Changes to RixShadingPlugin

Pure-virtual and default implementation in the RixShadingPlugin API

⚠ Because the `RixShadingPlugin` API and the associated subclasses (`RixBxdf`, etc...) also contain non-pure virtual methods, it is **strongly recommended** to use the C++ keyword `override` when overriding methods of the API.

Interactive editing and Options / Render State queries.

RenderMan 23 introduces the possibility of editing displays and LPEs during an interactive rendering session, without needing to restart the entire rendering session (e.g. translate the entire scene again).

As a consequence, several quantities that `RixShadingPlugin` could previously query in the `RixShadingPlugin::Init()` and `RixShadingPlugin::CreateInstanceData()` may change during the rendering session, leading to shading plugins using out-of-date values.

For example, the following may now change between renders:

- resolution, number of displays, and other display-related options
- contents of the `RixIntegratorEnvironment`
- quantities available through the `RixLPEInfo` and `RixCustomLPE` interfaces

In order to avoid counter-intuitive behavior (e.g. restarting a render suddenly yields a very different render), it's is highly recommended for shading plugin to stop accessing these inside the `Init()` and `CreateInstanceData()`. Instead, the `Synchronize()` and (newly introduced) `SynchronizeInstanceData()` methods should be used, allowing the user to update the data stored for the plugin and plugin instances before a new render.

Note that:

- `RixLPEInfo` and `RixCustomLPE` interfaces are not available in Init() and CreateInstanceData() anymore.
- `RixRenderState` and calls to `GetOptions()` are still available, but may return out-of-date values. It is strongly recommended to move the associated code to the synchronization methods.

⚠ The restriction to `RixLPEInfo` implies that functions like `RixBXLookupLobeByName` shouldn't be called in `Init()` or `CreateInstanceData()` either. This function should usually be called in `Synchronize()`.

RixPlugin::CreateInstanceData() and RixPlugin::SynchronizeInstanceData()

Plugin Instance Initialization

The return type of `RixShadingPlugin::CreateInstanceData()` has been changed from `int` to `void`.

```
    virtual void CreateInstanceData(RixContext& rixCtx, RtUString const handle,
                                    RixParameterList const* parameterList,
                                    InstanceData* instanceData)
```

Previously, when a value of `-1` was returned, the renderer would ignore the `InstanceData::data` member, and `nullptr` would be provided as 'instancedata' to the various shading plugin methods (e.g. `ComputeOutputParams()`).

In 23, the renderer directly inspects the value of `InstanceData::data` to detect if the plugin instance allocated private instance data. The (newly introduced) `SynchronizeInstanceData()` will then be given a chance to update the contents of this private instance data, before the pointer is provided to the usual shading plugin methods.

Plugin Instance Synchronization

`RixShadingPlugin::SynchronizeInstanceData()` is a new API that allows user to update the private instance data associated with a plugin instance.

```
    virtual void SynchronizeInstanceData(RixContext& rixCtx, RtUString const handle,
                                         RixParameterList const* instanceParams,
                                         uint32_t const editHints,
                                         InstanceData* instanceData)
```

Because this method will be executed before a new render starts (i.e. after edits have been processed), it allows for options and render state queries to return up-to-date values.

For performance reasons, this method will only be called if the shading plugin instance has explicitly subscribed to it. This is done by setting the (new) `InstanceData::synchronizeHints` member to a non-zero value during `CreateInstanceData()`. This member is a bit field that the plugin instance can use to specify on which kind of edit category should the `SynchronizeInstanceData()` be called. Currently, only two values are exposed, but additional categories may be added shortly (e.g. k_DisplayEdit, k_OptionEdit, k_LPEEdit, etc...).

```
    enum SynchronizeHints
    {
        k_None = 0x00000000,
        k_All =  0xFFFFFFFF
    };
```

Example

Let's consider a plugin instance with a non-trivial `RixShadingPlugin::CreateInstanceData()` method, using option queries and storing the returned values in its own private instance data class.

```
int MyBxdfFactory::CreateInstanceData(RixContext& ctx, RtUString const handle,
                                      RixParameterList const* plist, InstanceData* result)
{
    MyInstanceData* data = new MyInstanceData;

    // This method fills the members of `data` required for RixBxdfFactory::GetInstanceHints().
    computeBxdfInstanceHints(data);

    // This method uses the RixRenderState object to query options values, that are then stored in
    // the `data` structure.
    RixRenderState* rst = (RixRenderState*)ctx.GetRixInterface(k_RixRenderState);
    doSomething(rst, data);

    result->data = data;
    result->freefunc = &ReleaseMyInstanceData;

    return 0;
}
```

Because `MyBxdf::CreateInstanceData()` will only be called once during an interactive rendering session, the option values stored in `data` may become out-of-date, yielding incorrect and/or counter-intuitive results.

In RenderMan 23, this plugin should do the following:

```
void MyBxdf::CreateInstanceData(RixContext& ctx, RtUString const handle,
                                RixParameterList const* plist, InstanceData* result)
{
    MyInstanceData* data = new MyInstanceData;

    // This method fills the members of `data` required for RixBxdfFactory::GetInstanceHints().
    computeBxdfInstanceHints(data);

    // Note: some members of the `data` structures are still initialized to their default values.
    // They will be updated for the current render during the call to SynchronizeInstanceData().
    result->data = data;
    result->freefunc = &ReleaseMyInstanceData;

    // We want SynchronizeInstanceData() to be called in all cases.
    result->synchronizeHints = RixShadingPlugin::SynchronizeHints::k_All;
}

void PxrSurfaceFactory::SynchronizeInstanceData(RixContext& ctx, RtUString const handle,
                                                RixParameterList const* plist, uint32_t editHints,
                                                InstanceData* result)
{
    MyInstanceData* data = static_cast<MyInstanceData*>(result);

    // This method uses the RixRenderState object to query options values, that are then stored in
    // the `data` structure.
    RixRenderState* rst = (RixRenderState*)ctx.GetRixInterface(k_RixRenderState);
    doSomething(rst, data);
}
```


If `MyBxdfFactory::GetInstanceHints()` relies on some member of the private instance data, `CreateInstanceData()` still needs to initialize these, because `RixBxdfFactory::GetInstanceHints()` is currently only called once per rendering session, before the first call to `SynchronizeInstanceData()`.

A similar issue happens with `RixPattern::Bake2dOutput()` and `RixPattern::Bake3dOutput()`: these methods are run before `SynchronizeInstanceData()`, so it is important to ensure that `CreateInstanceData()` is properly initializing the required members of the private instance data.

Alternatively, if no computations need to happen in `RixShadingPlugin::CreateInstanceData()`, it is possible to delegate the management of the private instance data entirely to `SynchronizeInstanceData()` by doing the following:

```
void MyBxdfFactory::CreateInstanceData(RixContext& ctx, RtUString const handle,
                                       RixParameterList const* plist, InstanceData* result)
{
    // We do not allocate any memory for the private instance data here. This is done for each
    // render in SynchronizeInstanceData().

    // We want SynchronizeInstanceData() to be called in all cases.
    result->synchronizeHints = RixShadingPlugin::SynchronizeHints::k_All;
}

void MyBxdfFactory::SynchronizeInstanceData(RixContext& ctx, RtUString const handle,
                                            RixParameterList const* plist, uint32_t editHints,
                                            InstanceData* result)
{
    // Because this method is called before each render, we need to make sure we properly delete
    // the private instance data that was created for the previous render.
    if (result->data && result->freefunc)
    {
        (result->freefunc)(result->data);
    }

    MyInstanceData* data = new MyInstanceData;

    // This method uses the RixRenderState object to query options values, that are then stored in
    // the `data` structure.
    RixRenderState* rst = (RixRenderState*)ctx.GetRixInterface(k_RixRenderState);
    doSomething(rst, data);

    result->data = data;
    result->freefunc = &ReleaseMyInstanceData;
}
```

RixProjectionFactory::CreateProjection() and RixProjection::RenderBegin()

For similar reasons, the `RixProjection` API has changed. Previously, the `RixIntegratorEnvironment` structure was provided to `RixProjectionFactory::CreateProjection()`:

```
    virtual RixProjection* CreateProjection(
        RixContext& ctx,
        RixProjectionEnvironment& env,
        RtUString const handle,
        RixParameterList const* pList) = 0;
```

In order for a `RixProjection` object to respond to edits affecting the `RixIntegratorEnvironment` structure, a new API has been introduced: `RixProjection::RenderBegin()`. Note that we do not pass `RixIntegratorEnvironment` to `CreateProjection()` anymore:

```
class RixProjectionFactory : public RixShadingPlugin
{
// ...
    virtual RixProjection* CreateProjection(
        RixContext& ctx,
        RtUString const handle,
        RixParameterList const* pList) = 0;
// ...
};


class RixProjection
{
// ...
    virtual void RenderBegin(RixContext& ctx, RixProjectionEnvironment const& env,
                             RixParameterList const* instanceParams) = 0;
// ...
};
```

Any code previously in the `RixProjection` constructor that was using `RixIntegratorEnvironment` should now live in `RixProjection::RenderBegin()`.

RixIntegratorEnvironment::deepMetric

Previously, the projection plugins were allowed to modify the value of `RixIntegratorEnvironment::deepMetric` during the creation of the `RixProjection` object. This is not the case anymore, and in `RixProjection::RenderBegin()`, this structure is now read-only.

Projection plugins should now use the a `RixProjection::GetProperty()` mechanism to communicate a particular metric to the renderer (similarly to what was already happening for `RixProjection::DicingHint`):

```
class RixProjection
{
public:
// ...
    /// Expresses depth metric to use for samples in deep output.  Set to k_cameraZ for standard
    /// perspective or orthographic projections. Use k_rayLength for spherical, cylindrical,
    /// etc... with -Z rays.
    /// Default value (if no value is returned) will be: k_cameraZ
    enum DeepMetric
    {
        k_cameraZ   = 0,
        k_rayLength = 1
    };

    enum ProjectionProperty
    {
        // enum DicingHint - see above
        k_DicingHint,

        // enum DeepMetric - see above
        k_DeepMetric,
// ...
};
```

A typical implementation of the `RixProjection::GetProperty()` method would look like this:

```
RixSCDetail GetProperty(
        ProjectionProperty property,
        void const** result) const override
    {
        switch (property)
        {
            case k_DicingHint:
                *result = &dicingHint;
                return k_RixSCUniform;
                break;
            case k_DeepMetric:
                *result = &deepMetric;
                return k_RixSCUniform;
                break;
            case k_FieldOfView:
                *result = &fovHint;
                return k_RixSCUniform;
                break;
            default:
                return k_RixSCInvalidDetail;
        }
    }
```

RixIntegratorFactory

The integrator plugins are now using a Factory, similarly to RixProjection.

In general, it should be sufficient adding something like this to the existing integrator plugins:

```
class MyIntegratorFactory : public RixIntegratorFactory
{
public:
    virtual RixSCParamInfo const* GetParamTable() override;

    virtual void Synchronize(RixContext&, RixSCSyncMsg, RixParameterList const*) override
    {}

    virtual int Init(RixContext& ctx, RtUString const) override
    {
        return 0;
    };
    virtual void Finalize(RixContext&) override{};

    virtual RixIntegrator* CreateIntegrator(RixContext& rixCtx, RtUString const handle,
                                            RixParameterList const* pList) override;
    virtual void DestroyIntegrator(RixIntegrator const* integrator) override;
};


RixIntegrator* MyIntegratorFactory::CreateIntegrator(RixContext& rixCtx, RtUString const handle,
                                                     RixParameterList const* pList)
{
    PIXAR_ARGUSED(handle);
    PIXAR_ARGUSED(pList);
    return new MyIntegrator(this, rixCtx);
}

void MyIntegratorFactory::DestroyIntegrator(RixIntegrator const* integrator)
{
    delete static_cast<MyIntegrator const*>(integrator);
}
```

And replace the existing `RIX_INTEGRATORCREATE` and `RIX_INTEGRATORDELETE` by the following:

```
RIX_INTEGRATORFACTORYCREATE
{
    PIXAR_ARGUSED(hint);
    return new MyIntegratorFactory();
}

RIX_INTEGRATORFACTORYDESTROY
{
    delete ((MyIntegratorFactory*)factory);
}
```