










## RenderMonkey SDK Version 1.82



**© 2006–2008 Advanced Micro Devices, Inc.** All rights reserved.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. (“AMD”) products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. The information contained herein may be of a preliminary or advance nature and is subject to change without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD’s Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD’s products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD’s product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

<b>OVERVIEW .....</b>	<b>5</b>
RENDERMONKEY PLUG-IN ARCHITECTURE PHILOSOPHY .....	5
IMPORTANT CHANGES WHEN PORTING EXISTING PLUG-INS .....	5
<b>GENERAL .....</b>	<b>6</b>
GENERATING A RENDERMONKEY PLUG-IN FRAMEWORK .....	6
RENDERMONKEY APPLICATION DESIGN .....	7
SUPPORTED API AND COMPATIBILITY .....	7
<b>APPLICATION AND SDK LAYOUT .....</b>	<b>8</b>
SDK LIBRARIES .....	9
<i>RmCore</i> .....	9
<i>RmUtilities</i> .....	9
<i>RmMFCUtilities</i> .....	9
<i>RmGrfxUtil</i> .....	9
SDK INCLUDES .....	9
PLUG-IN DLL ORGANIZATION .....	12
<i>RmInitPlugInDLL</i> .....	12
<i>RmGetNumPlugIns</i> .....	12
<i>RmGetPlugIn</i> .....	12
<i>RmFreePlugIn</i> .....	12
<i>RmUninitializePlugInDLL</i> .....	12
<b>APPLICATION INTERACTION.....</b>	<b>13</b>
APPLICATION ACCESS .....	13
APPLICATION MESSAGES.....	13
<i>Database Transactions</i> .....	13
<i>Application State</i> .....	15
<i>Shader Compilation</i> .....	16
<i>Viewer Plug-ins</i> .....	16
HIDDEN DATABASE NODES .....	17
<i>Workspace Editor Support</i> .....	17
<i>Registry Branches</i>  .....	18
<i>Plug-in Data Nodes</i>  .....	19
<i>Stream Channel</i>  <i>and Stream Nodes</i>  .....	19
<i>State Nodes</i>  .....	20
<i>Shader Constants</i>  .....	20
<i>Sampler Nodes</i>  .....	20
<b>USER INTERACTION .....</b>	<b>21</b>
WINDOW CREATION AND MANAGEMENT IN RENDERMONKEY .....	21
<i>Dialogs</i> .....	21
<i>Docking Windows</i> .....	21
<i>MDI Child Windows</i> .....	23
UNDO / REDO OPERATIONS .....	23
APPLICATION PREFERENCES MANAGEMENT .....	24
<b>PLUG-IN MANAGEMENT .....</b>	<b>26</b>
SUPPORTED PLUG-IN TYPES .....	26
PLUG-IN DESCRIPTION STRUCTURE .....	27
PLUG-IN INTERFACES .....	28
<i>Generic Plug-in Interface</i> .....	28

<i>Editor Plug-in Interface</i> .....	29
<i>Importer Plug-in Interface</i> .....	29
<i>Exporter Plug-in Interface</i> .....	30
<i>Geometry Loader Plug-in Interface</i> .....	30
<i>Texture Loader Plug-in Interface</i> .....	31
<i>Generator Plug-in Interface</i> .....	31
RMModule and RMModuleManager Helper Classes .....	32
<b>PLUG-IN PROJECT SETUP</b> .....	<b>33</b>
SETUP FOR VISUAL STUDIO 2005 .....	33
<i>Requirements</i> .....	33
<i>Creating a New Project</i> .....	33
<i>Project Settings</i> .....	34
<i>Project Debugging</i> .....	38
<b>SDK FEEDBACK</b> .....	<b>39</b>

## Overview

### ***RenderMonkey Plug-in Architecture Philosophy***

The entire RenderMonkey application was created using the supplied Plug-in SDK. The plug-in architecture allows the development and addition of new features to RenderMonkey, without the need to re-compile the entire application to include the new feature. Developers can create custom plug-ins to solve unanticipated problems specific to their projects, when the need arises. This also allows a convenient method for AMD to share new RenderMonkey tools in the future.

### ***Important Changes When Porting Existing Plug-Ins***

When porting over any existing plug-ins to work with RenderMonkey 1.71, the SDK version must be up-to-date for the plug-in to get loaded. The developer can use the following new built-in #defines for this purpose:

```
RENDERMONKEY_SDK_CURRENT_VERSION_MAJOR  
RENDERMONKEY_SDK_CURRENT_VERSION_MINOR
```

Plug-Ins without the current SDK version will not get loaded.

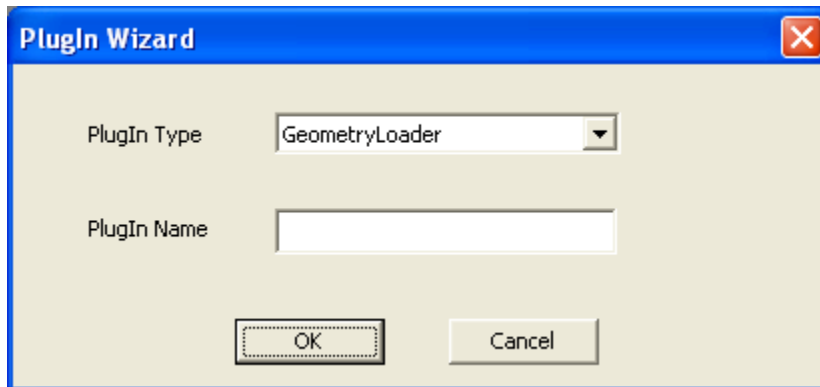
## General

### ***Generating a RenderMonkey Plug-In Framework***

RenderMonkey includes a *PlugIn Wizard* that will generate a Microsoft Visual Studio 2005 project and code for new RenderMonkey plug-in development. The *PlugIn Wizard* is available from the *Help* menu within RenderMonkey.



Selecting *PlugIn Wizard* will make a dialog box appear that allows the user to choose the type of plug-in to build (GeometryLoader, Exporter, Importer, Editor, TextureLoader, or Generator) and to specify the name of the plug-in.



Using the *PlugIn Name* as a base for the class name, the wizard will generate a project that contains all the necessary entry points for the selected *PlugIn Type*. The entry points will also contain code specific to that *PlugIn Type* that provides a framework for the necessary functionality of that plug-in. Generated plug-in projects are located in the 'RenderMonkey 1.71\SDK\Projects' directory. This document, along with comments throughout the code, will guide the user in understanding the necessary functionality of each entry point to aid in development of a working custom plug-in.

Compiling the SDK PlugIns will require that the DirectX 9 SDK from February 2007 is installed on the computer. Installing the DirectX SDK should set the *DXSDK\_DIR* environment variable to the folder that the DirectX SDK was installed to. This variable must be set to compile the generated plug-ins in Visual Studio.

## ***RenderMonkey Application Design***

RenderMonkey is a single document application. This means that only one workspace file (.rfx) can be open at a time. However, multiple instances of RenderMonkey can be run at the same time, allowing the transferring of data across separate instances.

All of the necessary data to render an effect is stores in the run-time database. All real-time changes to the database are managed by the application and propagated to the plug-ins through the plug-in messaging system. The application will send out Windows-style messages to the plug-ins message handler, allowing the appropriate behavior to be implemented by the plug-in itself. Note that all API specific rendering resources exist in the viewer plug-ins, meaning other plug-ins will have no default access to this specific data.

The run-time database consists of:

- Effect Group(s)
  - Effect(s)
    - Pass(es)
      - Render State Block
      - Pixel Shader
      - Vertex Shader
      - Tessellation Node
      - Model Reference
      - StreamMap Reference
      - Texture Object(s)

Variables, Models, Textures, and StreamMapping nodes can live at most levels of the workspace.

Please refer to RMEffect.h for full node definitions, as well as SDK\Docs\html\index.html for class hierarchy documentation.

## ***Supported API and Compatibility***

The SDK is written in pure C++. RenderMonkey version 1.71 and SDK 1.71 support plug-in development in Visual Studio 8.0 (2005). Support for Visual Studio 6.0 and Visual Studio 7.1 (.Net) is no longer available. Developers can create plug-ins using either the Win32 API, or MFC as they please. Please refer to the sample projects included with the SDK for examples of both MFC and Win32 plug-ins.

# Application and SDK Layout

RenderMonkey will install the following directories:

## 1. *Data*

The Data directory stores initialization files pertaining to API specific state information, parser / keyword information, and XML / DTD information. Generally, these files will not normally be edited by the user, and the information contained within is mostly tied to the rendering API's, as well as the RenderMonkey application itself. The one file that may be edited by the user is DefaultWorkspace.rfx. This file determines the default initialization that will accompany a newly added database node in the workspace. Please refer to the DefaultWorkspace section for a full explanation on how to use the DefaultWorkspace.rfx file.

## 2. *Examples*

The Examples directory is where example workspaces and related files are stored. The workspaces themselves are stored in the Dx9 / GL2 subdirectories, and the related texture / model files are all stored in the Media subdirectory.

## 3. *Plugins*

The Plugins directory is where the application will look for all available plug-in DLLs to load upon application startup. Any new plug-in DLL should be placed in this directory for the application to find.

## 4. *SDK*

The SDK directory will contain the necessary components for the creation of new plugIn DLLs. The following subdirectories are included:

### a. *Docs*

The Docs subdirectory contains any SDK related documentation.

### b. *Examples*

The Examples subdirectory contains example plug-in projects for both Visual Studio 6 and Visual Studio 7.1. There are example projects for most available plug-in types.

### c. *Include / Lib*

The Include and Lib subdirectories include the appropriate header (.h) and library (.lib) files that may be required for plug-in project development.



- d. *Projects*  
The Projects directory contains the plug-in projects that are generated by the Plug-In Wizard.
- e. *Wizard*  
The Wizard subdirectory contains code samples used by the plug-in wizard.

## **SDK Libraries**

The following libraries (.lib) with associated headers (.h) are included for plug-in development:

### **RmCore**

This library contains the node database definitions, plug-in interfaces, application interface, various manages interfaces, as well as custom classes. This is the main library that must be used with all plug-ins.

### **RmUtilities**

This library contains many useful utilities such as stl-like list and array classes, a string handling class, window flicker reducing utilities, window message hook utilities, dynamic control placement utilities, etc. This library can optionally be included for plug-in development.

### **RmMFCUtilities**

This library contains many useful MFC based utilities such as custom dialog controls and widgets, an iconic menu class, etc. This library can optionally be included for plug-in development.

### **RmGrfxUtil**

This library contains texture image creation and management utilities, image conversion utilities, and API specific device utilities. This library can optionally be included for plug-in development.

## **SDK Includes**

### **RmApplication.h**

Provides the definition for the `IRmApplication` interface. The `IRmApplication` interface, which is globally available through the `getRmApp()` function, provides the main access point for the runtime database as well as the user interface.

## RmArray.h

Provides a templated `RmArray` class interface. Created to eliminate the need for an STL implementation, this general interface provides standard array class functionality.

## RmCore.h

Includes all core library headers in one convenient header include file.

## RmDefines.h

Contains RenderMonkey used `#defines`. This includes plug-in Message ID's, and reserved window control ID's, and other commonly used `#defines`. Refer to this file for parameter details of RenderMonkey plug-in messages.

## RmEffect.h:

Contains definitions of the entire runtime database node structure, and related helper definitions. Refer to this file for a complete description of each node type's class definition.

## RmFile.h

Contains file helper functions to open / close a file.

## RmLinkedList.h

Contains a templated `RmLinkedList` class interface. Created to eliminate the need for an STL implementation, this general interface provides standard linked list class functionality.

## RmMath.h

Contains basic 3D math helper (`RmVector`, `RmBoundingBox` and many more)

## RmMatrix.h

Contains `RmMatrix4x4` helper class

## RmMesh.h

Contains mesh classes to help a programmer to keep information about geometry in a nicely packed manner.

## RmPlugIn.h

This file contains declarations for all supported plug-in interfaces for creation of RenderMonkey plug-ins.

### RmRegistryManager.h

`IRmRegistryManager` interface definition. The registry manager is used to manage RenderMonkey application registry (also called 'global registry') as well as workspace-specific registry branches. Each registry branch can store information about either application settings (in global registry) or about plug-ins (for example) or specific window information (window placement information, for example).

### RmStateManager.h

`IRmStateManager` interface definition. The state manager is used to manage the available render states, texture states, etc.

### RmString.h

Contains `RmString` template interface definition. This class provides a convenient interface for common string manipulation routines.

### RmString.inl

Contains `RmString` template interface implementation.

### RmStringToPtr.h

Contains `RmStringToPtrMap` class definition. Created to eliminate the need for an STL implementation, this general interface provides the commonly used mapping from a string to a pointer.

### RmTypes.h

Contains commonly used structure definitions, enumerations, etc. This file also contains the RenderMonkey recognized string versions of all runtime database node types.

### RmUndo.h

Contains helper class definitions for Undo / Redo operations. These classes are used in conjunction with `IRmApplication::StartUndoMaking` and `IRmApplication::EndUndoMaking` to perform undoable runtime database node manipulations.

### RmVariableManager.h

Contains `IVariableManager` interface definition. This interface manages lists of variables of specific types (such as RenderMonkey's predefined variables).

### RmXMLManager.h

Contains `IRmXMLManager` interface definition. This interface is used to load / save RenderMonkey xml (.rfx) files, and query the xml data for RenderMonkey specific node data.

## ***Plug-in DLL Organization***

A plug-in DLL can contain one or more plug-ins. Each plug-in DLL must implement the following entry points:

### **RmInitPlugInDLL**

This function will get called to initialize and perform any required setup for the actual DLL. This is a good place to instantiate all plug-in instances.

### **RmGetNumPlugIns**

This function returns the number of plug-ins implemented in a particular DLL.

### **RmGetPlugIn**

This function is called to retrieve a particular plug-in from the DLL, based on a zero based index.

### **RmFreePlugIn**

This function is called to free a particular plug-in from a DLL.

### **RmUninitializePlugInDLL**

This function will get called before the DLL is unloaded by the application.

## Application Interaction

All necessary data to render an effect is stored in the node database. This includes the effect nodes, pass nodes, model nodes, etc. RenderMonkey enforces node rules to ensure the database is maintained in a valid state. For example, only one vertex shader (or program) is allowed in each pass, only one model reference if allowed in each pass, etc. For further information regarding the node rules, please refer to the XML DTD, which largely mirrors the rules RenderMonkey will enforce. Also, the Workspace Editor section on the RenderMonkey user documentation also will state many of the rules surrounding node placement.

## Application Access

The `IRmApplication` interface is accessible from any plug-in through the global function `getRmApp()`. The `IRmApplication` interface is the main entry point for window creation and management. It allows users to clear output window text and to specify new text to output, and contains an instance of the current workspace (node database). Through the interface, all node transactions can be performed, including the launching of node editors.

`IRmApplication` also provides access to various manager interfaces. These manager interfaces include the application registry manager, predefined variable manager, XML manager, etc.

## Application Messages

All application events (non-Windows) and all node database transactions are propagated to the plug-ins by RenderMonkey messages. All plug-ins must support a message handler to process these messages. Please refer to the plug-in interface descriptions for details regarding this handler. Any plug-in can send out a message at any point by notifying the application through the `IRmApplication::BroadcastMessage(...)` function. Please refer to the SDK file `RmDefines.h` for a full listing of messages and supporting message data.

### Supported Transactions and Application Messages

#### Database Transactions

`RM_MSG_NODE_UPDATED`

This message will be sent when a node has experienced a structural change. This structural change could be relevant to the node itself, or any of the children. Plug-ins should

recalculate any information that may be affected by this node or any of its children. It is possible that child nodes either no longer exist, or have been added to the node itself.

RM\_MSG\_NODE\_DELETED

This message is sent to inform the plug-ins that a node is getting deleted from the database. At the point the message is broadcast, the node has been removed from the database, but the heritage information is still contained within the removed node. This allows plug-ins to react accordingly before the node is actually deleted from memory.

RM\_MSG\_NODE\_ADDED

This message gets sent out when a node has been added to the database. The added node may have accompanying child nodes, whose additions will not be notified through similar messages. The plug-ins are expected to handle these child nodes accordingly.

RM\_MSG\_NODE\_NAME\_CHANGED

This message gets sent out when a node name has changed. The message will contain the old node name.

RM\_MSG\_NODE\_VALUE\_CHANGED

This message is sent out to notify plug-ins of a simple value change within a node. Normally, value change messages are associated with numeric variable nodes.

RM\_MSG\_VARIABLE\_SEMANTIC\_CHANGED

This message is sent out to notify plug-ins of a simple value change within a node. Normally, value change messages are associated with numeric variable nodes.

RM\_MSG\_MODEL\_ORIENTATION\_CHANGED

This message is sent out to notify plug-ins that a model orientation has changed, and users of the model should reload their data.

RM\_MSG\_TEXTURE\_ORIGIN\_CHANGED

This message is sent out to notify plug-ins that a texture origin has changed, and users of the texture should reload their data.

RM\_MSG\_BEGIN\_MESSAGE\_BLOCK / RM\_MSG\_END\_MESSAGE\_BLOCK

This message is sent out to notify plug-ins that a block of related messages are about to be sent out. This is typically done then multiple changes are taking place at the same time, such as node drag and drop, etc.

Please note that with most node transactions, the plug-in must send out an application message to notify all plug-ins about the database change. This is done through the `IRmApplication::BroadcastMessage` function. This is necessary to give all plug-ins a chance to update any pointers, close editors, etc. By not informing plug-ins of database changes, RenderMonkey plug-ins will become unstable.

## Application State

`RM_MSG_APP_FILE_NEW`

Sent out to all plug-ins to notify them that a new workspace was created.

`RM_MSG_APP_FILE_OPEN`

Sent out to all open modules to notify them that a file was opened.

`RM_MSG_APP_FILE_OPEN_BEGIN / RM_MSG_APP_FILE_OPEN_COMPLETE`

Sent out to all open modules to notify them that a file open operation has started / completed.

`RM_MSG_APP_QUERY_TO_SAVE_DATA`

Sent out to all open modules before RenderMonkey saves the workspace. This is to ensure that all of the plug-ins propagate any data that they may have into the run-time database before it's saved out to XML. The plug-ins may query the user at this point to whether the data should, or should not, get saved.

`RM_MSG_APP_SAVE_DATA`

Sent out to all plug-ins before RenderMonkey saves the workspace to make sure that the plug-ins propagate any data that they may have into the run-time database before it's saved out to XML. The plug-ins should not query the user whether the data should be saved, but just propagate it directly.

`RM_MSG_APP_SAVE_DATA_BEGIN / RM_MSG_APP_SAVE_DATA_COMPLETE`

The saving data operation has started / finished.

`RM_MSG_APP_FILE_CLOSE`

Sent out to all plug-ins to notify them that a workspace file is about to be closed.

`RM_MSG_APP_FILE_CLOSE_BEGIN / RM_MSG_APP_FILE_CLOSE_COMPLETE`

Sent out to all plug-ins to notify them that a file is about to be closed and the operation has started / finished.

RM\_MSG\_APP\_CLOSING

Sent out to all open modules to notify them that the application is about to close.

RM\_MSG\_APP\_SILENT\_BEGIN / RM\_MSG\_APP\_SILENT\_COMPLETE

Sent out to all open modules to notify them that the application may not / may display any dialog boxes that block, waiting on user input.

## Shader Compilation

RM\_VIEW\_COMPILE\_SHADERS

This message forces compilation of all modified ("dirty") shaders, usually in the active effect.

RM\_VIEW\_COMPILE\_SHADERS\_BEGIN /  
RM\_VIEW\_COMPILE\_SHADERS\_COMPLETE

This message notifies the plug-ins that shader compilation has started / finished.

## Viewer Plug-ins

RM\_VIEW\_CHANGE\_ACTIVE\_EFFECT

This message notifies the viewer plug-in that the active effect has changed. The viewer plug-in should prepare all appropriate resources and begin rendering the new active effect.

RM\_VIEW\_UPDATE\_TEXTURES / RM\_VIEW\_UPDATE\_MODELS

This message tells the viewer to reload all textures / models in the active effect.

RM\_VIEW\_UPDATE\_RENDER\_TEXTURE\_CONTENTS

This message requests that the active viewer update the contents of a given renderable texture.

RM\_VIEW\_UPDATE\_ALL



This message gets sent to request that the viewer recreate all rendering related resources. This is a global graphics update for the active effect.

RM\_VIEW\_RESET

This message gets sent to request that the viewer resets to default positioning and orientation.

RM\_VIEW\_SET\_UI\_MODE

This message gets sent to change the view's UI mode (Rotate, Pan, etc).








Note that these viewer related messages can be triggered by any plug-in that wishes to update the rendering of the active effect.

RM\_VIEW\_RENDER\_SNAPSHOT

This message is sent out to request that the active viewer save the current frame to disk.

## ***Hidden Database Nodes***

Database nodes that are normally hidden are:

1. Registry Branches ()
2. Plug-in Data Nodes ()
3. Stream Channels ()
4. Streams ()
5. States ()
6. Shader Constants ()
7. Samplers ()

These nodes are normally added and modified through the various RenderMonkey SDK interfaces, and do not come with default editors.

## **Workspace Editor Support**

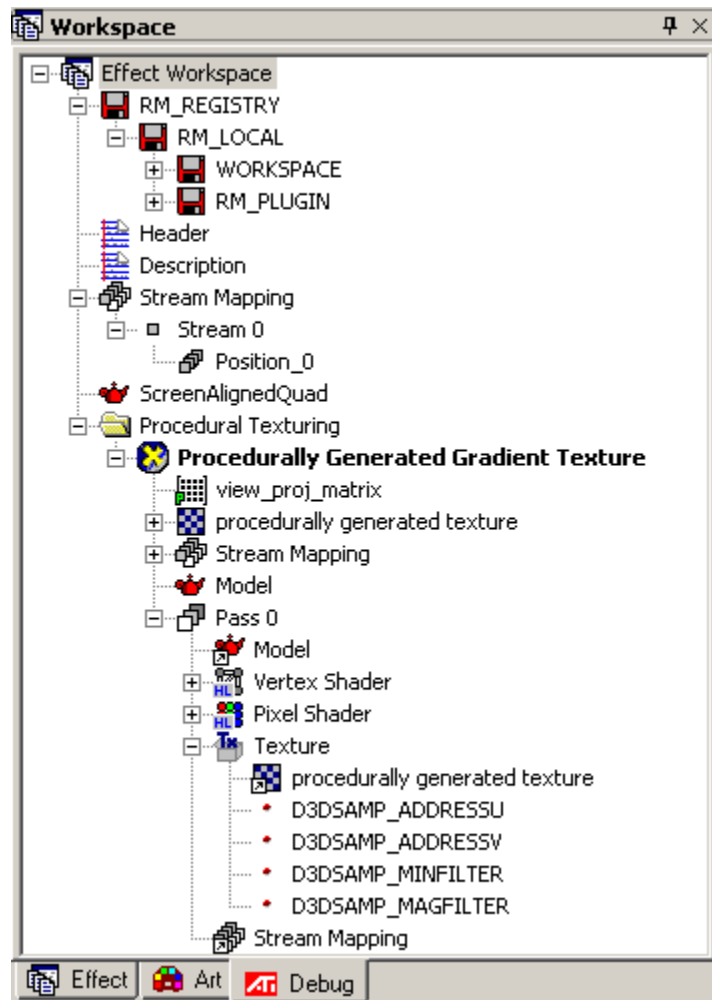
The workspace editor is a dock-able window usually positioned on the left of the main interface containing a tabbed tree control which provides a high level view of the effect database.

This editor can be used to access all elements, or nodes, in the workspace. The individual effects can be grouped by their common attributes in the workspace as seen fit by the user (either by rendered effects style, or by the fallback paths, or by rendering API).


There are normally two tabs in the workspace editor: The *Effect* tab (🔧) and the *Art* tab (🎨). The *Effect* tab (🔧) is used to view and modify the entire workspace – with all variable and pass nodes visible. The *Art* tab (🎨) is used to view only the artist-editable variables that are present in the workspace. The *Art* tab will only allow the user to edit artist-editable nodes, without the ability to add, delete, rename, etc. This functionality allows the programmers to develop the full effect and then allow the artists to modify the effect's rendering output without worrying about accidentally modifying the effect's contents.

The workspace editor also contains a normally hidden tab called the *Debug* tab (🔍). Invoked, or revoked, through the IRmApplication interface function `SetDebugMode (...)`, the Debug tab will show all workspace nodes, including the normally hidden nodes.

Example:




## Registry Branches 📁

Registry Branch nodes () are used to add local or global plug-in specific data into the workspace. Normally, these nodes are used as root nodes of various child nodes holding plug-in specific data. The local or global registry branches are available through the `IRmRegistryManager` interface, using the appropriate `GetLocalRegistryRootPlugInBranch()` or `GetGlobalRegistryRootPlugInBranch()` functions. The plug-in should never modify or change registry branches (or associated child nodes) it doesn't own, as to not cause unpredictable behavior. Normally, a plug-in will add a single registry branch (named after the plug-in) to the root plug-in branch obtained from the `IRmRegistryManager`. Working from that child branch, the plug-in is then free to modify the child nodes as necessary.

Registry Branch nodes may contain the following child nodes:

1. Registry Branches
2. Integer Variables
3. ~~/0/1~~ Boolean Variables
4. Float Variables
5. Notes







## Plug-in Data Nodes

Plug-in Data nodes () are used in conjunction with generator plug-ins, to store the data required to regenerate the node data upon request. Normally, these nodes are used as root nodes of various child nodes holding plug-in specific data. The plug-in should never modify or change plug-in data nodes (or associated child nodes) it doesn't own, as to not cause unpredictable behavior. A plug-in may add a plug-in data node (named after the plug-in) to the appropriate node, but only one plug-in data node is allowed for any given node. The plug-in data node must supply the appropriate `RmPlugInID` to the node to ensure the associated generator plug-in will receive the call to re-generate the node data when necessary through the `IRmGeneratorPlugIn::GenerateData(...)` function.

Registry Branch nodes may contain the following child nodes:





1. Integer Variables
2. ~~/0/1~~ Boolean Variables
3. Float Variables
4. Notes

## Stream Channel and Stream Nodes






Stream Channel () and Stream () nodes are directly manipulated through the standard stream mapping editor. The stream () node is only found as a child of the stream mapping node (). The stream channel node () is only allowed as a child of a stream node (). It is recommended that these nodes are not manipulated by another plug-in. The only case



where these nodes should be directly manipulated is if a node has become corrupted, and is not accessible through the stream mapping editor. In this case, the corrupted node may be deleted through use of the workspace editor debug tab.

## State Nodes






State nodes () are directly manipulated through the render state or texture state editors. State nodes () are only allowed as children of a render state block () or a texture object (). It is recommended that these nodes are not manipulated by another plug-in. The only case where these nodes should be directly manipulated is if a node has become corrupted, and is not accessible through the appropriate state editor. In this case, the corrupted node may be deleted through use of the workspace editor debug tab.



## Shader Constants

Shader Constants () are added through two methods. The first method is indirectly by the user, when a variable node () is dropped onto a shader () node in the workspace editor. The second method is through the preview windows during compile operations. Shader constants () are only allowed as children of shader nodes (). The only case where these nodes should be directly manipulated is if a node has become corrupted, and is not removed when the affected shader is compiled. In this case, the corrupted node may be deleted through use of the workspace editor debug tab.

If a shader constant is invalid, it will be shown with a slash (  ) through it. Example: . The cause of this is if the shader constant cannot find the associated variable node it is associated with.

## Sampler Nodes

Sampler Nodes () are added through two methods. The first method is indirectly by the user, when a texture object node () is dropped onto a shader () node in the workspace editor. The second method is through the preview windows during compile operations. Sampler nodes () are only allowed as children of shader nodes (). The only case where these nodes should be directly manipulated is if a node has become corrupted, and is not removed when the affected shader is compiled. In this case, the corrupted node may be deleted through use of the workspace editor debug tab.

If a sampler node is invalid, it will be shown with a slash (  ) through it. Example: . The cause of this is if the sampler node cannot find the associated texture object node it is associated with.

## User Interaction

### ***Window Creation and Management in RenderMonkey***

The following window creation methods exist to ensure compatibility between the Win32 / MFC APIs, as well as compatibility between compiler versions.

When a window is registered with the application through `IRmApplication::RegisterWindow(...)`, the plug-in must be sure to un-register the same window upon destruction with the `IRmApplication::UnregisterWindow(...)` function.

Note that multiple window registrations are sometimes required to ensure the proper handling of nested window message routing. For examples of all window types, please refer to the SDK projects. These projects illustrate the proper methods to properly manage plug-in windows of every type.

Please refer to `SDK/Include/Core/RmApplication.h` for details on the `IRmApplication::RegisterWindow / UnregisterWindow` functions.

Please note that it is up to the plug-in to take appropriate action when a node associated with a window is modified or deleted. The plug-in must handle the appropriate measures such as closing an open window, updating any affected data, etc.

The following window types are supported:

### **Dialogs**

To provide proper handling of dialog messages, the plug-in must register the created dialog window using the `IRmApplication::RegisterWindow(...)` function. This will ensure the appropriate window messages get routed to the plug-in created dialog. The registration of dialogs applies to both stand alone dialog windows, as well as dialogs created as children with docking and MDI child windows.

### **Docking Windows**

The following functions are used when creating and managing docking windows:

- a. `IRmApplication::CreateDockingWindow`

This function will create a docking window frame that a plug-in can then attach to.

b. `IRmApplication::FindDockingWindow`

This function will search for an existing docking window (which may be in a hidden state). If the function successfully returns a window handle, the plug-in is to attach to the existing window, instead of creating a new docking window.

c. `IRmApplication::GetAvailableDockingWindowID`

If a docking window has not been created, the plug-in must request a docking window ID from the application. This ID is to be saved by the plug-in, as this ID will be used when loading a workspace to ask the plug-in to re-create the window when required. The application asks for this creation through the `RM_MSG_DOCKPANE_CREATED` application message.

d. `IRmApplication::ShowDockWindow`

If a plug-in wishes to close the docking window, or temporarily hide the docking window, calling this function will cause the docking window to disappear (or reappear, if called upon).

As with dialogs, the window attached to a docking frame must register itself to RenderMonkey through the `IRmApplication::RegisterWindow(...)` function.

Standard procedure for managing docking windows is as follows:

1. If creating a new docking window, the plug-in must store the returned ID from the `IRmApplication::GetAvailableDockingWindowID( ... )` function. This ID will be used in all subsequent management of the created window.
2. If the plug-in receives an application message `RM_MSG_DOCKPANE_CREATED`, and its stored ID matches that passed in the message, it is up to the plug-in to create the associated window and attach it to the already created docking frame.
3. If the plug-in has a stored ID already, and wishes to activate the associated window, it should first call `IRmApplication::FindDockingWindow(...)` to check if the docking frame exists already in a hidden state. If it does not yet exist, then a call to `IRmApplication::CreateDockingWindow(...)` will create a docking frame with the desired ID.
4. Destruction of a docking window is done through a call to `IRmApplication::ShowDockWindow(...)`. The application will handle the appropriate destruction / hiding of the associated docking frame.

The docking frame ID's are important! Proper management of the ID's is a must for complete docking frame handling. Please refer to the docking editor SDK projects for examples on managing docking windows. Please note that the docking frame must be registered as well as any attached child windows such as dialogs.

## **MDI Child Windows**

Calling `IRmApplication::CreateMDIChildFrame(...)` will create a child MDI frame that the plug-in can use to attach its own windows to. Minimally, the plug-in must register the returned MDI child frame with the application, and un-register the same handle upon destruction. With MDI windows, the plug-in must be sure to use the appropriate window messages when creating (`WM_MDIACTIVATE`) or destroying (`WM_MDIDESTROY`) the returned MDI child frame.

## ***Undo / Redo Operations***

RenderMonkey allows developers create their own complex undoable operations. These operations can be nested, and will be fully managed by RenderMonkey once added to the undo stack.

To start making an undo operation, or to nest an additional undo operation within an existing operation, call `IRmApplication::StartUndoMaking(...)`. Once the undoable operation has been completed, calling

`IRmApplication::EndUndoMaking()` will finish wrapping the undo operation initiated by the innermost call to `IRmApplication::StartUndoMaking(...)`. Each call to `IRmApplication::StartUndoMaking(...)` must have a companion `IRmApplication::EndUndoMaking()` to close the scope at each nesting level.

RenderMonkey supplies the following undo operations by default:

`RmAddNodeUndoOp`

This undo operation will handle the undoing and redoing of adding a new node to the database. Push an instance of this class onto the undo stack after the new node is added to the database.

`RmUpdateNodeUndoOp`

This undo operation will handle the undoing and redoing of a general node update within the database. Push an instance of this class onto the undo stack before the node is update in the database.

`RmRenameNodeUndoOp`

This undo operation will handle the undoing and redoing of a node name change within the database. Push an instance of this class onto the undo stack before the node name has actually changed within the database.

`RmDeleteNodeUndoOp`

This undo operation will handle the undoing and redoing of a node deletion within the database. Push an instance of this class onto the undo stack after the node is removed from the database, but before the node is actually deleted from memory.

`RmChangeActiveEffectUndoOp`

This undo operation will handle the undoing and redoing the changing of the active effect. Push an instance of this class onto the undo stack either before or after the active effect has been changed.

`RmSemanticChangeUndoOp`

This undo operation will handle the undoing and redoing the changing of the predefined variable semantic. Push an instance of this class onto the undo stack before the variable semantic has changed. The new semantic must be provided upon construction.

`RmModelOrientationChangeUndoOp`

This undo operation will handle the undoing and redoing of a model orientation change. Push an instance of this class onto the undo stack before the orientation has changed. The new orientation must be provided upon construction.

`RmTextureOriginChangeUndoOp`

This undo operation will handle the undoing and redoing of a texture origin change. Push an instance of this class onto the undo stack before the origin has changed. The new origin must be provided upon construction.

## **Custom Undo Operation Creation**

To create a custom undoable operation, derive the appropriate class from the `RmUndoOp` base class, and pass an instance of that class as the parameter to the `IRmApplication::StartUndoMaking(...)` function. Overriding the virtual functions `RmUndoOp::Undo()` and `RmUndoOp::Redo()` to perform the custom operations.

## ***Application Preferences Management***



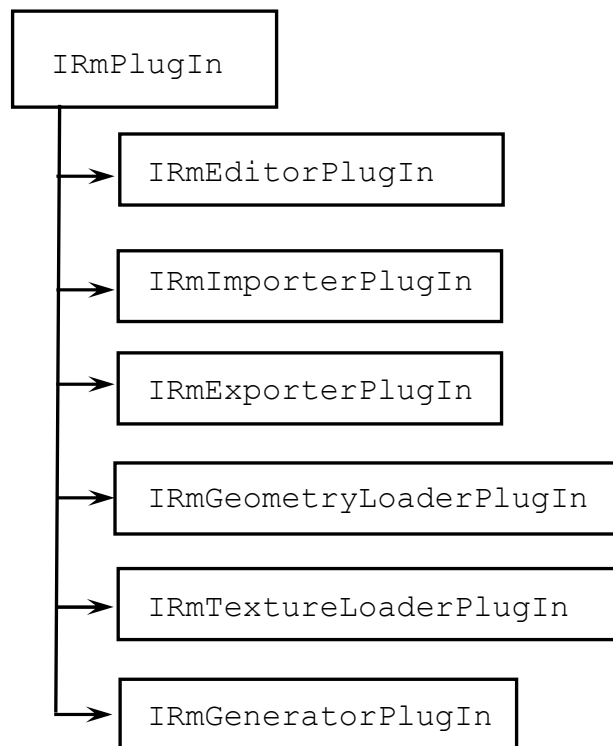
As indicated in the `IRmPlugIn` plug-in interface, RenderMonkey provides plug-ins the ability to create an application property page. These property pages are available through the main menu, by selecting Edit->Preferences. The associated data can be stored in the application registry file. To gain access to the application registry, use the `IRmRegistryManager` interface. This interface is available through the `IRmApplication` interface using the `GetRegistryManager()` method. Once the appropriate registry node (`RmRegistryBranch`) has been attained, normal node transactions can be used to save the related data.

## Plug-in Management

Upon the application startup, all plug-in DLL's are loaded from the \plugins directory, and stored internally. All plug-ins are managed according to their type, and will be automatically associated with the appropriate nodes through the workspace tree context menu.

The user can create multiple plug-ins (editors, generators, etc.) for the same node type. The application will simply provide the appropriate list when called upon to do so.

### *Supported Plug-in Types*



IRmPlugIn

Generic RenderMonkey plug-in interface designed to receive main communication messages from the main application and to specify a property page dialog for main application preferences. A creator of this plug-in type must specify `RM_PLUGIN_TYPE_GENERIC` as the plug-in type value in the plug-in description.

IRmEditorPlugIn

A node editor plug-in. This plug-in is used by the main application to edit nodes supported

by this plug-in. Use plug-in description to specify which nodes can be edited by this plug-in in the workspace. The plug-in must return `RM_PLUGIN_TYPE_EDITOR` in its plug-in description structure.

`IRmImporterPlugIn`

A plug-in implementing this interface is used by the main application to import data from external formats into the run-time database. An importer plug-in must use `RM_PLUGIN_TYPE_IMPORTER` in its plug-in description structure.

`IRmExporterPlugIn`

A plug-in implementing this interface is used by the main application to export data from the run-time database into an external format. An exporter plug-in must use `RM_PLUGIN_TYPE_EXPORTER` in its plug-in description structure.

`IRmGeometryLoaderPlugIn`

This plug-in type is used by the main application to import geometry data from a supported file format into RenderMonkey model data node. The plug-in must use `RM_PLUGIN_TYPE_GEOMETRY_LOADER` in its plug-in data description structure.

`IRmTextureLoaderPlugIn:`

This plug-in type is used by the main application to import texture data from a supported file format into RenderMonkey texture data node. The plug-in must use `RM_PLUGIN_TYPE_TEXTURE_LOADER` in its plug-in data description structure.

`IRmGeneratorPlugIn`

This plug-in type is used by the main application to generate data for a RenderMonkey node. The plug-in must use `RM_PLUGIN_TYPE_GENERATOR` in its plug-in data description structure.

Please refer to the SDK example projects for plug-in implementation examples.

## ***Plug-in Description Structure***

One of the key components of a plug-in is the plug-in description structure. When a plug-in is loaded into the application, the application will extract the necessary information from this structure, allowing the appropriate node and type associations to take place.

Elements of the plug-in description structure include:

1. Plug-in type
2. Plug-in ID
3. A list of supported node types
4. SDK version
5. Supported rendering API (DX, OpenGL, API-agnostic)
6. Plug-in name

The following plug-in types are supported:

1. Generic Plug-in
2. Editor Plug-in
3. Importer Plug-in
4. Exporter Plug-in
5. Geometry Loader Plug-in
6. Texture Loader Plug-in
7. Generator Plug-in

## ***Plug-in Interfaces***

### **Generic Plug-in Interface**

A generic plug-in is indicated by setting the plug-in type (in the plug-in description struct) to `RM_PLUGINTYPE_GENERIC`. This plug-in, as do all plug-in types, must implement the `IRmPlugIn` interface.

The `IRmPlugIn` interface provides the basic mechanism for initialization / un-initialization, as well as the method for receiving of messages from the application. It also gives the plug-in the ability to add its own property page to the application preferences property dialog. A plug-in may choose to act upon an individual node, or upon a group of nodes.

The `IRmPlugIn` interface contains the following entry points:

- `Init/Uninitialize`

These functions are to provide plug-in initialization / un-initialization.

- `GetPlugInDescription`

This function is to retrieve the plug-in description structure.

- `MessageHandler`

This function is to provide application message processing.

- `HadPropertyDlg / AddPropertyDlg`

These functions provide the ability to add a plug-in preference page in the application preferences dialog.

## Editor Plug-in Interface

An editor plug-in is meant to provide an intuitive interface where the user can edit a single, or a group or related nodes. Examples of editor plug-ins include the color node editor, shader editor, and vector editor. An editor plug-in is indicated by setting the plug-in type (in the plug-in description struct) to `RM_PLUGIN_TYPE_EDITOR`. This plug-in must implement the `IRmEditorPlugIn` interface, which is derived from the `IRmPlugIn` interface.

The `IRmEditorPlugIn` interface contains the following entry point:

- `EditNode`

This function is intended to launch the appropriate editor for the appropriate node, or group of nodes.

## Importer Plug-in Interface

An importer plug-in is meant to provide a method to import external data into the RenderMonkey database. An example of an importer plug-in is the package importer. An importer plug-in is indicated by setting the plug-in type (in the plug-in description struct) to `RM_PLUGIN_TYPE_IMPORTER`. This plug-in must implement the `IRmImporterPlugIn` interface, which is derived from the `IRmPlugIn` interface.

Importer plug-ins can be used to parse custom engine scripts or data files, providing a mechanism where different custom datasets can be easily transferred across to work within RenderMonkey.

The `IRmImporterPlugIn` interface contains the following entry point:

- `ImportNode`

This function is intended to launch the appropriate importer, allowing the user to select the external data source intended for conversion and eventual use within RenderMonkey.

## Exporter Plug-in Interface

An exporter plug-in is meant to provide a method to export internal data from the RenderMonkey database for possible use within another application. An example of exporter plug-ins are the package importer, and the fx exporter. An exporter plug-in is indicated by setting the plug-in type (in the plug-in description struct) to `RM_PLUGIN_TYPE_EXPORTER`. This plug-in must implement the `IRmExporterPlugIn` interface, which is derived from the `IRmPlugIn` interface.

The `IRmExporterPlugIn` interface contains the following entry point:

- `ExportNode`

This function is intended to launch the appropriate exporter, allowing the user to convert internal RenderMonkey data into an external format for use outside of RenderMonkey.

## Geometry Loader Plug-in Interface

A geometry loader plug-in is meant to provide a method to load external model files and convert the data into a RenderMonkey compatible format. Examples of geometry loader plug-ins are 3DS, X, and OBJ file loaders. The loader is invoked whenever a user selects a file to load geometry for a model node. A geometry loader plug-in is indicated by setting the plug-in type (in the plug-in description struct) to `RM_PLUGIN_TYPE_GEOMETRY_LOADER`. This plug-in must implement the `IRmGeometryLoaderPlugIn` interface, which is derived from the `IRmPlugIn` interface.

The `IRmGeometryLoaderPlugIn` interface contains the following entry points:

- `GetSupportedExtensions`

This method is used to query a geometry loader plug-in about the file types that it supports for importing geometry models.

- `CanLoadGeometry`

This method lets the main application know whether the plug-in that implements the geometry loader interface can load a specified file described by the given file name.

- `LoadGeometry`

Load geometry object(s) from the given.

## Texture Loader Plug-in Interface

A texture loader plug-in is meant to provide a method to load external texture files and convert the data into a RenderMonkey compatible format. An example of a texture loader plug-in is the image loader. The loader is invoked whenever a user selects a file to load a texture for a texture node. A texture loader plug-in is indicated by setting the plug-in type (in the plug-in description struct) to `RM_PLUGIN_TYPE_TEXTURE_LOADER`. This plug-in must implement the `IRmTextureLoaderPlugIn` interface, which is derived from the `IRmPlugIn` interface.

The `IRmTextureLoaderPlugIn` interface contains the following entry points:

- `GetSupportedExtensions`

This method is used to query a texture loader plug-in about the file types that it supports for importing texture files.

- `CanLoadTexture`

This method lets the main application know whether the plug-in that implements the texture loader interface can load a specified file described by the given file name and texture type.

- `LoadGeometry`

Called to load a texture from the given file.

## Generator Plug-in Interface

A generator plug-in is meant to provide the ability to populate the node database based on specified parameters in the plug-in. This is to provide a method of procedural geometry or texture creation. Examples of generator plug-ins include the geometry generator and the texture generator plug-ins. A generator plug-in is indicated by setting the plug-in type (in the plug-in description struct) to `RM_PLUGIN_TYPE_GENERATOR`. This plug-in must implement the `IRmGeneratorPlugIn` interface, which is derived from the `IRmPlugIn` interface.

The `IRmGeneratorPlugIn` interface contains the following entry point:

- `GenerateData`

This method is called to generate contents for the specified input node.

## ***RmModule and RmModuleManager Helper Classes***

Each plug-in can have multiple instances of the actual plug-in associated with various nodes. For example, if we take the vector editor plug-in, it may have multiple instances (modules) existing at the same time if the user selected to edit multiple (say, 5) vectors at the same time. The developer implementing the plug-in may wish to manage individual instances in any way they choose, however, the RenderMonkey SDK provides a convenient method of managing these individual modules within a plug-in by using `RmModule` and `RmModuleManager`.

The `RmModuleManager` class will help manage a group of registered `RmModule` derivative classes. Each `RmModule` derivative class helps organize modules that are associated with a single, or a group of nodes. For instance, one module may support the editing of all render state blocks within the same effect. In this case, if the user is using our editor to edit a render state block, the module manager will search to see if an existing module is available to handle the editing of that node. If there is no existing module to handle the editing of that node, the plug-in should then create a new one, otherwise the module manager will direct the editing of the node to the existing module.

These classes are defined in the SDK header file `RmPlugIn.h`



## Plug-in Project Setup

If the user would like to use the included RenderMonkey SDK MFCUtilities library, which is useful for GUI development, then the developer must compile and link to the VS8.0 MFC libraries.

### *Setup for Visual Studio 2005*

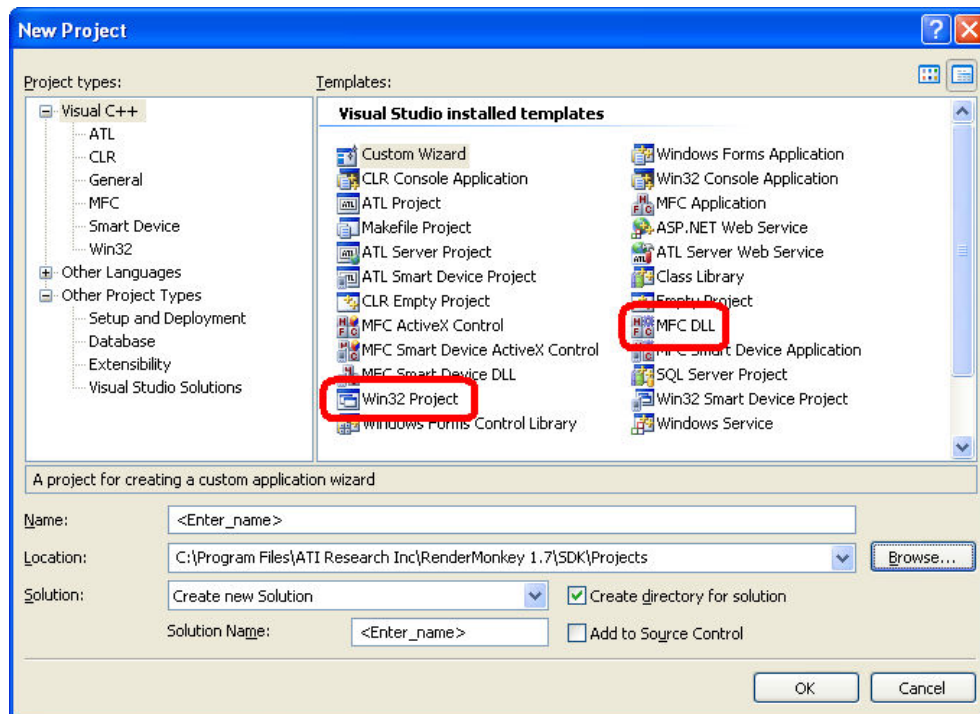
#### Requirements

- a. RenderMonkey version 1.71 must be installed with the SDK. Make a note of the directory it is installed into.
- b. You must have Visual Studio 2005 installed

#### Creating a New Project

- a. File -> New -> Project

Select “MFC DLL” ( if you want to use MFC ) or “Win32 Project” otherwise.

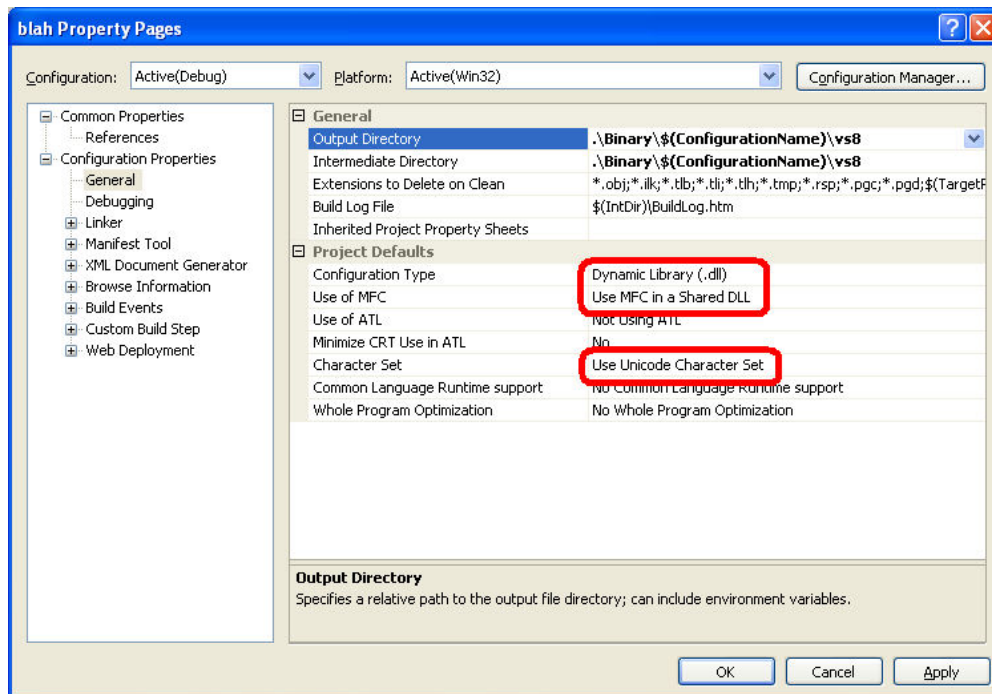


## Project Settings

- a. Project -> Properties
- b. *Configuration Properties->General Properties*

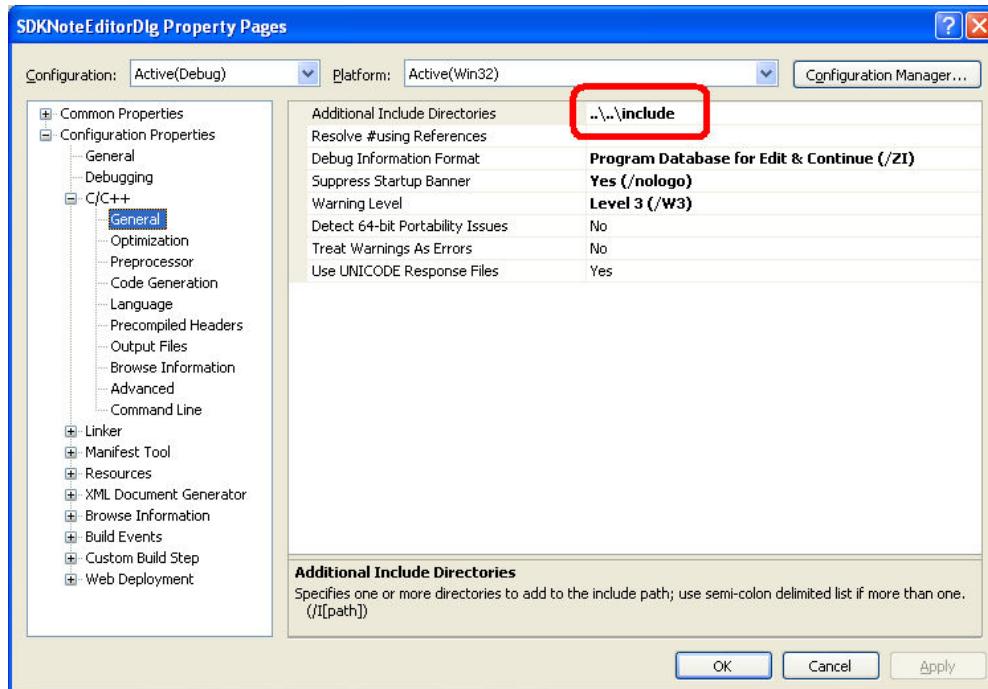
*Configuration Type* -> “Dynamic Library  
*Character Set* -> “Use Unicode Character Set”

If you want to use MFC,  
set *Use of MFC* to “Use MFC in shared DLL”



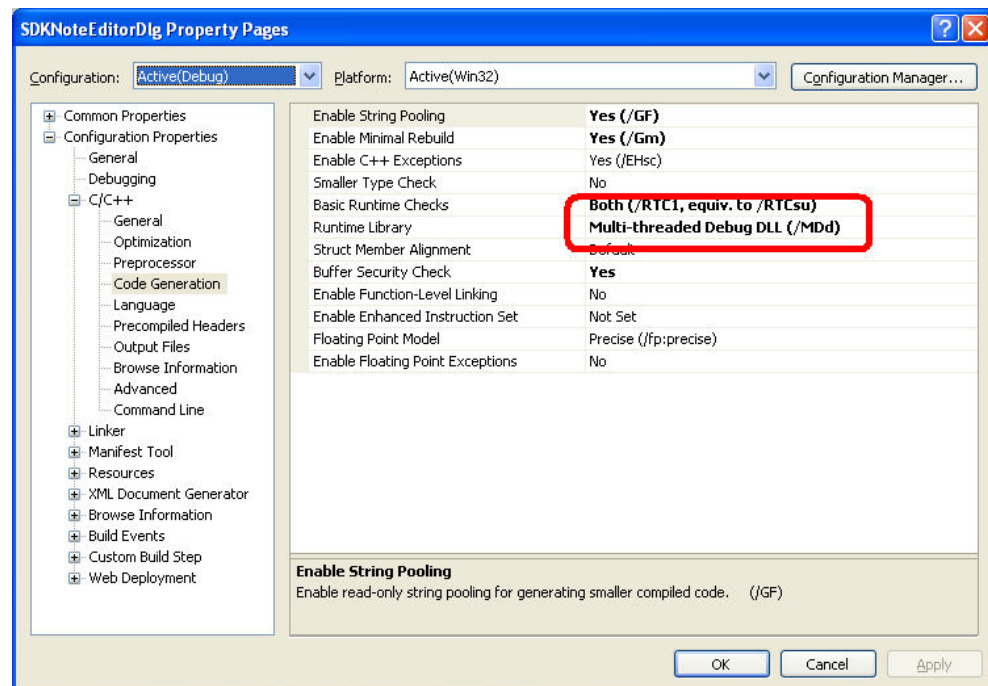
- c. *Configuration Properties->C/C++->General Properties*

*Additional Include Directories* must contain RenderMonkey’s SDK include directory. (<MyRenderMonkeyInstallDirectory>/SDK/Include). This path can be an absolute path or a relative path as shown below:



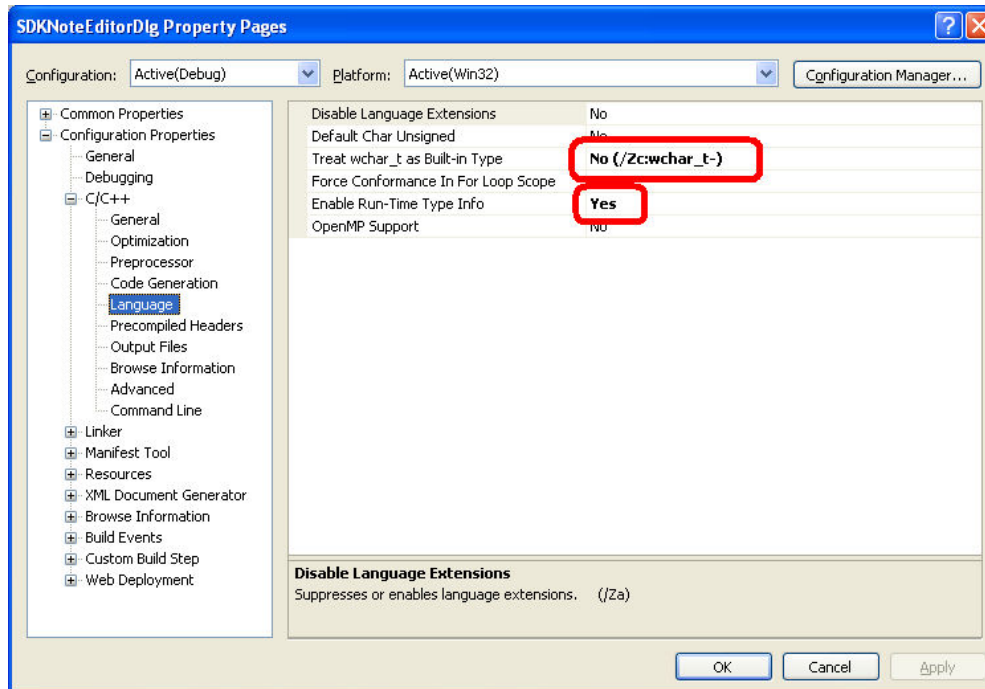
d. *Configuration Properties->C/C++->Code Generation Properties*

*Runtime library* must be set to “Multi-threaded Debug DLL” or “Mutli-threaded DLL” depending on the build configuration selected.



e. *Configuration Properties-> C/C++->Language Properties*

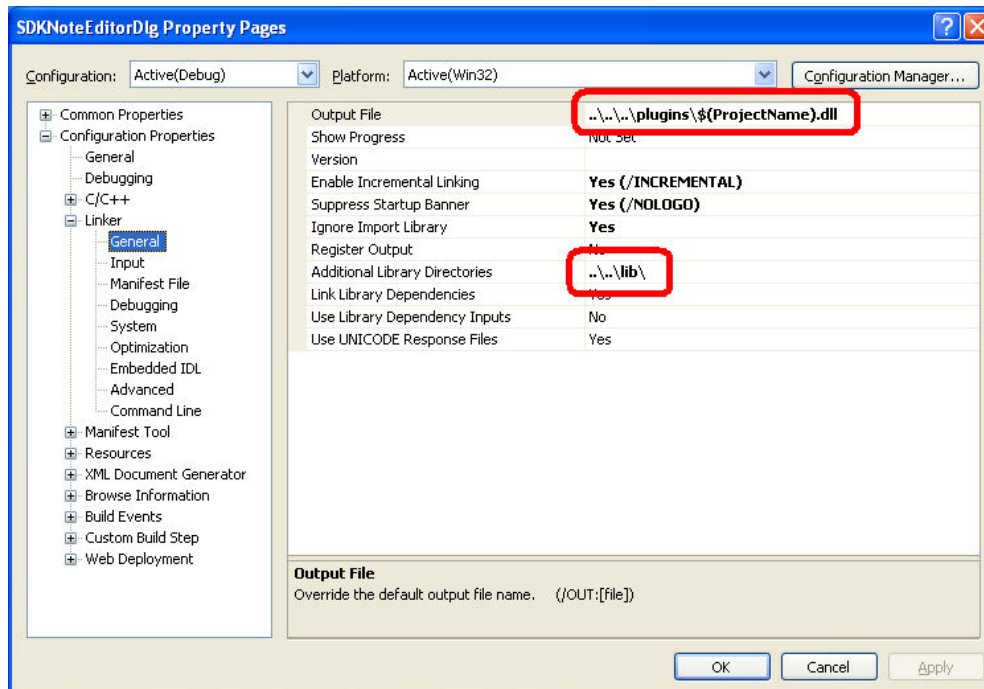
*Treat wchar\_t as Built-in type* must be set to “No”  
*Enable Run-Time Type Info* must be set to “Yes”



f. *Configuration Properties-> Linker->General Properties*

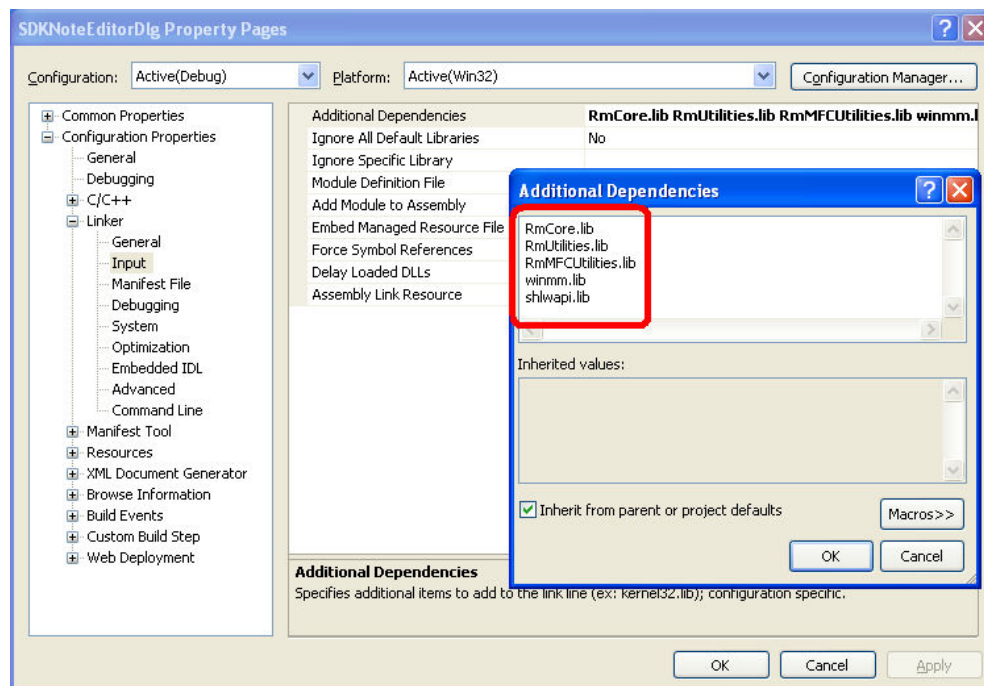
*Output File* must be set to output to RenderMonkey’s plug-ins directory (<MyRenderMonkeyInstallDirectory>/plugins). This path can be an absolute path or a relative path as shown below.

*Additional Library Directories* must include RenderMonkey’s SDK library (<MyRenderMonkeyInstallDirectory>/SDK/lib). This path can be an absolute path or a relative path as shown below.



g. Configuration Properties->Linker->Input Properties

*Additional Dependencies* must include RmCore.lib and RmUtilities.lib. If your plug-in uses MFC, you should also add RmMFCUtilities.lib.

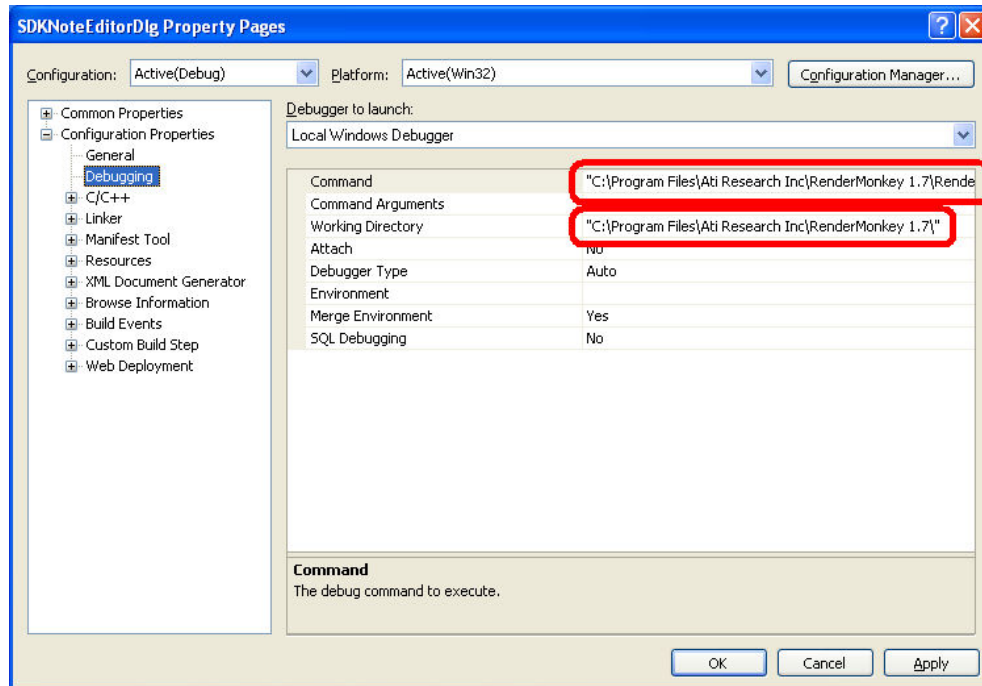


## Project Debugging

Under Configuration Properties -> Debugging,

Set *Command* to point to RenderMonkey.exe (which is found in directory where you installed RenderMonkey).

Set *Working Directory* to RenderMonkey's installed directory (the same directory as above).



## SDK Feedback

This SDK is provided for the convenience of developers to allow them to extend existing RenderMonkey framework to suit their needs. Please send us any feedback about this SDK in regards to the usefulness of its features as well as about completeness of its API. Any additional feature requests or bug reports are always welcome. We look forward to hearing from you about the types of plug-ins that you will create and anything else that you may wish to share with us!

Please send feedback to [gputools.support@amd.com](mailto:gputools.support@amd.com)