Motion Capture Retargeting

Final Report

April 23, 2015

Ahmed Beshry
Ulvi Ibrahimov
Devon Sigurdson
Jeremy Unger

Outline

Our programs allow a user to take any .bvh motion capture file and retarget it to any previously rigged .fbx file in MotionBuilder. This is of course dependent on the rigging of the character to be correct with a basic skeletal model. Through our implementation we hoped to allow users to be able to perform this task with as little interaction with the actual MotionBuilder program as possible. Once this was done, we decided to add a few extra features which will be detailed below.

Background

Motion capture data retargeting is a very powerful tool. It enables applying motion capture data amongst many models. Motion correspondence often relies directly on retargeting. Motion correspondence is the ability to link two separate motion capture files together. The implications of this for animation are massive. Imagine taking a model and applying a capture of a figure running, which could transition to swimming and so on. It would enable using a library of mo-cap files, rather than capturing new files to fit the animator's need.

The task of retargeting motion capture data does not have a best practice established. As it currently stands there are many different method of completing this task. Each method has its own difficulties. The two main technologies for motion capture data are online motion capture, and offline. Online is real-time data retargeting, and offline is the process where the mo-cap data is retargeted later. Some approaches are semi-automatic requiring some human interaction, like renaming bones, or providing initial posing. Another aspect of motion capture retargeting is retargeting data from a human to a non-human skeleton. This creates many issues, as a dog does not have the same number of bones, or structure of a human.

"In order to accomplish motion retargeting and transition in different skeletons, we have to define the correspondence of the source and target skeletons. Then after aligning their initial poses, we can retarget the motion data between them. Moreover, motion transition can be done by constructing a meta-skeleton which

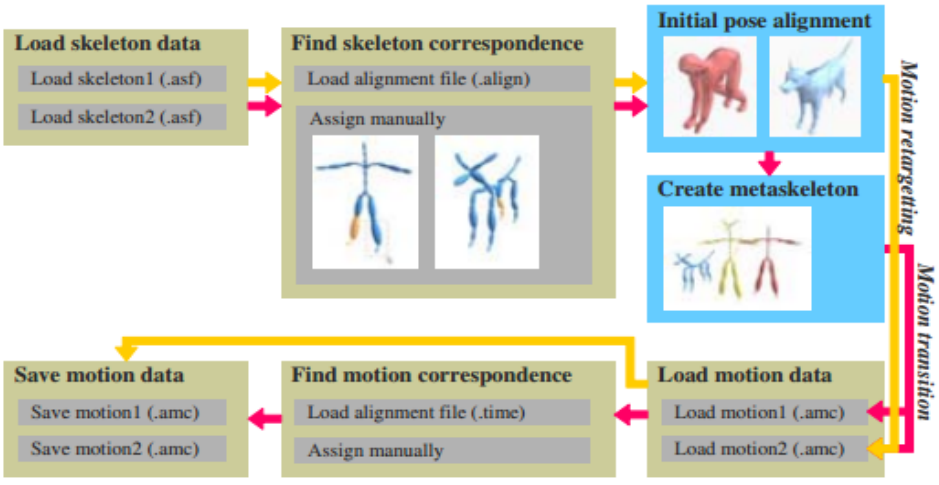


Figure 1

contains both skeleton structures, then blending similar frames and bone lengths of these two motions." (*Hsieh M.*)

The retargeting of different figures can lead to issues, as more interpretation of the model must be done. This leads to the challenge of retargeting different articulated figures in real time, as human oversight is often required.

The mapping of a captured limb to the limb of a target can be done as either one to one (a leg mapping to a leg) or many to one (two legs mapping to a fin or tail). This can often create difficulties that require human interaction. Once these skeletons are lined up and meshed, than the motion capture file can be applied.

Online motion capture data does not have the advantage of any human modification, as the transfer is done in real time. One technique for "on-line motion retargeting (*OMR*) is to track the given reference end-effector trajectory x1(t), and the secondary goal is to imitate the pattern of joint angle trajectory src(t) as much as possible." (*Choi K*). This technique measures the angles between joints in the motion capture file and maps the angles to the target. For example, if at frame 83, the motion capture file had a 90-degree angle between its forearm and upper arm then the mesh would be converted to match these angles at that scene.

In order for the transforms to occur live, the motion is recorded and passes a stream of joint angle vectors. These angles are outputted to the mesh and interpreted live allowing the mesh to follow what the motion capture file is feeding it. Along with the angle between joints, the motion capture data must also translate its orientation, and endpoints of each joint. This approach has limitations. It was not used to retarget skeletons of different types, but just for similar targets. It was successful it transferring data from a human skeleton to another human skeleton with different lengths. It was not used for transferring from a human to a bird or other creature, but did have the advantage of being fully automated and completed in real time.

One other approach is that "the input motion is processed by a motion-analysis algorithm, which classifies the motion, determines its structure and identifies the constraints as an integral part of the structure." (*Savenko*) This allows for interpretation of what task the capture is completing. Utilizing motion analysis allows for decomposed of tasks, so they are smaller and more manageable. If the data represents a model walking, and then knowing the biomechanics of the movement can allow for better animation. The obvious issue that arises from motion analysis is its inability to handle retargeting from varied sources, as well as misunderstood tasks.

Intermediate Skeletons

Motion Capture Retargeting

Another technique used in some motion capture retargeting implementations is the use of an intermediate skeleton to smooth the transition between bodies. As seen in the paper by *Monznai* et al. using an intermediate skeleton allows a user to better compensate for the differences between the captured body and the target skeleton. Motion capture files do not have a set number of points attributed to their skeletons. Because of this, users will occasionally run into the problem where the skeleton that the user is retrieving the information from has far too many points to translate properly to the skeleton they wish to animate. Using an intermediate skeleton, the information can be processed in a way, as to which allows the skeleton to move in a fashion that closely imitates the original skeleton while containing less join information. In some cases, this may prove to be difficult because you must translate the end position of two bones in one skeleton to the resulting end point of a single bone using vector mathematics. However, once this has been calculated, the retargeting to the end skeleton will be almost trivial.

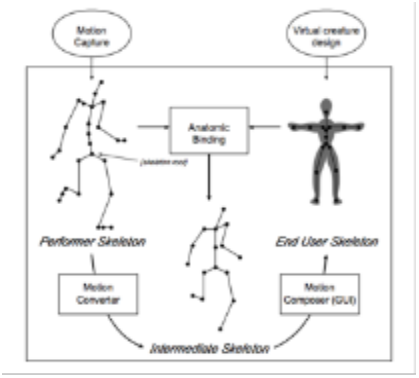


Figure 2

Previous research in the field of retargeting has established methods of retargeting using Mesh Graphs. This has been done by generating a mesh graph from the target mesh by using sphere packing, or a using a Reeb graph. This two techniques result in a Mesh graph that allows for a skeletal retargeting based on a mesh we see that a mesh graph must be obtained (*Borenstein, 2011*). The key to the retargeting problem lies in constructing a graph from the target mesh. An analysis of the two most successful retargeting methods currently are to follow.

Rig retargeting using a constructed Reeb graph is implemented by Martin Poirier and Eric Paquette. Poirier and Paquette propose an approach of using a multiresolution topology graph that is extracted from the target mesh's data (*Poirier & Paquette, 2009*). With this extracted one dimensional graph, Poirier and Paquette then use an extracted one dimensional graph from the skeleton data to do a symmetrical comparison between the two graphs. The comparisons provide information on where the skeleton graph will require scaling and translations to better fit the mesh. After the comparison is made, the system commits to a retargeting.

The following figure depicts the algorithmic overview of the mesh retargeting using a Reeb graph generated from the 3D mesh (*Poirier & Paquette, 2009*). Step B is a rather simplistic step, simply because we are translating a one dimensional set of information into a one-dimensional graph. Step A in contrast possesses high complexity due to the translation from a three dimensional mesh to an equally informative one dimensional graph (*Poirier & Paquette,*

2009). This complexity requires the previously researched Reeb graph generation techniques. A Reeb graph is a graph that tracks topological changes in a three dimensional structure (*Cole-McLaughlin, Edelsbrunner, Harer, Natarajan, & Pascucci, 2004*). The complexity of this step results in limitations and erroneous behaviours with respect to the graph output, depending on the input. With many arcs in the Reeb graph, the smaller arcs from overly detailed meshes are smoothed out, resulting in a highly effective, symmetric and easy to match mesh graph *(Poirier & Paquette, 2009*).

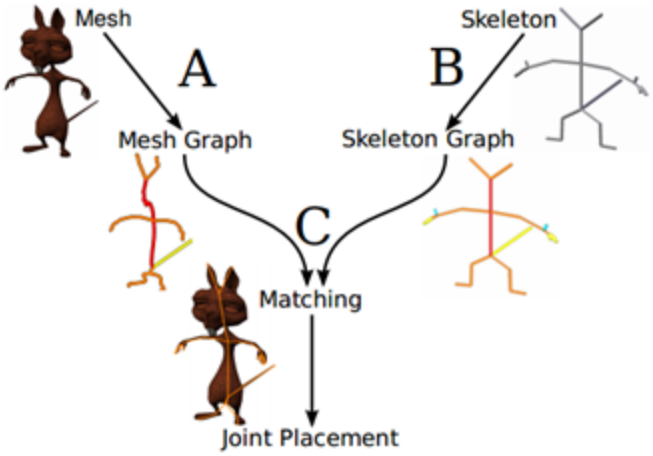


Figure 3

With the two graphs now in usable one dimensional representation, they are then tagged and scaled using another algorithm. In this approach, there is a matching of the skeleton arc to the mesh arc at a low resolution level in the multiresolution topography of the mesh graph *(Poirier & Paquette, 2009*). Once the first resolution is matched, they delve into finer resolution levels and increase the matching of finer skeletal fragments, such as smaller limbs like fingers. This is a recursive process, and done until we have reached the finest resolution layer and it is matched with the mesh graph. This technique results in a very high speed retargeting technique that is highly effective in comparison to older techniques (*Yamane, Ariki, & Hodgins, 2010*). A mesh with 2428 faces and 51 joints was retargeted and skinned in just 1.8 seconds, while one with 27,232 faces and 136 joints was retargeted and skinned in only 32.6 seconds, showing the effectiveness of this retargeting strategy (*Poirier & Paquette, 2009*).

The next automatic rigging strategy was made with the intent on being user friendly, to the extent that children may use it, while also being highly effective and time efficient. This strategy is called Pinocchio and was developed at the Massachusetts

Figure 4

Institute of Technology (*Baran & Popovic, 2007*). The method of implementation, like the previous system, requires a graph from the mesh. This mesh graph is developed in Pinocchio by using a sphere packing technique. This technique first requires a medial surface approximation, and based on the sorting of these approximated points, placement of spheres in locations furthest from the surface takes place. Spheres continue to fill the mesh, following the rule that no sphere must engulf the center of another sphere. This continues until all spheres have been added that do not contain any other centers, and whose radius is the distance to the surface (*Baran & Popovic, 2007*). Once a sphere packing is complete, a graph construction then takes place upon that mesh. The Pinocchio system will add an edge between any two sphere centers that had overlapping sphere boundaries. This will result in an overwhelmingly complex skeleton that is then simplified by having many of its edges reduced (*Baran & Popovic, 2007*).

Once the sphere packing constructed graph is completed, based on the sphere centers within the mesh, the simplification techniques take place. This simplification is dependant on the provided initial skeleton. Based on the number of joints within the skeleton, the constructed mesh graph will be reduced to that number of joints as w ell, removing all edges that are not in close correspondence to the edges of the skeleton. Lastly, the joints of the skeleton are translated and scaled to be the perfect fit for the mesh (*Baran & Popovic, 2007*).

Use of constraints in Retargeting Techniques

One of the techniques for applying motion created for one figure to another figure with identical structure is using spacetime constraints, which was introduced by Michael Gleicher. This is accomplished by requiring the basic features of the motion – for example that the feet touch the floor when walking – to be identified as constraints. If the constraints are violated when the motion is applied to a different figure, an adaptation to the motion is found that re-establishes the constraints in a manner that fits with the motion.

The core of this retargeting method is a numerical solver that computes an adaptation to the original motion. Many of the previous retargeting methods consider only individual frames. However, this method proposes a solution to the problem: space-time constraints. The idea behind this method is simple: What is the best motion that meets a specified set of constraints?

The configuration of an articulated figure is denoted as a vector that concatenates position for the root of the hierarchy and the angles of its joints denoted by q. A motion is a vector-valued function that provides a configuration given a time. Retargeted motion is the original motion plus differences: $\mathbf{m}(t) = \mathbf{m}_0(t) + \mathbf{d}(t)$. (*Gleicher, 1998)*

After defining the constraints, these are the steps to retarget motion to another articulated figure:

1. Begin with an initial motion with identified constraints.

2. Find an initial estimate of the solution by scaling the translational parameters of the motion, and then adding a translation to define the center of scaling. This translation is computed by finding the constrained displacements of the scaled motion for the target character, interpolating these values, and smoothing.

3. Choose a representation for the motion-displacement curve based on the frequency decomposition of the original motion.

4. Solve the non-linear constraint problem for a displacement that when added to the result of step 2 provides a motion that satisfies the constraints.

Below is the 10-frame motion applied to a smaller ladder-climbing figure. Hand is constrained to be attached to handhold. A) is the original motion capture data and D) shows the discussed method applied to it.

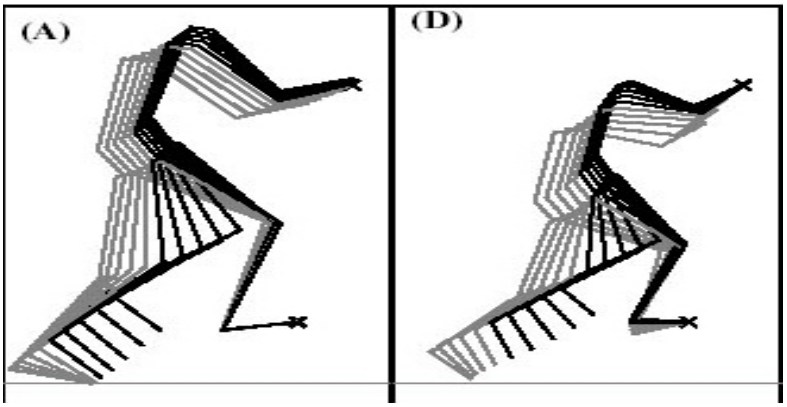


Figure 5

Another constraint-based approach is casting the motion editing problem as a constrained state estimation problem based on the per frame Kalman filter framework. (*Tak & Ko, 2003)*

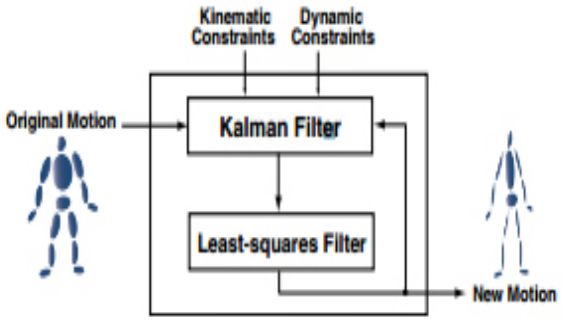


Figure 6

The method works as a filter that sequentially scans the input motion to produce a stream of output motion frames at a stable interactive rate. This is called per-frame algorithm.
Constraints are solved by black box composed of the Kalman filter and least-squares filter.

The method requires three type of constraints that force proper motion retargeting. Kinematic constraints provide an effective way to modify a given motion to achieve dynamic balance. Torque limit constraints force the algorithm to modify the given motion such that the joint torques of the new motion are within the animator-specified limits. The momentum constraints are derived from Newton's second law, which states that the rates of change of the linear and angular momenta are equal to the sums of the resultant forces and moments acting on the figure, respectively.

Once the above constraints are formulated, the task of modifying the motion is accomplished by Kalman Filter.

Another method that is proposed by Komura & Shinagawa & Kunii relies on the inner structure of human body. It uses the data collected by Delp (*Delp* 1990) that includes the attachment sites of 43 muscles on each leg, physiological parameters such as the length of tendons, range of joint angles. The upper half of the body is composed of the chest, head, upper arms, lower arms, and hands. The lower half has the pelvis at the top, and each leg consists of the femur, tibia, patella, talus, calcaneus, and toes.

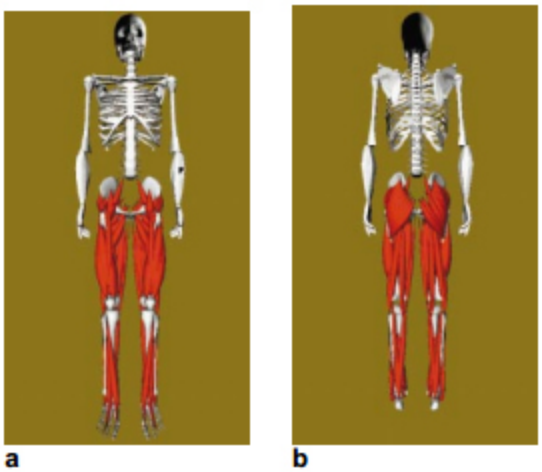


Figure 7

Muscle force from the joint torque and the profile of the motion are predicted from moment arm and force exerted by muscle. The balance of the human body is defined by a function that evaluates the stability of the posture. The zero moment point (ZMP) is used to define such a function.

The method is also based on spacetime constraints. A dynamic system of the whole human body including the muscles of the entire lower extremities is presented. The physical differences of individuals, such as muscle strength and size, can be handled easily. Such a model is useful to analyze and simulate actual human motions. (*Komura & Shinagawa & Kunii*)

Implementation

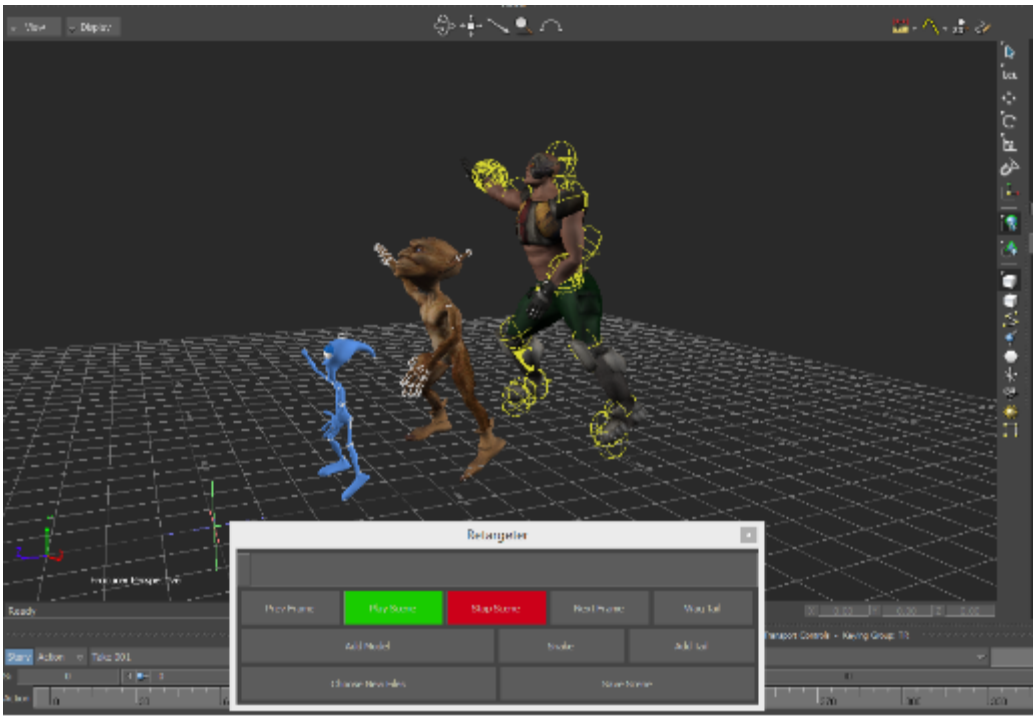


Figure 8

Implementation of General Retargeting and UI

The methods of our implementation are a limb based retargeting approach. That is that each limb that is properly named from the BVH motion file, is retargeted to the FBX model skeleton. This allows for a size independent retargeting approach due to the automatic size manipulation approach of MotionBuilder retargeting. The BVH file is loaded, and upon loading the BVH file, a new prompt will request the loading of the FBX file. The FBX folder automatically opens to the folder containing sample model FBX files that come with MotionBuilder. Once the FBX is loaded, the remainder of the UI, which is to be discussed further, is loaded. The retargeting of simple bipeds is done immediately after loading the FBX and BVH files. This retargeting is done on a limb by limb base. That is, each limb from a properly named BVH file is retargeted to an FBX file with corresponding limb names. The addJointToCharacter function is the primary characterizing function called on the BVH file to allow for retargeting.

```
def addJointToCharacter ( characterObject, slot, jointName ):

        myJoint = FBFindModelByLabelName(jointName)

        if myJoint:

             proplist = characterObject.PropertyList.Find(slot + "Link")

        proplist.append (myJoint)
```

Figure 9

Once this function is called on the BVH file, this function allows for each properly named limb to be prefaced with the "BVH:" prefix. Now that each limb has the BVH:limb prefix, a characterization of the BVH motion capture file can take place. The simple one line SetCharacterizeOn function can now work, now that the proper naming convention is taking place on the BVH file. This allowed for the limbs of the BVH file to become a character in the scene, which is critical for the future processes, because it will be this character that is retargeted to another character in the scene. This process is scene in the following code segment:

```
lBipedMap = (('Reference', 'BVH:reference'),
             ('Hips','BVH:Hips'),
             ('Tail', 'BVH:Tail'),
             ( 'LeftUpLeg', 'BVH:LeftUpLeg' ),
             ( 'LeftLeg', 'BVH:LeftLeg' ),
             ( 'LeftFoot', 'BVH:LeftFoot'),
             ( 'RightUpLeg', 'BVH:RightUpLeg'),
             ( 'RightLeg', 'BVH:RightLeg'),
             ( 'RightFoot', 'BVH:RightFoot'),
             ( 'Spine', 'BVH:Spine'),
             ( 'LeftArm', 'BVH:LeftArm'),
             ( 'LeftForeArm', 'BVH:LeftForeArm'),
             ( 'LeftHand', 'BVH:LeftHand'),
             ( 'RightArm', 'BVH:RightArm'),
             ( 'RightForeArm', 'BVH:RightForeArm'),
             ( 'RightHand', 'BVH:RightHand'),
             ( 'Head', 'BVH:Head'),
             ( 'Neck', 'BVH:Neck'))
```

Figure 10

Now that a new character is in the scene, one that has been characterized based on simple nodes from a motion capture file, the retargeting can take place. Motion Builder has an input feature that takes in a character as the input for another character. This only can occur if the character has been properly named and characterized, which resulted in the previous steps of the algorithm. The FBX character that is in the scene, contains a skeleton, this skeleton must also follow a naming convention. Depending on if the proper naming was used, the previously characterized BVH skeleton can now be retargeted. The FBX character is given an InputCharacter type of the BVH characterized character.

```
fbxCharacter.InputCharacter = bvhCharacter

        fbxCharacter.InputType = FBCharacterInputType.kFBCharacterInputCharacter

        fbxCharacter.ActiveInput = True
```

Figure 11

This allows for the retargeting of the character using the motion builder retargeting techniques. This retargeting automatically resizes the BVH skeleton limb to the targeted FBX limb size, and follows the BVH animation patterns. This limb resizing works on the skeletal level therefore if an initial BVH is doing any motion, this can be scaled properly to models of all varying sizes and characteristics. The skeleton matching the FBX model is shown with an overlap of a skeleton the exact same size as the sample Aragor.FBX model in the motion builder directory. This example is shown with the first initial load of the character model. This overlap of the skeleton shows how the varying initial size of a skeleton does not impede on the models abilities to interact in the scene, and allows for arbitrary character sizes to still work properly. Now that the simplistic retargeting mechanism is in place, we can take these steps much further to allow for arbitrary retargeting types and features, such as retargeting a snake or an animals with tails.

The next useful feature we had implemented was a user interface that allows for the user to not have to use or learn the motion builders complicated user interface. The Simple 600 x 200 pixel user interface allows for all of the features required for loading and retargeting as well as all of our additional features. The user interfaces simplest features are the simple scene frame navigation features, such as the simple play, pause and frame navigation features. In addition, a slider is added atop these buttons to allow for custom scene navigation. We created a createButton feature that allowed the creation of buttons to take much fewer lines of code and allowed the code to be easier to read.

```
def createButton(text, color):

        newButton = FBButton()

        newButton.Caption = text

        newButton.Style = FBButtonStyle.kFBPushButton

        newButton.Justify = FBTextJustify.kFBTextJustifyCenter

        newButton.Look = FBButtonLook.kFBLookColorChange

        if color != None:

         newButton.SetStateColor(FBButtonState.kFBButtonState0, color)

        return newButton
```

Figure 12

The createButton function extracts all of the of the redundant reused features of the button creation methods and allows for this extracted method to be called everytime a button is created for code cleanliness. The creation of all the buttons is done first, and upon the click of any of the buttons, the corresponding function is called. The play and pause of the scene simply follow scenePlayer.Play() and scenePlayer.Play() features respectively of Motion Builder. The next user interface feature is the Choose New Files button, which allows for the reloading of the BVH and the FBX files into the scene. This is done by simply reloading the loadFiles function that is called at the beginning of the program. This will go through and retrigger the pop ups that require the BVH and FBX files to be loaded from the users directories.

The imperativeness of having a properly named BVH file resulted in our next feature. That being the ability to rename limbs on the fly, with a drop down menu that lists the existing limbs within the scenes BVH file. The next input section allows for the required limb names to be entered and the following section to right will save it. Next are two buttons for the snake feature, add model and the wagging the tail function that will be discussed in further detail in a later section. Lastly is a saving button that allows for this scene to be saved as an FBX to be revisited later and reloaded exactly as it was left. The sections to follow will focus on retargeting to arbitrary models and extra innovative features.

One of the goals of our method was to allow for animating characters that don't match our model exactly and to be able to extend the FBX files that were imported. The applications of this are that you would be able to adapt FBX files to meet the needs of the animator. We wanted a way so a BVH file that perfectly met the needs of an animator but was lacking a small portion, such as a tail, would still be able to be used. we were able to accomplish this using MotionBuilders API.

```
def addTailResponse(control, event):
    global tailadded
    if(tailadded==False):
        hipRef = FBFindModelByLabelName('BVH:Hips')
        tail = FBModelSkeleton('BVH:Tail')
        tail.Parent = hipRef
        tail.Show = True
        tail.Translation = FBVector3d(0, 0, -5)
        tail.Scaling = FBVector3d(0.5,0.5,0.5)
        hipRef2 = FBFindModelByLabelName('BVH:Tail')
        tail2 = FBModelSkeleton('BVH:Tail2')
        tail2.Parent = hipRef2
        tail2.Translation = FBVector3d(0, 1, -7)
        tailadded=True
```

Figure 13: Python code for adding a tail located behind models hip

Our method adds a tail directly behind the hips of the BVH skeleton. This allows for modifying a BVH file such that it could be matched to a FBX file in way that the new bone, in this case a tail, could be animated. Our application went one step further by demonstrating the possibility of animating the tail.

```
            global bvhList
            global wagcnt
            for comp in FBSystem().Scene.Components:
                #print comp.LongName
                if (comp.LongName == "BVH:Tail"):
                    comp.Selected = True
                    comp.Rotation.SetAnimated(True)
                    print wagcnt
                    if ( wagcnt == False):
                        comp.Rotation = FBVector3d(30, 0, 0)
                        wagcnt = True

                    elif( wagcnt==True):
                        comp.Rotation = FBVector3d(-30,0,0)
                        wagcnt = False


                else:
                    comp.Selected = False
```

Figure 14: Python code for wagging a tail

The tail will rotate up and down by 30 degrees from the origin. The animator selects the way button and the frames they wish to animate it at. MotionBuilders autokey ability creates the frames in between the keyframes such that the result is a smooth wagging motion. The animator can create a fast wag by setting many keyframes, or a slower on by choosing a large spacing between frames.

The ability to add a tail and animate it using our UI is merely a proof of concept that given enough time and resources it is possible to start creating a library of animations. Adding and animating a tail is just one application. This process could be done for animating anything. For example, if one was wishing to animate Blackout, the helicopter transformer, the ability to use a humans BVH file for most of the animation would be ideal. Rather than having a add tail and wag feature, our application could be modified to animate the helicopter blade when necessary. This means a human bvh can easily be adapted to a humanoid figure to save time for the animators. The file could also be saved and exported for future use and storage leading to a library of animation files.

Figure 15: Blackout from the live-action transformers film.

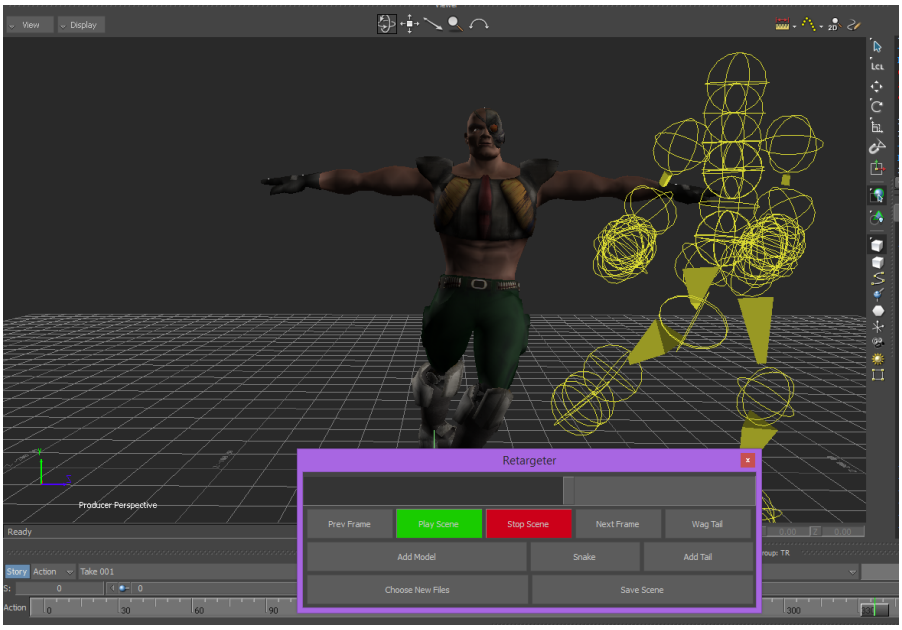
Our implementation allows user to load multiple bvh files into one scene. The purpose of this functionality is testing retargeting for different characters and see the differences. The following method is the implementation. The method asks for a fbx file. After loading the character into the scene, the originally chosen motion is retargeted to the new character. Additionally, characters are spaced based on their size. Because characters that are larger will travel farther distances, they are offset by that distance. Throughout the animation, if multiple characters are loaded, they will travel through each other to reach their final destination.

```
def addModel(control, event):
    global app, bvhCharacter, scenePlayer
    fbxName = fbxPopup()
    app.FileMerge(fbxName, False)
    fbxCharacter = FBSystem().Scene.Characters[len(FBSystem().Scene.Characters) - 1]

    print 'Number of characters in scene = ', (len(FBSystem().Scene.Characters))

    fbxCharacter.InputCharacter = bvhCharacter
    fbxCharacter.InputType = FBCharacterInputType.kFBCharacterInputCharacter
    fbxCharacter.ActiveInput = True
    #scenePlayer.LoopStop = sceneLength
    scenePlayer.SetTransportFps(FPS)
    print "The scene length will be set to " + str(sceneLength)
    FBSystem().CurrentTake.LocalTimeSpan = FBTimeSpan(
    FBTime(0, 0, 0, 0, 0),
    FBTime(0, 0, 0, sceneLength, 0)
    )
```

Another one of the functionalities that we implemented is retargeting snake motion capture file to different characters. To avoid inappropriate leg and arm movements, legs and arms joints movements are not added to the motion that the model receives. The snake button in the interface, prompts user to load the fbx and bvh files. After loading them, the lims are found and unparented. This allows us to remove the arms and one of the legs from the animation and



have appropriate snake motion for different characters. In fact, if we had a model rigged (such as a snake) that didn't have any arms or legs added in the rigging, the motion may even look somewhat natural. As it stand however, the model only has reduced movement because of the constraints on the foot that the motion wasn't added to. Currently, MotionBuilder's API is a lot more restrictive with python programming then it is with C++. If we were programming in C++ we may have been able to find a way to remove that constraint, but as it stands now, it doesn't seem as though it is possible.

renaming.py

Late into the development cycle of the application we realized that we overlooked a critical part of the implementation we were working on. We began testing using .fbx models that were not provided to us by MotionBuilder. These models however, continuously produced errors and would not be accepted as input by our retargeting program. Something was obviously wrong so we quickly had to come up with a solution so that our product would be more accessible and modular.

It was discovered that our program only accepted models that had been characterized within MotionBuilder. For a model to be characterized, it must already be in compliance with the basic skeleton model that MotionBuilder recognizes. To be able to be characterized, a rigged model must have at least all of the following 15 named nodes:

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

Once all of these nodes have been correctly named, a simple press of the characterize button will format the .fbx character to be ready for use in our main program.

To assist in situations where a user may not know the naming conventions that the original model artist used to name the joints of the character, or when the character has been named in a different language, the joint that is currently selected will be highlighted on the screen. (Figure XII) To rename the bone, the user simply has to select the correctly named MotionBuilder equivalent within the second dropdown menu, and hit the rename button. The change will be reflected in both the model window as well as the character dropdown menu.

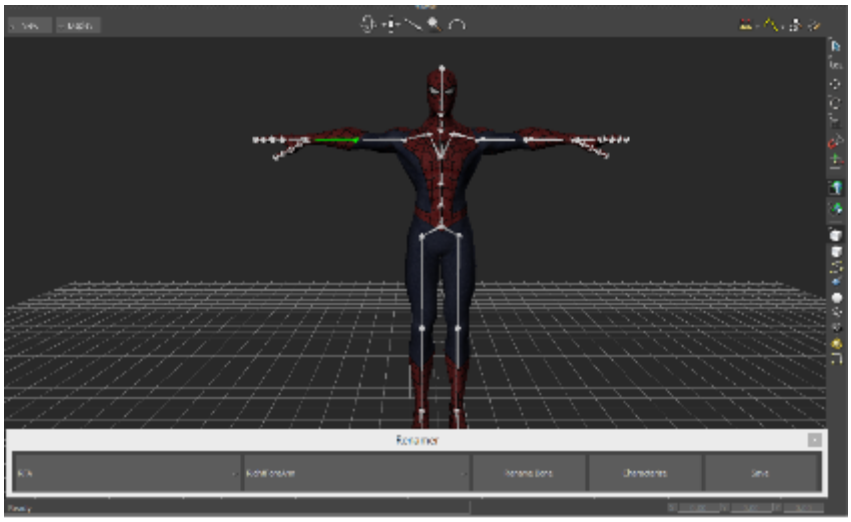


Figure 18: The renamer.py program in use

Future Work

Bugs

Currently in our implementation, there is a major bug in that reopening files that we in use the last time our program was used will cause the models to load in very awkward positions. This is caused because the last position the model was in is mistaken for the T-pose of the model. Because the model was not actually in T-pose however, the motion capture overlay causes the model to over rotate it's limbs for each motion and the limbs remain comically over rotated for the entire animation.

In the future, this could be compensated for by ensuring that no memory of the pose the model was left in remains. Whether that is to be done through resetting the animation to frame zero or ensuring to close the .fbx file properly before exiting however, is yet to be determined.

Another thing that we would like to address later is the overlapping of models when multiple are loaded into the scene. Currently, models are offset based on size, but that usually means that through the animation, the models will eventually run through each other to reach their end pose. To correct this, we would like to automatically offset the starting position of new models by a specified distance as they are loaded in.

renaming.py

In the future we would like to expand the renaming program to allow it to be able to characterize a larger variety of models as well as prevent it from creating artifacts in its characterized models that could lead to awkward looking animations one the motion capture file has been attached to the model. This could be accomplished through a T-pose checking subroutine that detects whether the model is in a proper T-pose. In MotionBuilder, a proper T-pose is defined as a model having it's arms parallel to the plane of the floor and having it's hips facing in the positive Z axis. The T-pose also requires the models legs to be as close to vertical as possible.

Given more time, this could be enforced in a step after all of the bones have been renamed. If the bones have been named properly, we can find the spatial coordinates of the nodes in each limb and determine whether the constraints listed above are correct. If they are not, we could use the ability to move a node (ie. the tail example) to correct these deficiencies. A simple rotational fix could be implemented to start, but eventually we could correct for bent elbows and other oddities in the original rigged model.

Another possibility for future expansion is adding the extensibility to be able to articulate motion capture files with more joints than the basic skeleton. MotionBuilder has the ability to characterize more complex skeletons such as those with fingers or shoulder joints, Adding these motions however would depend on the rigging of the of the model skelton and if these joints have been implemented in the skeleton.

Conclusion

Our software's first goal was achieving the ability to retarget a BVH file to an FBX file. We were able to achieve this with MotionBuilders API and their naming templating system. This worked with a great deal of success. From there we moved on to some less trivial goals. We should how our system would be able to remove limbs from a BVH file such that it would be able to retarget a non-human animation to a humanoid character. In the example we retargeted a snake motion to a human. After achieving retargeting from a non-human BVH to a human we looked at adding limbs to a human BVH. This was done to show how one could capture a humans motion and apply it to a non human FBX. We added in the ability to move and animate the tail to show how keyframe animation was possible using our system in combination with motion builder. This would easily be used for applying a human BHV to a kangaroo FBX for example. The last additional feature we added was the ability to add several models to the scene. Imagine trying to animate an army marching. With our system you could import all the models you wish and have the BVH file correspond to them. With a few clicks you have 5 men

marching in perfect unison. Our design is based around naming constraints. In order to overcome incorrectly named FBXs we developed a second script that allowed to properly name the components of an FBX such it was compatible with our file. Our system is self contained in its UI. It allow for loading multiple FBX models, BVH files, pause/play, scrolling through frames,exporting and adding and wagging a tail. We chose to self contain these so that the user has as little to do as possible. We decreased the ability of incorrectly naming objects through highlighting what bone was being renamed and providing a drop down name rather than entering the name manually. All in all our system provides basic retargeting in combination with less trivial aspects so that users can achieve their desired animation.

Bibliography

Ban, X., & Han, D & Jin T. (2011). Vector-Mapping Method for Motion Retargeting of the Virtual Articulated Figures and its Application. *Journal Of Software, Vol. 6, No. 12,*.

Baran, I., & Popovic, J. (2007). Automatic Rigging and Animation of 3D Characters. *ACM Transactions on Graphics*, 72:1 - 72:8.

Bharaj G., Thormahlen T., Seidel H.-P., Theobalt C. (2012). Automatically rigging multi-component characters. *Computer Graphics Forum 31*(2) 755–764. doi: 10.1111/j.1467-8659.2012.03034.x

Bindiganavale, R., & Badler, N. I. (1999). Automatic recognition and mapping of constraints for motion retargetting. *ACM SIGGRAPH 99 Conference Abstracts & Applications*, 234. doi:10.1145/311625.312123

Borenstein, E. (2011). *A Skeletal Motion Capture Library.* Worcester: Worcester Polytechnic Institute.

Celikcan, U., Yaz, I. O. and Capin, T. (2015). Example-Based Retargeting of Human Motion to Arbitrary Mesh Models. Computer Graphics Forum, 34: 216–227. doi: 10.1111/cgf.12507

Choi, K. (2015). On-line Motion Retargetting. *IEEE*. Retrieved from IEEE. http://graphics.snu.ac.kr/~kjchoi/publication/omr.pdf

Cole-McLaughlin, K., Edelsbrunner, H., Harer, J., Natarajan, V., & Pascucci, V. (2004). Loops in Reeb Graphs of 2-Manifolds. *Discrete & Computational Geometry*, 231-244.

Gleicher M. (1998). Retargeting Motion to New Characters. *Autodesk Vision Technology Center*, 33-42

Hecker C, Raabe B, Enslow R, DeWeese J, Maynard J, van Prooijen K. (2008) Real-time motion retargeting to highly varied user-created morphologies. *ACM Transactions on Graphics (TOG)*. 27(3).

Komura, T. & Shinagawa, Y. & Kunii, T (2003). Creating and retargeting motion by the musculoskeletal human body model. Tokyo:University of Tokyo.

Monzani, J.S., Baerlocher, P., Boulic, R., Thalmann, D. (2000). Using an Intermediate Skeleton and Inverse *XYZ Sympos*Kinematics for Motion Retargeting. *Computer Graphics Forum, 19*(3), 11-19. doi:10.1111/1467-8659.00393

N, N., & N, N. (2008). Motion Analogies: Automatic Motion Transfer to Different Morphologies. *ium* , 1-8.

Poirier, M., & Paquette, E. (2009). Rig Retargeting for 3D Animation. *CHCCS Graphics Interface*, 103-110.

Savenko, A. (2015). Using Motion Analysis Techniques for Motion Retargetting. *IEEE, 4*(2). Retrieved from IEEE.http://ieeexplore.ieee.org.login.ezproxy.library.ualberta.ca/stamp/stamp.jsp?tp=&arnumber=1028764

Tak, S. & Ko, H. (2003). A Physically-Based Motion Retargeting Filter. Seoul:Seoul National University.

Yamane, K., Ariki, Y., & Hodgins, J. (2010). Animating Non-Humanoid Characters. *SIGGRAPH Symposium on Computer Animation* , 169-178.

Zhao Y, Liu X G, Peng Q S, Bao H J. (2009) Rigidity constraints for large mesh deformation. *Journal of Computer Science and Technology, 24*(1): 47-55.

Figure X: http://transformers.wikia.com/wiki/Blackout_(Movie)