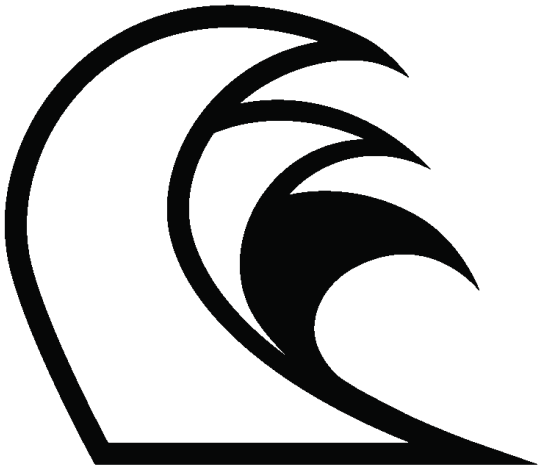
Programming in MotionBuilder || Focusing on Python
Autodesk MotionBuilder 2013

Autodesk Developer Network
May 2012
Module 6: Characters



AUTODESK®
MOTIONBUILDER

Contents

Programming in MotionBuilder || Focusing on Python

Autodesk Developer Network

Module 6: Characters

Agenda

- Takes and Layers
- Common Character Workflows
- Less Control more automation, another way to plot
- Miscellaneous Character Information
- Character Poses

7.0 Takes and Layers

A take is a level of animation in your scene. A take's start and end determines when the Timeline indicator starts and stops.

In the Transport Controls, the Action timeline can display the current length of a take, or a zoomed section of the current take. The Start and End fields define the current take length, and the Zoom Start and End time codes define parameters for a zoomed section of a take on the Action timeline. You can also use the Takes settings to manage your takes.

One key thing to note about takes is a model is present on every single take in a scene; it's the animation that differs on each take. Each take can consists of layers called Animation layers,

Takes

Getting list of Current Takes

Current take list is available in FBScene::Takes, this returns a list of FBTake.

Getting the current take

Working with FBSystem::CurrentTake

```
from pyfbsdk import *

lSystem = FBSystem()
scenetakes = lSystem.Takes
ldesiredTake = 'Take 001'

for curtake in scenetakes:
        if curtake.Name == ldesiredTake:
                lSystem.CurrentTake = curtake
```

Creating a take and adding it to the scene

When you created a take with the Python FBTake constructor, you need to append the new take to the scene using FBSystem().Scene.Takes.append(FBTake(“My new take”)) or FBSystem().Scene.Components.append(FBTake("My new take")).

Copying a take

Once you have a handle to take you can use the function CopyTake inside the FBTake class to replicate the take.

Deleting a take

Use the function FBDelete on the take for example:

```
myTake.FBDelete()
```

But it will not remove the take that's already been added into the scene, to remove the take, you need to use

```
FBSystem().Scene.Takes.remove(myTake)
```

Working with Takes

The new ClearAllProperties method enables you to clear the animation on all properties associated with a take. The ClearAllProperties method accepts a boolean parameter that enables you to clear the animation on all properties (false) or clear the animation on the currently selected properties (true).

For example, to clear the animation on all properties, you would call

```
FBSystem().Take[0].ClearAllProperties(False).
```

Creating custom properties on Takes

When creating custom properties on takes for the property to get saved you have to have animation on your take, because by default takes are not saved in the FBX file if there is no data on it.

```
from pyfbsdk import *

def OnTakeChangeCallback(scene, event):

    for takes in FBSystem().Scene.Takes:
        if not takes.PropertyList.Find('IsCurrentTake'):
            lProp = takes.PropertyCreate( "IsCurrentTake",
FBPropertyType.kFBPT_bool, "Bool", False, True, None)
            break
    for takes in FBSystem().Scene.Takes:
        if takes.PropertyList.Find('IsCurrentTake'):
            if takes == FBSystem().CurrentTake:
                takes.PropertyList.Find('IsCurrentTake').Data = True
            else:
                takes.PropertyList.Find('IsCurrentTake').Data = False
```

```
lScene = FBSystem().Scene
lScene.OnTakeChange.Add(OnTakeChangeCallback)
```

Layers

In MotionBuilder, a layer is a level of animation in a scene. You can have multiple layers in a scene and make changes to one layer without affecting the others. Layers are especially useful for adjusting motion capture data without altering the original data.

For example, you can add a layer for a cube's translation animation, creating a second level of animation on top of the Base Layer. The data on the Base Layer never changes because the animation is on a separate layer.

The animation on the two layers — the Base Layer animation and the additional layer of Translation animation — combine and apply to the cube in real time. When you are satisfied with your changes, you can merge your layers with the Base Layer.

All the layers functions and attributes sit inside the FBTake class, there is not a separate FBLayer class.

Creating Animation Layers inside of your takes

Here is an easy way to set this up, buy just grabbing the CurrentTake and creating the layer there, however the take that you want to create layers on does not have to be the current one to have this happen.

```
from pyfbsdk import *

lSystem = FBSystem()
lSystem.CurrentTake.CreateNewLayer()
```

Removing a Layer

To remove a layer, you need to pass the index of the layer you would like to remove in as the parameter in the function.

```
myTake.RemoveaLayer(1)
```

Working with Layers

GetLayerName: get the name of a layer at the specified index.

SetCurrentLayer: set the current layer for the take.

GetCurrentLayer: get the current layer for the take.

GetLayerCount: get the layer count.

```
from pyfbsdk import *

lSystem = FBSystem()
count = lSystem.CurrentTake.GetLayerCount()
print "count", count
print lSystem.CurrentTake.GetCurrentLayer ()

lSystem.CurrentTake.CreateNewLayer()
lSystem.CurrentTake.SetCurrentLayer(1)

count = lSystem.CurrentTake.GetLayerCount()

print lSystem.CurrentTake.GetCurrentLayer ()
```

7.1 Common Character Workflow

Working with characters is a big part of MotionBuilder, Python very nicely incorporated access to working with characters including the main workflows so that you can automate all the aspects of it. Below we will go through the process of starting from a model all the way to characterizing the model to plotting the animation on to the control rig and then the skeleton.

Adding a character asset to your character model and characterizing it

The Character asset helps you to map out the structure of your character model so that it can be animated in MotionBuilder. Once you have completed this mapping process, you 'activate' the character model by characterizing it. Characterizing lets MotionBuilder know that this character model is ready to be animated.

All major character animation features in MotionBuilder, including Control rigs and animating in the Story window, require a characterized character.

So let's create a character in our scene that we can work with:

```
from pyfbsdk import *

lChar = FBCharacter ("Character")
```

The function SetCharacterizeOn lets you choose what you would like to choose as the character type when you characterize your character your options are a biped (two leg walking character, i.e. human) which you set to true and is the default or quadruped (four leg walking character, i.e. animal) which you set to false.

```
test = lCharacter.SetCharacterizeOn( True )
print lCharacter.GetCharacterizeError()
```

This is the error that could print out depending on what your specific issue is:

```
ERROR:
The characterization process could not be completed because some
required nodes are missing or do not follow naming conventions.
Click Ok to view a list of the missing nodes.
The following required nodes are missing and need
to be mapped using the Character Definition pane.

  - Hips
  - LeftUpLeg
  - LeftLeg
```

- LeftFoot
- RightUpLeg
- RightLeg
- RightFoot
- Spine
- LeftArm
- LeftForeArm
- LeftHand
- RightArm
- RightForeArm
- RightHand
- Head

This means that our character mapping inside of our Character Definition pane is not set up for our character we can do this manually or set it up through python.

Character Mapping

If you have a skeleton that doesn't follow MotionBuilder's bone naming conventions, MotionBuilder cannot recognize the structure of your model's skeleton. Before you can start animating, you need to manually define each bone.

Adding a Control rig and customize it to fit your character animation needs.

Control rigs are an animation tool that make it easy to control and position your character model.

This creates a control rig, if you want to create your control rig that is FKIK then set it to True, if you want to have a control rig that is only IK, set this function to false.

```
lChar.CreateControlRig(True)
```

Retarget your animation between Character models.

Although not a necessary step in creating animation within MotionBuilder, during animation projects, the Character model you are using might change. Instead of re-creating the animation on the new model, you can simply retarget the animation from your existing file.

Point one animation from one character to another:

```
lChar.InputCharacter = lScene.Characters[1]

lScene.Characters[0].InputType =
FBCharacterInputType.kFBCharacterInputCharacter
```

```
lScene.Characters[0].ActiveInput = True
```

Plot your finished animation to your model's skeleton

Depending on the animation features that you are using to create your character animation, plotting may consist of plotting from your Control rig to your character model skeleton, or plotting the tracks in the Story window to a single take.

Whatever method you use to animate, the finished result must be plotted to the skeleton of your character model before you export it. Save your plotted model as an *.fbx* file.

```
lPlotOptions = FBPlotOptions()
lPlotOptions.PlotAllTakes = False
lScene.Characters[0].PlotAnimation (
FBCharacterPlotWhere.kFBCharacterPlotOnControlRig, lPlotOptions )
```

7.2 Less Control more automation, another way to plot

Section 7.1 shows you how you can control each step of the process, but if you know exactly what you want to do, and just have to do it multiple times, you can use the FBApplication function FileBatch, which allows you to batch load animation files and plot the animation on a character's control rig or onto the skeleton.

```
# Setting all the plot options
# Creating FBPlotOptions Constructor
lPlotOptions = FBPlotOptions()

# Setting all the boolean values for the plot
lPlotOptions.PlotPeriod.SetSecondDouble  = ( 1.0 / 24.0 )
lRotationFilter = lPlotOptions.RotationFilterToApply.kFBRotationFilterNone
lPlotOptions.PlotAllTakes = False
lPlotOptions.PlotOnFrame = True
lPlotOptions.UseConstantkeyReducer = False
lPlotOptions.PlotTranslationOnRootOnly = False
lPlotOptions.PreciseTimeDiscontinuities       = False

# Setting all the batch options

# Creating FBBatchOptions Constructor
lBatchOptions = FBBatchOptions()

# Look at all the characters in the scene.
lBatchOptions.Character = lScene.Characters[0]

# Setting the InputFile Format to be FBX Kaydarea Animation Only
lFileFormat = lBatchOptions.InputFileFormat.kFBBatchFileFormatFBX

# Setting the input and output directory path
```

```
lBatchOptions.InputDirectory = lFp.Path
lBatchOptions.OutputDirectory = lFp.Path

# Set load  type
lType = lBatchOptions.ProcessType.kFBBatchProcessTypeLoad

# Setting all the boolean values for the batch
lBatchOptions.FramAnimation = True
lBatchOptions.KeepCharacterConstraint = False
lBatchOptions.KeepDummyBones = False
lBatchOptions.OverwriteScaling = False
lBatchOptions.WriteTranslation = False
lBatchOptions.WriteRate = False
lBatchOptions.UseSingleTake = False
lBatchOptions.UseBatchSuffix = False
lBatchOptions.StartAnimationAtZero = False
lBatchOptions.SkeletonFile = ""
lBatchOptions.PlotToControlSet = False
lBatchOptions.PlotToCharacter = False

# Start Batch Process
lBatchStatus = FBBatchStatus()
lStatus = lApplication.FileBatch( lBatchOptions, lPlotOptions )
print lStatus
```

7.3 Miscellaneous Character Information

Bounding Box of a Character

MotionBuilder does not have any collision system shipped with it. To get updated bounding box values, you can force the evaluation of the deformations by rendering the frame you want to evaluate. See the script below that demonstrates this workaround. Just load the FBX file and run the script. You will see in the console that the bounding box values we fetch are updated.

```
from pyfbsdk import *

def GetBoundingBox():
    lMesh = FBFindModelByName( "Plasticman:PlasticMan" )
    if lMesh:
        lMin = FBVector3d()
        lMax = FBVector3d()
        lMesh.GetBoundingBox( lMin, lMax )
        print "----"
        print "Min:", lMin[0], lMin[1], lMin[2]
        print "Max:", lMax[0], lMax[1], lMax[2]

def RenderFrame( pFileName, pFrameStart, pFrameEnd ):
    lVideoGrabOptions = FBVideoGrabber().GetOptions()
```

```
    lVideoGrabOptions.TimeSpan = FBTimeSpan(FBTime(0, 0, 0, pFrameStart),
FBTime(0, 0, 0, pFrameEnd))
    lVideoGrabOptions.OutputFileName = pFileName
    lVideoGrabOptions.ViewingMode =
FBVideoRenderViewingMode.FBViewingModeCurrent
    FBApplication().FileRender( lVideoGrabOptions )


FBPlayerControl().Goto( FBTime( 0, 0, 0, 1 ) )
FBSystem().Scene.Evaluate()

RenderFrame( "c:\\render1.tga", 1, 1 )
GetBoundingBox()

FBPlayerControl().Goto( FBTime( 0, 0, 0, 15 ) )
FBSystem().Scene.Evaluate()

RenderFrame( "c:\\render1.tga", 15, 15 )
GetBoundingBox()

FBPlayerControl().Goto( FBTime( 0, 0, 0, 30 ) )
FBSystem().Scene.Evaluate()

RenderFrame( "c:\\render1.tga", 30, 30 )
GetBoundingBox()

FBPlayerControl().Goto( FBTime( 0, 0, 0, 45 ) )
FBSystem().Scene.Evaluate()

RenderFrame( "c:\\render1.tga", 45, 45 )
GetBoundingBox()
```

Copy Cloning

If you want to copy an object in your scene you can use copy function in the module copy or you can use the Clone function in the module pyfbsdk in the class FBCharacater, this applies to elements, materials, textures and characters, etc:

```
from pyfbsdk import *
import copy

sys = FBSystem()
scene = sys.Scene

# Test built in __copy__ method that dispatches on Clone

# character
if len(scene.Characters):
    c = scene.Characters[0]
```

```
c2 = copy.copy(c)
```

or

```
from pyfbsdk import *

lCharacter = FBCharacter('Character')
lCharacter.Show = False

#Clone everything
lCharacter2 = lCharacter.Clone()
lCharacter2.Show = True
```

Character Extensions

Character Extensions can be created to associate any type of object with your character, whether it is an extra limb, a weapon your character carries, a camera, or a spot light that follows your character around. Any type of object or property that you want to control and key along with your character can be added as a Character Extension.

```
from pyfbsdk import *

lCharacter = FBSystem().Scene.Characters[0]
print lCharacter.Name

lExtension = FBCharacterExtension('Rig Objects')

lAdd = lCharacter.AddCharacterExtension(lExtension)
lName = 'Mia:Mia_left_leg'
pModel = FBFindModelByName ( lName )
lExtension.AddObjectProperties(pModel)
```

7.4 Character Poses

The class FBCharacterPose is fairly new to Python; it was exposed in the last couple of releases. Before this addition there was only the class FBPose, which was not used to perform character poses manipulations

Here is a way to find the poses in the scene, by going through FBScene's Components, these items are also FBPose objects.

```
from pyfbsdk import *

for lComp in FBSystem().Scene.Components:
    if lComp != None and lComp.Is(FBPose_TypeInfo()):
        print lComp.Name
```

Setting Character Pose Options

When pasting poses using FBCharacterPose, you need to first set the pose options using the FBCharacterPoseOptions.

For example

```
from pyfbsdk import *

poseName = "Walk"
lApplication = FBApplication()
lScene = FBSystem().Scene
lSelectedCharacterList = []

for lChars in lScene.Characters:
    if lChars.Selected:
        lSelectedCharacterList.append( lChars )

if len( lSelectedCharacterList ) == 0:
    lSelectedCharacterList.append( lApplication.CurrentCharacter )

for lCharacter in lSelectedCharacterList:
    poseOptions = FBCharacterPoseOptions()
    poseOptions.mCharacterPoseKeyingMode =
FBCharacterPoseKeyingMode.kFBCharacterPoseKeyingModeFullBody
    poseOptions.SetFlag(FBCharacterPoseFlag.kFBCharacterPoseMirror, True)
    poseOptions.mMirrorPlaneType = FBMirrorPlaneType.kFBMirrorPlaneTypeZY

for pose in lScene.CharacterPoses:
    if pose.Name == poseName:
        #found our pose
        targetPose = FBCharacterPose(pose.Name)

        print "found targetPose: '%s' in scene" % targetPose.Name

        targetPose.PastePose(lCharacter, poseOptions )
```