

## Choosing Between Utility and Modifier Plug-Ins for 3D Studio Max

By David Lanier

Programmers spend a lot of time developing tools for their artists to use, but the question of how best to develop the tool they need is not always an easy one to answer. The first time I had to develop a plug-in for 3D Studio Max, I was faced with a choice: "What kind of plug-in should I develop - a utility or a modifier?" I then spent a considerable amount of time performing tests to determine which would be the best way to go. Now, lucky for you, rather than going through that same process yourself, you can just read this article, which will hopefully help shorten that testing time.

The main topic of this article, then, will be the difference between developing a modifier and a utility plug-in. We'll discuss the advantages and drawbacks of each of these plug-in types, as well as the use of the Microsoft Foundations Class (MFC) inside a utility. Although this article could potentially be of interest to anybody working with 3D Studio Max, I'd recommend that the reader have a basic knowledge of developing 3D Studio Max plug-ins in order to get the most out of this. A good resource to check out for a little more background is the 1998 Gamasutra article "[From 3D Studio Max to Direct3D](#)" by Loic Baumann.

### Comparing Modifier and Utility Plug-Ins

First, the basics. A modifier is a plug-in that modifies objects (also called nodes) in some way. A utility is a plug-in that is useful for implementing modal procedures such as 3D paint, dynamics, and so forth. The major difference is that utilities are not designed to modify objects, so you won't find a utility that, say, modifies the texture vertices of a mesh as the UVW Unwrap modifier does (although it can theoretically be done, as we will see later).

Let's first take a look at the advantages of using modifiers.

### Advantages of Developing a Modifier

The biggest advantage a modifier has over a utility is the geometry pipeline. It's a little bit complicated at the beginning to understand, but once you have an understanding of how this pipeline works, modifiers can become very powerful.

An understanding of the geometric pipeline system is important for developers creating plug-ins that deal with geometry in the scene. The pipeline is the system used by 3D Studio MAX that allows a node in the scene to be altered, perhaps repeatedly, through the application of modifiers. At the beginning of a pipeline is the Base Object. This is either a procedural object or just a simple mesh. At the end of a pipeline is the world space state of the object. This world space state is what appears in the 3D viewports and is rendered.

For the system to evaluate the state of the object at the end of the pipeline, it must apply each modification along the way, from beginning to end. For example, say a user creates a procedural cylinder in the scene, applies a Bend modifier to it, and then applies a Taper modifier to it. As the system evaluates this pipeline, it starts with

the state of the cylinder object. As this object state moves along the pipeline, it first encounters the Bend modifier, and the system asks the Bend modifier to apply its deformation to the object state. The result of this operation is then passed as the source into the Taper modifier, and the Taper then applies its own deformation. The result of this operation is then passed to the system which translates the result into world space, and the state of the node in the scene is complete.

To maximize the speed with which the system can evaluate the state of a node, the system maintains a World Space Cache for each node in the scene. This world space cache is the result of the node's pipeline; it reflects the state of the object in world space after everything has been applied to it. Along with the cache, the system maintains a Validity Interval. The validity interval indicates the period of time over which the cache accurately reflects the state of the node.

With the cache structure inside modifiers, the user is able to remove, copy, or paste a modifier from the modifier stack. The pipeline is then re-evaluated to build a new world space state for the objects that have been modified, and we are able to collapse the modifier stack of an object. Collapsing the modifier stack means that:

1. Every modifier on the object applies its changes;
2. The resulting world space state becomes the current state of the object;
3. All modifiers are erased, so going back is no longer possible.

It's more or less like flushing the modifier pipeline.

Remember, in Kinetix' 3DS4, the user wasn't even able to change past modifications, so this is now a very powerful feature! But this modifier pipeline is also modifiers' biggest drawback...

### **Drawbacks of Developing a Modifier**

Developing a modifier is no easy task even for even experienced programmers, because you have to know exactly how the pipeline works. For example, you have to tell the pipeline which channels of the object you are going to change (i.e., the channels that the modifier needs in order to perform its modification) and which channels you are going to use (i.e. the channels that the modifier actually modifies). For instance, you have the texmap channel (texture vertices and procedural mappings), the geometry channel (the vertices of the object), and so on. If you don't tell the pipeline which channels you are going to change, your changes won't affect the object! This just goes to show that you need to know exactly how the pipeline works and what you are going to do to objects before developing a modifier.

As an example of the complexity of the pipeline, let's say you want to have a modifier that lets the user manually select some given number of faces, then perform several operations on those faces. To do so, you'll need to copy and paste about 800 lines from the mesh select or edit mesh modifier, and then you'll have to ensure that the code actually works in your modifier. You have to think about that before developing a modifier.

The geometry pipeline is a very nice feature when you discover it as a user, but when you use it everyday, it quickly becomes a burden. As another example, if you want to modify the texture vertices of some given number of faces, you have to first select these faces, then modify their texture vertices. To do that you'll need the edit mesh or mesh select modifier to select faces, and then you'll have to add a UVW Unwrap modifier to change the texture vertices of the selected faces. In this silly example, two modifiers have been used to perform the desired operation. And if you want to do this operation again, you will need to add these two modifiers once again. As a result, the modifier stack keeps on growing and you are forced to collapse the modifier stack regularly to avoid filling memory with the local cache of each modifier.

So while modifier plug-ins have distinct benefits, they've also got some serious disadvantages. Not let's take a look at utility plug-ins.

### **Advantages of Developing a Utility**

Utility plug-ins are easier to develop than modifiers; you don't need to understand a complex pipeline to develop a utility. Only a few basics of 3D Studio Max programming are even necessary really, and you generally only need to know where in the scene you can find the information that you need (although even that can become a headache!) For example, to get the number of selected nodes as well as their pointers you would use the following code:

```
// Get Interface pointer, Interface is the most useful // class in 3D
Studio Max
Interface* ip = GetCOREInterface();
If ( ! ip )
    return;

// Get the number of selected nodes in NumSelNodes
const int NumSelNodes = ip->GetSelNodeCount();

// Now scan all the selected nodes one at a time
for (int i = 0 ; i < NumSelNodes; i++) {
INode* node = ip->GetSelNode( i );
if (!node)
    continue;
DoSomethingOnThisNode(node);
}
```

Let's have a quick overview of materials and the material editor in 3D Studio Max and how these materials are used for real time games. The material editor is used to manage the materials. It can create materials such as Blend, Composite, Double sided, Matte / Shadow, Morpher, Multi / SubObject, RayTrace, Shellac, Standard, Top / Bottom, etc.

For real time games, we only use the Standard and Multi/Subobject materials. Standard materials have properties such as ambient color, diffuse color, specular color, glossiness, opacity, bump, etc. For real time games, we set a bitmap texture in the diffuse channel of a material. (In other words, we replace the overall color with an overall texture.)

In 3D Studio Max, a node can reference one - and only one - material, any kind of material previously described. There is not (as in Alias|Wavefront Maya) the possibility to set a different material for each face. That's why the Multi/Subobject materials exist! When you want an object to have several different materials applied to individual parts of the object, you use a Multi/Sub-object material, which is a material that holds a number of different materials (called submaterials ) inside it.

We usually create a multi/subobject material that contains standard materials with each a bitmap texture in the diffuse channel. So how can we now set the good texture on a part of a mesh from this multimaterial?

Each face of the mesh has an integer which is the submaterial number in the node's multi/subobject material called the face material ID. It maintains the identification with the submaterial within the multi/subobject material. So when a multimaterial is applied on a node, if the faces from 0 to 100 have the face material ID 0, they reference the first submaterial (so it references the bitmap texture contained in this submaterial); if faces 101 to 200 have the face material ID 1, they reference the second submaterial; and so forth. When a standard material is applied on a mesh, these face material IDs are not used.

Now to get all materials that are in your scene, you can use the following code ("In the scene" means that all these materials are applied to at least one of the nodes in the scene):

```
//Get Interface pointer, Interface is the most useful class of 3D Studio
Max
Interface* ip = GetCOREInterface();
If ( ! ip )
    return;

//Get all materials stored in a library of materials
//The MtlBaseLib class is a library of MtlBase entries.
MtlBaseLib* lib = ip->GetSceneMtls();
if (!lib)
    return;

const int NumMat = lib->Count();
for( int i = 0; i < NumMat; i++ ){
MtlBase* mtl = static_cast<MtlBase*>( (*lib)[ i ] );
If ( ! mtl)
    Continue;
DoSomethingOnThisMat( mtl );
}
```

One advantage of using a utility over a modifier plug-in is that you can use several utilities at the same time if they have modeless dialog boxes instead of dialog boxes set in the 3D Studio Max rollup panel. This is not possible with modifiers, where the active modifier is the one which is selected in the modifier stack, but two modifiers can't be active at the same time.

Another advantage is that you can drive modifiers from a utility. Let's see how to instantiate modifiers in 3D Studio Max.

First, you can't use the new operator to instantiate all the classes you need in 3D Studio Max. It will work for some classes but most of them are not instantiable - for example the modifiers classes. The 3D Studio Max system uses unique class Ids (a unique set of 2 32-bits wide integers) to identify each class. So to instantiate a class you have to use the `CreateInstance` function from the class interface . You pass the `CreateInstance` function a class ID and it returns a pointer to an instance of the desired class. This is a class factory design pattern that we can find as well in the Microsoft Component Object Model (COM) with the `CoCreateInstance` function.

In our case, if we have the Edit Mesh and UVW Unwrap modifiers class IDs, we can create instances of these modifiers. And we are able to apply them programmatically on some objects in the scene (**HTML link on the project**).

Example 1: How to create an instance of the UVW Unwrap modifier

```
#ifndef UNWRAP_CLASSID
#define UNWRAP_CLASSID Class_ID(0x02df2e3a,0x72ba4e1f)
#endif
Interface* ip = GetCOREInterface();
If ( ! ip )
    return;
Modifier* UVWUnwrapMod =static_cast<Modifier*> (ip->
CreateInstance(OSM_CLASS_ID, UNWRAP_CLASSID));
```

But now that you are able to add through code modifiers to some nodes, you'd surely like to use their functions, right? This is where things become complicated. As you've probably already seen, the C++ projects generated by the 3D Studio Max wizard have the same structure: the classes are declared in the same file as their implementation in the cpp file. It's a pity that header files don't contain class definitions, because it would be so simple to include the header file of the class you need (in our case a modifier) in your project settings and link with the lib file from the class project, as we usually do when using a library.

But it's not possible to include the header files with the 3D Studio Max modifiers classes, so here's the workaround I've found. It may well not be not the best way to do this, and I imagine I'll receive a flood of e-mails suggesting better methods, but if you do need one, this works fine. It's actually just a copy and paste operation; I copy the class definitions from the original modifier cpp file in a header file in my project, then I copy all the functions of the class that I want to use (and their dependencies) in a cpp file in my project. Then I add my own functions in the class definition to extend this class.

I was then able to call functions from the modifier. As I said, there are probably much better solutions, but as usual, I only had a limited time to reach my goal so I had to make it work in any way I could! And it does work; I succeeded in driving the Edit Mesh, the UVW Map and the UVW Unwrap modifiers from a utility plug-in.

(Notably, most of these problems concerning the header files will be solved in the future, because 3D Studio Max is becoming an open source project. See [http://www2.discreet.com/games/developers\\_corner.html](http://www2.discreet.com/games/developers_corner.html) for more information.)

## Drawbacks of Developing a Utility

The big drawback of utilities is that when you modify an object in a utility, say, by changing the face selection set, the changes don't "stay forever." Once the viewport is redrawn, the geometry pipeline is re-evaluated (as previously described), and your changes are lost. To make your changes "stay forever," you'll need to collapse the modifier stack (**HTML Link on the project**). This is the main constraint that you have to keep in mind when developing a utility plug-in: modifications are only allowed on collapsed objects (objects that have their modifier stack collapsed). And sometimes the artists don't want you to collapse the modifier stack of their objects because if you do so, they won't be able to go back into the modifier stack history.

Let's introduce the Physique and Skin modifiers. They are used to assign bones to some vertices of a mesh to animate it and use inverse kinematics. Physique is designed for biped character animation; it has a predefined biped skeleton with 2 arms, 2 legs, 2 feet, a head, etc. Skin is designed to add bones to any part of a mesh; there is no predefined skeleton. If Skin or Physique modifiers are applied on a node, this node can't have its modifier stack collapsed - or all the bones-to-vertices assignment will be lost! (If a node has a modifier that stores its own information and this information can't be stored in the mesh, the modifier stack of this node can't be collapsed. Well, theoretically, it can be collapsed... it won't crash, but all your information will be lost!)

## Possibility of Using the Microsoft Foundation Classes in Utility Plug-Ins

Plug-ins are usually developed using the Microsoft Windows Win32 API. But with utility plug-ins you can use the Microsoft Foundation Class as well. The Microsoft Foundation Class Library is an "application framework" for programming in Microsoft Windows. The MFC framework is a powerful approach that lets you build upon the work of expert programmers for Windows. As far as I know, utility plug-ins are the only plug-in type where MFC can be used. To download a skeleton utility that uses MFC, have a look at the Discreet web site: <http://www.ktx.com/mfc>.

## Advantages of Using MFC

MFC shortens development time, provides tremendous support without reducing programming freedom and flexibility, and gives its user easy access to "hard to program" user-interface elements and technologies. MFC makes it easy to program features like property sheets ("tab dialogs"), print preview, and floating, customizable toolbars.

If you use the Microsoft Visual C++ compiler, MFC is fully integrated with the development environment interface. For example: you have the MFC ClassWizard which helps you to map messages and controls from your dialog boxes; since version 6.0 of Microsoft Visual C++, it automatically shows the member functions of the class while typing. This is not possible when using the Win32 API because a lot of functions use a window handle (HWND) and global functions to deal with your dialog boxes controls. This means, for example, that there is no class regrouping all

functions of the list boxes or combo boxes - while in MFC, you have the `CListBox` and `CComboBox` classes.

For example, when you want to empty a list box using the Win32 API, you have to use the following function:

```
SendMessage (
    (HWND) hWnd,      // handle to destination window
    LB_RESETCONTENT, // message to send
    (WPARAM) wParam; // not used; must be zero
    (LPARAM) lParam; // not used; must be zero
);
```

You have to know the `HWND` of your list box. And messages processing is a burden to use.

In MFC, you just have to do :

```
CListBox MyListBox;          // list box that has been mapped
MyListBox.ResetContent();
//This is part of the CListBox class, no message processing in //your code
(although it is done implicitly)
```

Moreover, a strong advantage of MFC over the Win32 API is that you can download a lot of complete MFC projects with source code. So you rarely start from scratch when you have to do, for instance, a treeview with the drag and drop functionalities enabled. Have a look at the following web site to download some examples of MFC projects: <http://codeguru.com/> . This web site has everything you need to use MFC; I managed to save a couple of weeks using existing code.

## Drawbacks of Using MFC

You should probably know before choosing to use the MFC inside 3D Studio Max that you won't be getting any support from Discreet. MFC is not officially supported, so you have to use it at your own risk. This will be a concern with pure MFC problems and problems with integration of MFC inside 3D Studio Max. Here are some of the strange problems you may encounter when combining the two:

Most of the problems met when using MFC are 3D Studio Max functions from the kernel that don't work. For example, when using release 3 or later, if you want to have customizable keyboard shortcuts for your plug-in (which are very useful), you need to call the functions `ActivateShortcutTable` and `DeactivateShortcutTable` as follows:

```
//This is a class to create the keyboard shortcuts
class PluginShortcutManagerCB : public ShortcutCallback
{
    virtual BOOL KeyboardShortcut (int id);
}
const ShortcutTableId kMappingShortcuts = 0x34f274f4;

//Create an instance of this class
PluginShortcutManagerCB* mappingShortcutCB = new PluginShortcutManagerCB;
```

```
Interface* ip = GetCoreInterface();

//Activate the shortcuts
ip->ActivateShortcutTable(static_cast < ShortcutCallback* >
    (mappingShortcutCB), kMappingShortcuts);

//DeActivate the shortcuts
ip->DeactivateShortcutTable(static_cast < ShortcutCallback* >
    (mappingShortcutCB), kMappingShortcuts);
```

And `DeactivateShortcutTable` crashes each time you call it when using MFC, while it works fine when you use just the Win32 API.

To solve this problem, I have created another DLL, using only the Win32 API, which links with my MFC DLL and then I register, activate and deactivate the keyboard shortcuts by calling functions in this Win32 DLL. It's a patch, but it works fine!

So there are issues with using MFC, but to date I have always been able to solve the problems that I've run into when using MFC.

## Conclusions

To sum up then. A modifier is a plug-in made to modify objects in some way and that benefits from the geometry pipeline, while a utility is a plug-in that is useful for implementing modal procedures but is not designed (in theory) to modify objects.

In my opinion, modifiers should be used when you can't develop a utility, because utility plug-ins are simpler to develop. A utility should be developed to:

1. Get information from the scene without modifying objects, or
2. Modify objects only if the objects can have their modifier stacks collapsed, or
3. Drive modifiers and build a "super utility plug-in".

So knowing the constraints and the specifications of the tool that you have to develop, you should now be able to choose wisely between the two types of plug-in, as well as the Microsoft Foundation Classes. And remember: programming in 3D Studio Max, your code might not always look nice, not when you have to make things work in any way you can, but you usually can make things work.

## Acknowledgments and References

Thanks to Jurie Horneman for his help in writing this article.

Further information about this topic can be found at:

- The Discreet web site : <http://www.discreet.com/>

**David Lanier has been a programmer working for Kalisto Entertainment France. In year 2K, Kalisto had developed games such as Dark Earth, Ultimate Race Pro, The Fifth element, 4 Wheel Thunder and Nightmare Creatures II. David has worked in the R&D department as a tool developer during two years then became the Lead Engineer of the tools team.**

With his 5+ years of experience, he is now working as a Freelance 3D Tools developer and training specialist for CG programming.

He can be reached at [contact@dl3d.com](mailto:contact@dl3d.com). Visit [www.dl3d.com](http://www.dl3d.com) for more information.