3D Studio Max SDK

1. About the David Lanier 3D ® company

David Lanier 3D ® is a company working as a service provider for the 3D graphics industry. We are proposing **consultancy** as well as leading edge **development of custom software tools** to dramatically **speed up your 3D production process**.

Our business is designed around companies that need to create cutting edge 3D content very quickly without sacrificing quality.
The range of applications include :
- Video games
- Feature Films and Visual effects
- Visualization and web (e.g. : Architecture, Advertisement, Design, Scientific Research, Education …)

We improve your 3D production workflow by developing custom plug-ins and scripts for graphic software, such as :
- 2D : **Photoshop**, **Illustrator** …
- 3D : **3D Studio Max** and **Character Studio**, **Maya**, **XSI**…
- Database managers : **Visual Source Safe**, **NxN**…
- Others…

About David Lanier, founder of David Lanier 3D ® :

After my Applied Mathematics Engineer education, David started to work in the video games industry at Kalisto Entertainment, France in 1998. Kalisto had about 100 graphic artists (modellers, texturers, animators…) using ***3D Studio Max, Character Studio, Maya, SoftImage, Photoshop*** and ***Illustrator*** to create cutting edge game content.

He has worked during 4 years in the R&D department as a Lead Engineer of the tools team. He has proven experience in analysing 3D production process and developing quick and painless custom solutions per project to shorten their production time.

For example, “saving mouse clicks on complexes or repetitive process” and “Designing tools to be used by artists” are part of his core values. He is also used to delivering high-reliability software on time and fully documented.

David’s experience includes texturing, materials, importers-exporters, skinning, rendering, dynamics, animation, camera, checking assets for PS2 constraints, triangle-strips, database managing, automation of complexes and repetitive process, objects properties editing, …

He started to work with 3D Studio Max and Maya since the 2.5 releases in 1998. In 2000, he has written an article for a professional web site in the video game development : Gamasutra. It dealt with 3D Studio Max programming :
http://www.gamasutra.com/features/20000614/lanier_01.htm.

During one year, David has worked at Bastilday, France. It was a company dealing with leading edge middleware technology about vectorized textures compression. I was in charge of developing the creation pipeline of this technology and its authoring tools.

With his 5+ years of experience, he is now working as a Senior 3D Tools Developer and training specialist for CG programming. He is highly motivated, enthusiast in being a creative service provider for artists, that is why he has developed this activity.

David has recently worked on the game from Ubisoft named **Tom Clancy's Splinter Cell : Pandora Tomorrow**.

Please, feel free to contact us at contact@dl3d.com to see how we can help you shorten your production time.

2. Introduction

3D Studio Max is based on a plug-in architecture. One can extend its functionalities by developing plug-ins and scripts using the SDK and the Maxscript language.

This software has 2 goals :

- Create images or movies using 3D content (short movies, special FX, architecture, marketing, advertisements…)
- Create 3D content to be used by another software (such as a game engine in a video game application)

Plug-ins development can be done in both fields because both are common process.

In this document, we are going to see the main functionalities of 3D Studio Max programming using the SDK in C++ language. Max SDK is something difficult to get into, so be patient… With a little bit of training, it becomes an application that you can extend easily with pleasure.

This document is not intended to replace the documentation and we won't cover all features from the documentation here. It exposes the main features of 3D Studio Max SDK and gives our experience on how using this SDK in the best way.

Technically speaking, programming 3D Studio Max is accomplished using Microsoft Visual Studio 6 (for versions prior to 5.1) or Visual Studio .NET 2002 for versions of 3D Studio Max 6 and above. It is not recommended to use another compiler.

So to start this training session, you will need Microsoft Visual Studio C++ (6 or .NET) with the SDK installed. Use the "full configuration" option when installing 3D Studio Max from the install CD because the SDK is not installed by default.

Let's now see the different plug-ins categories we can develop for 3D Studio Max :

3. The different plug-ins categories

Procedural Objects
Procedural Objects are the general class of developer-defined objects that can be used in 3ds max.

Geometric Objects
These are the only procedural objects that actually get rendered.

Primitives
Primitive objects such as boxes, spheres, cones, cylinders, and tubes are implemented as procedural geometric objects. These are derived from class GeomObject or SimpleObject. Example code may be found in **`\MAXSDK\SAMPLES\OBJECTS\BOX.CPP, SPHERE.CPP, CONE.CPP`**, etc.

Particles
Developers may create procedural object particle system plug-ins. Some examples are particles that depend upon procedural motion like fireworks, explosions, and water, or particles that track the surface of objects like electrical fields or flame. Applications can be derived from ParticleObject or SimpleParticle. Example code is available in **`\MAXSDK\SAMPLES\OBJECTS\RAIN.CPP`** and **`\MAXSDK\SAMPLES\MODIFIERS\GRAVITY.CPP`**.

Loft Objects
The 3ds max Lofter is implemented as a procedural object plug-in. Developers may define other modeling modules that fit this form.

Compound Objects
Compound objects take several objects and combine them together to produce a new object. Examples are Booleans (which produce a new object using operations Union, Intersection and Difference) and Morph objects. Sample code for the boolean object can be found in **`\MAXSDK\SAMPLES\OBJECTS\BOOLOBJ.CPP`**.

Patches
Developers can create patch modeling systems that work inside MAX. The TriPatch and Patch Grid are examples of patch objects. These plug-ins are derived from PatchObject. See **`\MAXSDK\SAMPLES\OBJECTS\PATCHGRD.CPP`**, and **`TRIPATCH.CPP`** for sample code.

NURBS
The NURBS API provides an interface into the NURBS objects used by MAX. Using the API developer can create new NURBS objects or modify existing ones. See the Advanced Topics section Working with NURBS for more information.

Helper Objects
Helper objects are items such as dummy objects, grids, tape measurers and point objects. These objects may be derived from classes **`HelperObject`** or **`ConstObject`**.

Sample code may be found in **\MAXSDK\SAMPLES\OBJECTS\HELPERS\GRIDHELP.CPP**, and **PTHELP.CPP**. See Class HelperObject and Class ConstObject.

Shape Objects

These are shapes such as Circles, Arcs, Rectangles, Donuts, etc. New splines may be subclassed off SimpleSpline. Sample code may be found in **\MAXSDK\SAMPLES\OBJECTS\CIRCLE.CPP, ELLIPSE.CPP, ARC.CPP**, etc.

Procedural Shapes

These are shapes that are defined procedurally. An example procedural shape, Helix, may be found in **\MAXSDK\SAMPLES\OBJECTS\HELIX.CPP**. When an edit spline modifier is applied to a procedural shape it is converted to splines with segments that provide vertices in a linear approximation of the shape. This allows the procedural shape to be edited. Procedural shapes may be derived from class SimpleShape. Other examples include Procedural lines, and Text.

Any of these objects can be edited with an edit spline modifier or extruded or surfrev'd.

Lights

Developers may create custom plug-in lights. There are several classes from which light plug-ins may be derived. These are LightObject, and GenLight. Example code may be found in **\MAXSDK\SAMPLES\OBJECTS\LIGHT.CPP**.

Cameras

Developers may create custom cameras. The two classes from which cameras may be derived are CameraObject and GenCamera. An example of a plug-in camera may be found in **\MAXSDK\SAMPLES\OBJECTS\CAMERA.CPP**.

Object Modifiers

Object modifiers are applied to objects in their own local transform space to modify them in some way. Deformations like Bend, Taper, and Twist are examples of Object Modifier plug-ins. Example code may be found in **\MAXSDK\SAMPLES\MODIFIERS\BEND.CPP, TAPER.CPP**, etc.

Extrude and Surfrev are also object modifier plug-ins. Sample code for Extrude can be found in **\MAXSDK\SAMPLES\MODIFIERS\EXTRUDE.CPP**.

Developers may also create surface modifier plug-ins to alter smoothing groups, texture coordinates, and material assignments. See Class Modifier or Class SimpleMod.

Edit Modifiers

These plug-ins allow specific object types to be edited. For example, an Edit Mesh modifier allows objects that can convert themselves into triangle meshes to be edited, while an Edit Patch modifier allows objects that can convert themselves into patches to be edited. Edit modifiers typically allow the user to select sub-object elements of the object (vertices faces and edges in the case of the Edit Mesh modifier) and perform at least the standard move/rotate/scale transformations to them. They may also support additional operations (such as the extrude option of the Edit Mesh modifier). Example code may be found in **\MAXSDK\SAMPLES\MODIFIERS\EDITMESH.CPP**.

Space Warps

Space Warps are basically object modifiers that affect objects in world space instead of in the object's local space (they were originally called 'world space modifiers'). Space Warps are

non-rendering objects that affect other objects in the scene based on the position and orientation of the other objects that are bound to the Space Warp object. For example, the Ripple Space Warp applies a sine wave deformation to objects bound to it. Other examples of Space Warps include things like explosions, wind fields, and gravity. Sample code may be found in **\MAXSDK\SAMPLES\MODIFIERS\SINWAVE.CPP**.
Space warps are created in the Creation branch of the command panel, which makes them slightly different from regular modifiers (because they are combinations of space warp objects and space warp modifiers).
Space warps may also affect particle systems. For example, a force field can be applied to a particle system by a space warp. The force field provides a function of position in space, velocity and time that gives a force. The force is then used to compute an acceleration on a particle which modifies its velocity. For details see **\MAXSDK\INCLUDE\OBJECT.H**, **\MAXSDK\SAMPLES\MODIFIERS\GRAVITY.CPP**, and **\MAXSDK\SAMPLES\OBJECTS\RAIN.CPP**.
A collision object can also be applied to a particle system by a space warp. The collision object checks a particle's position and velocity and determines if the particle will collide with it in the next period of time. If so, it modifies the position and velocity.

Controllers
Controller plug-ins are the objects in 3ds max that control animation. Controllers come in different types based on the data they control. The most common controllers are interpolating or keyframe controllers. Other controller types are position/rotate/scale, mathematical expressions and fractal noise. Example controller code may be found in **\MAXSDK\SAMPLES\HOWTO\PCONTROL\PCONTROL.CPP**, and **NOIZCTRL.CPP** etc. Controllers may be derived from Class Control or Class StdControl.

Systems
Systems are basically combinations of more than one type of procedural object, along with optional controllers, or modifiers, or space warps all working together. These plug-ins can provide high-order parametric control over very complex systems. An example system is Biped which uses procedural objects and master/slave controllers.

File Import
These plug-ins allows 3D geometry and other scene data to be imported and exported to file formats other than the 3ds max format. An example file import plug-in may be found in **\MAXSDK\SAMPLES\IMPEXP\3DSIMP.CPP**. These plug-ins are derived from Class SceneImport.

File Export
These plug-ins allows 3D geometry and other scene data to be exported to file formats other than the 3ds max format. Sample code may be found in **\MAXSDK\SAMPLES\IMPEXP\3DSEXP.CPP**. These plug-ins are derived from Class SceneExport.

Atmospheric Plug-Ins
These plug-ins are used for atmospheric effects. MAX's Fog, and Volume Fog are two atmospheric plug-ins. Certain particle system-ish effects can be accomplished via atmospherics more efficiently. For example, a fire effect that is not done with particles but rather as a function in 3D space (the Combustion plug-in is a good example of this). Instead of rendering particles you traverse a ray and evaluate a function. These plug-ins typically use very little memory relative to a particle system equivalent. Atmospheric plug-ins also have the

ability to reference items in the scene. (For example, MAX's Volume Lights reference lights in the scene.) These plug-ins are derived from Class Atmospheric.

Plug-In Materials
These are additional developer-defined material types. Examples are Standard, Mix, and Multi/Sub-Object materials. New materials are subclassed from Class Mtl. Also see the section Working with Materials and Textures. The sample code for these plug-ins is in **\MAXSDK\SAMPLES\MATERIALS**.

Plug-In Textures
Procedural Texture plug-ins define 2- or 3-dimensional functions which can be assigned as maps within the shader tree architecture of the Materials Editor. Maps may be assigned as ambient, diffuse, specular, shininess, shininess strength, self-illumination, opacity, filter (transmission) color, bump, reflection and refraction maps. These functions may vary over time to produce animated effects. There are both 2D and 3D procedural textures, compositors and color modifiers. These plug-ins are derived from Class Texmap. Also see the section Working with Materials and Textures. The sample code for these plug-ins is in **\MAXSDK\SAMPLES\MATERIALS**.

2D Procedural
Examples of 2D texture are **BITMAP.CPP** and **CHECKER.CPP**.

3D Procedural
Examples of 3D textures are **MARBLE.CPP** and **NOISE.CPP**.

Compositor
Some examples of compositors are **MASK.CPP** and **MIX.CPP**.

Color Modifier
An example color modifier is RGB **TINT.CPP**.
Developers that have created a 3D Studio/DOS SXP and a corresponding 3ds max texture plug-in may want to have a look at Class Tex3D. It provides a way to have an instance of your 3ds max texture plug-in created automatically when the corresponding SXP is found in a 3DS file being imported.

Image Processing Plug-Ins

Filters
Filters may be used to alter images in the video post data stream. Filters may operate on a single image or may combine two images together to create a new composite image. These plug-ins are derived from Class ImageFilter. Also see the section Working with Bitmaps.

One Pass Filter
This plug-in type allows a single image in the video post data stream to be adjusted in some manner. An example plug-in of this type is **\MAXSDK\SAMPLES\POSTFILTERS\NEGATIVE\NEGATIVE.CPP**.

Layer Filter
This plug-in allows two images to be composited to create a single new image. An example of this type of plug-in is **\MAXSDK\SAMPLES\POSTFILTERS\ADD\ADD.CPP** or **\MAXSDK\SAMPLES\POSTFILTERS\ALPHA\ALPHA.CPP**.

G Buffer
A G-buffer is used to store, at every pixel, information about the geometry at that pixel. All plug-ins in video post can request various components of the G-buffer. When video post calls the renderer it takes the sum of all the requests and asks the renderer to produce the G-buffer. Developers can use this information to create visual effects that are impossible to achieve without access to a G-buffer. See Class GBuffer.

Rendering Effects
This plug-in type is available in release 3.0 and later only.
There is a new item under the Rendering menu which displays the Rendering Effects dialog. From this modeless dialog, the user can select and assign a new class of plug-in, called a "Rendering Effect," which is a post-rendering image-processing effect. This lets the user apply image processing **without** using Video Post, and has the added advantage of allowing animated parameters and references to scene objects. The base class for these plug-ins is Class Effect. Sample code is available in the directory **\MAXSDK\SAMPLES\RENDER\RENDEREFFECT**.

Snap Plug-Ins
This plug-in type is available in release 2.0 and later only.
This plug-in type allows custom points to be provided to the 3ds max snapping system. For example a door plug-in could provide a custom snap for the hinge center. See Class Osnap for details. For sample code see **\MAXSDK\SAMPLES\SNAPS\SPHERE\SPHERE.CPP**.

Image Loading and Saving Plug-Ins
Image loading and saving plug-ins allow the image file formats loaded and saved by 3ds max to be extended. An example is the JPEG loader / saver. Sample code may be found in the sub-directories of **\MAXSDK\SAMPLES\IO**. These plug-in types are derived from Class BitmapIO. Device drivers are also derived from this class. See the sample code in **\MAXSDK\SAMPLES\IO\WSD\WSD.CPP**.

Utility Plug-Ins
These plug-ins are useful for implementing modal procedures such as 3D paint, dynamics, etc. These plug-ins are accessed from the Utility page of the command panel. Example code may be found in the subdirectory **\MAXSDK\SAMPLES\UTILITIES**. These plug-ins are sub-classes off Class UtilityObj.

Global Utility Plug-Ins
This plug-in type is available in release 3.0 and later only.
These simple utility plug-ins are loaded at boot time, after initialization, but before the message loop starts, and remain loaded. This is how the new 3ds max COM/DCOM interface is implemented. For details see Class GUP.

Track View Utility Plug-Ins
These plug-ins are launched via the 'Track View Utility' icon just to the left of the track view name field in the toolbar. Clicking on this button brings up a dialog of all the track view

utilities currently installed in the system. Most utilities will probably be modeless floating dialogs, however modal utilities may be created as well. These can provide general utility functions that operate on keys, time or function curves in Track View. Sample code is available in **\MAXSDK\SAMPLES\UTILITIES\RANDKEYS.CPP**, **ORTKEYS.CPP** and **SELKEYS.CPP**. These plug-ins are sub-classes off Class TrackViewUtility.

Plug-In Renderers
Plug-In renderers are derived from the class **Renderer**. The standard 3ds max scanline renderer is itself derived from this class. In a trivial sense, there are only a few methods to implement to create a renderer: **Open()**, **Render()**, **Close()**, **ResetParams()** and **CreateParamDlg()**. See Class Renderer for more details on this plug-in type.

Shader Plug-Ins
This plug-in type is available in release 3.0 and later only.
This plug-in type works with the new Standard material. It allows plug-in developers to add additional shading algorithms to the drop down list of available options (previously Constant, Phong, Blinn, Metal). This was only possible previously by writing an entire Material plug-in (which could be a major undertaking). See the base class for this plug-in type Class Shader for details.

Sampler Plug-Ins
This plug-in type is available in release 3.0 and later only.
This plug-in type works with the Standard material of release 3. A Sampler is a plug-in that determines where inside a single pixel the shading and texture samples are computed. The user interface of Samplers appears in the Super Sampling rollout in the Sampler dropdown. See Class Sampler for details.

Anti-Aliasing Filter Plug-Ins
This plug-in type is available in release 3.0 and later only.
This plug-in type is used for filtering and anti-aliasing the image. Documentation for the base class for these filters is in Class FilterKernel. Sample Code is available in the subdirectory **\MAXSDK\SAMPLES\RENDER\AAFILTERS**.

Shadow Generator Plug-Ins
This plug-in type is available in release 3.0 and later only.
The generation of shadows is accessible via this plug-in type. The standard 3ds max mapped and raytraced shadows have are plug-ins of this form. See Class ShadowType and ShadowGenerator for details. There is also a handy class for creating shadow map buffers. See Class ShadBufRenderer.

Sound Plug-In
A sound plug-in can take control of sound/music production in MAX. These plug-ins control not only the sounds they generate but also the system clock. They can thus coordinate the timing of external sound input / output devices with the animation. Sound plug-ins can provide their user interface as part of the 3ds max Track View.
Sound plug-ins are derived from Class SoundObj.

Color Selector Plug-In
This plug-in type is available in release 3.0 and later only.

This plug-in type provides the user with a custom color picker that appears whenever a standard 3ds max color swatch control is clicked. These plug-ins are selected in the General tab of the Preferences dialog. The color picker chosen is saved in the 3DSMAX.INI file in the "ColorPicker" section so that the choice is maintained between sessions. If the DLL for the selected color picker is not available, it will always default back to the "Default" color picker. See Class ColPick for details.

Front End Controllers
These plug-ins allow a developer to completely take over the 3ds max user interface. This includes the toolbar, pulldown menus, and command panel. See Class FrontEndController for details.

Motion Capture Input Devices
Motion Capture Input Device plug-ins can now be written that plug-in to the 3ds max motion capture system. See Class IMCInputDevice for details. Sample code is available in the subdirectory **`\MAXSDK\SAMPLES\MOCAP`**.

Image Viewer Plug-In
An image viewer is available from the 3ds max File menu under View File. A developer may replace the viewer DLL launched by this command to provide enhanced functionality for image browsing. The source code for this viewer is in **`\MAXSDK\SAMPLES\VIEWFILE\VIEWFILE.CPP`**. This plug-in is derived from Class ViewFile.

Notification Program
There is a program whose source code is in **`\MAXSDK\SAMPLES\UTILITIES\NOTIFY\NOTIFY.CPP`**. This program gets invoked by the network manager to handle network progress notifications.
A developer may write another "Notify" program in order to do any proprietary type of notifications. Note that "Notify" can be either a "*.exe", a "*.bat", or a "*.cmd" executable. This allows a user to create a simple script file to do something without having to resort to writing a binary program.
The current **`Notify.exe`** is very simple as it is used simply as a demonstration. It plays a different wave file for each of the event types. If invoked with no command line, it will bring up a dialog box asking the user to define each of the three wave files. The dialog has "Browse" buttons next to each wave file field which puts the user right into the Windows' "Media" directory where wave files are saved. There are also "play" buttons next to each sound so they can be tested.

4. Plugin Wizard

First of all, every one should know that to create plug-ins for 3D Studio Max, you can start from scratch by using a standard DLL Win 32 project or use the plug-in wizard for Visual Studio 6 or .NET.

The plug-in wizard helps you quickly create source code for template plug-ins. The plug-in wizard files must be set in the same directory as the others wizard for Visual Studio, just follow the instructions given with the wizard.

Figure 1: The 3D Studio Max plug-in wizard for Visual Studio .NET step 1 of 3

In the above figure, we can select the plug-in category we want to generate source code for.

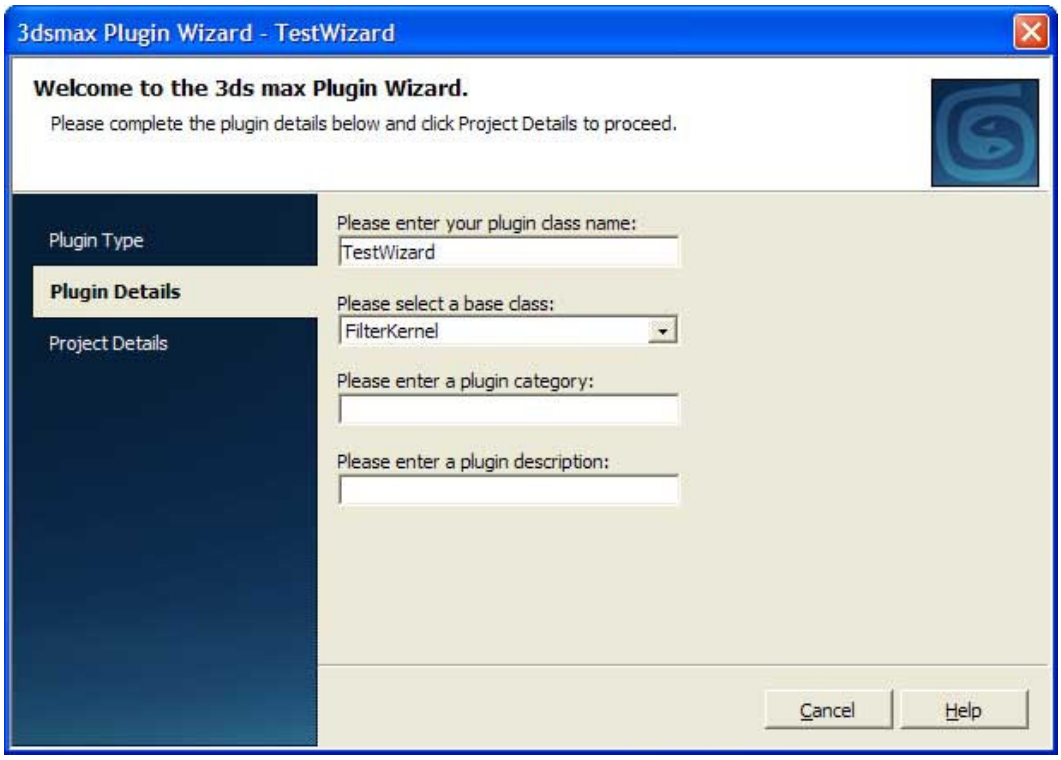

Figure 2 : 3D Studio Max SDK plug-in wizard step 2 of 3

In the figure above we set the name of our plug-in and its base class if several classes are available for this plug-in type. Then we enter a description as well as a category.

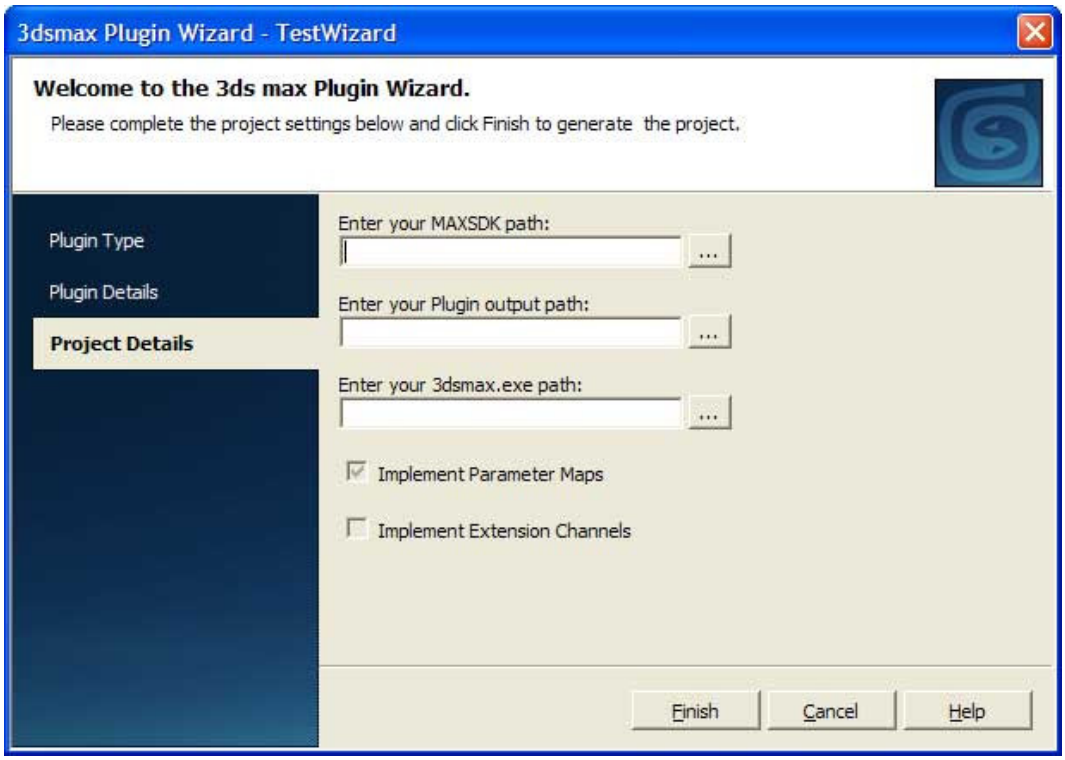

Figure 3 : 3D Studio Max Plug-in wizard step 3 of 3

In Figure 3, we enter our SDK full path directory, our plugin output path and our 3dsmax.exe directory so our settings are set correctly by the wizard.

5. 3D Studio Max plug-ins mechanisms

You can create plug-ins for 3D Studio Max. They are classical Win32 dynamic link libraries (DLLs). These DLLs are loaded from directories defined by default in MaxExeDirectory\stdplugs and MaxExeDirectory\plugins where MaxExeDirectory is the path for the 3dsmax.exe file. Usually "c:\3dsmax6".
You can also add some plugins paths into the user interface (UI) by using the menu "customize" then the sub-menu "configure paths" and then plug-ins.
These paths are usually saved in the 3dsmax.ini file in the same path as the 3dsmax.exe directory.

5.1. Functions every Max plug-in should implement

Each DLL is loaded by the 3D Studio Max Plugin Manager to know which classes are implemented and from which classes they are subclassed.

To do so, the Plugin Manager uses the 4 following functions :

```
DLLMain() : classical function we found in every DLL.

LibDescription() : the DLL description, it returns a character string that
is used to be presented at the user when the DLL is not present while this
DLL has been used in the scene the user is trying to load.

LibNumberClasses() : In a single DLL, we can define several 3D Studio Max
plug-ins (which is very rare in practice and should not be used if that's
possible). So this function returns the number of classes (  = plug-ins)in
this DLL.

LibClassDesc() : This returns an instance of ClassDesc class which contains
the description of the plug-in(s) implemented in this DLL.

LibVersion() : This function returns the version of 3D Studio Max SDK used
to compile this DLL, if the version used is not the same as the current 3D
Stusio Max application, an error message is shown and the DLL is not
loaded.
```

Usually, a plug-in made for a specific version of 3D Studio Max must be recompiled for each new version of 3D Studio Max released.
Except from Max 4 to Max 5 the constant defined to VERSION_3DSMAX which contains the version of 3D Studio Max had not changed.

The SDK provides you a global function to retrieve the 3DSMax version :

```
DWORD Get3DSMAXVersion();
```

5.2. Plug-ins files extension

3D Studio Max plug-ins are not .DLL files. They use their own file extension. One of each plug-in category.

Example :
.DLU files : Plugin Utility
.DLI/.DLE files : Plugin Import/ Export
.GUP files : Global Utility Plugin.
.DLM files : Modifiers Plugin
etc

Warning :

Be careful, this file extension is used only for developers to know which plug-in category is this DLL, but the Plugin Manager doesn't care about the file extension.
Consequently, removing / changing a file extension is not enough to prevent a plugin from being loaded by the Plugin Manager. To do so, you will have to remove it from the directories where plug-ins are loaded.

5.3. Loading / saving plugin data

Each plug-in can save its own local data in the Max scene file. It will be saved in a binary format as a chunk of the Max file format. That's why it is difficult to be able to read a Max file directly without launching the Max exe application. If the plug-in developer saves data into a scene, the scene could not be opened correctly without its plug-in loaded.

Note :
Since 3D Studio Max release 6, it is possible to read a Max file without launching the 3dsmax.exe application. You can download this on the Sparks web site.

Most of plug-ins categories have a load / save functions to let you load / save custom data into a Max scene. These data are saved as chunks and not linearly. This lets you do versions of your plug-in and ensure backwards compatibility.

Example of a save function with a code snippet from the UVW Unwrap modifier plug-in :

```
#define VERTCOUNT_CHUNK             0x0100
#define FACECOUNT_CHUNK             0x0230
IOResult UnwrapMod::Save(ISave *isave)
{
      ULONG nb;
      Modifier::Save(isave); //Call the base class Save

      int vct = TVMaps.v.Count(), fct = TVMaps.f.Count();

      isave->BeginChunk(VERTCOUNT_CHUNK);
      isave->Write(&vct, sizeof(vct), &nb);
      isave->EndChunk();


      isave->BeginChunk(FACECOUNT_CHUNK);
      isave->Write(&fct, sizeof(fct), &nb);
      isave->EndChunk();

      etc.
}
```

And to read these data saved in the file :

```
IOResult UnwrapMod::Load(ILoad *iload)
{
	IOResult res;
	ULONG nb;
	Modifier::Load(iload); //Call the base class Load

	int ct, i;

	//check for backwards compatibility
	while (IO_OK==(res=iload->OpenChunk()))
	{
		switch(iload->CurChunkID())
		{
			case VERTCOUNT_CHUNK:
				iload->Read(&ct, sizeof(ct), &nb);
				TVMaps.v.SetCount(ct);
				TVMaps.cont.SetCount(ct);
				vsel.SetSize(ct);
				for (i=0; i<ct; i++) TVMaps.cont[i] = NULL;
			break;
			case FACECOUNT_CHUNK:
				iload->Read(&ct, sizeof(ct), &nb);
				TVMaps.f.SetCount(ct);
			break;
	}
}
```

6. Debugging under 3D Studio Max SDK

We will focus in this chapter on tips and tricks to debug plug-ins under 3D studio Max SDK.

6.1. English version

It is highly recommended to use an English language version of 3D Studio Max for developing 3D Studio Max plug-ins. This is only due to the fact that the SDK documentation is in English as well as the user interface. So it's really easier for a developer to have the software application, its menus and its documentation using the same language and obviously the SDK documentation is in English only as far as I know.

Note 1 :
Another drawback of using another version that the English language version is that features are translated into the local language. For example, the Character Studio plug-in is completely translated into French in its French version. So the bones' names are not any longer such as "Bip01 FootSteps" but the same translated into French language so you can't rely on names of features.

Note 2 : Object names are not necessarily unique anyway under 3D Studio Max…

6.2. SDK documentation and source code samples

The SDK documentation is very complete and kept up to date on the Sparks web site (we will talk later about what is the Sparks program). It is compiled using DOxygen or such tools to get information from the .h and .cpp files. The documentation also exists in CHM file format.

There are a lot of source code samples freely given with the SDK, they are very complete.

They are often complex for people beginning with 3D Studio Max programming, but anyway you should take time to have a full look at what they contain to know which fields are covered by these plug-ins even if you do not go into their source code details.

6.3. Multithreading

You should know that 3D Studio Max plug-ins should be multithreaded compliant. Because some 3D Studio Max core features use multiple threads, e.g : the renderer.
So all plug-ins functions could be re-entered by using another thread, or your plug-in could be launched several times by the user if you allow this…

These constraints are :

- Use the Debug Multithreaded and Multithreaded run-time C libraries to generate your source code. This is made by default in the settings if you use the plug-in wizard.
- Use « critical sections », see the MSDN documentation about Synchronization Objects to know what they are. If you use statics or globals variables or access some files or some other tasks that could result in a share violation if you access it several times at the same time. The functions must be thought as re-entered and could be executed asynchronously.

Note :
This is not true for all types of plug-ins. For example, if you do a utility and ensure that only one instance at a time could be launched even by using several 3D Studio Max applications, you will not need to take care of these constraints.
But in plug-ins such as modifiers plug-ins this is really necessary as you don't know when the core system will call your plug-in.

6.4. Release & Debug versions of 3D Studio Max

The installation version of 3D Studio Max is obviously a version compiled in release build. This means that when you will compile your plug-in in a debug build for this application, your plug-in will be in debug build while the application will be in release build. This will result in several problems.

First of all, you will not be able to get a call stack to show you where your code has crashed if this happens. This can be a big issue. This is due to the fact that no debugging information are set in the release build of 3D Studio Max.

Some problems of memory allocation/deallocation on the heap can happened. 3D Studio Max uses the release build of standard C run-time DLLs and your plug-in when compiled in debug build uses the Debug C run-time Dlls. The debug and release versions of a DLL do not use the same way to allocate/deallocate memory so it can cause exceptions when allocated memory in a release build is deallocated by the debug build and the other way round…

To solve the memory problems, a special build hybrid configuration is usually used in 3D Studio Max plug-ins development. It is called the "Hybrid" build and is covered in detail in the next section.

6.5. The Hybrid build configuration

This build configuration is a mix between debug and release configuration. It ensures memory allocation / deallocation will be made correctly using the same builds of standard C run-time DLLs in your plug-in and in the 3D Studio Max application.

You need to create another build configuration in your project (or solution).
Then you copy the settings from a debug build and you change the code generation to the Multithreaded DLL instead of Debug multithreaded DLL.

7. ADN Sparks : the Autodesk® developer network program

For all 3D Studio Max developers, the reference web site every programmer should know is :

http://www.autodesk.com/adnsparks

This web site is designed for all Autodesk's products developers. You can get source code samples, and a lot of others services. Some services are free while some are not.

Let's see in details what are the services provided by this web site :

7.1. Services

You can subscribe to the Sparks program. By doing so, you will receive a unique identifier to login on this web site. Then you will be able to access debug tools and additional source code samples.

For example : you can get a debug build of 3D Studio Max which will let you use the debug build of your plug-in and not the hybrid buid… so you could have a call stack when an exception occurred and you will be able to trace your code as well as core 3D Studio Max code as additional source code is provided in this debug build, e.g : mesh implementation, animation interpolation etc...

You also have an access to the technical support from Autodesk developers and submit them your questions and problems. You usually get an answer in 3 - 4 business days. This may vary.

About the pricing of this services, I will let you have a look at the web site because the prices may have changed when you will read this document.

Sparks members can choose between the 2 different services :

- Standard membership : for 1 developer only and for 1 year. This option lets you have access to the debug SDK and some additional source code but you don't have access to the technical support.
- Premium membership : up to 5 developers for 1 year. You get access to the same features as the standard memberships and have a access to the technical support.

7.2. SDK Support Webboard, Knowledge base, online documentation

There are webboards (forums) that are free to all 3D Studio Max developers. It's where the community of developers meet and ask their questions and talk about their problems using 3D Studio Max SDK.

Note :
As these forums are free of charge, nobody is forced to give you an answer to your problems which is not the case when you subscribe to Sparks and use the technical support from Autodesk developer named "Incident tracking".

There are at the following web address :

http://sparks.discreet.com/webboard/wbpx.dll/~maxsdk

On this page, you can create a login and get an access to the forums on miscellaneous topics about Max SDK development. The same forums exist for the Maxscript community.

Note :
Avoid asking the same question on several forums, cross-posting is unappreciated.

There also exists a Knowledge Base on the Sparks web site that provides you most answers to classical beginers questions. It is free of charge too.

And finally, the SDK documentation. It is installed with the SDK on your local drive but you should rather use the online version available on the Sparks web site. This online version is always up to date.

Sparks also gives an addin to extend Microsoft Visual Studio. It adds 3 buttons to the UI of Visual Studio one button for each features we have just mentioned, that is : SDK online documentation, Knowledge Base and SDK Forums. This addin exists for VC6 and VC7+ (provided with Max 6 SDK).

Let's now see in detail the SDK.

8. The SDK classes hierarchy

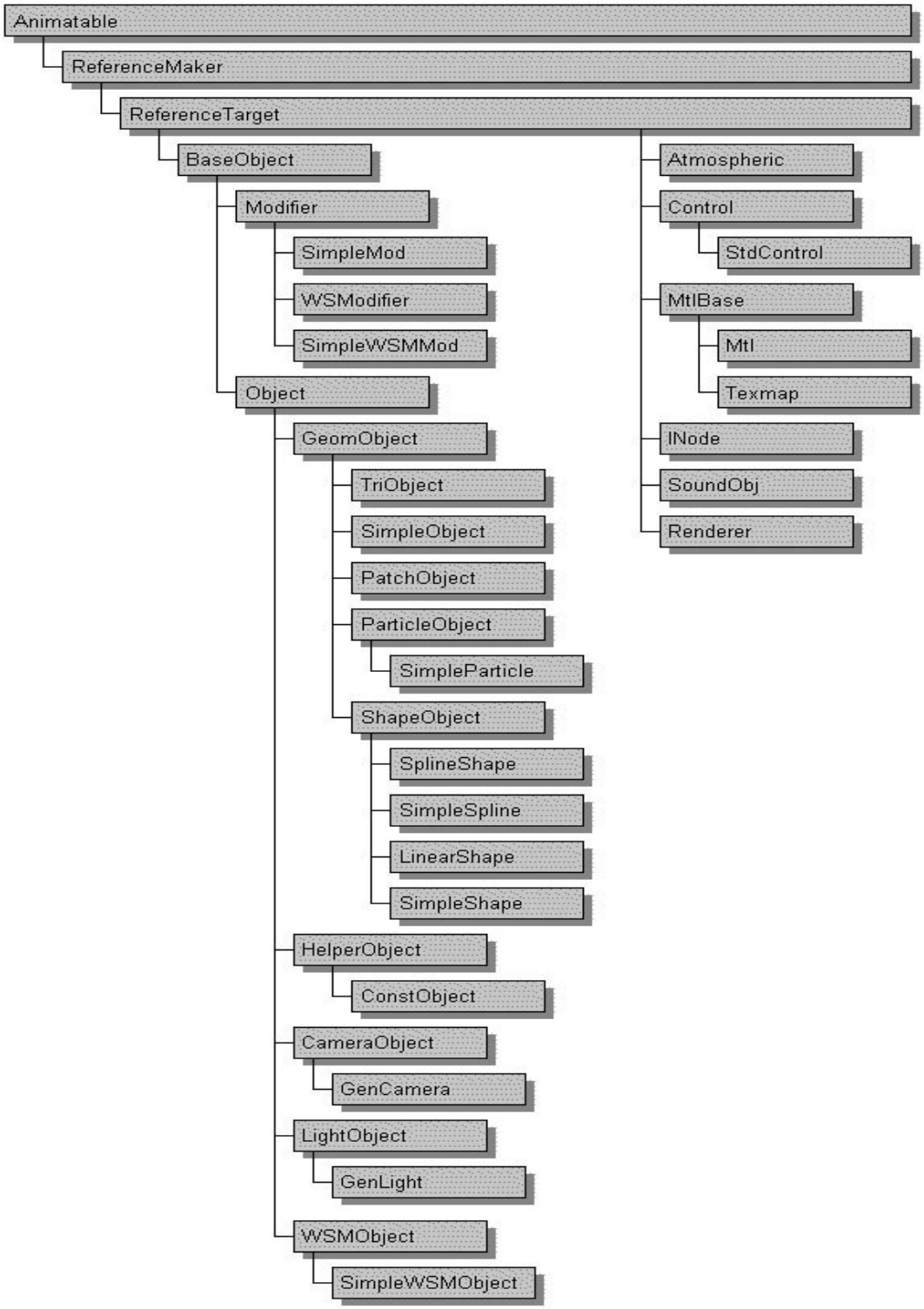

Figure 4 : hiérarchie de classes du SDK

To remember : All classes are subclasses of Animatable which means everyting can be animated !

9. Class IDs & Super Class IDs

All Max classes have two unique identifiers by class (not by instance of a class). They are called Class_ID and SuperClass_ID. You can think of it as the same system as the C++ RTTI (Run-Time Type Information). These class_ID are used to create an instance of a specific class using a factory class (see a book about Design Patterns to have further details about what are factories class).

These Class_IDs are made with a pair of 32 bits integers. Each pair is unique. For example a plugin has its own Class_ID to be identified uniquely by the system.
So when you create a new plugin, don't forget to change its default class_ID. The MaxSDK\Help directory contains an executable file named Gencid.exe which generates a Class_ID you can copy/paste on your plug-in source code.

Note :
If you use the plug-in wizard to create a new plugin, the Class_ID generated is already unique, so there is no need to change it.

Example of Class_ID :

```
Class_ID MyclassID(0x73e10f89, 0x44672278);
```

You can have a look at all class_ID defined in the SDK by looking at the plugapi.h header file in the folder MaxSDK\include .

For example, a triangle mesh, called TriMesh has a class_ID of :

```
#define TRIOBJ_CLASS_ID = 0x0009 //In plugapi.h
Class_ID( TRIOBJ_CLASS_ID, 0 ) //TriOBJ = Triangle Object =
                                  //Triangle Mesh
```

Most of the time, the SDK uses only the first part of the Class_ID, that is the first integer which is called part A in the SDK. The second integer which is called part B is rarely used and often set to 0. This is the case for our TriMesh class_ID.

The SuperClass_ID is the identifier of the super class. For example, for the TriMesh example, it's GEOMOBJECT_CLASS_ID which means Geometric object. You think of the Superclass_ID as the category of the object.

All classes from Max SDK inherit from Animatable class which contains the functions :

```
Class_ID Animatable::Class_ID()
SClass_ID Animatable::SuperClassID();
```

Consequently, when we get a Max object, say an animation controller, we can check of which type it is by using its Class_ID :

```
Control* _cont ; //valid pointer of my controller.
if ( _cont->ClassID() == Class_ID( TCBINTERP_POSITION_CLASS_ID, 0 ) )
{
      //This anim. Controller is a TCB interpolation controller
}
```

Let's now see the lost important class of the SDK, namely Interface class.

10. The Interface class

To remember : If you should know about only one class from Max SDK, then remember the Interface class ! ☺ It's the most useful class of the SDK !!! It is necessary for all plug-ins categories.

To get an instance of that class, there is a global function :

```
//ip = everywhere in the SDK, ip = Interface Pointer
Interface* ip = GetCOREInterface() ;
```

Note : Usually, this pointer should never bu NULL when using the previous function. However, plug-ins may save an instance of this pointer and set it to NULL when they have finished their job, so be careful with this pointer, check its value or use an assertion…

In all examples of this document we will refer to "ip" as the interface pointer. The same convention is used in the SDK samples and documentation.

Here are some examples of what the Interface class lets you do :

- Read / Save / Merge Max files
- Get/Set the current time, set the animation parameters
- Interact with the user interface
- Set call backs functions (see section 10.4) on the viewport, the animation time, the selection etc…
- Set command modes to deal with the mouse events (see section 22)
- Etc.

Let's see some of the main features of this class :

10.1. Create objects

The function of the Interface class named CreateInstance enables you to create all kinds of Max objects.

```
virtual void * CreateInstance (SClass_ID superID, Class_ID classID)=0;
```

Example : create a camera object from its class_ID.

```
//Create a camera
GenCamera *cob = (GenCamera *) ip->CreateInstance(CAMERA_CLASS_ID,
Class_ID(SIMPLE_CAM_CLASS_ID,0));
cob->Enable(1);
INode* _CameraNode = ip->CreateObjectNode(cob);
```

This function uses the superclass_ID et class_ID of an object to create it. It's a factory function.

10.2. Dealing with a nodes' selection

Nodes are the names set to Max objects. A node can be a triangle-mesh as well as a patch-mesh or a spline or a camera, a light etc…

The Interface class propose miscellaneous functions to deal with the current nodes's selection

```
int Interface::GetSelNodeCount();
INode* Interface::GetSelNode(int i);
```

- GetSelNodeCount returns the number of selected nodes
- GetSelNode return the ith selected node.

Example :

```
//Number of selected nodes
const int NumSelNodes = ip->GetSelNodeCount();

for ( int i = 0; i < NumSelNodes; i++ )
{
	//Get i-th selected node
	INode* _node = ip->GetSelNode(i);
	If (_node) //Should always be true
	{
		DoSomethingOnTheNode(_node);
	}
}
```

This is often used in the utility plug-ins to act on a user nodes' selection.

Next we are going to see the viewport functions from Interface class.

10.3. Viewport

There are several functions dealing with the viewport, the following list is not exhaustive, but these functions are the main functions to know in our opinion.

```
void RedrawViews(TimeValue t, DWORD vpFlags=REDRAW_NORMAL, ReferenceTarget
*change=NULL);
```

The RedrawViews function is used to update the viewports.
First parameter TimeValue t is the time at which we want to update the viewports. This parameter is usually set to the current time that we get by :

```
TimeValue t = ip->GetTime() ;
```

Second parameter is an enum : DWORD vpFlags which can take the following values :

- REDRAW_BEGIN – Call this before you redraw..
- REDRAW_INTERACTIVE – In case you would like to degrade the views when they are changed interactively.
- REDRAW_END – set back the undegraded view.
- REDRAW_NORMAL – redraw views in an undegraded state.

Third parameter is never used and should not be used.

This function redraws what has been invalidated in the scene.
It is usually used like this :

```
ip->RedrawViews(ip->GetTime(),REDRAW_BEGIN);
// More code ...
ip->RedrawViews(ip->GetTime(),REDRAW_INTERACTIVE);
// More code ...
ip->RedrawViews(ip->GetTime(),REDRAW_END);

or

ip->RedrawViews(ip->GetTime()); //REDRAW_NORMAL
```

Here is another function :

```
void ForceCompleteRedraw(BOOL doDisabled=TRUE);
```

The parameter lets you specify if the viewport that are disabled should be redrawn or not.
It is used to redraw all viewports, this function is guaranteed to be slow.

However, it is the function that we use the most often…

Here are 3 useful functions dealing with the viewport to interact with it :

```
virtual BOOL      SetActiveViewport ( HWND hwnd );
virtual ViewExp * GetActiveViewport ();
virtual void      ReleaseViewport   ( ViewExp *vpt );
```

The ViewExp class is the Viewport class. It provides you an access to all viewport parameters.

SetActiveViewport is used to set a viewport as active from its HWND (Win32 window handle).

GetActiveViewport is used to get the active viewport. By active we mean the viewport currently selected.

Finally, ReleaseViewport must be called once we have finished using an instance of a ViewExp class that we have get using GetActiveViewport.

10.4. Callback functions

Every one should know the callbacks functions mechanism which lets you be called by the system when an event occur by calling a specific function you have set as a call back.

The SDK provides you the same mechanisms in the Interface class. We are going to present in the following paragraph the functions to register a call back, we will let the user see the documentation about unregistering these callbacks.

```
void RegisterTimeChangeCallback(TimeChangeCallback *tc)=0;
void RegisterCommandModeChangedCallback(CommandModeChangedCallback *cb)=0;
void RegisterViewportDisplayCallback(BOOL preScene, ViewportDisplayCallback
                                     *cb)=0;
void NotifyViewportDisplayCallbackChanged(BOOL preScene,
                                          ViewportDisplayCallback *cb)=0;
void RegisterExitMAXCallback(ExitMAXCallback *cb)=0;
void RegisterAxisChangeCallback(AxisChangeCallback *cb)=0;
void RegisterRedrawViewsCallback(RedrawViewsCallback *cb)=0;
void RegisterSelectFilterCallback(SelectFilterCallback *cb)=0;
void RegisterDisplayFilterCallback(DisplayFilterCallback *cb)=0;
```

So we can set a call back when the following events happen (same order as functions above) :

- The current animation time has changed
- A new command mode has been set (see section 22)
- The viewport has been redrawn/updated
- A new callback has been set on the viewport
- Max is being shut down by the user
- The constrainst axis has been changed
- The selection object filter has been changed
- The display object filter has been changed

However, there is another structure used to set callbacks on events, it is the NotifyInfo structure.
A global function is used to register these callbacks on a lot of other events :

```
int RegisterNotification(NOTIFYPROC proc, void *param, int code);
```

We will let the user look at the documentation to know which events can have callbcaks set on them.

Note :
Instead of taking as a parameter a global / static function to be used as a callback such as the Win32 API uses, 3D Studio Max has chosen to use what we call the “functors”.
Functors are classes that contain one pure virtual function that is used as a callback. We instanciate a subclass of this functor class where we have defined the virtual function. And this function will be used as a callback.
This mechanism is interesting because you can’t make a mistake by giving the system a wrong function pointer to be used as a callback as this is possible in Win32 API. By using a

function pointer, no checking about the function prototype is made at compile time. So if you have set a wrong function, it will crash at run-time only.

Using a functor, you will detect at compile time a mistake because the prototype of the function to be used as a callback is known at compile time in the class because of the C++ virtual table mechanism.

11. The INode class

It's is the class to represent objects in the 3D scene under 3D Studio Max.
Nodes are for example :

- Triangle-meshes
- Tri-patches meshes
- Cameras
- Lights
- etc.

We get an read/write access to miscellaneous node's properties such as :

- its name
- its parent node, its children nodes in the nodes hierarchy
- its references on other nodes
- its display attributes, rendering attributes, vertex colors
- Its 3D transform which places it in the 3D space
- Its animation controllers
- Its material
- etc.

A node contains exactly 6 references on (see section 19 to go deeply into the reference system) :

- An animation controller for its 3D transform
- The object reference. Each node maintains a pointer to an object this a pointer to the base procedural object or derived object (BaseObject or IDerivedObject see section **Erreur ! Source du renvoi introuvable.**).
- The « Pin Node » for Inverse Kinematics (set to NULL by defaut).
- The material reference
- The visibility controller
- The image blurr controller (set to NULL by defaut)

To remember : It exist a node which is the highest node in the hierarchy. It is a virtual node called the root node. You can't select or see this node from a user point of view. When you create a new node, it is automatically set as a child node of the root node. You can get this root node by using INode* Interface::GetRootNode(). The root node pointer should never be null.

When we get an INode pointer we can check if it's a camera, a triangle-mesh, a light or whatever by checking its class_ID, let's see how to get this class_ID :

```
//Get current time from UI
TimeValue t = ip->GetTime();

//Evaluate obejct
ObjectState os = _nodePtr->EvalWorldState(t);
if( os.obj != NULL )
{
      TSTR ClassName;

      //Get class name
      os.obj->GetClassName( objClassName );

      switch( os.obj->SuperClassID() ) //récupère son super class ID
      {
            case GEOMOBJECT_CLASS_ID:
                  //It's a geométric object
                  //Let's compare its Class_ID with triangle-mesh class_ID
                  If (os.obj->Class_ID() == Class_ID( TRIOBJ_CLASS_ID, 0 ))
                        //It's a triangle-mesh.
            break;

            case CAMERA_CLASS_ID :
                  //It's a camera,Free or
                  //Target ? We could answer by looking at its Class_ID !
            break;
      }
}
```

The EvalWorldState function is used to evaluate the object. We will go into detail with this function in the modifiers pipeline in section **Erreur ! Source du renvoi introuvable.**.

To remember :
We can only assign one and only one material per node. That would mean only one texture bitmap per node ?
Fortunately no. If you need to have several texture bitmaps applied on the same node when it's a mesh for example, the SDK provides you a material called multimaterial which is a container of materials, it is called Multi/Subobject material in the UI.

Here is how we get the material applied on a node :

```
INode* node ;
Mtl* mtl = node->GetMtl() ;
```

We will get back on materials in section 12.

11.1. Nodes instances / references

It is possible for a node to be built from another node. This is the case when having a reference or instance node. It is a node that depends on another node's topology.

Note :
You must not think of instances and references in the SDK as C++ language instances or references. They have no common properties.

By using instances and references we can share the data between nodes so use less memory in a scene. This is often used in video games engine.

The difference between SDK instance and reference is that the modifications on an instance will have an impact on the node which was the base node of this instance while modify a reference will have not impact on the base node.
So for an instance, it's a bidirectional relationship while for a reference it's a unidirectional relationship.

In the versions prior to 3D Studio Max 6, it is possible to know which nodes are instances/references by checking their internal references (see section 19).

Since 3D Studio Max 6, a class has been created to deal with instances and references of a node. It is called IInstanceMgr. See the header file named iInstanceMgr.h in the SDK.

11.2. The nodes' 3D transforms

Let's introduce first what is the node's pivot. It is the center of its rotation and scale. It means if you rotate the node, it will rotate around its pivot point.

When you get the vertices and normals of a mesh, they are expressed in local coordinates. That is in the pivot coordinates.

So, the pivot defines :

- The center of the rotation and scale
- The origin of the transform for children nodes in the nodes' hierarchy.
- The origina of the « joint » for inverse kinematics.

Nodes have 2 transforms. Each is called a Matrix3 (see section 23 to know more about Matrix3). These transform are :

- The transform matrix of the node called NoteTM which is the pivot transform in the 3D world coordinates.
- The Object offset transform, which represents the offset between the object an dits geometry.

For example, it is possible to move the pivot of a node outside of its mesh. This is realized internally by modifying the object offset transform.
It has no influence on the children nodes of the hierarchy.

Notes :

- If you have made a ResetXForm on a node, the object offset transform has been reset to the identity matrix and the difference between object and its geometry has been applied directly on the mesh vertices coordinates.
- We use indifferently the node's transform or pivot transform to describe the same 3D transform.

To get the object offset transform components, we use the following functions :

```
void        INode::SetObjOffsetPos(Point3 p) ;
Point3      INode::GetObjOffsetPos() ;
void        INode::SetObjOffsetRot(Quat q) ;
Quat        INode::GetObjOffsetRot() ;
void        INode::SetObjOffsetScale(ScaleValue sv) ;
ScaleValue  INode::GetObjOffsetScale();
```

Then to move the geometry, the pivot or both you have the following functions :

```
void INode::Move(TimeValue t, const Matrix3& tmAxis, const Point3& val,
                 BOOL localOrigin=FALSE, BOOL affectKids=TRUE,
                 int pivMode=PIV_NONE, BOOL ignoreLocks=FALSE);

void INode::Rotate(TimeValue t, const Matrix3& tmAxis, const AngAxis& val,
                 BOOL localOrigin=FALSE, BOOL affectKids=TRUE,
                 int pivMode=PIV_NONE, BOOL ignoreLocks=FALSE);

void INode::Rotate(TimeValue t, const Matrix3& tmAxis, const Quat& val,
                 BOOL localOrigin=FALSE, BOOL affectKids=TRUE,
                 int pivMode=PIV_NONE, BOOL ignoreLocks=FALSE);

void INode::Scale(TimeValue t, const Matrix3& tmAxis, const Point3& val,
                 BOOL localOrigin=FALSE, BOOL affectKids=TRUE,
                 int pivMode=PIV_NONE, BOOL ignoreLocks=FALSE);
```

The parameters are :

1. t is the time where this operation occurs.
2. tmAxis is the axis system in which happens this move (usually set to the identity matrix). But you could use this to make it move in its aprent coordinates for example.
3. val is the delta of translation/rotatio/scale to add. Warning it is not the final value but the difference between the final and present values.
4. localOrigin specifies if the move takes place in local or world coordinates
5. affectKids specifiifes if the move affects the children nodes
6. pivMode is an enum of the values :
    - PIV_NONE : Moves the pivot as well as the geomtry at the same time
    - PIV_PIVOT_ONLY : Moves the pivot only
    - PIV_OBJECT_ONLY : Moves geometry only
7. ignoreLocks tells if the locked nodes should be affected or not by thismove.

We can also get/set the node's 3D transform in world coordinates by using :

```
Matrix3 INode::GetNodeTM(TimeValue t, Interval* valid=NULL);
void INode::SetNodeTM(TimeValue t, Matrix3& tm);
```

The first parameter is the time where we want this to happen.
In GetNodeTM, the second parameter which is an interval is intersected with the 3D transform validity interval.
It is usually set to the Interval value : FOREVER…

Note :

The 3D transform returned by GetNodeTM does not include the object offset transform but it includes the 3D transform of the parent node if any.

We use it this way :

```
//Get node transform
Matrix3 nodetm = node->GetNodeTM(ip->GetTime());
```

To get a node's transform with both the object offset and parent transforms included you should rather use :

```
Matrix3 INode::GetObjectTM(TimeValue t, Interval* valid=NULL);
```

The transform you get by using the previous function is the transform before any world space modifiers are applied. World Space Modifiers are plug-ins that change the node's transform from various parameters.

So it is recommended to use the following function to get the whole transform with anything applied :

```
Matrix3 INode::GetObjTMAfterWSM(TimeValue time, Interval* valid=NULL);
```

Ok, now to get the parent transform you can use :

```
Matrix3 INode::GetParentTM(TimeValue t)=0;
```

So a simple example to get the local transform of a node, we mean without its parent transform applied would be :

```
Matrix3 parentTM  = node->GetParentTM(ip->GetTime());
Matrix3 nodeTM    = node->GetNodeTM(ip->GetTime());
Matrix3 localTM   = nodeTM*Inverse(parentTM);
```

Note :
Since 3D Studio Max 6, the SDK provides you new functions to deal with pivot transform that are :

```
void INode::CenterPivot(TimeValue t, BOOL moveObject);
void INode::AlignPivot(TimeValue t, BOOL moveObject);
void INode::WorldAlignPivot(TimeValue t, BOOL moveObject);
void INode::AlignToParent(TimeValue t);
void INode::AlignToWorld(TimeValue t);
void INode::ResetTransform(TimeValue t,BOOL scaleOnly);
void INode::ResetPivot(TimeValue t);
```

12. Materials

They are really necessary to make our 3D scene more realistic. In 3D Studio Max, you have the material editor which is very complex :

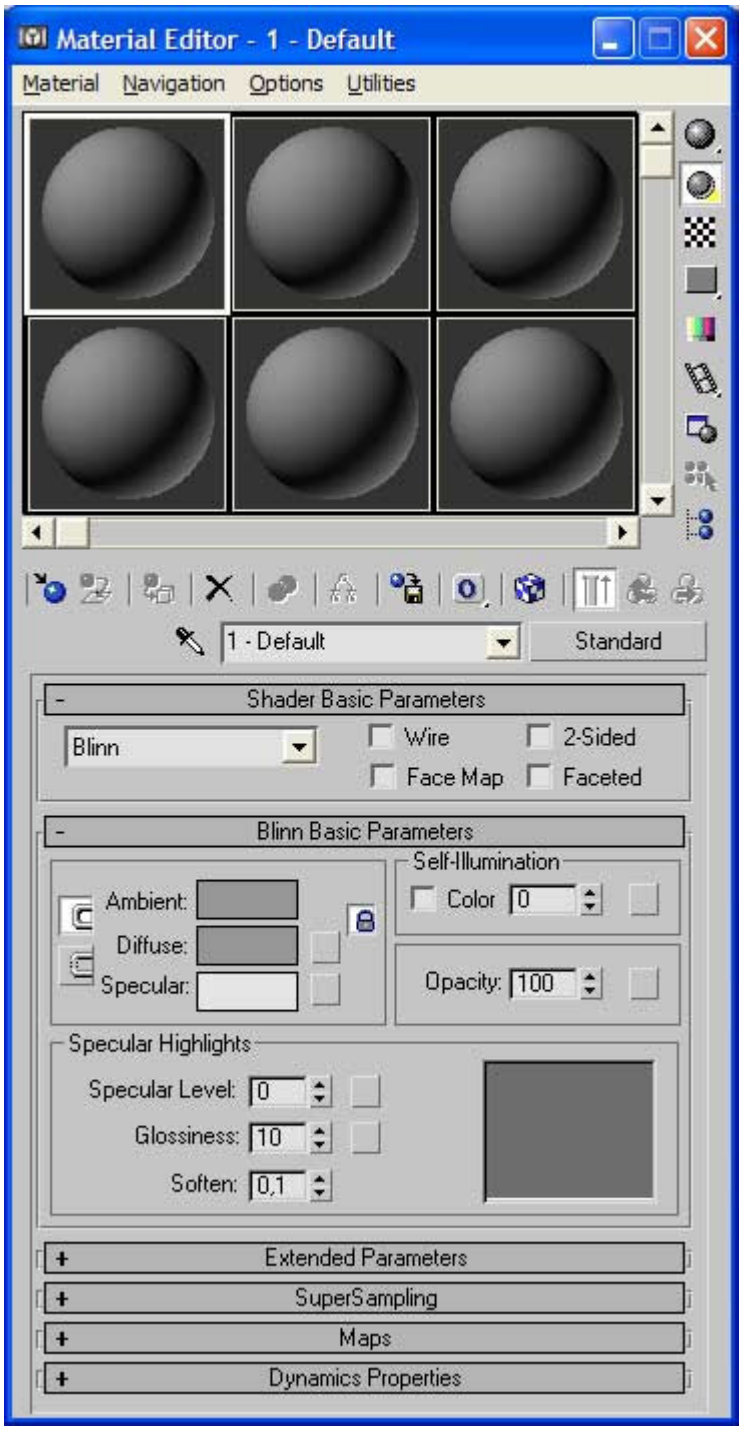

Figure 5 : The material editor

12.1. Materials philosophy

The material editor is very complete so very complex too. It helps you create effects, shaders etc.. It is also possible to use Microsoft Direct X shaders as well as NVidia CG shaders in the viewport in real time.

We are not going to describe all functionalities of the material editor in this section, but we are going to focus on the most used functions to get information from materials.

Let's see the standard materials which are the most common/basics materials.
They contain several slots for textures of type TexMap. TexMap is the base class for procedural textures, bitmap textures, vectorial textures and others textures…
Main textures slots are :

- ambient
- diffuse
- specular
- opacity
- etc..

Let's see how to get the material from a node and see its type :

```
INode* node ; //Is a valid node pointer
Mtl* _mat = _node->GetMtl();
if (! _mat)return;

//Is it a Standard Material ?
if ( _mat->ClassID() != Class_ID( DMTL_CLASS_ID, 0 ) )//Default material ?
      return;

//Yes, it is a StdMat, so cast it
StdMat* stdmat = static_cast<StdMat*> ( _mat );
```

Note :

- The Class_ID of the standard material is DMTL_CLASS_ID which means default material !
- **To remember : One and only one material can be assigned to a node at the same time.**

How from this standard material (StdMat) can we get the texture bitmap if any ?
Most of the time, the texture bitmap we want to get is in the diffuse texture slot.

The SDK provides you access to the material textures slots by using indices that are defined by :

- ID_AM : Ambient (= 0)
- ID_DI : Diffuse (= 1)
- ID_SP : Specular (= 2)
- etc…

So a complete example to get the texture bitmap from the diffuse slot of a material is :

```
//Get the Texmap in Diffuse slot
Texmap* TMap = stdmat->GetSubTexmap( ID_DI );
if ( ! TMap || (TMap->ClassID() != Class_ID( BMTEX_CLASS_ID, 0 ) ) )
      return;

//Cast it into a bitmaptex
BitmapTex*  BmapTex = static_cast<BitmapTex*> (TMap);

//Get the bitmap full path name
const TCHAR* BitmapFullpathName = BmapTex->GetMapName();

//Now get the bitmap from this BitmapTex
Bitmap* bmp = BmapTex->GetBitmap( ip->GetTime() ) ;
```

In this example, we have taken the teture bitmap from the diffuse slot by using the index ID_ID. Using ID_SP would have let us get the bitmap from the specular slot.

Then we have used the GetMapName function to get the name of the texture bitmap. It is in fact its full path name (with drive and directories concatenated to its name such as “c:\3dsmax6\bitmaps\mybitmap.tga”)

Finally, we have taken the Bitmap instance from the BitmapTex. Using the Bitmap class would let us access the pixels as well as miscellaneous properties of the bitmap such as its width, height and color depth.

12.2. What are the materials containers ?

There are 2 materials containers in a 3D Studio Max scene :

- Scene materials
- Current material library
- Material editor

The scene material are those which are applied on at least one node in the scene. They are set in a special library called the “scene materials library”.

Note :
Warning, when accessing the scene material library, it may be not up to date. This mibrary is updated when you load/save/merge… the scene. So to be sure it is up to date, save your scene before using it.

We can get the scene material library by using the Interface class function : Interface::GetSceneMtls.

Here is an example :

```
//Get the scene materials library
MtlBaseLib* scenematlib = ip->GetSceneMtls();

int i=0;

if (scenematlib)
{
	//Be careful, the scene materials can not be up to date
	//To ensure it is up to date you should save the current Max file.
	const int numscenematlibs = scenematlib->Count();
	for(i=0;i<numscenematlibs;i++)
	{
		MtlBase* mtl = static_cast<MtlBase*> ( (*scenematlib)[i] );
	}
}
```

So we can get the material with their super class MtlBase. Mtl is a subclass of MtlBase. By testing the class_IDs of these materials you can cast them into the suitable class pointer.

3D Studio Max also uses a current material library in which you can get/set materials programmatically or manually by using the material editor. This material library can be saved as a file so it can be loaded in another scene.

We can get this library by using the following function Interface::GetMaterialLibrary. See the following example :

```
//Get the current material library
MtlBaseLib& currentmatlib     =  ip->GetMaterialLibrary();
const int numcurrentmatlib    = currentmatlib.Count();
for(i=0;i<numcurrentmatlib;i++)
{
	MtlBase* mtl = static_cast<MtlBase*> ( currentmatlib[i] );
}
```

And finally, the last material container is the material editor which can contain up to 24 slots for materials.
You can get/set materials into the material editor (use Interface::PutMtlToMtlEditor as a starting point)

So to get the materials from material editor we use Interface::GetMtlSlot this way :

```
//Get the material editor's materials
const int NBSlotMEditor = 24;
for (i=0; i<NBSlotMEditor; i++)
{
      //Get Material
      MtlBase* Mat = ip->GetMtlSlot(i);
}
```

13. The Mesh class

We have seen how to check if node is a mesh. Now we are going to see how to get the mesh from the node. We will only focus on triangles meshes but the same principles remains for a patch mesh.

To do so, we need to use a subclass of Object class called TriObject (object made of triangles). In the Oject class there are the functions CanConvertToType and ConvertToType. They are used to know if an object can be converted into a certain type and convert it when possible. Using this you can convert a triangle mesh into a patch mesh for example.

To get the mesh from a node, we use the following function which can be found in the SDK while it is here in a slightly modified version :

```
TriObject *GetTriObjectFromNode(INode *node, BOOL &deleteIt)
{
	deleteIt	= FALSE;

	//Get current time from UI
	TimeValue t = ip->GetTime();

	//Evaluate object at current time
	Object *obj = node->EvalWorldState(t).obj;

	//Can we convert it in a TriObject ?
	if (obj && obj->CanConvertToType(Class_ID(TRIOBJ_CLASS_ID, 0)))
	{
		//Let's convert the object in a TriObject
		TriObject *tri = static_cast<TriObject *>
			( obj->ConvertToType(t,Class_ID(TRIOBJ_CLASS_ID, 0)));

		// Note that the TriObject should only be deleted
		// if the pointer to it is not equal to the object
		// pointer that called ConvertToType()
		if (obj != tri)
			deleteIt = TRUE;
		return tri;
	}
	else
		return NULL;
}
```

This function is pretty complex. The main point is that we pass a Boolean parameter deleteIt as a reference. This parameter is used to know if the TriObject returned by the function has to be deleted or not after use.
In the node's pipeline, it is possible that 3D Studio Max creates a temporary Object just for our purpose so we need to delete it afterwards.
See section 14.2 for more details about the geometry's pipeline.

So to use the previous function, you proceed this way :

```
BOOL deleteit     = FALSE;
TriObject* Tri    = GetTriObjectFromNode(node, deleteit);
if (!Tri) return; //Not a Node that can be converted in a triangle mesh.

Mesh& mesh        = Tri->GetMesh();

DoSomethingOnTheMesh(mesh);

//If necessary, delete the TriObject and all references to this object.
If (deleteit)
      Tri->DeletMe() ;
```

Ok, now we are able to get the mesh, let's see how to extract information from it.

We will introduce the Point3 class which is just the class used in 3D Studio Max for 3D floats vectors. Its components are x, y and z to represent the 3 coordinates of the vector on the X, Y and Z axis.

Meshes contains :

- Vertices (3D points that compose the mesh)
- Faces and their information
- Vertices colors (color set on 3D points and interpolated using usually Gouraud shading)
- Textures vertices (UV coordinates)
- An array of custom floats. One for each vertex
- 3 bit arrays, one for each selection type : vertices, faces and edges
- The current level of sub selection inside the mesh (object, vertex, face, edge)
- Normals

To remember : All 3D coordinates of the vertices and normals are expressed in local coordinates, that is in the pivot coordinates and NOT in world coordinates. To get them in world coordinates you will have to multiply them by the pivot transform (see section 11.2)

Let's see how to get the vertices.

13.1. The vertices

They are the 3D points that compose the mesh. They are Point3 instances to describe their coordinates.
They are defined in the mesh by :

```
Point3 *verts;
int numVerts;
```

So , the number of vertices in this mesh is defined in the numVerts variable and the verts variable is an array of Point3.

It is recommended not to access vertices directly from their array but instead use the following functions :

```
BOOL        setNumVerts (int ct, BOOL keep=FALSE, BOOL synchSel=TRUE);
int         getNumVerts () const;
void        setVert     (int i, const Point3 &xyz);
void        setVert     (int i, float x, float y, float z);
Point3&     getVert     (int i);
Point3*     getVertPtr  (int i);
```

Let's see a complete example on how to get the vertices :

```
void MyClass::GetVerticesFromSelectedMesh()
{
	//Get the selected mesh
	const int NumNodesSel = ip->GetSelNodeCount();

	//Check if we have only one object selected
	if (NumNodesSel != 1)
	{
		MessageBox(NULL,"I need to have one and only one object
				selected !","Warning",MB_OK);
		return;
	}

	// Get the selected node
	INode *node	= ip->GetSelNode(0);
	if (!node)return;

	BOOL deleteit;
	TriObject* tri = GetTriObjectFromNode(node,deleteit);
	if (!tri)return;

	Mesh& mesh = tri->GetMesh();

	const int NumVerts = mesh.getNumVerts();
	for (int i=0;i<NumVerts;i++)
	{
		const Point3& _vertex = mesh.getVert(i);
	}

	if (deleteit)
		tri->DeleteMe();
}
```

Note :

You have to keep in mind that not all vertices are necessarily used in the mesh. It may have isolated vertices that are not used any more.
To know which vertices are used in the mesh, you need to get the faces.

13.2. The faces

To define the mesh's faces, the SDK provides a class named Face. It contains among others variables :

```
DWORD v[3];
DWORD smGroup;
DWORD flags;
```

- v[3] is an array of three 32 bits integers which represent the indices in the verts array of the vertices used by this face
- smGroup is the smooth group number
- flags is a set of flags, they are not custom flags. They represent the edge visibility in a face, material information (rarely used) and the face visibility

Note :

- **To remember : Under 3D Studio Max, all faces are always TRIANGLES only. S**o they contain always 3 vertices. While it is possible to select quads in the UI, they are always composed of 2 or more triangles and use the edge invisibility for selection purpose.

- **To remember** : be careful with the vertices / faces indices. In the SDK, faces are zero-based indexed, we are used to that as programmers but in the UI they are 1-based indexed. So if you want to see face #12 under the SDK it will be referred as face #13 in the UI…

In the mesh class, we have the member variable :

```
Face* faces ;
```

It contains the array of faces. Thenumber of faces can be retrieved with :

```
int Mesh::getNumFaces() ;
```

Let's see a complete example on how to get the faces. We consider, using the previous code, we already have a Mesh instance.

```
void GetFacesInfoFromMesh(Mesh& _mesh)
{
	//Get the number of faces
	const int NumFaces = _mesh.getNumFaces();
	for (int i=0;i<NumFaces;i++)
	{
		//get the face class from its index
		Face& _face	= _mesh.faces[i];

		//Get its smoothgroup
		DWORD smGroup	= _face.smGroup;

		//Get the flags
		DWORD flags	= _face.flags;

		//Get the vertices indices from this face
		//One single vertices is called a facevertex.
		const DWORD* VerticesIndices = _face.getAllVerts();
		if (!VerticesIndices) return; //Should never happen but anyway

		//Get all 3 face vertices
		for (int j=0;j<3;j++)
		{
			//Get its index into the verts array
			const int VertexIndex	= VerticesIndices[j];

			//Get its 3D coordinates
			const Point3& Vertex	= _mesh.getVert(VertexIndex);
		}
	}
}
```

To get the face normals, you just have to take the 3 face vertices V0,V1 and V2 which are Point3 instances and do :

Point3 FaceNormal = Normalize((V1-V0) ^ (V2-V1)) ;

where ^ is the cross product of two Point3 (CrossProd function in the SDK).
Face vertices should always be oriented in clockwise order. If they are in counter clockwise order the parity of the node's matrix should reflect this (see the fnction Parity() from Matrix3 class).

Let's continue by seeing the edges.

13.3. The edges

You have probably noticed that we haven't talked about an edge array yet. This is because there is no array about edges ☺

As all faces are triangles, it is possible to retrieve an edge from its index this way :

```
void GetEdgeInfoFromMesh(Mesh& _mesh)
{
	//Get the number of faces
	const int NumFaces = _mesh.getNumFaces();
	for (int i=0;i<NumFaces;i++)
	{
		//get the face class from its index
		Face& _face	= _mesh.faces[i];

		//Get the vertices indices from this face
		//One single vertices is called a facevertex.
		const DWORD* VerticesIndices = _face.getAllVerts();
		if (!VerticesIndices) return; //Should never happen but anyway

		//Get all 3 facevertices
		for (int j=0;j<3;j++)
		{
			//Get its index into the verts array
			const int VertexIndex	= VerticesIndices[j];

			//Calculate the edge index
			//i = current face index, j = current facevertex index
			const int EdgeIndex	= i*3 + j;
			const Point3 Edge	= _mesh.getVert(VertexIndex) -
						  _mesh.getVert((VertexIndex+1)%3);
		}
	}
}
```

13.4. Materials assigned to faces

While it is possible to get this information at different places in the mesh, the good information is returned by this function :

```
MtlID getFaceMtlIndex(int i);
```

Where MtlID is a typedef of an integer.

When a material is applied on a node (so on its mesh) there are 2 interesting cases for us, they are when the material is a :

- Standard material
- Multimaterial (called Multi/subobject in the UI).

1) In the case of a standard material, there is only one bitmap texture assigned on the whole mesh. It can be a combination of the diffuse, opacity, bump textures but anyway only one texture is assigned to the mesh. Consequently, all faces have this texture assigned and the material IDs whatever are their values in faces are not used.
2) In the case of a multimaterial, the Face material ID is the index of the submaterial of the multimaterial used on this face. **Don't forget submaterials in multimaterals are 0-based index**.

Note :
As no check is made on the value the user can enter in the face material ID manually, you always have to apply a modulo operation (remainder) on the number of submaterials of the multimaterials.

Example :
I have a multimaterial containing 3 submaterials. I met in the mesh a face with number 5 as face material ID, the submaterial applied on this face is not the 5th submaterial as they are only 3 of them? But it's the 2nd (with index 1). The operation is index number = 1 = 5 % 2. And we use 2 because submaterials are 0-based index so 3 submaterials goes from indices from 0 to 2.

Let's see a complete example on how to use this function :

```
void GetMtlIDFromNode(INode* _node)
{
	if (!_node)return;

	//Now get the mesh from the node
	BOOL  deleteit;
	TriObject*  Tri = GetTriObjectFromNode(_node,deleteit);
	if (!Tri)return;

	//Get mesh from triobject
	Mesh& mesh = Tri->GetMesh();

	//Get the material from node
	Mtl* nodemtl = _node->GetMtl();
	if (!nodemtl)return; //No material applied

	if (! nodemtl->IsMultiMtl() ) return; //It's not a multimaterial

	MultiMtl* multimtl = static_cast<MultiMtl*>(nodemtl);

	const int NumSubMaterials = multimtl->NumSubs();

	const int NumFaces = mesh.getNumFaces();
	//Scan all faces of the mesh
	for (int i=0;i<NumFaces;i++)
	{
		//Get the ID in this face (apply the modulo)
		const int ID = mesh.getFaceMtlIndex(i) % NumSubMaterials;

		//Get the material applied on this face
		Mtl* MtlAppliedOnThisFace = multimtl->GetSubMtl(ID);
		if (! MtlAppliedOnThisFace) continue;
	}

	if (deleteit)
		Tri->DeleteMe();
}
```

So, that's all about materials.
We are going to see the mapping channels. They are the different layers to set UV coordinates on faces. There are 99 layers of mapping coordinates, channel 0 is reserved for vertex colors, the mapping channel 1 is usually the UVs and the other are at your discretion.

13.5. Vertex colors

Vertex colors are set in the mapping channel number 0. They are Point3 instances with each component x, y and z of the 3D vector representing the red, green and blue color components. They all are in the range from 0 to 1.

We can get the number of vertex colors by using

```
int getNumVertCol();
```

The array of vertex colors is the member variable Mesh::vertCol.

Here is an example of accessing the vertex colors array :
VertColor is a typedef set to Point3.

```
VertColor* vertCol = mesh.vertCol;
if(vertCol)
{
	const int NumVertCol = mesh.getNumVertCol();
	for (int i=0;i<NumVertCol;i++)
	{
		const Point3& vcol = vertCol[i];
	}
}
```

Note : be careful, as well as the vertices and UVs, the vertex colors array can contain unused vertex colors. To know which are used and which are not, you have to refer to the face vertex colors. Because it is not really in 3D Studio Max a color per vertex but a colr per face vertex. Consequently a vertex can have several face vertex colors…

To know which face vertex colors are used, you need to use the Mesh::vcFace (vertex colors faces) member variable. It is an array of TVFace structures, **the size of the array is the same as the number of faces of the mesh**.

TVFace is a class which contains essentially three 32 bits integers :

```
DWORD t[3];
```

These 3 integers are indices in the vertex color array of the vertex colors used on this face.

Here is a complete example on how to get the vertex colors :

```
void GetVertexColorInfoFromMesh(Mesh& _mesh)
{
	const int NumVertCol	= _mesh.getNumVertCol();
	VertColor* vertCol		= _mesh.vertCol;

	if(! vertCol || ! NumVertCol) return; //No vertex color

	//Get the complete array of vertex colors
	//(warning, some may be not used in the mesh !)
	for (int i=0;i<NumVertCol;i++)
	{
		const Point3& vcol = vertCol[i];
	}

	//Get the vertex colors used in the mesh using Mesh::vcFace array
	const int NumFaces = _mesh.getNumFaces();

	if(! _mesh.vcFace) return;

	for (i=0;i<NumFaces;i++)
	{
		//Get the TVface class from its index in vcFace (vertex color)
		TVFace& _vcface		= _mesh.vcFace[i];

		//Get the vertex color indices from this face
		for (int j=0;j<3;j++)
		{
			//Get its index into the vertCol array
			const int VertexColorIndex	= _vcface.t[j];

			//Get its color
			const Point3& vertexcolorused =
				vertCol[VertexColorIndex];

		}
	}
}
```

Note : From 3D Studio Max 6 and later, it is also possible to paint vertex colors thanks to the Vertex Paint modifier in any mapping channel, so you can retrieve vertex color in any mapping channel.

The same way we access vertex colors, is it possible to access UV coordinates. Let's go deeply in detail with that.

13.6. The UV coordinates

They are called the textures vertices or TVerts in the SDK. They are the coordinates used to set bitmap textures on faces of the mesh.

As we have seen before, it exists 99 mapping channel to set these UV coordinates. You can see that as to have 99 possibilities to have different UV coordinates for multitexturing for example.

- Mapping channel 0 is reserved for vertex colors
- Mapping channel 1 is reserved for default UV set on the mesh
- And you can use the 97 remaining mapping channels as you want to set UV coordinates or whatever you need…

The textures vertices are Point3 instances. So they have 3 possible coordinates. While we usually use only 2 of them, it is possible to have 3D textures that will use 3 coordinates instead of 2.
The UVVert class is used to represent the class of textures vertices. It is a typedef set to Point3.

13.6.1. UV coordinates in mapping channel 1

The UV coordinates are assigned on faces, by **default in mapping channel 1**.
It exists an array of texture vertices, in which some of the texture vertices may not be used in the mesh.

Let's see how to get this array in the mesh :

```
Point3 UV ;
const int NumTV = mesh.getNumTVerts();    //implicitly on mapping channel 1

for (int i=0 ;i< NumTV;i++)
      UV = mesh.tVerts[ i ]; //Get all components of UVs array
```

Now, we are going to see how to get the UVs that are used in the mesh. To do so, we need to use the information set on faces in the tvFace array. tvFace = texture vertices faces.

The TVFace class contains only an array of 3 integers in its t[3] member variable. Only 3 integers as faces are always triangles.
Let's see a complete example :

```
//Now get only the UVs used in the mesh
const int NumTV = mesh.getNumTVerts();    //implicitly on mapping channel 1
if (! NumTV)return;

//Get num faces in the mesh
const int NumFaces = mesh.getNumFaces();
for (i=0;i<NumFaces;i++)
{
      //Get the TVface class from its index from tvFace (texture vertices)

      TVFace& _tvface         = mesh.tvFace[i];

      //Get the texture vertices (=UV) indices from this face
      for (int j=0;j<3;j++)
      {
            //Get its index into the verts array
            const int UVIndex = _tvface.t[j];

            //Get its UV
            const Point3& UV = mesh.tVerts[UVIndex];
      }
}
```

Well, this seems to be complex enough, unfortunately, to be very complete an information is still missing… Remember, UV coordinates in 3D Studio Max are Point3, so 3 floats coordinates. So you are thinking that taking the 2 first coordinates of Point3 instances, that is x and y would be enough to represent the U and V coordinates respectively. Well you are right in most cases… But not exactly right.

Most of the time, when using 2D texture bitmaps, the x and y represent the U and V coordinates. But it is possible that in some cases it is the y and z or x and z that represent the U and V coordinates…

To solve that problem, you need to look at the material applied on the mesh !!! On more accurately on the BitmapTex instance for a bitmap texture assigned to the face you are processing.

It is possible to access the Point3 class coordinates using an array, so 0 is the x coordinate, 1 the y coordinate and 2 is the z. So the following works fine :

```
Point3 P ;
const float x = P[0]; //Equivalent to P.x
const float y = P[1]; //Equivalent to P.y
const float z = P[2]; //Equivalent to P.z
```

Consequently, in the material of the mesh, we only need to get the 2 indices that we should use in the Point3 to get its U and V coordinates.

To do so, we have the following function :
From a material, it get the bitmap texture (BitmapTex class) which is in the diffuse texture slot of the material.
Then this BitmapTex contains an instance of the class UVGen which is the part of the material which deals with UV coordinates. It contains information such as :

- A mirror is applied on UV coordinates
- The material has some texture tiling (U or V coordinates are greater that 1 in absolute value)
- The axis used to specify the UV coordinates etc…
- …

You have guessed that the UVGen information can be translated into a 2D matrix that should be applied to the UV coordinates.

Well, the information interesting for us is the axis used which says which indices in the Point3 coordinates should be used as U and V.
Let's now see this custom function (not part of the SDK) named GetIndicesFromTexMap :

```
void GetIndicesFromTexMap(int& i1, int& i2, Mtl* _mtl)
{
	if (!_mtl)return;

	//Get the bitmapTexture which is in the diffuse
	//(it's not a bitmap, it's a class with the bitmap)
	BitmapTex* TMap = static_cast<BitmapTex*>
			(_mtl->GetSubTexmap(ID_DI));
	if (!TMap)return;

	//Check if we have a Bitmap Texture material (BitmapTex)
	if (TMap->ClassID() != Class_ID(BMTEX_CLASS_ID,0))
		return;	//Not interesting

	//Get information about UVs from Material
	UVGen * pUVGen = TMap->GetTheUVGen() ;
	if(!pUVGen)return;

	//Get the Axis used to get the suitable indices
	const int iAxis = pUVGen->GetAxis();
	switch( iAxis )
	{
		case AXIS_UV :
		{
			i1 = 0;
			i2 = 1;
		}
		break ;
		case AXIS_VW :
		{
			i1 = 1;
			i2 = 2;
		}
		break ;

		case AXIS_WU :
		{
			i1 = 2;
			i2 = 0;
		}
		break ;
	}
}
```

Then with this new information we have all we need to get the real UV coordinates this way :

```
int i1 = 0;
int i2 = 1;

//Get the material applied on the node
Mtl* mtl = _node->GetMtl();
BOOL MultimaterialApplied = FALSE;
if (mtl)
{
	if (mtl->IsMultiMtl())
		MultimaterialApplied = TRUE;
	else
	{
		//Get the indices from this material
		GetIndicesFromTexMap(i1, i2, mtl);
	}
}
else
{
	//There is no material on this node
	return; //Not interesting
}

Point2 RealUV; //We are going to get the Real UV in a Point2 (2D vector)

//If we have a multimaterial applied on this node, we're going to scan the
//ID of the faces too
if (MultimaterialApplied)
{
	//Get the Multi Material
	MultiMtl* multimtl = static_cast<MultiMtl*>(mtl);

	//Get the number of sub materials
	const int NumSubMaterials = multimtl->NumSubs();

	//Scan all faces of the mesh
	for (int i=0;i<NumFaces;i++)
	{
		//Get the ID in this face
		const int ID = mesh.getFaceMtlIndex(i) % NumSubMaterials;

		Mtl* MtlAppliedOnThisFace = multimtl->GetSubMtl(ID);
		if (! MtlAppliedOnThisFace) continue;

		GetIndicesFromTexMap(i1, i2, MtlAppliedOnThisFace);

		//Get the 3 texture Vertices ( = UV )of this face.
		//the TV info is memorized on each face within the TVFace
		//class inside mesh
		const TVFace& textureVerticeFace = mesh.tvFace[i];

		//Get the 3 texture vertices
		for (int j=0;j<3;j++)
		{
			UV = mesh.tVerts[ textureVerticeFace.t[j] ];

			RealUV.x = UV[i1];
			RealUV.y = UV[i2];
		}
	}
} //To be continued on next page…
```

```
else //It is not a multimaterial applied on the mesh
{
	for (int i=0;i<NumFaces;i++)
	{
		const TVFace& textureVerticeFace = mesh.tvFace[i];

		//Get the 3 texture vertices
		for (int j=0;j<3;j++)
		{
			//Get the TV
			UV = mesh.tVerts[ textureVerticeFace.t[j] ];

			RealUV.x = UV[i1];
			RealUV.y = UV[i2];
		}
	}
}
```

Now let's see the UV coordinates in the other mapping channels from 2 to 99.

13.6.2. UV coordinates in others mapping channels

To access the others mapping channels, we can use the following functions members of the Mesh class :

```
int getNumMaps() const
BOOL mapSupport(int mp) const;
int getNumMapVerts(int mp) const;
UVVert *mapVerts(int mp) const;
TVFace *mapFaces(int mp) const;
```

- getNumMaps is used to know how many mapping channels are used in the mesh
- mapSupport is used to know if the mapping channel whose index is passed in parameter is used or not
- getNumMapVerts returns the number of texture vertices for a specific mapping channel
- mapVerts is the array of texture vertices for a specific mapping channel (note : UVVert is a typedef defined to Point3)
- mapFaces is the array of TVFace for a specific mapping channel

Note :
If your mesh contains 55 mapping channels for example. It is possible that only a few of them are used. Because you can set mapping UV coordinates in mapping channel numbers 1 and 54. so the number of mapping channels would be 55 but only 2 of them would be used. So don't forget to check if a mapping channel is used with the function Mesh::mapSupport.

Let's see an example on how to get the UV coordinates in another mapping channel that 1 :

```
//Get number of mapping channels used
const int NumFaces             = mesh.getNumFaces();
const int NumMapChannels = mesh.getNumMaps();

//Remark : if the user has set some UVs in the map channel, say 54.
//The NumMapChannels would contain 55.
//But only a few channels would be used such as 0, 1 and 54...
for (int i=0;i<NumMapChannels;i++)
{
      //Is this mapping channel used ?
      if (! mesh.mapSupport(i)) continue;

      //Get TVFace and TVerts arrays
      TVFace* _TvFaceArray          = mesh.mapFaces(i);
      UVVert* _TabTVertArray        = mesh.mapVerts(i);
      //UVVert = Point3

      //Does both exist ?
      if (_TvFaceArray && _TabTVertArray)
      {
            //Just a test...
            const int NumMapVerts    = mesh.getNumMapVerts(i);

            for (int faceidx=0;faceidx<NumFaces;faceidx++)
            {
                  const TVFace& tvface = _TvFaceArray[faceidx];
                  for (int j=0;j<3;j++)
                  {
                        const int CurrentTVIndex = tvface.t[j];
                        if (CurrentTVIndex >= NumMapVerts) continue;
                        //The last line should never happen
                        const UVVert& UV  = _TabTVertArray[CurrentTVIndex];
                  }
            }
      }
}
```

13.7. Faces and edges adjacency

It is possible to build an adjacency table between edges and between faces.
To do so, you have to create an instance of the AdjEdgeList and AdjFaceList faces.

To create an AdjEdgeList instance only the mesh is necessary.
But to create an AdjFaceList instance you will need the mesh as well as the AdjEdgeList instance from this mesh…

Note :
To create the face adjacency table, you will need a mesh that does not contain degenerated faces. By degenerated we mean that each edge should have exactly one or two faces, not more. If this is not the case, you could use the Mesh::RemoveDegeneratedFaces() function to attempt to correct the mesh to work with this class)

This results in :

```
void GetAdjacencyFromMesh(Mesh& _mesh)
{
	AdjEdgeList*  mAdjEdges   = new AdjEdgeList(_mesh);
	if (! mAdjEdges )return;

	AdjFaceList* mAdjFaces    = new AdjFaceList(_mesh, *mAdjEdges);
	if (! mAdjFaces )return;

	const int NumFaces = _mesh.getNumFaces();
	for (int faceidx = 0;faceidx<NumFaces;faceidx++)
	{
		//Get all adjacent faces
		for (int i = 0; i < 3; i++)
		{
			int adjfacenum = mAdjFaces->list[faceidx].f[i];
		}
	}
}
```

14. The most common plug-ins categories

In the plug-in categories list presented in the beginning of this document, only a few of them are of interest to us. They are importers/exporters, utilities and modifiers

14.1. Importers-exporters plug-ins

They are the most common plug-in category. All companies that wants to get 3D content in / out of 3D Studio Max have to use an importer / exporter plug-in.

It was not possible to read directly the .max file without launching 3D Studio Max application, the only way was to use an exporter plug-in.
It is now possible to read a .max directly without the 3D Studio Max application launched since 3D Studio Max 6 and later. It is called MaxfileReader and you can download this project o the Sparks web site.

The importer is a plug-in that lets you create a 3D scene in 3D Studio Max from a file. It is more rare than exporters.

14.1.1. What can we get from a 3D scene ?

Well, everything. ☺ For example, we usually get
- meshes
- materials and bitmap textures
- splines / patches
- cameras
- lights
- helpers objects (dummies for example)
- morphing information
- skinning information
- animation on anything that is animated (meshes, materials, bones)

The character animation and rigging is often made with the Character Studio plug-in integrated within 3D Studio Max. The default skeleton is bcalled a Biped and the modifier to skin a mesh with these bones is the Physique modifier (while it is possible to use the Skin modifier too)

We will se later some code examples using these objects.

14.1.2. The different types of file formats

It is possible to use your own custom file format to export data. The most common possible types of formats are ASCII and binary file formats.

The ASCII format is a text file, it is readable by a user/developer thanks to a text editor.
The pros of this format are :

- You can debug it easily by looking at the text file.

The cons are :

- Reading/writing are slow, but if time is not an issue that's fine.
- File size can become huge, several mega-bytes for a 60, 000 polygons scene.
- Anybody could read your format (this is an advantage as well as a drawback, it's up to you)

The pros of the binary file format are :

- It has a smaller file size than ASCII
- It is faster to read/write than ASCII, this can be a big advantage when time is important.

The cons of this format are :

- It is unreadable by a human in a text editor, even by the people that has created this file format. This is also an advantage if you want to protect your data.
- If you modify the format, you can have problems of versioning to read/write old versions. To overcome partly this problem, use a version number in the beginning of your format..

It is also possible to create a mix between file formats by using ASCII and binary.
For example, mixing XML file format and binary.

The XML file format is considered to be a great format for importers/exporters. It is widely used over the internet and most software use this format.
It is very convenient because it is a text file format with nodes properties that let you show/hide some part of the file by collapsing/expanding nodes hierarchy. And it is natively recognized in latest internet browsers.

However, you should be careful with existing XML parsers you can found on the internet. Most of them has been thought and developed for internet content, so small size files…
While these parsers could seem attractive, you will probably have to code your own parser…

```
<?xml version="1.0" standalone="yes" ?>
<!-- Please don't edit this file manually. Thank you.  -->
- <DL3D_REPLAY version="1.0">
  - <MESH>
      <VERTEX>0 0 0</VERTEX>
      <VERTEX>0 123.4 -45.5</VERTEX>
      <VERTEX>0 123 -90</VERTEX>
    </MESH>
  </DL3D_REPLAY>
```

Figure 6 : XML file format example under Micrososft Internet Explorer

14.1.3. Tips and tricks to develop and importer/exporter

The perfect importer/exporter is anyway a dream. So we give the reader some advices from the experience we had with developing importers / exporters for several video games projects at the same time.

- Create a report file. Reporting any problems with a level of gravity (warning, serious problem, crash etc..) can be interesting or really necessary. It gives the user a feedback about whether the operation worked fine or not. Then you can show this file optionally at the end of the import/export phase. You should avoid a modal dialog box or any information that stops the process in case the user is exporting several files using a batch processing file.

- Importers / exporters should always have the possibility to be launched using a batch file, a Maxscript file for example. So don't forget to have a Maxscript exposure to your plug-in.

- Thinking about showing the parameters of your importer / exporter is a fine thing but you should also think about the possibility to be able to set this parameter using the batch file or Maxscript file and removing any dialog box. It should exist a quiet mode where no dialog box is shown whatever is the message…

- This tip is mostly interesting if you have several projects that will use your importer/exporter. The idea is to let programmers customize your plug-in. To be able to customize your importer / exporter can be a critical part if you have not thought about that during the design. It can be interesting to separate your plug-in into several DLLs one to export a specific kind of object, say one for the mesh exporter, one for the camera exporter etc… So if a programmer wants to customize your plug-in he will only have to take care of a small part of it and not recompile the whole project.
  So if you have a plug-in factory to load and register plug-ins available to export your data you can send the data to the plug-in designed to export this kind of data. Say you have a mesh, send it to the DLL that exports meshes. This DLLs could be yours or another custom DLL. This solution will also avoid any #ifdef in your code for some specific needs of a project.

Let's see some examples about exporters :

An exporter class is always a subclass of SceneExport SDK class. The main function of this class is :

```
int SceneExport ::DoExport(const TCHAR *name, ExpInterface *ei, Interface
*i, BOOL suppressPrompts=FALSE, DWORD options=0)
```

From this function happens the export phases.
For example you can get all nodes from the scene and export them once the system has called the SceneExport::DoExport function. We can do that this way :

```
//Function to export all nodes from a scene.
void MyExport::GetAllNodes(INode* _node)
{
      If ( ! _node)DEBUGSTOPRETURN //Macro to stop in debug build and
                                   //return

      bool b = ExportNodeInformation(_node);
      if (!b)DEBUGSTOPWRITEWARNING(MyExport ::ExportNodeInformation didn't
      worked") //Stop in debug build and write a warning or error in a
               //report file.

      //Recursively get all nodes
      const int numChildren = _node->NumberOfChildren();
      for ( int i = 0; i < numChildren; i++ )
      {
            GetAllNodes (_node->GetChildNode( i ) );
      }
}
```

This function is called like this :

```
//This function is called that way :
Interface* _ip = GetCOREInterface();
INode* rootnode = _ip->GetRootNode();
GetAllNodes(rootnode);
```

Note : Remember the root node is a virtual node which is the highest node in the hierarchy. All nodes are children of the root node.

Then from a node, we can get its material and textures in the following manner :.

```
Mtl* _mtl = _node->GetMtl() ;
If (mtl)//Some nodes could have no material…
{
      bool b = ExportMaterial(mtl);
      if ( !b) DEBUGSTOPWRITEWARNING("MyExport::ExportNodeInformation
      didn't worked")
      //Stop in debug build and write a warning or error in a
      //report file.
}
```

We have already seen in the section 12 how to get the information from materials.

To remember : if you are beginning with 3D Studio Max SDK and you want to create an exporter, you can use the IGame interface that has been designed to simplify the export phase to beginners. See the section 25

Now let's go in details with the modifiers plug-ins.

14.2. The modifiers plug-ins

They are used to modify objects. They are pretty simple to setup except if you wish that the user is able to select some parts of an object like its vertices or whatever. In this case, you usually need to copy/paste a thousand lines of code from an existing modifer so this issue won't be discussed in this document.

The key function in modifier is Modifier ::ModifyObject. This function is called by the system when the modifier needs to apply its modification.

Note : Don't forget to be thread safe when developing a modifier. See section 6.3.

Before seing the ModifyObject function, we need to introduce how the 3D Studio Max geometry pipeline works.

14.2.1. The geometry pipeline

The pipeline used to modify objects is called the geometry pipeline, or modifier pipeline or pipeline in short. This pipeline acts on nodes to modify them in any way. It can act on any kind of node.

We are going to understand that by seeing a concrete example. It's a sphere procedural object where we have applied two modifiers : an edit mesh and a UVW mapping then another edit mesh and finally another UVW mapping.

Note :
The Edit mesh modifier is designed to edit parts of the mesh, such as move vertices, faces, edges, create vertices, faces etc…
The UVW mapping modifier is designed to set UV coordinates on a mesh or a part of the mesh.

The next figure represent the sphere and its modifiers applied. It is called the modifiers stack.

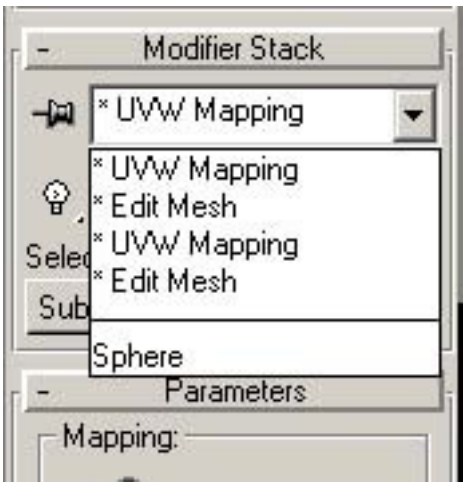

Figure 7 : The modifiers stack

This is called a modifiers stack because it is a stack. The chronological order in which modifiers are applied is from bottom to top.

In our example, the user has created a procedural sphere (at the bottom of the stack) then he has applied an edit mesh, he has probably modified something on the mesh e.g. the faces selection then he has applied a UVW Mapping modifier to modify the UV coordinates of this face selection and so on…

When the system asks for an update of the pipeline, it works this way :

1) Our sphere object is a BaseObject class instance with its procedural parameter such as ray of the sphere. And the edit mesh modifier on top of it in the pipeline applies its modification on our sphere and we get another object, let's call it Sphere 2 which is a modified version of our original sphere (e.g : the face selection has changed).

2) Then this new object Sphere 2 is used by modifier UVW Mapping to apply its modification on it. This result in a 3r^{d} object called Sphere 3…

3) Then the pipeline goes on with remaining modifiers on the stack linearly. …

To remember : The state where the object is after all modifiers from the stack have been applied on it is called the world space state. This is an important information because we use that to get the Object from a INode using the following function :

```
const ObjectState& Interface::EvalWorldState(TimeValue time,BOOL
evalHidden=TRUE);
```

It is used in our GetTriObjectFromNode function which enables you to get a Mesh from an INode, see section 13.

Note :
About utility plug-ins, if you have a node with some modifiers on it and you try to modify its mesh directly. It will work but only temporarily. But as soon as the system will update the geometry pipeline for example when redrawing the viewport is necessary, all your modifications will be erased because your modifications are not part of the pipeline and the geometry will be re-built from the BaseObject and adding each modifier's modifications.

Well, this was only a simplification of the geometry pipeline to understand it on the whole. Here are the details :

The first time a modifier is applied on a mesh, an instance of IDerivedObject is created. A derived object is a container of modifiers followed by a reference to another object.
See the following example :

This is a Cylinder which is a BaseObject and we have applied it a Bend modifier.

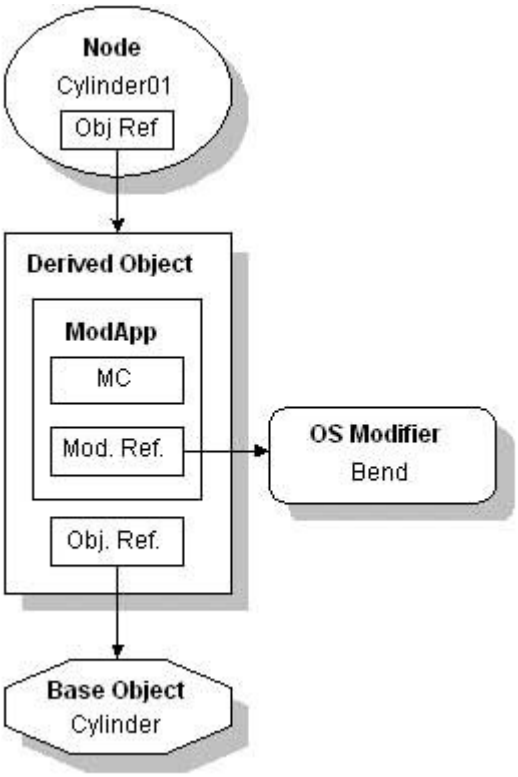

Figure 8 : A derived object inserted in the pipeline

A derived object is created, it contains a ModApp and an object reference. The ModApp is made of a ModContext and a modifier reference, on the Bend modifier in our example.

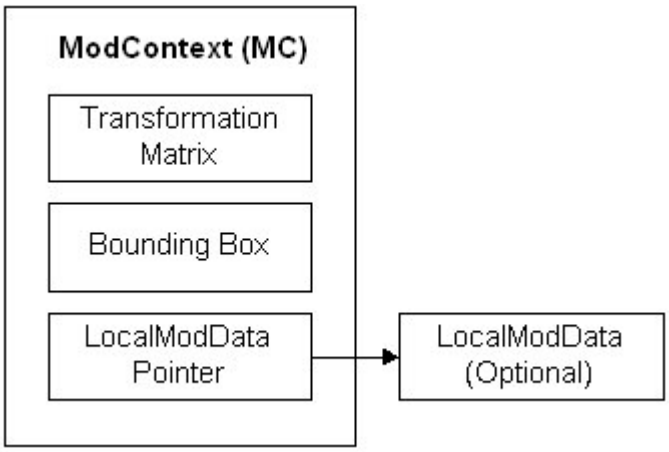

Figure 9 : The components of a ModContext

A Modcontext has 3 components.

1) The Transformation Matrix. This matrix represents the space the modifier was applied in. The modifier plug-in uses this matrix when it deforms an object. The plug-in modifier first

transforms the points with this matrix. Next it applies its own deformation. Then it transforms the points back through the inverse of this transformation matrix.

2) The Bounding Box of the Deformation. This represents the scale of the modifier. For a single object it is the bounding box of the object. If the modifier is being applied to a sub-object selection it represents the bounding box of the sub-object selection. If the modifier is being applied to a selection set of objects (and the user interface 'Use Pivot Points' checkbox is off), then this is the bounding box of the entire selection set. For a selection set of objects the bounding box is constant. In the case of a single object, the bounding box is not constant. For example, if the user applies a 90 degree bend to a cylinder, then changes the height of the cylinder, one would want the cylinder to still be bent 90 degrees. If the bounding box did not adapt, the cylinder would appear to move through the bend causing the bend angle to be incorrect.

3) A pointer to an instance of a class derived from LocalModData. This is the part of the ModContext that the plug-in developer controls. It is the place where a modifier may store application-specific data. The LocalModData class has two methods the derived class must implement. One is a Clone procedure so the system can copy a ModContext. The second is a virtual destructor so the derived class can be properly deleted.

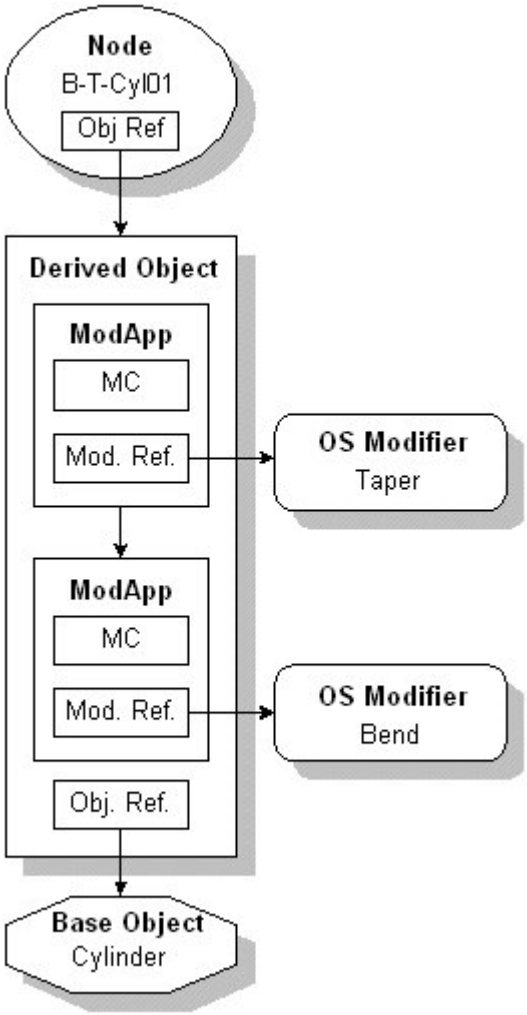

Figure 10 : A derived object with 2 modifiers

In this case a new ModApp is inserted into the existing Derived Object. The modifier reference of the new ModApp points to the Taper modifier.

The case where another derived object is created is when you create a reference on your node and add a modifier to the new created node. It results in the following figure :

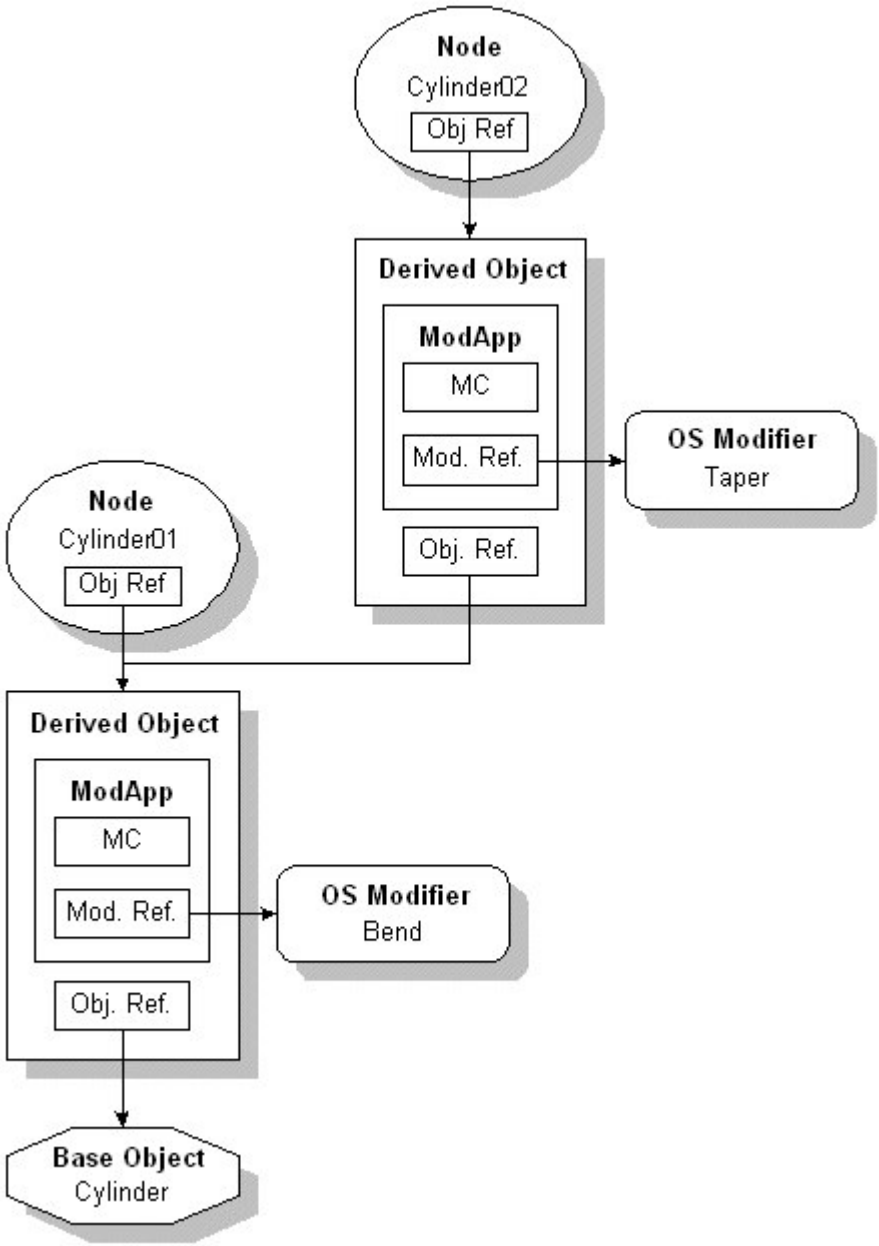

Figure 11 : a new derived object is created

The node named Cylinder02 on the top of the figure above is a reference on the node named Cylinder01. And we have added a Taper modifier on the Cylinder02 node.

Modifiers can also be instanced, it results in the following diagram :

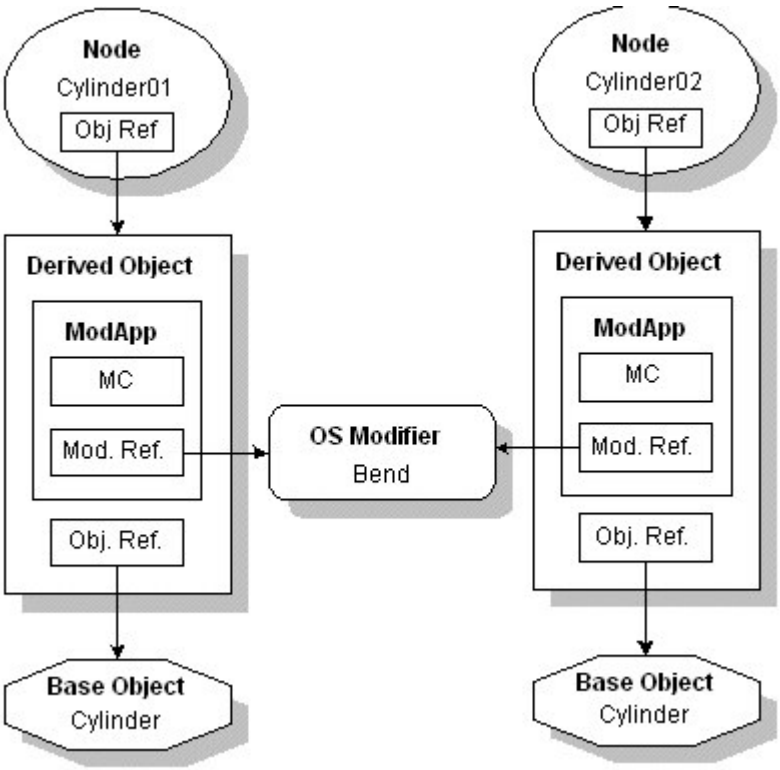

Figure 12 : Instanced modifier for 2 derived objects

Both derived objects refer to the same modifier.

The class IDerived object is used for example to check how many modifiers are there on a specific node. See the code examples with this support document.

14.2.2. Optimization of the geometry pipeline

To save speed and memory, the pipeline uses caches. It is called a world space cache. This cache contains a validity interval which tells if it needs to be updated or not.

When an object must be drawn, 3D Studio Max checks the node's world space cache and if it is valid, he takes the cached object, if not, it re-evaluates the node through the geometry pipeline.

For the same reason of optimization, the pipeline is separated into different geometry channels. They are called modifiers channels in the SDK and the enum is ChannelMask. Modifiers have the option of only modifying / using specific channels.

This way, only parts of the nodes that are needed by the pipeline are updated, the others remain unchanged and cached.

This channels are beyond :

- geometry (vertices),
- topology (face or polygon structures),
- texture vertices (UV coordinates),
- sub-object selection,
- level of selection
- display control
- Etc…

Channels used and changed are defined using Modifier::ChannelsUsed and Modidifer::ChannelsChanged.

So , when developing a custom modifier, if your goal is to modify the faces selection of a mesh, you will have to define the following function this way in your modifier subclass :

```
virtual ChannelMask ChannelsUsed    () {return SELECT_CHANNEL;};
virtual ChannelMask ChannelsChanged () {return SELECT_CHANNEL;};
```

If we wanted to use the faces selection as well as information about faces themselves to modify the faces selection, we would have defined them this way :

```
virtual ChannelMask ChannelsUsed    () {return  SELECT_CHANNEL |
                                               TOPO_CHANNEL;};
virtual ChannelMask ChannelsChanged () {return SELECT_CHANNEL;};
```

If we wanted to modify some vertices from a faces selection, our modidifer subclass would have implemented the 2 functions this way :

```
virtual ChannelMask ChannelsUsed    () {return SELECT_CHANNEL;};
virtual ChannelMask ChannelsChanged () {return GEOM_CHANNEL;};
```

In the source code samples provided with this support document, you could find a modifier example that moves the vertices of a mesh.

To validate these modifications ,the modifier must implement its validity interval for its caching system :

```
Interval Modifier::LocalValidity(TimeValue t);
```

If the current time is within the validity interval, the cached object is returned instead of re-evaluating the node's pipeline and returning the result.
And finally, once you are done with modifying the object, you must call the Object::UpdateValidity function.

```
void Object::UpdateValidity(int nchan, Interval v);
```

The first parameter is a modifier channel (called ChannelMask in the SDK) and the second parameter is the validity interval of these modifications.

So for example in a subclass of Modifier you could have the following code in ModifyObject function, heart of the modifier :

```
void DL3DModifierTutorial::ModifyObject(TimeValue t, ModContext &mc,
ObjectState* os, INode *node)
{
	//Is it a Triangle Mesh ?
	if(   (os->obj)  &&
		(os->obj->IsSubClassOf(triObjectClassID) == FALSE)
	  )return;

	//Yes, so cast its Object
	TriObject *triOb = reinterpret_cast<TriObject *>(os->obj);

	//Get the mesh
	Mesh& mesh                    = triOb->GetMesh();

	//We are going to modify the vertices
	//Get the mesh bounding box
	mesh.buildBoundingBox();
	Box3 bbox                     = mesh.getBoundingBox();

	//Compute half length of bounding box
	const float Length            = FLength(bbox.Width()) * 0.5f;

	//Add/remove this length to the x coordinates of vertices
	const int NumVertices   = mesh.getNumVerts();
	for (int i=0;i<NumVertices;i++)
	{
		Point3& p             = mesh.verts[i];

		//If our index is even, add it, if odd remove it
		if (i % 2 == 0)
			p.x                   += Length;
		else
			p.x                   -= Length;
	}

	//Update validity interval
	triOb->UpdateValidity(GEOM_CHAN_NUM, FOREVER);
}
```

This example changes the x coordinate of the mesh vertices thanks to its bounding box information.

14.2.3. Running through / Collapsing the modifiers stack

You should know that it is possible to collapse the stack. Collapsing the stack means flowing the object through the pipeline from the BaseObject to the world space cache object and removing all modifiers applied and considering the world space cache object as the current BaseObject.
To summarize, this means to apply all modifications on the object and taking the result of this as the current object and removing the history (the modifiers).
A triangle mesh object is concretely translated into an editable mesh object in that case.

Collapsing the stack is possible from the user point of view as well as programmatically. See the source code samples given with this support document.

Note :
Modifiers can be activated / deactivated by the user. When they are deactivated, they don't modify the object any more since they are become active again.
It is also possible to copy / paste modifiers from one object to another.

So it is possible to run through the modifiers applied on a node this way :

```
void ScanModifiersFromNode(INode* node)
{
	if (! node)return;

	Object *pObj 			= node->GetObjectRef();
	IDerivedObject *pDerObj 	= NULL;
	Modifier *sm 			= NULL;
	ModContext* mc 			= NULL;

	// Check for presence of derived object
	// if none exists, create a new one
	if( pObj->SuperClassID() == GEN_DERIVOB_CLASS_ID)
	{
		pDerObj = static_cast<IDerivedObject *> (pObj);
	}
	else
	{
		pDerObj = CreateDerivedObject();
		pDerObj->TransferReferences(pObj);
		pDerObj->ReferenceObject(pObj);
	}

	//Get num modifiers on object
	const int NumMods = pDerObj->NumModifiers();
	for (int i=0;i<NumMods;i++)
	{
		Modifier *tMod 		= pDerObj->GetModifier(i);
		if (! tMod) continue; //Should never happen

		const char* ModName 	= tMod->GetName();
		const Class_ID& cid 	= tMod->ClassID();
		ModContext* mc 		= pDerObj->GetModContext(i);
	}
}
```

In this example, we take an instance of the ModContext class, so we can access its LocalModData pointer where local data of the modifier to apply itself are stored.

14.2.4. «Surviving » a modifier stack collapse

Since 3D Studio Max 4, the SDK offers you the possibility to survive a stack collapse.

This is done through a mechanism which enables you to be notified before and after a stack colapse happens.
These functions are part of the BaseObject class and Modifier is a subclass of BaseObject :

```
void BaseObject::NotifyPreCollapse(INode *node, IDerivedObject *derObj,
                                   int index);
void BaseObject::NotifyPostCollapse(INode *node, Object *obj,
                                   IDerivedObject *derObj, int index);
```

Using these functions, the modifier clones itself before the objet is collapsed then it copies its clone after the stack has been collapsed.
A few modifiers (none ?) use this mechanism in the current standard modifiers of 3D Studio Max.

Let's see how this works :

```
// Between NotifyPreCollase and NotifyPostCollapse, Modify is
// called by the system.  Lets not be modified during the collapse
void PerFaceData::NotifyPreCollapse(INode *node, IDerivedObject *derObj,
int index)
{
	//Copy our LocalModData
	Collapsed = TRUE;
	ModContext* pModCtx = derObj->GetModContext( index );
	ClonedLocalModData = (CustomData*)pModCtx->localData->Clone();
	TimeValue t = GetCOREInterface()->GetTime();
	NotifyDependents(Interval(t,t),PART_ALL,REFMSG_CHANGE);
}

// We want to survive a collapsed stack so we reapply ourselfs here
void PerFaceData::NotifyPostCollapse(INode *node,Object *obj,
IDerivedObject *derObj, int index)
{
	Object *bo = node->GetObjectRef();
	IDerivedObject *derob = NULL;
	if(bo->SuperClassID() != GEN_DERIVOB_CLASS_ID)
	{
		derob = CreateDerivedObject(obj);
		node->SetObjectRef(derob);
	}
	else
		derob = (IDerivedObject*) bo;

	// Add ourselves to the top of the stack
	derob->AddModifier(this,NULL,derob->NumModifiers());

	// Reinsert our local mod data
	ModContext* mc = derob->GetModContext(derob->NumModifiers()-1);
	mc->localData = ClonedLocalModData;
	ClonedLocalModData = NULL;
	Collapsed = FALSE;
}
```

The example above comes from the modifier PerFacedata whose source code is in the SDK samples. This modifier is used to set flags on faces of a mesh.

Here is an example of stack collapse from a valid INode pointer called « node » :

```
// Eval the node's object (exclude WSMs)
Object *oldObj = node->GetObjectRef();

// Check for NULL
if (!oldObj) continue;

// Skip bones
if (oldObj->ClassID()==Class_ID(BONE_CLASS_ID,0)) continue;

// RB 6/14/99: Skip system nodes too
Control *tmCont = node->GetTMController();
if (tmCont && GetMasterController(tmCont)) continue;


NotifyCollapseEnumProc PreNCEP(true,node);
EnumGeomPipeline(&PreNCEP,node);

ObjectState os    = oldObj->Eval(g_ip->GetTime());
Object *obj       = (Object*)os.obj->CollapseObject();

if(obj == os.obj)
      obj = (Object*)obj->Clone();

if (os.obj->CanConvertToType(triObjectClassID))
{
      // Convert it to a TriObject and make that the new object
      TriObject* tobj = (TriObject*)obj->ConvertToType
(g_ip->GetTime(),triObjectClassID);
      oldObj->SetAFlag(A_LOCK_TARGET);
      node->SetObjectRef(tobj);
      oldObj->ClearAFlag(A_LOCK_TARGET);

      // NS: 4/6/00 Notify all mods and objs in the pipleine,
      //that they have been collapsed
      NotifyCollapseEnumProc PostNCEP(false,node,tobj);
      EnumGeomPipeline(&PostNCEP,oldObj);

      if (obj!=tobj) obj->AutoDelete();
}

GetSystemSetting(SYSSET_CLEAR_UNDO);//flush the undo buffer
```

14.2.5. The Extension Channel Objects

They are objects that will expand the geometry pipeline by allowing one to add a custom object to the pipeline object that can flow down the pipeline. This object will get notified whenever something in the pipeline changes.
They are inserted in the Modifier::ModifyObject function.

For example, if you want to indicate when a certain object becomes invalid for export to their game engine, invalid skin-vertex assignments, bound patches etc. By inserting an Extension Channel Object (XTCObject, for short) into the pipeline you can accomplish this, by constantly checking the structure of the object and displaying wrong faces/vertices etc. in the viewport.

So in the Modifier::ModifyObject, you create an instance of your subclass of XTCObject and you ad dit thanks to the Object:: AddXTCObject function :

```
void AddXTCObject(XTCObject *pObj, int priority = 0, int branchID = -1);
```

Which leads to :

```
Object* obj; //Valid object pointer
MyXTCObject* xtcobj = new MyXTCObject;
obj->AddXTCObject(xtcobj);
objSetChannelValidity(EXTENSION_CHAN_NUM, GetValidity(t));
```

14.3. The utility plug-ins

These plug-in category is widely used. It is used for example to extract information from a complete scene such as the number of faces, materials etc..
Utility plug-ins are usually not linked to one specific object as modifiers are, they must be thought as a toolbox plug-in. They are the easiest and fastest plug-ins to develop and they don't need a complete knowledge of the geometry pipeline to be developed (while you should feel at ease with it anyway now ☺).

Examples of utility plug-ins :

- A real time faces counter in a viewport
- A racing track editor
- A scene checker for exporter constraints
- A modifier driver to save clicks and time and improve the workflow
- A plugin to retrieve the faces and vertices in a mesh

To remember :
Utility plug-ins only let you modify the geometry of an object if the modifier stack is collapsed as we have seen before in section 14.2.1.
This can be a problem if you want to act ona node where modifiers such as Skin or physique are applied. If you collapse the stack all information about these modifiers will be lost as they don't survive a stack collapse and all their data are stored locally in the modifier.

If you hesitate between developing a utility or a modifier, you can have a look at an article on the Gamasutra web site : « Choosing Between Utility and Modifier Plug-Ins for 3D Studio Max ». See section 28.

For example, to modify a mesh within a utility plug-in, you will have to do it this way :

```
BOOL deleteit;
TriObject* tri          = GetTriObjectFromNode(node,deleteit);
if (!tri)return;
Mesh& mesh              = tri->GetMesh();

//Modify the mesh
FunctionthatModifyTheMesh();

//Update all mesh caches
mesh.InvalidateGeomCache() ;
mesh.InvalidateTopologyCache() ;

// Update the TriObject
Tri->NotifyDependents( FOREVER,PART_ALL,REFMSG_CHANGE );
Tri->NotifyDependents( FOREVER,0,REFMSG_SUBANIM_STRUCTURE_CHANGED );

//Update the Node
node->NotifyDependents( FOREVER,PART_ALL,REFMSG_CHANGE );
node->NotifyDependents( FOREVER,0,REFMSG_SUBANIM_STRUCTURE_CHANGED );

if (deleteit)
tri->DeleteMe();

ip->ForceCompleteRedraw();
```

14.4. Global Utility Plug-ins

They are called GUP plug-ins. They are utility plug-ins with no user interface and they are launched directly by the plug-in manager but there is no button to call them.

To remember : GUP are the first loaded plug-in category. Most of the time they are used to start a DCOM server so they need to be initialized first.

It is possible to use this kind of plug-in to modify the user interface by adding / removing menus from the 3D Studio Max UI. You will have to set your callbacks by subclassing the controls. See the MSDN documentation about that.

We can also use GUP as libraries or kernel of several plug-ins. This is because, the system offers a way to retrieve these plugins without being linked statically with them. You can retrieve a GUP just with its Class_ID by using the global function OpenGupPlugIn defined in the GUP.h header file.
So this way you will only include the header file of your GUP plugin which will contain its interface (pure virtual classes in C++) and you could cast the GUP pointer retrieved with OpenGupPlugIn into the suitable interface…
So it will be a kind of registered component inside 3D Studio Max without having the complexity of Microsoft COM objects. You could also have used the usual DLL system with the LoadLibrary and getProcAddress Win32 functions but it remains easier using a GUP plug-in to do so.

For example, considering MyGUP as a subclass of GUP and MYGUP_CLASS_ID uniquely defined as the MyGUP Class_ID we could do :

```
GUP* mygup = OpenGupPlugIn(MYGUP_CLASS_ID);
if( ! mygup )return;

MyGUP* m_MyGUP = (MyGUP*)( mygup );
```

It is also possible to list all registered GUP into 3D Studio Max through the GUP Manager class, a global variable named gupManager is instanced and defined in the GUP.h header file.

15. From C++ to Maxscript and vice versa

We have seen previously the GUP. It can be used also to call Maxscript commands easily from C++.
Because the GUP class provides you 2 functions to do so :

```
bool  ExecuteStringScript ( TCHAR *string );
bool  ExecuteFileScript ( TCHAR *file );
```

- ExecuteStringScript executes a Maxscript command string. This is a string as you would enter it into the Maxscript Listener.
- ExecuteFileScript takes a complete path filename of a Maxscript file to be executed.

Now let's see the contrary as it is possible to expose C++ functions from a plug-in to Maxscript calls.

There is a plug-in category specific for that purpose which is called a .DLX because its file extension is usually DLX. The X in DLX means Maxscript extension. But it becomes rare because you can expose your C++ functions in any plug-in category.
I usually use a GUP to do so. Why a GUP ? Because they are the first loaded plug-in category so if you launch a Maxscript file at startup your Maxscript extensions will be loaded.

Let's see how to do that.
You need to use the Maxscript SDK which is part of the 3D Studio Max SDK which deals with Maxscript classes, so you extend Maxscript by adding custom classes and functions.
First of all you will need to include Maxscript headers, they are in maxsdk\include\maxscrpt.

Warning : The subdirectory of maxsdk\include is "maxscrpt" without the "i" letter.

So as an example, including the headers files would result in :

```
#include "Maxscrpt\Maxscrpt.h"
#include "maxscrpt\Strings.h"
#include "maxscrpt\arrays.h"
#include "maxscrpt\numbers.h"
#include "maxscrpt\maxobj.h"
#include "maxscrpt\definsfn.h"
```

Warning :

- In this example, we have included the files to deal with strings, arrays, numbers, Max objects. Not all of them are necessary for all Maxscript C++ development, this is only an example.
- Including these headers files in the **good order** has a huge importance as if you don't set them in the good order, everything will compile and link however your Maxscript extensions, functions or classes, will not work.
  So the rules are :
  - Always let the maxscrpt.h file be the first Maxscript includes.

- Always let the definsfn.h file be the last Maxscript includes.

So if you need to add more Maxscript SDK header files, include them between these 2 include commands.

Now you can use Maxscript SDK into your code, say you want to expose a function named : "MyFunction" that just returns the Maxscript value false.
Do the following :

```
def_visible_primitive(MyFunction,   "MyFunction" );
Value* MyFunction_cf(Value** arg_list, int count)
{
      return &false_value ; //return the Maxscript false value
}
```

As you can see, we have used a macro command to register the function. This is the def_visible_primitive macro that exposes our function.
Then we implement it as a global C++ function named MyFunction_cf and avariable list of parameters.

Warning :

- Don't forget to add "_cf" to the implementation of your function !
- The function definition is fixed and can't be modified by returning something else than a Value for example. Value is the base class for all types in Maxscript SDK (integers, floats, nodes pointers etc…).

And in this case we return the Maxscript false global value.
So, by defining this, you have exposed a C++ function to Maxscript calls and your function can be called in the Maxscript listener or in a Maxscript file by the command : "MyFunction ()". This is the simplest example.

Now let's see how to set parameters to our function so that our function should be called with 2 parameters : a string and an array of INode pointers. It should return the Maxscript true value if the function worked fine or false if not.
The definition of the function remains the same. What changes is that we take care of the variable arguments passed to the function :

```
def_visible_primitive(MyFunction,   "MyFunction" );
Value* MyFunction_cf(Value** arg_list, int count)
{
      check_arg_count(MyFunction, 2, count);
      type_check(arg_list[0], String, "[The first argument of MyFunction
      should be a string that is a full path name of the file to export]");

      type_check(arg_list[1], Array , "[The 2nd argument of MyFunction
      should be an array of nodes]");

      //Get first parameter as a string
      const char* fullpathfilename  = arg_list[0]->to_string();
      bool res = DoSomethingWithThisString(fullpathfilename);
      if (! res) return &false_value);

      //Get 2nd parameter as an array
      Array* BonesArray       = static_cast<Array*>(arg_list[1]);
      const int   ArraySize   = BonesArray->size;

      //Get all elements from array
      for (i=0;i<ArraySize;i++)
      {
            //Check if each element is a INode
            if (BonesArray->data[i]->is_kind_of(class_tag(MAXNode)) )
            {
                  INode* _node = BonesArray->data[i]->to_node();
                  if (! node)return &false_value);

                  res = DoSomethingOnTheNode(node);
                  if (! res) return &false_value);
            }
      }

      return &true_value ; //return the Maxscript true value
}
```

So to check the number of parameters sent to the function, we use the macro `check_arg_count` which returns an error in the listener if the number of arguments is not the one we expected.
Then we check for the type of the arguments by using the macro : `type_check`, the first parameter is a parameter of the function from the array of parameters, the second is the type to check here String or Array which are Maxscript SDK classes whose headers should be included. And the third parameter is the error message if the parameter has not the expected type.

Then we use others macros to cast the untyped arguments into their suitable type. E.g. : `to_string` (); or to_integer(); etc…

16. The time

3D Studio Max lets you create animation, to do so, obviously, it uses a timer and the SDK provides you some functions to deal with time.
Fro example we have already seen the Interface::GetTime function to retrieve the current time in the UI.
Before talking about animation, we need to understand what are the time concepts used.

To remember :
Time is stored internally in 3D Studio Max as an integer number of ticks. Usually, each second is divided in 4800 ticks.

This value is chosen in part because it is evenly divisible by the standard frame per second settings in comm.on use (24 -- Film, 25 -- PAL, and 30 -- NTSC).

To remember :
The data type used to store a specific instance of time is the TimeValue. A TimeValue stores the number of ticks the time represents. **When a developer specifies time to almost all the functions in the SDK they use TimeValue**.

Above the tick units are the frames units then the number of frame per seconds (FPS) is defined by the user in the UI.

So to express a TimeValue in seconds, we have to translate the values in frames then in seconds to do that the SDK provides 2 global functions :

```
int GetTicksPerFrame();
int GetFrameRate();
```

So as an example you can translate a time expressed in seconds in TimeValue this way :

```
int t_in_seconds ; //Valid time in seconds units
const TimeValue t  = t_in_seconds  * GetTicksPerFrame() * GetFrameRate();
```

Now, we are going to see the key frames animation.

17. Retrieving keyframes animation

The SDK lets you get / set keyframes animation. To do so, 3D Studio Max uses controllers. A controller stores keyframes and interpolate between these ketframes. So it creates an animation curve that you can edit in the UI.
It exists most common types controllers e.g. float controllers, Point3 controllers, quaternions etc..

Note :
1) Each controller has its own interpolation algorithm, so all keys will be interpolated using this algorithm. However, you can change the tangent of the keys when that is possible to change the animation curve.

2) There are some cases where you can't get the keyframes directly. So you need to sample your animation by taking at a fixed time interval a sample of your animation curve. It is afterwards possible to simplify this animation curve using polynomial interpolation of Bezier curves fitting.

We are going to see in details how to get the keyframes of an animation controller about the position of an object. We will focus only on the X component of the position.

```
Control* _tmcontrol = _node->GetTMController();
if (! _tmcontrol)return;

Control* _xposcontrol = _tmcontrol->GetPositionController()
                                  ->GetXController();
if (_xposcontrol)
{
      //Please, don't use a Bezier position controller (Max default
      //controller) as an example, as it doesn't have a key control
      //interface...
      IKeyControl*      ikc   = GetKeyControlInterface( _xposcontrol );
      if (! ikc)
      {
            //we have to sample this animation...
      }
      else
      {
            const int numKeys = ikc->GetNumKeys();

            //Position
            //TCB Interpolation
            if ( _poscontrol->ClassID() ==
                  Class_ID( TCBINTERP_POSITION_CLASS_ID, 0 ) )
            {
                  ITCBPoint3Key key;
                  for (int i=0;i<numKeys;i++)
                  {
                        ikc->GetKey( i, &key );
                  }
            }
            else //Bezier interp.
            if ( _xposcontrol->ClassID() ==
                  Class_ID( HYBRIDINTERP_POSITION_CLASS_ID, 0 ) )
            {
                  IBezPoint3Key key;
                  for (int i=0;i<numKeys;i++)
                  {
                        ikc->GetKey( i, &key );
                  }
            }
            else //Linear interp.
            if ( _xposcontrol->ClassID() ==
                  Class_ID( LININTERP_POSITION_CLASS_ID, 0 ) )
            {
                  ILinPoint3Key key;
                  for (int i=0;i<numKeys;i++)
                  {
                        ikc->GetKey( i, &key );
                  }
            }
      }
```

18. Sub-anims

In the SDK the term **anim** refers to something derived from class Animatable.
This anim can be anything such as a node, object, controller, material, texmap, parameter block, etc.
Further, any item that has sub-items has what are referred to as **sub-anims**. This hierarchy of items, the parent item and it's sub-items, is referred to as the sub-anim hierarchy. The sub-anim hierarchy can be thought of as the Track View hierarchy.

Below is a partial screen capture of Track View showing a few anims and their sub-anims.

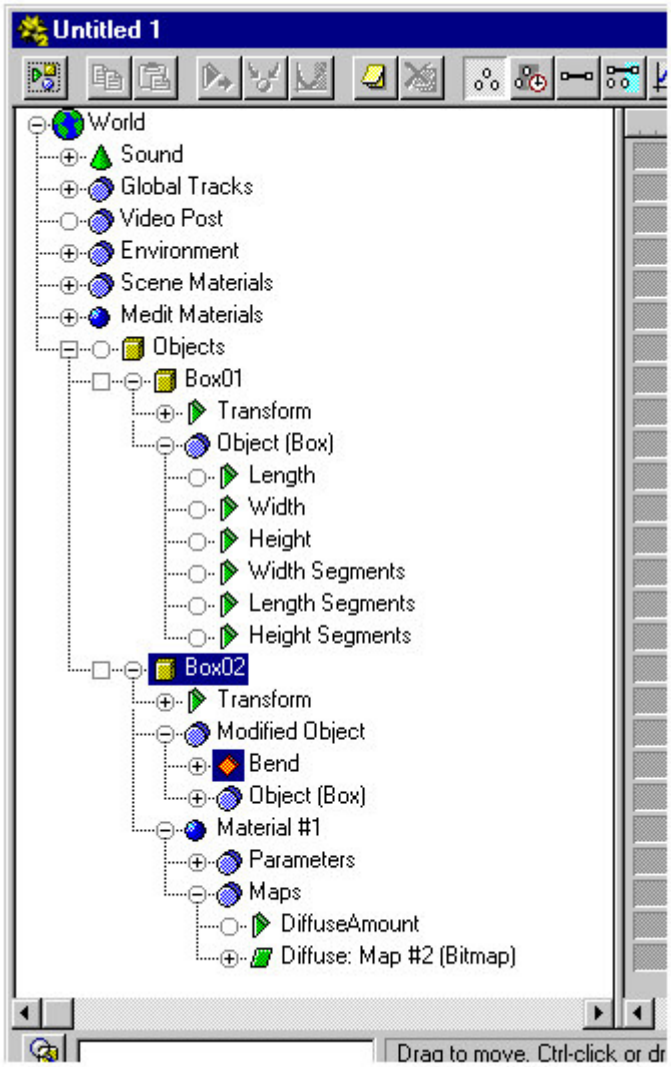

Figure 13 : The trackview showing subanims

For example we have seen that a node contains 6 sub-items that were :

- An animation controller for its 3D transform
- The object reference. Each node maintains a pointer to an object this a pointer to the base procedural object or derived object (BaseObject or IDerivedObject see section **Erreur ! Source du renvoi introuvable.**).
- The « Pin Node » for Inverse Kinematics (set to NULL by defaut).
- The material reference

- The visibility controller
- The image blurr controller (set to NULL by defaut)

All these are sub-anims of the node.

Another example is a procedural sphere. It contains parameter such as its ray. These parameters are sub-anims of the sphere.

Note :
Obviously, sub-anims can contain others sub-anims.

We can access all of them by using the following functions from Animatable class :

```
int Animatable::NumSubs();
Animatable* Animatable::SubAnim(int i)
TSTR Animatable::SubAnimName(int i);
```

- The NumSubs function returns the number of sub-anims of an anim object.
- SubAnim returns the ith sub-anim.
- SubAniname returns the name of the i^{th} sub-anim.

Example :
From a Standard material (StdMat), we can retrieve which sub-anims are animated.

```
StdMat* _stdmat; valid pointer

const int numsubanims = stdmat ->NumSubs() ;
for ( int j=0; j<numsubanims; j++ )
{
      Animatable* subAnim = _ stdmat ->SubAnim( j );
      if ( subAnim && subAnim->IsAnimated() )
      {
            //This sub-anim is animated
      }
}
```

To remember :
To retrieve the sub-anims of an anim, you can use the trackview in the UI.

19. References

In the 3D Studio Max architecture, elements of the scene often form dependencies on one another. The typical manner these dependencies are handled in 3ds max are through References pointers. A reference is a record of dependency between a reference maker and a reference target. The reference maker is said to be dependent upon the reference target. If the target changes in some way that affects the maker, the maker must be notified so it may take appropriate action.

So the 2 important classes are ReferenceMaker and ReferenceTarget to handle references.

Note :
This use of the term reference in this section should not be confused with the term reference used in the 3d Studio Max interface and user manuals. Nor is it to be confused with the C++ definition of reference.
In this section, the term reference will always apply to the notion of a dependent relationship unless specifically stated otherwise.

The ReferenceTarget class is derived from ReferenceMaker. And most plug-ins are subclasses of ReferenceTarget.

So, there are 2 main functions when working with references, they are :

```
RefResult ReferenceMaker::MakeRefByID (Interval refInterval,int which,
RefTargetHandle htarget);

RefResult ReferenceTarget::NotifyDependents(Interval changeInt, PartID
partID, RefMessage message, SClass_ID sclass=NOTIFY_ALL,BOOL
propagate=TRUE, RefTargetHandle hTarg=NULL );

RefResult ReferenceMaker::NotifyRefChanged(Interval changeInt,
RefTargetHandle hTarget, PartID& partID,RefMessage message);
```

MakeRefByID creates a reference between the object which calls the function, and the ReferenceTarget specified by the htarget parameter. The target then maintains this record of dependency via its pointer back to the reference maker. An index is used to retrieve this reference as well as a validity interval.

NotifyDpendents is used when a reference target changes to notify its dependent reference makers of this change. As in modifiers, the “channels” where the change has happened is specified using an enum.

NotifyRefChanged is the function called when a dependency has been changed. This function should be implemented in a subclass of ReferenceMaker.

When an instance of ReferenceTarget notifies a change, it gives an enum of what this change was dealing with. Here are a few examples of reference messages

- REFMSG_NODE_NAMECHANGE : specifies that the name of a node has changed.
- REFMSG_TM_CHANGE specifies that the 3D transform of a node has changed
- Etc..

The messages list is very complete and won't be described in detail. We will let the user see the SDK documentation in the section "List of Reference Messages and their PartID parameters".

As for sub-anims, it is possible to list and retrieve all references of an object by using :

```
int ReferenceMaker::NumRefs();
RefTargetHandle ReferenceMaker::GetReference(int i);
```

For maintenance about references, 4 functions are used, their names could be confusing, so you should really now what each of them do :

```
RefResult DeleteAllRefsFromMe();
RefResult DeleteAllRefsToMe()
RefResult DeleteAllRefs();
RefResult DeleteReference(int i);
void      DeleteMe();
```

DeleteAllRefsFromMe : this method deletes all references from this ReferenceMaker. This also sends the REFMSG_TARGET_DELETED message to all dependents.

DeleteAllRefsToMe : This method deletes all references to the target. This is often used when an item is being deleted and it wants to remove all references from itself..

DeleteAllRefs : Deletes all references to and from the calling reference maker..

DeleteReference : deletes a particular reference from its index. Chaque objet utilise en interne un compteur qui compte le nombre de fois où il est référencé, lorsque ce compteur tombe à 0 l'objet est effacé par la méthode DeleteThis().

DeleteMe deletes all references to and from the calling reference maker (by calling DeleteAllRefs function), sends the REFMSG_TARGET_DELETED message, handles Undo, and deletes the object.

To remember :
So the best way to delete a Max object correctly is using the DeleteMe function which is the more complete and will handle undo/redo.

Note :
The reference array inside an object can be scanned using NumRefs() and GetReference(int i) functions. But you should be careful as the GetReference may return NULL pointers sometimes. Because when you delete a reference by using DeleteReference function, it deletes the reference inside the array but it does not resize the array, its slot will contain a NULL reference pointer…

You should know that iit is also possible to find/ replace / read / save references from inside an object but we won't cover this topic in this document.

20. Custom User interface controls

A set of custom controls are available for use in the user interface design of 3D Studio Max plug-ins.

These controls provide an important element of consistency between the plug-in and the system, making new plug-ins appear fully integrated with 3D Studio Max.

This lets you create controls of type such as :

- Edit box
- Spinner
- Slider
- Check, Pus hand icons buttons.
- Status
- Toolbar
- Curve Control
- Image
- Color Swatch
- Rollup
- Thumb Tack (enables the “always on top” property for a window)
- Off screen buffer
- TCB Graph
- Drag and Drop

We won’t describe here how to set them in a dialog box from the resource as we will se that in a practical way.

The code samples from the SDK are full of examples. E.g. you can have a look at the cpp file CUSTCTRL.CPP and custcont.h for the header file about custom controls.

Let’s see an example on how to create a spinner of type integer with a range from 1 to 4 with 1 as a default value (initialization).

Spinners are created using 2 controls, a spinner control and an edit box which are linked in some way.

Let IDC_CUSTOM_SPINNER be the resource ID in the header file resource.h of the spinner control and IDC_CUSTOM_EDIT be the ID of the edit box linked to the spinner and m_hWnd be the HWND of our dialog box.

We create the spinner in our dialog box this way :

```
ISpinner* iSpin = SetupIntSpinner(m_hWnd, IDC_CUSTOM_SPINNER,
IDC_CUSTOM_EDIT, 1, 4, 1);
```

This is not the only way to create our spinner but it’s the quickest way as it does everything at once.

An dit the callback handling messages of our dialog box, we add the following messages handlers :

```
static INT_PTR CALLBACK MyDlgProc(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
	switch (msg)
	{
		case CC_SPINNER_CHANGE:
		{
			const int Val = iSpin->GetIVal();
			//Update something from new spinner value
		}
		break;
	}
	Return TRUE;
}
```

So, there are some others spinners message we won't describe here. Anyway, you have the main elements to deal with custom controls.

21. Parameters blocks and parameter maps

The parameter block class provides a mechanism for storing values for a plug-ins parameters. When a parameter block is created, the developer specifies the number of parameters and the types of each parameter. Parameter types consist of a range of built in types like integer, float, 3D Point, and Color.

Parameters may be animated or constant. In order for a parameter to be animatable, it must have a controller to control the animation. Different parameter types require different controller types. For example, a floating point value, like the angle parameter for the bend modifier, requires a floating point controller. A node transformation matrix requires a transform controller. The most common controllers are interpolating or 'key frame' controllers

One of the main purposes of parameter blocks is to manage the complexity of maintaining different controllers for different parameters.

This mechanism is interesting if you want to set a lot of parameters with different controllers into your plug-in.
It will also automatically do for you :

- Animation of these parameters
- Reading / saving of these parameters
- Building of the dialog box controls to control your parameters thanks to parameter maps
- Exposing the parameters to Maxscript

They can be automatically added to your project by using the plug-in wizard. This is the simple way to create them.

So, we are going to focus on how to create them from scratch manually by adding an instance of the ParamBlockDesc2 class.
This class is a little bit complex as we have to give to its constructor all parameters information we want to build. If you make a mistake giving these information, it won't work…

Here is an example which builds 3 parameter blocks with 2 checkboxes which will be Booleans values and a float spinner. The controls will be generated in the Rollup panel named IDD_PANEL (on next page) :

```
static ParamBlockDesc2 weldvertexanduvs_param_blk
( weldvertexanduvs_params, _T("params"),  0, &WeldVertexAndUVsDesc,
P_AUTO_CONSTRUCT + P_AUTO_UI, PBLOCK_REF,

      //rollout
      IDD_PANEL, IDS_PARAMS, 0, 0, NULL,

      // params
            shared_uvs_only,  _T("shared UVs only"),  TYPE_BOOL,0,"Weld the
                  shared UVs only",
            p_default,        1,
            p_ui,TYPE_SINGLECHEKBOX,IDC_CHECK_SHAREDUVONLY,
            end,

            weld_selected,    _T("Weld Selected"), TYPE_BOOL, 0,"Weld
                  Selected Vertices",
            p_default,        0,
            p_ui,TYPE_SINGLECHEKBOX,IDC_CHECK_WELDSELECTED,
            end,

            weld_select_threshold, _T("WeldSelected Threshold"),
                  TYPE_FLOAT, P_ANIMATABLE, "Weld Selected Threshold",
            p_default, 0.1f,
            p_range, 0.0f, 100.0f,
            p_ui, TYPE_SPINNER, EDITTYPE_FLOAT, IDC_CUSTOM_EDITTHRESHOLD,
                  IDC_CUSTOM_SPINTHRESHOLD, SPIN_AUTOSCALE,
            end,
      end
);
```

In your plug-in class you will have defined
IParamBlock2 *pblock; //ref 0

Then into your class constructor you will call the method MakeAutoParamBlocks

And inside your plug-in, you could call the following to get a parameter block value :

```
BOOL bSHaredUVsOnly ;
pblock->GetValue(shared_uvs_only,t, bSHaredUVsOnly,FOREVER);
```

Let's see the command modes.

22. The command modes

A plug-in may want to process mouse interaction in any of the viewports, this is done through the Command modes.

For example, the zoom, rotate, pan of cameras are command modes. Because 3D Studio Max works internally with command modes, there is always an active command mode (select, move, rotate, scale etc…).

To deal with this command modes, a stack of modes has been created. Here are the functions to use them :

```
void PushCommandMode(CommandMode *m);
void SetCommandMode(CommandMode *m);
void PopCommandMode();
CommandMode* GetCommandMode();
void SetStdCommandMode(int cid);
void PushStdCommandMode(int cid);
void RemoveMode(CommandMode *m);
void DeleteMode(CommandMode *m);
int GetCommandStackSize();
CommandMode * GetCommandStackEntry(int entry) ;
```

Note :
By default, in a command mode, right clicking with the mouse cancels the action and pops the last command mode from the stack so it gets back to the previous active command mode from the stack.

The mouse is handled using the function CommandMode::MouseProc. It returns a MouseCallBack class instance and the number of clicks necessary to perform a complete operation.

Note :
Be careful, in the number of mouse clicks when the user releases a button is counted as a mouse click. So to deal with one mouse click in the viewport, the command mode should returned 2 clicks (one for the mouse button down message and the other for the mouse button up message)

```
MouseCallBack *CommandMode::MouseProc(int *numPoints);
```

The MouseCallBack class has a few function whose main function is the MouseCallBack::proc function.

```
int MouseCallBack::proc(HWND hwnd, int msg, int point, int flags,
                        IPoint2 m );
```

This function gets called by the system when an event occur.

This function gives you in its parameters respectively :

- The HWND of the viewport wher the event has ocured.
- The current mouse message such as mouse down, mouse up, mose move… Be careful, these are not standard Win32 messages !
- The current point number : O is the first click (mouse down), & is the second click (mouse up) etc…
- Flags enables you to know if the shift, control or alt keys are pressed or not while the event occures.
- The 2D point in the viewport coordinates where the user has clicked. It is possible to translate this 2D point in a 3d ray to intersect anything in the 3D scene usin the viewport methods (ViewExp class)

It exists some others functions from Interface class to test if elements are inside a selection rectangle. The main function is :

```
int Interface::SubObHitTest(TimeValue t, int type, int crossing, int flags,
                            IPoint2 *p, ViewExp *vpt);
```

We will let the user see this in details in the documentation.

23. The Matrix3 class

3D Studio Max handles 3D transforms with the Matrix3 class.
It is just a matrix with 4 lines and 3 rows.
It contains a 2 dimensions float array :

```
float m[4][3];
```

Which leads into the following schema :

m[0][0]	m[0][1]	m[0][2]
m[1][0]	m[1][1]	m[1][2]
m[2][0]	m[2][1]	m[2][2]
m[3][0]	m[3][1]	m[3][2]

the following matrix is called the identity :

1	0	0
0	1	0
0	0	1
0	0	0

In fact, this matrix is a 3x3 matrix containing the scale and the rotation combined on the 3 firsts lines and the last line is the translation part.

The Matrix3 class is very complete and lets you build rotation Matrices from quaternions, Euler angles, axis and angle… We invite you to have a further look at the functions of this class.

To remember :

When we multiply a Point3 (3d vector) by a Matrix3, the operations are done in the following order :

- Add translation
- Multiply by rotation
- Multiply by scale

To remember :
To combine matrices, you will have to multiply them on the right (called premultiply).

Example :
To combine the 3 matrices A, B and then C in this order into a ResultMatrix, we need to write :

Matrix3 ResultMatrix = C * B * A ;

It can be important to decompose a matrix into its affine parts that are its translation, rotation scale and scale axis.
This can be handled by the AffineParts structure and the global function :

```
void decomp_affine(Matrix3 A, AffineParts *parts);
```

Decomp_affine decomposes the Matrix3 into its affine parts.
And the AffineParts structure contains :

```
Point3 t;
//The translation components.
Quat q;
//The essential rotation.
Quat u;
//The stretch rotation. This is the axis system of the scaling application.
Point3 k;
//The stretch factors. These are the scale factors for x, y and z.
float f;
//Sign of the determinant.
```

This decomposition comes from the following article :
Graphics Gems IV - Polar Matrix Decomposition by Ken Shoemake. ISBN 0-12-336155-9.

24. The template class Tab

This is a generic table class. This is a type-safe variable length array which also supports list-like operations of insertion, appending and deleting.

It is very useful as it avoids using hardcoded size arrays. So it will save memory usage.
It drawback is to be slower than using a fixed size array, because of memory re-allocation.
But you can partly overcome this issue.

To remember :
You can use these arrays only with elements of the array that don't allocate memory.
And if memory usage has to be preferred to speed optimizations.

Notes :
- Be careful, this class don't use the copy constructor when reallocating memory.
- If you want to use elements that allocate memory, use pointers on these elements instead of setting elements directly in the array.

These arrays can be used locally inside a function. They works internally with 2 numbers that are :
- nalloc : is the number of elements allocated in the array
- count : is the number actual used.

So you always get the relationship : count <= nalloc.

The allocation/deallocation is automatically done when you use the insert and remove functions.

Let's see the main methods of this class :

```
int Count() const
void ZeroCount()
void SetCount(int n, BOOL resize=TRUE)
int Append(int num, T *el, int allocExtra=0)
int Insert(int at, int num, T *el)
int Delete(int start,int num)
void Sort(CompareFnc cmp)
T& operator[](const int i) const
int Resize(int num)
void Shrink()
```

Count : returns the number of entries being used (so the count variable, not the nalloc number)

ZeroCount : set the number of elements in the array that are actually used to zero. But it does not deallocate anything.

SetCount : Set the number of elements in the array that are actually used to n. It allocates / deallocates by default the memory used.

Append : Appends "num" elements position on end of array, if it needs to enlarge the array, allocate "allocExtra" extra slots.

Insert : Inserts "num" elements positionned at "at" index.

Delete :
List-type delete of "num" elements starting at the "start" position. No memory is deallocated, count has been updated but nalloc has remained the same. You should call the Shrink() function after a Delete to ensure memory is deallocated.

Sort : sorts the elements of the array using the standard C library qsort function. You provide a function pointer to compare the elements in the CompareFunc type.

The [ ] operator lets you access directly the elements of the array.

Resize : Changes the number of allocated items to num, so it allocate/deallocate memory consequently.

Shrink : reallocates so there is no wasted space (nalloc = count)

Example with integers :

```
Tab<int> UsedFaces;
UsedFaces.SetCount(0);

//Process all indices
for (int i=0;i<numfaces;i++)
{
	int adr = i;
	UsedFaces.Append(1,&adr);
}
```

For each iteration, the array is going to be reallocated and all elements will be copied which can be time consuming.
To overcome that, you can specify the number of extra elements to allocate instead of adding one element at each iteration, so you will improve the speed of your function.
So it the las example, replace :

```
UsedFaces.Append(1,&adr);
```

By :

```
UsedFaces.Append(1,&adr,100);
```

100 extra elements will be allocated when i twill be necessary and nothing will be allocated is count < nalloc.
After the for loop is finished you can call Shrink() to ensure no waste of memory.

Our array is filled with integers from 0 to numfaces+1, we use the following function to read it :

```
const int NumUsedFaces = UsedFaces.Count();
for (int i=0;i< NumUsedFaces;i++)
{
	Const int IndexFromArray = UsedFaces[i];
}
```

The final example is on how to sort this array :

You have to define the integers comparison function this way :

```
int __cdecl SortTabInt(const void *elem1, const void *elem2)
{
	/*
	returns
	< 0if elem1 less than elem2
	0 if elem1 identical to elem2
	> 0   if elem1 greater than elem2
	*/

	int* Ielem1 = (int*)elem1;
	int* Ielem2 = (int*)elem2;
	if (*Ielem1 < *Ielem2)
		return -1;

	if (*Ielem1 == *Ielem2)
		return 0;

	//if (*Ielem1 > *Ielem2)
	return 1;
}
```

Then in another part of the code you use it like this :

```
CompareFnc cmp = SortTabInt;

//Sort our array
UsedFaces.Sort(cmp);
```

25. The IGame interface

The Game Export Interface is a high level API. It has been developed to greatly simplify data export from 3d Studio Max.

What you can retrieve easily using IGame includes :

- GeomObjects (Tri-mesh, Patch-meshes, Cameras, Lights…)
- Spline Objects
- Helper Objects
- Modifiers
- Unified direct access to Skin and Physique modifiers (to extract skinning information)
- Controllers access for key framed and sampled animation curves. Controllers access to Biped (character studio), Euler and constraints.
- Materials and bitmap textures
- All paramblocks

From these interfaces, you can also have a direct access to the original data so you can mix IGame and classic exporter functions.

Example about controllers
This is the original code, without using IGame to get the key frames of a TCb interpolation controller on the position of an object :

```
Control * cont = node->GetTMController()->GetPositionController()
if (! cont)return;

int i;
IKeyControl *ikc = NULL;

ikc = GetKeyControlInterface(cont);

// TCB point3
if (ikc && cont->ClassID() == Class_ID(TCBINTERP_POINT3_CLASS_ID, 0))
{
      for (i=0; i<ikc->GetNumKeys(); i++)
      {
            ITCBPoint3Key key;
            ikc->GetKey(i, &key);
            // do as you wish with the data
      }
}
```

The following does the same using IGame !

```
IGameKeyTab poskeys;
IGameControl * pGameControl = gameNode->GetIGameControl();
pGameControl->GetBezierKeys(poskeys,IGAME_POS));
```

So it’s worth trying... ☺

26. Undo / Redo

The SDK provides you a way to deal with the undo / redo stack. So that the user can undo/redo operations made into your plug-in.

To remember :
The class involved in undo/redo is the class Hold. There is a global variable of this class named "**theHold**".

The other class linked to this mechanism is the RestoreObj class. It provides you a way to store your modifications and apply/unapply them.

26.1. Undo / Redo mechanism

Usually, an undo / redo command is made using a block of commands always called in the same order which is :

```
theHold.Begin() ;
…
if ( theHold.Holding() )
{
      theHold.Put (new MyRestoreObj);
      theHold.Accept("My operation name");
}
```

Let's see in detail what are the Hold class main functions :

```
void Begin();
int Holding();
void Accept(int nameID);
void Accept(TCHAR *name);
void Cancel();
void Put(RestoreObj *rob);

void SuperBegin();
void SuperAccept(int nameID);
void SuperAccept(TCHAR *name);
void SuperCancel();
```

Begin : is used to tell the system we are going to use the undo/redo mechanism, after this function has been called the system is ready to accept restore objects. It should always be followed by the Put and Accept functions to record a complete undo operation.

Holding : returns ann integer saying if the system is waiting for a restore object. So it returns a non zero integer if Begin has been called but not the Accept function yet.

Accept : registers an undo object with the undo system. This will allow the user to undo the operation.

Cancel : is called instead of the Accept function, it throws out the registered restore object. This cancels the undo operation. You usually call it when the user can cancel your operation such as by using a mouse right click.

Put : the developer calls this method to register a new restore object with the system. Once this object is registered the developer should set the **`A_HELD`** flag of **`Animatable`**, i.e. call **`SetAFlag(A_HELD)`**. This indicates that a restore object is registered with the system. It uses the subclass of RestoreObj to apply/unapply the operation.

SuperBegin : normally this is NOT needed but in special cases this can be useful. This allows a developer to group a set of Begin()/Accept() sequences to be undone in one operation. Consider the case of the user using the Shift-Move command to create a new node in the scene. There are two parts to this process. First the node must be cloned and second the position must be tracked as the user moves the mouse to place the new node in the scene. Naturally if the user wanted to Undo this operation, they would expect a single selection of the Undo command would accomplish it. However the process was not a single operation. There was the initial cloning of the node, and then the iterative process of moving the node in the scene, restoring its position, moving it again, restoring it again, etc. Cases like this are handled using methods of theHold named SuperBegin(), SuperAccept() and SuperCancel(). These allow the developer to group several restore objects together so that they may be undone via a single selection of Undo.

SuperAccept : when a developer has used **`SuperBegin()`**, this method is used to Accept. It registers the restore object with the undo system. This will allow the user to undo the operation.

SuperCancel When a developer has used **`SuperBegin()`**, this method is used to Cancel. This restores the database to its previous state and throws out the restore object. This cancels the operation.

So to use the « Super » functions family, you should do It like this :

```
theHold.SuperBegin() ;

theHold.Begin() ;
theHold.Put (new MyRestoreObj1);
theHold.Accept("My operation name");

theHold.Begin() ;
theHold.Put (new MyRestoreObj2);
theHold.Accept("My operation name2");
…
…
…
theHold.Begin() ;
theHold.Put (new MyRestoreObj1);
theHold.Accept("My operation name 3");

theHold.SuperAccept("My operation grouped") ;
```

26.2. The RestoreObj class

This is the class that where modifications to undo/redo the operation are stored. Here are the main functions of this class :

```
void Restore(int isUndo);
void Redo();
TSTR Description();
```

Restore is called by the system to restore the scene when the user undo the operation, the scene is restored in the same state as when the call to the Hold::Accept was made. The IsUndo parameter is nonzero if Restore() is being called in response to the Undo command; otherwise zero. If isUndo is nonzero, the developer needs to save whatever data they need to allow the user to redo the operation.

Redo : this method is called when the user selects the Redo command. The developer should restore the database to the state prior to the last Undo command.

Description : is the character string which appears in the user interface to identify the operation.

Here is an example of a subclass of RestoreObj.
This class named NodeSelRestore stores a modification of a BitArray which tells which nodes of a graph are selected :

```
class NodeSelRestore : public RestoreObj
{
	BitArray		m_UndoNodeSelArray;
	BitArray		m_RedoNodeSelArray;
	PosGraphObject*		m_Obj;

	NodeSelRestore();

public:

	NodeSelRestore(PosGraphObject* _obj)
	{
		m_Obj = _obj;
		m_UndoNodeSelArray.SetSize(m_Obj->NodeSel.GetSize());
		m_UndoNodeSelArray = m_Obj->NodeSel;
	};

	void Restore(int isUndo)
	{
		if (isUndo)
		{
			m_RedoNodeSelArray.SetSize(m_Obj->NodeSel.GetSize());
			m_RedoNodeSelArray = m_Obj->NodeSel;
		}
		m_Obj->NodeSel.SetSize(m_UndoNodeSelArray.GetSize());
		m_Obj->NodeSel    = m_UndoNodeSelArray;
		GetCOREInterface()->ForceCompleteRedraw();
	}

	//To be continued on next page
```

```
	void Redo()
	{
		m_Obj->NodeSel.SetSize(m_RedoNodeSelArray.GetSize());
		m_Obj->NodeSel    = m_RedoNodeSelArray;
		GetCOREInterface()->ForceCompleteRedraw();
	}

	void EndHold() {}
	TSTR Description() {return GetString(IDS_UNDO_NODESEL);}
};
```

27. Function publishing system

The Function Publishing System, is a system that allows plug-ins to publish their major functions and operations in such a way that code outside the plug-in can discover and make enquiries about these functions and is thus able to call them though a common calling mechanism.

The whole system is very similar to Window's COM and OLE Automation systems and share many similar concepts in the architecture.
However, the Function Publishing System is not based on COM and OLE but instead is a custom architecture more suited and optimized for 3D Studio Max.
The Function Publishing API serves a number of purposes which allow 3rd party developers to open up important portions of their plug-ins for use by external sources, allowing for users to extend and control these directly. Some of the purposes of the API are:

- Modularizing plug-in code into various "engines" that are able to supply services to other parts of 3ds max and other 3rd party plug-ins and can be delivered to the user through various different user-interfaces.

- Providing automatic scriptability by exposing the published functions directly to MAXScript.

- Providing alternate means of invoking plug-in functions in the UI, such as via the new manipulator system, scripted menus, quad menus, hot keys, macroScripts, toolbar buttons, etc.

- Allowing the MAXScript Macro Recorder to automatically generate calls to the published functions, in the event that these are invoked by the expanded ParamMap2 system or other UI mechanisms such as hot keys or menu items.

- Facilitate automatic generation of COM interfaces and OLE Type Libraries in such a way that external COM clients can invoke the published functions in the plugin code.

What would a plug-in publish ?

The various kinds of functions published by a plug-in usually fall into the following categories. You can, however, publish anything you want.

- Important algorithms in the plug-in, for example, the Edit Mesh modifier might publish its face extrude and mesh attach functions, or the flex modifier its soft-body dynamics algorithms. In these cases, the functions would be parameterized in the most general way, independent of any current scene state or UI mode in 3D Studio Max, for example, the face extrude might take a Mesh, a set of faces and a distance.

- Functions that enquire about or affect the state of one of the plug-in's objects in the scene. Usually, these are unnecessary if the plug-in stores its state as parameters in ParamBlock2s, which are already accessible externally, but in cases were this is not

done or certain kinds of state are not cleanly accessible via the ParamBlock2 system, extra functions may be published by the plug-in. These usually take an instance of one of the plug-in's objects as one of their parameters, and should be independent of any UI mode, for example, they should not require that the object being manipulated be the current focus in the Command Panel.

- UI action functions. These basically provide a programmatic way of "pressing" buttons and keys in the UI for a plugin and are specifically meant to be UI modal. They take no parameters, since these are defined by the current state in the UI. For example, the vertex delete action function for an Editable Mesh object would operate on the current vertex selection for the current object in the Modify panel. As well as being automatically exposed in the scripter like other published functions, Action functions are automatically entered into the ActionTable system. It basically holds all the commands and actions that may be bound to hotkeys or added to menus or put in buttons on a toolbar using the Custom User Interface (CUI) system, so that by publishing your action functions using the FnPub system, you are making them automatically available for binding to hotkeys and placing in menus and toolbar buttons. Action functions are treated specially in the FnPub system, and have extra descriptor data for things like menu item text, tooltip text, enable predicates, etc.

27.1. The FP mechanism

The Function Publishing API consists of two main components, a function descriptor system and a function calling mechanism. The function descriptor is pretty similar to the COM .idl file (interface definition language) it is an instance of the class FPInterfaceDesc. The function calling mechanism is pretty similar to the COM interface IDispatch.

One or more interfaces are generated by your plug-in at your discretion. It can be useful to group functions together in separated interfaces. All interfaces will inherit from the FPInterface class.

You can use these interfaces to call the functions or use the calling mechanism and call the function using an ID and a table of parameters. This is similar to the IDispatch::Invoke command.

Each interface is defined in a unique way by an Interface_ID. It is similar to the class_ID. And each function of the interface has a unique identifier called a FunctionID which is basically the same idea than the BlockID and ParamID used in the ParamBlock2 class.

27.1.1. *Interfaces organization*

They are organized in 3 separated code sections.

- The public interface set in a header file apart.
- The interface implementation and virtual functions in a .cpp file.
- The interface description in a .cpp file.

2 kinds of interface are available :FPStaticInterface and FPMixinInterface.
The first is used for interfaces that can be seen as global static functions. That is that these functions are not linked to a specific object.
The second interface category is used to publish part of all functions that are linked to a specific object, so it won't work without having this object created.

Note :
If you choose to use the FPStaticInterface, you will have to give its functions all parameters necessary to perform their operation.
Example : for a function that modifies a Mesh in some way, you will need to pass a valid Mesh pointer to the function.
If you had used the FPMixinInterface linked to a Mesh object, it could directly act on this Mesh.

Example of published interface in a header file :

```
//Interface ID
#define BOOSTEDUNWRAP_INTERFACE Interface_ID(0x47f64915, 0x35f57490)

//Mixin interface
class IBoostedUnwrap : public FPMixinInterface
{
      public:
      // function IDs
      enum { unwrap_edit, unwrap_deconnectuv, unwrap_resizetexture};

      BEGIN_FUNCTION_MAP

      VFN_0( unwrap_edit            , iPressEdit            );
      VFN_0( unwrap_deconnectuv     , iDeconnectUV          );
      VFN_1( unwrap_resizetexture   , iResizeTexture, TYPE_BOOL   );

      END_FUNCTION_MAP

      FPInterfaceDesc* GetDesc();

      virtual void iPressEdit       (void)            = 0;
      virtual void iDeconnectUV     (void)            = 0;
      virtual void iResizeTexture   (BOOL resize)     = 0;
};
```

This interface is a FPMixinInterface because it is implemented inside a Modifier.

In the previous example, you can see that the definition of the interface is set in a block beginning with the macros BEGIN_FUNCTION_MAP and ended by END_FUNCTION_MAP. This is where is the description of our published function.

VFN_O means this is a function that returns void and takes 0 parameters
VFN_1 means this is a function that returns void and takes 1 parameters, the parameter is of type BOOL.
And we have the interface description and the 3 pure virtual function to implement in our class.

So here is what we define in our modifier from this FP interface :

```
class UnwrapMod : public Modifier, public IBoostedUnwrap
{
      public:

            //From Class IBoostedUnwrap
            virtual void iPressEdit       (void);
            virtual void iDeconnectUV     (void);
            virtual void iResizeTexture   (BOOL resize);

            //Function Publishing method (Mixin Interface)
            BaseInterface* GetInterface(Interface_ID id)
            {
                  if (id == BOOSTEDUNWRAP_INTERFACE)
                        return (IBoostedUnwrap*)this;
                  else
                        return FPMixinInterface::GetInterface(id);
            }
Etc …
} ;
```

We define the virtual function in our interface as well as the virtual function GetInterface by specifying which interface we can get from our modifier.
This is because you can use Animatable::GetInterface to retrieve a specific interface on a particular object.

And finally, in a .cpp file from our modifier project stands our interface description :

```
//Function Publishing descriptor for Mixin interface
//***************************************************
static FPInterfaceDesc boostedunwrap_interface
(
    BOOSTEDUNWRAP_INTERFACE, _T("InternalBoostedUnwrap"), 0, &unwrapDesc,
      FP_MIXIN,
      IBoostedUnwrap::unwrap_edit, _T("Edit"), 0, TYPE_VOID, 0, 1,
                  _T("object"), 0, TYPE_REFTARG,
      IBoostedUnwrap::unwrap_deconnectuv, _T("DeconnectUV"), 0, TYPE_VOID,
            0, 1, _T("object"), 0, TYPE_REFTARG,
      IBoostedUnwrap::unwrap_resizetexture, _T("ResizeTexture"), 0,
            TYPE_VOID, 0, 1,  _T("object"), 0, TYPE_REFTARG,
      end
);

//  Get Descriptor method for Mixin Interface
//  *****************************************
FPInterfaceDesc* IBoostedUnwrap::GetDesc()
{
      return &boostedunwrap_interface;
}
```

28. Additional resources

- Articles on Gamasutra :
  - « Choosing Between Utility and Modifier Plug-Ins for 3D Studio Max » by David Lanier. http://www.gamasutra.com/features/20000614/lanier_01.htm
  - "From Direct3D to 3D Studio Max" by Loïc Baumann http://www.gamasutra.com/features/19980320/baumann_01.htm
  - "Where'd It Go? It Was Just Here! Managing Assets for the Next Age of Real-Time Strategy Games" by Herb Marselas : http://www.gamasutra.com/features/20010221/marselas_pfv.htm
- Web sites:
  - ADN Sparks, the program for Autodesk developers : http://www.autodesk.com/adnsparks
  - SARL David Lanier 3D : http://www.dl3d.com for tools developers forums
  - Resources about 3D Studio Max programming on SARL David Lanier 3D forums : http://dl3d.free.fr/phpBB2/viewtopic.php?t=436 (a lot more online…)