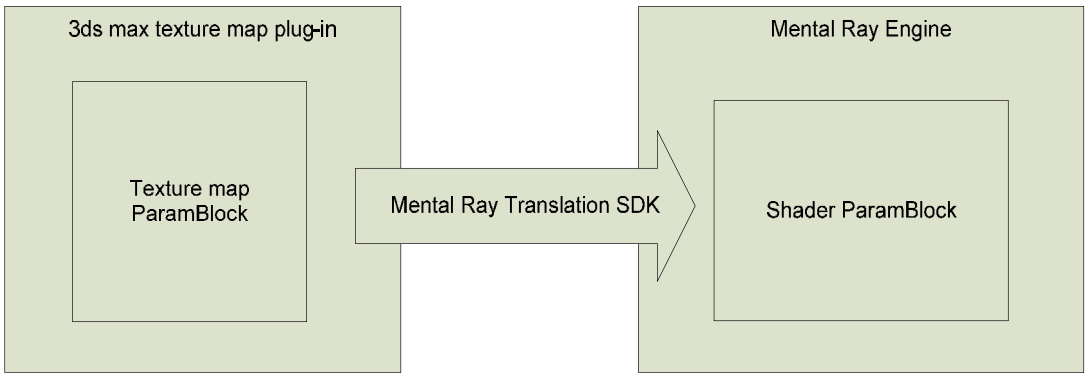
mental ray for 3ds max 7 – Software Development Kit

by Daniel Lévesque and Claude Robillard, Discreet

Introduction

3ds max ships with an integrated version of the mental ray renderer. The mental ray for 3ds max SDK is meant to extend the 3ds max SDK with access to the mental ray shaders. Its main goal is to enable 3rd party developers to:

- Access mental ray shaders and shader declarations from 3ds max plugins.
- Provide custom translation of their 3ds max plugins into mental ray shaders.



Overview of Classes

This section provides a brief description of each of the classes that are included in the SDK. More detailed descriptions about the classes and their methods are found in the header files.

Fundamental Interfaces

These abstract classes form the base of the mental ray integration in 3ds max.

class imrShader

Not to be implemented by 3rd party developers.

This is the base interface common to all mental ray shader plugins. In 3ds max, all mental ray shaders either derive from class Mtl or from class Texmap. Material phenomena derive from class Mtl (and are thus material plugins), while all other shader types derive from class Texmap (and are thus texture map plugins).

This interface provides basic access to the parameters of the shader.

The following function can be used to retrieve a pointer to this interface:

```
imrShader* GetIMRShader(InterfaceServer* iserver)
```

class imrPreferences

Not to be implemented by 3rd party developers.

This interface provides access to the mental ray preferences, which are also accessible in the “mental ray” tab of the 3ds max preferences dialog.

class imrShaderClassDesc

Not to be implemented by 3rd party developers.

The class descriptors of all mental ray shaders derive from this class. It provides access to the parameter block descriptors for each mental ray shader found by 3ds max. It also provides access to the apply types of the shader, and exposes a method for creating an instance of this shader.

To access the class descriptor of a shader, call the following method (found in class imrShader):

```
imrShaderClassDesc& imrShader::GetClassDesc()
```

class imrMaterialCustAttrib

Not to be implemented by 3rd party developers.

This interface is implemented by the “mental ray connection” material custom attributes. This custom attribute is assigned to every material in the material editor whenever the “enable mental ray extensions” preference is ON. This interface provides SDK-level access to what is already accessible through the user interface.

class IMtlRender_Compatibility_MtlBase

This class is part of the SDK of 3ds max 6.0 – not of the mental ray SDK. See maxsdk\include\IMtlRender_Compatibility.h.

This class is used to detect compatibility between a material/texmap and a renderer. By default, all material and texmap plugins are seen as compatible with the max scanline renderer, but not with mental ray. Unless you check the “Show Incompatible” checkbox in the material/map browser, only compatible materials/maps will be visible.

Custom Shader Translation Interfaces

These are the interfaces used in the custom translation system. They allow 3rd party developers to handle the translation of various 3ds max elements into mental ray shaders.

class imrShaderTranslation

To be implemented by 3rd party developers.

This is the fundamental interface used for the custom translation of a texture map plugin to a mental ray shader. A more detailed description of custom translation is included in this document.

class imrMaterialPhenomenonTranslation

To be implemented by 3rd party developers.

This interface extends class imrShaderTranslation. It is the fundamental interface used for the custom translation of a material plugin to a mental ray material phenomenon. A more detailed description of custom translation is included in this document.

class imrGeomShaderTranslation

To be implemented by 3rd party developers.

This interface is used to translate geometric objects (deriving from either GeomObject or ShapeObject) into mental ray geometry shaders. A more detailed description of custom geometry shader translation is included in this document.

class imrShaderCreation

Not to be implemented by 3rd party developers.

A pointer to this class is passed to class imrShaderTranslation to give 3rd party developers an easy way to create instances of mental ray shaders based on the name of the shader, as declared in the .mi file.

class imrShaderTranslation_ClassInfo

To be implemented by 3rd party developers.

This interface allows 3rd party developers to extend the class descriptors, for their texture map plugins, with “apply type” information. The apply type affects the locations where a shader can be assigned. For example, a shader with an apply type of “light” can only be used as a light shader. These constraints are enforced by the material/map browser, which only shows the shaders which are valid for any given slot.

Utility Classes

These classes are provided, along with an implementation, to facilitate the work of the 3rd party developer. They are provided with a source code implementation which must be included in the VC++ project.

class mrShaderFilter

Can be used as-is, or can be extended by 3rd party developers.

This class can be used to impose “apply type” and “return type” constraints on texture map buttons. It may be used to specify shader with which apply types and return types are accepted on a texture map button. By default, only shaders with apply types “texture” or “material” and which return a color are accepted.

For more information on the material/map browser filtering feature, see the 3ds max SDK file “IMtlBrowserFilter.h”.

class mrShaderButtonHandler

To be implemented by 3rd party developers.

This class is used by the SDK samples. It provides a generic UI handler for handling buttons which are to receive shaders.

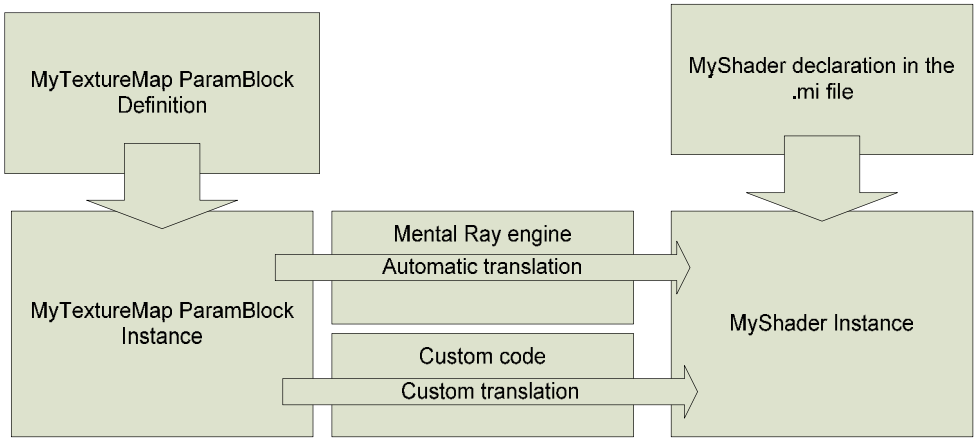
Custom Shader Translation

The custom shader translation system can be used to do the following:

- Translate a 3ds max texture map plugin to a mental ray shader.
- Translate a 3ds max material plugin to a mental ray generic material, or to a mental ray material phenomenon.

There are various motivations for using the custom shader translation system. Here are some of them:

- A 3rd party developer has written a 3ds max procedural texture map, and wants to make that procedural work with mental ray. (result: the procedural works both with MAX and mental ray)
- A 3rd party developer has written (or acquired) a mental ray shader. The mental ray shader has a complex set of parameters which is difficult to handle with 3ds max's automatic UI created for this shader. The 3rd party developer can easily create a texture map plugin for 3ds max, which will serve only as a UI handler, and use the custom translation system to translate this "UI proxy" to a mental ray shader.



Custom Translation of Texture Map Plugins

Preliminaries: The 3rd party developer should have an implementation of a texture map plugin (deriving from class Texmap).

Step 1: Derive the texmap plugin from class imrShaderTranslation

The texmap plugin must implement the imrShaderTranslation class to enable custom shader translation.

Step 2: Implement InterfaceServer::GetInterface()

The texmap plugin must provide a simple implementation for this method to return a pointer to itself as an imrShaderTranslation. Here is an example:

```
BaseInterface* YourTexmapPlugin::GetInterface(Interface_ID id) {
	if(id == IMRSHADERTRANSLATION_INTERFACE_ID) {
		return static_cast<imrShaderTranslation*>(this);
	}
	else {
		return NULL;
	}
}
```

Step 3: Implement imrShaderTranslation::GetMRShader()

This is where you create an instance of the shader to which you want your texture plugin to translate to. Use the interface “imrShaderCreation” to create it, and return a pointer to the shader.

Note: if you create instances of several shaders (which is possible if you are translating to a tree of shaders rather than a single shader), then make references to these shaders (through the ReferenceMaker::Get/SetReference() system). This is necessary since only the root shader will be passes on to the TranslateParameters() method. Then, return the _root_ shader.

Step 4: Implement imrShaderTranslation::ReleaseMRShader()

If you made references to any shaders in step 3, then here is the place to release those references (through SetReference(NULL)). This allows freeing up memory after rendering. If you didn’t make any references to shaders in step 3, then do nothing here.

Note that this is an optional step, meant to release memory. It won’t hurt if you don’t release your shaders here, unless they use a lot of memory.

Step 5: Implement imrShaderTranslation::NeedsAutomaticParamBlock2Translation()

This function needs to return true if you want to your shader to be translated automatically, but should return false otherwise.

For automatic translation to be possible, your Texmap plugin must contain a ParamBlock2 which is a one-to-one match to the parameters structure in the mental ray shader. Here is how it works:

> Every parameter block that you want to be automatically translated must have a matching structure in the mental ray shader. For example, if your Texmap plugin has a parameter block that is called “Parameters”, then the mental ray shader must have a structure that is called “Parameters”. The contents of the parameter block will be matched with the contents of the structure.

The contents of the parameter blocks must also be a one-to-one match with the contents of the structure in the mental ray shader. That is, each parameter that you want to be automatically translated must match by name and by type to the parameter in the mental ray shader. The following table describes type compatibility.

Parameter type in mental ray shader	**Possible types in the parameter block**
boolean	TYPE_BOOL, TYPE_INT
integer	TYPE_INT, TYPE_BOOL, TYPE_TIMEVALUE, TYPE_RADIOBTN_INDEX, TYPE_INDEX
scalar	TYPE_FLOAT, TYPE_ANGLE, TYPE_PCNT_FRAC, TYPE_WORLD, TYPE_COLOR_CHANNEL
vector	TYPE_POINT3, TYPE_RGBA
transform	TYPE_MATRIX3
color	TYPE_RGBA, TYPE_POINT3, TYPE_POINT4**, TYPE_FRGBA**
shader	TYPE_TEXMAP, TYPE_REFTARG***
colortexture	TYPE_FILENAME if the parameter doesn't have the GUI attribute <"textureInfo" "max_texmap">, and TYPE_TEXMAP/TYPE_REFTARG*** if it does have the GUI attribute.
scalartexture	same as "colortexture"
vectortexture	same as "colortexture"
light	TYPE_INODE, TYPE_REFTARG*** (reftarg must be INode*)
string	TYPE_STRING, TYPE_FILENAME
data	same as "shader"
lightprofile	unsupported
geometry	TYPE_INODE, TYPE_REFTARG*** (reftarg must be INode*)
material	TYPE_MTL, TYPE_REFTARG*** (reftarg must be Mtl*)
struct*	TYPE_PBLOCK2*
array	Tab of parameters (e.g. "array boolean" is compatible with TYPE_BOOL_TAB and TYPE_INT_TAB).

*Structure must also match. If the mental ray shader contains a sub-structure of parameters, then the parameter block in the Texmap plugin must contain a sub-parameter-block of the same name, and the contents of the sub-parameter-block will be matched with the contents of the sub-structure.
**As long as the parameter does not have the GUI attribute "no_alpha".
***The reference target must be of the correct type (e.g. it must be a Texmap* if a TYPE_TEXMAP is expected).

Note about case sensitivity: The automatic translation is _not_ case sensitive.

If automatic translation is not possible, then manual translation is also possible (see the next step).

Step 6: Implement imrShaderTranslation::TranslateParameters()

This method is used to translate parameters *manually*. It can be used *alongside* automatic parameter translation (see previous step), to translate some parameters automatically and some manually, or by itself, to translate *all* parameters manually.

The constraints for manual translation are similar to those of automatic translation. The names and parameter structures no longer have to match, but the types must still match.

The parameters must be translated into the parameter block of the shader. This parameter block is accessed through imrShader::GerParametersParamBlock(). The structure of this parameter block matches that of the mental ray shader as declared in the .mi file. The types of parameters contained in this parameter block are as follows:

Parameter type in .mi declaration	**Type of parameter in the parameter block of the imrShader**
boolean	TYPE_BOOL
integer	TYPE_INT
scalar	TYPE_FLOAT by default, TYPE_WORLD if GUI attribute <"units" "world"> is specified
vector	TYPE_POINT3
transform	TYPE_MATRIX3
color	TYPE_FRGBA by default, TYPE_RGBA if GUI attribute "noAlpha" is specified
shader	TYPE_TEXMAP by default, TYPE_REFTARG if GUI attribute "referenceTarget" is specified
colortexture	TYPE_FILENAME by default (should be the filename of a texture), TYPE_TEXMAP if GUI attribute <"textureInfo" "max_texmap"> is specified
scalartexture	same as "colortexture"
vectortexture	same as "colortexture"
light	TYPE_INODE with sclassID restriction of LIGHT_CLASS_ID
string	TYPE_STRING
data	TYPE_TEXMAP by default, TYPE_REFTARG if GUI attribute "referenceTarget" is specified
lightprofile	TYPE_STRING (cannot be used)
geometry	TYPE_INODE with sclassID restriction of GEOMOBJECT_CLASS_ID
material	TYPE_MTL
struct*	TYPE_PBLOCK2
array	Tab of the appropriate parameter (e.g. TYPE_BOOL_TAB for "array boolean").

Note about parameter IDs: when doing manual translation, make sure you lookup the parameters *by name* and not by parameter ID, since the parameter IDs are not necessarily in order. To get a parameter ID from a parameter name, you may use the following function*:

```
bool GetParamIDByName(ParamID& paramID, const TCHAR* name, IParamBlock2*
pBlock) {

	DbgAssert(pBlock != NULL);
	int count = pBlock->NumParams();
	for(int i = 0; i < count; ++i) {
		ParamID id = pBlock->IndextoID(i);
		ParamDef& paramDef = pBlock->GetParamDef(id);
		if(_tcsicmp(name, paramDef.int_name) == 0) {
			paramID = id;
			return true;
		}
	}

	DbgAssert(false);
	return false;
}
```

*Simply pass in a pointer to the parameter block that contains the parameter, and the name of the parameter you want to lookup. The parameter ID will be returned through the "paramID" reference parameter. The function will return false if no parameter was found with the given name.

Validity interval: It is *very* important that you store the validity interval of your manual translation into the "valid" parameter. The method TranslateParameters() is *only* called for the first frame, and then for each frame where it is invalid.
Time Value: All values that you store in the parameter block of the shader must be set at time 0. That is, there should be no call such as shader->GetParametersParamBlock()->SetValue(..., (TimeValue)t). Instead, you should call SetValue(..., (TimeValue)0). 3ds max, when translating the parameters of the shader, will only use values from time 0.

Step 7: Implement imrShaderTranslation::GetAdditionalReferenceDependencies()

Most plugins should not need to do anything in this method.

When mental ray is about to translate your plugin, it examines:

- The sub-references of your plugin (i.e. retrievable through GetReference())
- The parameter blocks of your plugin
- The sub-texture maps of your plugin (i.e. retrievable through GetSubTexmap())

The translator will then translate each of these *before* translating your plugin. This is absolutely essential.

Let's use a simple case as an example: Assume you have implemented a simple texture map which contains two sub-texture maps. Those two sub-texture maps must exist in the mental ray database before your plugin is itself entered in the mental ray database – that is how mental ray works.

- If your two sub-texture maps meet any of the three criteria above (they are either sub-references of your plugin, are contained in a parameter block belonging to your plugin, or are sub-texture maps of your plugin), then they will automatically be translated and you don't need to do anything in GetAdditionalReferenceDependencies().
- However, if your two sub-texture maps *do not* meet any of the three criteria below, then the translator will fail to assign them to your shader (your shader will get a miNULLTAG as the parameter values). In this case, you must pass references to those two sub-texture maps to the translator by implementing GetAdditionalReferenceDependencies().

Step 8: Implement the imrShaderTranslation_ClassInfo interface

This interface is used by 3ds max to get "apply type" information on your shader. By default (if you don't implement this interface), your shader will be considered as having the "texture" apply type.

a) Derive the *ClassDescriptor* of your plugin from class imrShaderTranslation_ClassInfo.
b) Add a call to imrShaderTranslation_ClassInfo::Init() in the constructor of the class descriptor.

c) Implement GetApplyTypes() to return all the apply types that are valid for your shader. The apply types is a combination of flags (see the enumeration imrShaderClassDesc::ApplyFlags, in file imrShaderClassDesc.h – included with the mental ray SDK).

Step 9: Implement the IMtlRender_Compatibility_MtlBase interface

This interface is essential to tell 3ds max that your plugin is compatible with the mental ray renderer.

a) Derive the *ClassDescriptor* of your plugin from class IMtlRender_Compatibility_MtlBase.
b) Add a call to IMtlRender_Compatibility_MtlBase::Init() in the constructor of the class descriptor.
c) Implement IsCompatibleWithRenderer(). This method should check the class ID of the renderer and return true if the plugin is compatible with this renderer. The class ID of the mental ray renderer is:

```
#define MRRENDERER_CLASSID Class_ID ( 0x58f67d6c, 0x4fcf3bc3 )
```

Custom Translation of Material Plugins

Preliminaries: The 3rd party developer should have an implementation of a material plugin (deriving from class Mtl).

The process is almost identical to that of implementing custom translation for a Texmap plugin, with a few differencs.

Step 1: Derive the Mtl plugin from class imrMaterialPhenomenonTranslation.

The Mtl plugin must implement the imrMaterialPhenomenonTranslation class to enable custom shader translation.

Step 2: Implement InterfaceServer::GetInterface()

The Mtl plugin must provide a simple implementation for this method to return a pointer to itself as an imrMaterialPhenomenonTranslation. Here is an example:

```
BaseInterface* YourTexmapPlugin::GetInterface(Interface_ID id) {
      if(id == IMRMTLPHENTRANSLATION_INTERFACE_ID) {
            return static_cast< imrMaterialPhenomenonTranslation
            *>(this);
      }
      else {
            return NULL;
      }
}
```

Step 3: Implement imrShaderTranslation::GetMRShader()

This step is identical to step 3 in the custom translation of Texmap plugins, with one exception:

> The method must return a *material phenomenon* (i.e. a shader which derives from class Mtl).

You may either create an instance of your own material phenomenon or, alternatively, you may create an instance of the material phenomenon called “max_default_mtl_phen” (see its declaration in 3dsmax.mi). This phenomenon is special – it is treated differently by 3ds max. 3ds max will automatically convert this to a generic mental ray material. Thus, mental ray will see it as a generic material instead of a material phenomenon. You can use the TranslateParameters() method to assign the proper material/shadow/photon/etc. shaders to the “max_default_mtl_phen” shader.

Step 4: Implement imrShaderTranslation::ReleaseMRShader()

This step is identical to step 4 in the translation of Texmap plugins.

Step 5: Implement imrShaderTranslation::NeedsAutomaticParamBlock2Translation()

This step is identical to step 5 in the translation of Texmap plugins.

Step 6: Implement imrShaderTranslation::TranslateParameters()

This step is identical to step6 in the translation of Texmap plugins.

Step 7: Implement imrShaderTranslation::GetAdditionalReferenceDependencies()

This step is identical to step 7 in the translation of Texmap plugins.

Step 8: Implement imrMaterialPhenomenonTranslation::SupportsMtlOverrides()

Advanced feature.

This is used by 3ds max to determine whether your material plugin can be assigned the “mental ray connection” custom attribute. This custom attribute is an additional rollup in the material editor that allows overriding particular shaders in the material.

Note that, to support material overrides automatically, your shader must translate itself to the “max_default_mtl_phen” phenomenon (see step #3 for details on this phenomenon). Otherwise, you’ll have to support the overrides yourself by querying the interface class imrMaterialCustAttrib on the custom attributes of the materials.

Step 9: Implement imrMateralPhenomenonTranslation::InitializeMtlOverrides()

Advanced feature.

You don't need to provide an implementation for this if you had SupportsMtlOverrides() reutrn false. This is used to initialize the material overrides when they are assigned to the material. You don't necessarily have to do anything here.

Step 10: Implement the IMtlRender_Compatibility_MtlBase interface

This step is identical to step 9 in the translation of Texmap plugins.

Custom Translation of Object Plugins to Geometry Shaders

Step 1: Derive the GeomObject or ShapeObject plugin from class imrGeomShaderTranslation.

The object plugin must implement the imrGeomShaderTranslation class to enable custom translation to a geometry shader.

Step 2: Implement InterfaceServer::GetInterface()

The object plugin must provide a simple implementation for this method to return a pointer to itself as an imrGeomShaderTranslation. Here is an example:

```
BaseInterface* YourTexmapPlugin::GetInterface(Interface_ID id) {
	if(id == IMRGEOMSHADERTRANSLATION_INTERFACE_ID) {
		return static_cast<imrGeomShaderTranslation*>(this);
	}
	else {
		return NULL;
	}
}
```

Step 3: Implement imrGeomShaderTranslation::GetGeomShader()

Return a pointer to something that translates to a geometry shader. The typical implementation will return a pointer to a Texmap. That Texmap will typically implement the imrShaderTranslation interface to translate itself to a specific geometry shader.

Step 3: Implement imrGeomShaderTranslation::GetScale()

In many cases, returning a scale of (1,1,1) is sufficient. For specific cases, however, where the geometry shader builds objects of a static size (that cannot be changed by passing a parameter to the shader), then you might want to expose a scale parameter in the UI and use this method to force the mental ray translator to apply that scale to the transform matrix of the instance.

Advanced Translation of Material/Texmap Plugins

Starting with 3ds max 7, a new method has been introduced to class imrShaderTranslation:

```
virtual void TranslateParameters_Advanced(imrAdvancedTranslation&
advancedTranslation, TimeValue t, Interval& valid);
```

This method can be used to translate parameters in cases where:

- The standard translation method, using TranslateParameters(), is not sufficient because, for example, one needs to translate complex structures of parameters.
- One needs to translate large arrays, in which case the advanced translation method is faster because it does not use ParamBlock2.

The advanced translation method uses an interface which exposes the following mental ray API functions directly:

- `mi_api_parameter_name`
- `mi_api_parameter_value`
- `mi_api_parameter_push`
- `mi_api_parameter_pop`
- `mi_api_new_array_element`
- `mi_mem_strdup`
- `mi_mem_release`
- `mi_mem_allocate`

For the full documentation on these function calls, refer to the mental ray manual.

SDK Samples

mental ray Material

Location: maxsdk\samples\mentalray\mrMaterial\mrMaterial.vcproj

This is the implementation of the mental ray material. It makes use of material/map browser filters to only accept valid apply types on each shader button, and it uses custom shader translation to translate to the “mr_default_mtl_phen” material phenomenon (see the section on custom translation of material plugins for details).

Shader List Shader

Location: maxsdk\samples\mentalray\mrShaders\mrShaders.vcproj

This is the implementation of the “shader list” shader. It makes use of the material/map browser filters to only accept valid apply types, and it use custom shader translation to translate itself to the corresponding mental ray shader.