingPlugin

Introduction

`RixShadingPlugin` is the base class for `RixBxdfFactory`, `RixDisplacementFactory`, `RixDisplayFilter`, `RixIntegrator`, `RixLightFilter`, `RixLightFactory`, `RixPattern`, `RixProjection`, and `RixSampleFilter`. These are plugins that implement services for the renderer.

Here, it is important to distinguish between two types of plugins: ones that need to create short-lived lightweight instances during the course of a render, and ones that do not. `RixBxdf`, `RixDisplacement`, and `RixLight` represent lightweight instances that may be created many times during the course of a single render, and therefore are not directly subclasses of `RixShadingPlugin`. Instead, instancees of those classes are returned by the appropriate Factory (e.g. `RixBxdfFactory`), with the Factory itself being the subclass of `RixShadingPlugin`.

`RixIntegrator`, `RixDisplayFilter`, `RixLightFilter`, `RixPattern`, `RixProjection`, and `RixSampleFilter` do not create many lightweight instances. As such, these classes are directly subclasses of `RixShadingPlugin`.

All `RixShadingPlugins` share common methods related to initialization, synchronization with the renderer, and management of lightweight instances.

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.

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.

Synchronization

The `Synchronize()` routine allows the plugin to respond to synchronization signals delivered by the renderer. 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.

Lightweight instancing

For shading plugin types which support the creation of multiple lightweight instance classes not derived from `RixShadingPlugin` (i.e. `RixBxdf`, `RixDisplacement`, and `RixLight`), the renderer can potentially create many, many instances over the course of the render. 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. Over the lifetime of a render there can be a one to many relationship between the former and the latter, and for performance reasons, it is important to keep the cost of the creation of the latter C++ objects low. In order to reduce the cost of these instantiations, the renderer offers the ability to track custom *instance data* with every shading plugin instance that can be shared and reused amongst all the C++ objects that share the same shading plugin instance.

The shading plugin 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, which is at the time the 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 with each new lightweight instantiation.

If the shading plugin does create instance data, it should be stored in the data field of the `InstanceData` struct. If the data requires non-trivial deletion, the `freefunc` field of the `InstanceData` struct 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 a non-zero value.

Any instance data that is created will be automatically returned to the shading plugin methods by the renderer when the lightweight instance is created - for example, when `RixBxdfFactory::BeginScatter()` is invoked to create a `RixBxdf`. The implementation of `BeginScatter()` is now free to use this instance data to reduce the cost of creating the associated `RixBxdf`.

The `RixParameterList` class 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.

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 `RixBxdf` interface in order to get an understanding of the requirements for presence and opacity. In the following code, `PxrDiffuse` 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;
        }
    }
```

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 `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);
```

`RixPattern` plugins do not fully follow the lightweight instancing pattern because they do not have an associated factory object, and do not create lightweight C++ objects because patterns are generally not expected to retain state. However, they do support instance data, and the renderer will return this instance data every time output is requested from the pattern plugin, with the understanding that any expensive computation that can be performed based on an understanding of the uniform parameters can be reused on each invocation of `ComputeOutputParams()`.

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`, along with a corresponding `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.

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

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.