XSI Scripting Primer

XSI Scripting Primer **1**

Introduction **1**

Overview **1**

Syntax of VBScript and JScript **2**

Arrays and Collections in VBScript and JScript **3**

XSI Commands **5**

Returning Objects from Commands *7*

XSI Object Model **9**

Properties and Methods *9*

Object References *10*

Traversing the Graph *12*

Where to find more information **15**

Introduction

This document provides a quick introduction to scripting inside XSI. There is a wealth of additional documentation available on this topic, but this can be a quick start to get you up and running. This covers v3.5 of XSI.

Overview

XSI does not have a proprietary scripting language, instead it supports the popular scripting languages **VBScript**, **JScript**, **PerlScript** and **Python**. The XSI API is made up of Commands and an Object Model. The Commands are action-oriented procedure calls, whereas the Object Model is well suited for traversing and manipulating scene data.

Using scripts you can add your own Custom Commands, as well as Custom Events, Custom Operators and Custom UI. XSI is widely exposed to scripting, in fact large part of its commands and user interface is written using the SDK. And because XSI uses **ActiveScripting** technology it is easy to communicate with other Windows applications and to integrate XSI scripts into dynamic web content hosted in Netview.

As users work in XSI their actions are logged as script commands, so it is easy to optimize repetitive tasks by bundling complex actions into Macro-like custom commands that can be executed with a single keystroke. XSI includes a built in script editor and it is possible to debug using the **Microsoft Script Debugger**, a free download from http://msdn.microsoft.com/scripting.

Syntax of VBScript and JScript

VBScript and JScript are the two most popular scripting languages so the examples will be presented in those languages. We will start with a very quick overview of the syntax of these languages.

VBScript	Jscript

`' this is a comment`
 	
`REM I'm also a comment–a throwback to Basic`
	`// this is a single-line comment`
	
	`/* this is also`
	`       a comment but can be`
	`              spread over many lines  */`

`' continuing lines needs "_" character`
`' and no semicolons at the end of statements`
`logmessage _`
`        "hello world"`
	`// continuing lines just like C/C++`
	`// and statements end with ";"`
	`logmessage(`
	`       "hello world");`

`' string concatenation`
`str = "hello" & " world"`
	`// string concatenation`
	`str = "hello" + "world";`

`' assigning an object to a variable`
`set o = selection`
	`// assigning an object to a variable`
	`o = selection;`

`' declaring variables`
`dim x,y`
	`// declaring and initializing variables`
	`var x=0,y=1;`

`' comparison and assignment of numbers`
`if ( x = 5 AND y <> 4 ) then`
`     z = 6`
`end if`
	`// comparison and assignment of numbers`
	`if ( x == 5 && y != 4 ) {`
	`  z = 6 ;`
	`}`

VBScript	Jscript

`' handling errors`
`on error resume next`
`  err.description = "fake error"`
`  err.raise vbObjectError + 1000`
`  logmessage err.code & ":" & err.description`
`on error goto 0`
	`// handling errors`
	`try {`
	`  throw new Error( 10, "fake error");`
	`}`
	`catch (e) {`
	`        logmessage(e.number +":"+e.message);`
	`}`

`' defining a function with a return value`
`function foo2( arg1 )`
`        foo2 = "hello " & arg1`
`end function`
 	
`' msg will have the value "hello world"`
`msg = foo2( "world" )`
	`// defining a function with a return value`
	`function foo( arg1 ) {`
	`        return "hello "+ arg1;`
	`}`
	
	`// msg will have the value "hello world"`
	`var msg = foo( "world" ) ;`

`' subroutines have no return values`
`sub mysub( arg1 )`
`    LogMessage arg1`
`end sub`
 	
`' Calling a subroutine is special case`
`call mysub( 45 )`
 	
`' or (using parentheses here is an error)`
`mysub 45`
	`// JScript functions don't need to return anything`
	`function mysub( arg1) {`
	`   LogMessage( arg1 ) ;`
	`}`
	
	`// Call mysub—JScript always needs parentheses`
	`mysub( 45 );`

Both languages work with typeless variables and are interpreted rather than compiled. This makes them ideal for rapid development. JScript is case-sensitive, VBScript is not.

Arrays and Collections in VBScript and JScript

VBScript	JScript

`' accessing arrays`
`dim a(9)            ' array of size 10`
`a(0) = 1`
	`// accessing arrays`
	`var a = new Array(10); // array of size 10`
	`a[0] = 1;`

`' get size of 1 dimensional array`
`count = ubound(a) + 1`
	`// get size of array`
	`count = a.length;`

`' resizing array to size 11`
`redim preserve c(10)`
	`// resizing array`
	`aa = new Array(1,2,3);`
	`a = a.concat( a, aa );`

`' looping over array`
`' you can also use: for each e in a`
`for i=lbound(a) to ubound(a)`
`        e = a(i)`
`next`
	`// looping over array`
	`for ( i=0; i < a.length; i++ ) {`
	`        var e = a[i];`
	`}`

VBScript	JScript

' convert to comma delimited string
str = join(a, ",")
a = split(str, ",")   ' back to array
	// convert to comma delimited string
	var str = a.join(",");
	a = str.split(",");

' VBScript arrays are called "SAFEARRAYS"	// JScript uses Array Object, not SAFEARRAY
	// Some XSI API return SAFEARRAY but
	// it can be converted.
	// (uvprop is a ClusterProperty object)
	var safearray = uvprop.elements.array;
	var a = new VBArray(safearray).toArray();

' multi-dimensional array dim b(2,9)	// JScript only supports 1 dimensional arrays

' When dealing with lists of objects, XSI often
' uses Collection Objects rather than Safearrays
 	
' Get a ParameterCollection
set oParameters = oSphere.Parameters
 	
for each oParameter in oParameters
    logmessage oParameter.Name
next
	// When dealing with lists of objects, XSI often
	// uses Collection Objects
	
	// Get a ParameterCollection
	var oParameters = oSphere.Parameters ;
	
	for (i=0;i<oParameters.Count;i++) {
	    var oParameter = oParameters.Item(i) ;
	    logmessage(oParameter.Name) ;
	}

' You can also construct your own XSICollection
' object using the CreateObject() function
set oRandColl = CreateObject("XSI.Collection")
 	
' Then you can populate it however you want
oRandColl.Add Rnd()
	// In JScript, you use the new ActiveX()
	// constructor instead
	var oRandColl = new ActiveX("XSI.Collection");
	
	// or remove all items
	oRandColl.Clear();

' The Selection object is a kind of Collection
set myPods = Selection(0)
for each p in myPods
        logmessage p.Name
next
	// Even if there is only one object selected, you
	// still need to treat it like a Collection
	var myPod = Selection(0);

XSI Commands

Many interactive operations in XSI, for example changing a value on a Property Page or clicking on a button, will log a script in the **History** window of the **Script Editor**. It is easy to build your own scripts and custom commands based on a series of XSI commands.

Some of the most common commands include `GetValue`, `SetValue`, `CreatePrim`, `AddProp`, `DeleteObj`, `ApplyOp`, `SelectObj`, and `CopyPaste`. Each area of XSI (Modeling, Animation etc) has its own rich set of Commands.

For example, if an object named `mycylinder` is selected, and the menu item **Animate > Deform > Shape > Store Shape Key** is invoked, then the following line is logged in the Script History:

VBScript:

```
'INFO : StoreShapeKey "mycylinder", , siShapeLocalReferenceMode
```

JScript:

```
//INFO : StoreShapeKey("mycylinder", null, siShapeLocalReferenceMode);
```

`StoreShapeKey` is the name of the command that was executed. If you select this word in the **History** window and press `F1` then the documentation for this command is displayed. The documentation defines the purpose of the command and explains each **argument**.

In this case the first argument defines the object we want to store the shape on. In the documentation it mentions that the default value for this argument is the current selection. This means that if we had been sure to select `mycylinder` first, then we would have been able to call `StoreShapeKey` without specifying the first argument. This would look like this:

VBScript:

```
StoreShapeKey , , siShapeLocalReferenceMode
```

JScript:

```
StoreShapeKey( null, null, siShapeLocalReferenceMode );
```

Notice how, in VBScript, we leave the argument blank to invoke the command with default arguments, but in JScript it is necessary to say `null` (or `""`).

In the documentation it says that the second argument is the name of the created key. This is left unspecified so that the default naming scheme is used.

The word `siShapeLocalReferenceMode` used in the third argument is a constant with the value `0`. Constants are used to make the script code easier to read, and always start with the characters `si`. The identical command call, without using the constant, would look like this:

VBScript:

```
StoreShapeKey "mycylinder", , 0
```

JScript:

```
StoreShapeKey("mycylinder", null, 0);
```

Here is an example of how several commands can be grouped together to make a little script:

VBScript

```
' Create a grid, move it and change its color
NewScene
CreatePrim "Grid", "MeshSurface"
Translate , 0, 4, 0, _
    siAbsolute, siParent, siObj, siY
ApplyShader "Lambert"
SetValue "grid.Material.Lambert.diffuse.red", 1
```

Jscript

```
// Create a grid, move it and change its color
NewScene();
CreatePrim("Grid", "MeshSurface");
Translate(null, 0, 4, 0,
    SiAbsolute, siParent, siObj, siY );
ApplyShader("Lambert");
SetValue("grid.Material.Lambert.diffuse.red", 1, null);
```

You can use **variables**, **subroutines**, **loops** and other programming techniques to generalize a sequence of commands and create your own powerful tools. You can even communicate with the user during the course of your script, for example by asking him or her to **pick** an object.

VBScript

```
' Basic routine showing how to replace hardcoded
' object and preset names with arguments

sub CreatePrimWithShader( primtype, shadertype )
    CreatePrim primtype, "meshsurface"
    ApplyShader shadertype
end sub

' Call our routine twice
CreatePrimWithShader "Sphere", "Lambert"
CreatePrimWithShader "Grid", "Phong"
```

JScript

```
// Basic routine showing how to replace hardcoded
// object and preset names with arguments

function CreatePrimWithShader( primtype, shadertype )
{
    CreatePrim( primtype, "meshsurface" ) ;
    ApplyShader( shadertype ) ;
}

// Call our function twice
CreatePrimWithShader( "Sphere", "Lambert" );
CreatePrimWithShader( "Grid", "Phong" );
```

Many commands return values which can be stored in variables when you call them; for example, the `IsAnimated` command returns a **Boolean** value, indicating whether or not the specified object has animated parameters. You can store the value directly in a variable like this:

VBScript

```
' Test a sphere and print a message indicating
' the result
```

JScript

```
// Test a sphere and print a message indicating
// the result
```

VBScript	JScript

`bFCurves = IsAnimated( "sphere", siFcurveSource )`
 	
`if bFCurves then`
`    logmessage "Sphere is animated"`
`else`
`    logmessage "Sphere is NOT animated"`
`end if`
	`bFCurves = IsAnimated( "sphere", siFcurveSource );`
	
	`if (bFCurves) {`
	`    logmessage( "Sphere is animated" );`
	`} else {`
	`    logmessage( "Sphere is NOT animated" );`
	`}`

Note: Some commands return objects that have been created by the command. Dealing with objects returned from functions is described in the next section.

Returning Objects from Commands

Many commands return the objects they create or manipulate. For example, `CreatePrim` returns the newly-created object; `ApplyOp` returns a collection of the newly-created operators; and `SIFilter` returns the filtered list of elements.

When we assign the output of a command to a scripting variable, it then refers (or points) to the value. So if the return value is an object, we could call this an **object reference** or **object pointer**. Object references generally have a corresponding object (class) type in the Object Model.

Note: Even though variables have no explicit type in VBScript and JScript, you can still test object references to see what kind of object type they contain using the XSI `ClassName` function. Similarly, we can use `TypeName` in VBScript or `typeof` in JScript to test what type of data (boolean, integer, string, etc.) the variable contains.

VBScript:
```
set oBox = CreatePrim( "cube", "meshsurface" )
logmessage oBox & " = " & ClassName( oBox )

'INFO : "cube = X3DObject"
```
JScript:
```
var oBox = CreatePrim( "cube", "meshsurface" );
logmessage( oBox + " = " + ClassName( oBox ) );

//INFO : "cube = X3DObject"
```

The crucial point is to recognize the difference between representing an object by its string name and an object reference. The trick in VBScript is to remember the *set* keyword; without it, you will be saving another type of item in your variable:

Note: JScript does not have this kind of ambiguity because it has no equivalent to the `set` keyword.

VBScript:
```
' By forgetting the set keyword, we are actually grabbing a string instead, which will
' produce an error when we try to use the ClassName() test
oBox = CreatePrim( "cube", "meshsurface" )
```

```
logmessage oBox & " = " & TypeName( oBox )
'INFO : "cube = String"

logmessage oBox & " = " & ClassName( oBox )
'ERROR : "Type mismatch: 'ClassName' - [line 5]"
```

Since we can get object references returned from commands, the `GetValue` command becomes very interesting, because we can use it to convert string expressions to a corresponding object:

VBScript:

```
set oRtn = GetValue( "Passes.Default_Pass" )
logmessage oRtn & " = " & ClassName(oRtn)

'INFO : "Scene.Passes.Default_Pass = Pass"
```

JScript:

```
var oRtn = GetValue( "Passes.Default_Pass" );
logmessage( oRtn + " = " + ClassName(oRtn) )

//INFO : "Scene.Passes.Default_Pass = Pass"
```

Once we have an object pointer, we can use any method or property available on that object (class). This concept is explained in detail in the next section.

XSI Object Model

The XSI **Object Model** exposes the XSI scene as a hierarchy of objects. Each object represents part of the scene and supports **Properties** and **Methods**. Scripts use the Object Model by getting access to objects in the scene and calling these methods and properties to get information or change the scene. Although the XSI object model is built with **Object Oriented** techniques like interfaces and inheritance, it is not necessary when writing scripts to have a deep understanding of these concepts.

Properties and Methods

A **property** is the scripting terminology for an **attribute** of an object. If you are familiar with C++ you can think of these as public member variables. Many properties allow both read and write access to the underlying value, but some only allow read access. The `Value` and `Default` properties of the Parameter object are demonstrated in the following example:

VBScript	JScript

```
OParameter.Value = 5

if ( oParameter.Value <> oParameter.Default
) then
   ' restore the default value
   oParameter.Value = oParameter.Default
endif
```
	```
	OParameter.Value = 5 ;

	if ( oParameter.Value != oParameter.Default
	) {
	   // restore the default value
	   oParameter.Value = oParameter.Default ;
	}
	```

A **method** is usually associated with an **action** that you want to perform on the object:

VBScript	JScript

```
OSceneRoot.AddLight "LightSpot"
OPolygonMesh.AddVertexColor
```
	```
	oSceneRoot.AddLight("LightSpot");
	oPolygonMesh.AddVertexColor();
	```

There are many different types of objects in the Object Model and most correspond directly to the different types of objects used in XSI. For example `Shader`, `Operator`, `Particle`, `Null`, `Model`, and `Fcurve`. These different types of objects define properties and methods which make sense for the particular object. For example, the `Fcurve` object has an `AddKey` method and the `Particle` object has a `Mass` property.

However certain properties and methods apply to many objects. For example, because practically every object has a name of some sort, it is desirable to make

the `Name` property available on each object. These common properties and methods are organized into interfaces, and each Object Model object will support (via inheritance) one or more of these interfaces. For example, the `SIObject` interface is supported on practically every object, and includes properties like `Name`, `Parent`, `Type` and the method `IsEqualTo`.
Some important interfaces that are supported on many types of objects are:

Interface Name	**Description**
`SIObject`	Basic information like `Name`, `Parent`, `Type`
`X3Dobject`	Corresponds to a 3D object, for example a `Primitive`, `Camera`, `Null`, `Light`, or `ParticleCloud`
`Property`	XSI properties, such as Materials, Custom Properties and StaticKinematicState are attached to an `X3DObject` and establish additional attributes and behavior for the object. Most of the state of a `Property` is exposed through its Parameters. XSI properties should not be confused with the **properties** of an object in the programming sense (as mentioned above).
`SceneItem`	Supported by objects that can have `Property` objects attached to it, of which `X3DObject` is the most common example.
`Parameter`	Corresponds to a parameter on an object. Parameters are often basic types like **String**s, **Integer**s and **Float**s, but can also be compound types like **Color** and **Vector**. For example the Phong shader has many parameters, including a color parameter called `Ambient`.
`ProjectIte m`	Supported by objects that can have `Parameters`. This includes both `X3DObjects` and `Properties`.

Object References
Before we can call any of the properties or methods of an XSI object you need to get a scripting variable that represents that object. As with assigning an object to the return value of a command (which we saw in the Returning Objects from Commands section), we could call this an **object reference** or **object pointer**.

Note: The object references that you get from return values are actually members of the XSI Object Model which means that you have access to all the methods and properties available to that object.

Again, the crucial point is to recognize the difference between representing an object by its string name and an object reference. The trick in VBScript is to remember the `set` keyword:

VBScript	JScript

`' sphere is a reference to the sphere object`
`set sphere = Dictionary.GetObject( "sphere" )`
	`// JScript always behaves as if the VBScript "set"`
	`// statement was specified`
	`var sphere = Dictionary.GetObject( "sphere" );`

`' This will print the name of the sphere`
`logmessage sphere.Name`
	`// This will print the name of the sphere`
	`logmessage( sphere.Name );`

`' We can test the type of a variable by using the`
`' VBScript TypeName() function, which returns`
`' either the standard data type (eg., String,`
`' Boolean, Float, etc.) or the XSI object type`
`' (eg., X3Dobject, Camera, Property, etc.)`
 	
`' This will print out "X3DObject"`
`logmessage TypeName( sphere )`
 	
`' This will print out "String" because the Name`
`' property always contains a String data type`
`logmessage TypeName( sphere.Name )`
	`// typeof is similar but not the same to VBScript`
	`// TypeName: typeof returns the data type of the`
	`// item if it corresponds to a native JScript type`
	`// but it returns "Object" if the item is an`
	`// ActiveX object (ie., any XSI object).`
	
	`// This will print out "Object"`
	`logmessage( typeof( sphere ) );`
	
	`// This will print out "String"`
	`logmessage( typeof( sphere.Name ) );`

`' This will also print out "X3DObject"`
`logmessage ClassName( sphere )`
	`// If you need to test which object you have, use`
	`// the Application.ClassName() method`
	
	`// This will print out "X3DObject"`
	`logmessage( ClassName( sphere ) );`

`' if "set" is not specified then`
`' sphere becomes the string "sphere"`
`sphere = Dictionary.GetObject( "sphere" )`
 	
`' This will be a syntax error (not an object)`
`logmessage sphere.Name`
	`// JScript does not have this kind of ambiguity,`
	`// since it decides whether the variable will be`
	`// an object reference or a string, etc.`

Note: To help improve the readability of scripts, it is a common convention to prefix object reference variables with the letter **o**. For example, `oSphere = Dictionary.GetObject( "sphere" )`.

One convenient, but potentially confusing, aspect of object references is that they can act as if they are strings. This is because the `Name` property is the **default property** of all Object Model objects so if no property or method is referenced XSI calls the `Name` property, which returns a string:

VBScript

```
set oSphere = Dictionary.GetObject( "sphere" )

' This will print out "sphere" ...
logmessage oSphere

' ... because it is equivalent to
logmessage oSphere.Name
```

JScript

```
var oSphere = Dictionary.GetObject( "sphere" );

// This will print out "sphere" ...
logmessage( oSphere );

// ... because it is equivalent to
logmessage( oSphere.Name );
```

We have shown how an object reference will automatically convert to a string. The Dictionary.GetObject() method permits going the opposite direction. Once this conversion process is understood it is possible to use the string name of an object and object references almost interchangeably.

Traversing the Graph

XSI objects are organized in a hierarchy of objects that corresponds closely to what is visible in the **Scene Explorer**. The objects have **Parent** and **Children** relationships between each other. Using these relationships it is possible to **traverse** the scene graph. Once you become comfortable with traversing the scene graph you will be close to mastering the XSI SDK.

VBScript

```
' Use the dictionary to reach the parameter
set oParameter = Dictionary.GetObject( _
                     "sphere.kine.global.posx" )

' Or use the object model to traverse to the
' parameter
set oSphere = Dictionary.GetObject( "sphere" )
set oKinematics = oSphere.Kinematics
set oGlobalKinematics = oKinematics.Global
set oParam = oGlobalKinematics.Parameters("posx")

' Or collapse all these calls together
set oParam = Dictionary.GetObject( "sphere" ). _
              Kinematics.Global.Parameters("posx" )
```

JScript

```
// Use the dictionary to reach the parameter
var oParameter = Dictionary.GetObject(
                      "sphere.kine.global.posx" );

// Or use the object model to traverse to the
// parameter
var oSphere = Dictionary.GetObject( "sphere" );
var oKinematics = oSphere.Kinematics;
var oGlobalKinematics = oKinematics.Global;
oParam = oGlobalKinematics.Parameters( "posx" );

// Or collapse all these calls together
oParam = Dictionary.GetObject( "sphere" ).
             Kinematics.Global.Parameters("posx");
```

In simple cases `Dictionary.GetObject` (as well as the `GetValue` and `SetValue` commands) are easier to use and require less typing. However they require that you know exactly the names of the specific object you want to access. The object model is very good for dealing with scenes with unknown or dynamic content and hence powerful for writing flexible tools. It can also be significantly faster.

Practically all relationships between objects in the scene graph can all be generalized as either a parent or child relationship. When considering the Scene

Explorer we would say that the nodes nested underneath a particular node are its **children**, and the node immediately above it is its **parent**.

Reaching the **parent** of a node is easy, because practically every object supports the `SIObject.Parent` property. Some objects may actually have multiple parents, such as a shared material, in which case the same object appears at multiple places in the scene explorer and in the graph. In this case all the "parents" can be reached via the `ProjectItem.Owners` property. Because there can be more than one this property returns a collection.

Reaching **children** nodes is also easy, but the property to access depends on the type of the child. For example, to get the parameters of a `ProjectItem` we access the `ProjectItem.Parameters` property, which returns a `ParameterCollection`.

The following table shows some of the Object Model properties which are used to reach children:

Object.Property	**Type of Child**
`ProjectItem.Parameters`	`Parameter`
`X3Dobject.Children`	`X3DObjects` nested under a X3DObject
`X3Dobject.Models`	`Models` nested under a X3DObject
`SceneItem.Properties`	`Properties`
`Primitive.ConstructionHistory`	`Operators` acting on the Primitive
`Geometry.Clusters`	`Clusters`
`X3Dobject.ActivePrimitive`	`Primitive` that creates the shape of the X3DObject
`SceneItem.Material`	`Material`
`Material.Shaders`	`Shaders` connected directly to the Material
`ParticleCloud.Particles`	`Particles`
`Parameter.Source`	`FCurve`, `Shader` or other source driving a parameter's value
`PolygonMesh.Vertices`	`Vertices`

The following examples show some of these properties in use:

VBScript	**JScript**

`' Look for property objects under the Scene Root`
`for each oProperty in ActiveSceneRoot.Properties`
`        logmessage "Found Property: " _`
	`// Look for property objects under the Scene Root`
	`oEnum = new Enumerator(`
	`                        ActiveSceneRoot.Properties`

VBScript	JScript
`& oProperty.Name` / `next`	`);` / `for (;!oEnum.atEnd();oEnum.moveNext())` / `{` / `logmessage( "Found Property: " + oEnum.item().Name );` / `}`
see below	see below

VBScript

```
                                        & oProperty.Name
next
```

JScript

```
                          );
for (;!oEnum.atEnd();oEnum.moveNext())
{
        logmessage( "Found Property: "
                        + oEnum.item().Name );
}
```

VBScript

```
' Find point clusters on a geometry
set oSphere = Selection(0)
set oClusterCollection =
oSphere.ActivePrimitive.Geometry.Clusters
for each oCluster in oClusterCollection
        if ( oCluster.Type = "pnt" ) then
                logmessage "Found a point cluster" _
                                & " with size " & _
                        oCluster.Elements.Count
        end if
next
```

JScript

```
// Find point clusters on a geometry
var oSphere = Selection(0);
var oClusterCollection =
oSphere.ActivePrimitive.Geometry.Clusters
for( i = 0 ; i < oClusterCollection.Count ; i++ ) {
        var oCluster = oClusterCollection.Item(i) ;
        if ( oCluster.Type == "pnt" ){
                logmessage( "Found a point cluster
                                with size " +
                        oCluster.Elements.Count ) ;
        }
}
```

Where to find more information

The SDK White Paper is a good introduction to the capabilities and features of the XSI SDK.

XSI ships with extensive SDK documentation. This includes:

- the *Scripting User Guide*
- the *Scripting Reference Guide*
- the *Plug-in Integration Guide*

For examples and tutorials check out http://www.softimage.com/Education/Xsi/ and http://www.softimage.com/xsinet.

There is also a large online community with examples and discussions available from sites like:

- http://www.xsibase.com/
- http://www.highend3d.com/
- http://www.edharriss.com/
- http://www.xsifiles.com/

Softimage also offers SDK Support contracts, including a members-only forum, extensive library of examples and technical support. For more information please visit the Partners section on www.softimage.com.

To learn more about the Object Model be sure to try out the **Info OM** Net View page that is part of XSI Net.

For information about the scripting languages there is a great deal of free information available on the internet. Here are a few sites to get you started:

- http://msdn.microsoft.com/scripting
- http://www.devguru.com
- http://www.winscripter.com/
- http://www.win32scripting.com/
- http://www.activestate.com/
- http://www.perl.org/
- http://www.perl.com/
- http://www.python.org/