Animator Friendly Rigging

Creating animation rigs that solve problems, are fun to use, and don't cause nervous breakdowns.

Jason Schleifer

CONTENTS

Finishing The Rig

In **Animator Friendly Rigging Part 4a**, we completed JJ's leg rig. Now we will continue with the process of making sure the whole rig works to the animator's delight!

Matching Controls

When working with multiple types of controls in Maya, it's important to be able to make things easy for the animators to switch back and forth between the various control structures. For example, with forward and inverse kinematics arms, it can literally save the animator days of work over the course of a project if they don't have to *manually* match the position and orientation of a control when they switch between FK and IK.

I've worked with rigs that have FK/IK matching--and rigs that don't—and I can tell you quite honestly that the rigs that don't allow for FK/IK matching don't only cause me more frustration, they remove acting choices that may make my performance better.

How do they remove acting choices? If I know that I have 4 days to animate a shot, and I think about making a character switch between FK and IK, if I *have* the ability to match the positions, then I don't worry about the technical aspects of the shot, I just animate it with the best performance in mind knowing that I can clean up any pops and make things match correctly. However, if there is no automatic matching, then I may decide to change my performance not because it's a better choice, but because I can't spare the hours it may take to match the FK and IK arms properly.

So, given the fact that we want to allow the animator to switch between forward and inverse kinematics, what's the best way to implement it? Well, the first thing we need to do is learn exactly *how* to match one object's position and orientation to another. Then, we'll look at the best way to implement it on our rig.

Since we're looking at doing all these things automatically for the animator, we'll be exploring the use of Mel commands to perform the matching tasks.

Matching the position of one object to another

When matching the position of one object to another, the methods we can use are vast. If the objects are in the same space – i.e. they're both a child of the same object – it's relatively simple to copy their positions.

For example, let's say we have a polygon sphere and a polygon cube, and both are in world space.

1. **Open example file *match_v1.ma***
   - Choose **File > Open**

- Navigate to the **example_files** directory and open ***match_v1.ma***

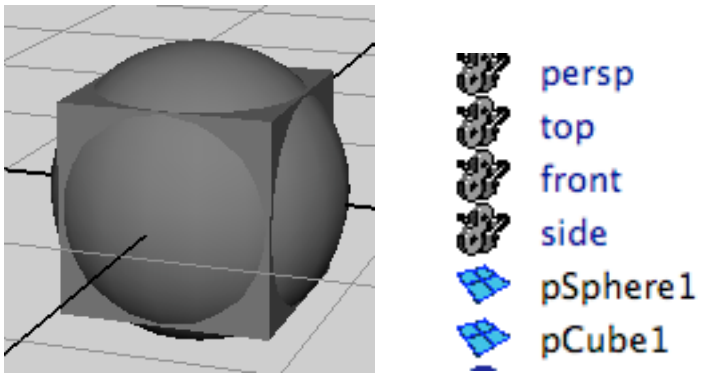

Figure 331 – pCube1 and pSphere1 sitting in the world in our scene

Now, let's say we move the sphere away from the cube somewhere.

2. Translate the sphere

- Select **pSphere1**
- Move it off to the side

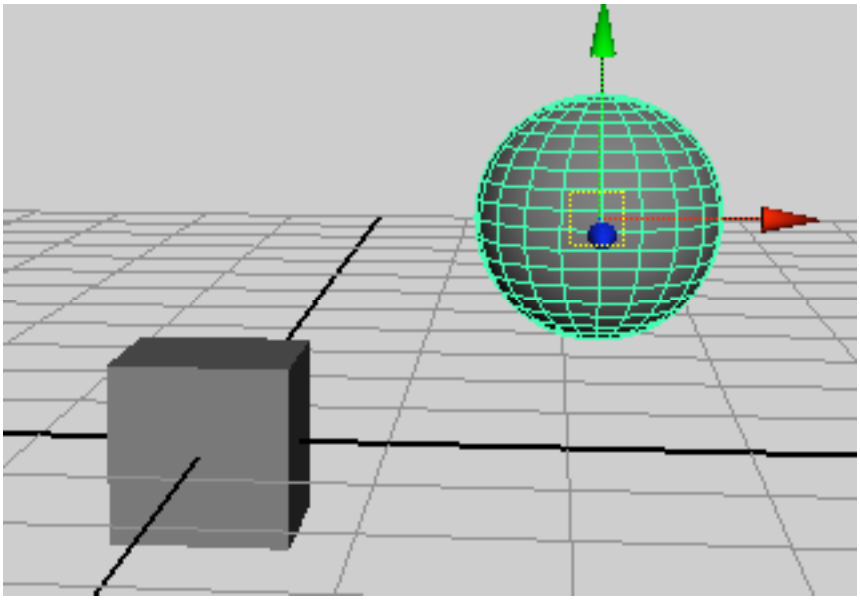

Figure 332 – moving the sphere

If we want to move the cube to match the position of the sphere, there are a few ways we can do it using Mel. The easiest of which is to use the **getAttr** and **setAttr** command, since we know we're grabbing the translation attributes:

```
getAttr object.attribute;

setAttr object.attribute;
```

getAttr will return the value of the attribute specified, and setAttr will set that attribute. Let's take a look at how we can use these commands in the script editor.
Open the Script editor and execute the following getAttr command:

```
getAttr pSphere1.translateX;
```

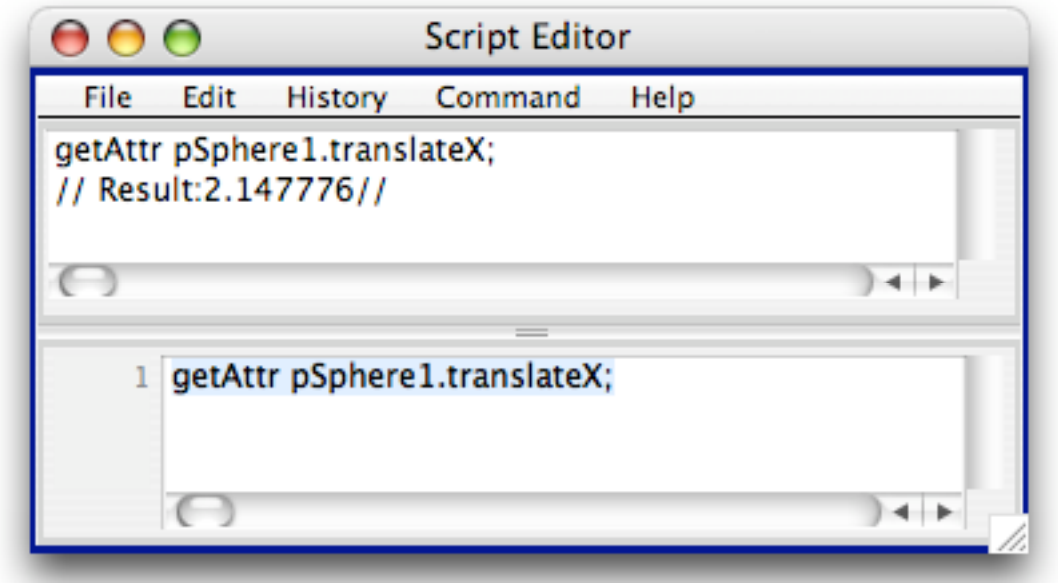

Figure 333 – Using getAttr to get the sphere's translateX value

You should see a result in the history of the script editor that looks something like the above image. Notice how you get a float value? That's great that it returns a value, but how do we actually *use* that value to do anything?

We can store the value as a *variable*!

Modify the line in the script editor to look like the following:

```
$tx = `getAttr pSphere1.translateX`;
```

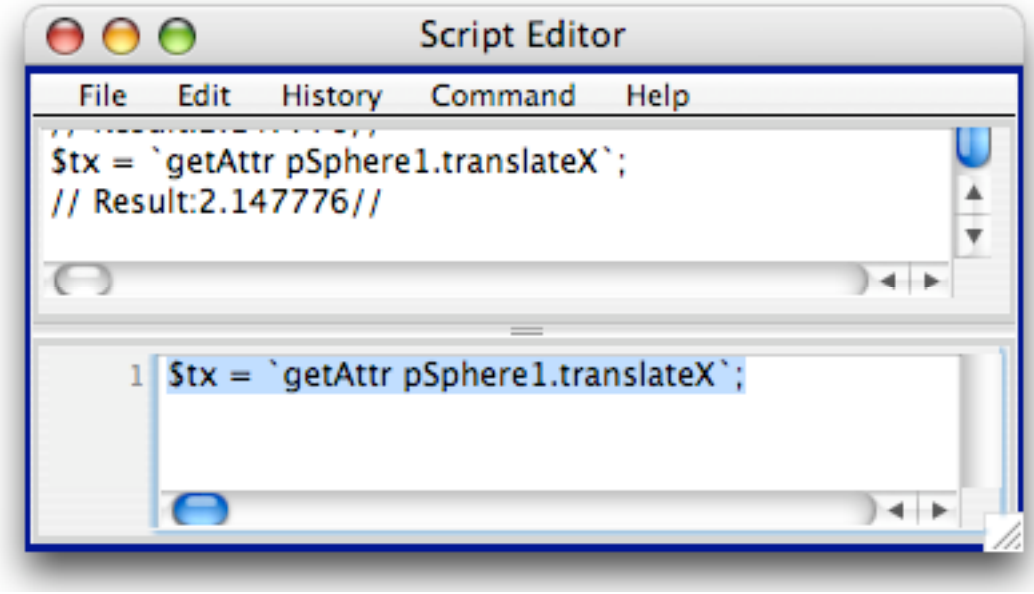

Figure 334 – Saving the value of the sphere's translateX into the variable $tx.

You'll notice that we added these single "tick" marks around the getAttr command itself. The tick mark is the one located on the keyboard on the ~ key. These tick marks tell Maya to take the command within the tick and pass the results back to the variable on the other side of the equals sign.

Basically, that means that whatever gets evaluated between the two ` marks will be passed into the variable **$tx**.

So we can use that $tx variable in a number of ways. For example, we can print it:

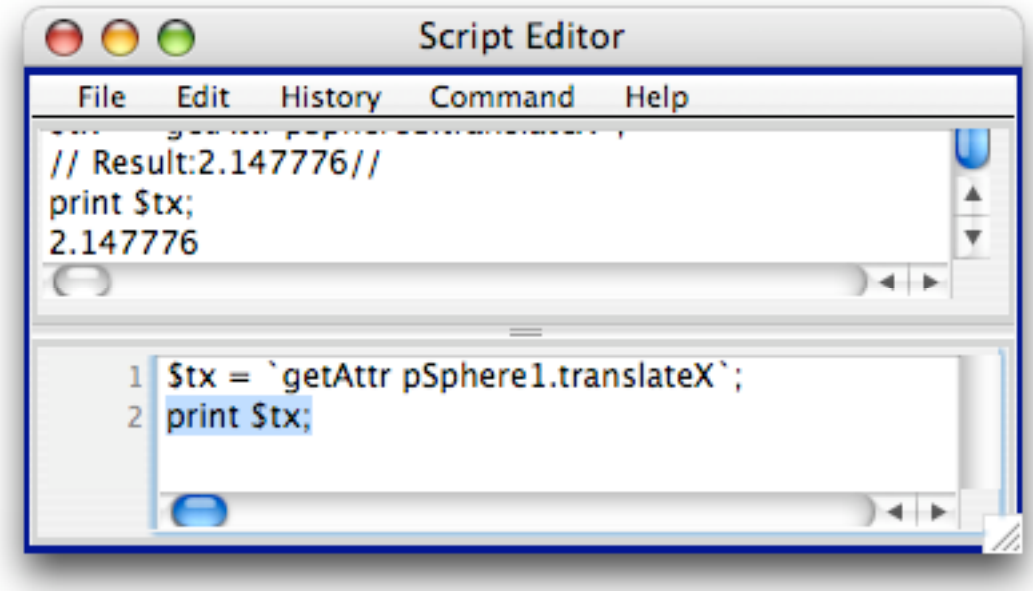

Figure 335 – Printing the results of the getAttr command by using the variable $tx

We can multiply it:

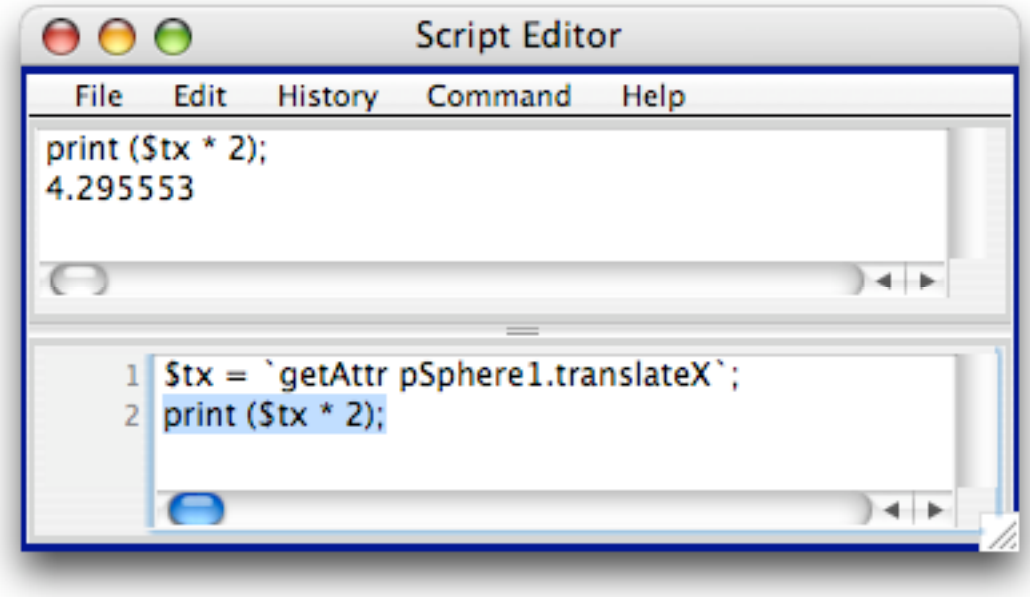

Figure 336 – Applying math to the $tx variable.

We can even use it with the **setAttr** command to set another object to *match* it:

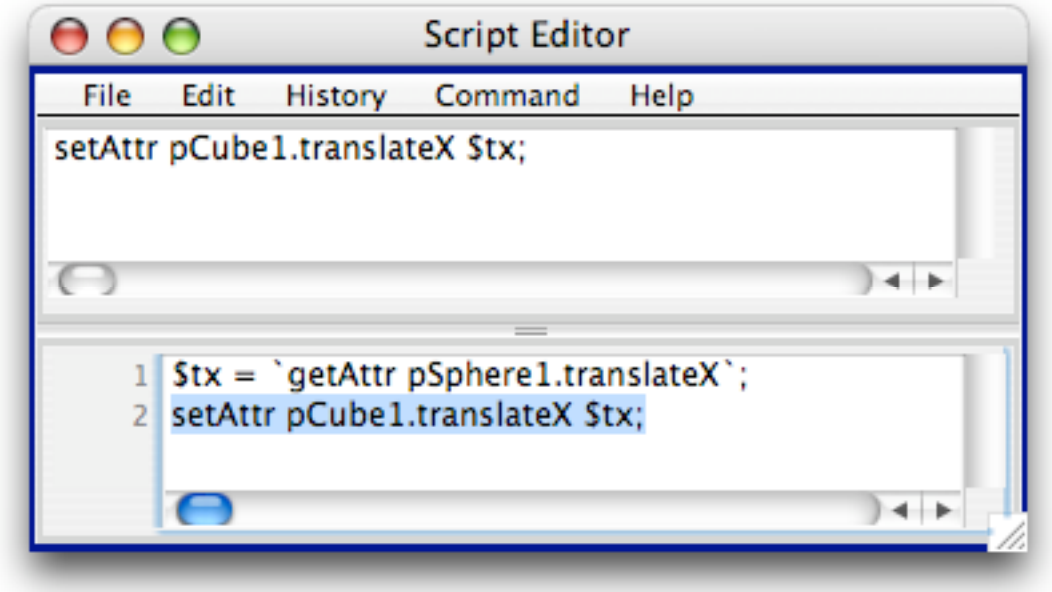

Figure 337 – Using the setAttr command in conjunction with the variable to set the value of the cube's translateX to the same value as the sphere's translateX.

Take a look at the sphere and what happens if we execute the above command:

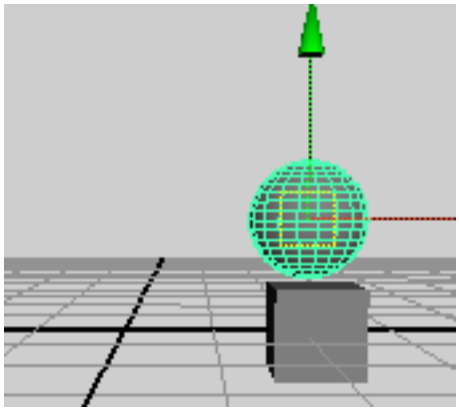

Figure 338 – The cube moves to line up with the sphere in translateX

Notice how the cube now matches the position of the sphere in the translateX axis? Let's move the sphere somewhere else, and execute the command again:

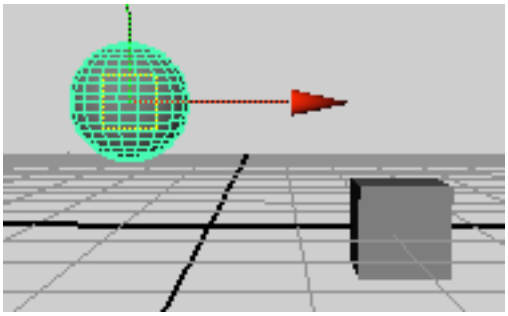

Figure 339 – Moving the sphere

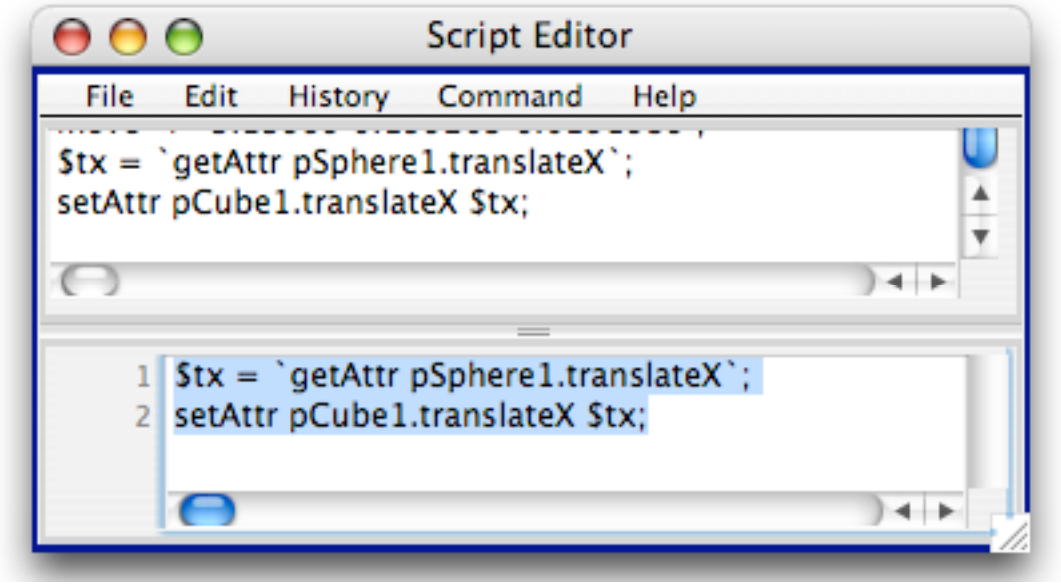

Figure 340 – executing the getAttr and setAttr commands

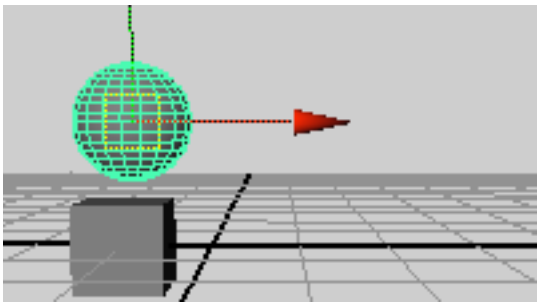

Figure 341 – The cube now lined up with the sphere.

See what's happening? We're able to use the getAttr and setAttr commands to match the position of the cube to that of the sphere, but only in the attribute we specify.

What if we want to match the position for all *three* translation axis? We could write three getAttr and setAttr commands:

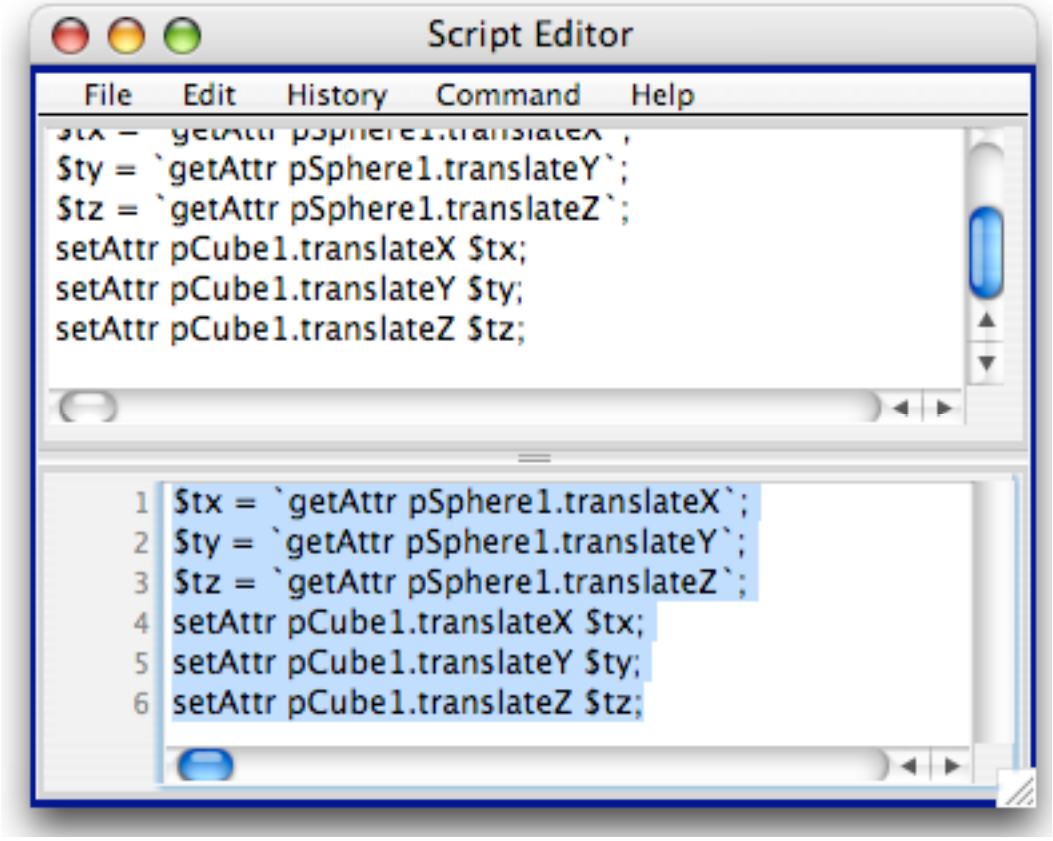

Figure 342 – three getAttr and setAttr commands to store the values of each attribute and then set them on another object.

This will certainly work, but it's quite long. Luckily, translation, rotation, and scale attributes are all "float array" array attributes, meaning they're automatically grouped in 3's.

For example, look what happens if instead of specifying the translateX attribute, we just request *translate:*

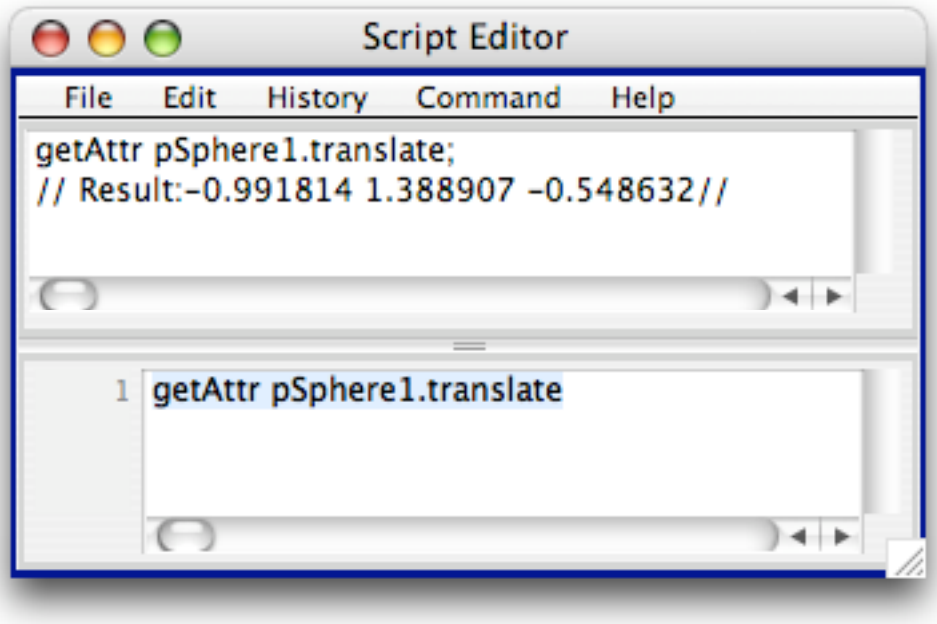

Figure 343 – requesting the value of translate

See the result? It contains *all three translation values*. So what happens if we try and grab that as an attribute and set it?

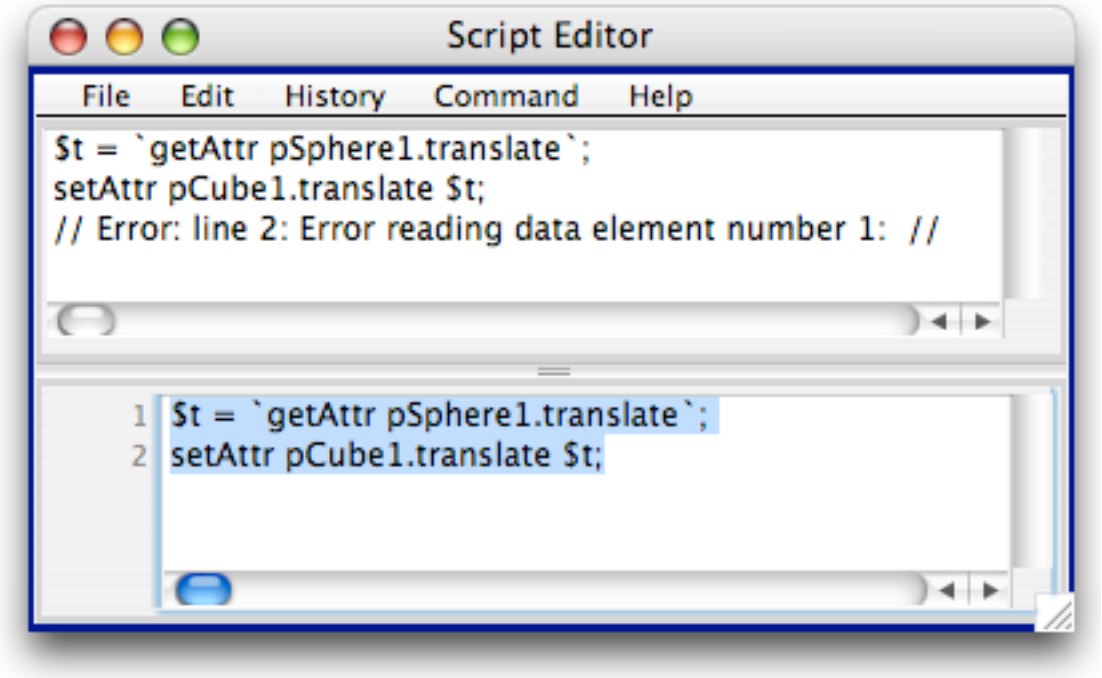

Figure 344 – you can't use an array with the setAttr command, it needs individual float values.

Oops, look at our result: ***//*Error: line 2: Error reading data element number 1: *//***

What does that mean? It means that while we can *grab* attributes of type float array simply by specifying the variable, when we set them we have to specifically tell Maya which item in the array we're telling it to use.

Which *what* in the *hoosah??!?*

Remember how I said that the translate attribute is considered an array? That means that it's actually a *set* of values. Three values, in this case. When you tell the getAttr command to return the set of values into a variable what you're actually doing is telling Maya that $t actually contains a set of 3 numbers.

With the above example, our values are**: -0.991814, 1.388907**, and **-0.548632**

So how do we access each individual value within the one **$t** variable? Simple! Individual items in an array variable can be accessed by telling Maya which one you want using an *index number*. For example, if we want the *first* item in the array (i.e. the first number returned by the getAttr command), we can use the notation of **$t[0]**. Remember, in computer-world, **0** is always first, **1** is second, **2** is third, etc. We start counting at ZERO.

So if we did this:

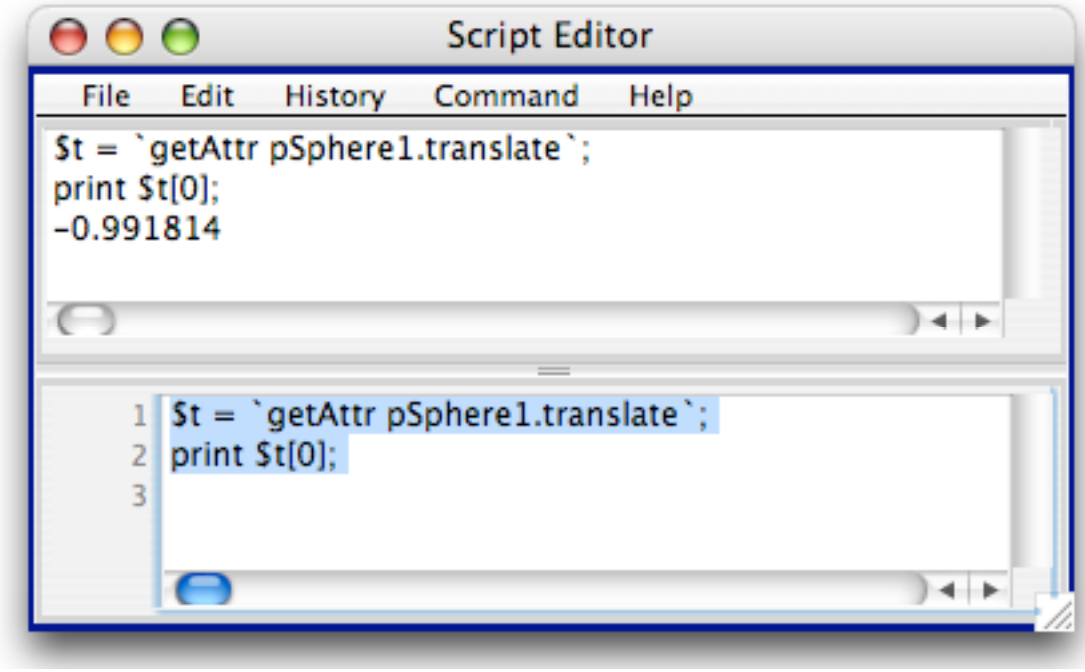

Figure 345 – Using an index number to print the first value of the $t array.

You can see that it will print the first item retrieved by the getAttr command.

Notice if we change the command to $t[1] instead of $t[0] we get:

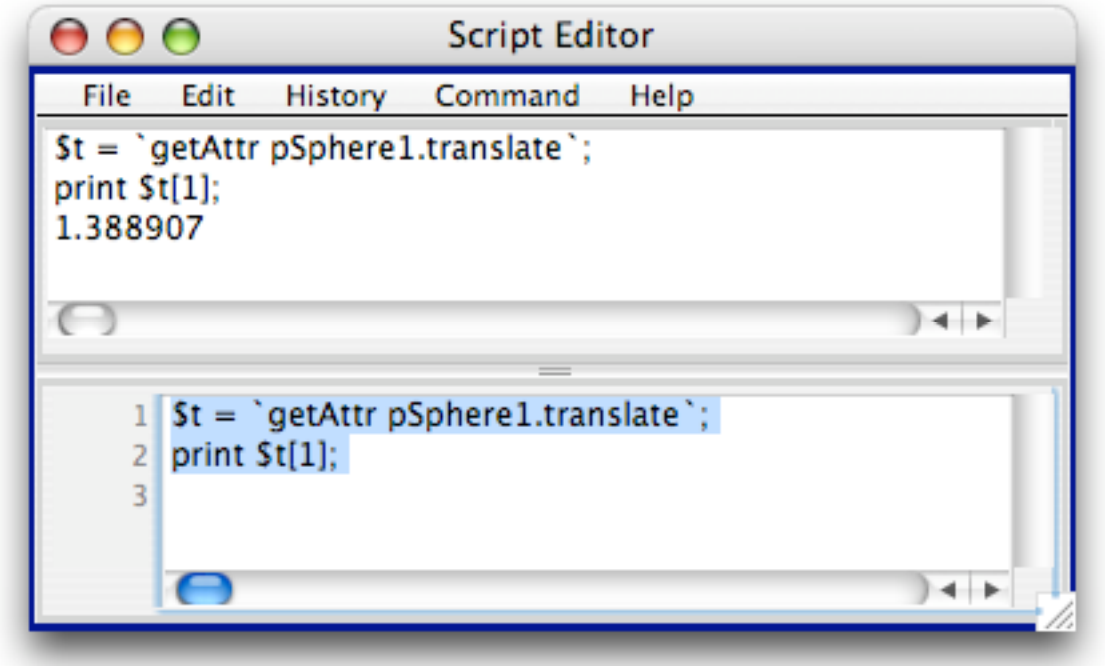

Figure 346 – Printing the second item in the array. Remember, computers count starting at 0, so the second item is 1, the third is 2, the fourth is 3, etc.

That's right, the *second item* in the array is printed.

So how do we assign all three items in the setAttr command? Like this:

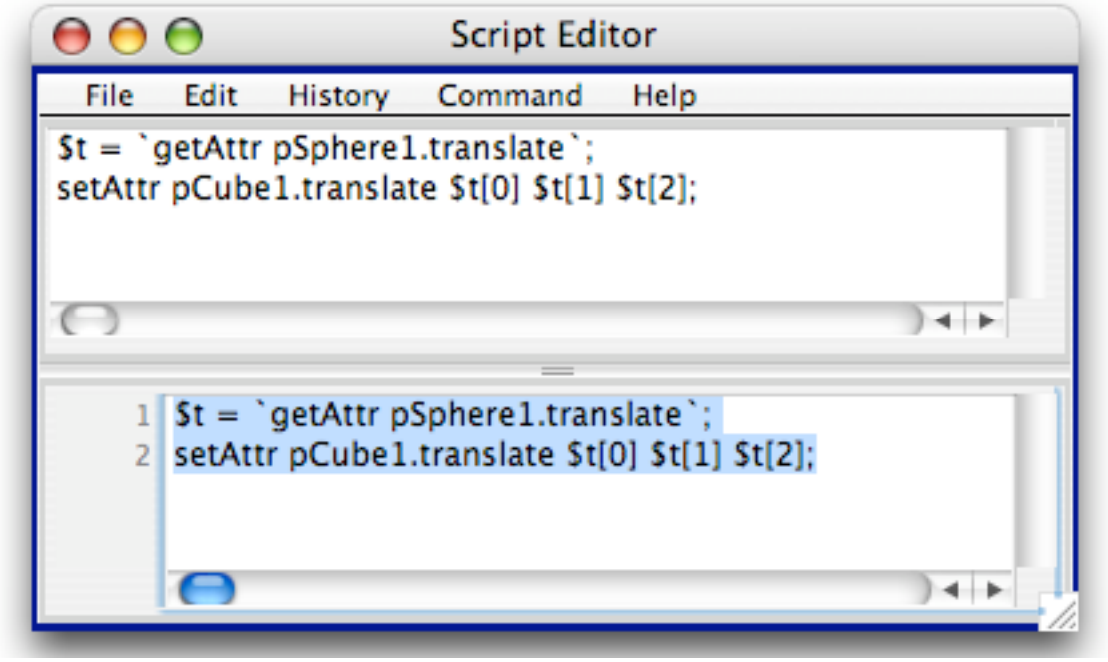

Figure 347 – Using the setAttr command and every index in the array.

See? No error!

We can do the same thing with rotation! Let's translate the sphere *and* rotate it.

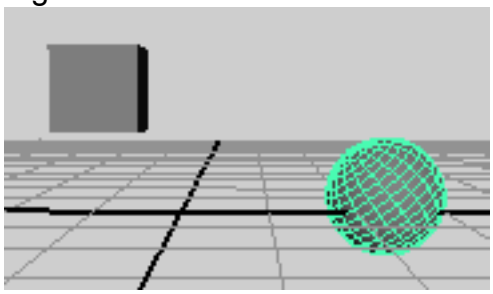

Figure 348 – Sphere is translated and rotated.

Now let's change our command so that we can get the position and rotation of the cube to match that of the sphere:

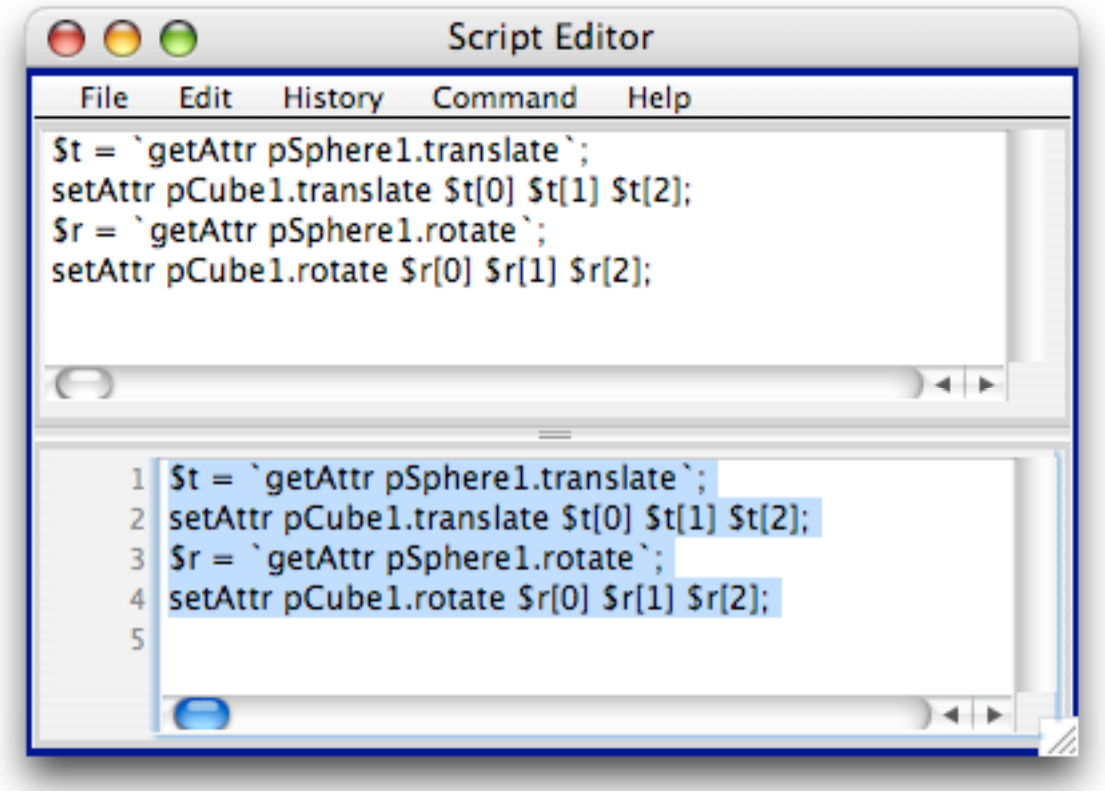

Figure 349 – Getting and setting translation and rotation.

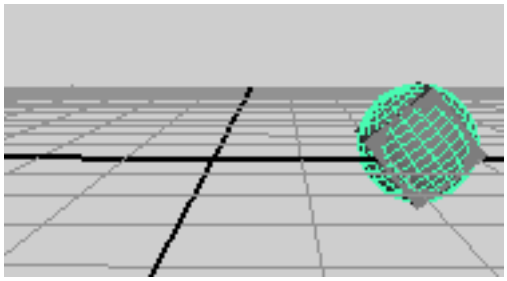

Figure 350 – the cube is now in the same location and orientation as the sphere.

Notice how the cube matches the position *and* orientation? Fantastic!

We now know how to match two objects when they're in the same hierarchy, but what about when the object is in a *different* hierarchy? Will this method still work?

Matching Different Hierarchies

Open ***match_v2.ma***

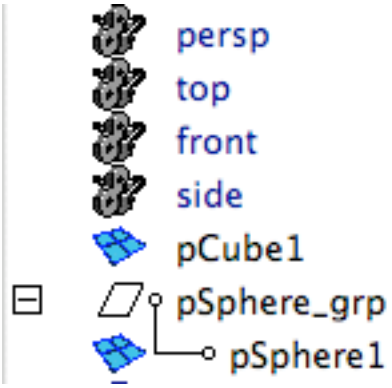

Figure 351 – Hierarchy in match_v2.ma

With this file, you can see that it's set up very similarly to match_v1, except that pSphere1 is under a different hierarchy than that of pCube1.

Let's go ahead and move pSphere1 off to the side by selecting pSphere1 and moving it

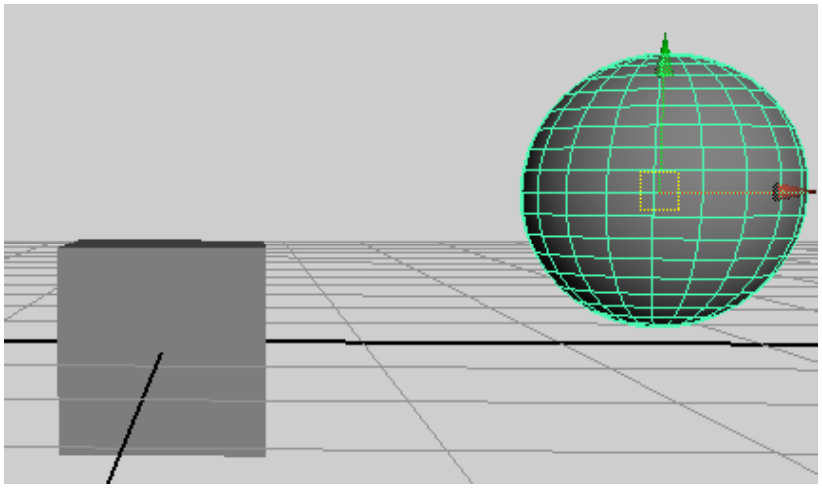

Figure 352 – Sphere is moved off to the side

Now let's use the getAttr command from before to match the position of the cube to the sphere:

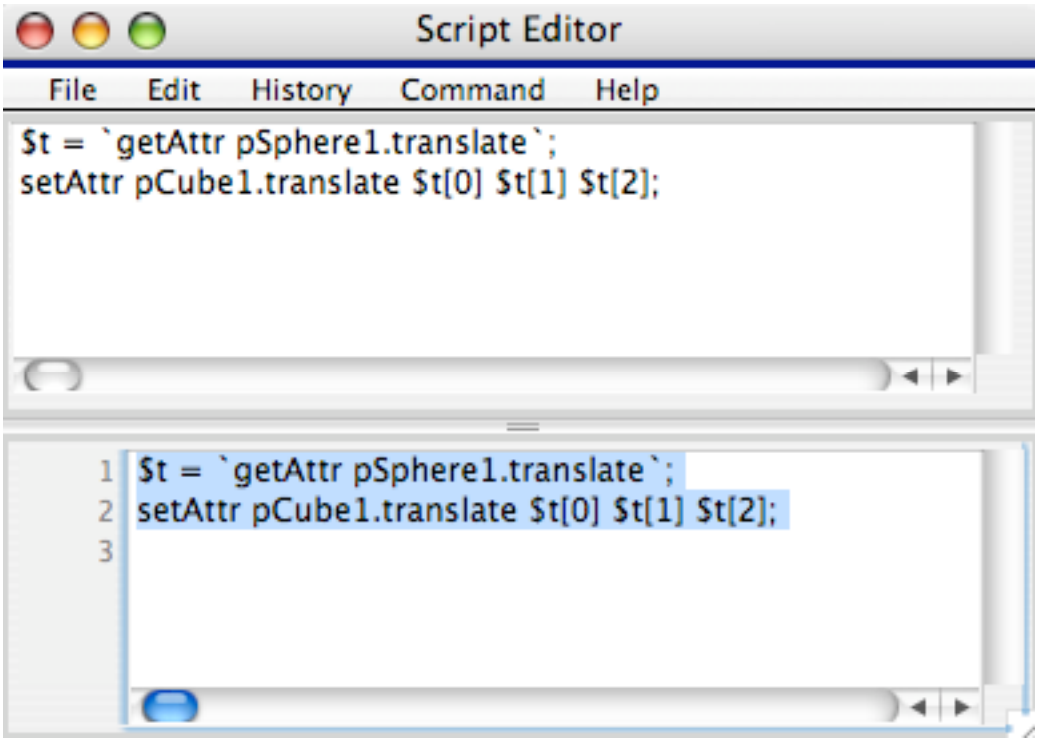

Figure 353 – using getAttr and setAttr to match the cube's translation to that of the sphere.

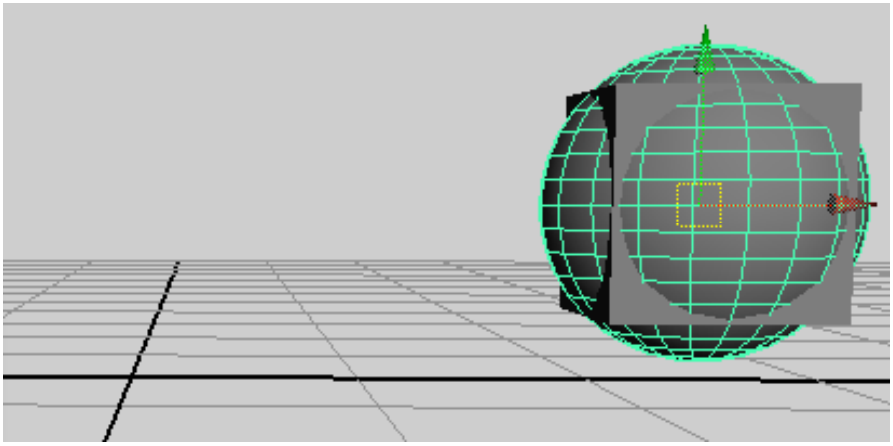

Figure 354 – The cube is now matched to the position of the sphere.

You can see that the cube did indeed move to match the position of the sphere. But, what if we move the *group* instead of pSphere itself?

Select **pSphere_grp** and move it as in the following image:

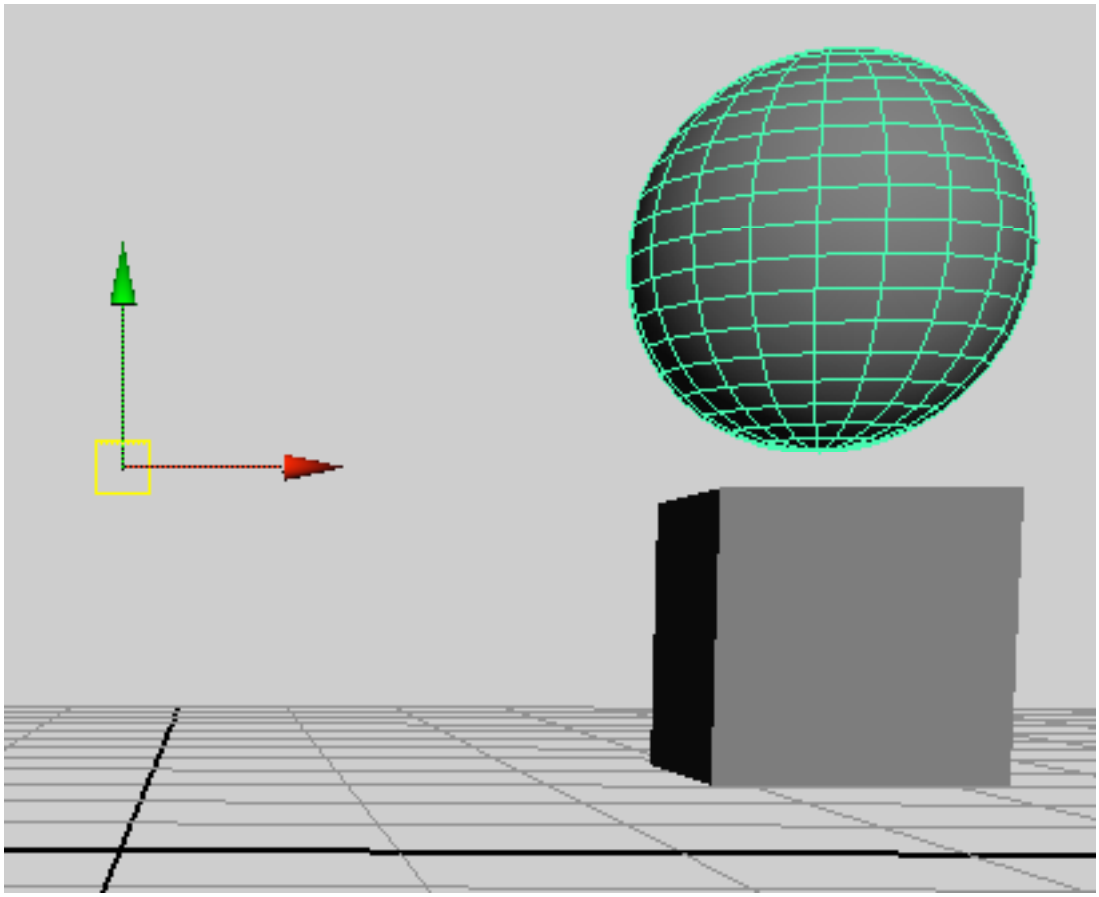

Figure 355 – Moving the node above *the sphere, pSphere_grp*

Now execute the command in the Script Editor:

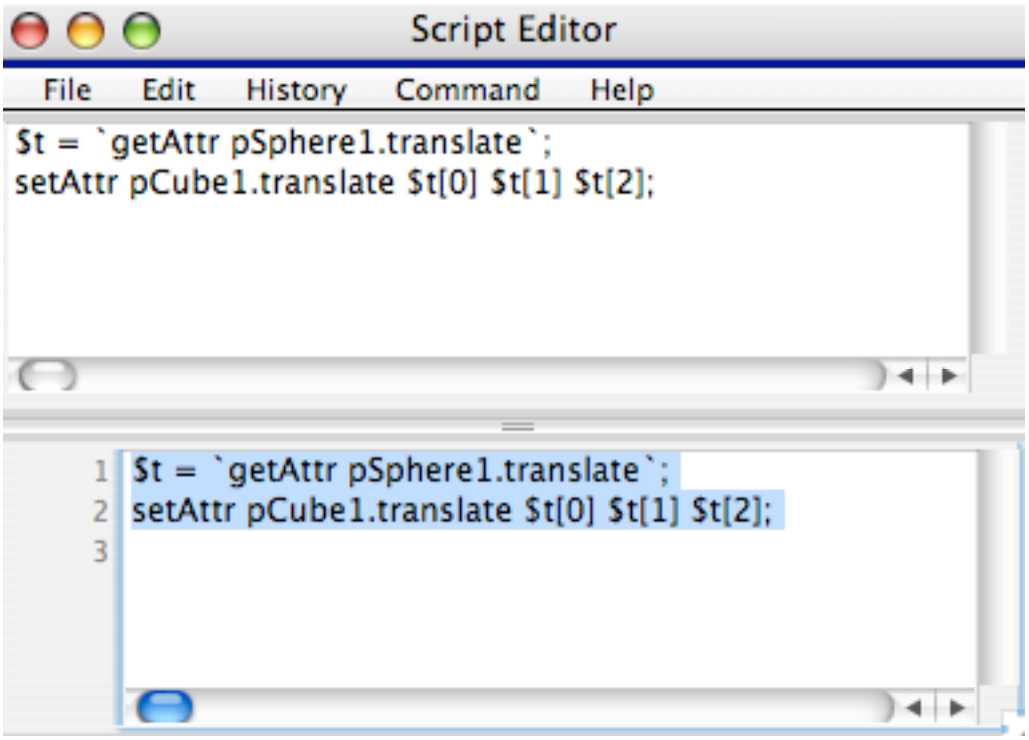

Figure 356 – Executing the getAttr and setAttr command in the script editor.

Notice that the cube did not move to match the position of the sphere. Why didn't it? It didn't move because we're using getAttr to grab the values of the **translate** attribute for the sphere. When we move the **group** above the sphere, the spheres own translation values don't change at all, *even though the sphere is in a different place in world space*.

So the command is doing exactly what we're asking it to, but it's not doing what we want. What we *want* is to grab the *world space position* of the sphere and set the cube to *that* value.

So how do we grab the world space position? Well, **getAttr** isn't going to do it for us, instead we're going to have to use the **xform** command.

Xform has a few more options—or "flags"—to use when using it. We use these flags to specify the type of information we're going to get back from the sphere, and in what "space" it is in. For our case, we want to use the following flags:

```
xform -query -worldSpace -translation object
```

The **–query** flag tells the **xform** command that we're in "query" mode, meaning to return the results instead of setting them.

The **–worldSpace** flag tells the command to return the values in world-space values, regardless of the parenting.

The **–translation** flag tells the command to return the values of the translation of the object.

So if we take the command and use it instead of the getAttr command, look at what happens:

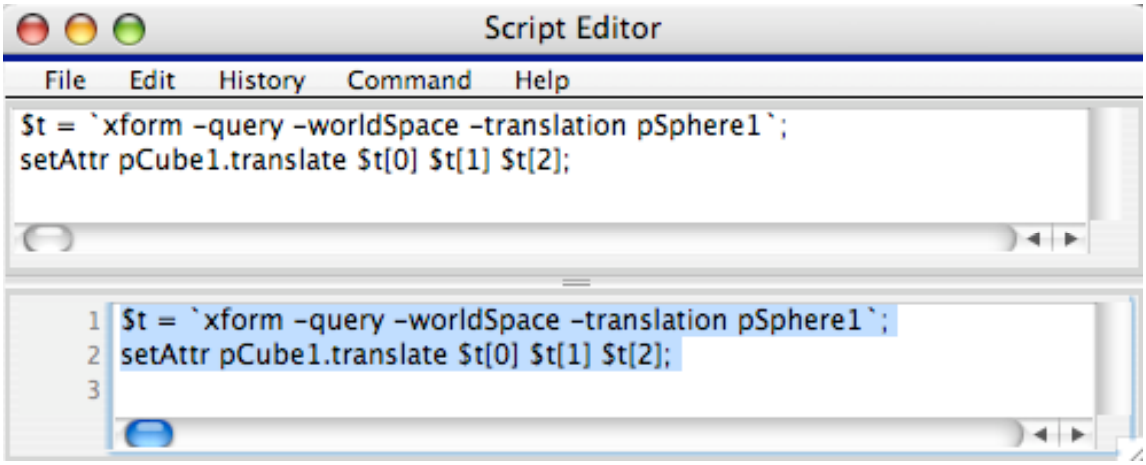

Figure 357 – Getting the world space translation of the sphere.

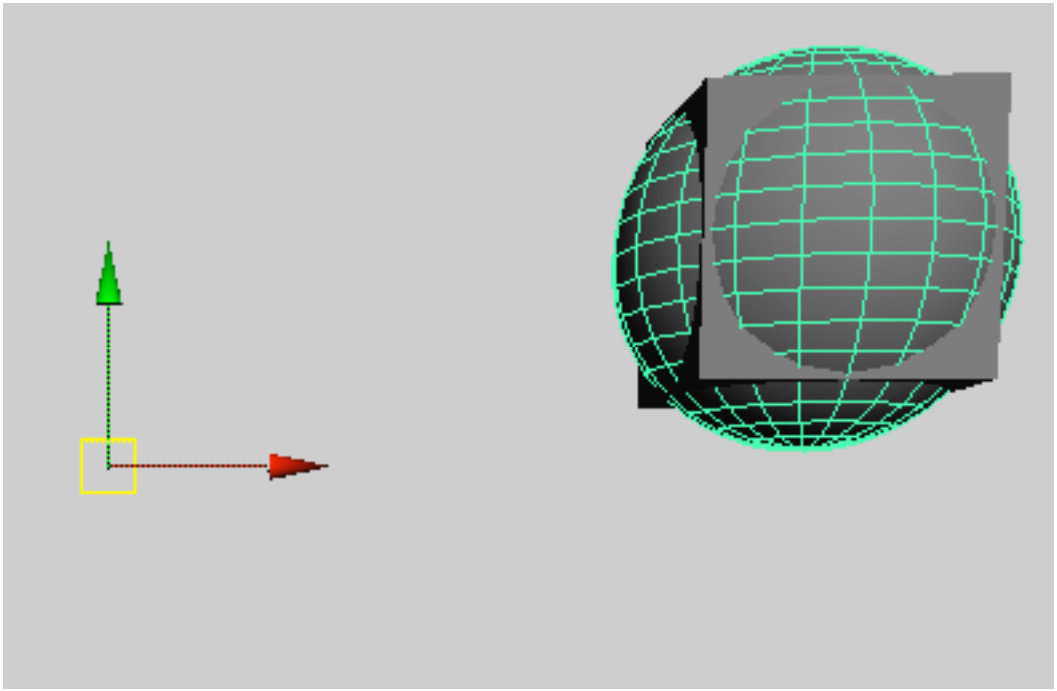

Figure 358 – The cube moves to match the sphere.

The cube moves to be in the same space as the sphere! Let's see what happens if we want to include rotation as well.

If you rotate **pSphere_grp** you'll see the sphere move away from the cube:

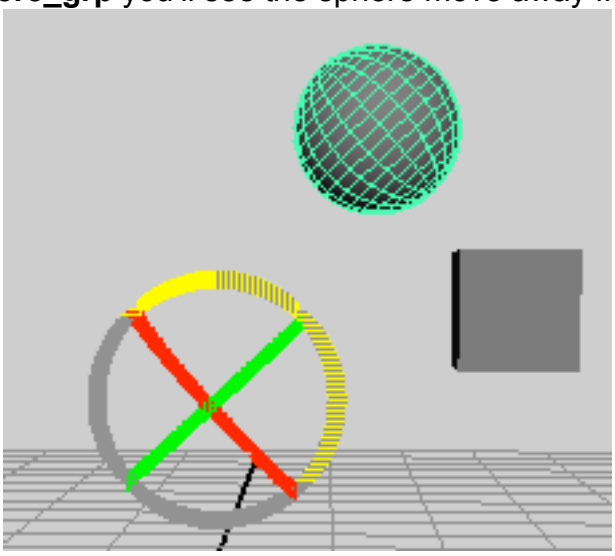

Figure 359 – Rotating pSphere_grp so the sphere doesn't line up with the cube.

Let's execute the same command as before and see what we get:

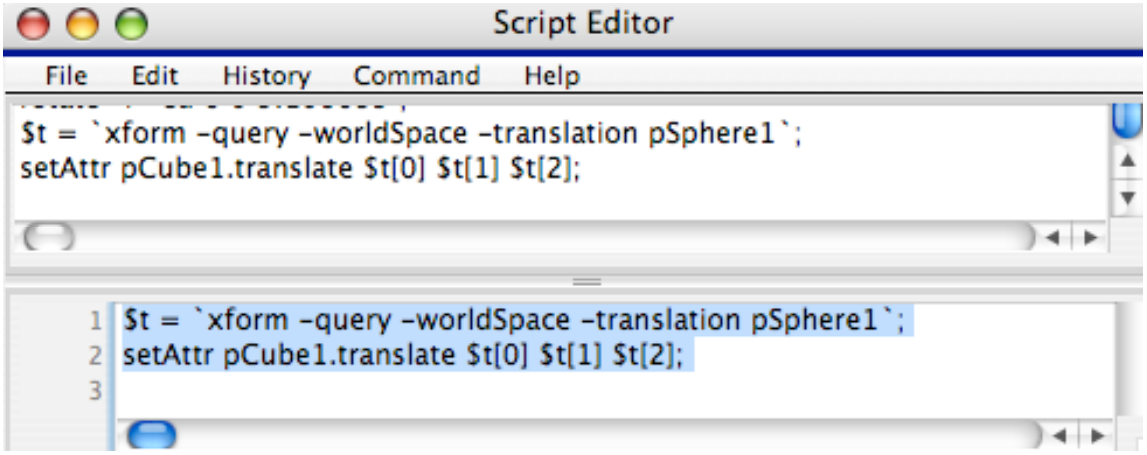

Figure 360 – Executing the same command, getting the world space translation and applying it to the cube with a setAttr command.

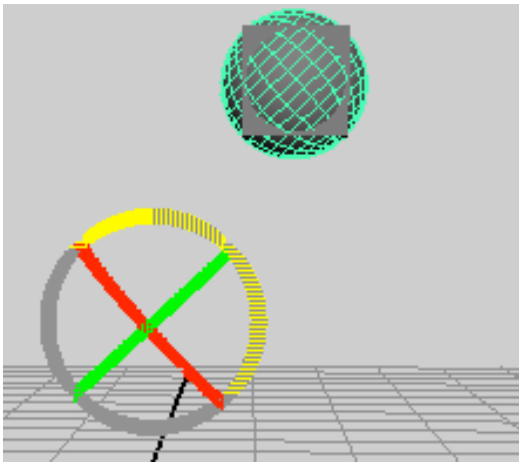

Figure 361 – The cube matches the translation of the sphere.

There cube moves to match the position of the sphere, but not the orientation. Can we use another flag for xform that will give us the rotation? Certainly!

Use the following xform command:

```
xform -query -worldSpace -rotation pSphere1
```

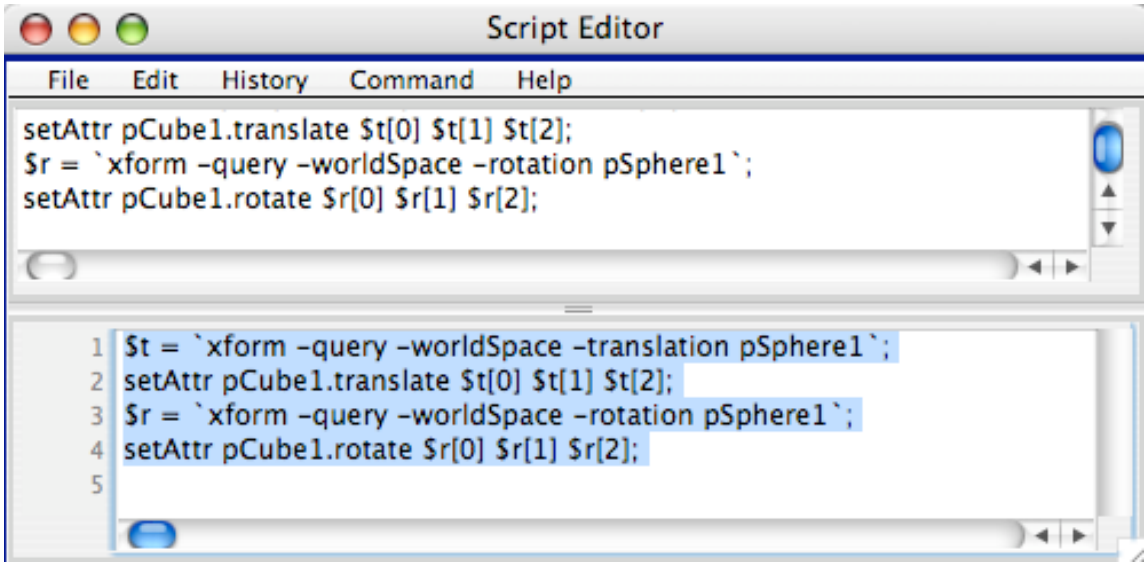

Figure 362 – Querying the translation and rotation in worldspace of the sphere and applying them to the cube.

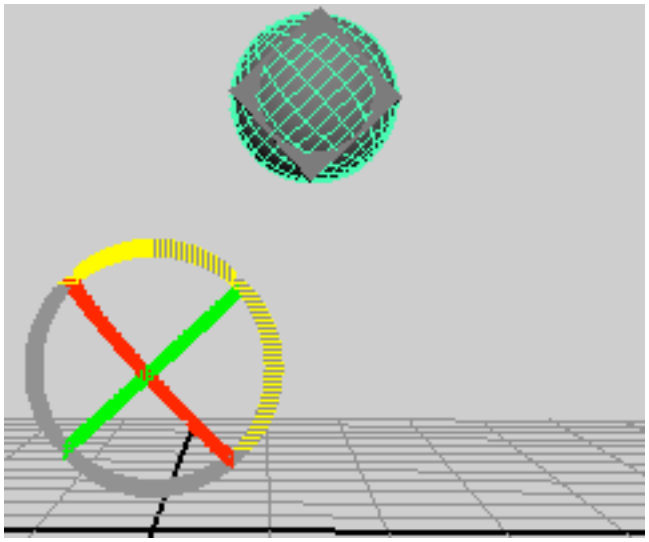

Figure 363 – The cube lines up with translation and rotation.

Both Objects in Different Hierarchies

As you can see, we were able to set the translation *and* rotation. However, you'll notice that we're still using the **setAttr** command to set the value for the cube. What happens if the cube is not in world space?