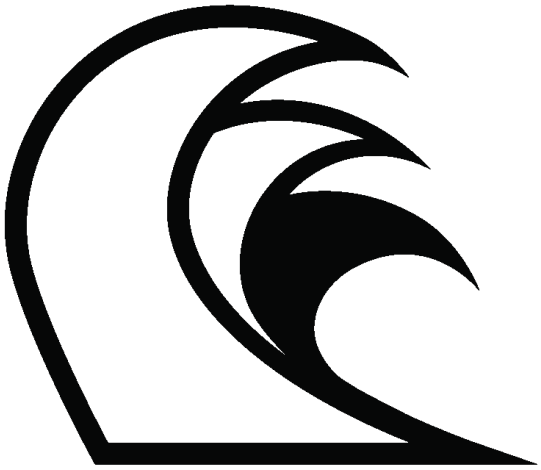
Programming in MotionBuilder || Focusing on Python

Autodesk MotionBuilder 2013

Autodesk Developer Network
May 2012
Module 3: MotionBuilder Architecture



AUTODESK®
MOTIONBUILDER

Contents

Programming in MotionBuilder || Focusing on Python

Autodesk Developer Network

Module 3: MotionBuilder Architecture

Agenda

- General Concepts
- MotionBuilder Architecture
- Elements
- User Interface
- Boxes
- Devices
- Constraints
- Tools
- Manipulators
- Utility Classes
- Reviewing the Python Documentation Closer
- More Control over Loading and Saving Scenes

3.0 General Concepts

Real-Time Architecture

- MotionBuilder is a real-time architecture dividing different functionality into different tasks with different priorities.
- This permits any development to that take advantage of the existing architecture in order to extend the software.

Multi-Thread Architecture

The architecture of MotionBuilder is based on a multi-threaded engine optimized for real-time performance. Separate threads having different priorities are launched in parallel, ensuring that MotionBuilder takes full advantage of the available hardware (including multiple processors and cores). For example, Constraint Evaluation, and Rendering are each performed in separate threads.

Don’t call us...We’ll call you

Programming in MotionBuilder is designed with a 'Don’t call us, we’ll call you' philosophy. Regardless of the type of development, the structure is essentially the same. A new class is derived from a class that exists in the Python SDK. Basically we do everything for you! Any objects you create get registered in MotionBuilder automatically without you having to do additional steps.

3.1 MotionBuilder Architecture

MotionBuilder Classes

Within MotionBuilder, there are six basic types of classes:

- Base Classes
- Scene Elements
- Animation
- User Interface
- Plug-in Types
- Utilities

Base Classes

These classes are the fundamentals of the Python SDK, defining the basic structure for the classes that are used in different plug-in types, for example FBComponent, FBProperty, and others. These classes provide the framework that enables you to develop scripts that are automatically integrated into MotionBuilder.

Scene Elements

These classes represent elements in MotionBuilder, such as models, cameras, and others, and are used to manipulate MotionBuilder elements from a script.

Animation

These are the classes that are related with operation on animation data. Such as takes, animation node or FCurve.

User Interface

These classes are derived from FBVisualComponent. They expose all of the widgets that are available in the MotionBuilder interface to be used in custom layouts.

Plug-in Types

The classes of this type are used to derive the main functionality of a new plug-in. These are the classes like FBDevice and FBConstraint that provide a base framework for plug-in development.

Utilities

Utility classes provide multi-purpose tools to create scripts for MotionBuilder. They are time management, I/O classes, and array management classes providing ready-to-go usable classes for any script based on the MotionBuilder architecture.

3.2 Base Classes

'FBComponent' Class

- At the root of most classes dealing directly with the MotionBuilder architecture, there is the FBComponent class, which contains all of the creation and management functions for objects integrated into MotionBuilder. Most Python classes derive from this class. FBComponent classes define common object characteristics, including creation and destruction methods. It also provides a scheme for property management.
- FBComponent is the base entity class for most objects in the MotionBuilder SDK. It defines the basic memory management methods for object creation and destruction.

'FBProperty' Class

A property is a container for callback functions that inherit from the FBProperty class. What appears to be a variable is in fact an interface to the proprietary Autodesk technology underneath. This alleviates the use of the Python SDK, letting you access the MotionBuilder functionality without requiring full knowledge of the underlying code.

- FBProperty class is the base property class. Most of the time, the state and behavior of an FBComponent object is defined by all of its properties, which are represented by various instances of FBProperty

'FBPlug' Class

- It is the common base class for FBComponent and FBProperty. The FBPlug class is responsible for managing the connections among components and properties.

3.3 Elements

Elements are the basic building blocks of MotionBuilder. They consist of:

- Models
- Devices (working with existing devices)
- Cameras
- Lights

There are many important details for elements because they incorporate the real-time architecture required for MotionBuilder. They are susceptible to evaluation callbacks, requiring both optimized programming and knowledge of the real-time architecture of MotionBuilder.

3.4 Animation

These are the classes that are related with operation on animation data. For example, takes are containers for animation in a scene. A take stores data about animation for objects, and you can plot animation onto takes. Animation nodes are channels to input and output animation data on a Box object and you can use animation node to script animation of a model. FCurve records keying information for animation node. We will have a more detailed discussion in the lesson in the next few days.

3.5 User Interface

All MotionBuilder UI controls are exposed to Python. They behave mostly like their OpenReality C++ counterparts. To help with UI building, helper classes are provided. These helper classes are defined in the module pyfbsdk_additions.

BoxLayout and GridLayout take care of most of the region handling.. BoxLayout will be familiar to users of the C++ UI toolkits Qt and GTK, or the Python toolkits Tkinter and wxPython. These layouts are used to help create simple tools by providing a basic layout of controls.

3.6 Boxes

MotionBuilder is built on the concept of a 'box'. Boxes connect inputs from one side, dealing with the information through calculations, transformations, and so on, and then send the data to outputs on the other side.

This fundamental process is the basic architecture of the real-time engine. Whenever the real-time engine is called to evaluate a frame to be displayed, it explores the tree of boxes that exists within the structure of the scene, running through the different calculations that it needs in order to display the current status of the scene.

A box is an operator in MotionBuilder. Most elements in MotionBuilder inherit the functionality of this item, which is to:

1. Read data from the input nodes.
2. Transform the data based on certain rules.
3. Send the data onto the output nodes.

These boxes appear in the Relations Constraint interface in order to be connected with other boxes or models, FBBox is the base class for anything that is Animatable.

Figure 2| Visual Representation of a Box with Input and Output

Note: If you are familiar with Maya's structure, think of a box like a node in Maya.

3.7 Devices

Creating custom devices is currently not exposed in Python, however you can still access and set data in already created devices via the FBDevice class in Python.

With MotionBuilder, it is possible to create many different types of devices based on your needs and intended use for MotionBuilder. Devices fall into one of the following three categories:

- Input
- Optical
- Output

Optical devices are a special case of an input device, but the management of its different features is performed in a specific way. Therefore, optical devices are treated as a separate category.

This is not something that is exposed to Python currently, I would be skeptical if it will be added only because this would be a scenario where you need real-time, and C++ is more optimized for this type of work.

3.8 Constraints

Constraints are real-time evaluation conditions for elements in a MotionBuilder scene. They constrain the movement and behavior of objects in a scene based on input criteria. This can be expressed in the form of formulae, positional information, hierarchal relationships, or many other different forms.

The Python SDK opens up the creation of constraints to you, exporting the necessary creation, management, and evaluation routines needed to add a constraint to MotionBuilder.

3.9 Tools

A tool in MotionBuilder is a new user interface layout adding to the current functionality of the software. Using the extensive user interface elements available with MotionBuilder, the possibilities are limitless as to what tools can be added.

3.10 Manipulators

Creating and accessing custom manipulators is currently not exposed in Python.

A manipulator is similar to a tool in usage, but bridges the gap between a tool's functionality and interaction with the 3D viewer. Manipulators are found in the Transforms window in MotionBuilder (Transformation, Parenting, and so on). They define how the input from the mouse in the 3D viewer is used in order to pick and manipulate the scene (and the objects in it).

With MotionBuilder, it is possible to add a new type of manipulator for custom interaction with the 3D viewer (selection, viewing, and so on).

3.11 Utility Classes

Along with the data structures necessary to interact with MotionBuilder, the Python SDK is distributed with a series of utility classes that are used to manipulate data or complete any of the normal functions that might be needed in the course of developing a script for MotionBuilder.

- Transport Control
- Media
- System Interface

Note: There are more utility classes in the OR SDK, but that is because there is a lot of these utility classes are already implemented in the standard Python modules, such as mathematics, and communication (TCPIP) classes, so the work did not need to be re-done.

3.12 Reviewing the Python Documentation Closer

If you are new to programming, or just object oriented programming or it's just been a while since you did it, I highly recommend brushing up on that object oriented programming, how classes work and can be inherited from other classes. Doing so will allow you to fully understand how the MotionBuilder SDK works, and you will be able to utilize all the classes and functions that are available to you so that you can realize the power of the SDK.

The Classes are divided like this:

- There are roughly 45 standalone classes.
- One enumeration parent class, where all the enumerations in Python inherit from this class. You don't need to learn or study these classes because when you need an enumeration for a function you just look it up and pick from the choices available.
- The base objects class that holds all the Python data types, tuple, list, str, etc.
- Everything not listed above resides under the Parent class FBPlug, this class has two direct children FBComponent and FBProperty, all other classes in MotionBuilder are children to FBComponent and FBProperty, basically this is this is where all the classes you will be using reside in.

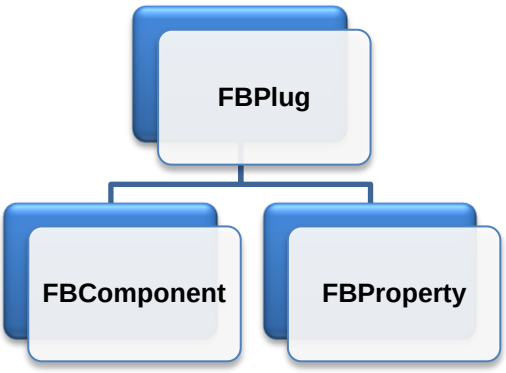


Figure 1 | Python Class Structure

To really simplify Object Oriented programming remember whatever classes sit above the current class you are, you can use those classes functions and attributes on your class object.

For example:

If you have an object of type FBModel, you can use the functions and attributes from FBBox, FBComponent and FBPlug. If you have an object of type FBBox, you can use the functions and attributes from FBComponent and FBPlug, you can no longer access FBModel functions and attributes.

To clarify you cannot use sibling or children's class functions or attributes, you can only move up the hierarchy, not down or sideward's (technically up then down).

TIP: If you create an object in the Python Editor and run the Python command dir on the object, you can print the tuple results, this way you can visually see what functions and attributes it can assess.

```
from pyfbsdk import *

lApp = FBApplication()
        print dir(lApp)
```

Using the auto completion in the Python Editor will give you an equivalent graphic list to choose from as well.

Finding the class that has the functionality you need

A really fabulous question people ask is so how do I find the class that holds the functionality I need in it, well there is no direct answer because there is not documentation that says "I need this, go to this class", so the best way to go about this, is to get familiar with the search capabilities in the documentation. The way I approach this problem is I take a look at what I am trying to do in the UI and see what it is called there, then I put that key word in the search and see if anything comes up, then I start reading the functions in the classes that appear.

For example:

If I wanted to create a filter, I would look for the key word 'filter' and which would come up with FBFilter and FBFilterManager, I would then read the documentation at the top of each class, look at the functions and attributes and look at the examples to see how to go about creating one.

Let's try something more obscure let's say I wanted to show the current camera in a renderer, so I put in the search field 'show camera', and I start

looking through the results, and I find the enumeration 'FBViewingOptions', hmm this seems promising, so then I copy the word 'FBViewingOptions', because I now need to find which class sets this enumeration, we are working backwards. This comes up with the class FBRenderer, where it has the function, 'SetViewingOptions', which sets this enumeration. So all I need to check now is if I can create an instance of this class by itself, yes is has a constructor.

Over time you will get better at finding the classes that you need for the right occasion, just keep in mind we try to be consistent with our naming conventions between the UI and the SDK.

3.13 Working with Callbacks

A callback is something you can setup to watch a specific behaviour, and when that behaviour is performed you call some code.

For example: You assign me to watch a front door, and when people come in the door I hand them a visitor pass, without me watching the door people would just come through the door and never get a visitors pass. Kristine = callback, door = MotionBuilder, visitor pass = is the behaviour I was to add.

In MotionBuilder, let's say I want everyone to follow a special naming convention for their scene files, so I would set up a callback to watch anytime a new file is created, when this happens I call a function in my code to save the file with a special name.

One might ask if this could cause a performance problem, callbacks are efficient in MotionBuilder so it doesn't slow the application even though it is always watching what you told it to watch. This is necessary to set up, you need something to constantly watch what you are looking for, this is where callback come into play.

Another example is when someone changes a property, you want to maybe give them a warning…but how will you know when the user changes the property, they could do it at any time during their usage, and you want the warning to be triggered right after they change it, not at a hard coded point of time, so in other words you need the programmer sitting there watching the user, until you saw the user change the property then you can call your original function.

A common use of callbacks in any application is in the UI, such as a button clicks, my callback said when someone clicks this button, go and do this functionality. You need your callbacks to always watch for the clicking of the button, so in other words it's a reactive behaviour based on the particular behaviour.

Examples

Setting up a callback when a new scene file is created

```
from pyfbsdk import *

def OnFileNewCallback(scene, event):
    print "New File Created, make we could do some scene set up here"

lApp = FBApplication()
lApp.OnFileNew.Add(OnFileNewCallback)
```

3.14 More Control over Loading and Saving Scenes

There are some classes that can be used in conjunction with FBApplication to provide a visual interface for users to perform file-based actions.

'FBFilePopup' Class and 'FBFolderPopup' class

- These classes create a UI dialog box for selecting files or directories/folders.
- FBFilePopup has a Style property that lets you specify whether you are using it for loading or saving a file.
- For both classes, when Execute() is called, it waits until the OK or Cancel button is pressed. It returns a boolean indicating which button has been pressed (true for OK and false for Cancel). Then certain properties get populated. For example, FBFilePopup's FileName represents the file that was specified, while FBFolderPopup's Path represents the folder that was selected.

Examples

Opening and selecting a folder from the browser window

```
from pyfbsdk import *

lFp = FBFolderPopup()
lFp.Caption = 'FBFolderPopup example: Select a folder'

lRes = lFp.Execute()

if lRes:
    lMessage = '%s has been selected' % ( lFp.Path)
```

Saving the current scene file with a new name or path

```
from pyfbsdk import *

lFp = FBFilePopup()
lFp.Style = FBFilePopupStyle.kFBFilePopupSave

lFp.Caption = "FBFilePopup example: Save file"
lFp.Filter = "*.fbx"
lFp.Path = r"C:\Users\wengn\Desktop"

lRes = lFp.Execute()

if lRes:
        lApp = FBApplication()
        lApp.FileSave(lFp.FullFilename)
```

The 'FBFbxOption' Class

In the last lesson we learned about the class FBApplication, this did all the basic things from the File menu, but what about that more complex things that happen when you save, open, merge or append a scene, you get this big 'Save Options' dialog box that appears. This dialog box allows us to merge, append or discard certain things in the scene when saving or opening, how do we control these different scene elements in Python?

Instances of FBFbxOptions can be used in the methods of FBApplication to specify which scene elements to load from or save to a file. FBApplication.

- The constructor of FBFbxOptions takes a boolean value as an argument to determine whether the instance will be used for saving or loading:
  FBFbxOptions(True) - used to load scene data from a file.
  FBFbxOptions(False) - used to save scene data to a file

- Enumeration class FBElementAction determines what operations you want to do with scene elements.

  FBElementAction.kFBElementActionSave --- save the element
  FBElementAction.kFBElementActionAppend --- append to current scene elements
  FBElementAction.kFBElementActionMerge --- merge elements from the file in the current scene
  FBElementAction.kFBElementActionDiscard --- do not save or load this element

For every scene element, you need to explicitly specify operation you want by assigning a FBElementAction enum to it.

Example

```
from pyfbsdk import *
app = FBApplication()
app.FileNew()
filename = 'C:\Program Files\Autodesk\MotionBuilder 2013\OpenRealitySDK\scenes\PlasticMan.fbx'
loadOptions = FBFbxOptions(True)
loadOptions.Materials = FBElementAction.kFBElementActionAppend
```

It corresponds to the “Element” section you see in the “Open” Or “Save” option window below:

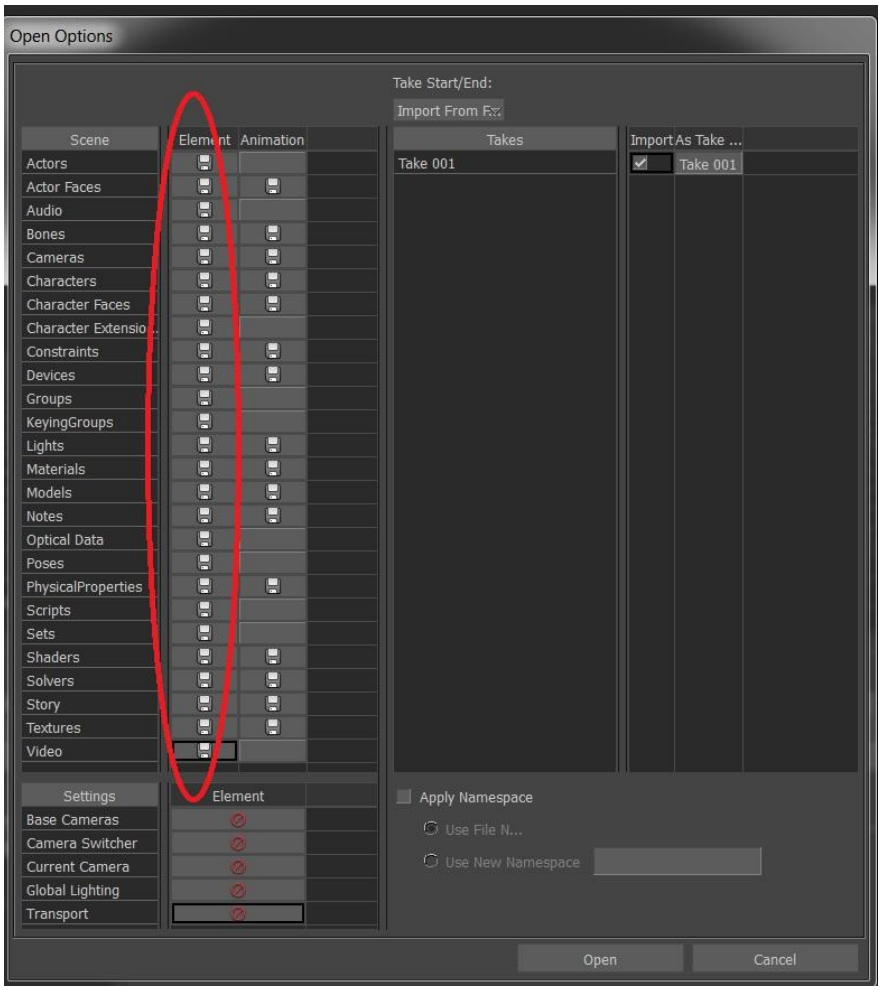


Figure 2 | Element action settings

- Element animation settings options

Not all elements type have animation associated with them, if they have, the saving and loading of animation data can be toggled through the use of several boolean (FBPropertyBool) properties.

Example

```
from pyfbsdk import *
app = FBApplication()
app.FileNew()
filename = 'C:\Program Files\Autodesk\MotionBuilder
2013\OpenRealitySDK\scenes\PlasticMan.fbx'
loadOptions = FBFbxOptions(True)
loadOptions.BonesAnimation = True
```

It corresponds to the “Animation” section you see in the “Open” Or “Save” option window below:

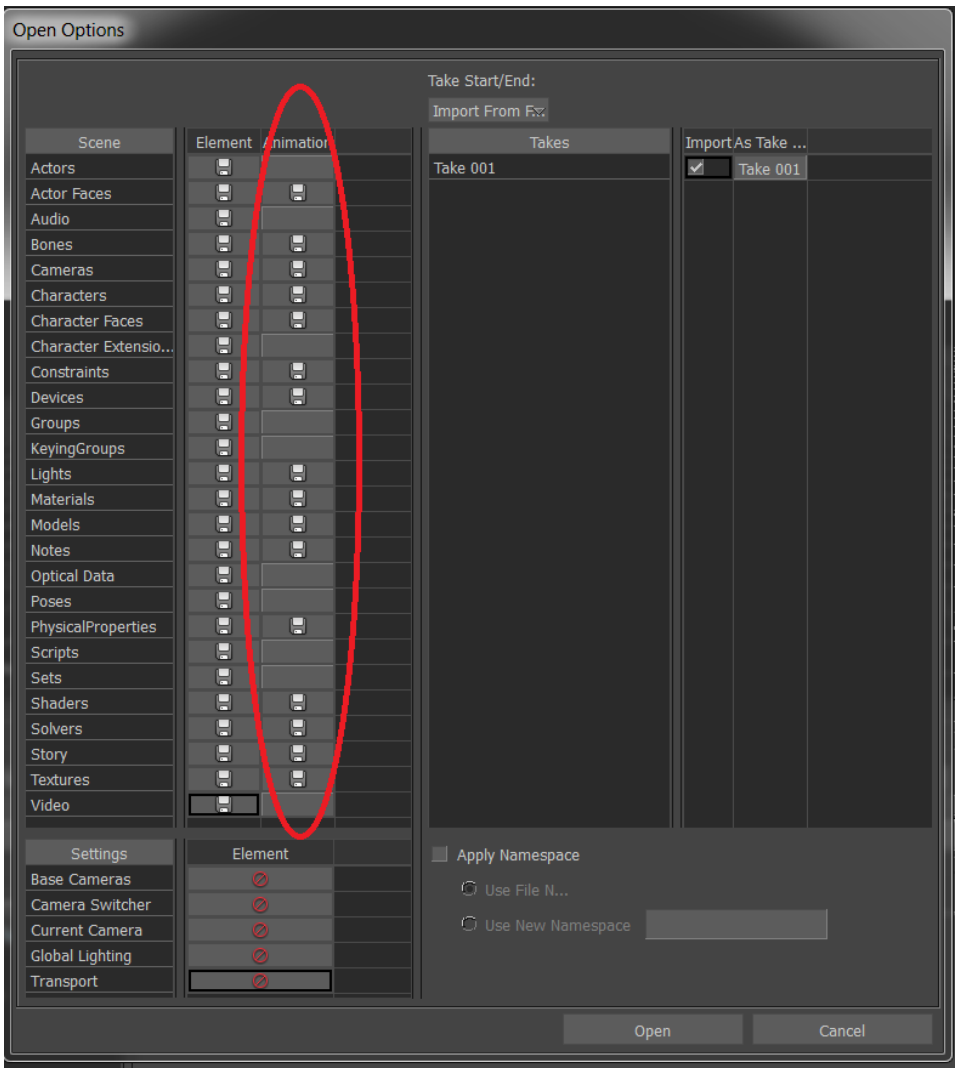


Figure 3 | Element animation settings

Saving or Loading Character Animation

Figure 4 | Load and Save Character Animation

LoadAnimationOnCharacter

In the “Character Controls” Tab, “File” menu, there is a submenu ‘Load Character Animation’, you can execute this functionality in Python by using the function FBApplication::LoadAnimationOnCharacter(), and customize it with the FBFbxOptions.

For example:

```
# Saves out the character and rig animation
# if there are multiple characters per file you will need to change to
accommodate.
```

```
from pyfbsdk import *
app = FBApplication()

loadOptions = FBFbxOptions(True)
loadOptions.TransferMethod = FBCharacterLoadAnimationMethod.
kFBCharacterLoadRetarget
loadOptions.ProcessAnimationOnExtension = True

plotOptions = FBPlotOptions()
…

FBApplication().LoadAnimationOnCharacter(A_Car, GoodCharacter, loadOptions
plotOptions)
```

SaveCharacterRigAndAnimation

Simliarly, in the “Character Controls” Tab, “File” menu, there is this submenu to ‘Save Character Animation’, you can execute this functionality in Python by using the function FBApplication::SaveCharacterRigAndAnimation(), and also customize it with FBFbxOptions. I won’t list any example here.