

Re-Targeting Character Pose and Animations

In Cinema 4D using Python

Bruce Kingsley

July 29, 2022

Summary

I've been working on Cinema 4D project (Python script) to enable me to transfer (re-target) poses and animations from one character to another. In this document, I'll refer to the characters created by DAZ Studio, though the uses of other characters from sources such as iClone and Mixamo apply.

I've been working on two different scripts. One to re-target joint animation and the second to re-target morphs. In this document I'll be focusing on retargeting joint animation specifically when the source character is not rigged the same as the target character.

The function of the script is not to be limited to just creating a mirror copy of the animation, but to also able to change the frames per second (FPS), reverse animation, frame skipping, extract a range of key frames, frames starting offset, extract specific joint chain animations, and add or mix with existing animations of the target character and the option to perform quaternion of target rotations to remove or reduce gimble noise.

Joint Mapping

To make the script universal and not be "hard coded" for specific source and target characters, I decided to have the script read a joint definition file created in Excel and exported as a .CSV text format. The idea is once a joint definition file is created for a specific source and target character, the file will not need to be changed.

The joint definition file provides the following information:

- Source and target joint names
- For each joint, define if joint position, or rotation, or both is used. Over 90% retargeting I only retarget joint rotations and only the hip joint I use position. This also enables me to overcome issues where the two character sizes are different. Some joint based face animations to do not completely transfer with rotations alone, though I find retargeting morphs have best end results.
- Each joint position and rotation gain and offsets.
- Each joint axis order.
- Each joint axis rigging offsets.

Re-Target Methods

The script is designed to enable two forms of animation re-targeting:

1. Transfer of joint animation from the source character's keyframe track.
2. Transfer of joint animation from the source character's current position and rotation at the specific keyframe interval provided. This unique mode is ideal for re-targeting animations of source character where the animation is being produced by IK-Joint chains driven by non-joint goals, an animation script, or the results of simulation dynamics.

In both methods, the script enables the option to set the keyframe interpolation of linear or cubic. Again, there is an option to perform quaternion of target rotations to remove or reduce gimble noise.

Pre-Targeting Functions

The script has two buttons that enable pre-targeting clean-up:

- Clear all target joint keyframes.
- Zero all target joint rotations. This results in the character being in its designed pose, such as the T-Pose or A-Pose.

Challenge of Re-Targeting Characters with Different Design Pose

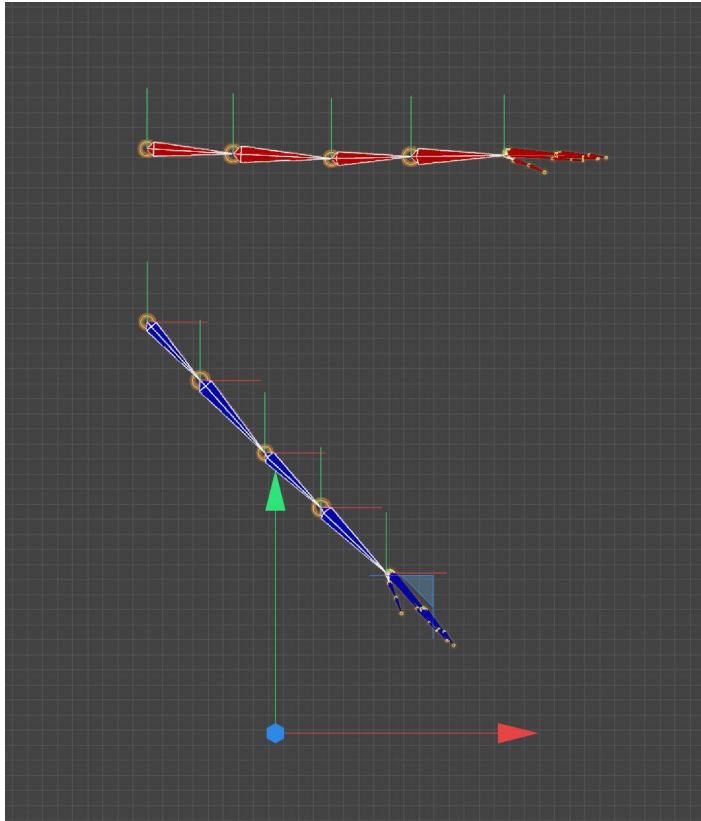
I think this was one of the areas where it took me the longest to figure out a solution. I've created the below illustrations demonstrated this challenge. In this example, I'm going to transfer the animation from a DAZ Studio Genesis 3 character to a Genesis 8 character.

The big difference between these two characters is the Genesis 3 joint rig is designed around a "T" pose, and the Genesis 8 character is in the "A" pose. To focus on the complexities of these differences, I focus on the arms.

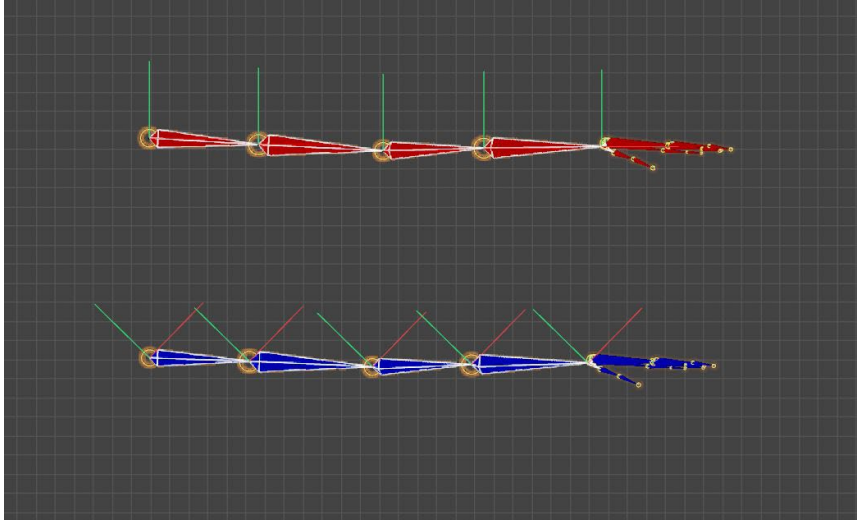
In the below illustrations:

- The Genesis 3 character is in **RED**
- The Genesis 8 character is in **BLUE**

Here is both the Genesis 3 and Genesis 8 characters in their designed root pose with rotation values all zero. You can see both characters joint world axis is the same (see the little green, red and blue lines point the same direction).

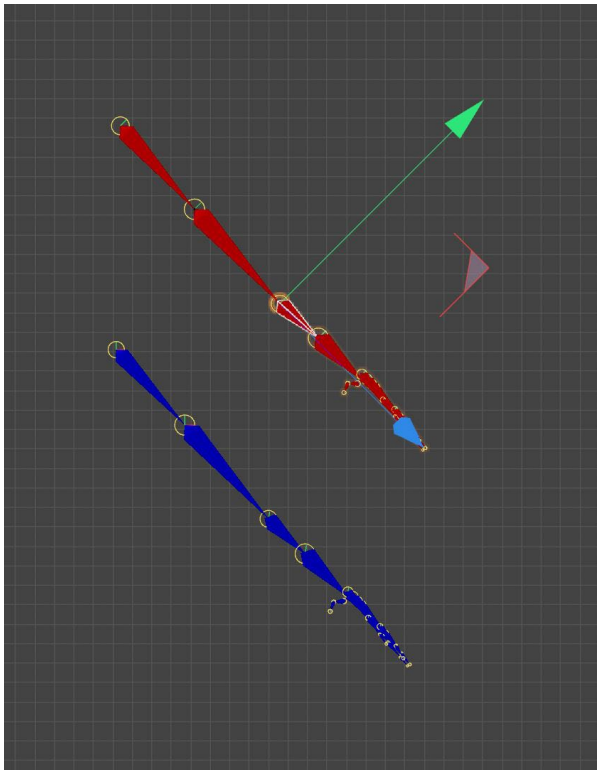


The Genesis 8 character's arm though is pointing 45.7 degrees down compared to the Genesis 3 characters. I read on many forums where users state all you have to is add or subtract 45.7 degrees to the shoulder joint. But it's not that simple.

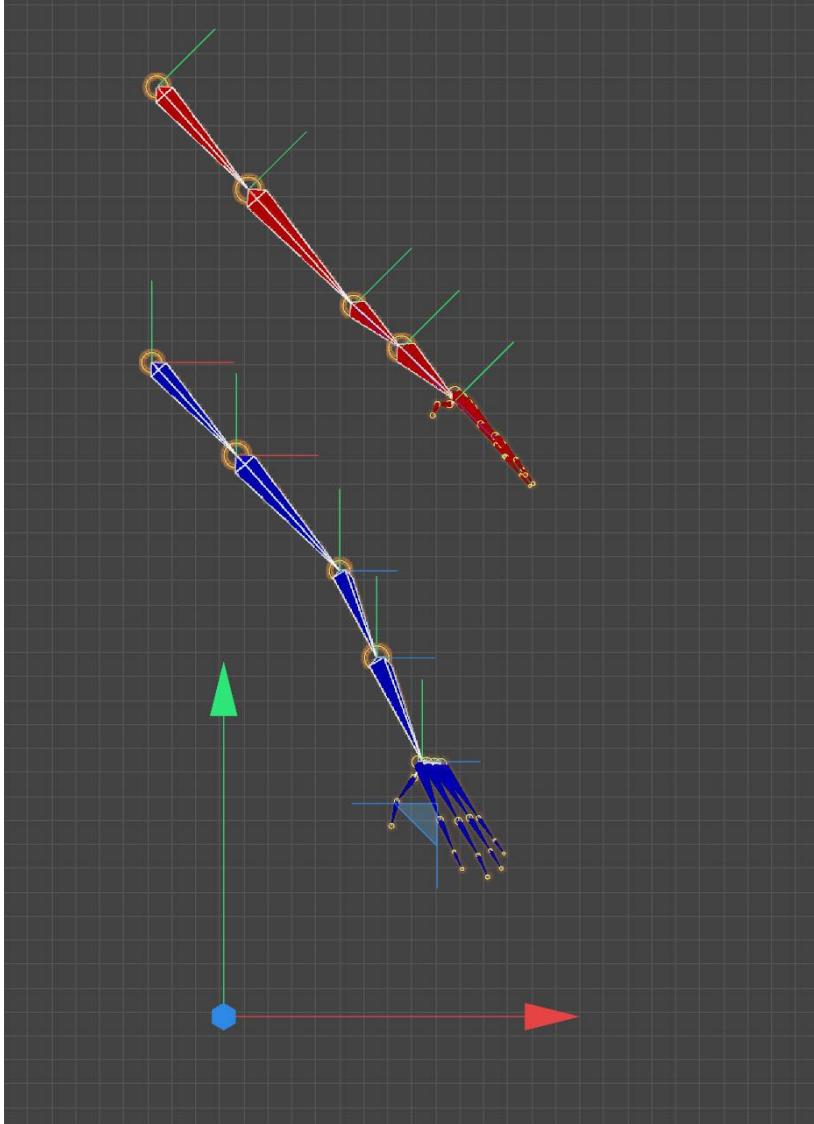


Side View

Above, I rotated the Genesis 8 character's shoulder 45.7 degrees in the Z-axis to make it look the same as the Genesis 3 character's arm. Notice the Genesis 8 character's joint axis lines, they're clearly not in the same orientation as the Genesis 3 character's arm in the same pose.



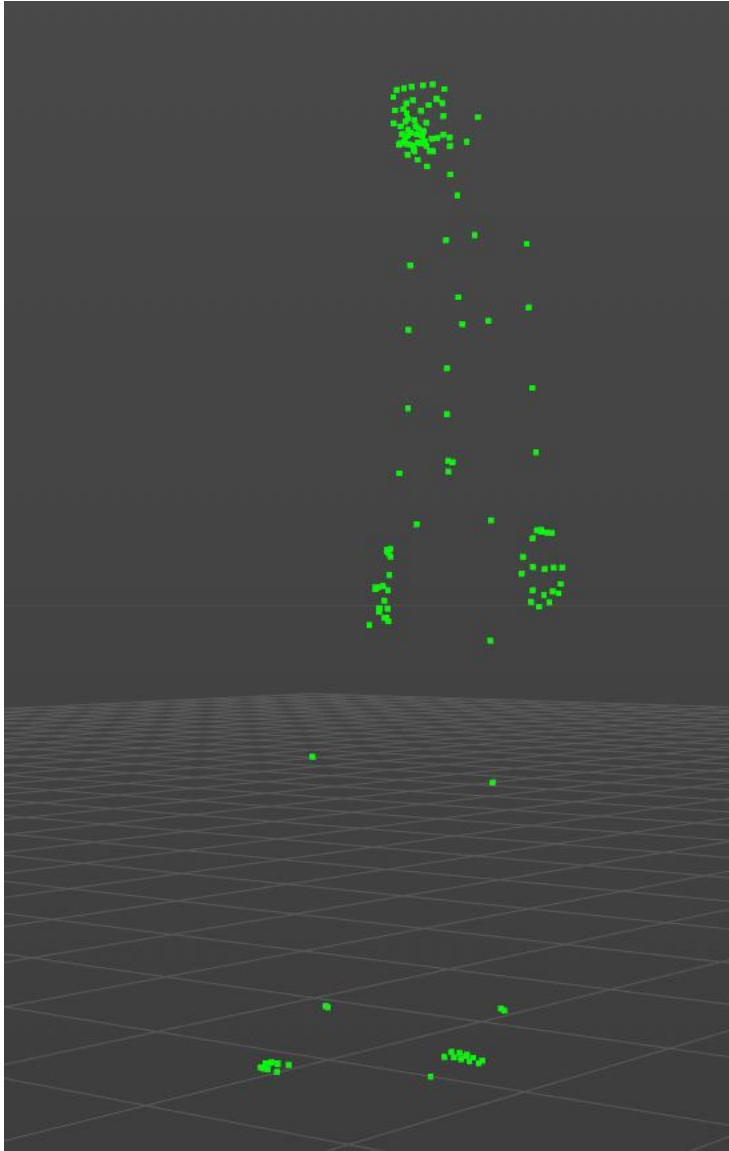
Above, let's look at what happens when try to rotate each character's forearm 45 degrees in the X axis. First the Genesis 3 character's forearm. Notice the axis of the Genesis 3 character's forearm joint.



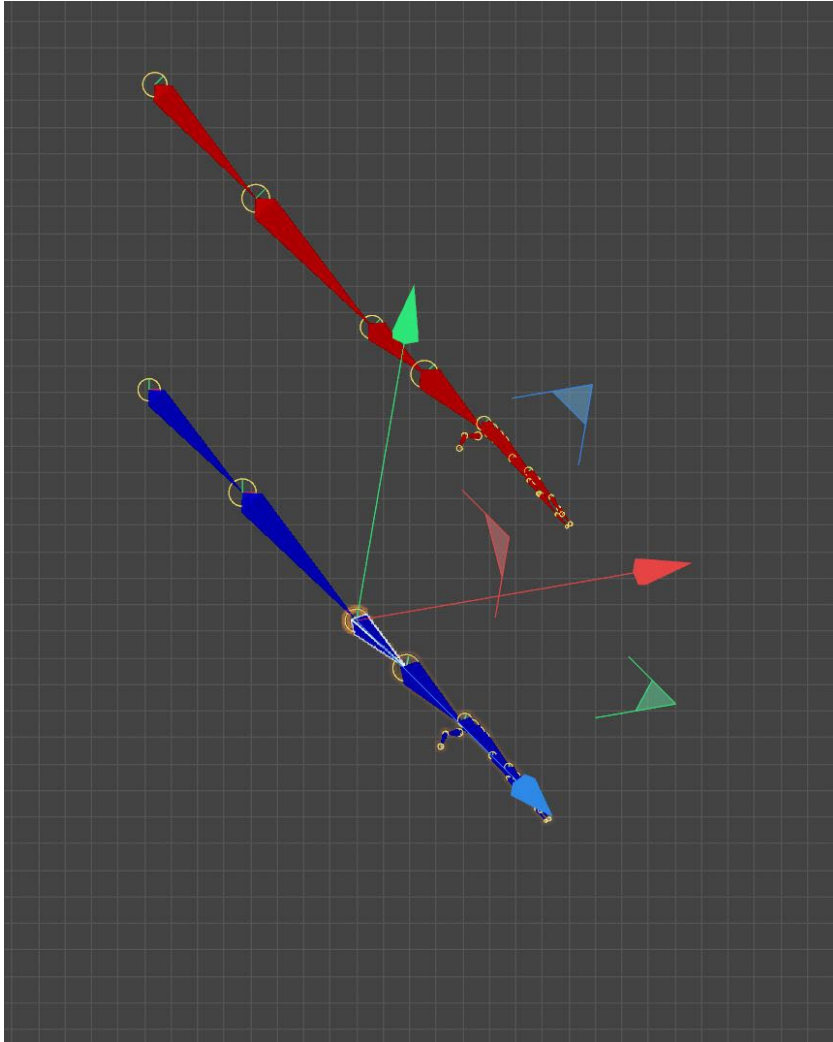
Next, I rotate the Genesis 8 character's forearm also 45 degrees in the X axis. As you can see the arms position is far from being the same. This is because once we rotated the shoulder, all the child joint rotational axis changes as well because they're all relative to their parent. Getting your head around this can really make your head hurt.

This is because rotational axes are not the same based on the joint rig design. Modifying one of the character's joint rigs to match can be done in Cinema 4D, but I don't want to modify my target character. I spent time perfecting just so I can get an animation from another that was designed differently. If I spend the time to modify the source character's rig, then the animation is now useless because the animation data is absolute joint rotation values but relative values.

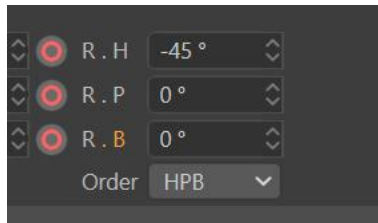
Then I wrote some Python code to create and animate a temporary rig made of null objects matched the source character but with the axis of the target. See below, each green dot is a joint. This was waste of time because I still would be faced with the same issue of the animation data being useless.



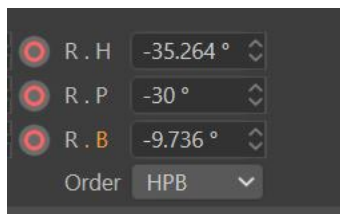
To better understand what I was faced with, I manually positioned the Genesis 8 character's entire arm to match the Genesis 3 character's arm. As you can see below, the two character's arm pose is the same but the Genesis 8 character's forearm coordinates are wildly different.



Below are the Genesis 3 character's forearm rotational values, as they would be stored in the animation track. A simple -45 degrees in the X-axis.



Below are the Genesis 8 character's forearm rotational values to pose the arm to match the Genesis 3 character's arm pose. This challenge wasn't going to be simple adding and subtracting numbers, and this would affect all joints from the shoulder down, as well as the legs that have smaller but still a different rig pose.



The solution I came up with was a means of just focusing on where each source joint rotational values are, and using Cinema 4D Python Matrix functions, calculate what the rotational values would be with the parent joint in a different axis orientation and the impact to its children joints.

The below Python function I created "CreateJointHPBOffsetHPB" does this.

In my re-targeting script, I'm not always directly reading an object, just animation keyframe values. Because of this, the function makes its own matrix from scratch, and only needs to be provided three variables:

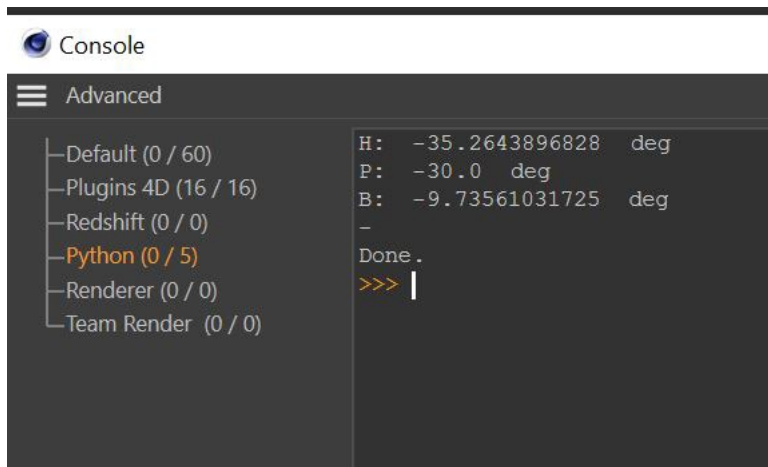
1. The source joint native rotational values in radius.
2. If the source joint is a child of joint that has axis offset different than the target joint, pass that value in degrees. If there is no offset, what goes in, comes out.
3. The last variable is a bool. True if it's the parent joint and False if it's a child joint.

The function returns the required rotational values in radius only, no position, to plug into the target character's joint, or without actually reading or writing to either joint object. This is important to me where I want the flexibility to create or modify custom animation values from Python such as my scripts animation keyframe mixing functions.

Below is the function in use, where I used the same forearm challenge example above.

```
# Loop through all joints per definition mapping .CSV file
axisOffset = c4d.Vector(h_axis_off, p_axis_off, b_axis_off)
newRot = CreateJointHPBOffsetHPB(sourceJoint.GetRelRot(), axisOffset, parent_joint)
targetJoint.SetRelRot(newRot)
c4d.EventAdd()printHPBVectorDeg(newRot)
c4d.EventAdd()
print "Done."
return
```

And here below, is the output values of the function. You can see it's spot on:



```
Console
Advanced
Default (0 / 60)
Plugins 4D (16 / 16)
Redshift (0 / 0)
Python (0 / 5)
Renderer (0 / 0)
Team Render (0 / 0)
H: -35.2643896828 deg
P: -30.0 deg
B: -9.73561031725 deg
-
Done .
>>> |
```

Below is my Python CreateJointHPBOffsetHPB function code:

```
def CreateJointHPBOffsetHPB(sourceHPB, axisOffset, isParentJoint):
```

```
    # Source values are the animated HPB radi values of the source joint
```

```
    # Offset values are the axis offset in degrees to apply to the animated values to match the target object
```

```
    # Returns a new HPB vector that can be applied to the target joint
```

```
    # Get the target joint original matrix rotations.
```

```
    mTargetOriginalTransforms = c4d.utils.HPBToMatrix(sourceHPB, c4d.ROTATIONORDER_XYZGLOBAL)
```

```
    # Rotation axis offset of the parent joint to subtract from the child joints after the transform
```

```
    # so we can use the source HPB values without having cascading offsets
```

```
    offsetXValueInvert = axisOffset.x * -1
```

```
    offsetYValueInvert = axisOffset.y * -1
```

```
    offsetZValueInvert = axisOffset.z * -1
```

```
    mX = c4d.utils.MatrixRotX(c4d.utils.DegToRad(axisOffset.x))
```

```
    mY = c4d.utils.MatrixRotY(c4d.utils.DegToRad(axisOffset.y))
```

```
    mZ = c4d.utils.MatrixRotZ(c4d.utils.DegToRad(axisOffset.z))
```

```
    mXInvert = c4d.utils.MatrixRotX(c4d.utils.DegToRad(offsetXValueInvert))
```

```
    mYInvert = c4d.utils.MatrixRotY(c4d.utils.DegToRad(offsetYValueInvert))
```

```
    mZInvert = c4d.utils.MatrixRotZ(c4d.utils.DegToRad(offsetZValueInvert))
```

```
    # Combine all offset rotation matrixes into one.
```

```
    mInvert = mXInvert * mYInvert * mZInvert
```

```
    mInvert.off = c4d.Vector(0, 0, 0)
```

```
    # Combine all source joint rotation matrixes into one.
```

```
    mTotalTargetRelativeTransforms = mX * mY * mZ
```

```
    # We just want the rotation so we can reset the offset
```

```
    mTotalTargetRelativeTransforms.off = c4d.Vector(0, 0, 0)
```

```
    # Combine target joint original matrix with new relative rotation matrix.
```

```
    if isParentJoint == False:
```

```
# if a child joint remove the axis offset  
newTargetMatrix = (mTotalTargetRelativeTransforms * mTargetOrginalTransforms) * mInvert  
else:  
newTargetMatrix = mTotalTargetRelativeTransforms * mTargetOrginalTransforms  
  
# As we just want to keep the rotation, we "reset" the position with the  
# previous one.  
newTargetMatrix.off = mTargetOrginalTransforms.off  
  
# To be sure the scale remain to 1 we normalize the matrix  
newTargetMatrix.Normalize()  
  
# Convert back to a rotational vector  
newRotVec = c4d.utils.MatrixToHPB(newTargetMatrix, c4d.ROTATIONORDER_XYZGLOBAL)  
return newRotVec
```