The University Of Teesside
School Of Computing
Middlesbrough
Cleveland TS13BA

Plug-in Development with C++ and the 3D's Max SDK

BA (Hons) Computer Animation

Afaq Sabir

May 21, 2012

Supervisor: Nick Daniel
Second Reader: Paul Noble

Contents

1. Abstract **3**

2. Introduction **3**

3. Technical Background **4**
3.1. 3D's Max and Houdini 4
3.2. Pipeline Development and Data Management 4

4. Plug-in Development with Max Script, Dot Net and C++ **7**

5. 3D's Max SDK and C++ **9**
5.1. Visual Studios and Plug-in Set Up 10
5.2. DLL Setup 11
5.3. Class Descriptors 13
5.4. Plug-in Classes 14
5.5. Nodes, Scene Graph and Interface 16
5.6. The Reference System 18
5.7. Broadcast Notification System 21
5.8. Objects and Meshes 22
5.9. Modifiers 24
5.10. Geometry Pipeline 27
5.11. Parameter Blocks and Maps 31
5.12. User Interface 33
5.13. Animation and Time 35
5.14. Controllers 36
5.15. Function Publishing 38
5.16. Useful Functions and Classes 41

6. Cache Plug-in Development **42**
6.1. GPD Library 43

7. Conclusion **43**

8. References **45**

9. Appendix A - Project Specification **46**

10. Appendix B - Parameter Blocks Template **47**

11. Appendix C - Plug-in Code **48**

1. Abstract

Modern day animation and games pipelines use lots of different software packages to facilitate in production. Many companies implement and build upon these software packages with custom plug-ins to improve workflow. The purpose of this thesis is to explore plug-in development with 3D's Max and it's SDK (Source Development Kit), by developing a plug-in which reads Houdini's native geometry cache format.

2. Introduction

3D's Max is one of Autodesk's flagship products and is used in feature films as well as the games industry. Max offers many different approaches to character rigging and is well known for its plug-ins such as Fume Fx, Krakato and Thinking Particles.

Max offers multiple methods for creating plug-ins such as Max Script, C++ and Dot Net. Max Script is an easy to use artist friendly scripting language which allows users easy access to all areas of Max, and it is also a good way to prototype plug-ins before taking them further. Although Max Script is user friendly it is very slow compared to C++ and Dot Net plug-ins. It is recommended that Max Script be used to develop plug-ins, if speed is not an issue, as it is much easier to work with.

In some cases speed is necessary when developing plug-ins and this is when plug-in developers turn to the SDK. However the SDK can be extremely difficult to use and does not have many learning resources available. This project will explore plug-in development with the Max SDK by creating a plug-in which will read geometry data from Houdini's native geometry cache format. The project is not about the plug-in and is instead about learning the Max SDK, the plug-in is simply a method of learning it. Finally some knowledge of 3D's Max and C++ is required to fully understand this thesis.

3. Technical Background

3.1. 3D's Max and Houdini

Autodesk 3Ds Max is a powerful 3D modelling, animation and rendering software package. It is used primarily in the film, games, and architectural industries and is a popular among many artists.

One of the major philosophies behind Max's workflow is artist friendly workflow and this can be seen in all areas of Max. For instance Particle Flow is an easy to use event driven particle editor, biped and CAT (Character Animation Toolkit) makes easy to set up characters, Max Script is a very simple and easy to learn scripting language and Max is also a very popular modelling tool among artists.

Max is built around a plug-in architecture which allows for third party developers to extend its abilities. Plug-ins are created as DLL files which are read by Max every time a new instance of it is opened. Because of its plug-in architecture, Max has multiple plug-ins which is the reason some studios and artists use it. For instance Blur Studios use Max in combination with Fume FX, Alan Mackay, who is a visual effects artist, also uses Max with Fume FX. Many effects for feature films such as 2012, Sucker Punch and many more have been developed using Max and plug-ins such as Fume FX and Thinking Particles.

To help learn more about plug-in development with the Max SDK a plug-in will be created. The plug-in will basically read a geometry cache file from disk every time the frame changes in Max. The file being read is Houdini's native cache file format which is a software package developed by Side Effects Software.

Houdini is built upon the idea of procedural workflows which makes it a very powerful software package, particularly for visual effects. It is used more commonly by large animation and visual effects studios on feature films, but it has recently been used in games studios as well. Houdini was developed to follow the same procedural workflow as its predecessor Prism which was released in 1987 by Side Effects.

Due to its procedural nature, Houdini has an entirely node based environment for users to work in. This gives artist and TD's a non destructive workflow, as nodes are combined to build a network which can then connect to other networks. This is unlike other software packages which delete or don't manage their construction history. Another advantage of using a node based workflow is that artist can build tools and export them for other to use, without writing a single line of code.

3.2. Pipeline Development and Data Management

Many different studios use Houdini in their pipeline, primarily in visual effects, and combine it with other software packages such as 3D's Max or Maya. For example Digital

Domain uses Houdini for visual effects and Maya for character animation. Studios which use this type of pipeline will have developed tools which allow easy data transfer between different software packages.

Most of the time when transferring data between different software packages, only geometry data would need transferring. An example of this would be a character which has been animated in one application, such as Max, and then cached out to disk for another application, such as Houdini, to do the cloth or hair simulation. Although this is an extremely powerful pipeline that gives artists the freedom to utilize more tools, it can be problematic. For example both applications have to be able to read the file format and have to read it quickly as new geometry data is being bought into the application at every frame. This can't be done with scripting as it is simply not fast enough.

Max does have some tools which cache geometry to disk, but these tools are limited, are not as powerful as Houdini's cache format and nor do these tools allow any kind of data transfer to Houdini. One of the ways Max caches geometry is with a point cache modifier, though this only works on objects which geometry does not change dynamically over time as it only saves vertex data in the file. For instance you could use point cache on a character that has been rigged and animated as the number of vertices and faces is always going to be the same at every frame, no new vertex or face is being added to the geometry over time. On the other hand it wouldn't work for a particle simulation as new points and faces are being added or deleted at every new frame. Particle Flow, which is Max's native particle editor, does have the ability to cache out dynamically changing geometry, but this only in Particle Flow.

There are geometry cache plug-ins available for Max such as Xmesh which is a plug-in developed by Thinkbox Software. Xmesh is a very powerful geometry cache system but is currently only available to Max. Another plug-in currently in development on most of Autodesk's products and many other software packages is Alembic. Alembic is an open source exchange format developed by Sony Image Works and ILM to send animation data between different scenes. It is capable of importing and exporting cached geometry data between different software packages, and it is already implemented in some applications such as Houdini and Maya. Alembic is a recent development and when it is fully integrated into more software applications will be an extremely powerful tool.

Houdini has the ability to cache geometry data out to multiple file formats but the main file format it uses is Bgeo and Geo. These are both Houdini's native cache file format, Bgeo is a binary file format and Geo is an ASCII (text) file format. Both do exactly the same thing but the binary format, Bgeo, is much faster and takes up less space on disk. As the Bgeo format is binary it can be written straight to memory unlike the ASCII format Geo which has to be interpreted by the computer.

The Bgeo and Geo formats are extremely powerful file formats that can be used to cache out more than just geometry data, and both formats are used in multiple areas in Houdini.

To work with the Bgeo or Geo formats, Side Effects Software has released the GPD C++ library and has made it open source. By doing so users can read and write to both file format as well as add custom data to the format.

4. Plug-in Development with Max Script, Dot Net and C++

Max Script was originally developed as a plug-in for Max in 1996 by John Wainwright. Autodesk realized the importance of Max Script and hired John Wainwright to further develop it. It wasn't till later releases of Max that Max Script was implemented fully in to Max.

Max Script, from the start was developed to be an artist friendly scripting language and its syntax is very similar to the programming language BASIC. Because of its simplicity Max Script doesn't follow certain rules which other languages have to. For instance it is case insensitive, variables don't need to be assigned a specific type, such as integer or float, and user interfaces can be created very easily.

Max Script's syntax is sometimes much simpler than other scripting languages such as Mel script (Maya Embedded Language). Although some technical directors and programmers may not prefer this, it does make it simpler for artists. Max Script comparison with other scripting languages:

Max Script Loop - for i in 1 to 10 do
Mel Loop - for($i=0; $i<10; ++$i)

Max Script String Array - array = #("first", "second", "third")
Mel String Array - string $array[3] = {"first\n", "second\n", "third\n"};

Max Script Creating sphere with radius of 10 - Sphere radius:10
Mel Creating sphere with radius of 10 - sphere -r 10;

Although it is extremely easy to develop plug-ins with Max Script these plug-ins are much slower than ones developed with the SDK. As well as this Max Script doesn't allow complete access to the operating system and is restricted to 3D's Max. To fix these issues developers can use Dot Net combined with a language such as C#. Dot Net plug-ins are very similar to C++ SDK plug-ins in the way that they require certain classes to allow Max to read them and the plug-in is compiled as a DLL (Dynamic Link Library) file. To create a Dot Net plug-in developers have to use the files MaxCustomControl.dll and Autodesk.max.dll. These files give developer's access to the Max SDK using Dot Net which means developing C++ and Dot Net plug-ins is very similar. However Dot Net plug-ins are still not as fast as C++ plug-ins but are easier to develop.

Other software packages such as Maya, Houdini and Softimage also have built in scripting languages, as well as an API open to other languages such as Python. Python is a very popular programming language in the computer graphics industry and is used in most if not all major 3D software packages. Autodesk have not implemented Python into Max but Blur Studios did release a plug-in which allows developers to combine Max Script with Python.

Overall scripting is much easier and reliable than creating C++ plug-ins in any 3D software package. However C++ plug-ins will run much faster than scripted plug-ins. The purpose of this thesis is to only focus on plug-in development with C++ and the Max SDK. It will not go into detail with Max Script or Dot Net as these are large subjects within themselves.

5. 3D's Max SDK and C++

The Max SDK is a very large and complicated set of C++ classes and libraries used to develop plug-ins for 3D's Max. Because Max is built around a plug-in architecture, a lot of its core functions are available through the SDK. This means third party developers have lots of freedom to further build upon in it. However this does not mean the SDK is easy to work with as some of its class can be extremely large, and were written over fifteen years ago.

To develop plug-ins with the SDK developers must use the same IDE (Interactive Development Environment) that was used to develop Max by Autodesk. This means that with every new release of Max the plug-in has to be recompiled for that version. For Max 2012, Autodesk recommends using Visual Studios 2008 with service Pack 1. However Visual Studios 2010 also seems to work fine.

Max plug-ins are compiled as DLL files which Max reads from a plug-in folder on start up. These DLL files must have a specific extension depending on the type of plug-in being created. Many different types of plug-ins can be created with the SDK such as:

- **Extension - Plug-in Type**
- BMI - Bitmap Manager IO DLLs.
- BMF - Bitmap Manager Filter plug-ins.
- BMS - Bitmap Manager Storage DLLs.
- DLB - Shader plug-ins.
- DLC - Controllers.
- DLE - Scene Export plug-ins.
- DLF - Font Loaders.
- DLH - Sampler plug-ins
- DLI - Scene Import plug-ins.
- DLK - Filter Kernels (Anti-aliasing filters)
- DLM - Modifiers.
- DLO - Procedural Objects.
- DLR - Renderers.
- DLS - Object Snap plug-ins.
- DLT - Materials and Textures.
- DLU - Utility plug-ins.
- DLV - Rendering Effects
- DLX - Max Script extension plug-ins.
- FLT - Image Filter plug-ins.
- GUP - Global Utility plug-ins.

5.1. Visual Studios and Plug-in Set Up

Developers have multiple methods when beginning to develop plug-ins such as using an app wizard, using existing plug-in samples and starting from scratch. The app wizard is basically a Visual Studios plug-in which automatically sets up a Max plug-in project, but it can be difficult to get it to work. The SDK also comes with a library of existing plug-ins which developers can use as a starting point for creating a plug-in by copying only the code they need. Finally developers can build plug-ins from scratch which can be difficult but gives a better understanding of the SDK and the plug-in being developed.

When starting to develop a plug-in from scratch certain items must be implemented in the code, and certain setting must be set properly so the plug-in compiles. After a new empty Win32 project has been created in Visual Studios, the following settings need to be changed:

First the SDK include files need to be specified in the Additional Include Directories and can be found in the SDK folder.

Fig 1. Visual Studios project properties under C/C++ tab then General tab.

Secondly the library files directories also need to be added to the project under Additional Library Directories and can be found in the SDK folder.

Fig 2. Visual Studios project properties under Linker tab then General tab.

The third step is to specify which library files need to be used in the project under Additional Dependencies. The files added depend on the project, for example if the project utilizes parameter blocks then the paramblk2.lib is need.

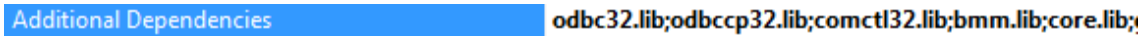

Fig 3. Visual Studios project properties under Linker tab then Input tab.

After this a output directory, target name and extension need to be set which can all be found under the general tab in the Property Pages window. The Target Extension depends on the type of plug-in.

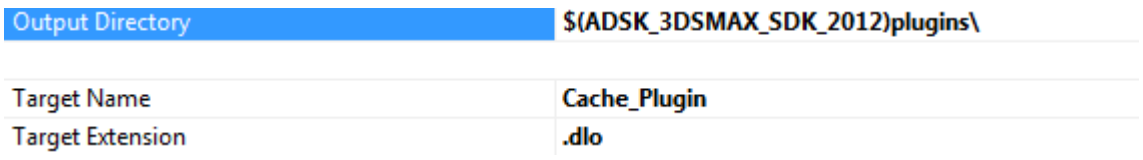

Fig 4. Visual Studios project properties under Configuration Properties tab then General tab.

Finally a DEF file needs to be created and add to the project which will be explained later.

Fig 5. Visual Studios project properties under Linker tab then Input tab.

The easiest way to set up directories is to create a environment variable in windows which can be used to easily get the SDK directory. This should already be created when installing the SDK and is usually named ADSK_3DSMAX_SDK_2012. Other settings may also need changing depending on the plug-in.

5.2. DLL Setup

To start programming a DLL plug-in file, developers must implement a DLL main function which is the entry point of the DLL file. This function doesn't need to do anything but hold the DLL HINSTANCE and call the DisableThreadLibraryCalls() function. After this function, come DLL export functions which are specific functions that are exported to Max along with a DEF file (Module Definition File). A DEF file has two main sections, which are LIBRARY and EXPORTS. The first section, LIBRARY, is the name of the DLL and the second section, EXPORTS, is a list of all of the functions being exported. The DEF file tells Max what functions are being exported and the name of the plug-in. Below is an example of a DEF file.

```
LIBRARY MyPlugIn.dll
EXPORTS
  LibDescription   @1
  LibNumberClasses @2
  LibClassDesc     @3
  LibVersion       @4
SECTIONS
.data READ WRITE
```

Every Max plug-in must implement six DLL export functions for it to be recognized by Max. These functions are LibDescription, LibNumberClasses, LibClassDesc, LibVersion, LibInitialize and LibShutdown. The functions are explained below and the last two are not absolutely necessary for your plug-in to work, but it is recommend that they be implemented.

LibDescription() – This function describe the plug-in and returns a string. The string is returned if the user does not have the plug-in and is working on a Max file which used it.
LibNumberClasses() – This returns the number of classes the plug-in uses and is usually set to return one.
LibClassDesc() – This function returns ClassDesc2 which is the class descriptor and will be explained later.

LibVersion() – This returns the version of Max the plug-in was compiled with and it usually just returns the function Get3DSMAXVersion(). Max can use this function to identify older and obsolete plug-ins.
LibInitialize() – This function returns true if the plug-in was loaded correctly by Max. If the plug-in does not load correctly it returns a message and calls FreeLibrary() on the DLL.
LibShutdown() – This is called when the plug-in is about to be unloaded and its return value is ignored by the system.

```
extern ClassDesc2* GetClassDescInstance();

HINSTANCE hInstance;

// This function is called by Windows when the DLL is loaded.  This
// function may also be called many times during time critical operations
// like rendering. Therefore developers need to be careful what they
// do inside this function.
BOOL WINAPI DllMain(HINSTANCE hinstDLL,ULONG fdwReason, LPVOID)
{
    if( fdwReason == DLL_PROCESS_ATTACH )
    {
        hInstance = hinstDLL;
        DisableThreadLibraryCalls(hInstance);
    }
    return(TRUE);
}

// This function returns the number of plug-in classes this DLL.
__declspec( dllexport ) int LibNumberClasses()
{
    return 1;
}

// This function returns a pre-defined constant indicating the version of
// the system under which it was compiled.
__declspec( dllexport ) ULONG LibVersion()
{
    return VERSION_3DSMAX;
}

// This function returns a string that describes the DLL and where the user
// could purchase the DLL if they don't have it.
__declspec( dllexport ) const TCHAR* LibDescription()
{
    return _T("Cache Plugin");
}

// This function returns the number of plug-in classes this DLL
__declspec( dllexport ) ClassDesc* LibClassDesc(int i)
{
    // Returns a class descriptor
    return GetClassDescInstance();
}
```

Fig 6. Plug-in DLL main and export functions.

5.3. Class Descriptors

The class descriptor is a class derived from ClassDesc2 which Max uses to gather information about the plug-in, and it also allows Max to create a new instance of the plug-in. ClassDesc2 is derived from ClassDesc and has eight virtual functions which must be implemented. As well as this ClassDesc2 is called by the DLL export function LibClassDesc and there can be more than one ClassDesc2 class in a project. Max first calls the DLL export function LibNumberClasses and uses this to get each class descriptor by calling LibClassDesc. The functions the class descriptor must implement are:

IsPublic() - This function indicates if the plug-in should be accessible to user through the user interface. It returns a boolean value and is most of the time set to true.

Create(BOOL loading = FALSE) - This function basically allows Max to create a new instance of the plug-in by returning a pointer to a new instance of the plug-in class.

ClassName() - This function returns the name of the class in the user interface.

SClass_ID SuperClassID()- This function returns a system defined constant which tells Max the type of plug-in being created. For instance an object plug-in would return GEOMOBJECT_CLASS_ID. A list of these super class ID's can be found in the SDK in plugapi.h file.

Class_ID ClassID() - This function returns a unique ID for the plug-in generated using the program Gencid.exe, which is provided with the SDK.

Category() - This function tells Max where the plug-in should be placed in the user interface and can create a new area for the plug-in. For instance an object in Max such as a box can be created from the standard panel in the create panel. Using this function an object can create a new user interface area for the plug-in in the create panel.

GetInternalName() - This function returns a fixed parsable string for the plug-in name used by Maxscript.

HINSTANCE HInstance() - This returns the DLL instance.

```
//--------------------------------------------------------------------------
//Class descriptor implementation

#define CacheObject_CLASS_ID Class_ID(0x218f77ba, 0x25666b07) //Uniqe ID for plug-in descriptor.

class CachePluginClassDesc : public ClassDesc2
{
public:
    //ClassDesc2 overrides

    //Plugin is accessable through the user interface.
    int     IsPublic()                      { return TRUE; }
    //Return the plug-in class as a pointer so Max can get a new instance of it.
    void*   Create(BOOL loading = FALSE)    { return new CacheObject();}
    //The plug-ins name in the user interface.
    const   MCHAR* ClassName()              { return _M("BGEO/GEO"); }
    //This plug-in is geometric object plug-in which is an object that can be created in the scene by users.
            SClass_ID SuperClassID()        { return GEOMOBJECT_CLASS_ID; }
    //Return a uniqe class ID for this plug-in.
            Class_ID ClassID()              { return CacheObject_CLASS_ID; }
    //Make an area in the create panel of Max for this plug-in.
    const   MCHAR* Category()               { return _M("Max BGEO/GEO"); }
    //Returns fixed parsable name.
    const   MCHAR* GetInternalName()        { return _M("Max BGEO/GEO"); }
    //Return the DLL hinstance.
            HINSTANCE HInstance()           { return hInstance; }
};

//Returns a singleton instance of the class descriptor.
static CachePluginClassDesc CacheDesc;
ClassDesc2* GetClassDescInstance() { return &CacheDesc; }
```

Fig 7. Plug-in class descriptor.

Gencid.exe is a program which creates a unique Class_ID which consists of two 32bit quantities. The purpose of these unique ID's is to makes sure plug-ins don't have any conflict with existing plug-ins. A Class_ID looks like, Class_ID(0x218f77ba, 0x25666b07).

5.4. Plug-in Classes

The last few pages described how to set up a plug-in and implement the main classes and functions to get it working. Now we will look at the actual plug-in code and the class the plug-in will derive from.

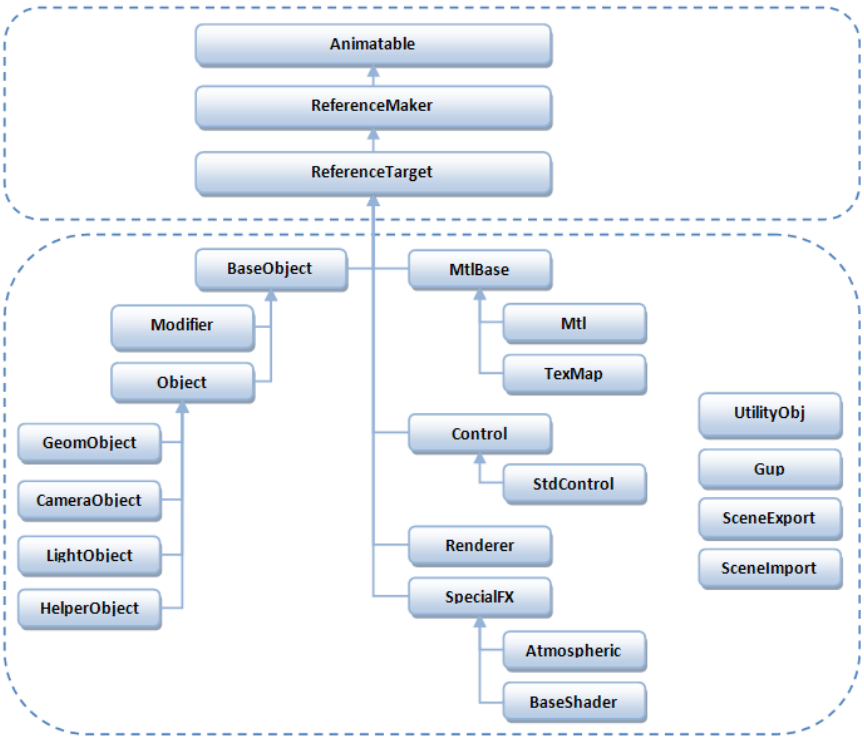

Fig 8. Diagram of the SDK class hierarchy.

Almost all classes in the SDK inherit from the main Animatable class and most plug-ins have this as the base class. After this comes the ReferenceMaker and ReferenceTarget classes which make up the reference system. Finally most of the classes at the bottom of the hierarchy can be derived from to create a plug-in.

Every plug-in must derive from a class in SDK depending on what type of plug-in is being developed. For instance a modifier plug-in would derive from the Modifier class and an object plug-in could derive from the GeomObject class. This derived class is the same class passed to ClassDesc2, specifically to the Create function. Below is some code from an object plug-in.

```
#define Widget_CLASS_ID Class_ID(0x986df9b4, 0x792ce6d7)

class Widget : public SimpleObject2 {
    public:
        //Plug-in functions go here.....................

        //From Animatable
        Class_ID ClassID() {return Widget_CLASS_ID;} //PLug-in unique ID
        SClass_ID SuperClassID() { return GEOMOBJECT_CLASS_ID; } //Plug-in type

        //Constructor/Destructor
        Widget();
        ~Widget();
};
```

Fig 9. Object plug-in class.

One of the most basic forms of plug-in that can be created in the SDK is a utility plug-in.

These plug-ins are not saved with the scene and only have one instance loaded at any given time. A utility plug-in derives from the class UtilityObj and has five main functions which developers can use. A utility plug-in can be thought of as a separate program in Max which does something, like edit geometry, and is then closed when it's finished with.

```
class SampleUtil : public UtilityObj
{
public:

    //Constructor/Destructor
    SampleUtil();
    virtual ~SampleUtil();

    //plug-in functions
    virtual void DeleteThis() { }

    virtual void BeginEditParams(Interface *ip,IUtil *iu);
    virtual void EndEditParams(Interface *ip,IUtil *iu);

    virtual void Init(HWND hWnd);
    virtual void Destroy(HWND hWnd);

    // Singleton access
    static SampleUtil* GetInstance() {
        static SampleUtil theUtilitySample;
        return &theUtilitySample;
    }
};
```

Fig 10. Utility plug-in class.

Some more common plug-in types such as object and modifier plug-ins have a special simple class which plug-ins can derive from. These simple classes are much easier to work with than other classes as they don't have to implement as many functions. For instance objects plug-ins can be derive from the SimpleObject2 class instead of the GeomObject class. SimpleObject2 derives from the GeomObject class and implements a lot of functions to make easier than having to derive from GeomObject to create an object plug-in. These simple classes do have limitations as SimpleObject2 creates only triangulate objects but are much easier to work with.

5.5. Nodes, Scene Graph and Interface

Almost all plug-ins will have to edit, create or work with nodes in some way. Nodes are stored in the scene graph and hold information about scene objects, as well as their relationship between each other. For instance nodes can hold information such as the objects transform data, parent and child information, assigned material and a pointer to the objects geometry. Nodes are not real objects visible in the scene, but instead hold a pointer to the actual object in the scene as these objects don't hold any information about their position, rotation, scale, materials and relationships with other objects. Scene objects are the objects in the viewport that users can create and edit, nodes hold information about these object.

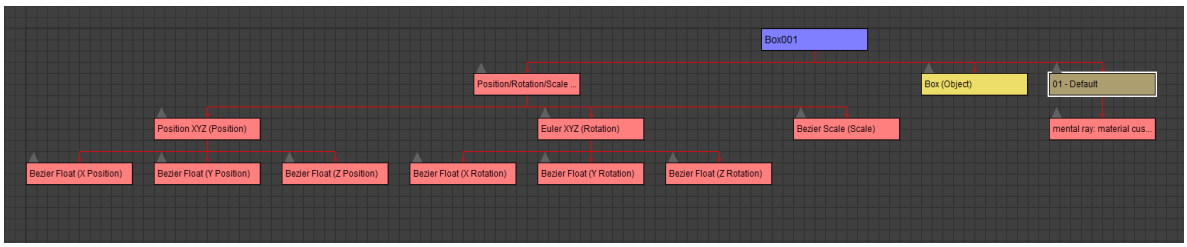

Fig 11. Scene graph nodes of a box object with material assigned to it.

Nodes in the scene graph can only link to one object but can have multiple objects linked to them. This means nodes follow a tree like data structure and loops are not allowed to be created when linking nodes. At the root of the tree is a virtual node which has no parent and this node is not visible in the scene graph. Finally objects in the viewport do not get displayed if they do not have a correspondent node in the scene graph. This means if a mesh object is created in the SDK, a node must be created for it as well.

There are many ways to access scene nodes, and once a node is accessed numerous things can be done to it. For instance the objects transform property can be changed and the object mesh can also be accessed and edited. To get a node a pointer to the node must be stored by either creating a new node or by accessing an existing node. The code below shows how to get or create a node and retrieve its data.

```
//Create a node and attach an object
Interface *ip = GetCOREInterface();
INode *np;
np = ip->CreateObjectNode(meshObject);

//Get selected node
Interface *ip = GetCOREInterface();
INode *np;
np = ip->GetSelNode(1);

//Access node data
Interface *ip = GetCOREInterface();
INode *np = ip->GetSelNode(1);

np->GetObjectRef();
np->GetChildNode(1);
```

CreateObjectNode(meshObject) – This function creates a new object node but must be passed a mesh.

GetSelNode(1) – This function gets the selected scene graph node and is passed an integer value. The integer value decides which node in the viewport selection array gets selected. For example if there are five objects selected in the viewport the second will be gotten as the array starts from zero.

GetObjectRef() – This function gets a pointer to the nodes object. This can then be used to edit the geometry.

GetChildNode(1) – This function gets the first child node of the node.

```
//Get the interface.
Interface* ip = GetCOREInterface();
//Create a new poly object.
PolyObject *polyOb = new PolyObject;
//Assign the poly object to an object.
Object *obj = polyOb;

//New mesh taken from the poly object.
MNMesh &mnmesh = polyOb->GetMesh();

//Create a new node using the object.
INode *node = ip->CreateObjectNode(obj);
```

Fig 12. How to create an object and apply it to a new node.

The examples above introduce some of the main classes in the SDK, Interface and INode. Interface gets a pointer to the Max interface and has lots of functions for accessing different areas of Max. In the examples above Interface is used to get the selected node and create an object node. INode is used to store a node and get its data. The Interface pointer can be initialized by using the function GetCoreInterface(), and this gives access to the interface.

```
//Get the interface.
Interface* ip = GetCOREInterface();

//Get the current viewport.
ip->GetActiveViewport();
//Get the first node in the selection array.
ip->GetSelNode(0);
//Delete a node.
ip->DeleteNode(0);
//Save the file.
ip->FileSave()
```

Fig 13. Interface class functions.

The interface class is an extremely useful when developing plug-ins as it gives access to a lot of functions. These functions can be used to get and set all kinds of data within Max. All plug-in developers should know of this class as it will be used in all plug-ins in some way.

5.6. The Reference System

When a plug-in wants to be aware of any changes in the scene, it can use the reference system. The reference system is a set of classes in the SDK and should not be confused with C++ references.

The reference system works by allowing plug-ins to create references to other scene entities and allows them to be reference makers and reference targets. This means that if a plug-in is making a reference to another scene object a message is sent from that object to the plug-in, as to notify it of any changes. As well as this a plug-in can be made a reference target, and can be notified if another scene entity is referencing it. An example of this would be a target camera object in Max, as the camera is dependent on the target to know which way to align its self. The camera is a reference maker to its target object and is notified of any changes to the target such as movement and when it is deleted. The messages sent from reference makers and targets are predefined in the SDK.

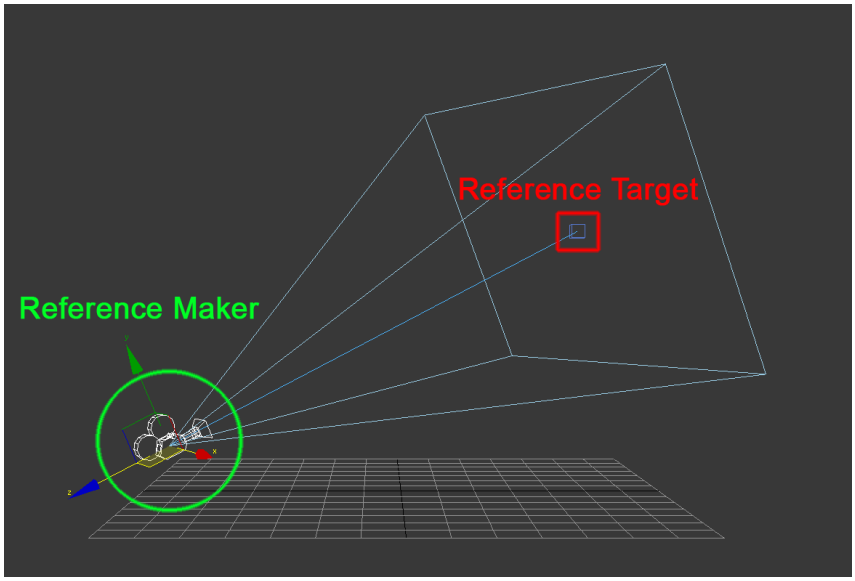

Fig 14. Interaction between a camera and its target object using the reference system.

Developers can use three kinds of references when creating plug-ins, strong, weak and indirect references. A strong reference is the most commonly used one and is found in most plug-ins. It is a reference to a scene entity which the plug-in owns. This means the plug-in cannot function without that reference and both the plug-in and its reference target lifetimes are connected. An example of this would be a target camera as it only exists in the scene if its target object does. As well as this, when we refer to or use the word reference we are always talking about strong references.

A weak reference is when the plug-in depends on reference target but can exist without it. This means the target can be deleted and the plug-in will still work. An example of this would be a look at constraints target object.

Indirect references can be used to stop loop dependences and allows plug-ins to reference scene entities that rely on the plug-in. An example of this would be the ring array system which uses indirect references to allow its master controller to interact with each node in the system.

To use the reference system a plug-in must derive from the ReferenceMaker class and must override some virtual functions. For a plug-in to reference another scene entity and become a reference maker, it would override the functions:

NumRefs() – This function tells Max how many references the plug-in holds, and a plug-in can have multiple references.

SetReference() – This function allows Max to set a reference to the scene entity and is not called directly.

GetReference() – Is used by Max to access the plug-ins references.

NotifyRefChanged() – This function notifies a reference maker when its target sends a message and allows it to process that message.

NotifyDependants() – This function does not need to be over written in the plug-in if it is a reference maker. It allows reference targets to send a message to its dependant reference makers of any change. For example if a reference target, such as a geometry object, changed its name, this function would notify all of its dependant scene entities. The entities notified can be the objects node and other dependant objects such as a plug-in which is using the geometry object as a reference target.

When a reference maker wants to add an entity as a reference target it should use the function ReferenceMaker::ReplaceReference(). ReplaceReference is used instead of calling ReferenceMaker::SetReference() directly and allows the changing of a reference at anytime.

All most all entities in Max can be reference targets and this includes plug-ins. Plug-ins can derive from the ReferenceTarget class to notify their observers. The main functions that must be overwritten are ReferenceTarget::NotifyDependants() and Clone(). NotifyDependants() is the same as above and Clone() is used when a user clones an object in Max.

When references are no longer needed they should be deleted, as they are stored in memory. As well as this references can cause errors if they are not managed properly. Functions such as ReferenceMaker::DeleteAllRefs() and ReferenceMaker::DeleteAllRefsFromMe() can be used to delete references.

The reference system uses predefined messages and the developer does not create their own messages. The types of messages the reference system would send are:

#define REFMSG_NODE_DISPLAY_PROP_CHANGED
#define REFMSG_NODE_NAMECHANGE
enum REF_SUCCEED

Most of the time a reference maker would just return REF_SUCCEED to notify the plug-in that it has received the message.

All the functions above can be used to create reference makers and targets. The code below is taken from a simple object plug-in and shows how to reference another scene object.

```
    //Refrence Maker
    //Number of references this plug-in holds.
    int NumRefs();
    //Get the set reference.
    ReferenceTarget *GetReference(int i);
private:
    //Set a reference to something.
    void SetReference(int i, ReferenceTarget* pTarget);
public:
    //Notify this plug-in when a reference changes.
    RefResult NotifyRefChanged(Interval changeInt, RefTargetHandle hTarget, PartID& partID, RefMessage message);
    //Manage references when cloning an object.
    RefTargetHandle Clone( RemapDir &remap );


//This plug-in has 2 references.
int CacheObject::NumRefs()
{
    return 2;
}

//Get both references.
ReferenceTarget *CacheObject::GetReference(int i)
{
    switch(i)
    {
        case 0:
            return (RefTargetHandle)pblock2;
            break;
        case 1:
            return (RefTargetHandle)RefTargetNP;
            break;
    }
}

//Set references to parameter block and a node.
void CacheObject::SetReference(int i, ReferenceTarget* pTarget)
{
    switch (i)
    {
        case 0:
            pblock2 = (IParamBlock2*)pTarget;
            break;
        case 1:
            RefTargetNP = dynamic_cast<INode*>(pTarget);
            break;
    }
}

//Notify the plug-in when the reference changes in some way.
RefResult CacheObject::NotifyRefChanged(Interval changeInt, RefTargetHandle hTarget, PartID& partID, RefMessage message)
{
    return REF_SUCCEED;
}
```

Fig 15. Object plug-in class using the reference system to reference another scene object.

5.7. Broadcast Notification System

The Broadcast Notification System can be used to notify a plug-in when a certain event happens and can be an alternative to the reference system at times. This system uses

callbacks to notify the plug-in of any changes such as time change and when a node is deleted.

The callback system can be easier to use than the reference system at times and there are three main parts to it. First a function needs to be made to process the notification, then another function is used to register a callback and finally a similar function is used to unregister the callback. The code below shows how a callback is used to notify the plug-in when an object is deleted.

```
void NodeDeleteNotify(void *param, NotifyInfo *info)
{

}

//Plug-in constructor.
CacheObject::CacheObject()
{
    RegisterNotification(NodeDeleteNotify, this, NOTIFY_SCENE_PRE_DELETED_NODE);
}

//Plug-in deconstructor.
CacheObject::~CacheObject()
{
    UnRegisterNotification(NodeDeleteNotify, this, NOTIFY_SCENE_PRE_DELETED_NODE);
}
```

Fig 16. Node delete callback.

The callback system can be extremely useful, especially when values needs to be changed during animation.

5.8. Objects and Meshes

Behind any 3d application is a geometry pipeline which manages how geometric objects are created and edited. Max has a robust and powerful geometry pipeline which can be used to create all kinds of object such as polygon, shape(spline), nurbs and tri objects.

Geometric objects derive from the GeomObject class which in turn derives from the Object class. The Object class is used to create and store the actual scene object which the scene node holds a pointer to, and this object can be of many different types. For instance the object can be a geometric, camera, light and helper object type.

Fig 17. Different nodes in Max with their corresponding object node.

Geometric objects can convert themselves into different object types when Max or a plug-in requires them to be a specific type. For instance a poly object can be converted into a shape object and all objects must be able to convert themselves into triangulated objects. So this means that the GeomObject class has a set of derived classes which are these different geometric object types.

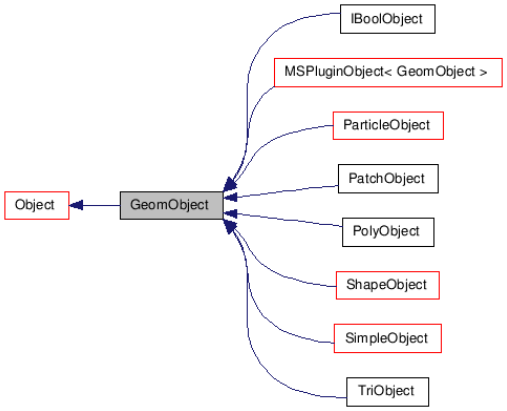

Fig 18. GeomObject class and its derived classes.

The Object class can be passed any of the geometric object types to create a scene object. We will only be focusing on polygon and triangulated objects as these are the most common types used.

Although the TriObject and PolyObject classes both hold geometric objects, they are not used to build the mesh of that object. Instead the SDK has two other classes which can be used by developers to build the mesh depending on the type. For instance a developer can use the Mesh class to build mesh for a TriObject, and the MNMesh class to build mesh for a PolyObject. The mesh can then be applied to the object using a function.

The Mesh class is only for triangulated objects and has many functions which make it easy to develop a mesh. It is also faster than its equivalent MNMesh class but is limited to triangulated mesh. MNMesh class is not limited to triangulated mesh and can be used to build any kind of polygon, however is much slower. It is recommended that developers use TriObject and Mesh classes whenever a polygon objects is not needed, because of speed issues. However the Mesh class can hide edges making it seem as if the user is working with polygons.

```
//Evaluate the current node and get its object state at the end of the geometery pipline.
Object *obj = node->EvalWorldState(curTime).obj;

//Check if the object can be converted into a poly object.
if (obj->CanConvertToType(Class_ID(POLYOBJ_CLASS_ID, 0)))
{
    //If true convert it into a poly object.
    PolyObject *polyConvert = (PolyObject *)obj->ConvertToType(0, Class_ID(POLYOBJ_CLASS_ID, 0));

    //Get the mesh of the object.
    MNMesh *mesh = &polyConvert->GetMesh();
```

Fig 19. Object being converted into a PolyObject.

Once objects have been built and have been attached to a corresponding scene node, they can be displayed in the viewport in many different ways. Most of the time a plug-in would implement and override the BaseObject::Display() function from the BaseObject class. However a forced redraw of the viewport can also be made using functions like Interface::RedrawViews() from the Interface class.

As a Max object flows through the geometry pipeline it uses object states to represent the data. The object state contains a material index, matrix, some flags for channels and a pointer to the object. This object state is what is sent to end of the geometry pipeline and is used to display, hit test and render the object. A plug-in can get an object state at any point in the pipeline using the function Object::Eval(). As well as this an object state is a C++ class called ObjectState and derives from the MaxHeapOperators class, which is a class above Animateable in the class hierarchy. Nothing really needs to be done with the ObjectState class apart from using it to store an object state and using its functions. The cache system within Max holds the object depending on how long it is valid, and the system stores objects as an object state along with its validity interval.

5.9. Modifiers

Modifiers are used by Max to edit objects geometry whilst keeping its original object data intact. They can be used to edit geometry procedurally with an algorithm or allow the user to edit geometry. As well as this modifiers can be layered on top of each other to change the final outcome of the object.

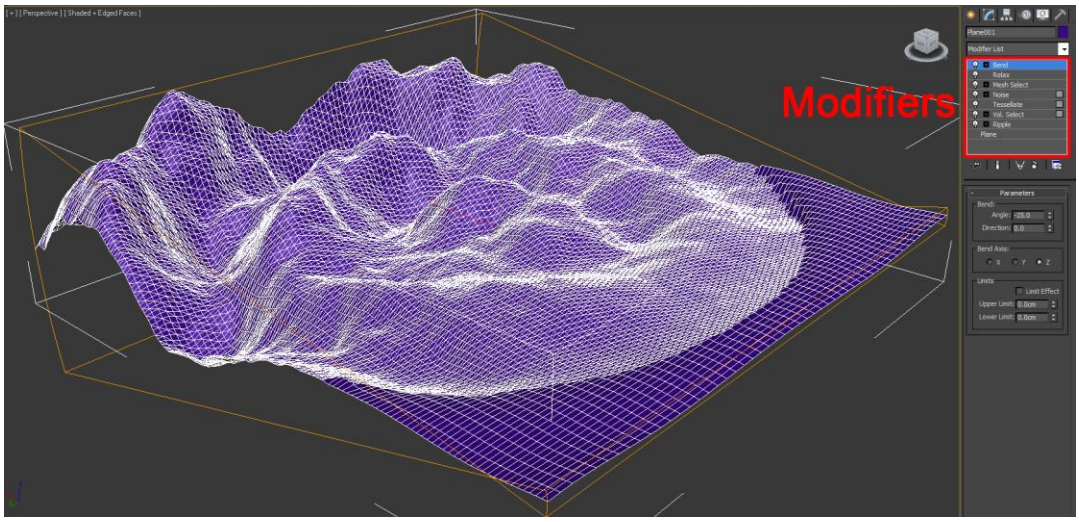

Fig 20. Modifiers used to procedurally edit a simple grid object to create part of an environment.

There are three different types of modifiers, object space, world space and edit modifiers. Objects space modifiers are the most common type and basically edit the object in its own local transform space. World space modifiers edit the object in world space and are applied after the object is transformed into world space. This means in the modifier stack the object space modifiers are applied first then the world space ones. Edit modifiers allow the

user to edit the geometry and model out the object, as well as convert mesh into different geometry types, for example polygon to triangular mesh. This still keeps the original object intact as the modifier is applied after it, but the modifiers end result can be collapsed into a single object with no modifiers. Modifier plug-ins can be created by deriving from the Modifier class.

Different object types such as shapes, polygon and patch objects have modifiers specific to their type but most modifiers will work on more than one type. For instance the bend modifier is compatible with many different types because it converts all incoming objects into triangulated mesh, if the object can be converted.

As some modifiers have to be evaluated over time and modifiers can be stacked on top of each other, they have to be fast and efficient in the geometry pipeline. To do this modifiers use a channel system which allows them to only work with specific object data. Modifiers must define which channels they require and Max only sends a copy of the required channels to that modifier. So if a modifier wants to edit an objects vertex colour information then it would use the vert colour channel.

Modifiers themselves don't hold a pointer to the object, but instead access the object using a derived (not to be confused with C++) object and then modify it in some way. Derived objects are like containers for managing modifiers, they hold a pointer to the object and contain a ModApp. ModApp (Modifier Application) is a class not available in the SDK, but is simply a container for the ModContext and holds a reference to the modifier. The ModContext holds information about each instance of the modifier and contains data such as the space the modifier was applied in. The SDK uses the IDervidedObject class to access the modifier stack so developers can add, delete and work on the stack.

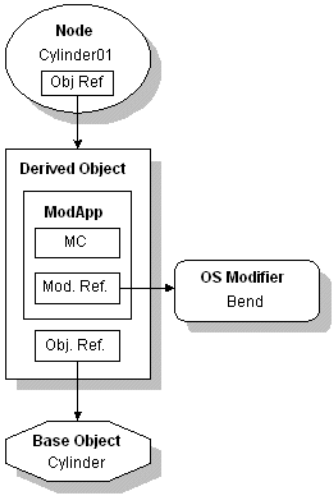

Fig 21. Diagram of a cylinder with a bend (object space) modifier attached.

When a modifier is added to an object in Max, the node creates a pointer to the derived object and no longer to the base object. The derived object holds the pointer to the base object and passes the object to the modifier and node.

```
class ADNModifier : public Modifier
{
public:
    //Constructor/Destructor
    ADNModifier();
    virtual ~ADNModifier();

    // -- From Animatable --
    virtual Class_ID ClassID() {return ADNModifier_CLASS_ID;}
    virtual SClass_ID SuperClassID() { return OSM_CLASS_ID; }
    virtual void GetClassName(TSTR& s) {s = GetString(IDS_CLASS_NAME);}

    virtual void DeleteThis() { delete this; }

    virtual int NumSubs() { return 1; }
    virtual TSTR SubAnimName(int i) { return GetString(IDS_PARAMS); }
    virtual Animatable* SubAnim(int i) { return pblock; }

    // return number of ParamBlocks in this instance
    virtual int NumParamBlocks() { return 1; }
    // return i'th ParamBlock
    virtual IParamBlock2* GetParamBlock(int i) { return pblock; }
    // return id'd ParamBlock
    virtual IParamBlock2* GetParamBlockByID(BlockID id) {
        return (pblock->ID() == id) ? pblock : NULL;
    }

    // -- From Reference Maker --
    virtual RefTargetHandle Clone( RemapDir &remap );
    virtual RefResult NotifyRefChanged(Interval changeInt, RefTargetHandle hTarget,
        PartID& partID,  RefMessage message);

    // TODO: Maintain the number or references here
    virtual int NumRefs() { return 1; }
    virtual RefTargetHandle GetReference(int i) { return pblock; }
    virtual void SetReference(int i, RefTargetHandle rtarg) { pblock=(IParamBlock2*)rtarg; }

    // -- From BaseObject --
    virtual TCHAR *GetObjectName() { return GetString(IDS_CLASS_NAME); }
    virtual CreateMouseCallBack* GetCreateMouseCallBack() {return NULL;}

    virtual void BeginEditParams(IObjParam *ip, ULONG flags,Animatable *prev);
    virtual void EndEditParams(IObjParam *ip, ULONG flags,Animatable *next);

    // -- From Modifier
    virtual ChannelMask ChannelsUsed()  { return GEOM_CHANNEL|TOPO_CHANNEL; }
    virtual ChannelMask ChannelsChanged() { return GEOM_CHANNEL; }
    virtual BOOL ChangeTopology() {return FALSE;}

    virtual Interval LocalValidity(TimeValue t);
    Class_ID InputType() {return defObjectClassID;}

    virtual void ModifyObject(TimeValue t, ModContext &mc, ObjectState *os, INode *node);

private:
    // Parameter block
    IParamBlock2    *pblock;    //ref 0
    static IObjParam *ip;           //Access to the interface

};
```

Fig 22. Modifier plug-in derived from the Modifier class. The modify object plug-in does most of the work as it holds the code which actually modifies the object. The channel functions also define which channels to use.

5.10. Geometry Pipeline

Now that we know how scene objects are made, edited and how modifiers work, we take a more in depth look at the geometry pipeline. This chapter will look into how nodes reference objects and how objects move through the pipeline and end up on screen.

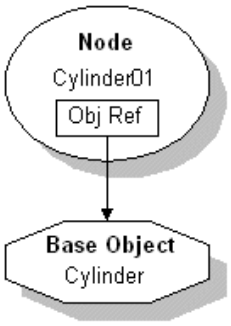

Fig 23. Cylinder object pipeline.

Fig 23 shows a simple cylinder in the geometry pipeline. Its scene node represents its transform and has a pointer to its base object which is a cylinder. The arrow shows the reference between each node but objects travel from bottom up in the geometry pipeline. This means the first object to enter the pipeline is the base object node. This is the most basic form of relationship any object in Max can have.

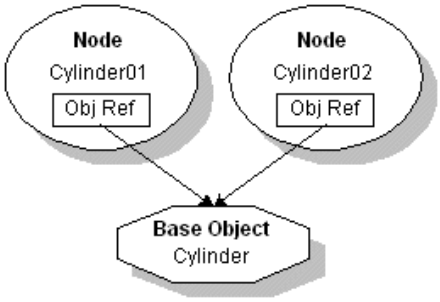

Fig 24. Two cylinders instancing one object.

The geometry pipeline in Fig 24 shows two scene node referencing one object and this relationship is called an instance in Max. This means two scene nodes exist and have separate transform data but both use the same base object. This can be useful when modelling something like car wheels as they would transform separates but both models would be the same. However both nodes will also reference the same derived object if a modifier is created on either objects. This means modifiers will also be instanced to both nodes.

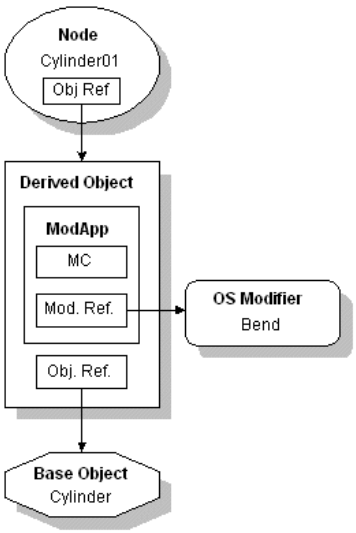

Fig 25. Cylinder with bend modifier.

Fig 25 is an example of a modifier in the geometry pipeline. In this pipeline the base object is referenced by the derived object, this applies the modifier and the scene node references the derived object. Whenever a new modifier is added to an object a derived object is automatically created.

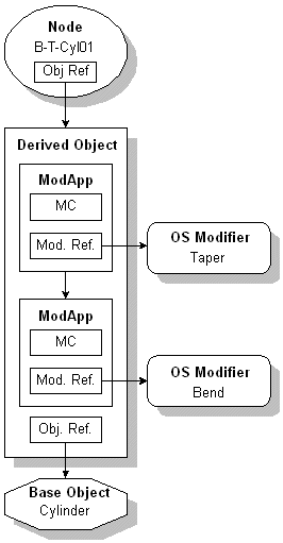

Fig 26. Cylinder object with two modifiers.

Fig 26 is exactly the same pipeline as Fig 25 but with two modifiers. The object is passed through the pipeline in the same way and two ModApps are used to hold information on each modifier. The top modifier in modifier stack would be the taper one which means the bend modifier would affect the geometry then the taper.

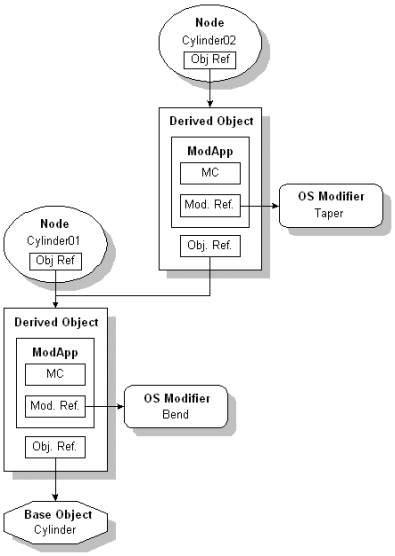

Fig 27. Two cylinder nodes referencing the same base object but with separate derived objects.

Fig 27 is called a reference relationship in Max and lets objects instance the same base objects. However the second cylinder doesn't reference the base object, but instead references the first cylinders derived object. This means that once the cylinder01 has finished modifying its object cylinder02 uses that object as its base object. However cylinder01 must finish applying all its modifiers before cylinder02 uses it as its object and cylinder02 cannot choose at what point in the modifier stack to use cylinder01 as a base object.

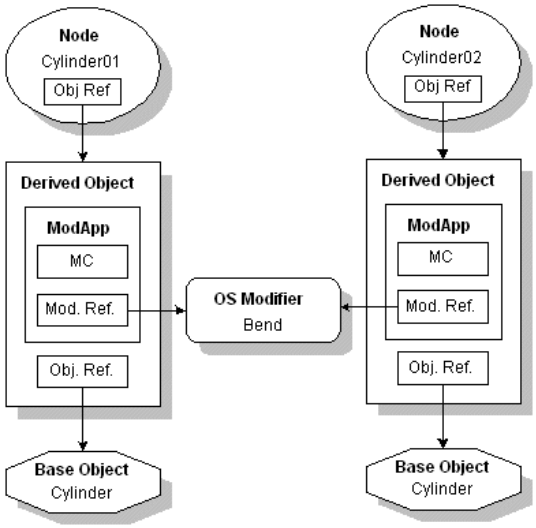

Fig 28. Two objects sharing one modifier.

Fig 28 Geometry pipeline shows two objects sharing one modifier and the modifier would edit both objects in the same way, as the parameters would be the same for both objects. So the bend modifier would bend both objects equally. However this does depend how the modifier is applied in Max as the ModContext inside the ModApp contains the bounding box data for each object. This decides if the modifier is applied locally to both objects or independently to each object. Fig 29 shows this.

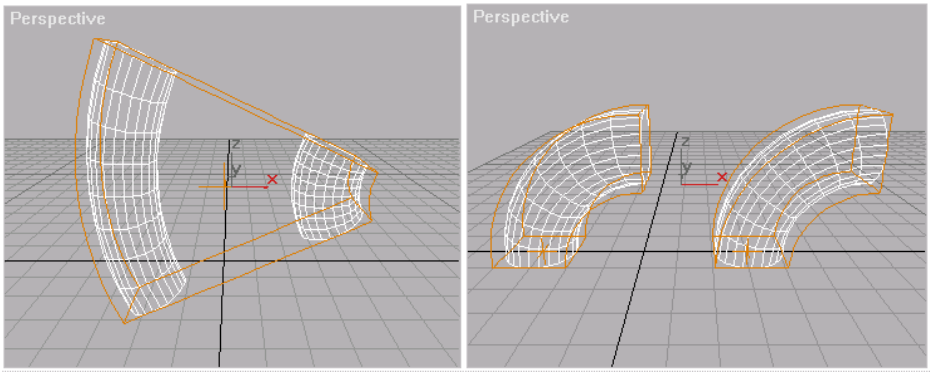

Fig 29. The left image has one modifier affecting both geometries locally around their common centre. The right image show the same modifier affecting both geometries independently using their own centre.

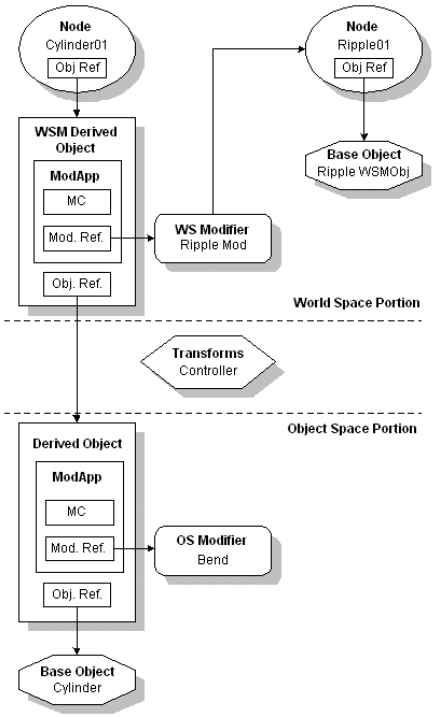

Fig 30. Shows a cylinder with a world space modifier (WSM) appended to it.

The final geometry pipeline show an object space and world space modifier attached to a cylinder object. A world space modifier is empty which means it has no user interface or parameters. Instead it makes a link to a deformer object and the modifier is created or deleted when the deformer is linked or unlinked. When a world space modifier is applied to an object it is first modified by its object space modifiers and then transformed into world space where it is modified by the world space modifier and sent to the node. The world space modifier references the deformer object.

5.11. Parameter Blocks and Maps

Parameter blocks are a way of holding values in Max and these values can be mapped to a user interface. An example of a parameter would be the sphere radius value, when creating a sphere object. The sphere radius parameter value has a corresponding user interface and can be animated. Almost all plug-ins utilize parameter blocks apart from utility plug-ins and maxscript plug-ins.

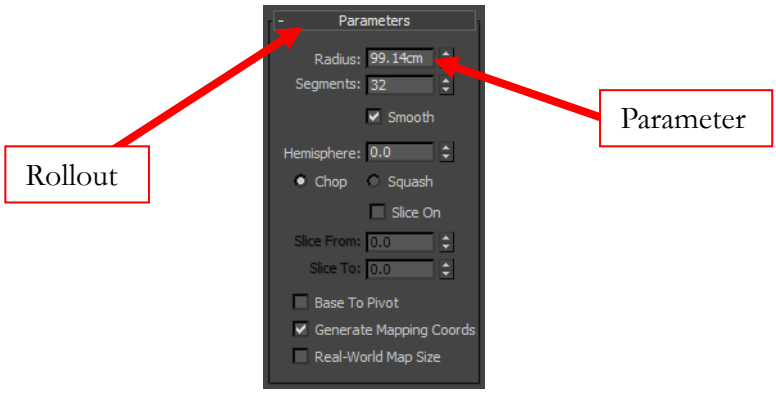

Fig 31. Sphere objects parameters and rollout.

Parameters values can be many different Max types such as float, integers, colour, node and many more. As well as this parameters that can be animated must have a matching controller which will be explained later.

The SDK uses the ParamBlockDesc2 class to create parameters. This class is a parameter block descriptor and holds all of the arguments for creating a parameter block. To create parameter blocks using ParamBlockDesc2 developers must create a static instance of the class and fill in all the required arguments in its constructor. As well as this the plug-ins constructor must contain the function ClassDesc2::MakeAutoParamBlocks(this) which tells Max to automatically set up and create the parameter block.

Parameter blocks use a parameter map to attach parameters to user interfaces. However ParamBlockDesc2 automatically manages and creates the parameter map, if the correct arguments have been set in its constructor. There are other and older methods which can be used to create parameter blocks and these require the set up of parameter maps. Other methods for creating parameter blocks involve a lot more work and use older and deprecated classes.

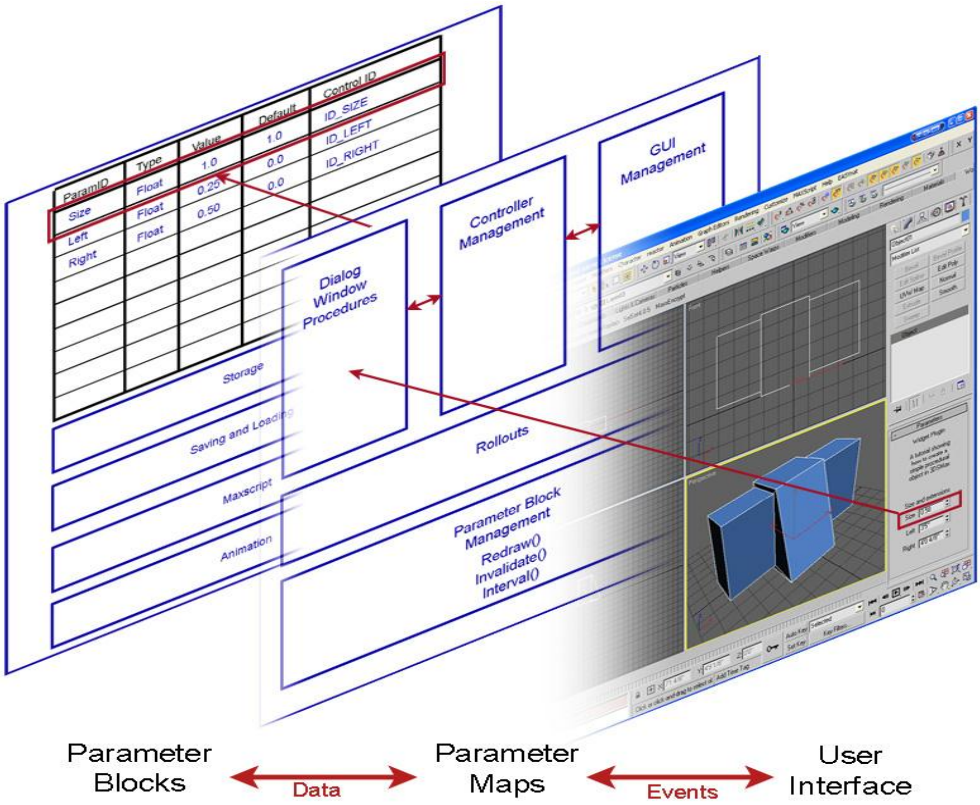

Fig 32. This diagram shows how a parameter map attaches to a parameter block.

Parameter blocks are reference targets which means they automatically manage saving, loading and deleting in Max. This means that a strong reference should be made to the parameter block using the reference system.

A plug-in can use two functions to get and set the values of a parameter at any time, GetValue() and SetValue(). These functions have many different overloaded functions.

```
float size, sizeLeft, sizeRight;
pblock2->GetValue(widget_size, t, size, ivalid );
pblock2->GetValue(widget_left, t, sizeLeft, ivalid );
pblock2->GetValue(widget_right, t, sizeRight, ivalid );
```

Fig 33. Functions for getting parameter values and applying those values to variables.

For a plug-in to be aware of when to show the interface to the user they must implement two functions, BeginEditParams() and EndEditParams(). For instance if a sphere is selected we can see its user interface in the modify panel, but if we deselect it or select a new object the interface disappears or changes. These functions are from the Animatable class and notify a plug-in when to display its user interface via rollouts. Rollouts are where the parameter blocks are stored in the user interface and there can be how ever many parameters in a rollout.

```
static ParamBlockDesc2 CacheObjectParam(
    //Reuired parameters.
    object_params,
    _T("Max BGEO/GEO"),
    0,
    &CacheDesc,

    //Flags
    P_AUTO_CONSTRUCT + P_AUTO_UI,

    //Dependent arguments --------------------------
    //required because P_AUTO_CONSTRUCT was flagged
    //This declares the reference number of the Parameter Block.
    PBLOCK_REF,

    //required because P_AUTO_UI was flagged.
    //This is the Rollout description
    IDD_PANEL, IDS_PARAMS, 0, 0, NULL,

    //Parameter Specifications ---------------------
    // For each control create a parameter:
    file_path,          _T("File Path"),        TYPE_STRING,        P_INVISIBLE,        IDS_FILEPATH,
        p_default,      "C:\\PATH\\FILE",
        p_ui,           TYPE_EDITBOX,           IDC_FILE_PATH,
        ParamTags::end,

    file_extension,     _T("File Extension"),   TYPE_STRING,        P_INVISIBLE,        IDS_FILE_EXTENSION,
        p_default,      ".bgeo or .geo",
        p_ui,           TYPE_EDITBOX,           IDC_FILE_EXTENSION,
        ParamTags::end,

    file_offset,        _T("File Offset"),      TYPE_INT,           P_ANIMATABLE,       IDS_FILE_OFFSET_SPIN,
        p_default,      0,
        p_range,        0,10000,
        p_ui,           TYPE_SPINNER,           EDITTYPE_INT,       IDC_FRAME_OFFSET_EDIT,      IDC_FRAME_OFFSET_SPIN, 1,
        ParamTags::end,
    ParamTags::end
);
```

Fig 34. Code used to develop a parameter block using ParamBlockDesc2 and a template can be found in Appendix B. The template explains each argument.

5.12. User Interface

Most user interfaces will be managed by the parameter blocks system but they still have to be created using Visual Studios dialog editor. Two items need to be created in Visual Studios to attach the interface to the parameter block, a resource script and resource header file. We will only look into user interface development for parameter blocks and rollouts as this is the most common kind of interface.

The resource header file is automatically created by Visual Studios when the rc (resource script) file is created. The rc file contains a visually created user interface and a string table. These are both created by the developer and the variables created in the header file are then used in the ParamBlockDesc2 constructer to attach the user interface to the parameter blocks.

```
#define IDS_LIBDESCRIPTION              1
#define IDS_CATEGORY                    2
#define IDS_CLASS_NAME                  3
#define IDS_PARAMS                      4
#define IDS_SPIN                        5
#define IDD_PANEL                       101
#define IDC_CLOSEBUTTON                 1000
#define IDC_DOSTUFF                     1000
#define IDC_COLOR                       1456
#define IDC_EDIT                        1490
#define IDC_SIZE_EDIT                   1490
#define IDC_LEFT_EDIT                   1491
#define IDC_RIGHT_EDIT                  1492
#define IDC_SPIN                        1496
#define IDC_SIZE_SPIN                   1496
#define IDC_LEFT_SPIN                   1497
#define IDC_SPIN3                       1498
#define IDC_RIGHT_SPIN                  1498

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        102
#define _APS_NEXT_COMMAND_VALUE         40001
#define _APS_NEXT_CONTROL_VALUE         1001
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Fig 35. Resource header file created by visual studios. The variables created are used by a parameter block to attach the user interface to it.

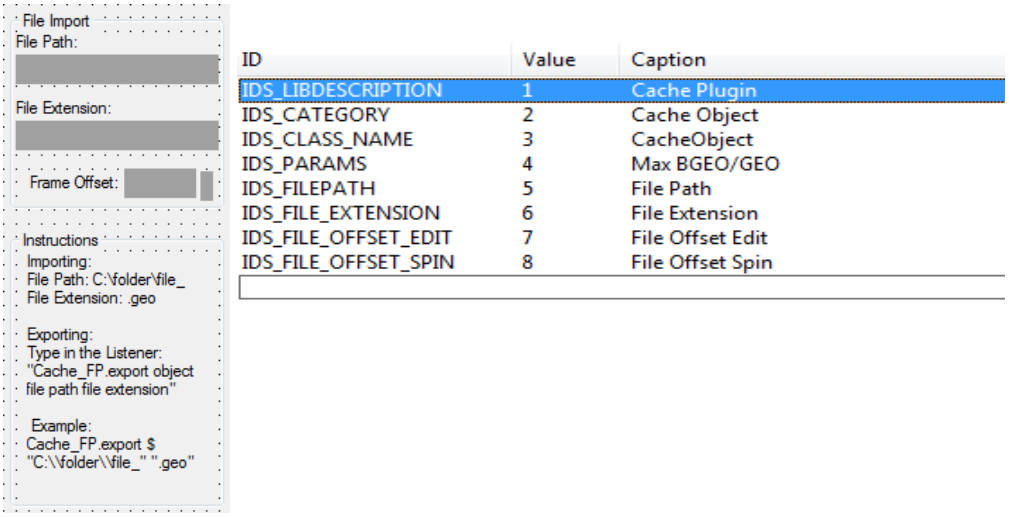

Fig 36. String table and dialog editor used to create user interface.

Some user interfaces items, such as buttons, are not managed by the parameter blocks system and must be managed manually. This is because they don't have a matching parameter map and thus Max has no way to use the interface. This means a parameter map must be created and is done so by creating a class which derives from the ParamMapUserDlgProc class.

```
class CacheObjectDlgProc : public ParamMap2UserDlgProc {
private:
    CacheObject *ob;
public:
    CacheObjectDlgProc(CacheObject *s) { ob = s; }
    INT_PTR DlgProc(TimeValue t, IParamMap2 *map, HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
    void DeleteThis() { delete this; }
};

INT_PTR CacheObjectDlgProc::DlgProc(TimeValue t, IParamMap2 *map, HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    //Implement cutsom ui
    return FALSE;
}
```

Fig 37. Code used to manage custom buttons.

5.13. Animation and Time

Max handles time in ticks and can convert from ticks into other time formats such as frames, seconds and minute. There are 4800 ticks per second in Max and the reason for this is that 4800 can be divided by the common fps (frames per second) formats, 24, 25 and 30 fps. In reality there are actually 10 million ticks per second.

The SDK handles time in ticks as well and contains a lot of functions for developers to get and set time values, but there are some functions that work with frames.

```
//Get the interface.
Interface* ip = GetCOREInterface();
//Number of ticks per frame in Max.
int ticksPF = GetTicksPerFrame();
//Get start frame in Max.
int startFrame = ip->GetAnimRange().Start()/ticksPF;
//Get end frame in Max.
int endFrame = ip->GetAnimRange().End()/ticksPF;

//Loop through the frames.
for(int f=startFrame; f<endFrame; f++)
{
    //Set the current time.
    ip->SetTime((f*ticksPF), 0);
    //Get the current time.
    int curTime = ip->GetTime();
```

Fig 38. Code used to work with time in Max.

Max uses object states to allow objects to flow through the geometry pipeline and to store them in the system cache. However it only stores objects depending on how long they are valid and uses an interval system to control their validity. The interval system is a class called Interval and it represents a period of time. This time period can be used to decide if an item is valid within the cache system. So when Max needs the result of a nodes pipeline it first checks the cache to see if its interval is valid and then uses the cache if it is found to be valid. However if the nodes cache is found to be invalid, Max re-evaluates its geometry pipeline and makes it valid. Once re-evaluated a new object state and validity interval is stored in the cache. This kind of interval is called a validity interval.

An example of a validity interval would be an object such as a sphere with its radius parameter animated so that it gets larger over time. The object is not valid as it is growing over time and is being revaluated at every frame. However only the objects geometry channel is not valid and its topology channel is always valid as its vertices are being moved and its topology is not changing. This means validity intervals can be set to only affect certain channels to speed things up.

The Animatable class is one of the most important classes in SDK and is at the top of the class hierarchy in the SDK. This class has lots of different functions when working with the track view and animation data. It is also responsible for managing objects sub-animatable objects.

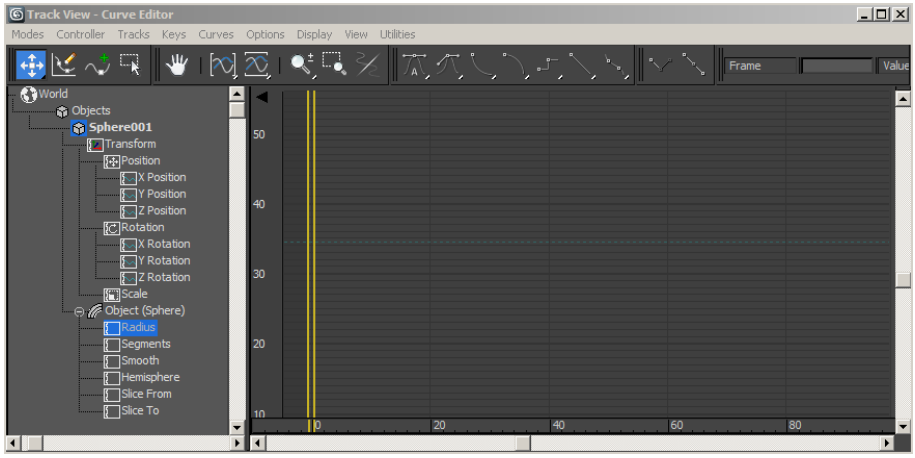

Fig 39. Track view and curve editor.

Sub-animatable objects are the objects below the actual object in the track view. Fig 39 has a sphere object and its sub-animatable objects are Transform and Object. The Object sub-animatable has a set of parameter as its sub-animatables such as Radius, Segments and Smooth. This is because parameters can be set to be animated on, and any sub-animatables which can be animated on also have an equivalent controller. A plug-in must implement some functions from the Animatable class to create sub-animatables. These functions are:

```
//Set number of sub animatables.
int NumSubs() { return 1; }

//Return each sub animatable.
Animatable *SubAnim(int i) { return pblock2; }
```

5.14. Controllers

Animation controllers provide a way of managing and creating animation data and do so by outputting a value at a specific time. Controllers can be procedural or keyframe based. Procedural controllers create values using an algorithm whilst with keyframe controllers the user creates values at specific frames. There are six types of controllers which return certain values:

- Point3 controllers (Point3)

- Position controllers (Matrix3)
- Rotation controllers (Quaternion)
- Scale controllers (ScaleValue)
- Transform controllers (Matrix3)

Every Max scene object has a default controller which is used to transform the object around the scene. This controller is connected to the scene objects node and controls the node position, rotation and scale. For most objects the default controller is the PRS (Position Rotation Scale) controller and this is a transform controller which uses a matrix. However this controller has sub controllers for position, rotation and scale which again have more sub controllers. The last sub controllers are Bezier controllers which accept float values and these are much easier to animate with than having to use a matrix value. This is the reason there are so many controllers and the values of the controllers can be seen in the curve editor. Controller plug-ins can be created by deriving from the Control class and anything that can be animated has a controller. For instance parameter blocks have controllers as they can be animated.

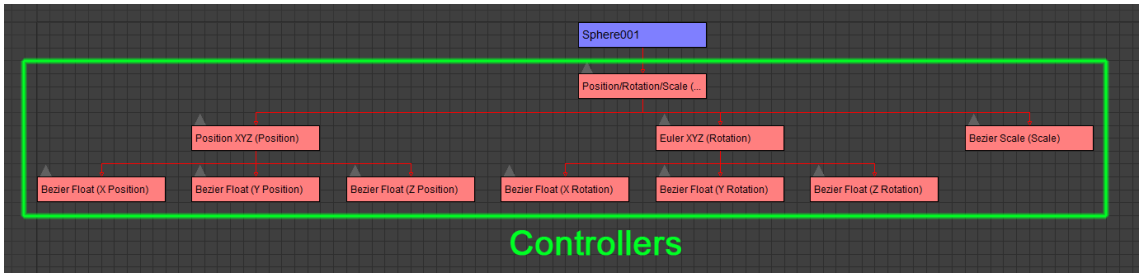

Fig 40. Node graph showing the controller nodes of a object.

Constraints are like controllers in Max but instead reference another scene object. This is the only difference between constraints and controllers. An example of a constraint would be a path constraint as it references another scene object. Constraints still use the same classes.

```
class PSParticlePosition : public Control {
private:
    // This is the only variable we have to store
    Point3 m_vForce;
    Point3 m_vVelocity;
    Point3 m_vPosition;

public:


    //From Animatable
    Class_ID ClassID()          {return PSParticlePosition_CLASS_ID;}
    SClass_ID SuperClassID()    { return CTRL_POSITION_CLASS_ID; }
    void GetClassName(TSTR& s)  {s = GetString(IDS_POS_CTRL_CLASS_NAME);}

    void DeleteThis() { delete this; }

    // From ReferenceTarget
    RefResult NotifyRefChanged(Interval changeInt, RefTargetHandle hTarget,
        PartID& partID,  RefMessage message) { return REF_SUCCEED; }

    // From class Control
    void Copy(Control *from) {};

    // These are the 2 ones need - They are how we recieve and return calculated values
    void GetValue(TimeValue t, void *val, Interval &valid, GetSetMethod method/* =CTRL_ABSOLUTE */);
    void SetValue(TimeValue t, void *val, int commit/* =1 */, GetSetMethod method/* =CTRL_ABSOLUTE */);

    //Constructor/Destructor
    PSParticlePosition();
    ~PSParticlePosition() {}

};
```

Fig 41. This code show how to create a controller plug-in using the Control class. The get and set value functions do most of the work here and that's where the plug-in code would go.

5.15. Function Publishing

The function publishing system allows developer to create functions which can be exposed to other plug-ins and maxscript. Using a mixture of C++ and maxscript to develop plug-ins can make plug-in development much easier, less error prone, more stable and plug-ins can be faster than standard maxscript plug-ins. Function publishing can also be used to publish functions to different areas of Max such as the user interface. For instance functions can be published and then called from macro scripts, toolbars, hot keys and quad menus.

Published functions can be used for almost anything in Max but fall into three main categories to why they would be published:

- The first reason is to create important algorithms which can then be imported into other plug-ins and maxscript.
- The second reason would be to affect the plug-in object in the scene by exposing values which edit the object. Normally the parameter system would do this, but the function publishing system can be used in certain situations.
- The third reason would be to create functions which are then exposed to the user interface and can be called by custom buttons.

The function publishing system consists of an interface and an implementation for that interface. There are different methods for publishing function but most methods will require certain items:

- Each interface which is published requires a constant interface ID like any other plug-in.
- A public interface class which holds a list of virtual functions that will be exported.
- An implementation class which inherits from the interface class to implement the virtual functions and holds a function map.
- An interface definition which is a static instance of the public interface class. Its constructor is used to describe the functions being exported similar to parameter blocks.

There are other methods to publish functions such as the mixin interface method which allows developers to publish functions using their own plug-in class. This allows for the creation of functions which only work for that plug-in object.

If functions have been exported to maxscript they are called on a specific way. To call a published function in maxscript users would first call the interface then the function. For example:

Interface.function paramteres
CacheInterface.import “file path”

```
#include"iFnPub.h"
//Interface ID.
#define CACHE_FP_INTERFACE Interface_ID(0x5f034f3e, 0x16793b37)

#define GetCachePIInterface(cd) \
   (CacheFPPI *)(cd)->GetFPInterface(CACHE_FP_INTERFACE)

//Holds the virtual functions that will be exported.
class CacheFPPI : public  FPStaticInterface
{
    public:
        virtual void Import (char *filePath) = 0;
        virtual void Export (INode* node, char *filePath, char *fileExtension) = 0;
        virtual void About () = 0;
        virtual float Multiply (float x, float y) = 0;

        enum
        {
            em_import,
            em_export,
            em_about,
            em_multiply
        };

    private:
        INode* node;
};
```

```
//Derived from the previous class and defines a function map
//as well as the functions being exported
class CacheFPImp : public CacheFPPI
{
public:
    //Function map.
    DECLARE_DESCRIPTOR(CacheFPImp)
    BEGIN_FUNCTION_MAP
        //1 Function type VFN1-Void function takes one argument,
        //2 function number,
        //3 the actual function,
        //4 the type of parameter that will be accepted by maxscript
        //1     2           3           4
        VFN_1(em_import, Import, TYPE_STRING)
        //1     2           3           4           4           4
        VFN_3(em_export, Export, TYPE_INODE, TYPE_STRING, TYPE_STRING)
        VFN_0(em_about, About)
        FN_2(em_multiply, TYPE_FLOAT, Multiply, TYPE_FLOAT, TYPE_FLOAT)
    END_FUNCTION_MAP

    //Functions Implementation

    void Import(char *filePath) { ... }

    void Export(INode* node, char *filePath, char *fileExtension) { ... }

    void About() { ... }

    //Test function multiplies two floats.
    float Multiply(float x, float y) { ... }
};

 //Interface definition defines how to use the functions in maxscript.
 static CacheFPImp CacheIDef
 (
     CACHE_FP_INTERFACE,
     _T("Cache_FP"),
     0,
     NULL,
     FP_CORE,

     CacheFPPI::em_import, _T("import"), 0, TYPE_VOID, 0, 1,
         _T("char_filePath"), 0, TYPE_STRING,
     CacheFPPI::em_export, _T("export"), 0, TYPE_VOID, 0, 3,
         _T("INode_node"), 0, TYPE_INODE,
         _T("char_filePath"), 0, TYPE_STRING,
         _T("char_fileExtension"), 0, TYPE_STRING,
     CacheFPPI::em_about, _T("about"), 0, TYPE_VOID, 0, 0,
     CacheFPPI::em_multiply, _T("multiply"), 0, TYPE_FLOAT, 0, 2,
         _T("float_X"), 0, TYPE_FLOAT,
         _T("float_Y"), 0, TYPE_FLOAT,
     ParamTags::end
 );
```

Fig 42. The code show how the function publishing system can be used to publish a function to maxscript.

5.16. Useful Functions and Classes

The Max SDK has a many useful functions and class which can be used to help in plug-in development.

For memory management the SDK has some custom functions as standard C++ functions like malloc should not be used.

Max_malloc() - This is the same as the standard C++ malloc.
Max_free() - This is the same as the standard C++ free.

Point3() - This is a vector type and support only three values.

The SDK has a collection class which is used to hold multiple values just like an array. However this class can resize, delete and add to the array and has more functions than the standard C++ array.

Tab<>- This is how a collection is made.

Max has many ways to debug a plug-in but one method is to print strings out into the maxscript listener. The maxscript.h file must be included into the project to use these functions.

mprintf() - The string that is going to be printed to the listener.
mflush() - Sends everything to the listener.

The reference system provides a useful way to get a pointer to a node of an object. This can be helpful when a object plug-in is being made as the node is created but never stored.

```
//Handle
ULONG handle = 0;
//Gets a handle to the node
NotifyDependents(FOREVER, (PartID)&handle, REFMSG_GET_NODE_HANDLE,
NOTIFY_ALL, TRUE, NULL);
//Uses the handle to get the node and stores a pointer to it.
INode *node = GetCOREInterface()->GetINodeByHandle(handle);
```

6. Cache Plug-in Development

One of the most important parts of a geometry cache system is its file format. For this plug-in the file format is an already existing one and it just needs to be imported and exported from and to 3D Max.

There are two main options for creating plug-ins in Max which edit or create geometry, object and modifier plug-ins. For a cache plug-in the most obvious choice would be to create an object plug-in as an existing object is not being modified. However a modifier plug-in could be used to do the same thing.

Although an object plug-in would import the file at every frame, the plug-in still needs a way to export geometric objects. This could be done using a utility plug-in or using the function publishing system. The function publishing system would be the easier method as it could be made in the same plug-in project and would be available to maxscript. Even though two plug-ins can be created in one Visual Studios project, both plug-ins would require a class descriptor and it would be a pain to manage two plug-ins. As well as this the function publishing method allows users to use the export function in their maxscripts.

As a cache plug-in would import geometry which consists of polygons and not solely triangles, this plug-in will derive from the PolyObject class. Although the SimpleObject2 class is easier to use it is limited to triangle mesh. This makes it much more difficult to develop an object plug-in. However the PolyObject class uses the MNMesh class to create mesh which is slower than the Mesh class.

The plug-in code (see Appendix C) shows how to set up and create an object plug-in from scratch. The code also shows lots of different classes and functions from the SDK which can be useful for all plug-ins. To fully understand this thesis and some of the SDK it is recommend to refer to the code.

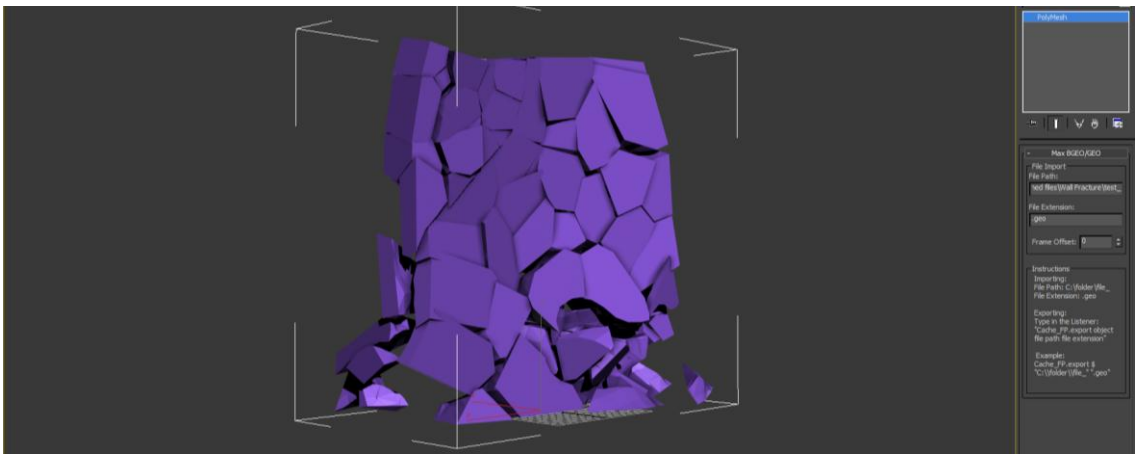

Fig 43. Show a screenshot of imported geometry and user interface.

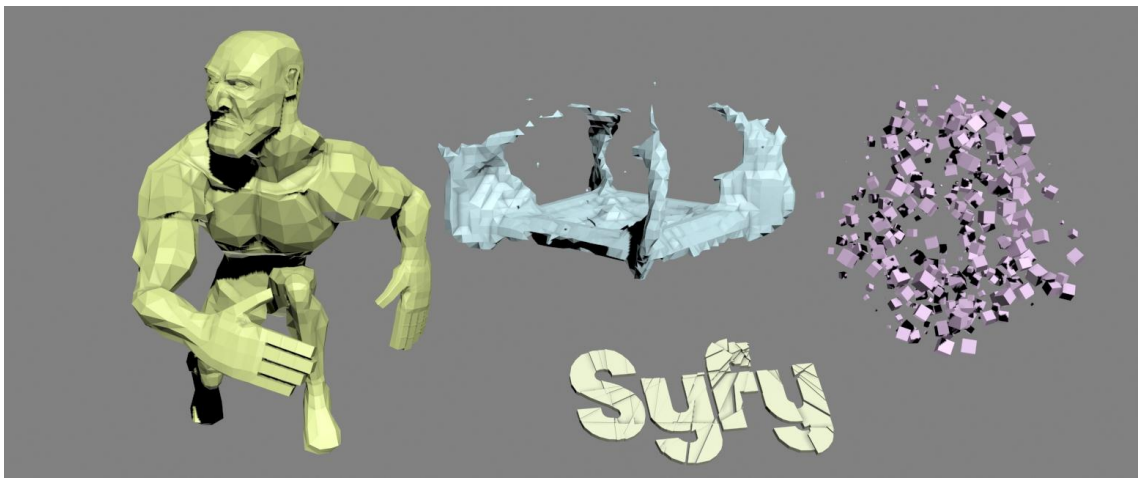

Fig 44. Screen shots of the final plug-in in Max used to bring geometry from Houdini into Max. All the objects are animated.

6.1. GPD Library

To read and write data to Houdini's geometry cache file format, the GPD library was used. This library is old and consisted of many errors before it was implemented into the plug-in. Because of this the plug-in doesn't fully utilize the cache format as it only works for the Geo one and not the Bgeo format. This means no reading and writing to a binary file format which is much faster than the Geo text file format. To fully make use of the format a lot of editing and adding to the GPD library would need to be done.

Using the Geo format does cause performance issues but this is not the only performance problem in the plug-in. The plug-in reads a file into memory at each frame and then dumps it when the frame changes. It does this for every frame a file is being read which slows down the plug-in a lot. This could be fixed by using memory mapped files which maps all the files to virtual memory and then this memory can be used to read the data.

7. Conclusion

The point of this project was to learn about plug-in development using C++ and the Max SDK. The project was ambitious from the start and a massive amount of learning was necessary to move from scripting to programming for the first time. With this in mind the project was successful and the plug-in does work even though it's not as fast as it could be.

The report looked over the more important areas of the Max SDK but did cover everything. However the areas covered supply enough information to develop common plug-ins using the SDK. For instance a lot of information is provided on how to develop object and modifier plug-in.

The plug-in itself provides lots of information about the SDK because of all the classes and functions used to create it. This is why an importer and exporter type of plug-in was developed.

The SDK doesn't have much information or learning resources available. The main learning resource is the online documentation and it is aimed towards more experienced programmers. However there are tones of sample projects provided by Autodesk and this is the best place to look when learning the SDK. This does mean a lot of time is spent looking through the entire SDK.

To fix the performance issues the plug-in will have to work with the binary format and use memory mapped methods. This will require more work to be done with GPD library than the Max SDK.

References

Koenig, A.K., Moo, B.E.M (2000) *Accelerated C++ Practical Programming by Example.* New York: Addison Wesley

Autodesk. (2012) *3ds Max SDK Programmer's Guide and Reference.* http://download.autodesk.com/global/docs/3dsmaxsdk2012/en_us/index.html [Accessed November 2011]

CPlusPlus. http://www.cplusplus.com/ [Accessed October 2011]

Gamasutra. (2012) http://www.gamasutra.com/view/feature/131579/creating_procedural_objects_in_3d_.php?print=1 [Accessed November 2011]

Side Effects Software, (2012) *SideFx.* http://www.sidefx.com/index.php?option=com_content&task=view&id=22&Itemid=51 [Accessed October 2011]

Jian, Z. (2012) Build Mesh and Convert to MNMesh, *Acacia Echo*, 06 April. http://acaciaecho.wordpress.com/2011/04/06/build-mesh-and-convert-to-mnmesh/ [Accessed 10 January 2012]

Diggins, C. Adding New Functions Written in C++ to Maxscript, *Area*, 16 September. http://area.autodesk.com/blogs/chris/adding_new_functions_written_in_c_to_maxscript [Accessed 02 Feburary 2012]

Taylor, S. (2011) *3Ds Max - Webcasst Training* [Video]. Autodesk

Taylor, S. (2011) *DevTv Introduction to 3Ds Max Programming* [Video]. Autodesk

Appendix A - Project Specification

Pipeline and Tool Development

For my project I am going to be focusing on plug-in development and will be creating a set of tools to improve data management between software packages.

I am going to be creating a custom file importer and exporter for Houdini and 3D Max. It will basically import and export geometry cache data between both software packages. The cache file will have to be able to load geometry into the viewport in real time. I will also look into existing file types such FBX and seeing what the advantages and disadvantages are of custom files.

My overall goals are to learn more about tool and pipeline development using C++ and the 3Ds Max source development kit.

Appendix B - Parameter Blocks Template

```
ParamBlockDesc2
(
   // Required specifications
   BlockID     ID,
   TCHAR*      int_name,
   int        local_name,
   ClassDesc2* cd,
   BYTE       flags,

   // ONLY IF P_VERSION is specified as a flag
   int  version_number,

   // ONLY IF P_AUTO_CONSTRUCT is specified as a flag
   // This is the same reference number that specifies the
   // parameter block used in GetReference() and SetReference().
   int  reference_number,

   // ONLY IF P_AUTO_UI is specified above
   // Automatic UI parameter map specifications
   int  dialog_template_ID,
   int  dialog_title_res_ID,
   int  flag_mask,
   int  rollup_flags,
    ParamMap2UserDlgProc* proc,

   // ONLY IF P_INCLUDE_PARAMS or P_USE_PARAMS is specified as a flag
   // Address of source IParamblockDesc to use
    ParamBlockDesc2* pSourceDesc,

     // OPTIONAL
     // FOR EACH variable control create a parameter specification:
     // There is one of these parameter specifications for each control.
     ParamID   id,
     TCHAR*    internal_name,
     ParamType type,
     // Optional. Use this argument ONLY IF you use a Tab<> type
     [int      table_size,
     int       flags,
     int      local_name_res_ID,

       // FOR EACH parameter specification, specify zero or more parameter tags.
        // and optional parameter information
       Parameter_Tag

     end, //End for each parameter specification
   end //End for all the variable arguments
);
```

Appendix C - Plug-in Code

```
\\Cache plug-in


#include "Cache_Plugin.h"

#define PBLOCK_REF	0

#define CacheObject_CLASS_ID Class_ID(0x218f77ba, 0x25666b07) //Uniqe ID for
plug-in.

//This plug-in is a poly object.
class CacheObject : public PolyObject, public TimeChangeCallback
{
public:
    CacheObject();
        ~CacheObject();

        //Parameter block pointer.
        IParamBlock2 *pblock2;
        //The mesh.
        MNMesh mnmesh;
        //Interval of the object.
        Interval ivalid;

        //Reference another scene node. This is a test.
        INode *RefTargetNP;
        //Pointer to this objects node as this is a object a node has to be
ccreated for it.
        INode *thisnp;

        //Interface pointer.
        static IObjParam *ip;

    //Custom Build MNMesh used to build the objects mesh.
    void BuildMnmesh(TimeValue t);

        //From Interval
        void InvalidateUI();

    //From BaseObject
        //Create the mouse callback used to create the object in the scene.
    CreateMouseCallBack* GetCreateMouseCallBack();
        //Evaluate the object state whenever.
        ObjectState Eval(TimeValue time);
        //Set the objects validity to only be valid for one frame.
        Interval ObjectValidity(TimeValue t) { Interval iv; iv.Set(t,t); return
iv; }
        //Check if the object can be converted to another type. Modifiers ask for
this.
        int CanConvertToType(Class_ID obtype);
        //If can convert then convert. Modifiers will use this to convert the
object in to what it needs.
        Object* ConvertToType(TimeValue t, Class_ID obtype);
        //Display the object on screen
        int Display(TimeValue t, INode* inode, ViewExp *vpt, int flags);
        //Check if it is ok to display the object.
        BOOL OKtoDisplay(TimeValue t);

        //Custom Functions
        //Create the uv data from file.
        void createUV(MNMesh mnmesh);
        //Create normal data from file.
```

```
        void createNormals(MNMesh mnmesh);

        //Parameter Blocks
        int NumParamBlocks() { return 1; }
        IParamBlock2 *GetParamBlock(int i) { return pblock2; }
        IParamBlock2 *GetParamBlockByID(BlockID id) { return (pblock2->ID() ==
id) ? pblock2 : NULL; }

        //From Animatable
        //Display the ui when needed.
        void BeginEditParams( IObjParam  *ip, ULONG flags,Animatable *prev);
        //Stop displaying the ui when needde.
        void EndEditParams( IObjParam *ip, ULONG flags,Animatable *next);
        //Number of sub animatables.
        int NumSubs() { return 1; }
        //Return each sub animatable.
        Animatable *SubAnim(int i) { return pblock2; }

        //Refrence Maker
        //Number of references this plug-in holds.
        int NumRefs();
        //Get the set reference.
        ReferenceTarget *GetReference(int i);
private:
        //Set a reference to something.
        void SetReference(int i, ReferenceTarget* pTarget);
public:
        //Notify this plug-in when a reference changes.
        RefResult NotifyRefChanged(Interval changeInt, RefTargetHandle hTarget,
PartID& partID, RefMessage message);
        //Manage references when cloning an object.
        RefTargetHandle Clone( RemapDir &remap );

        //Notifiy this plug-ins dependants such as it node. Not used.
        //RefResult NotifyDependents(Interval changeInt, PartID partID,
RefMessage message, SClass_ID sclass, BOOL propagate, RefTargetHandle hTarg);

        //From TimeChangeCallback. Is called when the time changes.
        void TimeChanged(TimeValue t);

    //From Animatable
    Class_ID ClassID() {return CacheObject_CLASS_ID;}
    SClass_ID SuperClassID() { return GEOMOBJECT_CLASS_ID; }
    void GetClassName(TSTR& s) { s = "Cache Geometric Object";}

        //Delete this object properly.
    void DeleteThis() { delete this; }

        HWND hWnd;
};


//--------------------------------------------------------------------------
//Class descriptor implementation

class CachePluginClassDesc : public ClassDesc2
{
public:
    //ClassDesc2 overrides

        //Plugin is accessable through the user interface.
        int             IsPublic()                                          {
return TRUE; }
        //Return the plug-in class as a pointer so Max can get a new instance of
it.
```

```
    void*       Create(BOOL loading = FALSE)   { return new CacheObject();}
        //The plug-ins name in the user interface.
    const       MCHAR* ClassName()                          { return
_M("BGEO/GEO"); }
        //This plug-in is geometric object plug-in which is an object that can be
created in the scene by users.
                SClass_ID SuperClassID()            { return
GEOMOBJECT_CLASS_ID; }
        //Return a uniqe class ID for this plug-in.
                        Class_ID ClassID()                          { return
CacheObject_CLASS_ID; }
        //Make an area in the create panel of Max for this plug-in.
    const       MCHAR* Category()                           { return _M("Max
BGEO/GEO"); }
        //Returns fixed parsable name.
    const       MCHAR* GetInternalName()            { return _M("BGEOGEO"); }

        //Return the DLL hinstance.
                        HINSTANCE HInstance()               { return hInstance;
}
};

//Returns a singleton instance of the class descriptor.
static CachePluginClassDesc CacheDesc;
ClassDesc2* GetClassDescInstance() { return &CacheDesc; }

//--------------------------------------------------------------------------
//Parameter Blocks

IObjParam *CacheObject::ip     = NULL;

enum
{
        object_params
};
enum
{
        file_path,
        file_extension,
        file_offset
};

static ParamBlockDesc2 CacheObjectParam(
        //Reuired parameters.
        object_params,
        _T("Max BGEO/GEO"),
        0,
        &CacheDesc,

        //Flags
        P_AUTO_CONSTRUCT + P_AUTO_UI,

        //Dependent arguments ---------------------------
        //required because P_AUTO_CONSTRUCT was flagged
        //This declares the reference number of the Parameter Block.
        PBLOCK_REF,

        //required because P_AUTO_UI was flagged.
        //This is the Rollout description
        IDD_PANEL, IDS_PARAMS, 0, 0, NULL,

        //Parameter Specifications ----------------------
        // For each control create a parameter:
        file_path,                      _T("File Path"),              TYPE_STRING,
                P_INVISIBLE,            IDS_FILEPATH,
```

```
		p_default,		"C:\\PATH\\FILE",
		p_ui,			TYPE_EDITBOX,
	IDC_FILE_PATH,
		ParamTags::end,

	file_extension,		_T("File Extension"),	TYPE_STRING,
	P_INVISIBLE,		IDS_FILE_EXTENSION,
		p_default,		".bgeo or .geo",
		p_ui,			TYPE_EDITBOX,
	IDC_FILE_EXTENSION,
		ParamTags::end,

	file_offset,		_T("File Offset"),		TYPE_INT,
		P_ANIMATABLE,		IDS_FILE_OFFSET_SPIN,
		p_default,		0,
		p_range,		0,10000,
		p_ui,			TYPE_SPINNER,
	EDITTYPE_INT,		IDC_FRAME_OFFSET_EDIT,
	IDC_FRAME_OFFSET_SPIN, 1,
		ParamTags::end,
	ParamTags::end
);

//Custom button ui handeling.
class CacheObjectDlgProc : public ParamMap2UserDlgProc {
private:
	//This plug-in.
	CacheObject *ob;
public:
	CacheObjectDlgProc(CacheObject *s) { ob = s; }
	INT_PTR DlgProc(TimeValue t, IParamMap2 *map, HWND hWnd, UINT msg, WPARAM
wParam, LPARAM lParam);
	void DeleteThis() { delete this; }
};

//Handle button pressing.
INT_PTR CacheObjectDlgProc::DlgProc(TimeValue t, IParamMap2 *map, HWND hWnd, UINT
msg, WPARAM wParam, LPARAM lParam)
{
	//Implement cutsom button.
	return FALSE;
}

//--------------------------------------------------------------------------------
//CacheObject Implementation

//void NodeDeleteNotify(void *param, NotifyInfo *info)
//{
//
//}

//Plug-in constructor.
CacheObject::CacheObject()
{
	//Set pointers to NULL as they must be defined or Max will crash.
	RefTargetNP = NULL;
	pblock2 = NULL;
	//Create the parameter blocks automatically.
	CacheDesc.MakeAutoParamBlocks(this);
	//Register time change.
	GetCOREInterface()->RegisterTimeChangeCallback(this);
	//RegisterNotification(NodeDeleteNotify, this,
NOTIFY_SCENE_PRE_DELETED_NODE);
}
```

```
//Plug-in deconstructor.
CacheObject::~CacheObject()
{
	//UnRegister time change
	GetCOREInterface()->UnRegisterTimeChangeCallback(this);
	//Delete all references from the plug-in when the plug-in is unloaded.
	DeleteAllRefs();
	//UnRegisterNotification(NodeDeleteNotify, this,
NOTIFY_SCENE_PRE_DELETED_NODE);
}

//--------------------------------------------------------------------------------
//Mouse Callback

class CacheObjectMouseCallback : public CreateMouseCallBack
{
	IPoint2 sp;			//First point in screen coordinates
	CacheObject *ob;	//Pointer to the object
	Point3 wp;			//First point in world coordinates
public:
	//Handel mouse stuff.
	int proc( ViewExp *vpt,int msg, int point, int flags, IPoint2 m, Matrix3&
mat);
	void SetObj(CacheObject *obj) {ob = obj;}
};

int CacheObjectMouseCallback::proc(ViewExp *vpt,int msg, int point, int flags,
IPoint2 m, Matrix3& mat)
{
	TimeValue t (0);
	if (msg==MOUSE_POINT||msg==MOUSE_MOVE) {
		switch(point)
		{
		case 0:
			{
				sp = m;
				wp = vpt->SnapPoint(m,m,NULL,SNAP_IN_PLANE);
				mat.SetTrans(wp); // sets the pivot location
				break;
			}
		case 1:
			{
				ob->mnmesh.InvalidateGeomCache();
				ob->BuildMnmesh(t);
				ob->InvalidateUI();
				break;
			}
		case 2:
			{
				return CREATE_STOP;
			}
		}
	}
	else {
		if (msg == MOUSE_ABORT)
		{
			return CREATE_ABORT;
		}
	}
	return TRUE;
}

static CacheObjectMouseCallback CacheObjectCreateCB;

CreateMouseCallBack* CacheObject::GetCreateMouseCallBack()
```

```
{
	//Create the node for the object.
	CacheObjectCreateCB.SetObj(this);

	//Get the newly created node by its handle and add it to thisnp pointer.
	ULONG handle = 0;
	NotifyDependents(FOREVER, (PartID)&handle, REFMSG_GET_NODE_HANDLE,
NOTIFY_ALL, TRUE, NULL);
	thisnp = GetCOREInterface()->GetINodeByHandle(handle);

	//Set a reference to the created node
	//ReplaceReference(1, thisnp);

	return(&CacheObjectCreateCB);
}

//This plug-in has 2 references.
int CacheObject::NumRefs()
{
	return 2;
}

//Get both references.
ReferenceTarget *CacheObject::GetReference(int i)
{
	switch(i)
	{
		case 0:
			return (RefTargetHandle)pblock2;
			break;
		case 1:
			return (RefTargetHandle)RefTargetNP;
			break;
	}
}

//Set references to parameter block and a node.
void CacheObject::SetReference(int i, ReferenceTarget* pTarget)
{
	switch (i)
	{
		case 0:
			pblock2 = (IParamBlock2*)pTarget;
			break;
		case 1:
			RefTargetNP = dynamic_cast<INode*>(pTarget);
			break;
	}
}

//Notify the plug-in when the reference changes in some way.
RefResult CacheObject::NotifyRefChanged(Interval changeInt, RefTargetHandle
hTarget, PartID& partID, RefMessage message)
{
	return REF_SUCCEED;
}

//RefResult CacheObject::NotifyDependents(Interval changeInt, PartID partID,
RefMessage message, SClass_ID sclass, BOOL propagate, RefTargetHandle hTarg)
//{
//	if(message == REFMSG_GET_NODE_HANDLE)
//	{
//
//	}
//
```

```
//      return REF_SUCCEED;
//}

RefTargetHandle CacheObject::Clone(RemapDir& remap)
{
        CacheObject *newob = new CacheObject();
        // Make a copy all the data and also clone all the references
        newob->ReplaceReference(0,remap.CloneRef(pblock2));
        newob->ivalid.SetEmpty();
        BaseClone(this, newob, remap);
        return(newob);
}

void CacheObject::BeginEditParams(IObjParam *ip,ULONG flags,Animatable *prev)
{
        this->ip = ip;

        CacheDesc.BeginEditParams(ip, this, flags, prev);

        //CacheObjectDlgProc.SetObject(this);
}

void CacheObject::EndEditParams( IObjParam *ip, ULONG flags, Animatable *next )
{
        CacheDesc.EndEditParams(ip, this, flags, next);

        this->ip = NULL;
}

void CacheObject::createUV(MNMesh mnmesh)
{
        mnmesh.SetMapNum(2);
        MNMap *map = mnmesh.M(1);

        int numVerts = mnmesh.VNum();
        int numFaces = mnmesh.FNum();

        map->setNumVerts(numVerts);

        for(int i=0; i<numVerts; i++)
        {
                UVVert &textVert = map->v[i];
                textVert.x = 0.0f;
                textVert.y = 0.0f;
                textVert.z = 0.0f;
        }

        //MNMapFace texFace = map->f[0];

}

void CacheObject::createNormals(MNMesh mnmesh)
{
        int numVerts = mnmesh.VNum();
        int numFaces = mnmesh.FNum();

        mnmesh.SpecifyNormals();
        MNNormalSpec *normalSpec = mnmesh.GetSpecifiedNormals();
        normalSpec->ClearNormals();
        normalSpec->SetNumNormals(numVerts);
        normalSpec->SetNumFaces(numFaces);

        for(int i=0; i<numVerts; i++)
        {
                normalSpec->Normal(i) = Normalize (Point3(1,1,0));
```

```
                normalSpec->SetNormalExplicit(i,true);
        }

        for(int i=0; i<numFaces; i++)
        {
                MNNormalFace &normalFace = normalSpec->Face(i);
                normalFace.SpecifyAll();
                normalFace.SetNormalID(0, i);
        }
}

//Build the mesh at every frame.
void CacheObject::BuildMnmesh(TimeValue t)
{
        //The object is always valid when created.
        ivalid = FOREVER;

        //Get the mesh.
        MNMesh &mnmesh = this->GetMesh();
        //this->mnmesh = this->GetMesh();

        GPD_Detail gpd;

        //Remove and delete anything left over from last frame.
        mnmesh.freeFaces();
        mnmesh.freeEdges();
        mnmesh.freeVerts();

        //Get the current frame.
        int curFrame = t/GetTicksPerFrame();

        //Get the frame offset parameter and assign it to a variable.
        int frameOffset;
        pblock2->GetValue(file_offset, t, frameOffset, ivalid);

        //Get the string paramters.
        const char *filePath = pblock2->GetStr(file_path, t); //Gets first
parameter
        const char *fileExtension = pblock2->GetStr(file_extension, t); //Gets
second parameter

        //Create the final string START.
        char *fileFrame;
        fileFrame = (char*) MAX_malloc(1);

        itoa((curFrame+frameOffset), fileFrame, 10);

        int fileStrLength = strlen(filePath) + strlen(fileFrame) +
strlen(fileExtension) + 1;
        char *fileStr;
        fileStr = (char*) MAX_malloc(fileStrLength);
        fileStr[0] = 0;

        strcat_s(fileStr, fileStrLength, filePath);
        strcat_s(fileStr, fileStrLength, fileFrame);
        strcat_s(fileStr, fileStrLength, fileExtension);

        MAX_free(fileFrame);
        //Create the final string END.

        //Load the file from disk using the created string.
        gpd.load(fileStr);

        //Get num verts and face from disk.
        int numVerts = gpd.numpoint();
```

```
	int numFaces = gpd.numprim();

	//Loop through vert count and create verts.
	for(int i=0; i<numVerts; i++)
	{
		//Get the verts from the file and conveter from meters to inches
		float x = gpd.point(i)->pos[0] * 39.370;
		float y = gpd.point(i)->pos[1] * 39.370;
		float z = gpd.point(i)->pos[2] * 39.370;
		//Creates vert using Y up from file
		mnmesh.NewVert(Point3(x, z, y));
	}

	//Loop through face count and create verts.
	for(int i=0; i<numFaces; i++)
	{
		//Get the number of verft for the i face.
		int numFaceVerts = gpd.prim(i)->getVertexNum();

		//Allocate memory for a face vert array.
		int *faceVerts;
		faceVerts = (int*) MAX_malloc(sizeof(int)*numFaceVerts);

		//Loop throught i face verts.
		for(int n=0; n<numFaceVerts; n++)
		{
			//Put each face vert into the array.
			faceVerts[n] = gpd.prim(i)->getVertex(n)->getPt()-
>getNum();
		}

		//Create the face.
		mnmesh.CreateFace(numFaceVerts, faceVerts);
		//Deallocate the memory.
		MAX_free(faceVerts);
	}

	//Print to listner.
	mprintf(fileStr);
	mprintf("\n");
	mflush();

	//Tests Start
	//createNormals(mnmesh);
	//createUV(mnmesh);

	//Print the reference targets nodes name and create a point at its
position
	//char *name;
	//if(RefTargetNP != NULL)
	//{
	//	name = RefTargetNP->GetName();
	//	Point3 vTargPos = RefTargetNP->GetNodeTM(t, &ivalid).GetTrans();
	//	mnmesh.NewVert(vTargPos);
	//}
	//else
	//{
	//	name = "NULL";
	//}

	//Print this objects name
	//char *thisname;
	//if(thisnp != NULL)
	//{
	//	thisname = thisnp->GetName();
```

```
        //}
        //else
        //{
        //      thisname = "NULL";
        //}

        //Print to listner
        //mprintf(thisname);
        //mprintf("\n");
        //mprintf(name);
        //mprintf("\n");
        //mflush();
        //Test end

    mnmesh.InvalidateGeomCache();
        mnmesh.InvalidateTopoCache();
}

void CacheObject::InvalidateUI()
{
        pblock2->GetDesc()->InvalidateUI();
}

ObjectState CacheObject::Eval(TimeValue t)
{
        return ObjectState(this);
}

int CacheObject::CanConvertToType(Class_ID obtype)
{
        return 1;
}

Object *CacheObject::ConvertToType(TimeValue t, Class_ID obtype)
{
        //Make sure the object is only valid for one frame.
        ivalid = FOREVER;
        ivalid.Set(t, t);

        MNMesh &mnmesh = this->mm;

        TriObject *ob = new TriObject();
        Mesh &mesh = ob->GetMesh();

        //Convert the object to tri object.
        mnmesh.OutToTri(mesh);

        //Sort out channels.
        ob->SetChannelValidity(TOPO_CHAN_NUM, ivalid);
        ob->SetChannelValidity(GEOM_CHAN_NUM, ivalid);
        ob->UnlockObject();

        return ob;
}

BOOL CacheObject::OKtoDisplay(TimeValue t)
{
        BOOL displayOk = TRUE;
        return displayOk;
}

int CacheObject::Display(TimeValue t, INode* inode, ViewExp *vpt, int flags)
{
   if (!OKtoDisplay(t)) return 0;
   GraphicsWindow *gw = vpt->getGW();
```

```
   Matrix3 mat = inode->GetObjectTM(t);
   //BuildMnmesh(t); // UpdateMesh just calls BuildMesh() if req'd at time t. //
DONT DO THIS AS TIMECHANGE IS BEING USED.

   gw->setTransform(mat);
   mnmesh.render( gw, inode->Mtls(),
         (flags&USE_DAMAGE_RECT) ? &vpt->GetDammageRect() : NULL, COMP_ALL,
         inode->NumMtls());
   return(0);
}

void CacheObject::TimeChanged(TimeValue t)
{
        //Get interface.
        Interface *ip = GetCOREInterface();

        //Check if the object exists
        ULONG handle = 0;
        NotifyDependents(FOREVER, (PartID)&handle, REFMSG_GET_NODE_HANDLE,
NOTIFY_ALL, TRUE, NULL);
        thisnp = GetCOREInterface()->GetINodeByHandle(handle);
        //If it dosent exist unregister time change and tell listner the object
dosent exist.
        if(thisnp == NULL)
        {
                GetCOREInterface()->UnRegisterTimeChangeCallback(this);
                mprintf("Object Was Deleted\n");
                mflush();
        }
        //If it exists rebuild the mesh and update the viewport.
        else
        {
                BuildMnmesh(t);

                ViewExp *vpt = ip->GetActiveViewport();
                Display(t, thisnp, vpt, 1);
                //ip->RedrawViews(t);
        }

        //Set the interface pointer to NULL just incase. Pointers left over can
cause errors.
        ip = NULL;
}
```