

Conquering the Code in Softimage

Jared Glass

Version 1.1

Introduction

Hi there 😊

This Guide will cover Basic to Advanced Scripting, Programming, Plugin Development, Class Creation and much more. Including lots of handy titbits and explanations, like how to successfully use the Softimage help files. The goal being to equip you with the knowledge and tools you need to create awesome scripts, plugins and even external tools that speed up your workflow and make you a super programming genius!!! Which all results in your life being made a whole lot easier and giving you more time to spend on the artistic side of things, or you could even move over completely and work as a super programming genius. Either way this book puts the power in your hands 😊

Hope you enjoy!

Terms of Use:

This book is free for personal use.

If this book is to be used to educate or as part as any form of education, a donation towards the author is required. The amount is subject to the educator or payee's discretion provided it exceeds the value of one US dollar. See the Donations section at the end of this book.

This book may be published if the publisher has obtained written consent from the author and is subject to an agreement on a gross profit share between the author and the publisher.

Legal:

Copyright © 2012 by Jared Glass

No part of this publication may be altered or reproduced in any form except as permitted by the Berne Convention and the Universal Copyright Convention, without either the prior written permission of the Author. Reproduction is subject to the above mentioned "Terms of Use" section.

Limit of Liability/Disclaimer of Warranty: The author makes no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales, promotional or free materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the author is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. The author shall not be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

All trademarks are the property of their respective owners.

The author has taken care in the preparation of this book, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Acknowledgements

Even though this is a short book I would like to thank Sandy Sutherland and Simon Anderson for getting me started and mentoring me on my journey through the more technical side of Softimage. I'd also like to mention a compositing HOD I worked under, Dave Clark, who was the first to recognize and support my interest and talent for the programming side of CG.

And a big thanks to everyone I've worked with and received help from over the years, if I were to mention you all by name I'd be able to write another book but you know who you are. It's been a working, learning, teaching and experiencing the ups and downs of the industry with you all...mostly :p

Contents

The Lessons are split up into sections:

1. [Recruit](#) – Introduction to Scripting
2. [Trooper](#) – The simple programming side of scripting
3. [Hardened](#) – Basic Programming and Plugins
4. [Veteran](#) – Classes, Modules, Threads and PyQt for Softimage.

And the Sections are made up of the following lessons:

Recruit:

1. [Easily turn echoed commands into buttons.](#) 6
2. [Applying a command to multiple objects.](#) 8

Trooper:

1. [Finding Parameters \(Introduction to the “Object Model”\).](#) 10
2. [Finding How to Access Parameters Visually.](#) 12
3. [Finding parameters and their values with code.](#) 13
4. [Successfully using the help files.](#) 15

Hardened:

1. [Creating a Plugin.](#) 18
2. [Adding a Plugin to a Menu.](#) 20
3. [Plugins that take Input and give Output](#) 23
4. [On Event Plugins](#) 28
5. [What is PyQt and how does it make my plugins AWESOME?!](#) 32
6. [Installing PyQt for Softimage](#) 33
7. [Creating a Simple PyQt Plugin](#) 34

Veteran:

1. [Modules](#) 44
2. [Multi-Model Importer](#) 47

Recruit:

Easily turn echoed commands into buttons.

Firstly you need to have your script editor open (Alt+4) or use the icon .

For this tutorial we're only going to be interested in the bottom half of the script editor (the "output"). This is the section of the script editor that shows the basic commands resulting from you clicking and editing things in Softimage.

Its often cluttered with all the commands that have been run so right-click on it and from the popup menu choose "Clear All". This will make it easier for us to see exactly what code needs to go into our button.

In my case I want to freeze the modeling on all of the objects in my scene. So, I select all the objects in my scene (with Ctrl+A) and then using the Freeze M button .

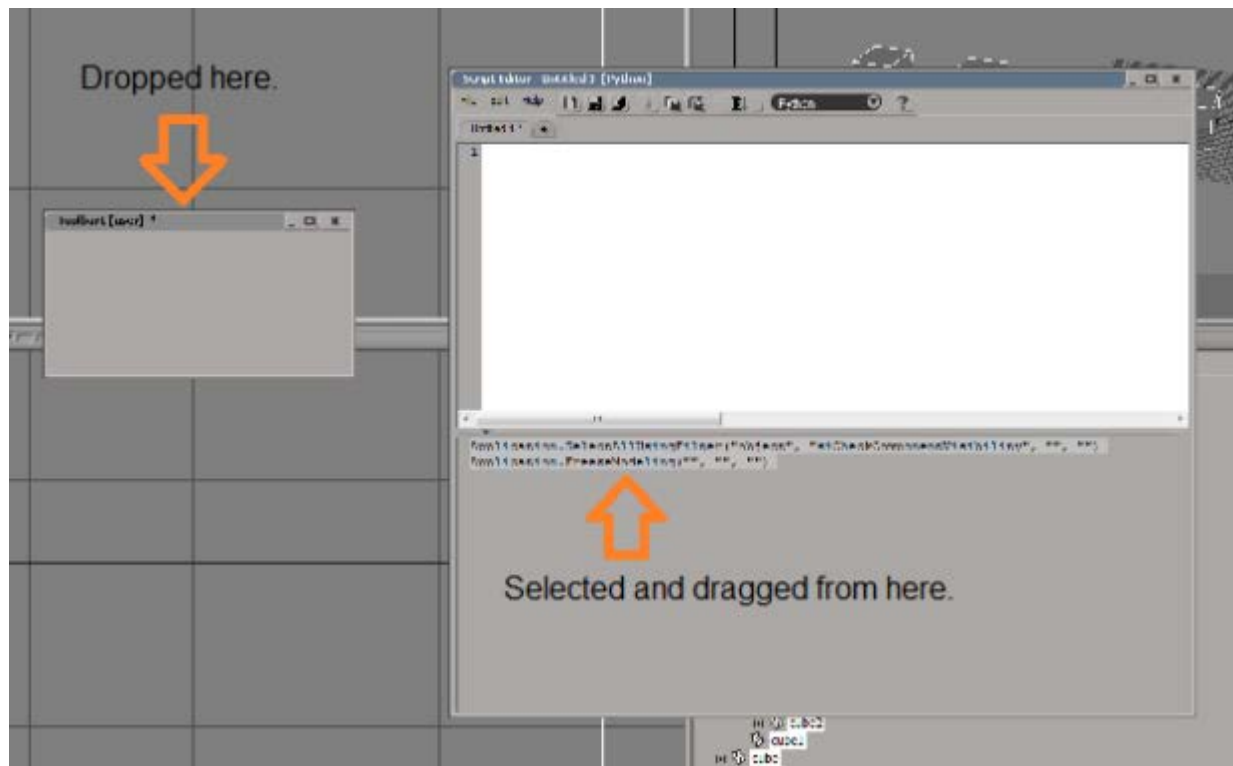
The script editor echoes the commands:

```
Application.SelectAllUsingFilter("object", "siCheckComponentVisibility", "",  
")  
Application.FreezeModeling("", "", "")
```

This is what I need to add to my button. But I need a "Toolbar" to add the button to first.

You can either add a button to an existing toolbar or you can make a new toolbar (View > New Custom Toolbar).

Then all I need to do is to select the commands from and drag them onto the toolbar.



Once you let go Softimage will ask you which type of button you wish to create, leave “Script Button” checked and press OK. This brings up the button’s settings. You’ll notice your code in the top “Script Commands” section. Give it a name (something descriptive, its not much fun having a toolbar filled with buttons called button1, button2, button3 etc). I also reccomending giving the button a tooltip (description of what the does).

And thats it! You’ve just made your very own custom button 😊

Before you close Softimage and start bragging to your friends about the awesome button you’ve just made, you might want to save the toolbar so you can use the button next time you open Softimage. To do this just right-click the toolbar and choose save. Give your toolbar a name and after you’ve clicked OK a file browser will open asking you where you want to save your toolbar. Don’t change anything, just press OK again and your toolbar will be saved to your personal Softimage folder filled with all your preferences and other things.

Applying a command to multiple objects

We start off in pretty much the same manner as Part 1 (Easily turn echoed commands into buttons.) by opening the script editor (Alt+4) and clearing its previous output.

In this example I have a few cubes in my scene that I want to be able to hide and show with the click of a button. This is really easy to do 😊

I haven't changed the names of the cubes, they are called "cube1", "cube2" etc.

So now same as in Part 1 I manually do what I want and then copy the echoed commands.

So, first I select one cube. Then I press the "h" key to toggle its visibility.

The script editor echoes the commands:

```
Application.SelectObj("cube1", "", True)
Application.ToggleVisibility("", "", "")
```

This is great but if I were to put this into a button it would only toggle the visibility of "cube1", I want to be able to do All the objects who's names start with "cube".

This is done with this little guy: * (an asterix or wildcard). This guy basically tells Softimage to look for anything who's name starts with "cube", regardless of what it ends with.

So when I run the commands

```
Application.SelectObj("cube*", "", True)
Application.ToggleVisibility("", "", "")
```

All the cubes in the scene are hidden, and when i run it again they are unhidden. When you want to edit a command, copy it from the bottom half of the script editor ("output") to the top half ("editor"). In here you can change the commands as you please and test them too by pressing F5 or clicking the run button .

When you are happy with your code, select it and drag it onto a toolbar (you can turn code in from both the output or the editor section of the script editor, though in this case we are dragging from the editor section).

This asterix trick can be applied to many Softimage commands, here are a few more examples that would work:

To assign a material to all the cubes, after assigning manually the command in the script editor output is:

```
Application.AssignMaterial("Sources.Materials.DefaultLib.MyMaterial,cube1",
```

```
"siLetLocalMaterialsOverlap")
```

so I change it to:

```
Application.AssignMaterial("Sources.Materials.DefaultLib.MyMaterial,cube*",  
"siLetLocalMaterialsOverlap")
```

If I want to turn off the render visibility for all the cubes, manually I get:

```
Application.SetValue("cube1.visibility.rendvis", False, "")
```

which I change to:

```
Application.SetValue("cube*.visibility.rendvis", False, "")
```

Trooper:

Finding Parameters (Introduction to the “Object Model”)

In Softimage, with scripting we have 4 different ways of getting the value of a parameter.

Say I have a sphere and I want find its X position. I can do any of the following:

```
print Application.GetValue("sphere.kine.global.posx")
print Application.Dictionary.GetObject("sphere.kine.local.posx").Value
print
Application.ActiveSceneRoot.FindChild("sphere").Kinematics.Global.PosX.Value
```

You did ready correctly, I did say there are 4 ways though I’ve only shown 3. The fourth way is by using the FindObjects method. The reason I’ve left it out is it is a bit more complex to use than the others and in most cases you can get by by using one of the mentioned 3. But when you’re more comfortable coding in Softimage look it up in the SDK Guide.

The first method is by far the easiest. All you have to do to find what to put inbetween the brackets of the Application.GetValue(). We can do this by changing the paramater we want in the viewport, seeing what the Script Editor Output says and changing it slightly. E.g: When I move my sphere along the X axis the Script Editor Outputs:

```
Application.SetValue("sphere.kine.global.posx", 3.435, "")
```

Now if I want to print the X Position of the sphere all I need to do is change the “SetValue” to “GetValue” and take out anything other than the sphere’s path:

```
Application.GetValue("sphere.kine.global.posx")
```

The first method is great for its ease but if I wanted to set the values of a few parameters on the sphere it would be quicker and easier to read in a script if we went with method 2 or 3. The reason for this is we can “get” the sphere and store it in a variable so we can access it whenever we want:

```
my_sphere = Application.Dictionary.GetObject("sphere")
or
my_sphere = Application.ActiveSceneRoot.FindChild("sphere")
```

The difference between method 2 and method 3 is that method 2 goes straight to the sphere object in memory and returns it. This method is by far the fastest of the three, Method 3 is the

slowest because Softimage does a search for an object named “sphere”. It searches under the Scene_Root, looking at the name of every object until it finds one called “sphere”. For now we will go with the speedy method 2 but method 3 will come in quite handy later on.

So, back to the code. We now have the sphere object stored in the variable called “my_sphere”. This means we can set whatever parameter we wish quite easily. For example

Set sphere’s X position to 100:

```
my_sphere.Kinematics.Global.PosX.Value = 100
```

Set sphere’s Y Rotation to 20:

```
my_sphere.Kinematics.Global.RotY.Value = 20
```

Get the sphere’s Z Scale, divide it by two and store it in a variable called “half_scale” and finally set the sphere’s scale Z to our half_scale_z value:

```
scale_z = my_sphere.Kinematics.Global.SclZ.Value
```

```
half_scale_z = scale_z/2
```

```
my_sphere.Kinematics.Global.SclZ.Value = half_scale_z
```

Note: If you use Application.Dictionary.GetObject in your scene and the object you’re looking for doesn’t exist (or if you’re not sure) add a comma followed by False before your closing bracket. This tells Softimage NOT to throw an error if it can’t find the object. eg: If I want to try get an object named “cube” in my scene but I’m not sure if it exists in my scene or not I run

```
my_cube = Application.Dictionary.GetObject("cube", False) # This won't throw an error if the object named "cube" doesn't exist.
```

So I always write my Application.Dictionary.GetObject with a False at the end just in case 😊

Finding How to Access Parameters Visually

In the last post we found the X Position value of a sphere with:

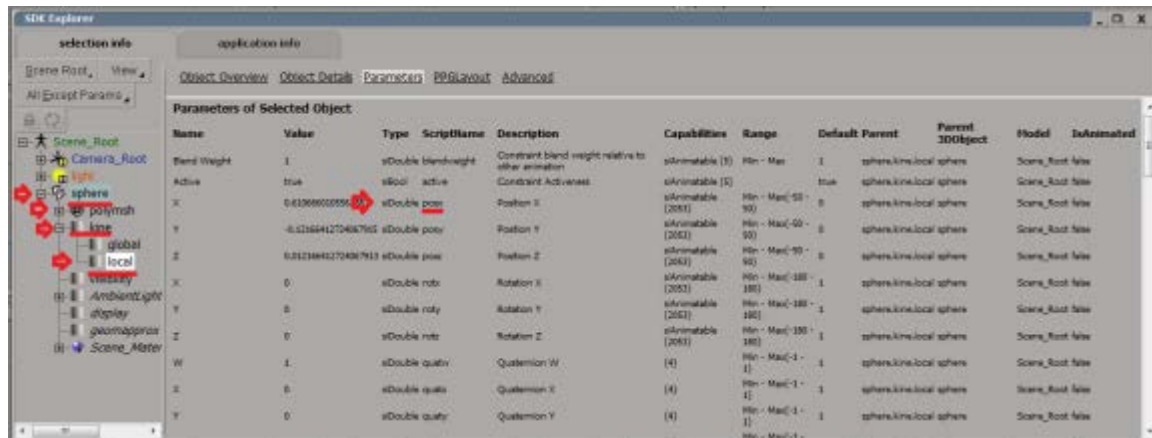
```
print Application.Dictionary.GetObject("sphere.kine.local.posx", False).Value
```

but how do you find the path to the X Position without copying it straight out of the Script Editor?

Lucky for us Softimage has a very useful tool called the “SDK Explorer”. You can find this under the View menu >> Scripting >> SDK Explorer (or Ctrl+Shift+4).

This neat little tool is like the scene explorer except for the fact that it shows you all the information of the object you have selected.

Also you can change the View in the SDK Explorer to “Use Script Names”. This changes the names of the objects in the explorer section to match their script names so you can see exactly how to access a parameter with your code.



In the above image you can see exactly where we get the “sphere.kine.local.posx” from. Each new part of the path to the parameter is separated by a “.” (dot). So from here we can see if we wanted the Z Scale we just need to change the path to “sphere.kine.local.scalez”.

I can also get other data from the SDK Explorer. If I click in the sphere and change the SDK Explorer’s tab from “Parameters” to “Object details” a whole bunch of useful information is presented like whether the object is animated or not (“IsNodeAnimated”) and much more. This will be handy later on when we need to find objects by their type (i.e find all the lights in the scene).

If I select a light in my scene I can see the “Type” in the SDK Explorer (Near the bottom of the “Object Details”) is just called “light”. So if I want to print all the names of the various lights in my scene I just have to run:

```
for light_obj in Application.ActiveSceneRoot.FindChildren("*", "light"):  
    print light_obj.Name
```

What this code is doing is firstly navigating to the “Scene_Root” (Application.ActiveSceneRoot gets the active “Scene_Root” model). We could also do this with Application.Dictionary.GetObject(“Scene_Root”, False). Just like how we got the sphere, though this way is a bit less to type.

Then it’s looking for all the children under the Scene_Root that have the name “*” (remember the asterix means match everything) and are the Type “light”.

Finding parameters and their values with code

In some cases it can be easier and faster to find an object's parameters and their values via code as opposed to having to look through the SDK Explorer (though it is very likely you will often use a combination of both).

As we already know, to gain access to a object we can do this:

```
sphere_obj = Application.Dictionary.GetObject("sphere", False)
```

From here we can either look for the parameters in the SDK Explorer or we can “loop” through them with code:

```
for param in sphere_obj.Parameters:  
    print param.FullName
```

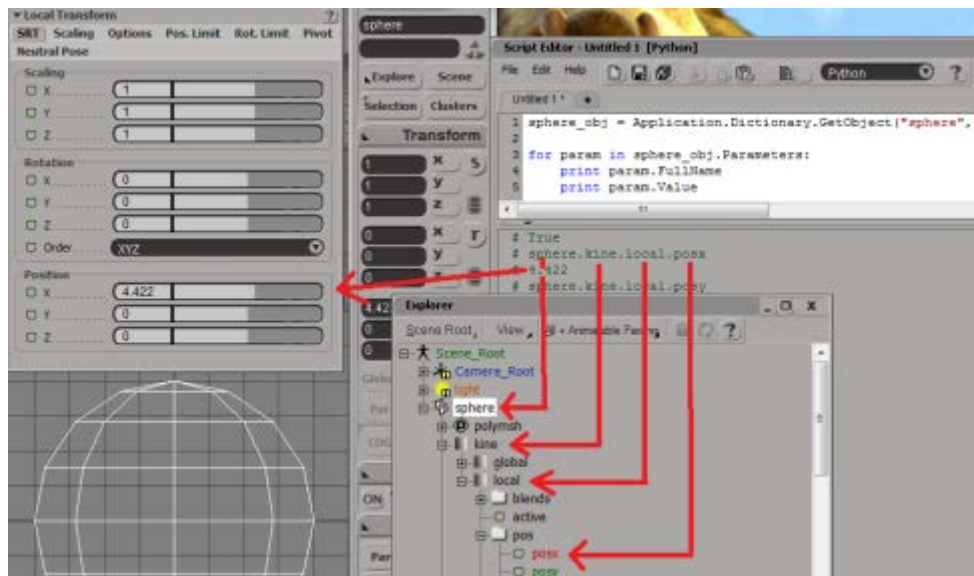
This will print the full path (FullName) to each parameter.

If we wanted the values too we could just print param.Value:

```
for param in sphere_obj.Parameters:  
    print "Parameter:" + param.FullName  
    print "Value:" + str(param.Value)
```

Now we can see exactly how to access each parameter and just to make sure its the right one we can check the value we see in Softimage (eg the Local X Position) compared to the value we see in the Script Editor Output (eg sphere.kine.local.posx)

The parameter's path is made up of a combination of Objects followed by Properties and the finally the parameter's ScriptName.



In this case “sphere” is an object and “kine” and “local” are Properties.

An Object holds a collection of Properties and a Property holds a collection of Parameters.

You can loop through Properties just as you looped through Parameters with:

```
for prop in sphere_obj.Properties:
    print "Property:" + prop.FullName
```

Now that we know the Properties we can even loop through the Parameters of a specific Property:

```
for param in sphere_obj.Properties("visibility").Parameters:
    print "Parameter:" + param.ScriptName
```

From this we can narrow it down even further to:

```
print sphere_obj.Properties("visibility").Parameters("viewplaybackvis").Value
```

If there are other objects parented under the sphere and we only want to get those we can use:

```
for obj in sphere_obj.Children:
    print obj.FullName
```

which can be turned into (assuming you have an object named cone parented under your sphere_obj):

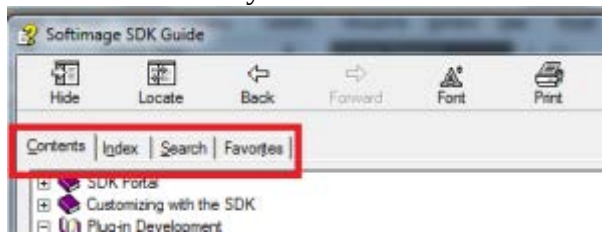
```
for prop in sphere_obj.Children("cone").Properties:
    print "Property:" + prop.FullName
```

And lastly we can call `sphere_obj.NestedObjects`. This returns any of the above mentioned things that are directly under the `sphere_obj`. I tend to use this as more of a last resort when calling `Children`, `Properties` or `Parameters` won't work.

Successfully using the help files

In Softimage you get two main help files or "Guides". If you click on the help menu in or press F1 in Softimage the "User's Guide" will open. This can be very useful but not for these tutorials. The one we want is called the "SDK Guide" and is just under the User's Guide in the help menu. The other way you can access the SDK Guide is by pressing F1 with the Softimage Script Editor as your active window (you can make it active by clicking on it).

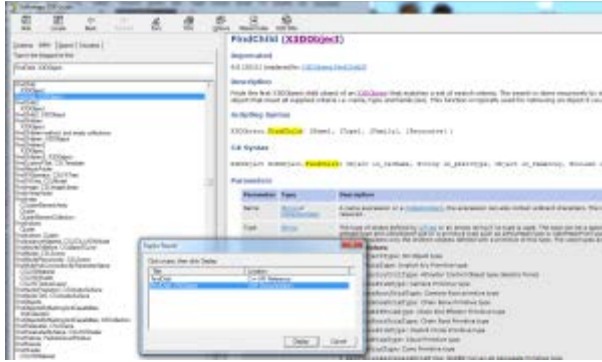
In the SDK Guide you will see four tabs:



Out of the four tabs I only really ever use "Index" and occasionally "Contents". The Contents tab is packed with tutorials and is great to use for reference and to learn a new section of Softimage. I recommend at least taking a brief look through Contents just to see all the possibilities there are and if you have enough time I recommend you go through the whole thing, Starting with "Script Development", then moving to "Plug-in Development" and after that you'll know enough to be decide what you want to do next.

The Search tab I don't use because the Index tab is just a LOT quicker and easier to find what I'm looking for.

In a previous tutorial we used `Application.ActiveSceneRoot.FindChild` to find an object under the "Scene_Root". If I type in "FindChild" into the search box under the Index tab I get quite a few options.



From here I choose “FindChild, X3DObject” but when I double click it to show the page I get a popup asking me if I want the C++ or SDK version. While using Python, VBScript or JavaScript we will always choose SDK. Then I click Display and the SDK Guide shows the FindChild Function along with the possible arguments (inputs).

So say I have 10 cubes in my scene, each one parented under another object and I don’t know any of their names. All I know is that there only one cube directly under the Scene Root and that is the one I want to get. According to the SDK Guide I can set the “Recursive” argument to False. If set to False Softimage says the FindChild function will only look directly under the Scene Root for my Object and not under any of the Scene Root’s children.

Now to only get cubes I can specify a Type. In the section that says Type, for its description there is a bit of blue text “siType” with a line underneath. Just like web links clicking on these will take you to a different page. To find the correct argument for a cube, click the siType link.

For now we only need to use the right side of the page is displayed.



So now that we have all the arguments we need we’re ready to go:

As we can see the FindChild can take up to four arguments: “X3DObject.FindChild([Name], [Type], [Family], [Recursive])”

for [Name] we will use “*” – to find all names since we don’t know what the name could be. [Type] = “cube”

[Family] doesn’t really matter for this case

and finally [Recursive] needs to be set to False.

Making our command look like:

```
cube_obj = Application.ActiveSceneRoot.FindChild("*", "cube", "", False)
```

Note that for [Family] we used “”. The “” just tells Softimage to use the default value.

Also when setting arguments, you have to set them in the correct order.

```
Running: cube_obj = Application.ActiveSceneRoot.FindChild("*", "cube", False)
```

would not have the same effect. Softimage will assume that you are setting the [Family] to False as [Family] is the option for the third argument. This is why even though we don't want to change or set the [Family] argument, we still need to put something in there to be able to set the fourth argument correctly.

The help file may be a bit overwhelming now but most are at first and as time goes on you'll get more comfortable with it. With this help file and google you'll be able to achieve many marvelous things 😊

Hardened:

Creating a Plugin

Creating plugins or to be more specific – a “Plugin Command” is really easy in Softimage. All you need to do is go to your Plugin Manager and choose to Create a new plugin for you and it sets it all up and even opens your new plugin in the script editor for you.

So, click File >> Plug-in Manager. In the plugin Manager click the “File” button. Choose “New” and then “Command”.

Softimage will now ask us to set some options for the plugin. The first but is the Command Name. This is what we will call when running the plugin. E.g for Softimage’s command: `Application.DeselectAll()`, the “DeselectAll” is the Command Name.

For this exercise I’m going to call the plugin “LogSelection” and what I will make this plugin do is to log (Softimage’s version of print) the names of all the objects I have selected in my scene.

Then when I hit enter the “Plug-in Name” also updates.

Make sure the “Coding Language” is set to Python.

For now I’m going to leave the “Output Directory” as is but if I wanted to save this plugin to an Softimage Workgroup all I’d have to do is change the path from:

`C:\Users\firstname.lastname\Autodesk\Softimage_Subscription_Advantage_Pack_SP1\Application\Plugins`

to

`\\server\workgroups_folder\workgroup\Application\Plugins`

And lastly I click “Generate Code”.

Now the Script Editor pops up with my brand spanking new plugin. Any code that I want the plugin to run I just put under the “LogSelection_Execute()” function. This function is what is run when we call “`Application.LogSelection()`”

So now I can start coding away:

```
for obj in Application.Selection:
    Application.LogMessage(obj.FullName)
```

so the Execute part of my plugin looks like this:

```
17  is_req.Name = 1
18  is_req.Name = 0
19
20  is_req.RegisterCommand("LogSelection", "LogSelection")
21  #RegisterCommand(is_req) - do not remove this line
22
23  return true
24
25  def LogSelection_Execute( is_req ):
26      is_req.Name = is_req.Name
27      Application.LogMessage(str(is_req.Name) + str(" has been selected."), context, is_req)
28      return true
29
30  def LogSelection_Init( is_req ):
31      is_req = is_req.Name
32      is_req.Description = ""
33      is_req.ReturnValue = true
34      return true
35
36  def LogSelection_Execute( ):
37      Application.LogMessage("LogSelection_Execute called", context, is_req)
38
39      The obj is Application.Selection
40      Application.LogMessage(obj.FullName)
41
42      return true
43
44
```



Now all we need to do is test it.

For this I'm going to close and re-open my Script Editor to get the bottom bit (The Output Section) as that is where the obj.FullName is getting logged to.

In a new tab of the Script Editor, type and run:

```
Application.LogSelection()
```

to run your command. Just make sure you have some objects selected or it won't log anything.

So, we've made a plugin and it's working but I don't like that it's just logging the names without giving us more info. If we had just run a few other commands that log data we wouldn't know which command the object names came from and might confuse them. So for this I'm going to re-open my plugin and change the line:

```
Application.LogMessage(obj.FullName)
```

to

```
Application.LogMessage("LogSelection: " + obj.FullName)
```

All we need to do now is save the plugin and Softimage will automatically reload it for us. So this time when we run `Application.LogSelection()` we'll get something like:

```
# INFO : LogSelection: Camera_Root
# INFO : LogSelection: light
# INFO : LogSelection: sphere
```

Adding a Plugin to a Menu

To add a plugin to a menu we have two options.

Option 1: When Creating the Plugin.

Softimage has the option to set the menu you want the plugin to show up in when creating the plugin as we did in the previous tutorial.

But when I try change the “Add to Menu” option, there are no other options to select. So for now we will just go with option two.

But don’t close the SDK_Wizards window just yet. Click the question mark next to the “Add to Menu” option. The following window will popup with various menu options.



Keep this open for **Option 2**.

Option 2: Adding a few lines to an Existing Plugin.

For this all we do is to manually add the code that **Option 1** would have done (had it worked >.<).

The code to add the plugin to the menu is as follows:

```
# Under the XSILoadPlugin function, after the RegisterCommand line
    in_reg.RegisterMenu( [Menu Number], [Plugin Name followed by "_Menu"] )

# And then a function
def [Plugin Name]_Menu_Init( in_ctxt ):
    oMenu = in_ctxt.Source
    oMenu.AddCommandItem( [Command Title in Menu], [Command as text] )
    return True
```

The menu number we get from the page that opened in the SDK Guide by clicking on the question mark next to the “Add to menu” option on the SDK_Wizards window, which I told you to keep open. So say I want to add my command to Softimage’s main File Menu.

This is where a bit of digging needs to be done but I managed to find it without too much trouble:

siMenu/MenuCreateID	401	ICE = Create menu (Simulate toolbar).
siMenu/MenuBuildID	402	ICE = Build menu (Simulate toolbar).
siMenu/MainFileMenuID	1000	Same section as the File menu on the main menu bar.
siMenu/MainFileProjectID	1001	Project section in the File menu on the main menu bar.
siMenu/MainFileExportID	1002	File > Export menu on the main menu bar.
siMenu/MainFileImportID	1003	File > Import menu on the main menu bar.
siMenu/MainWindowID	1004	Windows menu on the main menu bar.

So now that I know the File Menu's number is 1000 I can add it to my plugin.

If I modify the LogSelection plugin I made in the previous my code looks like:

```
# LogSelectionPlugin
# Initial code generated by Softimage SDK Wizard
# Executed Mon Jul 16 18:03:37 UTC+0200 2012 by jared.glass
#
# Tip: To add a command to this plug-in, right-click in the
# script editor and choose Tools > Add Command.
import win32com.client
from win32com.client import constants

null = None
false = 0
true = 1

def XSILoadPlugin( in_reg ):
    in_reg.Author = "jared.glass"
    in_reg.Name = "LogSelectionPlugin"
    in_reg.Major = 1
    in_reg.Minor = 0

    in_reg.RegisterCommand("LogSelection","LogSelection")
    in_reg.RegisterMenu(1000, "LogSelection_Menu", False, False)
    #RegistrationInsertionPoint - do not remove this line

    return true

def XSIUnloadPlugin( in_reg ):
    strPluginName = in_reg.Name
    Application.LogMessage(str(strPluginName) + str(" has been
unloaded."),constants.siVerbose)

    return true

def LogSelection_Init( in_ctxt ):
    oCmd = in_ctxt.Source
    oCmd.Description = ""
    oCmd.ReturnValue = true

    return true

def LogSelection_Menu_Init( in_ctxt ):
    oMenu = in_ctxt.Source
    oMenu.AddCommandItem("Log Selected Object's Names","LogSelection")
```

```

return true

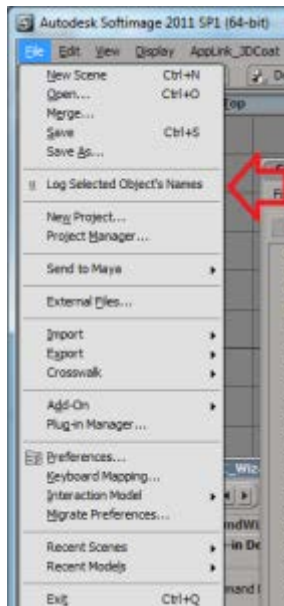
def LogSelection_Execute( ):
    Application.LogMessage("LogSelection_Execute called", constants.siVerbose)

    for obj in Application.Selection:
        Application.LogMessage("SELECTED: " + obj.FullName)

    return true

```

and so when I save my plugin, it appears in the File Menu, woohoo!



P.S. Perhaps you noticed that I added two False arguments to the line

```
in_reg.RegisterMenu(1000, "LogSelection_Menu", False, False)
```

To see why, copy the code “RegisterMenu” and paste it in the text box of the Index tab of the SDK Guide 😊

Plugins that take Input and give Output.

For this section I'm going to start backwards.

Output:

To set a plugin to give Output is really, really easy. All we need to do is change the line at the end of the plugin's "_Execute()" function from:

```
return true
```

to

```
return [Whatever we want]
```

We can return a string, an object, a list, a list of objects and pretty much every other python data structure we could ever want.

The reason I may want a plugin to give output is if I have a scenario where I need to do something, and some of the code is already in the plugin but the same plugin is also used by other tools so I can't change the actual way the plugin works or other tools may break.

So for this example what I want to do is to select all the polymeshes in my scene except any called "cube". Now I already have a plugin that selects all the polymeshes in my scene (called SelectAll_PolyMeshes). So all I need to do is to get that list and see if there are any objects called "cube" in it and if so, remove them from the selection. Simple 😊

First I need to set my SelectAll_PolyMeshes command to output (return) the list of polymeshes so the plugin looks like:

```
# SelectAll_PolyMeshesPlugin
# Initial code generated by Softimage SDK Wizard
# Executed Thu Jul 19 18:18:51 UTC+0200 2012 by jared.glass
#
# Tip: To add a command to this plug-in, right-click in the
# script editor and choose Tools > Add Command.
import win32com.client
from win32com.client import constants

null = None
false = 0
true = 1

def XSILoadPlugin( in_reg ):
    in_reg.Author = "jared.glass"
    in_reg.Name = "SelectAll_PolyMeshesPlugin"
    in_reg.Major = 1
    in_reg.Minor = 0
```

```

    in_reg.RegisterCommand("SelectAll_PolyMeshes","SelectAll_PolyMeshes")
    #RegistrationInsertionPoint - do not remove this line

    return true

def XSIUnloadPlugin( in_reg ):
    strPluginName = in_reg.Name
    Application.LogMessage(str(strPluginName) + str(" has been
unloaded."),constants.siVerbose)
    return true

def SelectAll_PolyMeshes_Init( in_ctxt ):
    oCmd = in_ctxt.Source
    oCmd.Description = ""
    oCmd.ReturnValue = true

    return true

def SelectAll_PolyMeshes_Execute( ):

    Application.LogMessage("SelectAll_PolyMeshes_Execute
called",constants.siVerbose)

    # Find all the polymeshes in the scene and store them in a python list.
    polymesh_lst = list(Application.ActiveSceneRoot.FindChildren("*",
"polymsh"))

    # Create a list to store the meshes names in
    polymeshName_lst= []
    # Loop through the mesh objects and store their names
    for polymesh in polymesh_lst:
        # Add the meshes name to the mesh name list
        polymeshName_lst.append(polymesh.FullName)

    # Join the names in the list into a string and sepperate each name by a
comma. eg "sphere, cone, cube"
    polymeshSel_str = ", ".join(polymeshName_lst)

    # Tell Xsi to select the names in the polymeshSel_str
    Application.Selection.SetAsText(polymeshSel_str)

    # The return that we will change
    return true

```

Now, we want the list of polymesh objects to check for “cube” in the name and then to remove the object from the selection. All we need to do is to change the

```

return true

```

right at the end to

```

return polymesh_lst

```

After you’ve added that save the “SelectAll_PolyMeshes” plugin to update it in Softimage.

now if we:

```
print Application.SelectAll_PolyMeshes()
```

we should get something that looks like:

```
# [<COMObject <unknown>>, <COMObject <unknown>>, <COMObject <unknown>>, <COMObject <unknown>>]
```

That's the list of Softimage objects.

From that I can do

```
polymesh_lst = Application.SelectAll_PolyMeshes()
```

instead of printing it so I can actually store the data in a list.

From here the rest is cake. So in the end my code looks like this :

```
# Get the list of polymeshes from the plugin.
polymesh_lst = Application.SelectAll_PolyMeshes()
# Loop through the polymeshes and check the names.
for polymesh in polymesh_lst:
    # Check if the polymesh's name is cube
    if polymesh.name == "cube":
        # If it is, remove it from the selection
        Application.Selection.Remove(polymesh)
```

Input:

(Also called Arguments)

Setting up a plugin to take input is just as easy, it just requires a few more lines of code to be replaced. For this example we're going to change the "SelectAll_PolyMeshes" Plugin, but in such a way that it won't change the way it works with any of the other tools that may use it (like the one we just made that removes cube from the selection). This is a lot easier than it sounds. So we're going to set the plugin to take in part of an object's name, and only if that part of the name is in a PolyMesh's name, does the PolyMesh get selected.

So we will be able to call the command `Application.SelectAll_PolyMeshes()`

like `Application.SelectAll_PolyMeshes("part_of_some_object_name")`

First thing we need to do is to add a variable to the plugin's "Init" (The Init is run when the plugin is initially compiled by Softimage – When you launch Softimage or Reload the plugin by Saving it from the Script Editor). This variable will tell Softimage that the plugin takes some input.

So in the “SelectAll_PolyMeshes_Init” we add:

```
oArgs = oCmd.Arguments
oArgs.Add([Argument Name], constants.siArgumentInput, [Default Value])
```

I’m going to call the argument “part_of_name” and the default will be None.

This means that if something just calls Application.SelectAll_PolyMeshes(), part_of_name will be set to None and I can tell the plugin to ignore the part of name search and run as it had before my changes.

Next bit is to pass the argument (part_of_name) into the _Execute function. This will allow us to use part_of_name inside the _Execute.

After this I can edit the actual code within the _Execute as I please:

```
# Loop through the mesh objects and store their names
for polymesh in polymesh_lst:
    # If the part_of_name was specified and therefore is not the default
    None...
    if not part_of_name == None:
        # If the part_of_name is in the object's name...
        if part_of_name in polymesh.Name:
            # Add the meshes name to the mesh name list
            polymeshName_lst.append(polymesh.FullName)
        else:
            # Add the meshes name to the mesh name list
            polymeshName_lst.append(polymesh.FullName)
```

So my entire plugin looks like:

```
# SelectAll_PolyMeshesPlugin
# Initial code generated by Softimage SDK Wizard
# Executed Thu Jul 19 18:18:51 UTC+0200 2012 by jared.glass
#
# Tip: To add a command to this plug-in, right-click in the
# script editor and choose Tools > Add Command.
import win32com.client
from win32com.client import constants

null = None
false = 0
true = 1

def XSILoadPlugin( in_reg ):
    in_reg.Author = "jared.glass"
    in_reg.Name = "SelectAll_PolyMeshesPlugin"
    in_reg.Major = 1
    in_reg.Minor = 0

    in_reg.RegisterCommand("SelectAll_PolyMeshes", "SelectAll_PolyMeshes")
    #RegistrationInsertionPoint - do not remove this line
```

```

return true

def XSIUnloadPlugin( in_reg ):
    strPluginName = in_reg.Name
    Application.LogMessage(str(strPluginName) + str(" has been
unloaded."), constants.siVerbose)
    return true

def SelectAll_PolyMeshes_Init( in_ctxt ):
    oCmd = in_ctxt.Source
    oCmd.Description = ""
    oCmd.ReturnValue = true

    oArgs = oCmd.Arguments
    oArgs.Add("part_of_name", constants.siArgumentInput, None)
    return true

def SelectAll_PolyMeshes_Execute( part_of_name ):

    Application.LogMessage("SelectAll_PolyMeshes_Execute
called", constants.siVerbose)

    # Find all the polymeshes in the scene and store them in a python list.
    polymesh_lst = list(Application.ActiveSceneRoot.FindChildren("*",
"polymsh"))

    # Create a list to store the meshes names in
    polymeshName_lst= []
    # Loop through the mesh objects and store their names
    for polymesh in polymesh_lst:
        # If the part_of_name was specified and therefore is not the default
None...
        if not part_of_name == None:
            # If the part_of_name is in the object's name...
            if part_of_name in polymesh.Name:
                # Add the meshes name to the mesh name list
                polymeshName_lst.append(polymesh.FullName)
            else:
                # Add the meshes name to the mesh name list
                polymeshName_lst.append(polymesh.FullName)

    # Join the names in the list into a string and sepperate each name by a
comma. eg "sphere, cone, cube"
    polymeshSel_str = ", ".join(polymeshName_lst)

    # Tell Xsi to select the names in the polymeshSel_str
    Application.Selection.SetAsText(polymeshSel_str)

    return polymeshName_lst

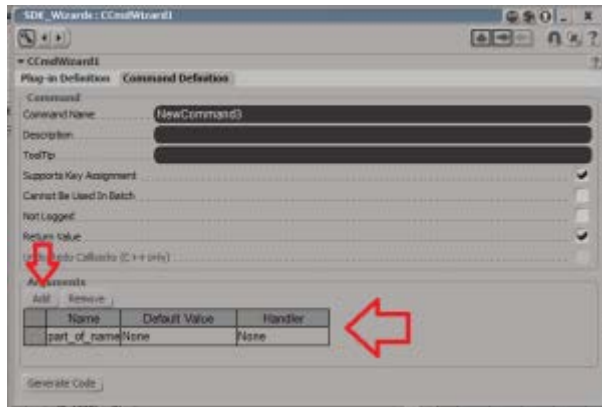
```

And if I want to select ALL the PolyMeshes I run `Application.SelectAll_PolyMeshes()`

but if I only want to select PolyMeshes with say “sphere” in the name, I run `Application.SelectAll_PolyMeshes("sphere")`. This sets the `part_of_name` variable from None to "sphere".

Note:

When creating a plugin you also get the option to add input arguments:



I do use this but I find I end up changing existing plugins that don't take arguments (or that I want to take even more arguments) much more than I use this tool to create them.

On Event Plugins

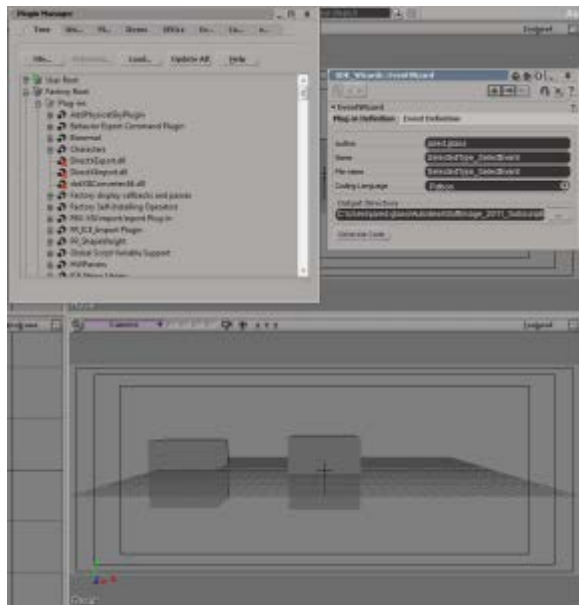
If you have worked much with any programming language you should be familiar with Triggers/Events/Callbacks or one of the other many names for them. If not, don't worry; Events in Softimage are really easy. Basically an On Event plugin is run when a certain event takes place. So if we create a "siOnSelectionChange" Event, the plugin will be run every time change what is selected i.e. by selecting something else.

So to illustrate this I'm going to create two cubes, a Polygon Mesh cube and a Surface cube. If I set my Viewport to display Shaded, the two cubes look exactly the same. What I want is a quick way to tell what "Type" the object I have selected is so I can easily tell if it's a Polygon Mesh, Surface or whatever else. For this we want our plugin to check what Type the selection is every time we change what we have selected, so the "siOnSelectionChange" Event will be perfect for this.

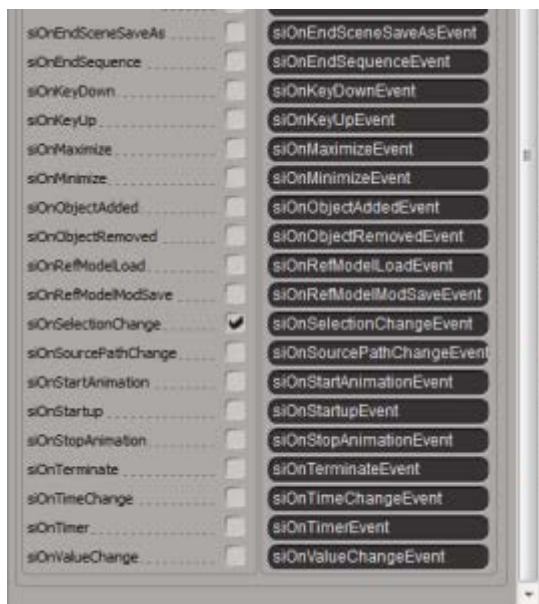
To create an On Event plugin you start the same way as every other plugin.

Click File >> Plug-in Manager >> File >> New >> Event

Then we setup the plugin. I've set the name to "SelectedType_SelectionEvent" so that just by looking at the plugin name I get an idea of what it does.



After that click on the “Event Definition” tab to select the event you want. Near the bottom you’ll find the “siOnSelectionChange” Event.



Once you’ve checked that, scroll to the top and hit the “Generate Code” button.

To get the selected object’s Type we can do something like:

```
# Get a the selected objects and store in a list called "selected_lst"
selected_lst = Application.Selection

# If no objects are selected
if len(selected_lst) == 0:
    Application.LogMessage("No Selection")
```

```

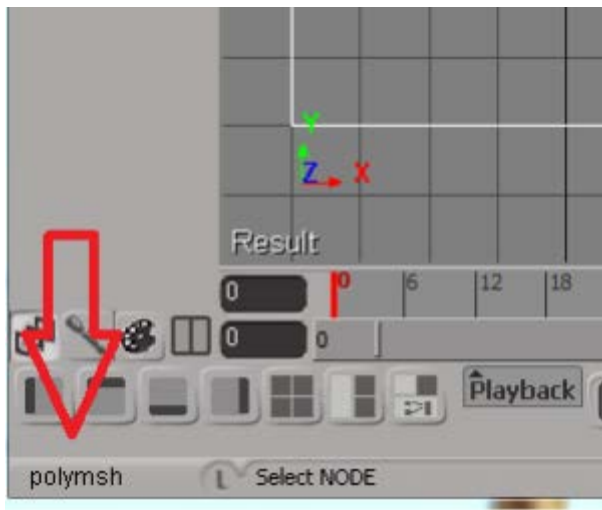
# If there is more than one object selected
elif len(selected_lst) > 1:
    Application.LogMessage("Multi Selection")

# Otherwise there must only be one object selected
else:
    # Note if I want to get the first item in a list I use my_list[0] and not
    my_list[1]
    # A list's order actually goes [0, 1, 2, 3]
    # So if I have a list ["banana", "apple", "pear", "grape"], "pear" is at
    position [2]
    selected_obj = selected_lst[0]
    # Log the type of the selected object
    Application.LogMessage(selected_obj.Type)

```

And just like with the commands, all we need to do is add our code to the plugin's "_Execute" function and then save the plugin.

So now when I select an object, the plugin tells me what I've got selected:



And as soon as I select something else or even if I select nothing the plugin will detect the selection has changed and Log what ever we told it to.

The entire plugin looks like this:

```

# SelectedType_SelectedEvent
# Initial code generated by Softimage SDK Wizard
# Executed Wed Aug 1 17:57:30 UTC+0200 2012 by jared.glass
#
# Tip: To add a command to this plug-in, right-click in the
# script editor and choose Tools > Add Command.
import win32com.client
from win32com.client import constants

```

```

null = None
false = 0
true = 1

def XSIloadPlugin( in_reg ):
    in_reg.Author = "jared.glass"
    in_reg.Name = "SelectedType_SelectionEvent"
    in_reg.Major = 1
    in_reg.Minor = 0

    in_reg.RegisterEvent("siOnSelectionChangeEvent", constants.siOnSelectionCh
ange)
    #RegistrationInsertionPoint - do not remove this line

    return true

def XSIUnloadPlugin( in_reg ):
    strPluginName = in_reg.Name
    Application.LogMessage(str(strPluginName) + str(" has been
unloaded."), constants.siVerbose)
    return true
    # Callback for the siOnSelectionChangeEvent event.

def siOnSelectionChangeEvent_OnEvent( in_ctxt ):
    Application.LogMessage("siOnSelectionChangeEvent_OnEvent
called", constants.siVerbose)

    Application.LogMessage("ChangeType: " +
str(in_ctxt.GetAttribute("ChangeType")), constants.siVerbose)

    # Get a the selected objects and store in a list called "selected_lst"
    selected_lst = Application.Selection

    # If no objects are selected
    if len(selected_lst) == 0:
        Application.LogMessage("No Selection")

    # If there is more than one object selected
    elif len(selected_lst) > 1:
        Application.LogMessage("Multi Selection")

    # Otherwise there must only be one object selected
    else:
        # Note if I want to get the first item in a list I use my_list[0] and
not my_list[1]
        # A list's order actually goes [0, 1, 2, 3]
        # So if I have a list ["banana", "apple", "pear", "grape"], "pear" is
at position [2]
        selected_obj = selected_lst[0]
        # Log the type of the selected object
        Application.LogMessage(selected_obj.Type)

    # Return value is ignored as this event can not be aborted.
    return true

```

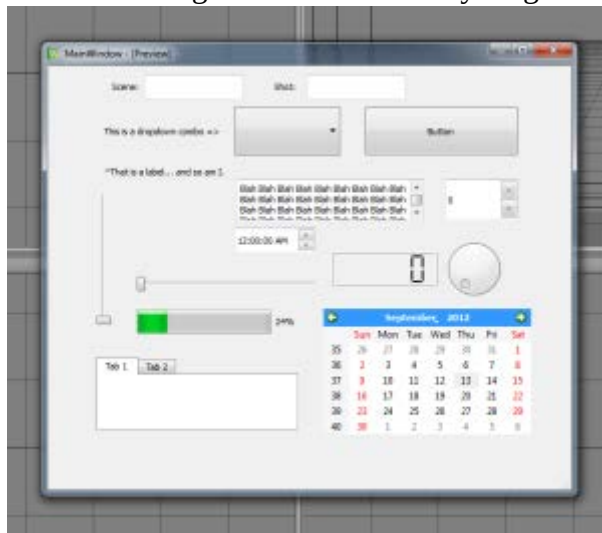
What is PyQt and how does it make my plugins AWESOME?!

GUIs (Graphical User Interfaces) add a world of functionality to your code but most importantly make the tools you create user friendly.

Softimage provides its own GUI system called PPGs (Property Pages). Starting off though these can be a nightmare and are quite limited in what they can do for you. As an artist dabbling in code they may seem pretty cool but as a programmer dabbling in Softimage they are very limited and the logic is quite different from what the average coder should be used to. Because of this I'm going to completely ignore PPGs and instead go through the way more powerful world of PyQt.

PyQt is just a set of modules that create “links” between python commands and your operating system's. All this means is that when you create a Window with PyQt on windows, your window will look like a windows window and not the standard grey Softimage windows (Property Pages). On mac the PyQt windows will look like Mac windows etc.

Below is a rough idea of the visually magnificent things you can create in PyQt:



I'm currently working on Windows and will go through the setup for windows but it is possible to get this working on Mac and Linux.

The PyQt plugin for Softimage (which I will cover later) has only been compiled for the 64bit version of windows. So if you're on Linux or Mac you have two choices: Either switch over to Windows x64 OR go looking online for help on compiling a version and if you're lucky a compiled version for your operating system.

Once PyQt and the PyQt Plugin for Softimage are installed on your system all the code you write will be the same, whether on Windows, Mac or Linux. This is the beauty of Python and PyQt. You only need to write a bit of code once and it works on all three operating systems!!!

But enough talk. Lets get started!

Installing PyQt for Softimage

This is a lot easier than you may think. All you need is the PyQt install and the PyQt for Softimage Plugin.

Before we start though, you need to download and install [pywin32](#) (for windows users only). Its a straight forward install, just click next till it's done. Pywin just adds some extra functionality to the windows version of python that is needed to keep things running smoothly.

Now, you can download PyQt from [Riverbank website](#).

If you're on Mac or Linux you're going to have to compile PyQt yourself, but for Windows users you simply have to download one of the installers, run it and viola; all done!

The current version of Softimage is using Python 2.6 and as I mentioned in the previous post, the PyQt Plugin for Softimage is x64 so you'll need to download [PyQt-Py2.6-x64-gpl-4.9.4-1.exe](#). The important bits in PyQt-Py2.6-x64-gpl-4.9.4-1.exe show that the **python version is 2.6** and **system type is x64**. So if you use a version later than 4.9.4 it should still work as long as you download the one where those two are right.

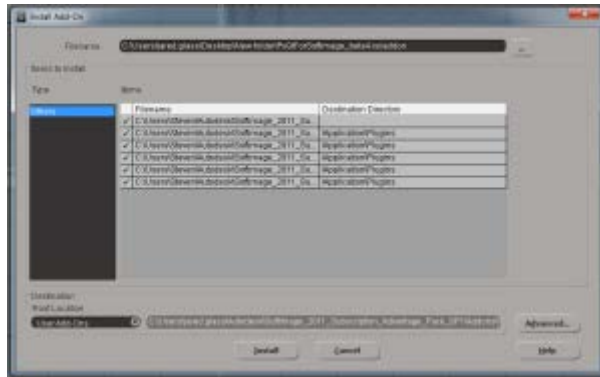
For the PyQt for Softimage Plugin go to <https://github.com/caron/PyQtForSoftimage> and find the link https://github.com/downloads/caron/PyQtForSoftimage/PyQtForSoftimage_beta4.xsiaddon. Then Right-Click on it and choose to save the link. This will download the plugin for you (you can also do it on the link example posted here).

For Mac and Linux users: there are some build instructions to help you out too on the [PyQt for Softimage](#) page.

Now that we've downloaded everything we need, all that is left is to install, so install the PyQt file you downloaded from the Riverbank site and afterwards, open Softimage.

In Softimage go File Menu >> Add-On >> Install.

Click the button to the right of “Filename” and select your downloaded “PyQtForSoftimage_beta4.xsiaddon”. If you want to install it to a workgroup you can by selecting “Workgroup” from the “Root Location” combo. Otherwise leave it as is to install the plugin to your Softimage User Preferences folder.



Then click Install and you're done!!!

After restarting Softimage you can start creating PyQt GUIs in Softimage and take your plugins to the next level 😊

...and in the next lesson I will show you how 😊

ALSO: A huge HUGE thanks to Steven Caron and Jo Benayoun for this awesome plugin.

Creating a Simple PyQt Plugin

To create a PyQt Softimage Plugin you start off the same way as creating any other Command Plugin. There are only really two main differences between writing a plugin that has PyQt and one that doesn't. The one comes into play when you want to test your plugin while developing it but I'll show you an easy little trick later. The other is that you create your own custom class (well you actually just add onto one of the PyQt ones) and in there you put all your GUI and PyQt Code. You don't have to do it this way but it is recommended and does give you ULTIMATE POWER!!! (Also as I keep saying, it's not that difficult 😊)

For this exercise we'll make an Animation Copying tool that will allow a user to select two objects and transfer animation from one to the other. That part of the plugin will only be a few short lines though. The most important part of this tutorial will be the introduction to how PyQt works.

So, lets get started. Create a Command Plugin called “Animation_Copy”:

File >> Plug-in Manager >> File >> New >> Command.

Set the “Command Name” field to “Animation_Copy” and hit “Generate Code.

Now we have the skeleton of our plugin. All we need to do is add the fleshy bits.

Firstly, we need to import the PyQt modules. This is done with the commands:

```
from PyQt4.QtGui import *
```

“PyQt4” is the version of PyQt we are using. “QtGui” is the module we want, filled with all the GUI elements called “Widgets” and finally the “import *” tells python to just import everything in that module. As opposed to having to go:

```
from PyQt4.QtGui import QPushButton, QTextEdit
```

and so on.

Next we create that “custom class” thing I wrote about earlier.

```
class AnimationCopy_Dialog( QDialog ):  
    def __init__( self, parent ):  
        QDialog.__init__( self, parent )
```

The name of the class above is “AnimationCopy_Dialog”. The “QDialog” in the brackets that follow it just mean that this is actually a QDialog (a standard PyQt Widget). This means that our class currently is just a QDialog class with a different name.

the function “__init__” is run whenever the class is innitialized (when you call “AnimationCopy_Dialog()”). “QDialog.__init__(self, parent)” is calling the original QDialog’s init function to run to setup the widget. “self” is the same as java’s “this” or kinda like “this_model” in ICE. Its just a variable that references this instance of the class. And “parent” we will get to in a moment.

To display the AnimationCopy_Dialog we created we need to add some code to the “Execute” function of our plugin:

```
import sip  
sianchor= Application.getQtSoftimageAnchor()  
sianchor= sip.wrapinstance( long(sianchor), QWidget )  
dialog    = AnimationCopy_Dialog( sianchor )  
dialog.show()
```

The PyQt for Softimage plugin that we installed earlier amongst other things, acts as a parent for all QWidgetts you want to use within Softimage. The first 3 lines will always be the same. After

that we create and store the AnimationCopy_Dialog in a variable named dialog and finally we call the “.show()” function that makes the AnimationCopy_Dialog (stored in the dialog variable, remember 😊 visible.

The sianchor gets passed into the AnimationCopy_Dialog which sets the AnimationCopy_Dialog’s parent to be the sianchor.

To test the plugin, just create a button that calls it. I.e the button code would be:

```
Application.Animation_Copy()
```

Everytime you want to run your plugin, you just need to save it and press the button. This is the one little trick I was talking about.

You need to do this as opposed to just calling the plugin’s “Execute” function otherwise Softimage won’t reload your PyQt code and it won’t work properly.

At this point your code should look like this:

```
# Animation_CopyPlugin
# Initial code generated by Softimage SDK Wizard
# Executed Mon Sep 17 18:00:17 UTC+0200 2012 by jared.glass
#
# Tip: To add a command to this plug-in, right-click in the
# script editor and choose Tools > Add Command.
import win32com.client
from win32com.client import constants
from PyQt4.QtGui import *

null = None
false = 0
true = 1

class AnimationCopy_Dialog( QDialog ):
    def __init__( self, parent ):
        QDialog.__init__( self, parent )

def XSILoadPlugin( in_reg ):
    in_reg.Author = "jared.glass"
    in_reg.Name = "Animation_CopyPlugin"
    in_reg.Major = 1
    in_reg.Minor = 0

    in_reg.RegisterCommand("Animation_Copy","Animation_Copy")
    #RegistrationInsertionPoint - do not remove this line

    return true

def XSIUnloadPlugin( in_reg ):
    strPluginName = in_reg.Name
    Application.LogMessage(str(strPluginName) + str(" has been
unloaded."),constants.siVerbose)
```

```

        return true

def Animation_Copy_Init( in_ctxt ):
    oCmd = in_ctxt.Source
    oCmd.Description = ""
    oCmd.ReturnValue = true

    return true

def Animation_Copy_Execute( ):

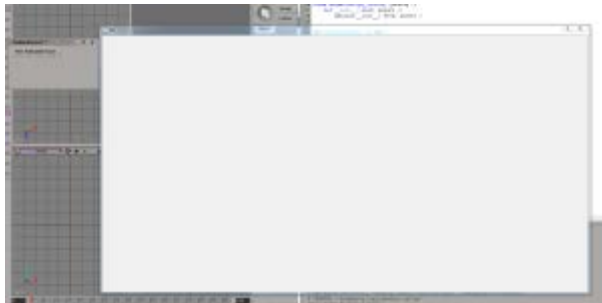
    Application.LogMessage("Animation_Copy_Execute
called", constants.siVerbose)

    import sip
    sianchor= Application.getQtSoftimageAnchor()
    sianchor= sip.wrapinstance( long(sianchor), QWidget )
    dialog    = AnimationCopy_Dialog( sianchor )
    dialog.show()

    return true

```

And when you click the button you made to test the plugin you should get a QDialog that looks like this:



Now to add some Widgets to the QDialog.

What we need to be able to do is

1. Select an object to copy the animation from.
2. Select an object to copy the animation to.
3. Click a button that runs the code to copy the animation.

So lets get started.

Widgets:

The code to create widgets will need to go in our AnimationCopy_Dialog classes “__init__” function so that they are created as soon as the AnimationCopy_Dialog is initialized and we do that like this:

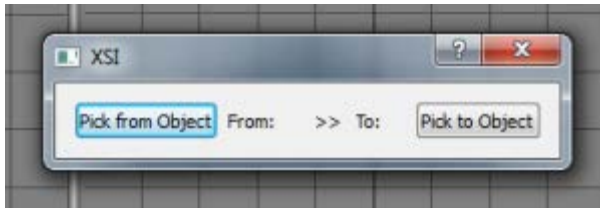
```
# Create Widgets
from_label      = QLabel("From: ")
self.fromObj_label = QLabel()
fromPick_button = QPushButton("Pick from Object")
center_label    = QLabel(" >> ")
to_label        = QLabel("To: ")
self.toObj_label = QLabel()
toPick_button   = QPushButton("Pick to Object")
```

Layout:

But that only created the widgets. We still need to add them to the AnimationCopy_Dialog's layout for them to actually show.

```
# Layout
layout = QHBoxLayout(self)
layout.addWidget(fromPick_button)
layout.addWidget(from_label)
layout.addWidget(self.fromObj_label)
layout.addWidget(center_label)
layout.addWidget(to_label)
layout.addWidget(self.toObj_label)
layout.addWidget(toPick_button)
```

Now when you press click your test button, the widgets should be added to the AnimationCopy_Dialog yay!



The Creation of the widgets is pretty straight forward. We just create widgets and store them in variables. With the layout though when creating it we do `QHBoxLayout(self)` as opposed to just calling `QHBoxLayout()`. This sets “self” as the layout’s parent. Making the layout the layout used by `AnimationCopy_Dialog` and that’s why whatever we add to the layout gets added to the `AnimationCopy_Dialog`.

Also you may have noticed that for some of the created Widgets I give the variables they get stored in a “self.” prefix (like `self.toObj_label`). This makes sure the variable is accesable anywhere within the class whereas the other variables only exist within the “__init__” function. We do this because later if someone clicks the `toPick_button` we want to set or change the text displayed by the `self.toObj_label`.

Connections:

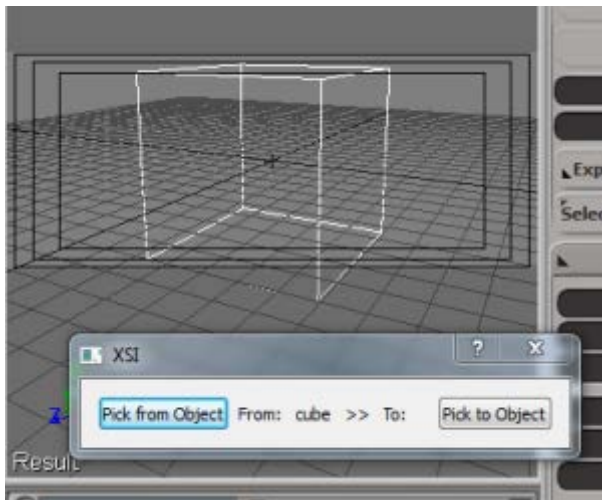
If you click the buttons you'll notice that nothing happens. This is because they are not "connected" to anything. We need to connect the button to a function so that everytime the button is pressed, the function is run. First we need to create the function we want to run:

```
def Pick_FromObj(self):  
    from_obj = Application.PickObject()[2]  
    self.fromObj_label.setText( from_obj.FullName )
```

And then in the "__init__" we "connect" the button we want to our function. (Remember to get more info on what exactly "Application.PickObject" does, open the Softimage SDK Guide and under the index tab type "PickObject").

```
# Connections  
fromPick_button.clicked.connect(self.Pick_FromObj)
```

Now when we test our plugin and click the fromPick_button the self.fromObj_label will display our picked object's FullName.



Now we just need to copy the code and alter it for the toPick_button and self.toObj_label.

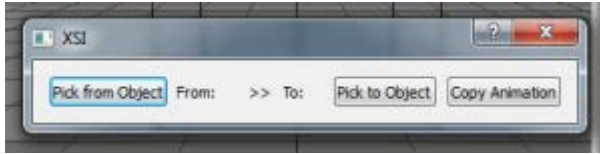
All that is left now is to setup the part of the plugin that actually copies the animation from the "from object" to the "to object". For this we're just going to create another button called "Copy Animation" and connect it to a function that does just that.

So, add the code:

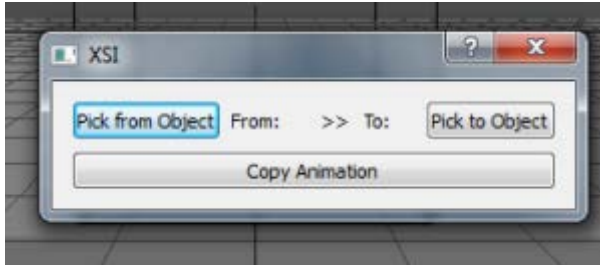
```
copy_button = QPushButton("Copy Animation")
```

to the section of code where you created all your other widgets.

This is all fine and dandy but if you were to just add this button to your layout now, you'd get something like



which to be honest doesn't look as good as



To setup the Plugin's layout to look like the second image we need to use the `QGridLayout` instead of the `QHBoxLayout`.

You can find documentation on the `QGridLayout` [here](#) and all the other QtGui Widgets you can create [here](#).

From the `QGridLayout` documentation there are two things that we need to take a look at:

- `addWidget(self, QWidget, int row, int column, Qt.Alignment alignment = 0)`
- `addWidget(self, QWidget, int row, int column, int rowSpan, int columnSpan, Qt.Alignment alignment = 0)`

The first `addWidget` function tells us that we can specify a `QWidget` followed by the row and column to place the widget in. The second also gives us these options but it also goes further to allow us to stretch the widget over an amount of columns and rows.

All the previous widgets will be placed on the first row of our `QGridLayout` and the `copy_button` will be placed on the second row but we'll set it to take up the entire second row. Also remember that with programming, things tend to start from 0 instead of 1. So the very first row and column of the `QGridLayout` will both be 0.

So replace all the previous `QHBoxLayout` code with the new `QGridLayout` code:

```
# Layout
layout = QGridLayout(self)
layout.addWidget(fromPick_button, 0, 0)
layout.addWidget(from_label, 0, 1)
layout.addWidget(self.fromObj_label, 0, 2)
layout.addWidget(center_label, 0, 3)
layout.addWidget(to_label, 0, 4)
layout.addWidget(self.toObj_label, 0, 5)
layout.addWidget(toPick_button, 0, 6)
layout.addWidget(copy_button, 1, 0, 1, 7)
```

Now when you run the plugin the copy_button should be on the second row and take up the width of the entire second row like in the picture above.

Next we need to create the function that does the actual Copying of the Animation:

```
def Copy_Animation(self):
    fromObj_fullName= str( self.fromObj_label.text() )
    toObj_fullName   = str( self.toObj_label.text() )
    # Make sure user picked from and a to objects.
    if not fromObj_fullName or not toObj_fullName:
        QMessageBox.critical(self, "Invalid Selection", "Please select a
valid from_object and to_object")
    else:
        # Copy and Paste animation
        Application.CopyAllAnimation2(fromObj_fullName)
        Application.PasteAllAnimation(toObj_fullName)
        QMessageBox.information(self, "Success", "Animation copied.")
```

First we get the names we stored in self.fromObj_label and self.toObj_label by calling their “.text()” functions. This gives us a QString, which is fine for PyQt but we cannot pass PyQt elements to Softimage functions (like “CopyAllAnimation2”) so we turn it into a string with the “str()”. Then we check if fromObj_fullName or toObj_fullName don’t have any names stored inside them. If they don’t we popup a QMessageBox telling them about the issue, otherwise if there is an object name in fromObj_fullName and toObj_fullName, we copy and paste the animation and let the user know it was a success with the last QMessageBox.

And that’s that. Another awesome plugin done!!!

Now that you have an idea of how to use PyQt there is a whole new dimension of functionality and epicness you can add to your tools.

I hope I have managed to help you on voyage across the sea of code. Thanks for reading 😊

And if you want, there’s even more in the [Veteran Category](#).

As usual I have included my version of the finished plugin below:

```
# Animation_CopyPlugin
# Initial code generated by Softimage SDK Wizard
# Executed Mon Sep 17 18:00:17 UTC+0200 2012 by jared.glass
#
# Tip: To add a command to this plug-in, right-click in the
# script editor and choose Tools > Add Command.
import win32com.client
from win32com.client import constants
from PyQt4.QtGui import *

null = None
false = 0
true = 1
```

```

class AnimationCopy_Dialog( QDialog ):
    def __init__( self, parent ):
        QDialog.__init__( self, parent )

        # Create Widgets
        from_label          = QLabel("From: ")
        self.fromObj_label  = QLabel()
        fromPick_button     = QPushButton("Pick from Object")
        center_label        = QLabel(" >> ")
        to_label            = QLabel("To: ")
        self.toObj_label    = QLabel()
        toPick_button       = QPushButton("Pick to Object")
        copy_button         = QPushButton("Copy Animation")

        # Layout
        layout              = QGridLayout(self)
        layout.addWidget(fromPick_button, 0, 0)
        layout.addWidget(from_label, 0, 1)
        layout.addWidget(self.fromObj_label, 0, 2)
        layout.addWidget(center_label, 0, 3)
        layout.addWidget(to_label, 0, 4)
        layout.addWidget(self.toObj_label, 0, 5)
        layout.addWidget(toPick_button, 0, 6)
        layout.addWidget(copy_button, 1, 0, 1, 7)

        # Connections
        fromPick_button.clicked.connect(self.Pick_FromObj)
        toPick_button.clicked.connect(self.Pick_ToObj)
        copy_button.clicked.connect(self.Copy_Animation)

    def Pick_FromObj(self):
        from_obj = Application.PickObject()[2]
        self.fromObj_label.setText( from_obj.FullName )

    def Pick_ToObj(self):
        to_obj = Application.PickObject()[2]
        self.toObj_label.setText( to_obj.FullName )

    def Copy_Animation(self):
        fromObj_fullName= str( self.fromObj_label.text() )
        toObj_fullName  = str( self.toObj_label.text() )
        # Make sure user picked from and a to objects.
        if not fromObj_fullName or not toObj_fullName:
            QMessageBox.critical(self, "Invalid Selection", "Please select a
valid from_object and to_object")
        else:
            # Copy and Paste animation
            Application.CopyAllAnimation2(fromObj_fullName)
            Application.PasteAllAnimation(toObj_fullName)
            QMessageBox.information(self, "Success", "Animation copied.")

    def XSIloadPlugin( in_reg ):
        in_reg.Author = "jared.glass"
        in_reg.Name = "Animation_CopyPlugin"
        in_reg.Major = 1
        in_reg.Minor = 0

```

```

    in_reg.RegisterCommand("Animation_Copy","Animation_Copy")
    #RegistrationInsertionPoint - do not remove this line

    return true

def XSIUnloadPlugin( in_reg ):
    strPluginName = in_reg.Name
    Application.LogMessage(str(strPluginName) + str(" has been
unloaded."),constants.siVerbose)
    return true

def Animation_Copy_Init( in_ctxt ):
    oCmd = in_ctxt.Source
    oCmd.Description = ""
    oCmd.ReturnValue = true

    return true

def Animation_Copy_Execute( ):

    Application.LogMessage("Animation_Copy_Execute
called",constants.siVerbose)

    import sip
    sianchor= Application.getQtSoftimageAnchor()
    sianchor= sip.wrapinstance( long(sianchor), QWidget )
    dialog    = AnimationCopy_Dialog( sianchor )
    dialog.show()

    return true

```

Veteran:

Modules

Technically any python file (a file with the “.py” extension) can be imported as a module.

The PyQt4.QtGui we were using earlier was a module and so is the win32com.client we have in our plugins. Anything you can “import” can technically be called a module.

Why make a module?

Most modules group a set of similar functions. This makes those functions a lot easier to find since they’re all grouped in the specific module but it also keeps your code neater since the various functions are sorted into various modules. Modular programming does not necessarily refer to creating as many modules as possible but it does mean grouping things that can be and turning code that is used more than once into a function rather than just typing out the same set of commands a few times. And modular programming is definately something to aspire to, for the same reasons that modules are created. It just makes for more efficient and managable programs.

e.g: the python “os” module contains varous functions that deal with the current opperating system. Like the command os.makedirs which makes a specified folder and os.listdir which lists all the files and folders within a specified folder etc.

How to make a module?

Making modules is as simple as saving a python file, literally. So lets create out own module called “test.py” and put in a function that prints test:

```
def Print_ItemsInList(list):  
    for item in list:  
        print item
```

Now save that as “Test.py”

Importing a custom module:

We have four ways to do this.

1. The easiest is to put the module we want to import into the same folder as the script we're running and then just import the module's file name.

Note this doesn't work if you're using Softimage's Script Editor though but you can try it with IDLE which is the [IDE \(Script Editor\)](#) that comes with python. I prefer using Notepad++ when I develop outside of Softimage and there's this awesome post [here](#) on how to get Notepad++ to run python.

So in IDLE, Notepad++ or whatever IDE you decide to use (except the Softimage Script Editor), type the following code:

```
import Test
my_list = [1, 2, 3, 4, 5, 6]
Test.Print_ItemsInList(my_list)
```

Then save that in the same folder as the "Test.py" module you created and run it.

You should see something like this outputted:

```
1
2
3
4
5
6
```

The above method is the ONLY method that DOESN'T work with Softimage. All the following methods do.

2. Adding the module's path to the system path (its a lot easier than it probably sounds :p)

In python there is a list stored in the sys module that contains all of the folders to look in that have modules to load. All we need to do is add our module's folder path to the sys.path and we'll be able to import it.

Note: This method DOES work for Softimage.

In my case my Test.py module is saved in "C:\Users\jared.glass\Autodesk\Softimage_2011_Subscription_Advantage_Pack_SP1\Data\Scripts" but note that with python, a "\" needs to be doubled to "\\". This only happens with the backslash because it can act as an escape character. You don't need to worry about that too much just remember that with python to double up your backslashes.

At the top of your script you run the code:

```
import sys
sys.path.append("C:\\Users\\jared.glass\\Autodesk\\Softimage_2011_Subscription_Advantage_Pack_SP1\\Data\\Scripts")
```

```
import Test
```

And the output should be the same as before.

You can also use python's `imp` module. It allows you to import a module directly into a variable and can be safer than just adding a module to your python path if you have many modules in different locations, but that have the same name.

```
import imp
```

```
Test = imp.load_source('Test',
"C:\\Users\\jared.glass\\Autodesk\\Softimage_2011_Subscription_Advantage_Pack_SP1\\Data\\Scripts")
```

```
etc
```

3. Python's site-packages folder.

When you install a python module (like PyQt) the parts of that module that python uses get placed in here. On my system it's `C:\Python26\Lib\site-packages`.

For this method all you need to do is copy your module into that folder and you'll be able to import it.:

```
import Test
```

```
etc
```

The reason I mentioned this method last though is that since you are placing your module in a local folder, if you create a plugin or tool that people use from other computers, they will have to copy the file too. So I recommend you rather place the modules you want to import in a folder on your server and use the `imp` or `sys.path` methods so that no matter which computer your plugin is being run from, it will be able to see and use your module.

Multi-Model Importer

Hi There

Something that can get a bit annoying is only being able to import one model into Softimage at a time. Yeah we can drag a whole bunch of models in from explorer but if we create our own Multi-Model Importer plugin, firstly we'll be rid of the mentioned annoyance and secondly we can add a whole lot of functionality to our import plugin, like where to look for the models, or what to call them and in this example in order to keep the scene nice and neat I'm even going to get this plugin to place each model type in a specific category.

My categories will be:

- Characters
- Lights
- Cameras
- Props

And to keep this from getting complicated, while I'm here I'm just going to make those models. And because this is a coding blog I'll make and export the models with code 😊

```
import os
from random import randint
xsi = Application

# This Xsi Project's folder path
project_path = xsi.ActiveProject.Path

def Randomize_Position(obj):
    # Get random numbers between -10 and 10 and set as position
    obj.Kinematics.Local.posx.Value = randint(-10,10)
    obj.Kinematics.Local.posy.Value = randint(-10,10)
    obj.Kinematics.Local.posz.Value = randint(-10,10)

# Create Cube called "Cube_Character"
cube_obj = xsi.CreatePrim("Cube", "MeshSurface", "Cube_Character")
Randomize_Position(cube_obj)
model_path = os.path.join(project_path, "Models", "Cube_Character.emdl")
xsi.ExportModel(cube_obj, model_path)

# Create Sphere called "Sphere_Character"
sphere_obj = xsi.CreatePrim("Sphere", "MeshSurface", "Sphere_Character")
Randomize_Position(sphere_obj)
model_path = os.path.join(project_path, "Models", "Sphere_Character.emdl")
xsi.ExportModel(sphere_obj, model_path)

# Create Spot Light called "Spot_Light"
spot_obj = xsi.GetPrimLight("Spot.Preset", "Spot_Light")
Randomize_Position(spot_obj)
model_path = os.path.join(project_path, "Models", "Spot_Light.emdl")
```

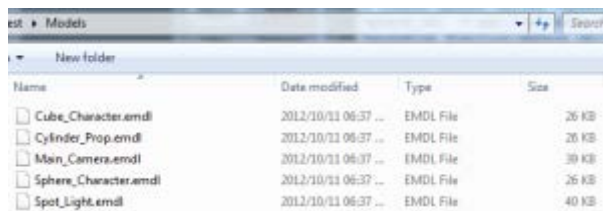
```
xsi.ExportModel(spot_obj, model_path)

# Create a Camera called "Main_Camera"
cam_obj = xsi.GetPrimCamera("Camera", "Main_Camera")
cam_obj.Kinematics.Local.posz.Value = 50
model_path = os.path.join(project_path, "Models", "Main_Camera.emdl")
xsi.ExportModel(cam_obj, model_path)

# Create a Cylinder called "Cylinder_Prop"
cyl_obj = xsi.CreatePrim("Cylinder", "MeshSurface", "Cylinder_Prop")
Randomize_Position(cyl_obj)
model_path = os.path.join(project_path, "Models", "Cylinder_Prop.emdl")
xsi.ExportModel(cyl_obj, model_path)
```

Note: os.path.join below adds the slashes between folder names depending on the operating system you're on. So for windows it would put "\\" between the project_path, "Models", and "Cylinder_Prop.emdl" and for mac or linux, "/" would get put between them. The using os.path makes your plugins a lot easier to take across multiple Operating Systems.

Now that that's done you should have all those objects exported to your Models folder



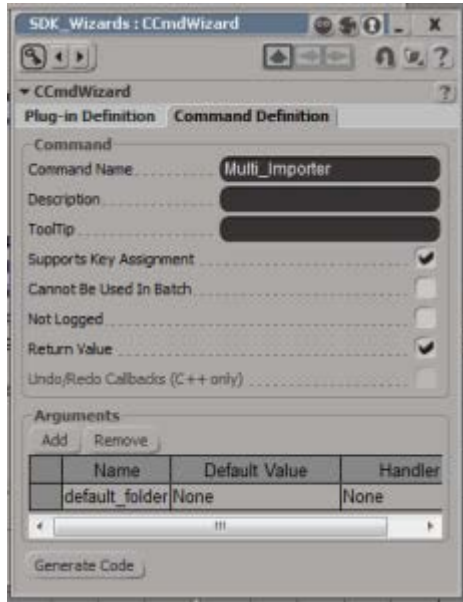
And we can continue with our plugin:

So as with all the previous plugins, you need to start by creating it 😊

File >> Plug-in Manager >> File >> New >> Command.

Set the Command Name to: Multi_Importer and in the "Command Definition" tab, Add an Argument (by clicking "Add").

Change the name to "default_folder". Then click "Generate Code".



The reason we added that “default_folder” option is that if we want to call this command from another plugin, we can tell it where to look for models to import. When you’re working with loads of models in loads of folders it can save quite a bit of time if your tools just “jump” to the correct folders instead of the users having to browse manually each and everytime they want to load or reload a model.

So let's setup this plugin to do some PyQt magic. At the top:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

And in the “Multi_Importer_Execute” we need to get the Qt Softimage Anchor:

```
import sip
sianchor = Application.getQtSoftimageAnchor()
sianchor = sip.wrapinstance(long(sianchor), QWidget)
```

But now instead of creating our own custom QDialog like we did with the AnimationCopy Plugin we’re going to use the existing QFileDialog. You can find out more about it [here](#). And if we look around a bit you go further down it’s even got the perfect function for us [here](#).

```
QStringList QFileDialog.getOpenFileNames (QWidget parent = None,
QString caption = QString(), QString directory = QString(), QString filter = QString(),
Options options = 0)
```

Now all we need to do is set that up. The `QStringList` tells us that this function returns a `QString` list, and the rest tells us what we need to input.

```
modelPath_lst = QFileDialog.getOpenFileNames(parent = sianchor,
caption = "Import Models", directory = default_folder, filter = "*.emdl")
```

The parent we set to be the sianchor (which we must always do), the filter we set to look for anything that ends in “.emdl”. Remember, the * tells Softimage and in this case PyQt to look for anything and everything, so when the QFileDialog is launched, we will only be able to see files ending in “.emdl” and folders so we can navigate to the desired files if need be. The directory is set to our default_folder variable but currently the default_folder isn’t set to anything so if we run the plugin without inputting a default_folder, its not going to take us anywhere useful.

To make it default to your Softimage project’s Models folder, at the top of your plugin import the os module.

Then after the bit that says:

```
null = None
false = 0
true = 1
```

Add a variable named “models_folder” and set it to this project’s path:

```
models_folder = os.path.join(Application.ActiveProject.Path, "Models")
```

And in the Multi_Importer_Init function, change the line:

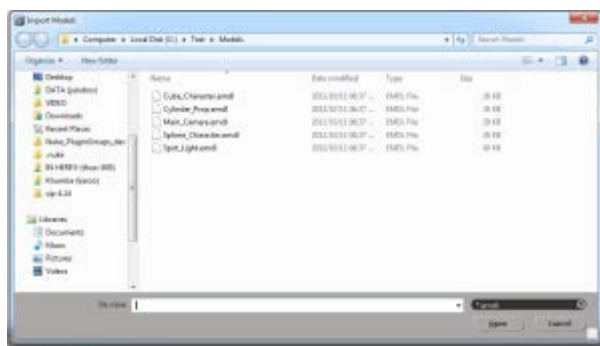
```
oArgs.Add("default_folder", constants.siArgumentInput, "None")
```

to:

```
oArgs.Add("default_folder", constants.siArgumentInput, models_folder)
```

Now the default_folder variable is set to this Softimage project’s Models folder by default.

So when you save and run your plugin, it should popup something looking like this:



To get the models the user selected after they press the “Open” button we loop through the modelPath_lst just like we do with standard python lists. From there we can import each model but remember, the QFileDialog returns QObjects, so the QStringList it returned will be full of QStrings. If you give Softimage a QString it won’t know what to do with it so turn the paths into standard python strings first:

```

for mdl_path in modelPath_lst:
    mdl_obj = Application.SIImportModel( str(mdl_path) )[1]

```

Now you have a Multi-Selection Model Importer 😊.

For those of you that have a programming background and know what threads are, unfortunately you cannot run Softimage's ImportModel function in anything other than the main thread. If you don't know what I'm talking about don't worry, you'll come across [Threading](#) sooner or later 😊

Next thing to do is add the imported models to their relevant categories.

So firstly, we need to determine the category of each imported model. To do this I'm going to create a function near the top of our plugin:

After:

```

null = None
false = 0
true = 1
models_folder = os.path.join(Application.ActiveProject.Path, "Models")

```

Add:

```

def Get_Cat(mdl_obj):
    mdl_cat = None
    # Look at each category name in the cat_lst
    for cat in cat_lst:
        # If the category name is in the model's name
        if cat in mdl_obj.Name:
            # Then set the mdl_cat that will be returned to this category
            mdl_cat = cat
            # Stop the loop here. There's no point looking further if the
            # category has already been found
            break
    return mdl_cat

```

And then in the loop that imports the models, after the line:

```
mdl_obj = Application.SIImportModel( str(mdl_path) )[1]
```

Add

```
mdl_cat = Get_Cat(mdl_obj)
```

This plugin may be used to import models that aren't in any of the categories though so rather than limit it to only being able to import models with a category, if the model has no category "None" is returned for the mdl_cat. So when this happens we can just LogMessage that the specific model has no category:

```
if mdl_cat == None:
```

```
Application.LogMessage("Multi_Importer: the model " +  
mdl_obj.Name + " doesn't have a valid category.", constants.siWarning)
```

Otherwise if the category is valid we:

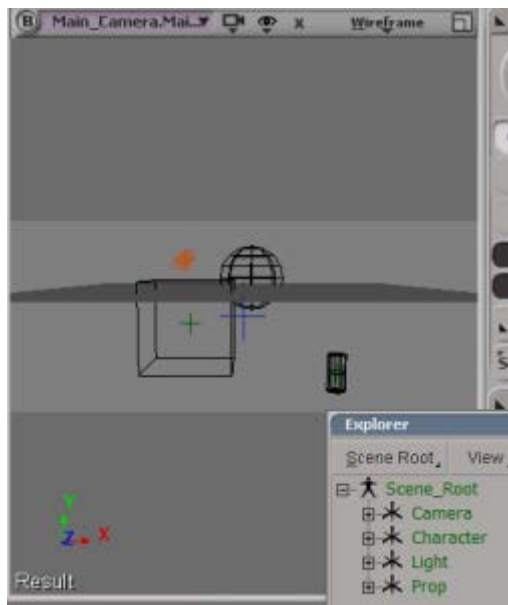
1. Look for that category null.
2. If the null doesn't exist, create it.
3. Parent the model under the null.

And the code for that is:

```
else:  
    cat_nll = Application.ActiveSceneRoot.FindChild(mdl_cat,  
constants.siNullPrimType, "", False)  
    if not cat_nll:  
        cat_nll = Application.GetPrim("Null", mdl_cat)  
    Application.ParentObj(cat_nll, mdl_obj)
```

Above, we look under the Active Scene Root for an null named whatever is stored in the mdl_cat variable. Note the False at the end. This tells Softimage to only look directly under the Scene_Root and not within objects under the Scene_Root incase a Character has a null in it named Character in it's rig setup or for any other reason.

And after running the tool, if you select all the available models for import you should get something like this in your scene:



Note: If your Camera null is named Camera1 and the same for your light null, it is because there is already a object named camera in your scene and another object named light. Softimage will automatically add a number to the end of an objects name if that name already exists elsewhere because no two objects may have the same name (unless they are under Models). So just delete the camera and light before running the pugin.

And finally:

When you want to export an object from Softimage it has to be in a model. If it isn't already Softimage will create and put the object you want to export in one. You could export Obj's but those won't keep all the information as models do. Models store everything that was applied to that object in the Softimage scene so when you import it again it will have the same animation, constraints, settings and everything else it had in the original scene. This is why exporting objects as models is in most cases the way to go.

The thing is when you import a model you may not necessarily want to keep the actual model but only want to keep the object inside. For instance, a Character or Prop would generally be rigged so you'd want that to stay in a model when in Softimage but a Light or Camera may have no reason what so ever to be kept in a model.

So for the Light and Camera Categories we'll set the plugin to remove the objects from the model:

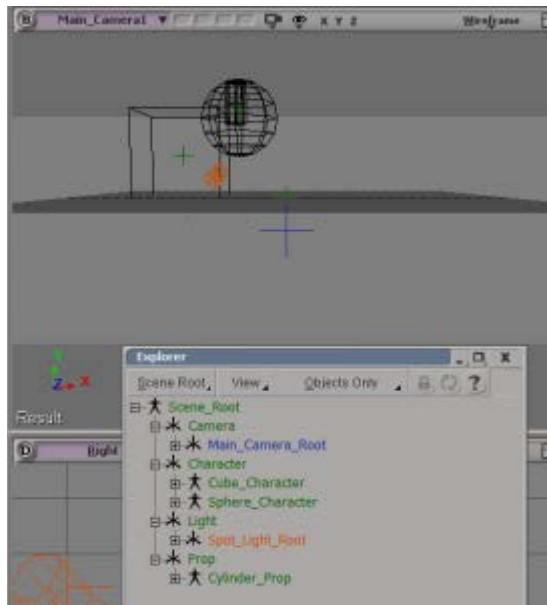
Within the loop that imports the models, after the line where you set the model's parent to be the relevant category null:

```
if mdl_cat in ["Light", "Camera"]:  
    for child_obj in mdl_obj.Children:  
        Application.ParentObj(cat_null, child_obj)  
        Application.DeleteObj(mdl_obj)
```

This checks if this model's category is a Light or Camera. If so it finds all the objects directly under the model and parents them directly under the category null (removing them from the model). Lastly we delete the empty model object.

And that's that 😊

When you run the plugin on an empty scene the explorer should look like:



```
# Multi_ImporterPlugin
# Initial code generated by Softimage SDK Wizard
# Executed Thu Oct 11 19:02:20 UTC+0200 2012 by jared.glass
#
# Tip: To add a command to this plug-in, right-click in the
# script editor and choose Tools > Add Command.
import win32com.client
from win32com.client import constants
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import os

null          = None
false        = 0
true         = 1
# Path to this project's Models folder
models_folder = os.path.join(Application.ActiveProject.Path, "Models")
# List of categories to be created
cat_lst      = ["Character", "Light", "Camera", "Prop"]

def Get_Cat mdl_obj):
    mdl_cat = None
    # Loop through categories and check if the category is in the model's
    name
    for cat in cat_lst:
        if cat in mdl_obj.Name:
            mdl_cat = cat
            break
    return mdl_cat

def XSI_LoadPlugin( in_reg ):
    in_reg.Author = "jared.glass"
    in_reg.Name = "Multi_ImporterPlugin"
```

```

in_reg.Major = 1
in_reg.Minor = 0

in_reg.RegisterCommand("Multi_Importer","Multi_Importer")
#RegistrationInsertionPoint - do not remove this line

return true

def XSIUnloadPlugin( in_reg ):
    strPluginName = in_reg.Name
    Application.LogMessage(str(strPluginName) + str(" has been
unloaded."),constants.siVerbose)
    return true

def Multi_Importer_Init( in_ctxt ):
    oCmd = in_ctxt.Source
    oCmd.Description = ""
    oCmd.ReturnValue = true

    oArgs = oCmd.Arguments
    # The default_folder is set to the models folder incase no default_folder
is specified
    oArgs.Add("default_folder",constants.siArgumentInput, models_folder)
    return true

def Multi_Importer_Execute( default_folder ):

    Application.LogMessage("Multi_Importer_Execute
called",constants.siVerbose)

    import sip
    sianchor = Application.getQtSoftimageAnchor()
    sianchor = sip.wrapinstance(long(sianchor), QWidget)
    modelPath_lst = QFileDialog.getOpenFileNames(parent = sianchor,
caption = "Import Models", directory = default_folder, filter = "*.emdl")

    # Loop through file paths of the selected models
    for mdl_path in modelPath_lst:
        mdl_obj = Application.SIImportModel( str(mdl_path) )[1]
        mdl_cat = Get_Cat(mdl_obj)
        if mdl_cat == None:
            Application.LogMessage("Multi_Importer: the model " +
mdl_obj.Name + " doesn't have a valid category.", constants.siWarning)
        else:
            # Try find existing Category Null, if there isn't one then create
it.
            cat_nll = Application.ActiveSceneRoot.FindChild(mdl_cat,
constants.siNullPrimType, "", False)
            if not cat_nll:
                cat_nll = Application.GetPrim("Null", mdl_cat)
            # Put the model under the specific category null
            Application.ParentObj(cat_nll, mdl_obj)

            # Check if this model's category is light or camera
            if mdl_cat in ["Light", "Camera"]:
                # Get all the objects inside the model (in our case there is
only one but there may be more)

```

```
        for child_obj in mdl_obj.Children:
            # Put each object that is inside the model under the
category null
            Application.ParentObj(cat_nll, child_obj)
            # Delete the mepty model
            Application.DeleteObj(mdl_obj)

    return true
```

Appendix

This book is based on the contents of my “Conquering the Code in Softimage Xsi” blog which can be found at:

<http://codingforxsi.wordpress.com/>

My other Coding related blogs are:

Maya: <http://CodingForMaya.wordpress.com/>

Nuke: <http://CodingForNuke.wordpress.com/>

After Effects: <http://codingforaftereffects.wordpress.com/>

Contact

You can contact me via my Softimage or Personal blogs:

<http://codingforxsi.wordpress.com/>

<http://mushutehk.wordpress.com/>

or on LinkedIn at:

<http://www.linkedin.com/in/JaredMushuGlass>

Donations

If you found this book educational or useful in any way, please support the author. This book has been released for free but the time and experience it took to write it did have its costs. Every single donation is very much appreciated and lets the author know that the book has been of value to someone and served its purpose, which will most likely lead to future releases. Please remember that a donation of one US dollar from someone is better than no donation from anyone. And as stated before, any amount is very much appreciated. Thank you.

<http://codingfordonations.wordpress.com/>

