
XSI Scripting Basics

Herman Tulleken (herman@luma.co.za)

February 26, 2007

Contents

1	Introduction	2
2	More XSI Objects	2
2.1	Materials	2
2.2	Lights	3
2.3	Cameras	3
2.4	Exercises	5
3	Scripted Operators	6
3.1	Exercises	7

1 Introduction

In this section we will look at creating and manipulating materials, cameras and lights through scripting. You will also be introduced to scripted operators—a powerful method of scripting the behaviour of objects.

2 More XSI Objects

2.1 Materials

Geometry that can have materials applied to them have the method `AddMaterial` with which this can be done. The method takes three arguments: the first is a string denoting the preset (such as `'Phong'`, `'Lambert'`), the second is a *branch flag*, a boolean indicating whether the material should be applied to all nodes in a branch, and the last is the name of the material. The method returns the material object.

The following example creates a sphere, and adds a simple Phong material to it. Note that the material is stored in a variable so that we can alter its other properties.

```
root = Application.ActiveSceneRoot
sphere = root.addGeometry('Sphere', 'NurbsSurface', 'redSphere')
material sphere.AddMaterial('Phong', false, 'redPhong')
```

After we have added the material, we would like to change the properties. These properties are located deep in the object hierarchy, and we will use variables and a helper function to reduce the amount of typing:

```
def setRGB(materialProperty, red, blue, green):
    materialProperty.red = red
    materialProperty.green = green
    materialProperty.blue = blue

root = Application.ActiveSceneRoot
sphere = root.addGeometry('Sphere', 'NurbsSurface', 'redSphere')
materialLib = sphere.AddMaterial('Phong', False, 'redPhong')
phong = materialLib.Surface.NestedObjects['redPhong']

ambient = phong.ambient
setRGB(ambient, 1.0, 0.0, 0.0)

diffuse = phong.diffuse
setRGB(diffuse, 1.0, 0.0, 0.0)

specular = phong.specular
setRGB(specular, 1.0, 1.0, 1.0)
```

Notice that the actual Phong is a element of `NestedObjects`, which is an property of the `Surface` object. The `NestedObjects` is a collection. Like lists, we can use integer indices to access elements. However, this is not always a safe approach: in cases where we cannot control an object's position in a collection, we cannot predict with complete accuracy where that object will lie. For this reason the `NestedObjects` collection emulates a dictionary.

dictionary: A *dictionary* is a collection that can be indexed using arbitrary objects, not just integers. In this case, the collection is indexed with strings—the (XSI) names of the objects in the collection. Other XSI collections also emulate dictionaries in this way, allowing us not to be concerned about an object’s position in a collection.

2.2 Lights

Lights are added to a scene much like geometry is, except that we use the `AddLight` method. This method takes three arguments: the first is the name of the preset we want (eg. `'Point'` or `'Neon'`); the second is a boolean indicating whether an interest should be added; the final argument is the XSI name of our light.

The following example sets up three lights around a sphere:

```
def setPos(obj, x, y, z):
    obj.posX = x
    obj.posY = y
    obj.posZ = z

def setLightIntensity(light, intensity):
    light.LightShader.NestedObjects['soft_light'].intensity = intensity

def addThreePointLights(root, x, y, z, r):
    light = root.AddLight('Point', False, 'MainLight')
    setPos(light, x + r, y + r, z + r)
    setLightIntensity(light, 1.0)

    light = root.AddLight('Point', False, 'AmbientLight')
    setPos(light, x - r, y - r, z + r)
    setLightIntensity(light, 0.2)

    light.SpecularContribution = False

    light = root.AddLight('Point', False, 'RimLight')
    setPos(light, x-r, y + 1.5*r, z-r)
    light.SpecularContribution = False

root = Application.ActiveSceneRoot
addThreePointLights(root, 0, 0, 0, 5)
root.addGeometry('sphere', 'NurbsSurface')
```

Note that we can access the `SpecularContribution` property of a light directly (through a short-cut), but to set the intensity we have to navigate through various objects to get to it. Notice again how we used a string to access the right element in the `NestedObjects` property. To see which objects we need to access for a certain property, we have to use the SDK explorer tree. It can be tricky, but you will soon get used to it.

2.3 Cameras

Since cameras are handled similarly to other XSI objects, we won’t say much about them here. The example below is a Python translation of the example

that comes with the XSI SDK documentation. This example contains a few things (not specific to cameras) that we have not mentioned before; they are explained below.

```
# Python example demonstrating how to access all
# Cameras in the scene.

from win32com.client import constants

def printCameraInfo(camera, i):
    print '-----Camera' + str(i) + '-----'
    print 'Camera Name: ' + camera.Name
    print 'Camera Root: ' + camera.Parent.Name
    print 'Camera Primitive: ' + camera.ActivePrimitive.Name

    #The Interest a 'sibling' of the Camera so
    #we find it via the parent
    childrenOfParent = camera.Parent.Children

    interest = None

    for obj in childrenOfParent:
        if obj.type == 'CameraInterest':
            interest = obj
            break

    print 'Camera Interest: ' + interest.Name

    #Many aspects of the camera can be manipulated by reading
    #and writing the parameters on these various objects

    if camera.ActivePrimitive.Parameters['proj'].Value == 1:
        print 'Projection: Perspective'
    else:
        print 'Projection: Orthographic'

    print 'Field of View: ' + str(camera.fov.Value)

    #Although these parameters actual belong
    #to the local kinematic property,
    #these shortforms are available to python
    #to make it even more convenient

    print 'Interest Pos(local): (' \
          + str(interest.PosX.Value) + \
          ',' + str(interest.PosY.Value) + \
          ',' + str(interest.posZ.Value) + ')

    globalKine = interest.Kinematics.Global
    print 'Interest Pos(global): (' \
          + str(globalKine.PosX.Value) + \
          ',' + str(globalKine.posY.Value) + \
```

```

        ',' + str(globalKine.posZ.Value) + ')')

#Create a sample scene with different cameras
Application.NewScene('', False)

root = Application.ActiveSceneRoot

root.AddCamera('Default_Camera')
root.AddCamera('Telephoto', 'Telephoto')

#Put the orthographic camera inside a model
model = root.AddModel()
model.AddCamera('Orthographic', 'Ortho')

#Hide one of the cameras underneath a cone
cone = root.AddGeometry('Cone', 'MeshSurface')
cone.AddCamera('Wide_Angle', 'WideAngle')

root.AddCamera('Wide_Angle', 'WideAngle')

#Perform a recursive search for all the cameras in the scene
cameras = root.FindChildren('', constants.siCameraFilter )

#Print out some information about the Cameras in the scene
print 'The scene contains ' + str(len(cameras)) + ' cameras'

for i, camera in enumerate(cameras):
    printCameraInfo(camera, i)

```

In the example above, you might be tempted to replace

```
... str(interest.PosX.value)
```

with something shorter:

```
... str(interest.PosX)
```

Unfortunately, the code above will not behave as expected. Remember that `PosX` parameter only *emulates* a numeric type, and only in certain contexts. In the above context, `PosX` behaves more like the complicated type it really is, and the `str` function converts it to the parameter *name*, which is `CameraInterest.kine.global.posz`.

If you code in VBScript, you can access XSI constants directly. If you use Python, however, you must import the `constants` object (from `win32com.client`), and access all constants through this object. XSI constants are usually prefixed with `si`, which make them easy to spot in the documentation and code snippets written in other languages.

2.4 Exercises

1. Implement a function that will add a Phong to an object that will make it transparent. The amount of transparency must be a parameter of the function.

2. Implement a function that will invert an object's ambient and diffuse colors. Your function must do this to *all* materials in the object's material library. Assume they are all Phongs.
3. Modify the function above to work with other material types as well. For each material in an object's material library, check which type of material it is, and change the appropriate values.
4. Implement a function that will replace all the Phong materials with suitable toon shaders. (Use the diffuse color for the toon shader's color).
5. Implement a function that will replace 1 light with three copies. Give the copies a slight random offset, and change their intensities to a third of the original light. Two of the copies must have specular and shadows disabled.
6. Implement a function that will find all the lights in an object, and make them slightly dimmer. (Hint: use the `FindChildren` method).
7. Implement a function that will create a strobe light (i.e. it flashes on and off). The number of frames between flashes must be a parameter of this function.
8. Implement a function that adds a camera to an object, and animate it in a circle around the root.
9. Implement a function that allows a user to select a point in space, and an object of interest. Add a camera at the point in space, and parent the camera's interest to the selected object.

3 Scripted Operators

scripted operator: A *scripted operator* is a function that controls a parameter of an object. This function might take parameters of other objects as its arguments. It is called automatically by XSI, so that the parameter you are trying to control is always up to date.

Because scripted operators are called automatically, they must be of a specific form. It always takes at least two arguments. The first is a *context* parameter, giving you some information about where the function was called from. The second parameter is the output argument. This parameter's `Value` attribute will be the value of your parameter; you should assign this with whichever value you want your parameter to be. If you want your scripted operator to take more arguments, you have to add them explicitly with the *New...* button. The explorer tree will pop up, and allow you to select the parameters you want directly. This alone makes scripted operators easier to use than normal scripts. Note that the parameters do not emulate numeric types in this context—you have to extract the `Value` explicitly using the `Value` attribute.

The following example is a scripted operator on a cylinder with unit radius. If the cylinder is rotated 90 degrees about the *x*-axis and parented to a cube, it will roll with the cube as it is translated in the *x* direction. Note that the first line is provided to emphasise that we are working with a Python

function; it is generated automatically, and appears just above the editing region.

```
def Update(In_UpdateContext, Out, Inposx):  
    Out.Value = -Inposx.Value * 360.0 / (2 * 3.14)
```

3.1 Exercises

1. Implement a scripted operator to control the intensity of a light. The intensity of the light must be proportional to the squared distance of the light to an object (any object you choose). (Hint: the squared distance between two points is given by $(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$).
2. Implement three scripted operators that will make an object (the predator) move in the direction of another object (the prey). Animate the prey, and see what the predator does! (Hint: Increment PosX if the prey's PosX is larger than the predators PosX, otherwise decrement it. Do the same for the PosY and PosZ).

Index

branch flag, 2

constants, 5

dictionary, 3

expression, 6

scripted operator, 6