Custom Component Development Using RenderMonkey SDK

Natalya Tatarchuk
3D Application Research Group
ATI Research, Inc

Overview

- Motivation
- Introduction to the SDK
- SDK Functionality Overview
- Conclusion

Why Do We Need a Plug-in SDK?

- Developers like having control in their hands
  - They want the ability to improve any program themselves – when and as they need it
- But it's more than that: the pluggable architecture works for us as well
  - The entire application is developed as plug-ins
  - Makes it easy to create new components without re-writing the application
  - We are using the SDK for development of features

Plug-in Architecture Philosophy

- Having a pluggable architecture allows you to solve problems you have not anticipated
  - Especially by developers themselves
    - Specific to their projects
  - Allows us to create new tools in the future as the need arises

RenderMonkey Application Design

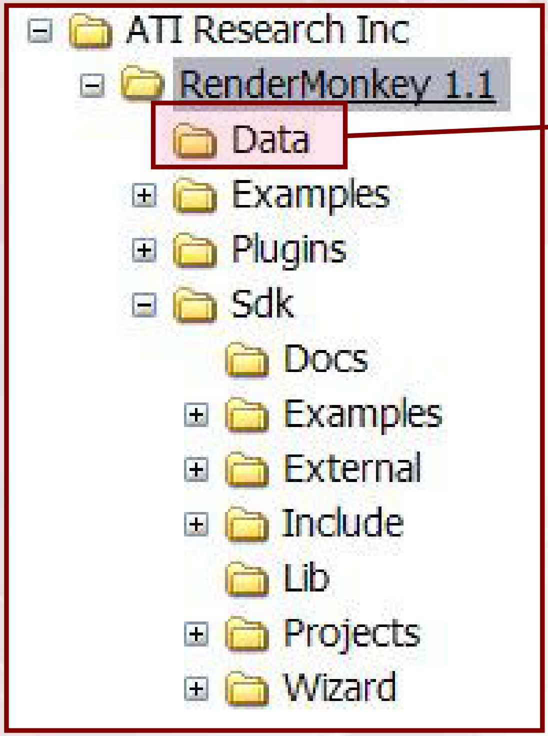
- Single document application: only one workspace edited at a time
- All data necessary to render effect is stored in a run-time database
  - Effect database node overview can be found in “Beginner Shader Programming with RenderMonkey” presentation from GDC 2003 on www.ati.com/developer
- All real-time changes to the database are managed by the application and propagated to the plug-ins
  - Application sends out Windows-style messages to plug-ins’ message handler
- All rendering resources exist in the viewer plug-ins: other plug-ins have no access to that data

Supported API and Compatibility

- The SDK is written in pure C++
- RenderMonkey version 1.5 and SDK 1.0 support plug-in development in *both* Visual Studio 6.0 and Visual Studio .NET
- Developers can create plug-ins using only Win32 API or MFC as they please

Application and SDK Layout

Installed application directories

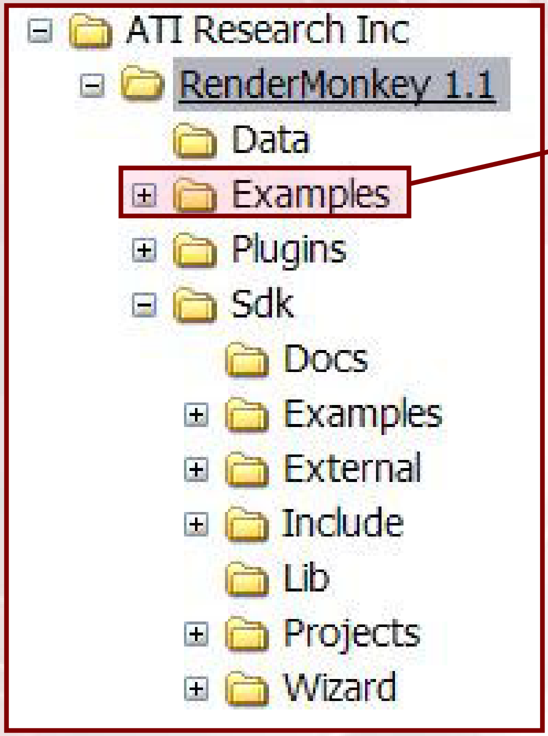


- ATI Research Inc
  - RenderMonkey 1.1
    - Data
    - Examples
    - Plugins
    - Sdk
      - Docs
      - Examples
      - External
      - Include
      - Lib
      - Projects
      - Wizard

Stores RenderMonkey data files:
- Shader editor initialization files
- Default workspace definition
- DTD
- RmInclude.h for HLSL includes
- Definition for supported rendering and texture states

Application and SDK Layout

Installed application directories

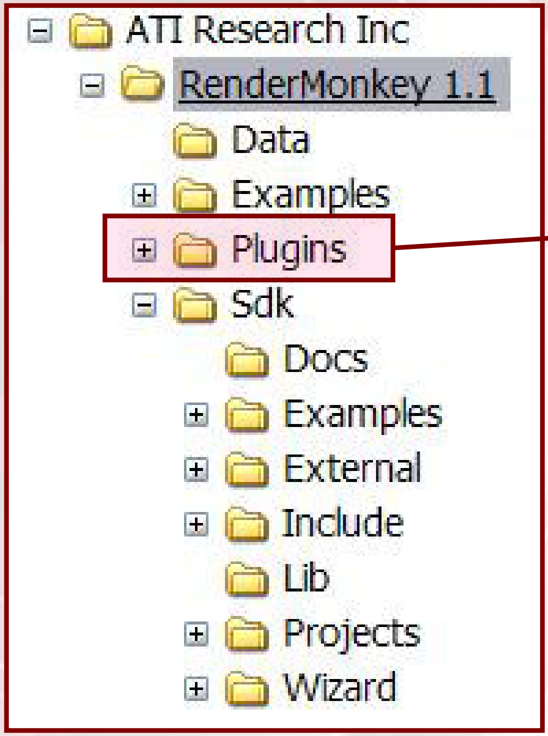


- ATI Research Inc
  - RenderMonkey 1.1
    - Data
    - Examples
    - Plugins
    - Sdk
      - Docs
      - Examples
      - External
      - Include
      - Lib
      - Projects
      - Wizard

Stores all example workspaces shipped with the application

Application and SDK Layout

Installed application directories

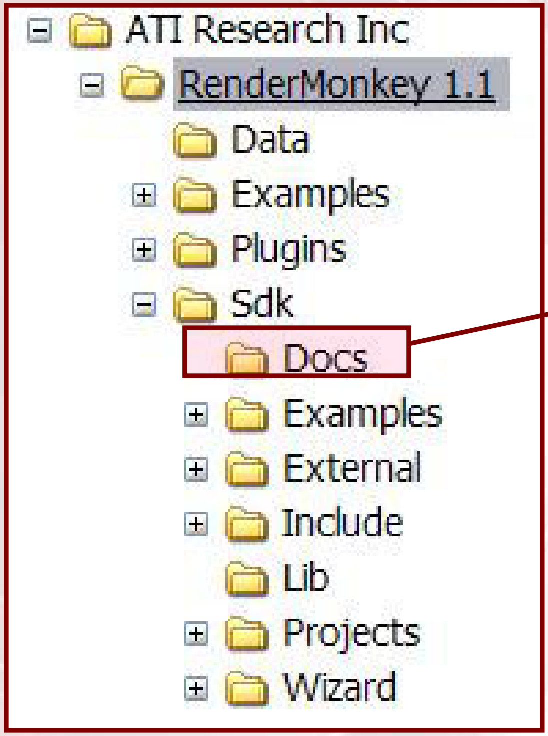


- ATI Research Inc
  - RenderMonkey 1.1
    - Data
    - Examples
    - Plugins
    - Sdk
      - Docs
      - Examples
      - External
      - Include
      - Lib
      - Projects
      - Wizard

Plug-ins depository. The application loads all DLLs from this directory on startup and parses them for plug-ins. New plug-ins should be placed there.

Application and SDK Layout

Installed application directories

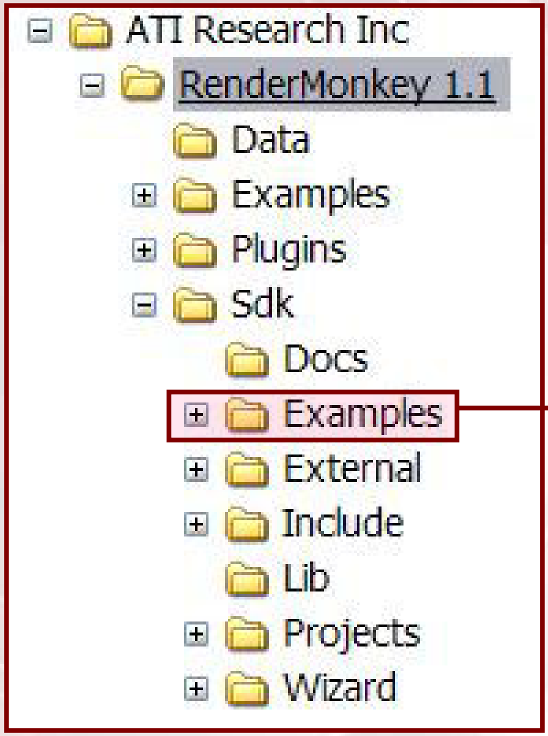


```
⊟ ATI Research Inc
    ⊟ RenderMonkey 1.1
          Data
        ⊞ Examples
        ⊞ Plugins
        ⊟ Sdk
              Docs
            ⊞ Examples
            ⊞ External
            ⊞ Include
              Lib
            ⊞ Projects
            ⊞ Wizard
```

SDK Documentation

Application and SDK Layout

Installed application directories

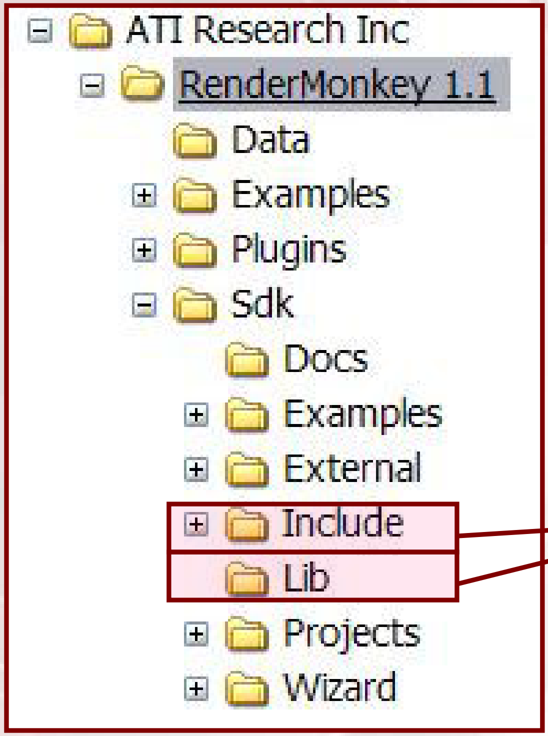


```
⊟ ATI Research Inc
    ⊟ RenderMonkey 1.1
        Data
      ⊞ Examples
      ⊞ Plugins
      ⊟ Sdk
            Docs
          ⊞ Examples
          ⊞ External
          ⊞ Include
            Lib
          ⊞ Projects
          ⊞ Wizard
```

SDK Example Plug-ins

Application and SDK Layout

Installed application directories

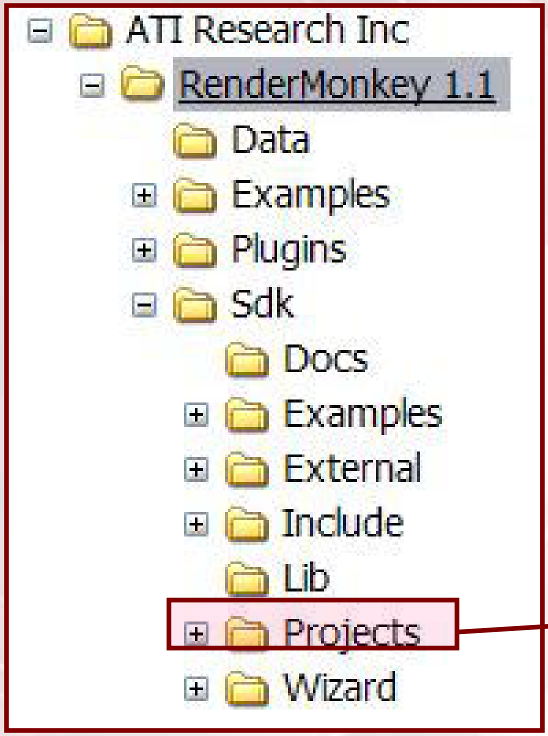


- ATI Research Inc
  - RenderMonkey 1.1
    - Data
    - Examples
    - Plugins
    - Sdk
      - Docs
      - Examples
      - External
      - Include
      - Lib
      - Projects
      - Wizard

RenderMonkey SDK Include files and libraries

Application and SDK Layout

Installed application directories

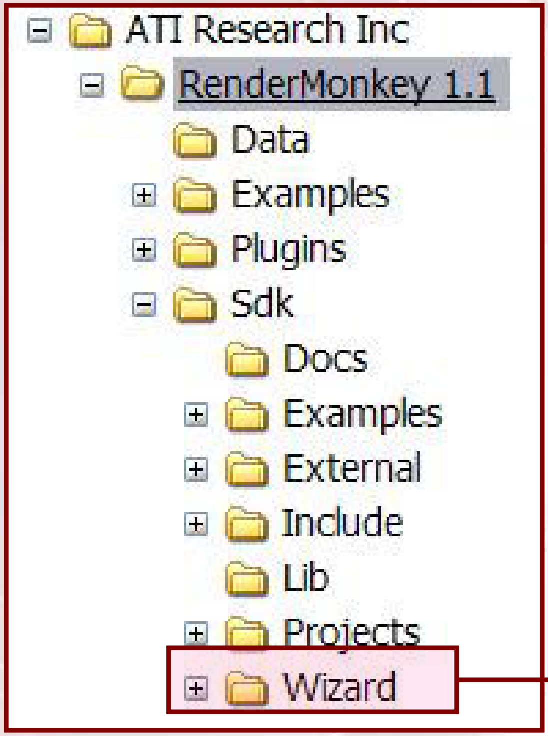


- ATI Research Inc
  - RenderMonkey 1.1
    - Data
    - Examples
    - Plugins
    - Sdk
      - Docs
      - Examples
      - External
      - Include
      - Lib
      - Projects
      - Wizard

This is where wizard-generated project files will be placed

Application and SDK Layout

Installed application directories



- ATI Research Inc
  - RenderMonkey 1.1
    - Data
    - Examples
    - Plugins
    - Sdk
      - Docs
      - Examples
      - External
      - Include
      - Lib
      - Projects
      - Wizard

Contains code samples used by the plug-in wizard

SDK Includes and Libraries

- Libraries shipped with the SDK
  - RmCore
    - Main RenderMonkey SDK library: contains the node database definition, plug-in interfaces, application interface and various manages interfaces, as well as custom classes
  - RmUtilities
    - Support for double-buffering UI windows, and Win32 hooks utilities
  - RmMFCUtilities
    - RenderMonkey MFC widgets and utilities
  - RmGfxUtil
    - Texture image management and creation
    - Image conversion
    - Device retrieval

Plug-in Project Setup

- Use RenderMonkey project wizard
  - Run from Utilities / Plug-In Wizard… menu option
  - Select plug-in type that you wish to create from available plug-in list
  - Type your project name and click “Ok”
  - The wizard will create all necessary code to create a new plug-in of that type in SDK/Projects directory – including project files for Visual Studio 6.0 and .NET with all the necessary project settings
- Do it by hand: Instructions are in SDK/Doc SDK Documentation.doc file
  - Tedious and prone to mistakes!

Plug-in DLL Organization

- A plug-in DLL can contain multiple plug-ins in a single DLL
- A single DLL must implement these entry points:
  - **`RmInitPlugInDLL`**
    - Initialization and setup particular for the actual DLL – a good place to instantiate all plug-in instances
  - **`RmGetNumPlugIns`**
    - The number of plug-ins implemented in a particular DLL
  - **`RmGetPlugIn`**
    - Retrieve a particular plug-in from the DLL by index
  - **`RmFreePlugIn`**
    - Free a particular plug-in from memory
  - **`RmUninitializePlugInDLL`**
    - This entry point gets called before the DLL is unloaded by the app

Node Database Overview

- All data necessary to render an effect is stored in nodes
  - Effect node, pass node, model node, etc.
- RenderMonkey maintains node rules to ensure valid node contents
  - Ex: Only one active vertex shader is allowed in a pass
  - Only one model reference is allowed in a pass
  - Multiple texture objects are allowed in a pass – but none in effect

Node Database (cont.)

- Custom nodes can be added by adding child nodes to existing nodes in the db
  - As long as it doesn't violate current node rules
  - If it does, new data can always be added as "annotation data" via adding a string node
- Currently no support for the ideal custom node solution in the workspace window
  - Cannot extend the database by creating new node classes at the moment
  - Will be added in the future releases

Application Access

- `IRmApplication` interface – accessible from any plug-ins from a singleton instance
  - `IRmApplication* pApp = getRmApp();`
- Main entry point for window creation and management
- Allows users to clear output window text and specify new text
- Contains an instance of edited workspace and provides plug-ins with access to it as well as new node creation and editing functionality
- Stores access to various manager interfaces:
  - Application registry manager
  - Predefined variable manager
  - More…

XML Management

- **`IRmXMLManager`** interface
  - Accessable from the main application by **`IRmApplication::GetXMLManager()`**
- Hides the implementation details of dealing with an XML file through MSXML
- All data from .rfx can be conveniently queried through this interface
  - Use this interface for loading data and saving to XML for custom nodes
- Node rules are described in the DTD shipped with the application
  - Allows automatic XML validation

Node Transactions

- All application events (non-Windows) and all changes to the node database are propagated to the plug-ins by RenderMonkey messages
  - All plug-ins must support a message handler:

```
virtual int MessageHandler( int nMessageID, int nMessageData,
                            int nMessageParameter = 0,
                            RmPlugInID plugInID = RM_PLUGIN_ID_NULL ) = 0;
```

  - Additional data is passed as message parameters
    - Can pass node information, data structures, etc.

- Any plug-in can send out any of these messages at any point by notifying the application
  - **`IRmApplication::BroadcastMessage(..)`** entry point
- All message definitions are delineated in RmDefines.h

Supported Transactions and Messages

- Run-time database related messages
  - Node Update / Value change / Name change
  - Node Added / Node Deleted
- Application notification messages
  - File New / File Opening / File Close
  - File Open Complete / File Close Complete
  - Application Closing
  - Query to save data:
    - Notification to the plug-in that a workspace is about to be saved – it should propagate any information about its nodes to the run-time database now

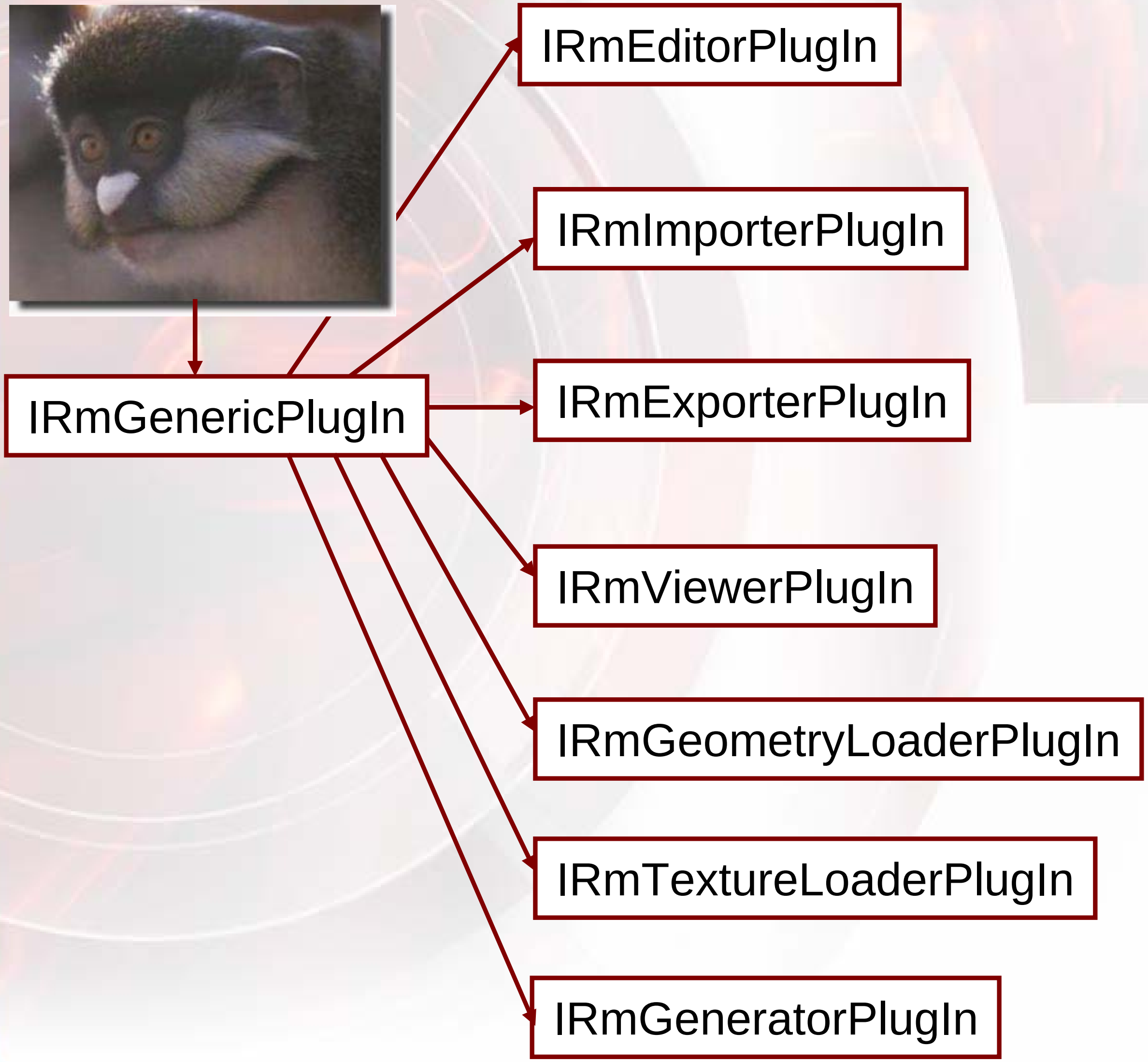
Additional Messages

- Effect-specific messages
  - Shader compilation messages – Pre-compilation / Compile / Post-compile
    - Received by all plug-ins
- A number of viewer-specific messages
  - Change active effect
    - Sent out to plug-ins prior the viewer receiving it
  - View management messages
    - Update all rendering / Update textures / Update models
    - Reset current view
    - View camera mode notification
- Viewer messages can be triggered by any plug-in that wishes to update the rendering

Plug-in Management

- Application loads all plug-ins from the \plugins directory
  - Sorts them by supported plug-in type
- Automatically manages all plug-ins according to their type
  - If you create a new editor, the application will automatically associate it with the node and add it to the context menu for that node
  - Application organizes the menus for all plug-in types automatically
- Application remembers the last plug-in used for editing a node
  - For node types that have multiple editor plug-ins associated with it, the user just has to select a plug-in from "Edit with.." menu and the next time they double-click on a node of that type, that plug-in will be executed

Supported Plug-in Types



IRmGenericPlugIn

IRmEditorPlugIn

IRmImporterPlugIn

IRmExporterPlugIn

IRmViewerPlugIn

IRmGeometryLoaderPlugIn

IRmTextureLoaderPlugIn

IRmGeneratorPlugIn

Plug-in Description Structure

- Each plug-in is identified by the interface it implements and a plug-in description structure
- Description structure contains
  - Plug-in type (see RmTypes.h for enumeration)
    - Must match plug-in interface
  - A list of node types supported by plug-in
    - Used by the application to associate and manage plug-ins
  - SDK version (major and minor)
  - Supported rendering API
    - Plug-ins can only be DX or GL plug-ins, or API-agnostic
  - Plug-in name
    - Used by the application to display in the context menus

Generic Plug-in Interface

- Base class for all plug-ins
- Designed to receive communication messages from the application
- Can create a property page dialog for main application preferences dialog for this plug-in – application automatically manages that dialog
- Entry points:
  - **`Init(..)`**
  - **`Uninitialize(..)`**
  - **`GetPlugInDescription()`**
  - **`MessageHandler(..)`**
  - **`HasPropertyDlg()`**
  - **`AddPropertyDlg(..)`**

Importer Plug-in Interface

- Allows developers to bring in data from other formats into RenderMonkey
  - – Custom engine scripts
- Flexible import association
  - – Users can select to import data to an entire workspace (through File / Import)
  - – Or into a particular node
    - That can be used to import textures or some other data directly into nodes
- Single entry point:
  - – ImportNode(RmNode *pNodeToImportInto)

Exporter Plug-in Interface

- Developers can export contents of a single node or the entire open workspace (through File / Export menu option) into their custom data format
  - Our own FX exporter is written as this plug-in type
- Entry point:
  - **`ExportNode(RmNode *pNodeToExport)`**

Editor Plug-in Interface

- All editor widgets shipped with RenderMonkey are editor plug-ins
- Developers can use this plug-in type to create custom widgets
- Entry point:

```
HWND EditNode(HWND    hParentWindow,
              RmNode *pNode)
```

  - Invoked by the main application whether a user double-clicked on a node supported by the plug-in or selected the plug-in through "Edit with.." menu or by direct EditNode() call

Window Creation and Management in RenderMonkey

- Application supports creation of Win32 and MFC windows in plug-ins
- RenderMonkey allows plug-in developers to create dialog windows, docking windows, MDI child windows
  - The latter two are created through entry points in the application interface to ensure Visual Studio 6.0 and .NET compatibility
    - `CreateDockingWindow(..)`
    - `CreateMDIChildFrame(..)`
  - The actual contents of docking and MDI windows can be added to the respective frame windows
  - All main plug-in windows must be registered with the application
    - `RegisterWindow()`

Generator Plug-in Interface

- Used to create contents for particular nodes or to create new nodes based on the selected node information
  - Procedural geometry generation
  - Procedural texture generation
- Entry point:
  - **`GenerateData(..)`**

Geometry Loader Plug-in Interface

- Used to load contents of geometry objects by the application
  - 3DS Loader / X / OBJ Loader plug-ins
- Invoked whenever a user selects a file to load geometry for a model node
- Entry points:
  - **`GetSupportedExtensions(..)`**
    - The application uses this method to determine which model file extensions it can support based on all of the geometry loader plug-ins
  - **`CanLoadGeometry(..)`**
    - Tests whether this plug-in can load geometry data from a given file
  - **`LoadGeometry(..)`**
    - Actually load geometry data into the specified model node

Texture Loader Plug-in Interface

- Used to load contents of textures by the application
- Invoked whenever a user selects a file to load a texture
- Entry points:
  - **`GetSupportedExtensions(..)`**
    - The application uses this method to determine which texture file extensions it can support based on all of the texture loader plug-ins
  - **`CanLoadTexture(..)`**
    - Tests whether this plug-in can load texture data from a given file
  - **`LoadTexture(..)`**
    - Actually load texture data into the specified model node

Support for Undoable Operations

- RenderMonkey allows developers create their own complex undoable operations – it will manage execution / redo of those
- Supports nested undoable operation
  - Start making an undo op by calling **`StartUndoMaking()`** with the op name
  - If you wish to nest additional undos, call **`StartUndoMaking()`** with a pointer to the parent undo operation
    - No limit on the number of nested undo ops
  - **`EndUndoMaking()`** finishes compositing current undo op – needs to be called as many times as **`StartUndoMaking()`**
- Add the undoable operation and the app will manage it

Application Preferences Management

- RenderMonkey has a number of application preferences
  - – Editable by the user from the Edit / Preferences menu
- Each plug-in can have its own property page in that dialog
- Each plug-in can save that data in the application registry file
  - – Use **`IRmRegistryManager`** interface from the application via **`GetRegistryManager()`** call

SDK Utilities

- RenderMonkey SDK provides a number of convenient classes:
  - Custom array, linked list, stl-like vector, string (with Unicode support) classes
  - Math helper functions, math vector and matrix classes and support
  - Scene graph mesh definition with hierarchical meshes
  - Image loading and an integrated image management library
  - Automatic Windows hooks utilities

MFC Utilities

- To encourage a consistent look for all plug-ins, RenderMonkey SDK provides a number of MFC widgets:
  - Numeric edit control with a popup slider
  - Color buttons / sliders / color wheel
  - Color picker widget
  - Iconic menu
  - And more…

Custom Plug-ins in the Making and Future Ideas

- Engine interface plug-in:
  - A plug-in connecting RenderMonkey and a running game engine – it can receive node database messages and reload and reapply shader in the running engine to see the finale look
- Importer / Exporter plug-in
  - Allows many developers to support their own data format
- Custom editor widgets
  - Create the look and feel consistent with your tools!
- Your imagination is the limit!

Future Work and Limitations

- Current version of the SDK doesn’t provide support for full custom node creation – we will be adding that in the future
- Plug-ins toolbar menus (future versions)
- If you want a new feature – send us a request!

  rendermonkey@ati.com

Conclusion

- RenderMonkey SDK is a flexible, powerful API for creating custom components for a great shader development IDE
- Puts the power into the hands of developers
- We hope to see many new tools on the base of this SDK!

Questions?