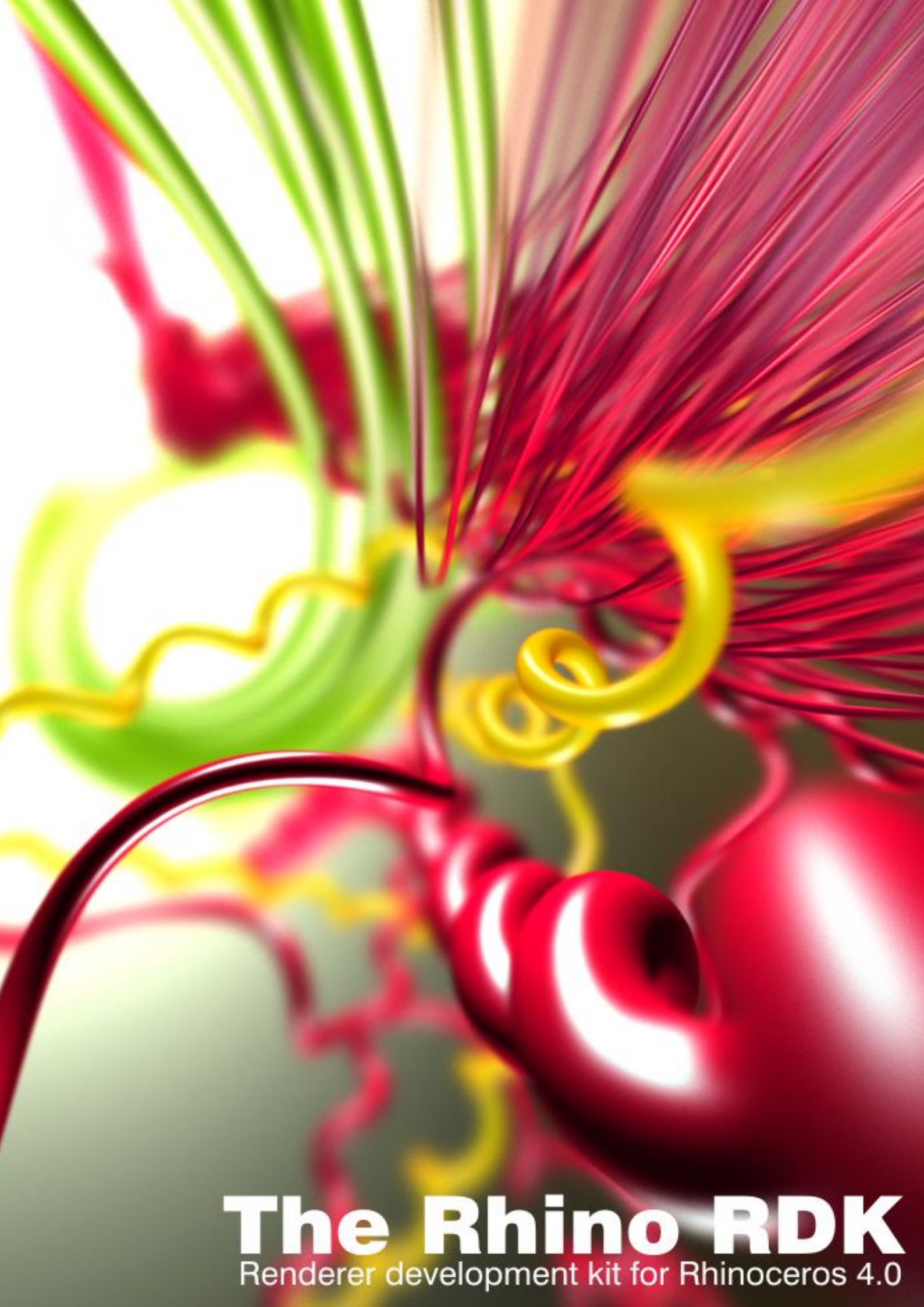

The Rhino RDK

Renderer development kit for Rhinoceros 4.0

Contents

Overview

The Rhino RDK is a collection of tools that extend the Rhino application platform with visualization specific capabilities. In particular, the RDK provides:

Features

- Extensible Material, Environment and Texture editor.
- Frame buffer implementation with post and channel handling.
- Pre-process custom mesh provision interface for 3rd party developers.
- Built in procedural textures, and a texture generation pipeline SDK for further extension.
- Improved render pipeline that makes it much easier for developers to implement a renderer engine in Rhino.
- Rhino sun light and sun-angle calculation tools.
- Automatic shader UI support for 3rd party Material/Environment/Texture providers.
- New UI widgets for developers.
- Several utility classes to aid in the development of renderers and visualization related tools.
- Decal support similar to Flamingo 2.0.

Many of these tools are available to Rhino users in their raw form, but the RDK's real power is as an API for developers of rendering software.

This document aims to cover the basics of how the RDK is structured and how to code the most common operations. For more detail, please consult the RDK help which is installed by default with the RDK SDK.

Tools for Rhino users.

Overview

Some of the RDK tools work right out of the box. However, many of them are really fairly useless without support from specific renderer developers.

The current RDK plug-in installer can be downloaded from here:

http://download.mcneel.com/rdk/1.0/redirect/rdk_plugin.asp

Features

Material Editor

Command: MaterialEditor

The Material Editor is a generalized component that allows the assignment and editing of custom materials to objects and effectively replaces the Object Properties method of material assignment in Rhino.

The RDK Material Editor displays a palette of materials stored in the document. These materials can be assigned to objects in the scene using Drag and drop, or by using the Assign To Object/Assign To Layer menu items. Materials that are not assigned to objects or layers are still stored in the document. Triangular icons on the corners of the thumbnails indicate those that are assigned.

Initially, three “Basic Materials” are added to the material editor. These can be edited by selecting their thumbnails and changing the parameters in the editor section below the thumbnail display. The thumbnail display area can be resized by dragging the border between the display and the parameter area. The “Tasks” and “Nodes” pane can be opened by clicking on the grey area to the left of the editor.

New materials can be added from the menu, the tasks pane or the thumbnail view right-click menu. Adding a new material displays all of the available material types – including new and existing materials. Initially, without an RDK-aware renderer installed, this list will only contain the Basic and Blend Materials. 3rd party renderers can add to this list.

Materials can be saved and loaded to files using the menus. It is possible to create libraries of materials for use in other documents using this method.

Environment Editor.

Command: EnvironmentEditor

The Environment editor works much like the Material Editor. It can be used to change basic rendering settings for the Rhino renderer or any renderer that

uses the built-in background settings in Rhino. 3rd party renderers can also use the Environment Editor to control custom environment/background settings that are specific to their product.

Without an RDK-aware 3rd party renderer installed, the Basic Environment is the only option in the Environment Editor.

Double clicking on an environment in the editor makes it the current Environment.

Texture Editor.

Command: TextureEditor

The RDK provides a Texture Editor that works in combination with Materials and Environments to allow texturing – including procedural textures – to be rendered by all renderers.

Although 3rd party renderer developers can extend it, the RDK provides a number of simple procedural and image based textures by default including Checker, Dot, Marble, Noise and Gradient.

Any renderer can render the textures added to materials by the Texture Editor, although RDK-aware renderers will be able to produce much higher quality versions of the textures.

Rhino "Sun" light

Command: Sunlight

The RDK adds a new command to Rhino that allows the placement of standard Rhino parallel lights with a sun angle calculator.

Tools for Developers

Overview.

As the RDK is a software development kit for rendering applications, the lion's share of the functionality is only available to software developers. This includes the ability to define your own materials, textures and environments and have them work seamlessly within their respective editor. Utilize the new standard frame buffer and built pre- and post-process effects (including defining your own effects).

SDK downloads.

The current RDK SDK installer can be downloaded from here:

http://download.mcneel.com/rdk/1.0/redirect/rdk_sdk.asp

The current RDK plug-in installer can be downloaded from here:

http://download.mcneel.com/rdk/1.0/redirect/rdk_plugin.asp

When developing your RDK-aware render plug-in, you will need the SDK installed on your workstation. When distributing your plug-in, you may decide to include the RDK installer as part of your own plug-in installer. To do this, simply install the RDK plug-in installer into the temp folder, and run it using the /silent command line option.

The RDK includes a demonstration project called "Marmalade" which you will find in the c:\program files\Rhino 4.0 SDK\Marmalade folder. This plug-in shows how to set up your paths for header and lib files, and demonstrates many common RDK tasks such as creating custom materials and using the frame buffer.

The CRhRdkPlugIn class

Every RDK-aware plug-in will have a class derived from CRhRdkPlugIn or CRhRdkRenderPlugIn. This class is one of the main methods of communicating with the RDK.

An example of a plug-in class is shown in the Marmalade example plug-in included with the RDK SDK download.

Materials, Environments and Textures.

In the RDK, Materials, Environments and Textures are all referred to as "Contents".

Much of the RDK functionality is based around the Material / Environment / Texture editors (content editors) and their implementation. To understand how custom contents can be added to the editor, it is necessary to understand how the basics of the objects that implement them are designed.

Implementing CRhRdkContent

The basic job of a content object is to provide some, or more usually all, of these functions:

- Serialization
- Simulation
- Shader-provision.
- A user interface.

All content classes are derived from CRhRdkContent and CRhRdkCoreContent. These two classes implement much of the back-end code for serialization, a method to create UI automatically from the serialization parameters, and provide a common interface to support the other two.

Actual custom content classes must be derived from CRhRdkMaterial, CRhRdkEnvironment or CRhRdkTexture.

Serialization

Every content object will have data. Since the RDK provides serialization support for all content objects, you will need to communicate with the RDK to serialize the data for your objects. Internally the RDK serialization is an XML stream, and while it is possible to write the XML stream directly, it is usually much easier to override the CRhRdkContent::AddParameters and CRhRdkContent::GetParameters. For details, please read the SDK documentation and see the example in MarmaladeAutoUIMaterial.cpp

Simulation

Content objects must provide a method for other renderers, including the standard Rhino display, to represent your custom content object. Without simulation, the rendered mode in Rhino would not update to reflect your custom definitions. In addition, simulation makes it possible for users to render objects with your content objects attached without having your plug-in installed.

Depending on their type, content objects should override SimulateMaterial, SimulateEnvironment or SimulateTexture and provide as much information as possible to represent that custom definition given the standard ON_Material, CRhRdkSimulatedTexture and (currently) CRhRdkSimulatedEnvironment classes.

Shader-provision

Most render engines have custom binary definitions for their materials, environments and textures. Usually these will be instances of classes with filled in “parameter blocks” initialized from the data in a content object. Sometimes they will simply be strings pointing to files or locations in a library.

These objects are termed “shaders” in the RDK.

When an object needs to be rendered by your rendering engine, you will need to get at these objects. As part of your content object implementation you should override CRhRdkContent::GetShader to return a pointer to this object.

Note that since you are both the provider and client of this function, how you implement it is up to you. The data is private and its type and allocation details are completely up to you. However, you must check the incoming render engine UUID to ensure that you can render shaders of that type. Usually you will call “IsCompatible” on the UUID and return if it returns false.

A user interface.

When a content object is selected in the content editor, a user interface is displayed in the lower panel. This interface is provided by the custom content object.

CRhRdkCoreContent provides a method to create an automatic user interface from your serialization. All that is required is to call “CreateAutomaticUI” in response to the call your CreateUI function, and to provide an implementation for the AddAutoParameters and GetAutoParameters functions (which will look very similar to your serialization functions). The RDK does the rest.

You can also elect to provide a custom user interface for your content object by creating your own CRhRdkExpandingSection derived dialogs and adding them to the editor. An example of this is shown in the MarmaladeCustomUIMaterial.cpp example file.

Other content functions

In addition to the basic “big 4” functions of a content object, you may want to add any of the following:

- Support for child objects like textures
- Support for the automatic “Texture Summary” section.
- Support for procedural texture evaluation (textures only – see below)

Procedural textures.

In addition to simulation (which essentially involves resolving a texture into a bitmap), custom textures can provide an evaluator that is capable of determining the color of a texture in UVW space. If provided, simulation for textures is not required, as it is implemented in terms of the evaluator.

A texture evaluator is a lightweight object that can be used to render a texture over the whole of UV(W) space and it returned from the CRhRdkTexture::NewTextureEvaluator method. Note that the evaluator object should not store a pointer to the texture.

An example procedural texture is provided as part of the Marmalade example.

The built in procedural textures all use this method to provide high quality procedurals to RDK-aware renderers.

CRhRdkContent clients

Overview

All rendering plug-ins that use custom content objects to represent materials, textures and environments will also be clients for the CRhRdkContent interface as they create content native to their rendering engine from the content objects in the document.

Retrieving materials assigned to objects

Given the UUID of an object in the database, you can retrieve the material assigned to it using the following code:

NB: In the following code samples, the variable “uuidMe” is the UUID of your render plug-in – ie, the value returned from CRhinoRenderPlugIn::PlugInID().

```
CRhRdkObjectDataAccess da(uuidObject);
const UUID uuidContent = da.MaterialInstanceId();

const CRhRdkContent* pContent = ::RhRdkFindContentInstance(uuidContent);

MyMatShader* pMatShader = NULL;

if (NULL != pContent && pContent->IsKind(RDK KIND MATERIAL))
{
     const CRhRdkMaterial* pMaterial = static_cast<const CRhRdkMaterial*>(pContent);

     pMatShader = reinterpret_cast<MyMatShader*>(pMaterial->GetShader(uuidMe,
NULL));

     if(NULL == pMatShader)
     {
            //This is not a native material – probably a basic material
```

```
            ON_Material mat;
            pMaterial->SimulateMaterial(mat);

            pMatShader = SimulateOneOfMyOwnShadersFromAnON Material(mat);
    }
}
```

Notice the call to “GetShader” – this is the correct way to retrieve shaders that you supply from the CRhRdkContent::GetShader function.

NB: If you are using the render mesh iterator (see below) there is no need to find the material instance as it is handed to you as part of the data. You should only use the above method if you are handling mesh extraction from Rhino yourself, or if you are using another method of rendering the objects in the database.

Getting the current environment

This is relatively easy – the current Environment can be queried by calling RhRdkCurrentDocumentContentInstanceId(RDK_KIND_ENVIRONMENT)

```
const UUID uuidInstance =
::RhRdkCurrentDocumentContentInstanceId(RDK KIND ENVIRONMENT);

const CRhRdkContent* pContent = ::RhRdkFindContentInstance(uuidInstance, false);

MyEnvShader* pEnvShader = NULL;

if (NULL != pContent && pContent->IsKind(RDK KIND ENVIRONMENT)
{
    const CRhRdkEnvironment* pEnvironment =
  static_cast<const CRhRdkEnvironment*>(pContent);

    pEnvShader = reinterpret cast<MyEnvShader*>(pEnvironment->GetShader(uuidMe,
NULL));

    if(NULL == pShader)
    {
            //This is not a native environment – probably a basic
            Proto ON Environment env;
            pEnvironment->SimulateEnvironment(env);

            pEnvShader = SimulateOneOfMyOwnShadersFromAProto_ON_Environment(env);
    }
}
```

Textures

While it is possible that you will encounter the need to find a particular texture instance from its UUID, more commonly you will need to evaluate textures when they occur as children of materials or environments. In this case you will retrieve the pointer to the texture object when you query the child slots of a content object (note that textures can also have sub-textures).

In addition to the two ways to represent a material or environment, native shader provision and simulation, textures also support “evaluation”. Generally you will prefer to deal with textures in the following order:

- Native-shader provision
- Evaluation
- Simulation

If you can create a native shader that your own rendering engine understands, then this will probably work best. You retrieve this shader using the standard “GetShader” function and you will need to provide the implementation when you write the texture class.

Most textures provide a texture evaluator which you retrieve using CRhRdkTexture::NewTextureEvaluator. Use this object to evaluate the texture over UV(W) space. You may want to implement a native shader that owns one of these objects to implement standard RDK procedural textures in your renderer.

Finally, simulation will convert the texture to an image. This is the least preferable of the 3 options since UV(W) space < 0.0 and > 1.0 will not be included, and the texture will be simulated at a lower resolution than would otherwise be possible.

For example, to get the diffuse texture for a custom material (where the texture is stored as a child of the material in the child slot labelled “DiffuseChildSlot”)

```
const CRhRdkContent* pContent = pMaterial->FindChild(L"DiffuseChildSlot");
if(pContent && pContent->IsKind(RDK_KIND_TEXTURE))
{
     const CRhRdkTexture* pTexture = static cast<CRhRdkTexture*>(pContent);

     MyTexShader* pTexShader = NULL;

     //First option – native shader
     pTexShader = reinterpret_cast<MyTexShader*>(pTexture->GetShader(uuidMe, NULL));

     if(NULL == pTexShader)
     {
            //Second option - evaluation
            IRhRdkTextureEvaluator* pTexEval = pTexture->NewTextureEvaluator();
            if(NULL != pTexEval)
            {
                   pTexShader = CreateNativeTextureShaderFromEval(pTexEval);
                   //Note:  The texture evaluator is now owned by the
            //shader and it should delete it
            }
            else
            {
                   //Third option – simulation
                   ON Texture tex;
                   pTexture->SimulateTexture(tex);

                   pTexShader = CreateNativeTextureShaderFromSim(tex);
            }
     }

     //Your renderers method to add a texture shader to a material shader
     if (NULL != pTexShader)
```

```
        pMatShader->AddTextureShader("Diffuse", pTexShader);
}
```

CCIs (Compound Content Implementers)

For the basic RDK content types, there is often a CCI class in the SDK which allows you to implement a method of creating a native shader version of the basic type. This is a better method of dealing with the basic types (such as Basic Material and Blend Material) than simulation because it is possible to render children using native shaders in this way. Without CCI implementation, all children are rendered as simulations for the built in types.

To implement a CCI for your render engine, derive and implement a class from one of the CCI classes and register it using IRhRdkCompoundContentImplementors::Add() in the CRhRdkPlugIn::RegisterCompoundContentImplementors virtual function.

```
virtual void
MyPlugIn::RegisterCompoundContentImplementors(IRhRdkCompoundContentImplementors&
ccis) const
{
    ccis.Add(new CMyBasicMaterialCCI);
}
```

The Render Pipeline

Overview

To preserve compatibility with the CRhSdkRender class, the RDK render pipeline uses a similar interface and name. To create and start a render pipeline in RDK, create an instance of a class derived from CRhRdkSdkRender and call the "Render" method.

An example of the use of the CRhRdkSdkRender class is included in the Marmalade example included with the RDK SDK download.

Features

Geometry and lights

The older CRhSdkRender class provides the renderer with geometry and lights using a system of callbacks. While this method is still supported in the RDK, the preferred method is to use a Render Mesh Iterator that you can get from the NewRenderMeshIterator function. You should delete the object when you are finished with it.

Note that the Mesh Iterator retains the meshes in memory during its lifetime so it is a good way to get a complete set of meshes for the scene during the rendering process. If you need to copy the meshes into your renderer's own format, you can delete the object when you are finished copying.

The Mesh Iterator provides materials for each mesh automatically, and deals with pre-process effects such as displacement without requiring any special coding.

Frame Buffer

The RDK implements a complete multi-channel frame buffer including post-effects, cloning, zooming and file output.

An example of the use of the frame buffer class is included in the Marmalade example included with the RDK SDK download.

All that is required to control the frame buffer is to set pixel colors during rendering and invalidate the correct area of the display.

Custom Render Mesh Providers

A custom render mesh provider is an object, registered with the RDK, which determines whether the render mesh for an object should be customised. The provider may provide any number of meshes and materials for an object.

Several labs plug-ins for Rhino 4.0 use this method to generate custom meshes at render time, including displacement, curve piping and edge softening.

Other uses may be to implement vegetation generators, view dependant objects etc.

Note that a licence request function in the CRhRdkPlugIn class allows the provision of proprietary meshes to specific renderer plug-ins. For more details, see the full SDK documentation.

Other Features for developers.

All of the following features are documented in the SDK documentation provided with the RDK SDK download. They are listed here to give developers a general idea of the scope of the RDK.