Course Title

Character setup from rig mechanics to skin deformations –
A practical approach.

Organizer

Yaron Canetti
Summer Breeze
yaron@summerbreezethefilm.com

Lecturers

Jason Schleifer
Animator, Weta Digital Ltd.
PO Box 15-208, Miramar, Wellington, New Zealand
jschleifer@paradise.net.nz

Raffaele Scaduto-Mendola
Character Setup Artist, DreamWorks SKG
raffaele@turbolinea.com

Yaron Canetti
Summer Breeze
yaron@summerbreezethefilm.com

Mark A. Piretti
Lead Rigger, Blue Sky Studios
40 Benedict Avenue, Apt. 2E, Tarry Town, NY 10591
markp@blueskystudios.com

Course Description

Module 1: rotation order, extra controls, selection masks, mirroring controls, feet and leg setup, IK fingers, limit notification, automatic positioning of animation controls.

Module 2: multi-layered rig setup, blending between behavioral and hand keyframe animation.

Module 3: Improving skin deformations, workflows, bones geometry (acquiring and reassigning skin weights), muscles (anatomy and layering deformers).

Module 4: Face rig (facial modeling, modeling in polygons, anatomy, subdivided surfaces, proper arrangement of geometry), jaw setup (multiple-joint jaw rigs with control expressions), facial deformers (using wire deformers, lattices, and blend targets to create expressions).

Prerequisites

Working knowledge of high-end 3D software and understanding of basic 3D animation concepts such as inverse and forward kinematics, key frames, geometry types, and deformations. Highly recommended: ability to script and write expressions. Working knowledge of Maya or Softimage XSI is an advantage.

Character Animation Rig Mechanics by Jason Schleifer, 1 hour 45 minutes

Introduction: Jason Schleifer (5 minutes)

- What is an "animation rig"?
- Course outline

Basic Rig Mechanics: (15 minutes)

Introducing some of the basic concepts of an "animation rig".

1. Rotation Order

    - What is it?
    - Examples of why rotation order is important.
    - How to change the rotation order of an object.

2. Extra Control Structures

    - Why do you need an extra control structure.
    - Adding extra controls dynamically.

3. Limiting object selection & keyability

    - Why limiting selection and keyability is important.
    - Examples.

4. Using Display Layers and sets to ease in animation/rig management.

    - Demonstrate scenes without proper layer designation.
    - Describe what proper layer management provides.

Body Segment Rigs: (1 hour)

Using specific examples, look at a few methods of controlling different aspects of an animatable character.

1. Mirrorable Controls

    - How to make sure controls make sense on *both* sides of the character.

2. Feet/Legs

    - Setup allows for rotation from the ankle when foot is in the air, and allows the animator to add weight and rest the character on the ball of it's foot.
    - Demonstrates techniques for grouping IK handles and rotation them around specified axis.
    - Demonstrates adding control attributes for ease of animating.

3. IK Fingers/Hands

    - Continues previous example with more complicated grouping and control structures.

Advanced Concepts: (25 minutes)

1. Colour Based Limit Notification

Q& A: (10 minutes)

Character Animation Rig Mechanics – Jason Schleifer

What is a Character Rig?

A "Character Rig" can be defined as a control structure designed to ease the animation of creatures and objects.

A Character Rigger's job is to make those controls as intuitive as possible. Their goal should be to be able to hand a rigged creature off to a team of animators and *never* receive a call saying one of the following:

- "I can't find the arm control.."
- "What does this control do?"
- "It doesn't work the way I want it to.."
- "I hate this, you've made my life a living hell!"

In order to accomplish this task the "rigger" must have a solid understanding of the types of situations their character is going to get into. They need to be able to juggle the need for absolute control, consistency between animators, and animation speed. A character rig that can do anything but is too slow to update at a sufficient rate will only cause frustration and irritation. In addition, a rig that can update at the speed of light, but doesn't have the controls that are necessary to complete a shot can cause an animator to struggle beyond necessity (a *little* struggling isn't always a bad thing).

Some of the general rules a character rig should follow are:

- All controls should be easy to understand both visually and by name.
- Controls should work the way the animator expects.
- Minimize counter-animation.
- Guard against gimbal lock.
- Update as fast as possible.
- Do not allow for animators to select or move anything you don't want them to.

Course Outline

This course is meant as a general introduction to some of the concepts in character rigging. It will cover some important basics, including:

- Rotation Order
- Extra Nodes to help with Gimbal Lock
- Limiting Selection and Keyability
- Using Display Layers

Then we will get into some more interesting setup examples. These range from beginner to advanced concepts.

- Mirroring Animation Controls
- Foot/Leg Setup
- IK Finger Controls

Finally, we will explore a more advanced concept of colour limit notification.

1. Basic Rig Mechanics – Rotation Order

What is "Rotation Order"?

Rotation order is, simply put, the order that the rotations on an object are evaluated. For example, if the rotation order on an object is XYZ (in Maya), then first the Z-axis is evaluated, then the Y-axis, then finally the X-axis. By evaluating these three values, a final orientation is established.

Why is Rotation Order Important?

It is important to understand how rotation order works in order to keep from hitting what is commonly called "Gimbal lock". Gimbal lock occurs when there is no more rotation axis available to get a desired orientation. Let's take a look at an example where the rotation order of an object adversely affects the animatability of that object.

The cube shown in figure 1 has three objects above it representing the available rotation axis: X, Y, and Z. These rotations have been separated out into nodes to help demonstrate how they get solved.

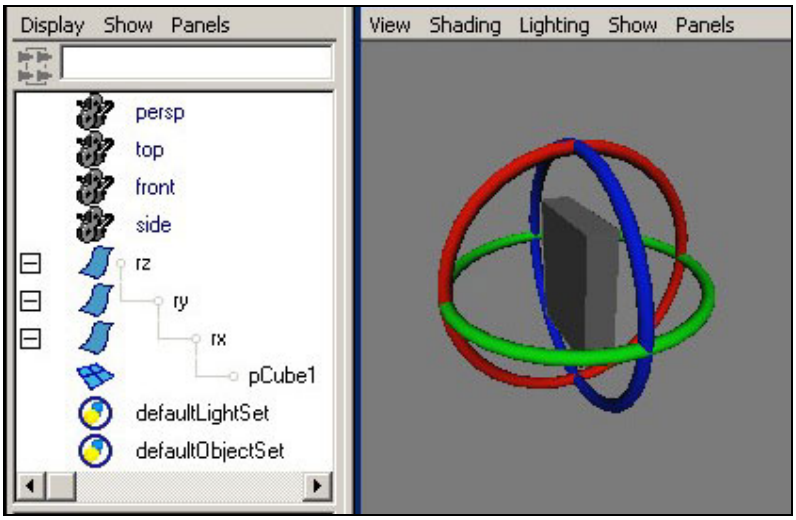


Figure 1 – Rotation Order

Looking at this image, we can tell that the **rz node** will get solved first. Anything we do to that node will happen to all the nodes beneath it. Next, if we rotate **ry**, both **rx** and the cube will rotate, but **rz** will not. That's because **rz has already been solved**. Finally, **rx** can be rotated. See Figures 2-4 for an example.

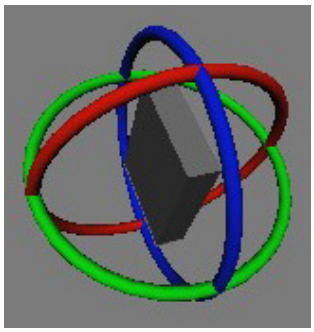


Figure 2 – **rz = -30**

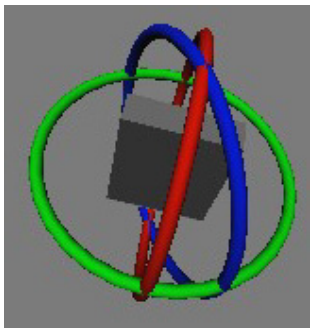


Figure 3 – **ry = 60**

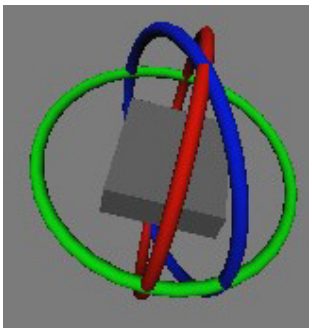


Figure 4 – **rx = 80**

Notice in figures 3 and 4 there is already a situation where we're hitting gimbal lock. What if we wanted to rotate the cube up on it's side (figure 5)? There are no more rotations available. This is why picking the correct rotation order is extremely important. There is no way to avoid gimbal lock completely, but you can minimize it's effects.

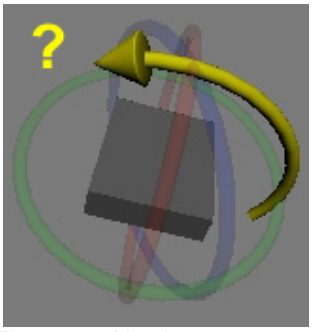


Figure 5 – Nowhere to rotate!

Changing the Rotation Order

In Maya, an object's rotation order can be changed easily in the **Attribute Editor**.

Simply select an object, bring up the **Attribute Editor**, and change the **Rotation Order** attribute to the value you want (figure 6).

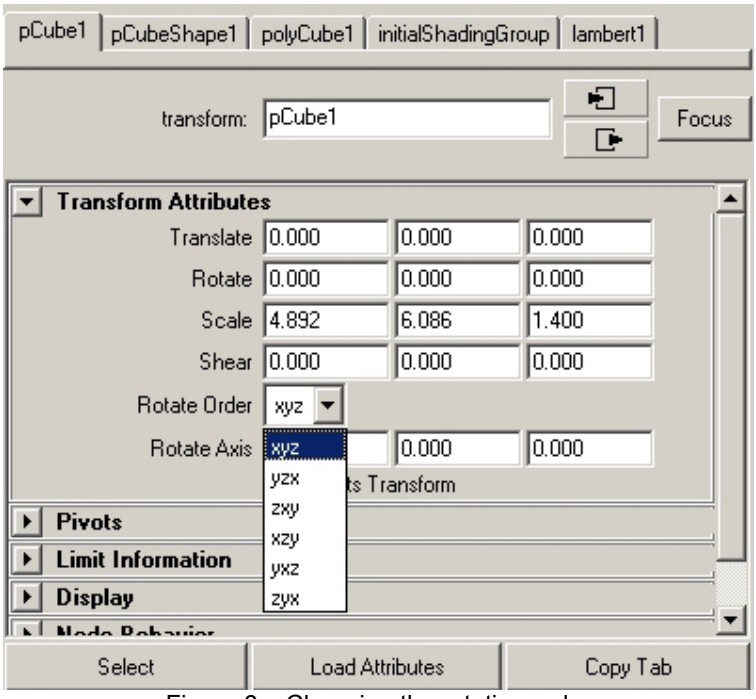


Figure 6 – Changing the rotation order

Note: The rotation order is solved in the *reverse* order it appears in Maya. So, for example, if the Rotate Order is set to **xyz**, the **z**-axis is actually the first one to solve.

Which Rotation Order should be chosen?

In most cases, I've found that it's best to have the **Y**-axis solved first, mainly because Maya's world has a **Y-up** orientation. This way, your characters can turn around in any direction and still lean over and side to side without hitting gimbal lock. Therefore, either **ZXY** or **XZY** should be used on most main controls (i.e. body, hand, foot, head, etc).

On other controls you should determine the rotation order on a case-by-case basis. Try the different rotation orders for yourself, testing the different situations your control might encounter, and decide upon the order that provides the most options.

2. Basic Rig Mechanics – Extra Control Structure

What is a Control Structure?

Before we define an "*Extra* Control Structure", it's important to specify what a "Control Structure" is. A Control Structure is simply a method for animating a series of nodes within your software package. If, for example, you have a character that has an Inverse Kinematic arm, instead of having the animator grab the ikHandle and move it around, you provide a **locator** or a **curve** that the ikHandle is parented to. This provides you (the person setting up the rig) more options as to how to present the controls to the animator.

A handy technique is to already have a number of controls generated that you use quite frequently. See Figure 1 for some examples.

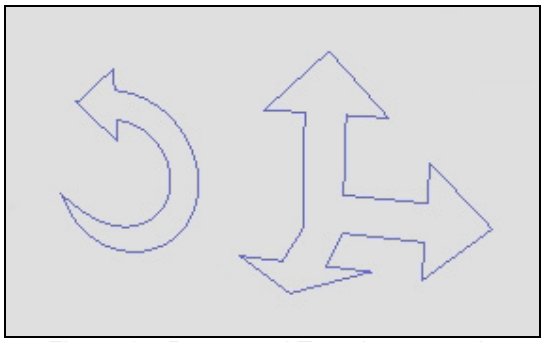


Figure 1 – Rotate and Translate controls

What is an Extra Control Structure?

An Extra Control Structure is simply another control that will help the animator by either allowing them to offset their animation, or help them solve a gimbal-locking problem. A good example is using this extra control to add high-frequency noise to a character's motion.

How does one get added?

This technique assumes that you have a curve, a locator, or a surface that is already controlling the node. To add an extra control, use the following steps:

1. **Duplicate** the control structure you wish to add more control to. (Figures 2 and 3)
2. **Delete** the children of this duplicated control. (Figure 4)
3. **Parent** the new duplicated control under the original control. (Figure 5)
4. **Parent** the children from the old control under the new duplicated control. (Figure 6)
5. **Provide** a method for the animator to select this new control. (Figure 7)
    a. **Selection Handles**
    b. **Offset geometry**
    c. **Button**

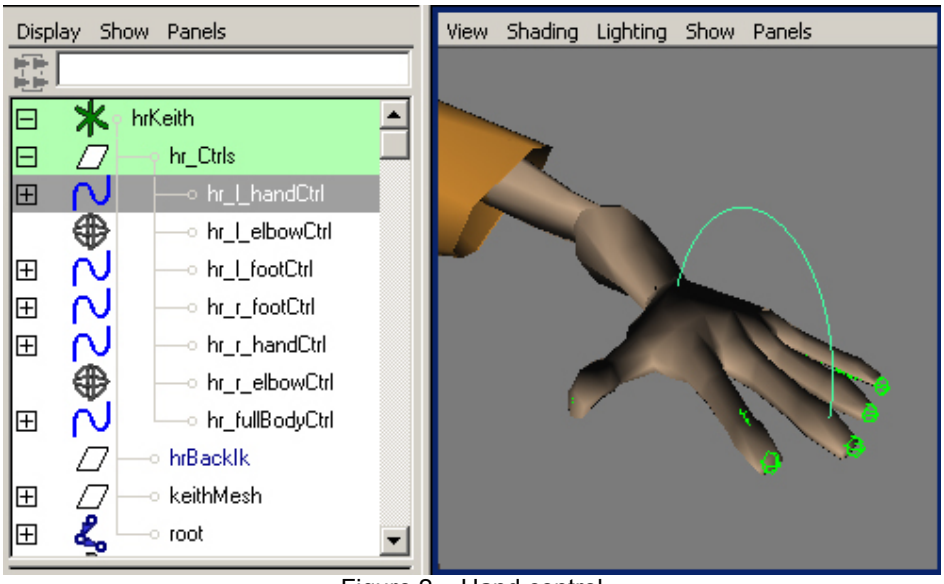


Figure 2 – Hand control

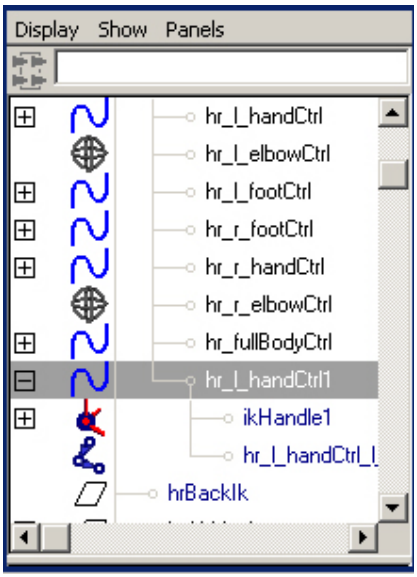


Figure 3 – Duplicated Hand Control

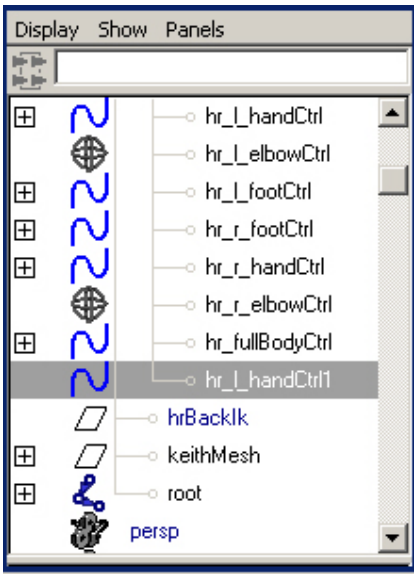


Figure 4 – Deleted Children

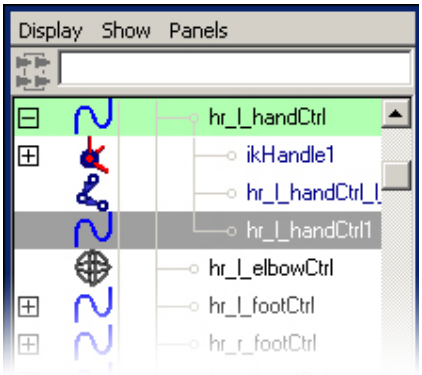


Figure 5 – Re-parented Extra Control

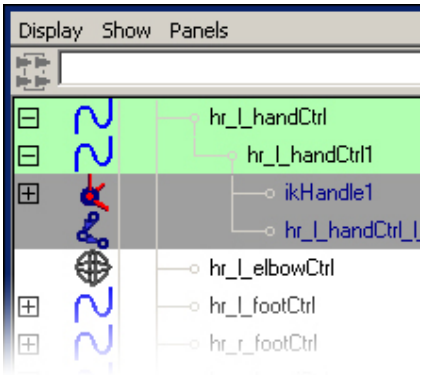


Figure 6 – Children Re-Parented

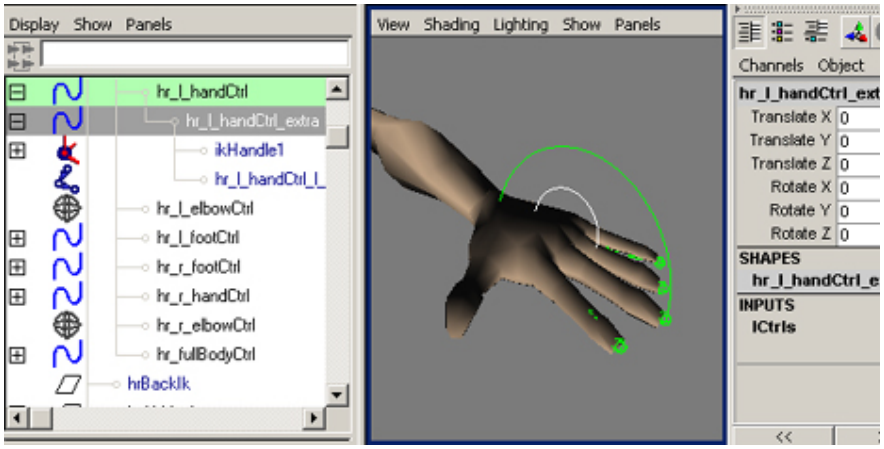


Figure 7 – Scaled Extra Control CV's and Renamed Control

3. Basic Rig Mechanics – Limiting Object Selection and Keyability

What is Limiting Object Selection and Keyability?

Limiting the "selectability" of objects in the creature rig is simply making sure that there is nothing selectable that shouldn't be. The only things the animator should be able to select are objects that you *want* them to select.

Limiting an objects "keyability" is similar, except it exists on a per-attribute basis. You're allowing only certain attributes to be keyable on an animatable node. That way animators aren't saving keys on an object's scale attribute without you intending them to.

Examples

Figure 1 – Translation not locked!

Figure 2 – Scale not locked!

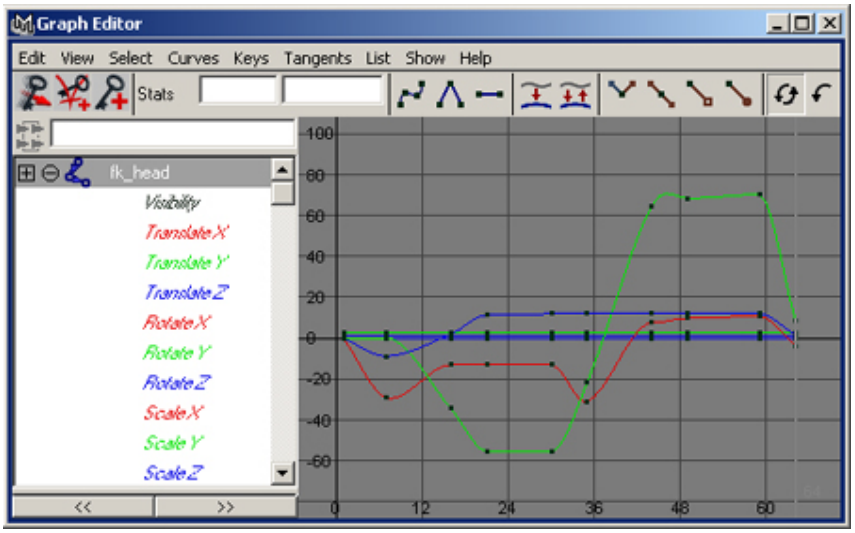


Figure 3 – Visibility, Translate, and Scale being keyed when not necessary

Why is Limiting Object Selection and Keyability important?

- Placing limits on what can and can't be animated allows the animator to work without having to worry that they may be selecting something incorrectly.
- Placing limits allows you as the person setting up the creature rig the confidence that you are eliminating opportunities for something to break.
- Removing attributes that shouldn't be keyed keeps animator's scenes easier to work with as no extra animation curves get generated.

4. Basic Rig Mechanics – Using Display Layers

Display Layers can be used for:

- Limiting what animators can and cannot select.
- Easily differentiating between different types of controls.
- Hiding and showing various controls and elements in a character.

Limiting What Animators Can and Cannot Select

As mentioned before, the ability to allow animators to *only* select what you intend is extremely important. Display layers within Maya can do this easily by specifying what you *don't* want them to select as a **reference** layer.

When you have *no* display layers for your character, everything looks the same and the animator can select any part of the body. In Figure 1, you can see that it would be extremely difficult for an animator to understand what they were supposed to animate.

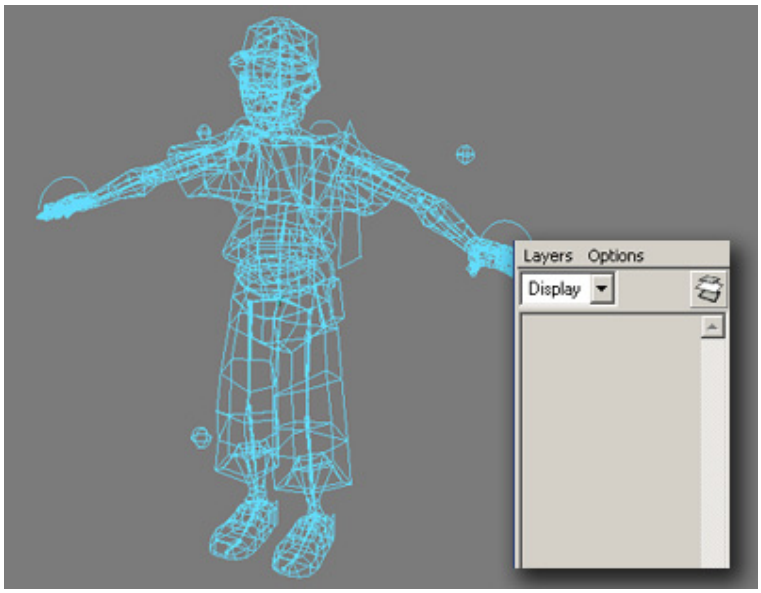


Figure 1 – No display layers.

In Figure 2, we have added display layers for the character, and set some of them to **Reference** mode. This ensures that the animator *cannot select* the items that are in the referenced layer. As you can see, it is now easier for the animator to determine which nodes they need to animate.

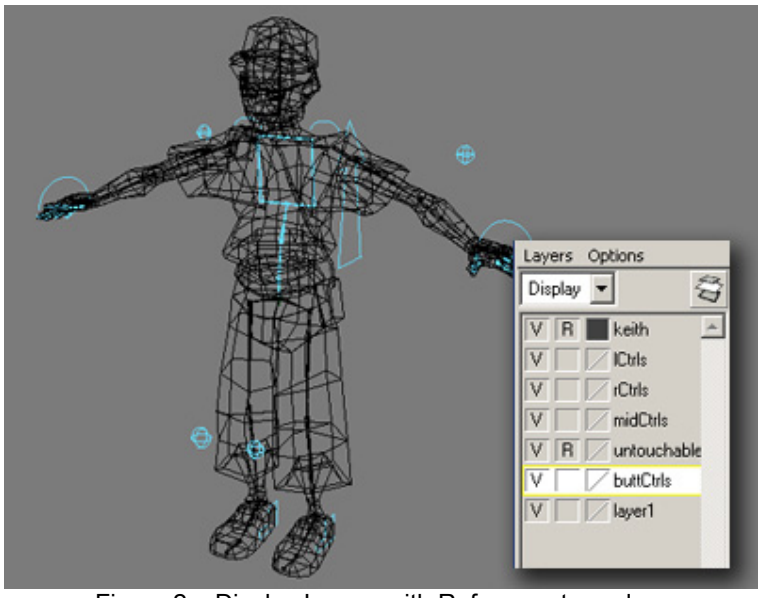


Figure 2 – Display Layers with Reference turned on

Figure 3 takes this idea even further by using colours to determine what type of control something is, and which side of the character it should be on. If you are consistent with your colouring, then the animator always knows that left controls are green, right controls are red, and middle controls are yellow.

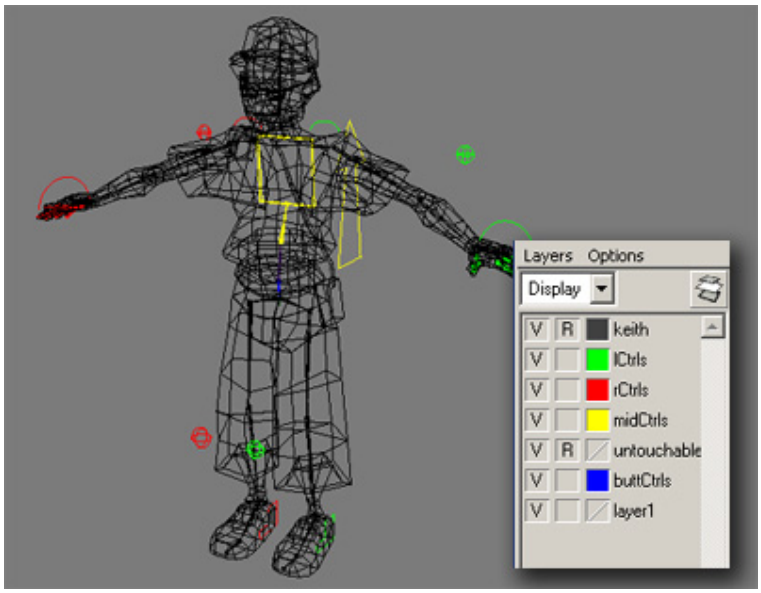


Figure 3 – Display Layers with colours per layer

Using Display Layers to Hide Parts of the Body

Often, animators will find it easier to block the animation of their character if they can hide various parts of the body. For example, in Figure 4 the arms of the character are hidden to make it easier to get the timing and motion right for the character's body spinning. Once the timing is correct, then the animator can show the arms and animate them.

Figure 4 – Animated Character With Hidden Arms

5. Body Segment Rigs – Mirrorable Controls

One of the most important aspects of creating a creature rig is ensuring that the animator gets the behaviour they expect from a control. Part of achieving this is making sure to follow these guidelines:

- Animators should know what a control does simply by looking at it (this is why you should use nurbs curves as controls).
- Animation controls should behave in an intuitive way. If the animator wants to move the object UP in Y, they should be able to simply animate the Y attribute of the object.
- **Whenever possible, make rotation controls mirrorable on both sides of a character.** This means that if an animator rotates a control in Y on the left side, the same value in Y on the right control should produce the same result.

Examples

With the examples in Figure 1, both wing controls have been rotated 30 degrees in X. In the image on the left, you can see that the controls were not mirrored correctly, giving an un-intuitive result. The image on the right shows correctly mirrored controls with results that are consistent for the animator.

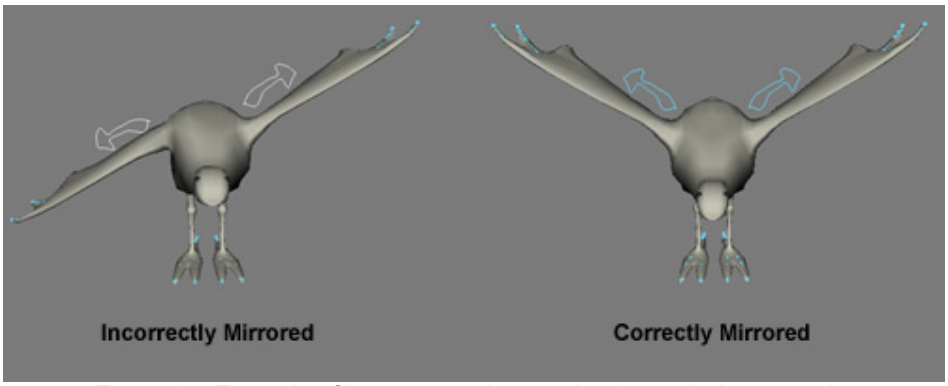


Figure 1 – Example of incorrect and correctly mirrored wing controls.

How to Create Mirrored Controls

The technique for creating mirrored controls is not as easy as one might think. Thinking about it logically, you would think that simply scaling the control across the Y-axis would give you the result you want. Unfortunately, it turns out that this is not the case. Scaling the control will provide *some* of the axis to mirror correctly, but not all of them. It is important for *all* axes to be mirrored to keep the animation intuitive. The animator should never have to remember that the *left* wing lifts up with negative numbers, and the *right* side lifts with positive numbers. Both sides should lift with positive, and drop with negative.

Step 1: Create the Left Control

1. **Create** a nurbs curve that the animator will use as a control for the wing (Figure 2).
2. **Move** the **rotate pivot** to the position of the joint (Figure 3).
3. **Parent** the curve under the wing joint and **Freeze Transforms**. This will align it so the rotations follow the joint.
4. **Group** the curve to itself and un-parent the group.
5. **Duplicate** the wing joint and **parent** the duplicated joint to the control curve.
6. **Orient Constrain** the original wing joint to the duplicated joint.
7. **Hide** the duplicated joint (Figure 4).

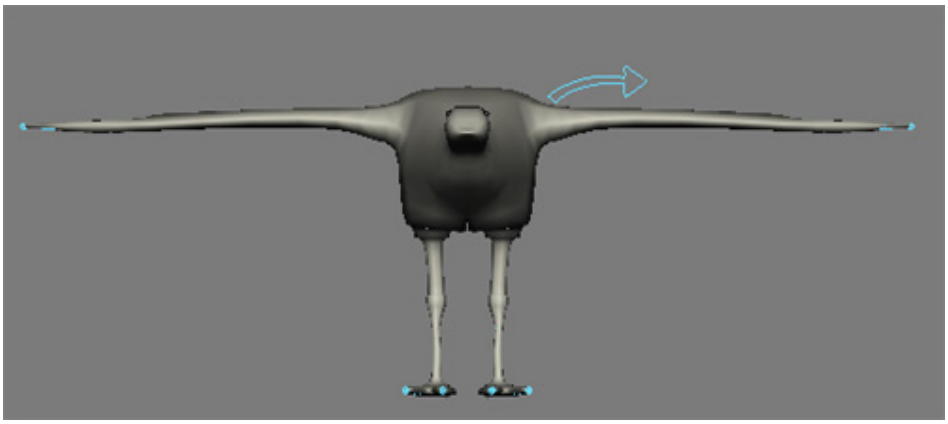


Figure 2 – Created Wing Control

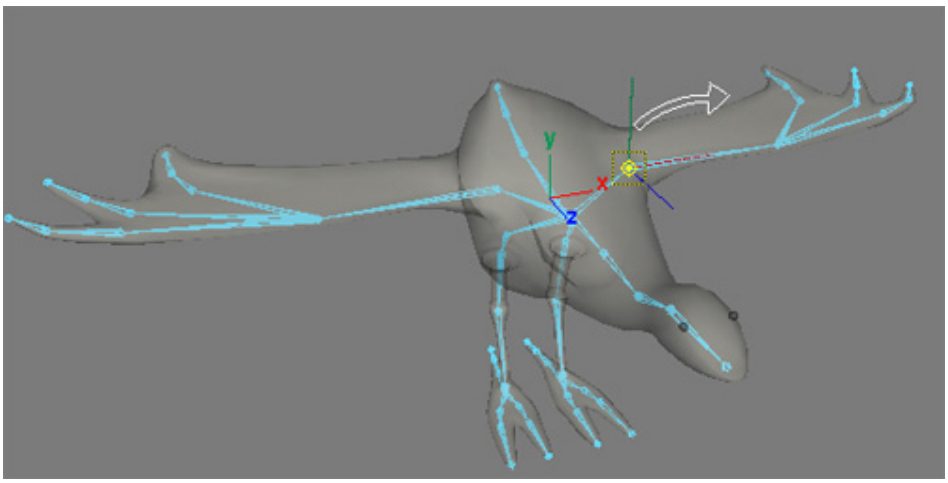


Figure 3 – Pivot Moved to Joint Position

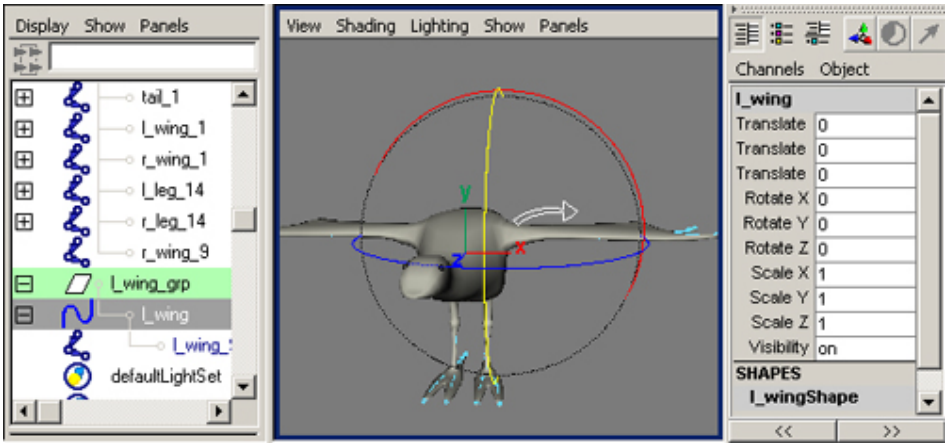


Figure 4 – Left Wing Control

Step 2: Create the Right Control

1. **Duplicate** the left control and name it **r_wing**. Delete the children (Figure 5).
2. **Group** it to itself.
3. **Rotate** the group so the new control is flipped 180 degrees around the **world Z**-axis* (Figure 6).
4. **Rotate** the group again 180 degrees around the **world Y**-axis. The control curve should be on the same side of the X-axis as the original control, but flipped upside down and below it (Figure 7).
5. **Move** the group to the position of the **right** wing joint (Figure 8).
6. **Un-parent** the group.
7. **Rotate** the **r_wing** curve 180 degrees around the **world Z**-axis. (Figure 9).
8. **Scale** the **r_wing** curve –1 in the **world Z**-axis (Figure 10).
9. **Freeze Transforms** on the curve.
10. **Duplicate** the right wing joint and **parent** the duplicated joint to the control curve.
11. **Orient Constrain** the original wing joint to the duplicated joint.
12. **Hide** the duplicated joint.
13. You now have controls that mirror each other! (Figure 11).

** When the **world** axis is mentioned, It is intended for you to rotate the object in relation to the direction it is facing* in the world *not, in relation to it's own orientation. It's easiest to do this using the rotation mode **Global** with snapping turned on.*

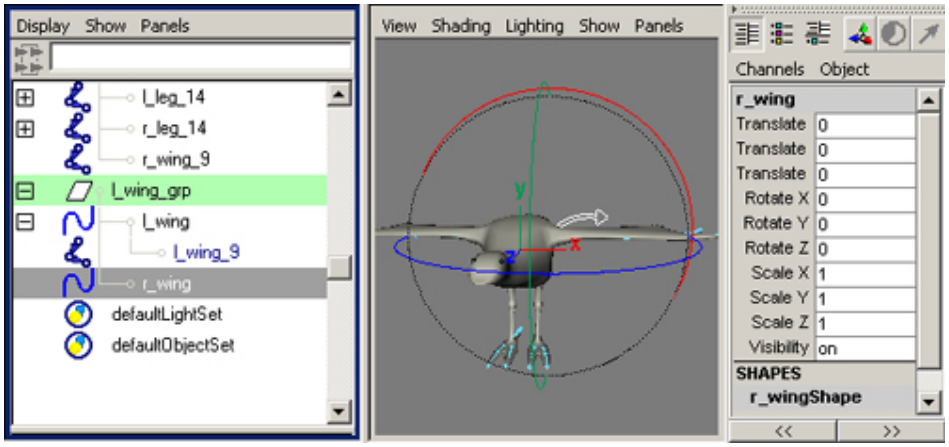


Figure 5 – Duplicate Left Control to Create Right Wing Control

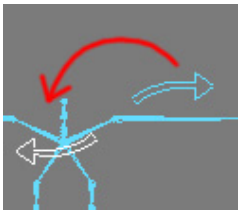


Figure 6 – Rotate Group in World Z

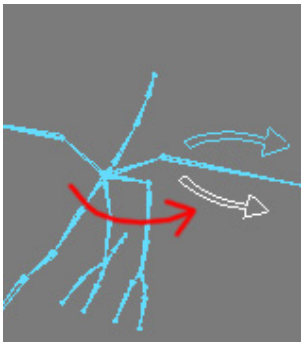


Figure 7 – Rotate Group in World Y

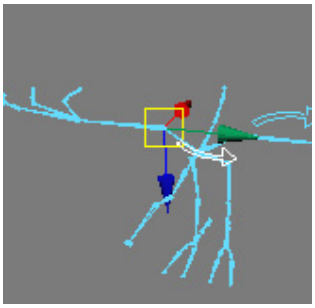


Figure 8 – Group Moved

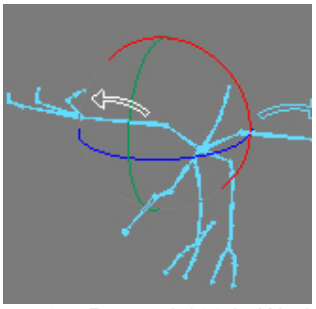


Figure 9 – Rotated 180 in World Z

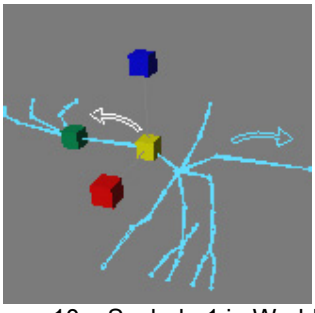


Figure 10 – Scaled –1 in World Z

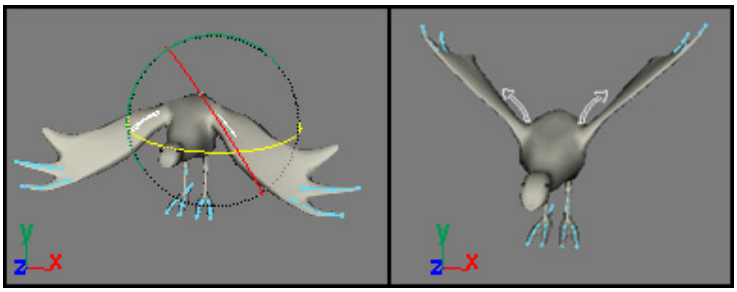


Figure 11 – Mirrored Controls

6. Body Segment Rigs – Feet and Legs

A solid leg and foot setup is key to animating a character successfully. In this example, we'll take a look at a standard rigging technique in order to fully appreciate how grouping ikHandles and adding attributes can make for an easier task when it comes to animate.

Step 1: Create the Joints

The leg joints for a standard setup are created as in Figure 1. Start with the Hip, and then go down to the knee, foot, ball of the foot, and toe. You can name the joints as shown in Figure 2 to keep it simple.

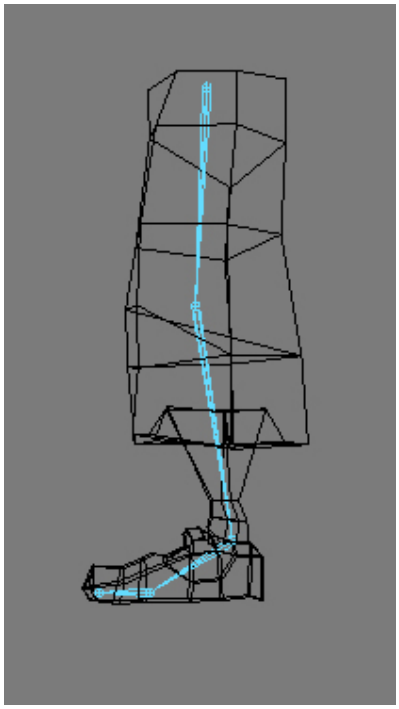


Figure 1 – Joint Layout

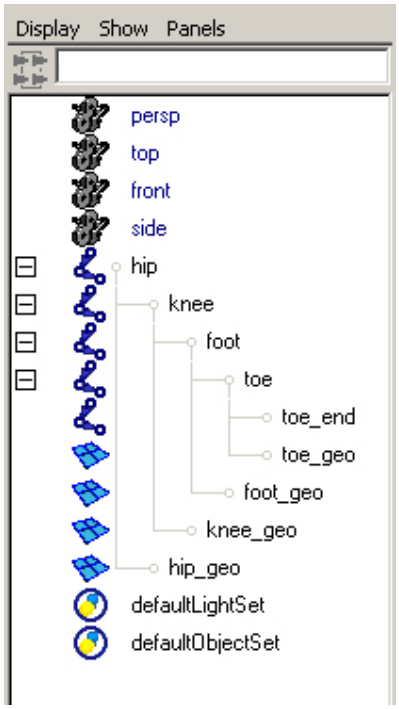


Figure 2 – Joint Names

Step 2: Create IK Handles

Three ikHandles are used in this example, one that goes from the hip to the foot, one from the foot to the toe, and one from the toe to toe_end. By grouping these handles together and adding control attributes, we will be able to achieve any of the poses we need for our animation.

1. **Create** an ikHandle using the **ikRPsolver** from **hip** to the **foot**. Call it **leg_ikHandle**.
2. **Create** an ikHandle using the **ikSCsolver** from the **foot** to the **toe**. Call it **foot_ikHandle**.
3. **Create** an ikHandle using the **ikSCsolver** from the **toe** to **toe_end**. Call it **toe_ikHandle**. (Figure 3)

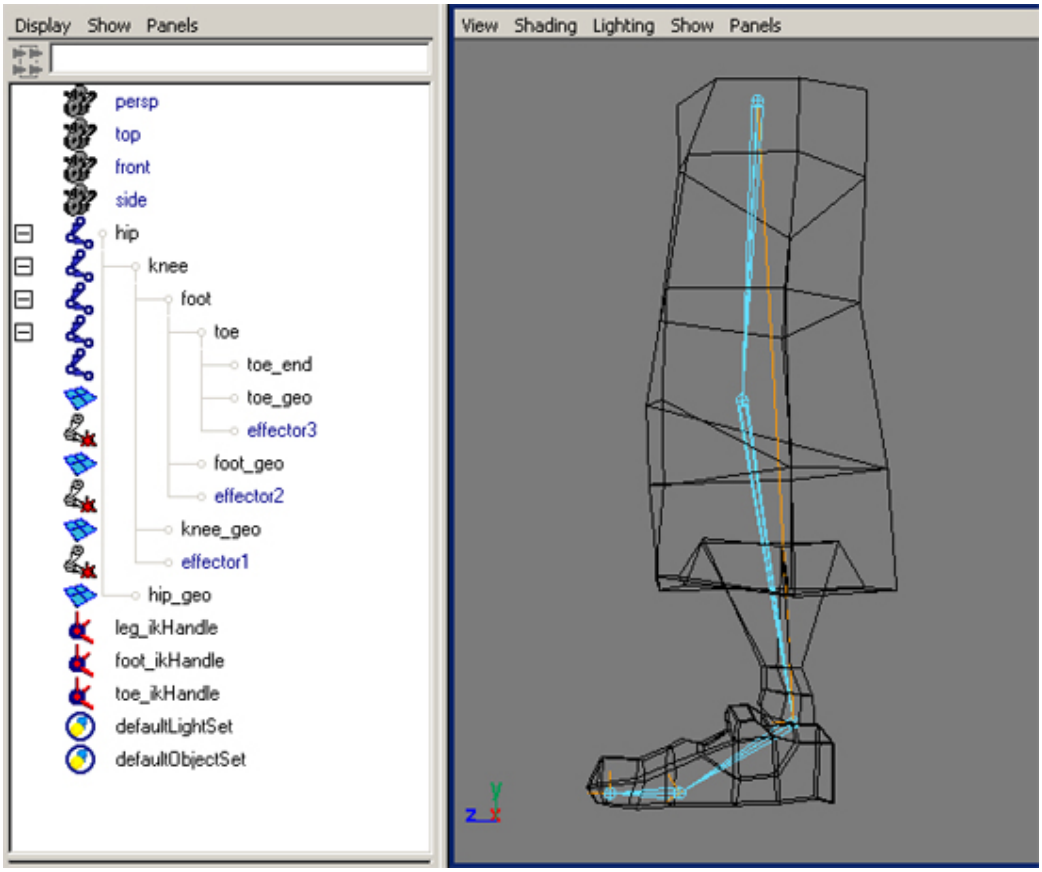


Figure 3 – Ik Handles

Step 3: Create a Foot Control

The foot control is simply a nurbs curve, which we will use to manipulate the character's foot. The idea is to make it instantly recognizable so the animator knows simply by looking at it that it will allow him or her to animate the leg.

1. **Draw** a curve for the foot control (Figure 4). Name it **foot_ctrl**.
2. **Move** the pivot so it's at the character's ankle or foot joint (Figure 5).
3. **Set** the **Rotation Order** so it's ZXY or XZY.
4. **Parent** the **ikHandles** under the **foot_ctrl** (Figure 6).

Figure 4 – Foot Control Curve

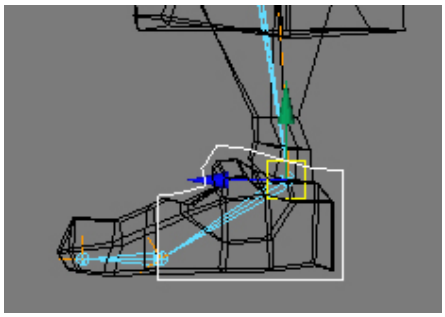


Figure 5 – Moved Rotate Pivot

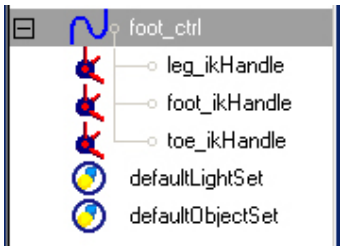


Figure 6 – Parented IkHandles

Step 4: Create a Ball Pivot

The next step is to begin adding the extra controls to make the foot setup more interesting. The first control we add will allow the character to pivot off the ball of it's foot.

1. **Select** the **leg_ikHandle** and **group** it to itself using **ctrl-g** (Figure 6). Name it **ball_pivot**
2. **Move** the **rotate pivot** of **ball_pivot** to the location of the **toe** joint (Figure 7).
3. **Rotate** the control to get an idea of how the heel can now pivot around the ball of the foot (Figures 8-9).

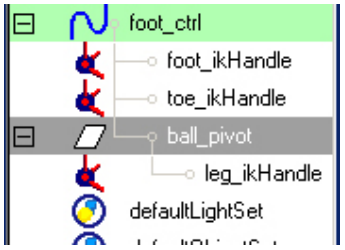


Figure 7 – Grouping Leg ikHandle

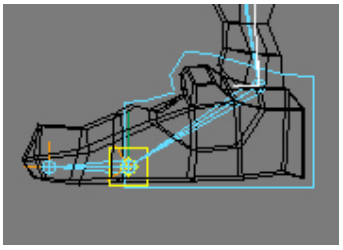


Figure 8 – Moved Rotate Pivot

Figure 9 – Not Rotated

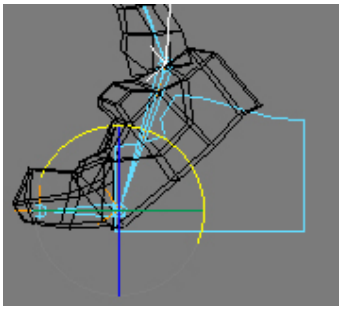


Figure 10 - Rotated

Step 5: Add Toe Wiggle

Toe wiggle is added in much the same way as ball_pivot.

1. **Select** the **toe_ikHandle** and **group** it to itself using **ctrl-g** (Figure 11). Name it **toe_wiggle**.
2. **Move** the **rotate pivot** of **toe_wiggle** to the location of the **toe** joint.
3. **Rotate** the control to get an idea of how the toe wiggles (Figure 12).

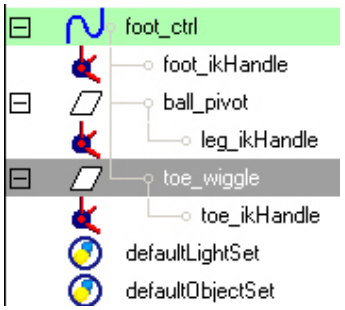


Figure 11 – Grouping Toe ikHandle

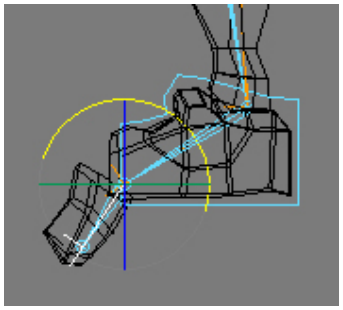


Figure 12 – Toe Wiggled

Step 6: Add Toe Lift

Toe lift is used in those rare cases when you really want your character to have it's toe planted in the ground. Unless you're animating a ballerina, you may never need this control, but it's good to have *just in case*. The toe lift control is again added in the same way as the ball pivot and toe wiggle.

1. **Select foot_ikHandle**, **ball_pivot**, and **toe_wiggle** and **group** them using **ctrl-g** (Figure 13). Name it **toe_lift**
2. **Move** the **rotate pivot** of **toe_lift** to the location of the **toe_end** joint (Figure 14).
3. **Rotate** the control to get an idea of how the foot can now lift off the ground (Figures 15-16).

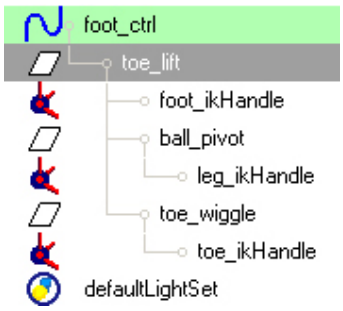


Figure 13 – Grouping Foot ikHandle, ball_pivot and toe_wiggle

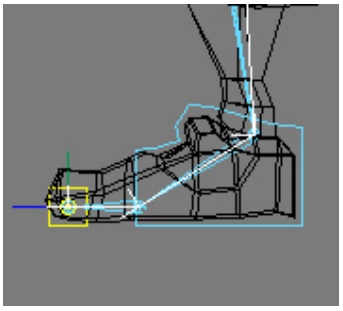


Figure 14 – Moving the Pivot

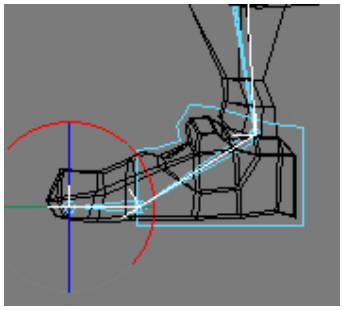


Figure 15 – Not Rotated

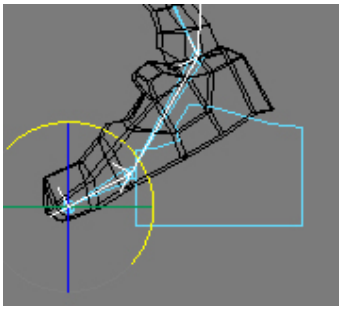


Figure 16 - Rotated

Step 7: Add All Controls to One Node

The job of the creature rigger is to make animation as simple as possible. In this case, the easiest way to do this is to put all the animation controls onto one node, so the animator doesn't have to select more than they need to. To accomplish this, we're going to add extra attributes to the foot_ctrl to handle ball_pivot, toe_wiggle, toe_lift, and even toe_twist.

1. **Select foot_ctrl**.
2. Go **Modify→Add Attributes** to bring up the add attribute dialogue box (Figure 17).
3. **Add** attributes: **ball_pivot, toe_wiggle, toe_lift,** and **toe_twist**. All should be of type **float** (Figure 18).

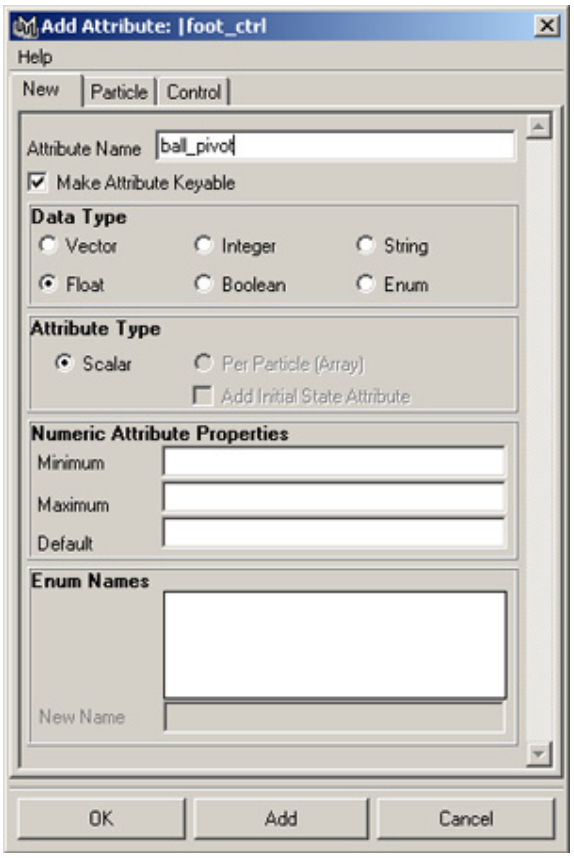


Figure 17 – Add Attributes Window

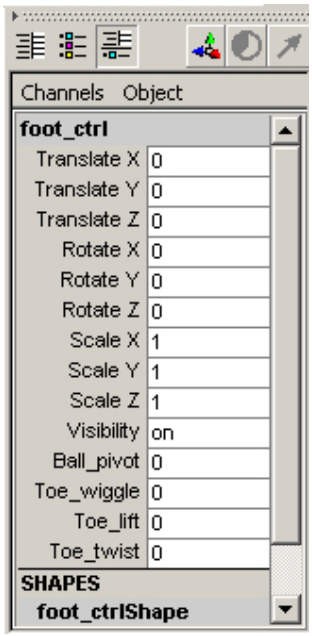


Figure 18 – Attributes Added

4. With the foot_ctrl still selected, bring up the **Connection Editor** (Window→General Editors→Connection Editor).
5. Click **Reload Left**.
6. **Select ball_pivot.**
7. Click **Reload Right** (Figure 19).
8. **Connect** the **ball_pivot** attribute on **foot_ctrl** to the **rotateX** attribute on **ball_pivot** (Figure 20).
9. **Select toe_wiggle**.
10. Click **Reload Right**.
11. **Connect** the **toe_wiggle** attribute on **foot_ctrl** to the **rotateX** attribute on **toe_wiggle** (Figure 21).
12. **Select toe_lift**.
13. Click **Reload Right**.

14. **Connect** the **toe_lift** attribute on **foot_ctrl** to the **rotateX** attribute on **toe_lift** (Figure 22).
15. **Connect** the **toe_twist** attribute on **foot_ctrl** to the **rotateY** attribute on **toe_lift** (Figure 23).

Now all the different aspects of the foot can be driven from one control.

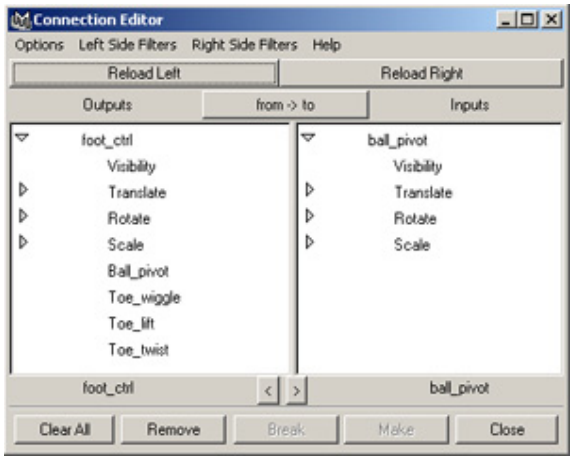


Figure 19 – Connection Editor

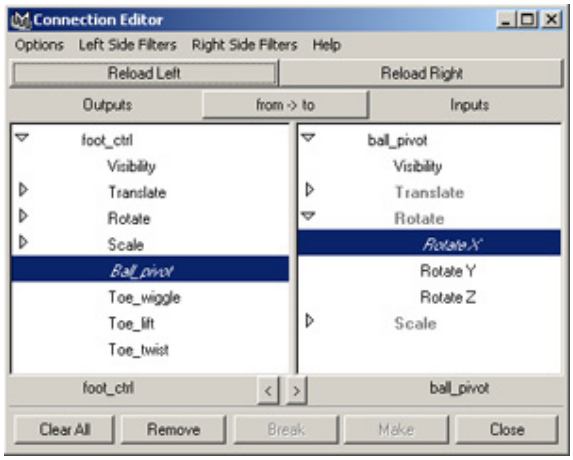


Figure 20 – Connected ball_pivot

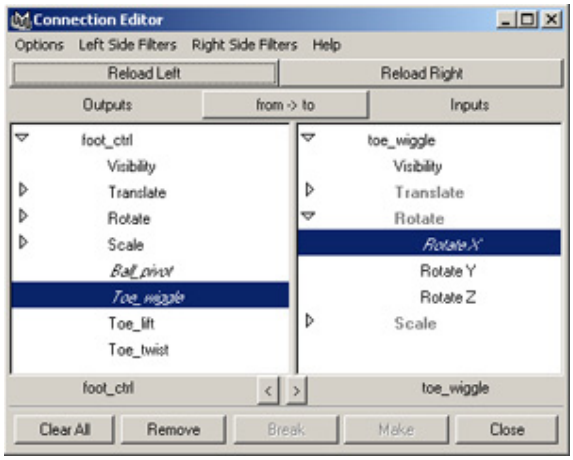


Figure 21 – Connected toe_wiggle

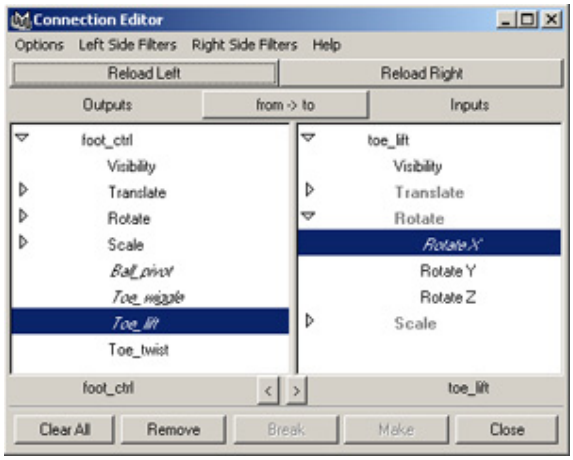


Figure 22 – Connected toe_lift

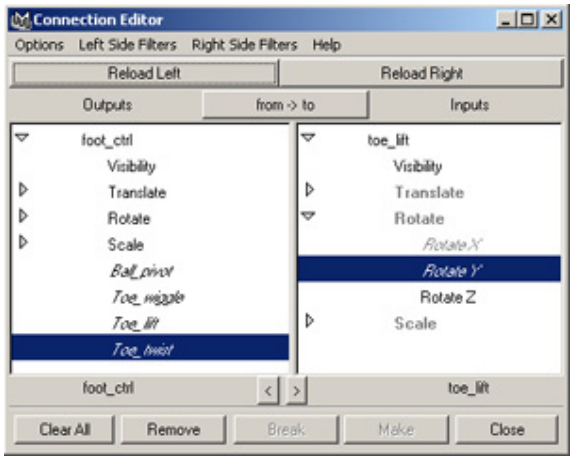


Figure 23 – Connected toe_twist

Step 8: Add Knee Control

Next we have to add a knee control for the leg. As we are using a **rotate plane** solver (ikRPsolver) for the leg, it's easy to add a **poleVectorConstraint** to an object and use that as a knee control. This will aim the leg at the knee control, making it easy for the animator to place the knee exactly where they want.

1. **Create** a nurbs curve to use as a knee locator (Figure 24). Call it **knee_ctrl**.
2. **Center** the pivot point of the knee control by going **Modify→Center Pivot** (Figure 25).
3. **Create** the **poleVectorContraint** by **selecting** the **knee_ctrl**, and then **selecting leg_ikHandle**. Now choose **Constrain→Pole Vector** (Figure 26).
4. **Parent** the knee to the **foot_ctrl**. Now when you move the foot, the knee will follow! You can also grab the knee_ctrl and modify it to guide the knee where you want (Figure 27).

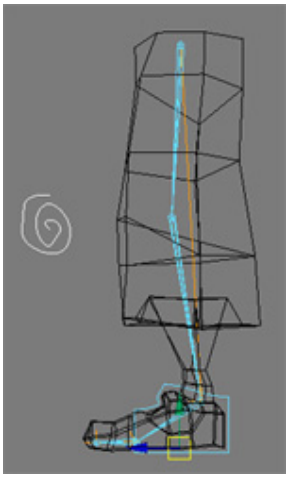


Figure 24 – Knee Control

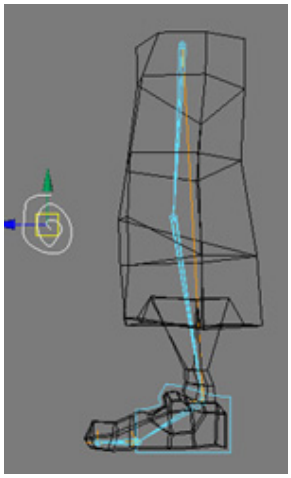


Figure 25 – Centered Pivot

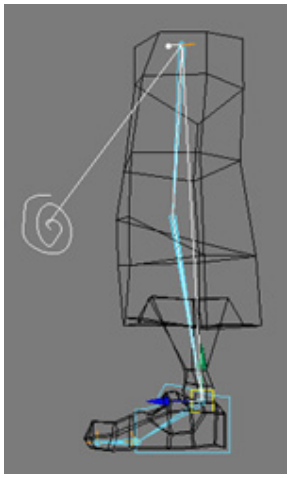


Figure 26 – Pole Vector Constraint

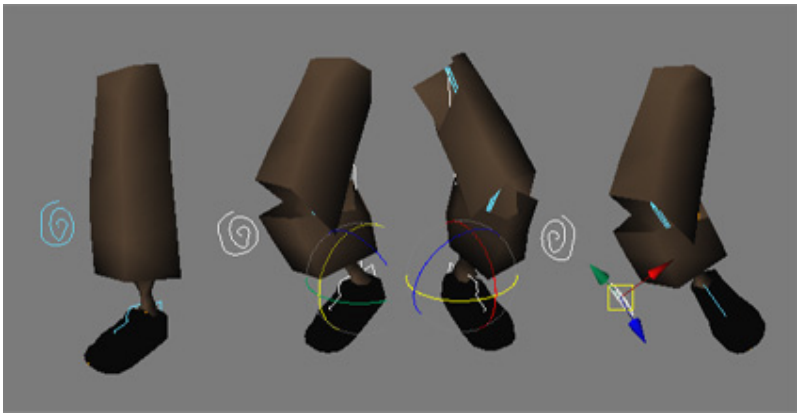


Figure 27 – Foot And Knee Moving

Step 9: Another Knee Parenting Option

While the above method for placing the knee will work in most cases, it can cause a problem when the foot is rotating in extremes. For example, in Figure 28 the foot is rotating up. Because the knee control is crossing the line between the hip and the foot, it causes the leg to flip. The following setup method helps prevent flips like this by allowing the knee to follow the foot's translation, but follows the rotation in *only* the Y-axis.

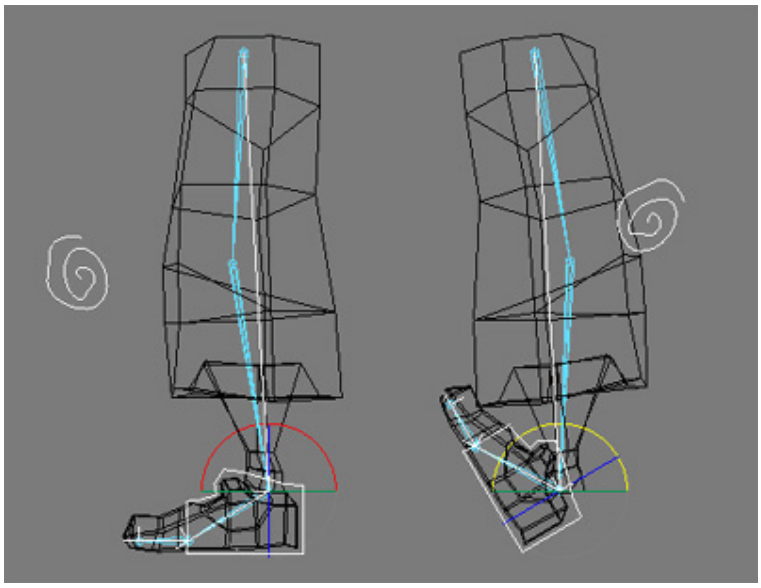


Figure 28 – Foot Rotates and Causes Knee Flipping

1. **Un-parent** the **knee_ctrl** so it's no longer a child of the foot control.
2. **Duplicate** the **foot_ctrl** and **delete all the children** of the duplicated control, *including the shape node* (Figure 29).
3. **Rename** the duplicated control **knee_gr**p (Figure 30).
4. **Parent** the **knee_ctrl** under **knee_grp** (Figure 31).
5. **PointConstrain** the **knee_grp** to the **foot_ctrl**, so when the **foot_ctrl** moves, the **knee_grp** follows.
6. Move the foot around. See how the knee follows it's motion? Now rotate the foot, the knee doesn't follow the rotation.
7. Use the **connection editor** (**Window→General Editors→Connection Editor**) to have the **rotateY** attribute of **foot_ctrl** drive the **rotateY** attribute of **knee_grp** (Figure 32).

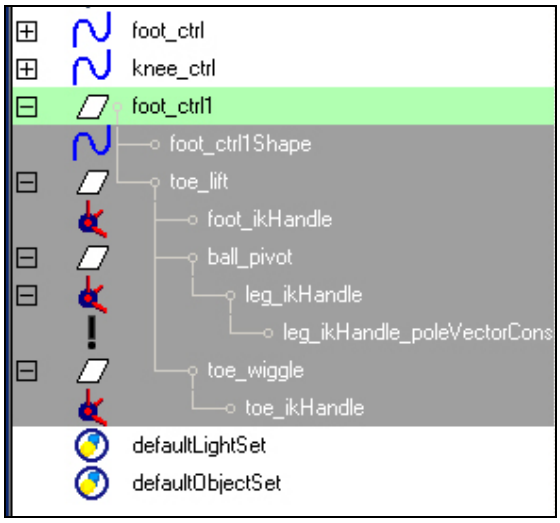


Figure 29 – Delete All Children of Duplicated Ctrl

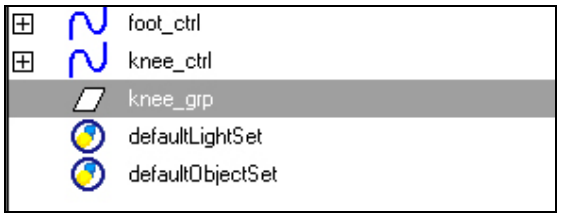


Figure 30 – Rename **knee_grp**

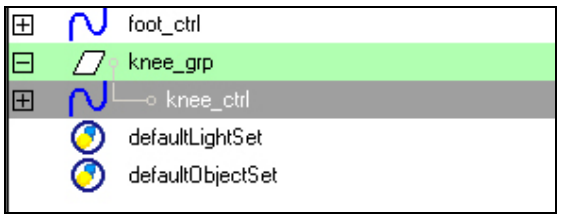


Figure 31 – Parent **knee_ctrl** to **knee_grp**

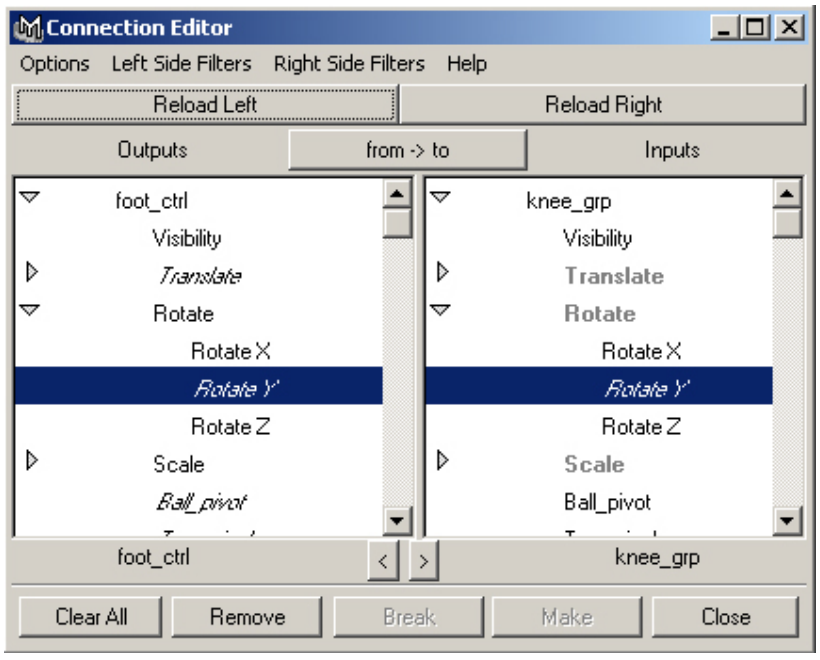


Figure 32 – Connect **foot_ctrl.rotateY** to **knee_grp.rotateY**

Step 10: Cleanup

The final step is all about cleaning up the scene and making it presentable to the animator. This involves:

- Hiding nodes you don't want seen (ikHandles, etc).
- Removing extra attributes (scale, etc).
- Creating Display Layers

Hiding Nodes

1. Hide all the ikHandles

Remove Extra Attributes

1. Bring up the Channel Control window (Window→General Editors→Channel Control).
2. Make the **scale** and **visibility** attributes non-keyable by moving them from Keyable to Non-Keyable (Figure 33). For extra protection, lock the attributes as well.

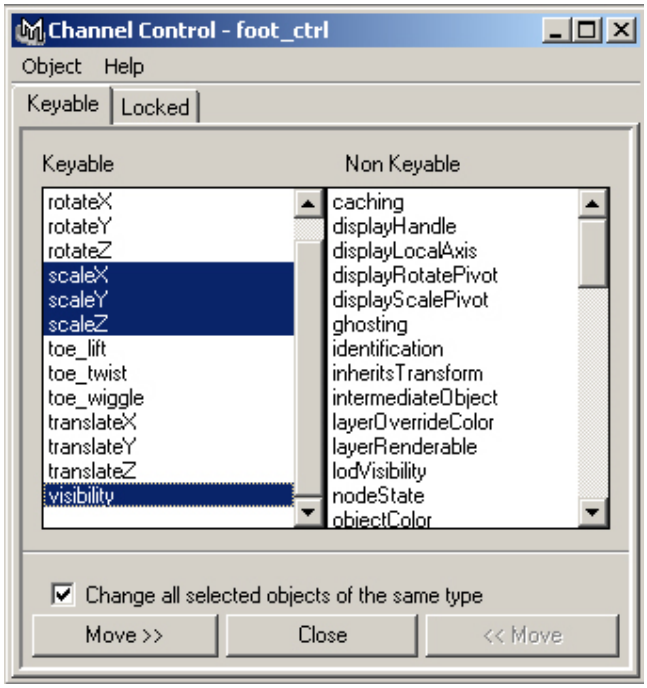


Figure 33 – Making Scale and Visibility Attributes Non-Keyable

3. Do the same for all attributes *except* **translate** on the **knee_ctrl** (Figure 34).

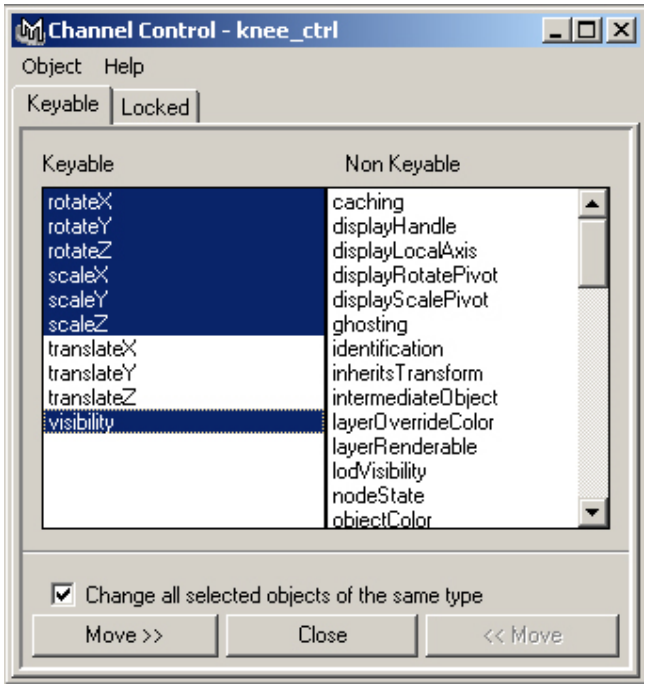


Figure 34 – All Attributes Except Translate are Non-Keyable

Create Display Layers

1. Click on the “Create a New Layer” button in the Layer Editor in the Channel box to create 2 new layers (Figure 35).
2. Name them “Untouchables” and “Controls” (Figure 36).
3. Make the “Untouchables” layer **referenced**, and make the “controls” layer **green** in colour (Figure 37).
4. **Select** the **hip** and assign it to the **Untouchables** layer.
5. **Select** the **foot_ctrl** and **knee_ctrl** and assign them to the **Controls** layer.

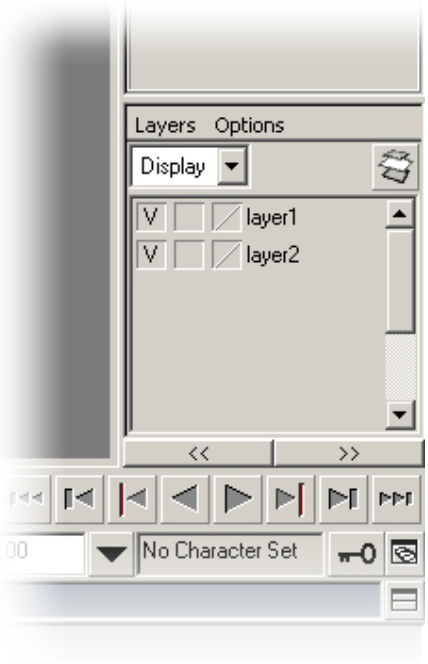


Figure 35 – Create Two Layers

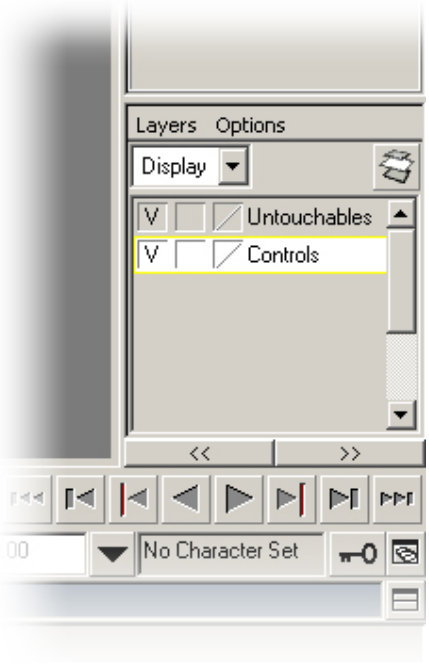


Figure 36 – Name the Layers

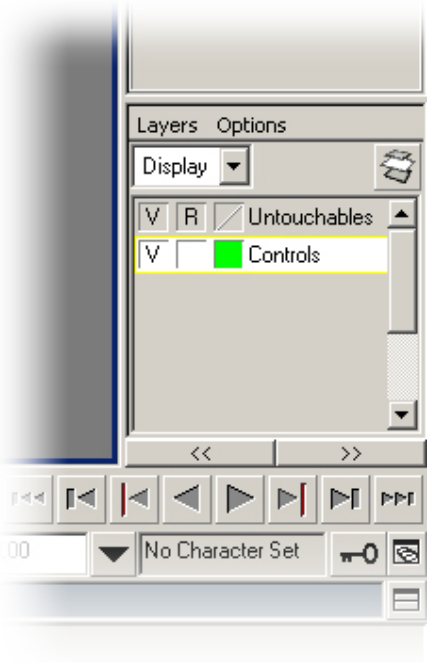


Figure 37 – Set the Layer Attributes

You now have a foot control that is fast, easy to use, intuitive, and allows for almost any type of action the animator wants to create.

7. Body Segment Rigs – Inverse Kinematic Fingers

Traditionally, a character's fingers are animated in one of two ways, either setting values based on explicit rotations of the finger joints, or by a series of expressions driven by clever attribute names such as “bend finger” or “spread”. In many cases, this works perfectly well, and may in fact cover most of what a character needs to do. However, there are cases where it is important to be able to place a character’s fingers on a table, or against a wall. In situations like this, an inverse kinematic solution can save hours of animation time. In order to provide the utmost control to an inverse-kinematic finger, it’s important to provide control that is traditionally not available in an IK solution, i.e. it’s necessary to “break” a joint in the finger.

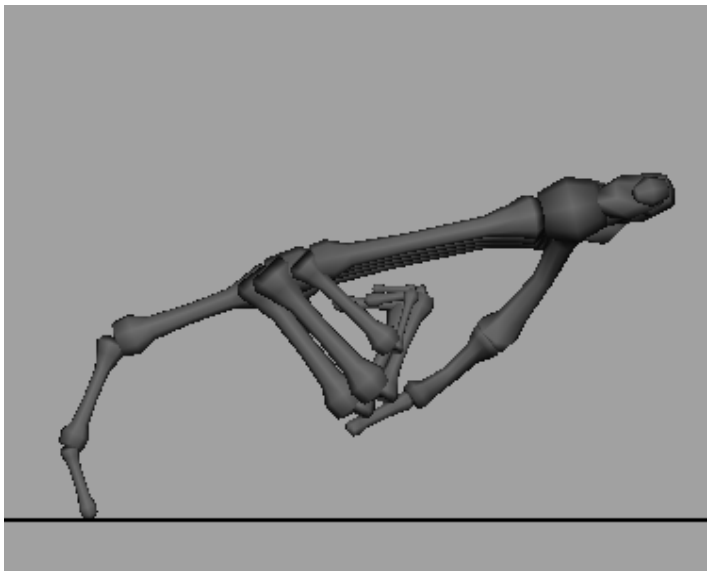


Figure 1 - normal finger

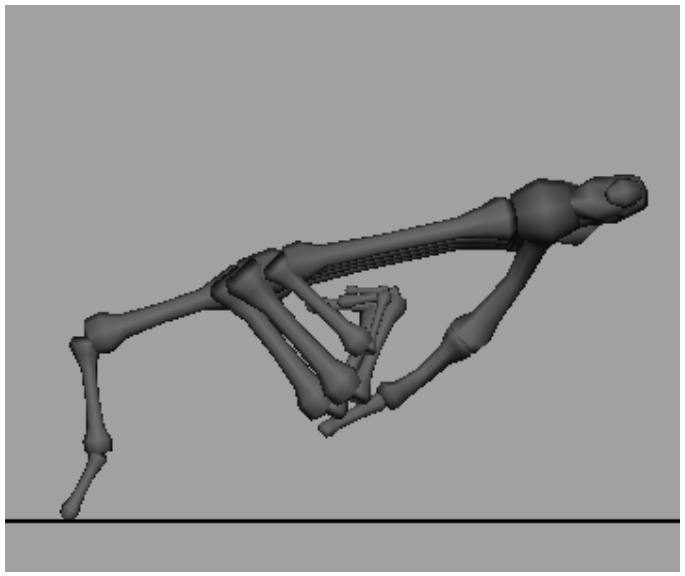


Figure 2- broken finger

Notice in Figure 1, the tip of the finger is bent in towards the palm. This is what a standard inverse kinematic solution would provide. In Figure 2, you can see how the fingertip is “broken”. Often, the animator will want to put a character’s fingers in a pose like this to show weight, or if they’re dragging the fingers along a surface. The following solution will demonstrate a method of providing controls for the animator that are intuitive and powerful.

Step 1: Create ikHandles for the fingers

The first step in creating an inverse kinematic solution for fingers is to actually create the ikHandles. To do this, we're going to need two ik handles for each finger. Figure 3 demonstrates the areas to be affected by IK.

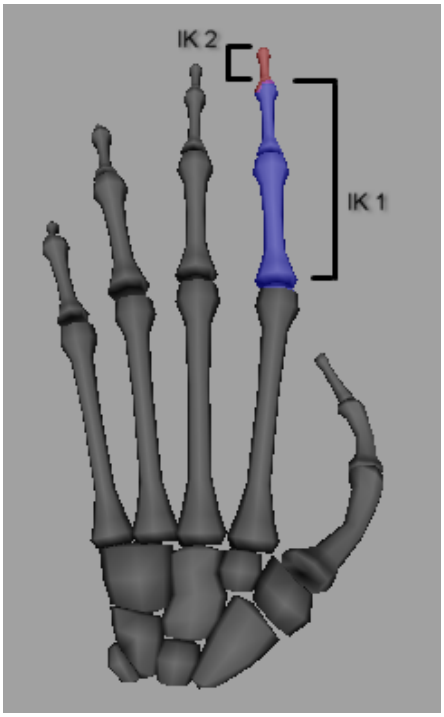


Figure 3 – ik sections

1. Create an ikSCsolver ikHandle going from **phalange_2_1** to **phalange_2_3**.
2. Name the ikHandle "phalange_2_1_ikHandle".
3. Create an ikSCsolver ikHandle going from **phalange_2_3** to **phalange_2_3_end**.
4. Name the ikHanlde "phalange_2_3_ikHandle".

Step 2: Create a finger tip control

Now that the IK is created, we need a simple control that the animator can grab which will allow them to move the fingertip around. A nurbs circle works well for this.

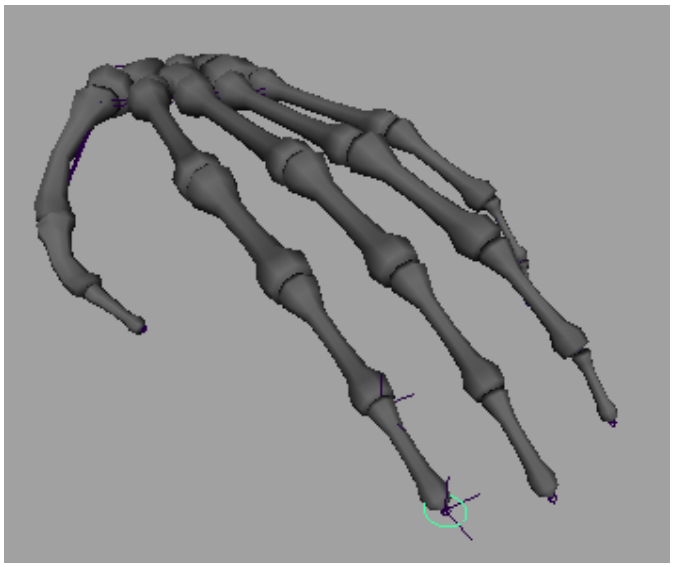


Figure 4 – Nurbs circle placed at the tip of the finger.

1. Create a nurbs circle
2. Position it at the tip of the finger.
3. Name the circle “phalange_2_ctrl”.

Step 3: Parent ikHandles under finger control

It would appear that all we would have to do at this point is parent the ikHandles under the fingertip control, and the setup would be done. Unfortunately, this isn't the case. Notice what happens when the ikHandles are parented and the finger control is moved to the side (figures 5 and 6).

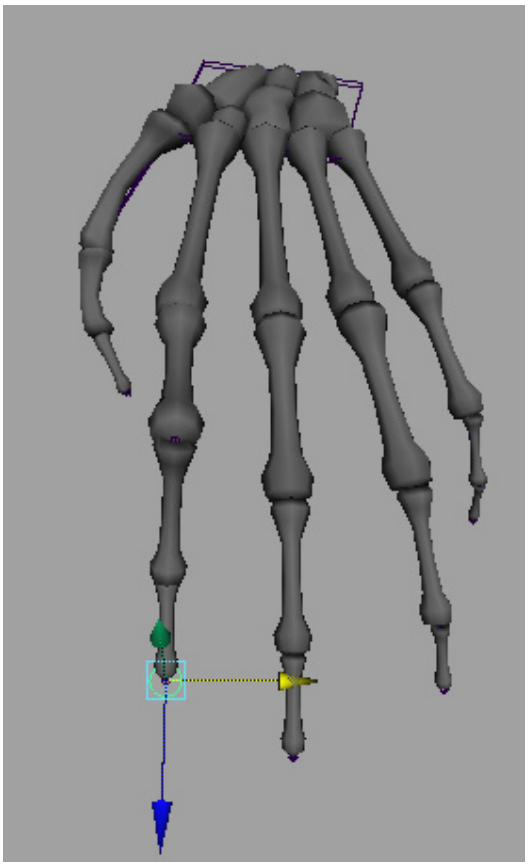


Figure 5 – control stationary

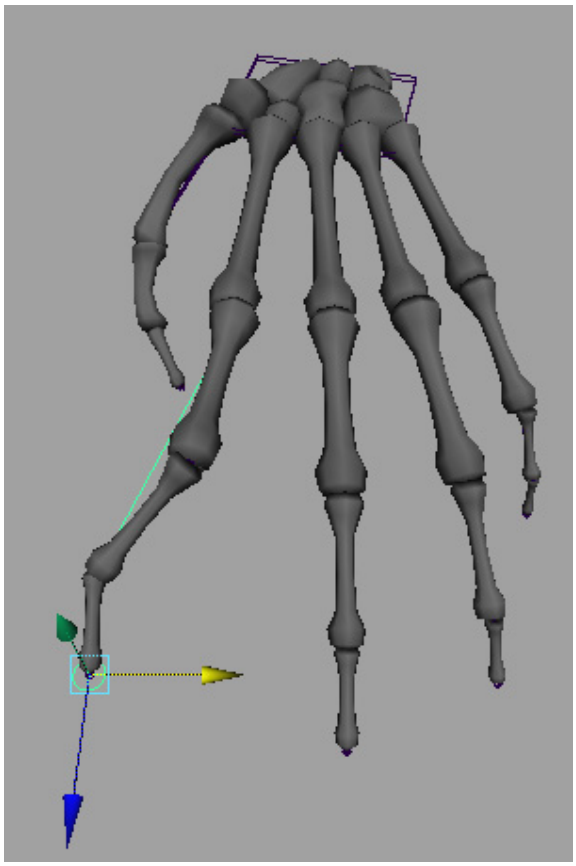


Figure 6 – Control moved to side

Due to the fact that we can't limit the start joint for the second ik handle, we're able to break the finger in a way that is completely undesirable. What we want is for the ikHandles to line up as if they're always pointing from the tip to the base of the finger (figure 7).

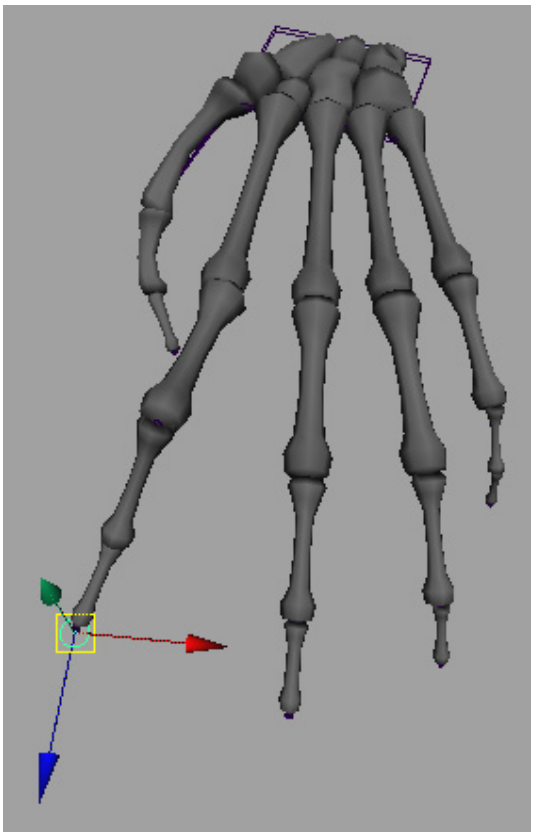


Figure 7 – Finger aims down length correctly.

Step 4: Create a control that will aim the ikHandles back to the hand.

The best way to keep the finger from breaking is to create a joint that will be positioned with the finger control, and will aim back to wherever the hand is located. Then we can parent the ikHandles under this joint to make sure they keep correct alignment.

1. **Create** a joint that starts at the end of the finger (phalange_2_3_end), and ends at the base of the finger (phalange_2_1_end). (Figure 8)
2. **Name** the joints **phalange_2_aim and phalange_2_aim_end**.
3. **Create** an **ikSCsolver ikHandle** that goes from phalange_2_aim to phalange_2_aim_end.
4. **Name** the ikHandle **phalange_2_aim_ikHandle**.
5. **Parent** the ikHandle to the joint just above phalange_2_1 (**meta_fing_1**). (Figure 9)
6. **Point Constrain** the phalange_2_aim joint to the phalange_2_ctrl.
    a. **Select** phalange_2_ctrl and phalange_2_aim.
    b. **Choose** Constrain→Point

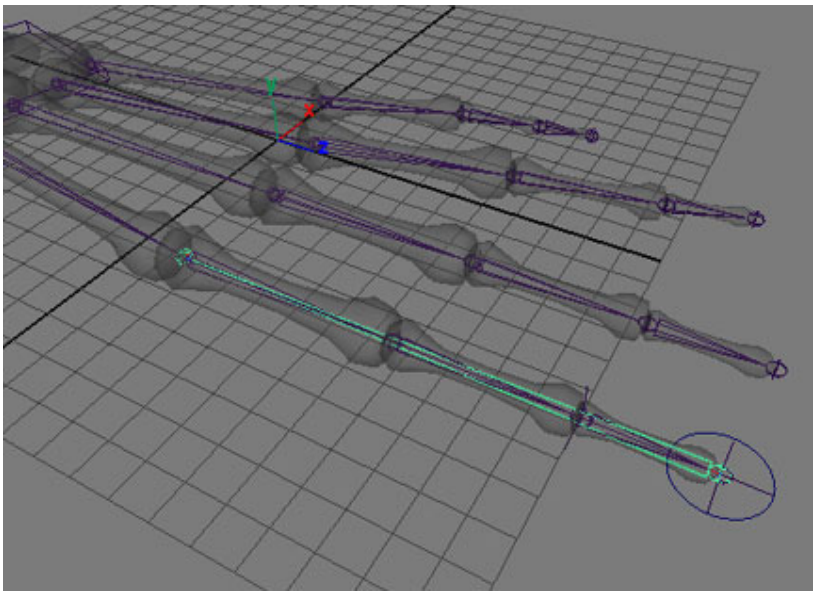


Figure 8 – phalange_2_aim joint

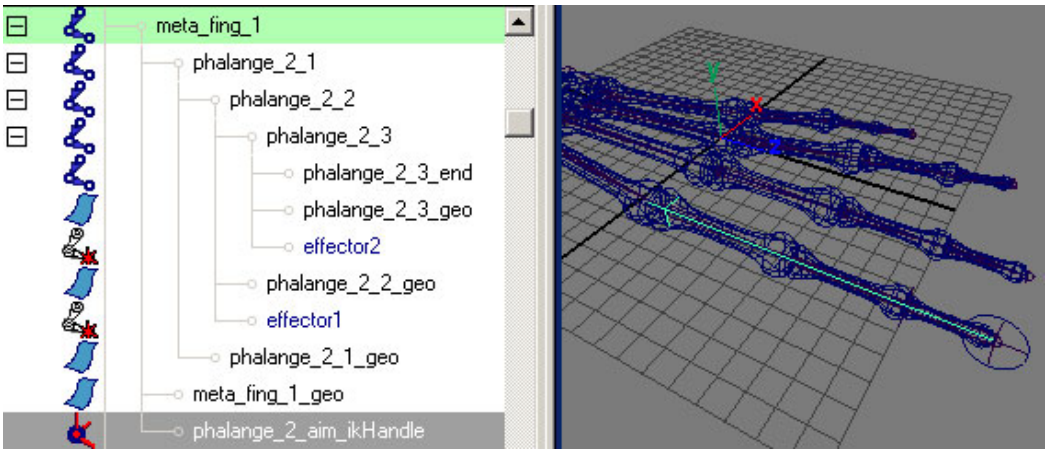


Figure 9 – aim ikHandle parented under meta_fing_1

Step 5: Parent the ikHandles to the aim joint

Now if you parent the ikHandles to the aim joint and move the finger control around, you'll notice that the ikHandles stay in perfect alignment with the finger. (Figures 10 -12).

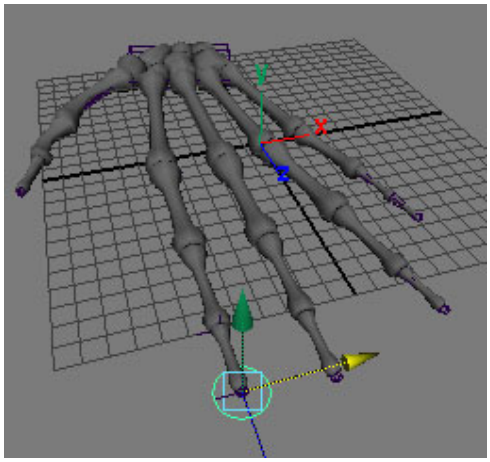


Figure 10

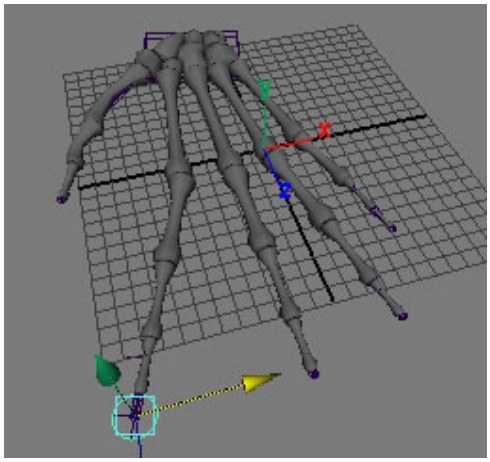


Figure 11

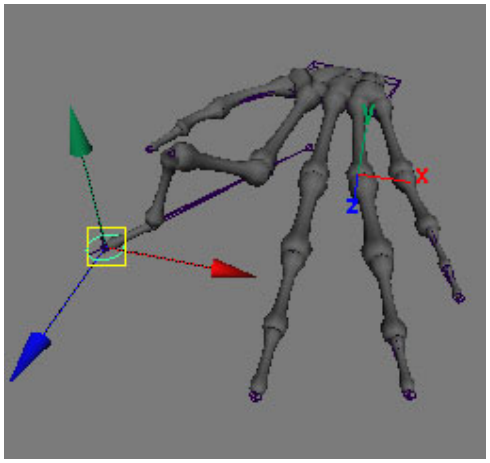


Figure 12

Step 6: Add Flex and Lean controls to the finger tip

Currently the finger doesn't allow for the tip to actually flex or lean from side to side. We need to add some control structures to handle this type of interaction. The first step will be to create a new control for the ikHandles that will be a child of the aimJoint. Once the control is oriented correctly, we can add attributes to make it easier to animate.

1. **Create** a nurbsCircle.
2. **Name** it phalange_2_flex.
3. In the channel box, click on the "makeNurbsCircle2" and **set** the **sweep** value to **180**.
4. Position phalange_2_flex in the same location as phalange_2_ctrl.
5. Parent phalange_2_flex under phalange_2_aim.
6. Set the rotation values so phalange_2_flex looks like figure 13.

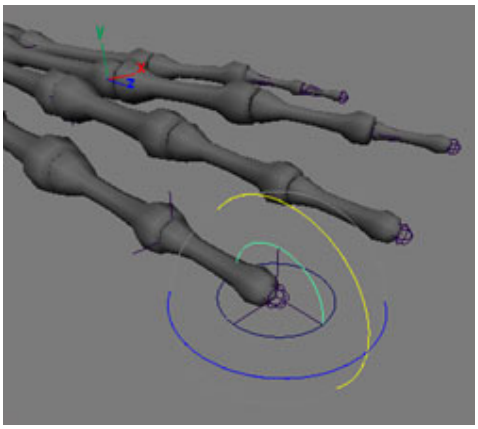


Figure 13 – phalange_2_flex

7. **Freeze transformations** so the phalange_2_flex now has rotations of 0 0 0.
8. Set the **rotation order** on the phalange_2_flex to **zyx** so that it can rotate correctly.
9. **Parent** the two ikHandles for the finger (**phalange_2_1_ikHandle** and **phalange_2_3_ikHandle**) under phalange_2_flex.
10. Rotate phalange_2_flex in X and Y to get an idea of how the finger control works (figures 14-16)

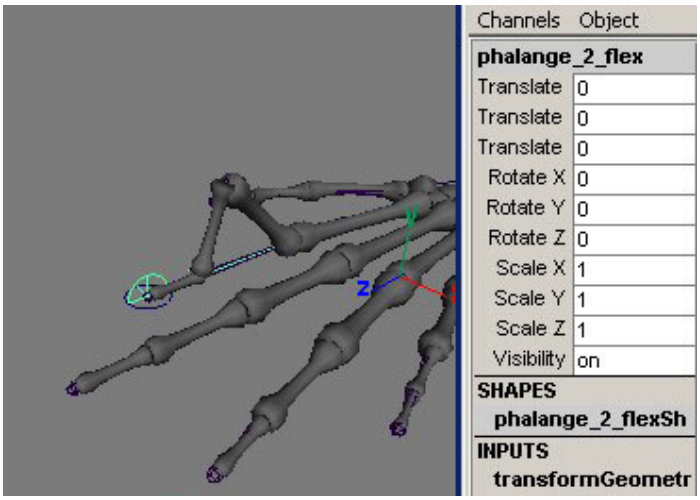


Figure 14 – No rotations on flex control

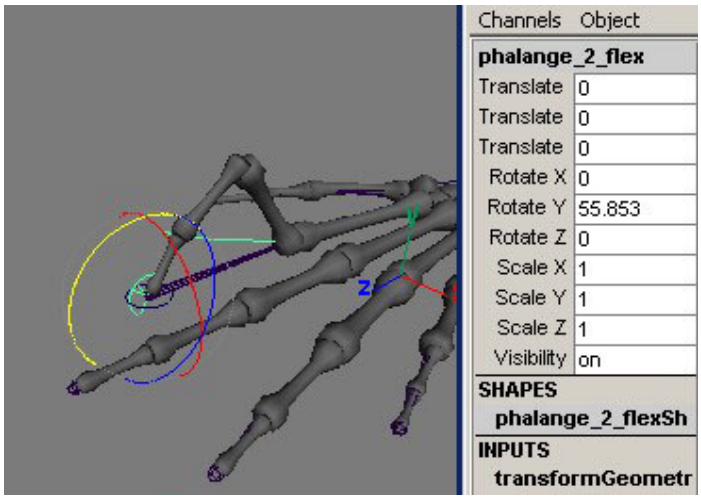


Figure 15 – Rotate Y on flex control

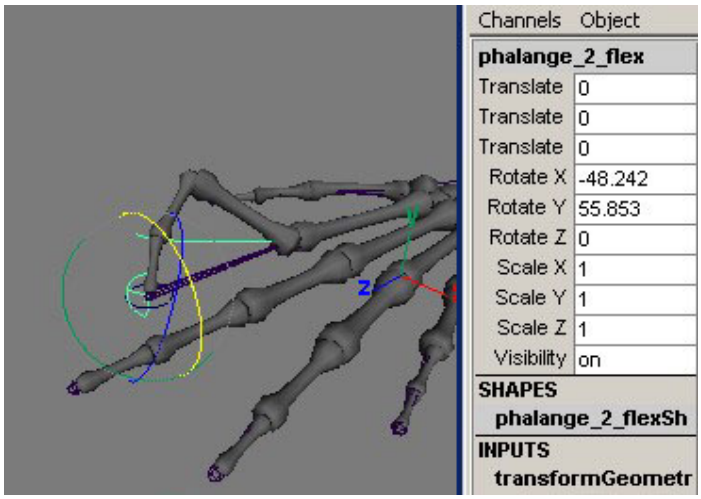


Figure 16 – Rotate X and Y on flex control

Step 7: Add Flex and Lean attributes to finger control

To make it easier for the animator to control the finger, it's best to put all controls on one node. In this case, we will add the **flex** and **lean** attributes on the finger control, and hook them up to the flex node.

1. Add **flex** and **lean** attributes to **phalange_2_ctrl**.
    a. Select phalange_2_ctrl.
    b. Go **Modify→Add Attribute**
    c. Type "flex" for the attribute name, and make sure it is a float.
    d. Click "Add".
    e. Type "lean" for the attribute name.
    f. Click "Add".
2. Bring up the **Connection Editor**
3. With the **phalange_2_ctrl** selected, click on **Reload Left**.
4. Select **phalange_2_flex** and click on **Reload Right**.
5. Connect the **flex** attribute from **phalange_2_ctrl** to the **rotateY** attribute of **phalange_2_flex**. (figure 17)
6. Connect the **lean** attribute from **phalange_2_ctrl** to the **rotateX** attribute of **phalange_2_flex**. (figure 18)

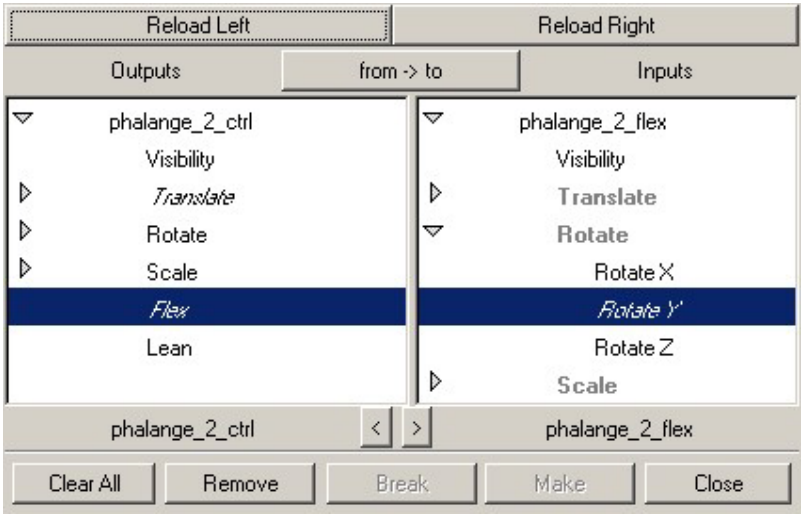


Figure 17 – Connecting the "flex" attribute

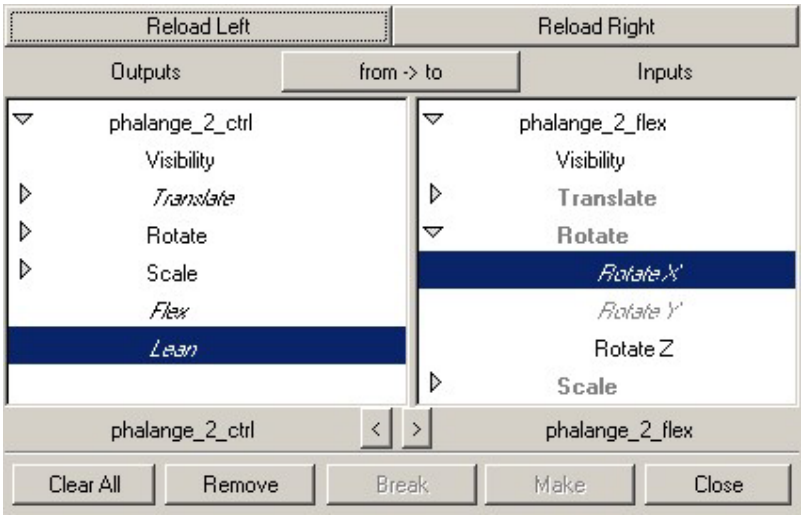


Figure 18 – Connecting the "lean" attribute

Step 8: Clean Up

The final step for creating the control is hiding all the objects that need to be hidden, and making sure that the animator can only control those items we want them to control.

1. **Hide phalange_2_aim** and **phalange_2_aim_ikHandle.**
2. Make the **rotate**, **scale**, and **visibility** attributes for **phalange_2_ctrl non-keyable**.
    a. **Select phalange_2_ctrl**.
    b. Bring up the **Channel Control** editor. (Window→General Editors→Channel Control)
    c. Move all the rotate, scale, and visibility attributes to non-keyable.

If desired, you can also add attributes for controlling the meta_fing_1 joint to the phalange_2_ctrl. This way, animators can have even more control over the way the fingers and hands move.

8. Advanced Concepts – Limit Notification

As mentioned before, one of your tasks when delivering a robust animation control rig is providing something that won't break easily. Animators will push rigs as far as they can go, which we *want* them to do! Animation results are much better when a rig doesn't physically limit an animator. However, sometimes there are things that an animator will do which you just can't allow. In this case, you need to come up with some way of *informing* the animator that what they're doing is illegal, *without* actually removing the ability to do it.

The best method that I've found to do this is with colour-coding the behaviour. For example, if you want to limit the character's arm so it doesn't pass through the body, you can set up a system where the arm will actually *change colour* if it passes through the body. This way the animator can be free to get the motion they want, but they'll be warned that it may break when you go to solve cloth, or apply the muscle model. The animator then becomes free to look for solutions that work *for the animation* instead of trying to work around limits you've placed on the rig.

Colour Based Notification

Let's take a look at a simple example, where we want to check and see if a locator is on the inside, or outside of a surface. If the locator is outside the surface, we'll leave it alone. If the locator moves *inside* the surface, we can turn the surface red to warn the animator that the locator's on the wrong side.

This example uses a couple of interesting Nodes within Maya.

Node	Description
closestPointOnSurface	Finds the closest point on a nurbs surface based on the given world position.
pointOnSurfaceInfo	Returns information about a nurbs surface based on the given UV point.
plusMinusAverage	Adds, Subtracts, or Averages two floats or vectors.
VectorProduct	Provides basic vector math (dot product, cross product, etc).

Using these nodes, our solution will take the following steps:

- Locator is positioned somewhere near the surface (above or below).
- Locator's position is passed into the closestPointOnSurface node in order to find the uv point on the surface that's closest to the locator.
- The u and v points on the surface are passed from the closestPointOnSurface node to the pointOnSurfaceInfo node, giving us the position in world space of the uv point.
- The position of the locator and the position in world space of the uv point are passed to the plusMinusAverage node where they're subtracted from each other. This gives us the vector between the point and the surface.
- The vector from the plusMinusAverage node and the normal from the pointOnSurfaceInfo node are fed to the vectorProduct where their dot product is figured. If the result is <<1, 1, 1>> then the locator is above the surface. If it's <<-1, -1, -1>> then the locator is under the surface.
- The result from the vectorProduct is passed to the shader where a setDrivenKey setup drives the colour of the shader.

This setup can be a bit confusing the first time you go through it, especially if you don't understand vector math. However, if you follow the instructions carefully, you'll soon see how powerful this method of limit notification can be.

Step 1: Create the Nurbs Plane and Locator

The first step is to create the nurbs plane and locator. The nurbs plane can be any shape, and any resolution (Figure 1).

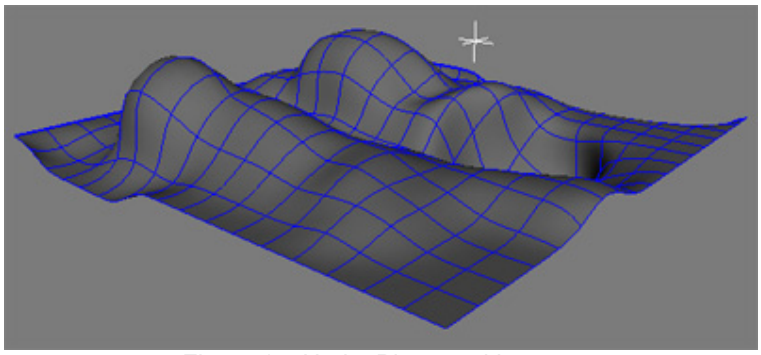


Figure 1 – Nurbs Plane and Locator

Step 2: Create the closestPointOnSurface and pointOnSurfaceInfo Nodes

1. Select the nurbsPlane and the locator and bring up the **Hypergraph**. Go **Graph→Up and Downstream Connections**. This will show you the hypergraph layout for the nurbsPlaneShape and the locator. This is where we're going to do all our connecting of nodes.
2. In the script editor, execute the following commands:

   createNode closestPointOnSurface;
   createNode pointOnSurfaceInfo;

3. Notice as you created the two nodes they appeared in the Hypergraph.
4. With the **Middle Mouse Button**, drag the **nurbsPlaneShape** onto the **closestPointOnSurface** node. This will bring up the connection editor (Figure 2).
5. Connect the **worldSpace** of the **nurbsPlaneShape** to the **inputSurface** of the **closestPointOnSurface** node (Figure 3).

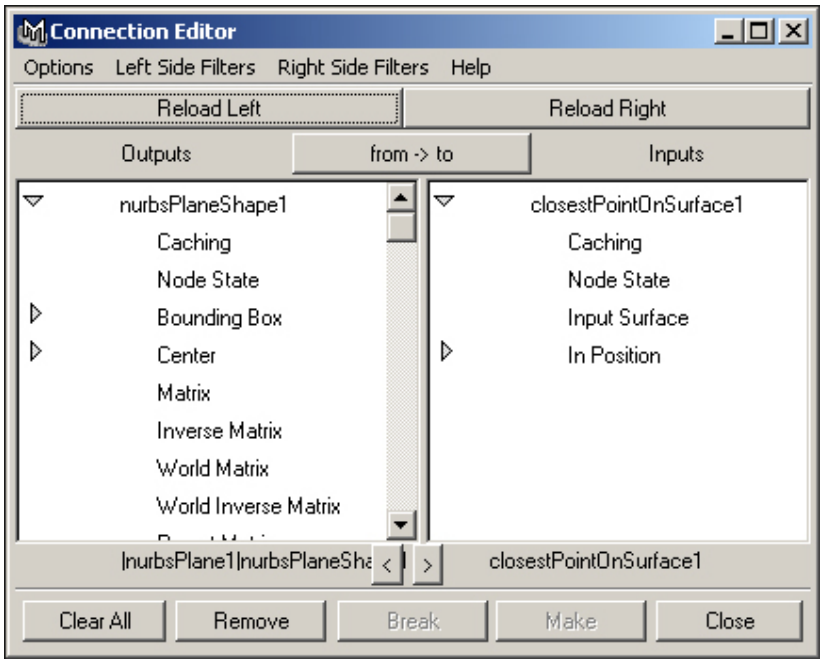


Figure 2 – Connection Editor

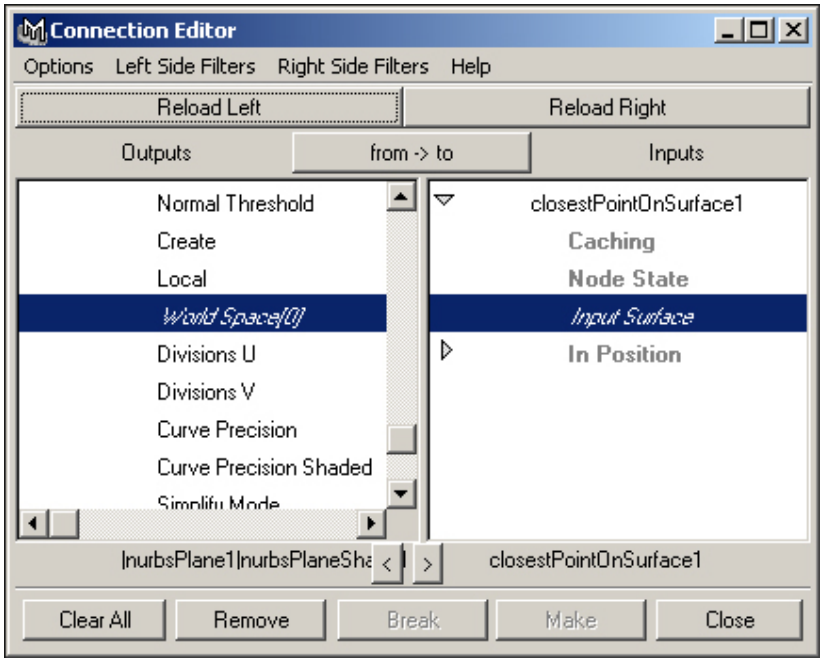


Figure 3 – Connect worldSpace of nurbsPlaneShape to inputSurface of closestPointOnSurface node.

6. With the **Middle Mouse Button** drag the **nurbsPlaneShape** onto the **pointOnSurfaceInfo** node. This will update the connection editor with the new nodes.
7. Connect the **worldSpace** attribute on **nurbsPlaneShape** to the **inputSurface** attribute of the **pointOnSurfaceInfo** node. The Hypergraph should now look like Figure 4.

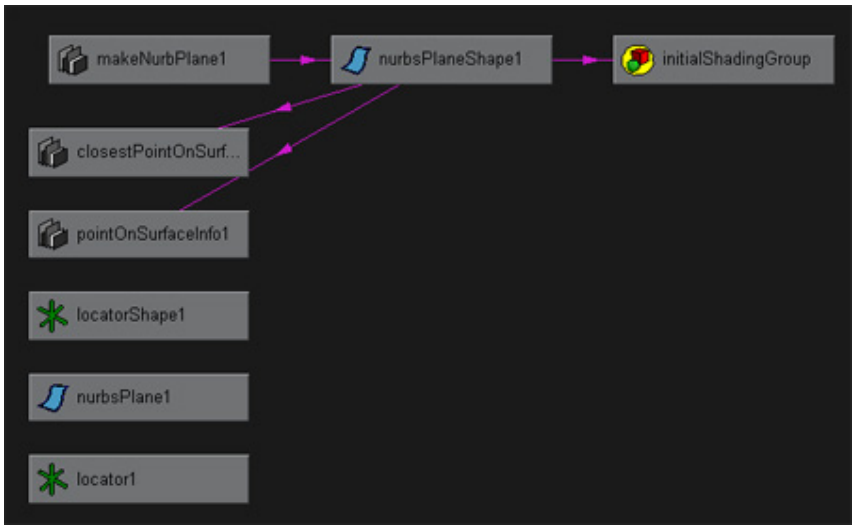


Figure 4 – Hypergraph After Connecting Nodes

Step 3: Connect Locator and Surface Info Nodes

1. With the **Middle Mouse Button** drag the **locator** onto the **closestPointOnSurface** node. This will bring up the connection editor.
2. Connect the **translate** attribute of the **locator** to the **inPosition** of the **closestPointOnSurface** node (Figure 5).

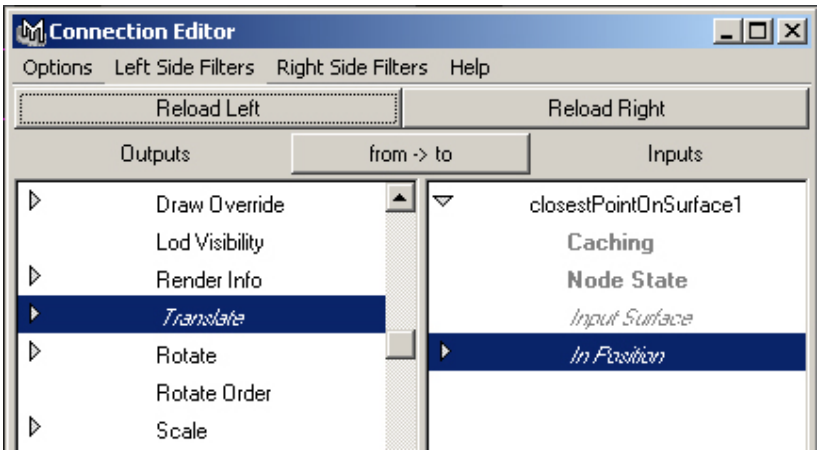


Figure 5 – Connecting the Locator to the closestPointOnSurface Node

3. Next, with the **Middle Mouse Button** drag the **closestPointOnSurface** node onto the **pointOnSurfaceInfo** node. This will update the connection editor.
4. Connect the **parameterU** attribute on the **closestPointOnSurface** node to the **parameterU** attribute on the **pointOnSurfaceInfo** node.
5. Connect the **parameterV** attribute on the **closestPointOnSurface** node to the **parameterV** attribute on the **pointOnSurfaceInfo** node (Figure 6).

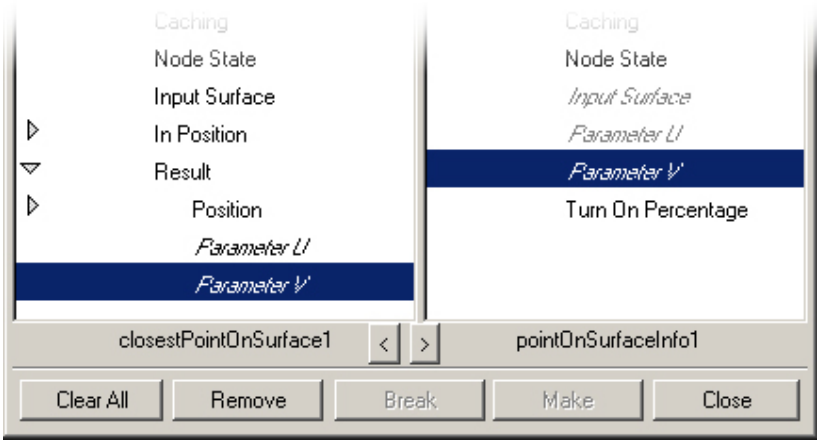


Figure 6 – Connecting the Parameter U and V from the closestPointOnSurface Node to the Parameter U and V on the pointOnSurfaceInfo Node.

6. Select the **pointOnSurfaceInfo** node and rebuild the Hypergraph. The Hypergraph should now look like Figure 7.

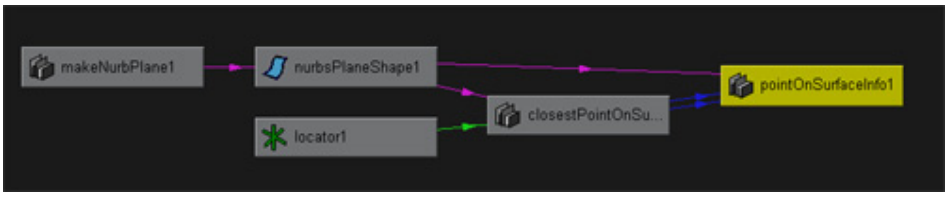


Figure 7 – The Hypergraph

Step 4: Creating the plusMinusAverage and vectorProduct Nodes.

Both the plusMinusAverage and vectorProduct nodes can be created in the interface (unlike the pointOnSurfaceInfo and closestPointOnSurface, which had to be created with Mel).

1. In the Hypergraph, click with the **Right Mouse Button** and choose **Rendering→Create Render Node**.
2. Click on the **Utilities** tab.
3. Click on the **plusMinusAverage** (Figure 8) and the **vectorProduct** (Figure 9) buttons.

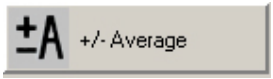


Figure 8 – plusMinusAverage Button

Figure 9 – vectorProduct Button

Step 5: Using the plusMinusAverage Node

In order to find out whether the locator is on the inside or outside of the surface, we first need to get the vector from the closest point on the surface to the locator. To do this, we can use the plusMinusAverage node to subtract the position on the surface from the position of the locator.

1. In the Hypergraph, with the **Middle Mouse Button** drag the **locator** onto the **plusMinusAverage** node. This will bring up the connection editor.

2. Connect the **translate** attribute on the **locator** to the **input3D** attribute of the **plusMinusAverage** node (Figure 10).

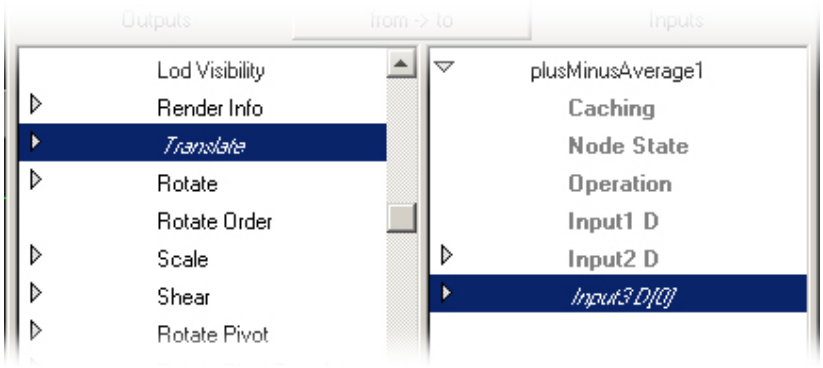


Figure 10 – Connecting the Locator to the First Input on the plusMinusAverage Node

3. With the **Middle Mouse Button** drag the **pointOnSurfaceInfo** node onto the **plusMinusAverage** node. This will update the connection editor.
4. When connecting the **position** attribute on the **pointOnSurfaceInfo** node, **instead of clicking with the **Left Mouse Button** on the **input3D** attribute, click with the **Right Mouse Button**. This will let you connect to the next input for the 3D attribute. This way you end up with the locator connected to the **input3D[0]** attribute, and the surface position connected to the **input3D[1]** attribute (Figure 11).

   ***IMPORTANT! If you click with the Left Mouse Button, you will* break *the connection from the locator! Use the Right Mouse Button to pick the correct input!*

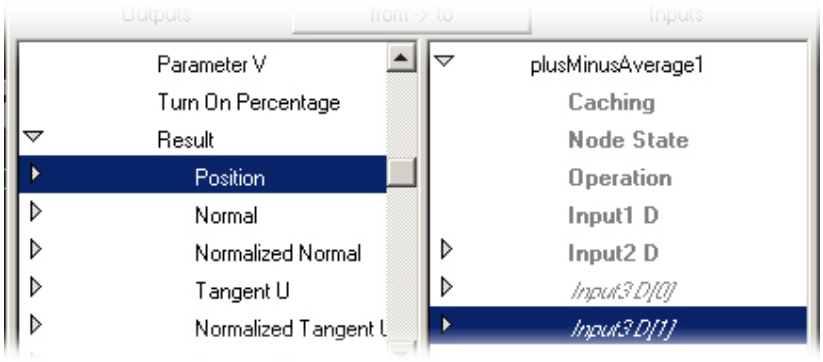


Figure 11 – Connecting the Position from the Surface to the Second Input on the plusMinusAverage Node

5. Bring up the **Attribute Editor** for the **plusMinusAverage** node. Set the **operation** to **subtract**.

Step 6: Using the VectorProduct Node

The vectorProduct node will allow us to compare the vector from the locator to the surface, and the surface normal at that point. If the vectorProduct node returns <<1, 1, 1>> then we know we're okay, the locator is on the outside of the surface. If it returns <<-1,-1,-1>> then we're inside the surface.

1. With the **Middle Mouse Button** drag the **plusMinusAverage** node on top of the **vectorProduct** node. This will bring up the connection editor.
2. Connect the **output3D** attribute of the **plusMinusAverage** node to the **input1** attribute of the **vectorProduct** (Figure 12).

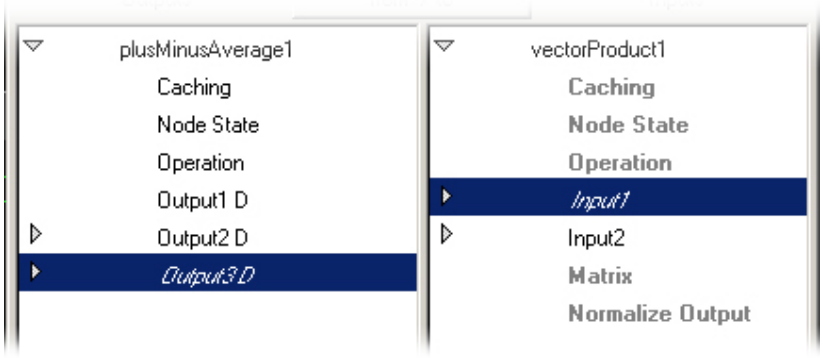


Figure 12 – Connecting the plusMinusAverage Node to the First Input of vectorProduct

3. With the **Middle Mouse Button** drag the **pointOnSurfaceInfo** node onto the **vectorProduct** node. This will update the connection editor.
4. Connect the **normal** attribute on **pointOnSurfaceInfo** to the **input2** attribute on **vectorProduct** (Figure 13).

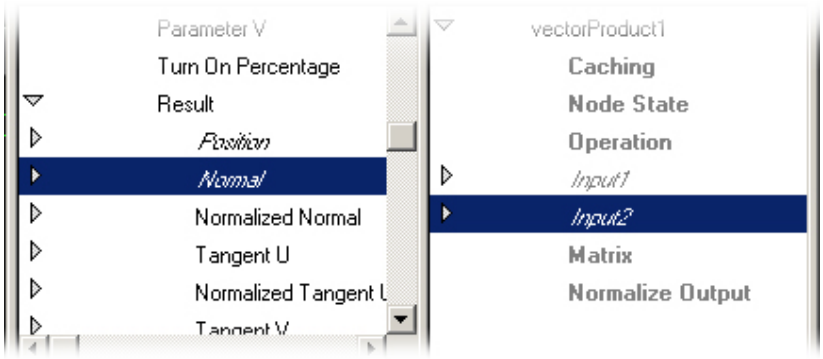


Figure 13 – Connecting the Surface Normal to the Second Input of vectorProduct

5. Bring up the **Attribute Editor** for the **vectorProduct** node.
6. Set **normalizeOutput** to on.

Step 7: Use SetDrivenKey to Change the Shader

Using the incandescenceR attribute on the shader, we can easily cause the object to glow red when the locator moves beneath the plane. To do this, we can use the outputX attribute from the vectorProduct to drive the incandescenceR attribute on the shader.

1. Bring up the **Attribute Editor** for the plane.
2. Scroll across the tabs until you reach the shader attached to the plane. Click **Select** in the bottom of your attribute editor.
3. Go **Animate→Set Driven Key→Set→Option Box** to bring up the setDrivenKey window.
4. Select the **vectorProduct** and click **Load Driver**.

5. Choose the **outputX** attribute from the top and the **incandescenceR** attribute from the bottom (Figure 14).

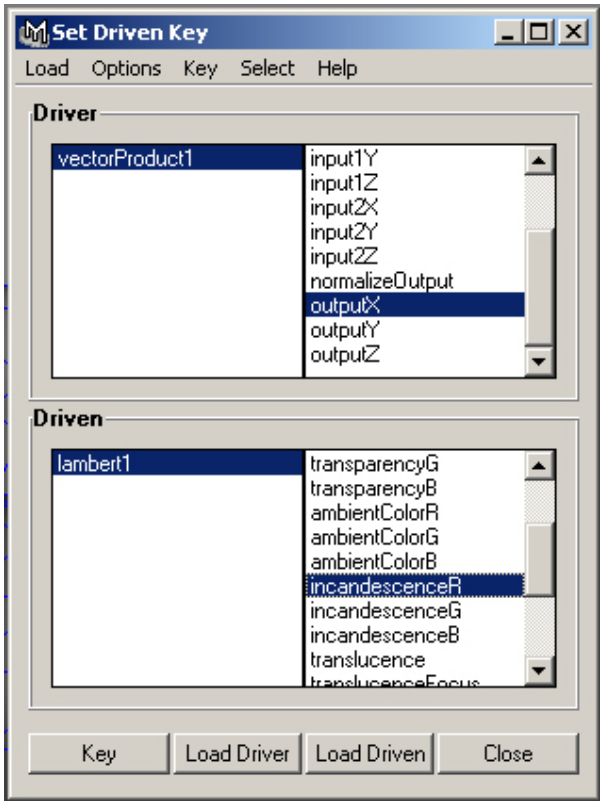


Figure 14 – SetDrivenKey Window

6. Move the **locator** so it's sitting outside the surface.
7. Click **Key**.
8. Move the **locator** so it's sitting inside the surface.
9. Click **lambert1** in the Driven side. This will select the shader. In the channelBox, set the incandescenceR to 1.
10. Click **Key.**
11. Now move the locator around. As you pass it through the plane, it should turn red to signify the locator is on the wrong side! (Figures 15-16)

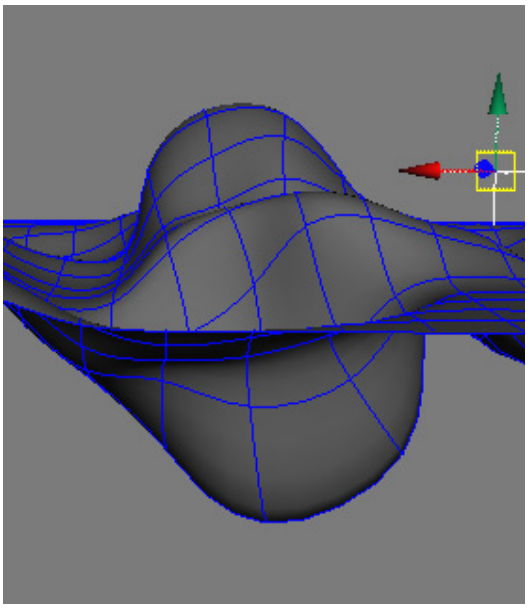


Figure 15 – Locator Outside Surface

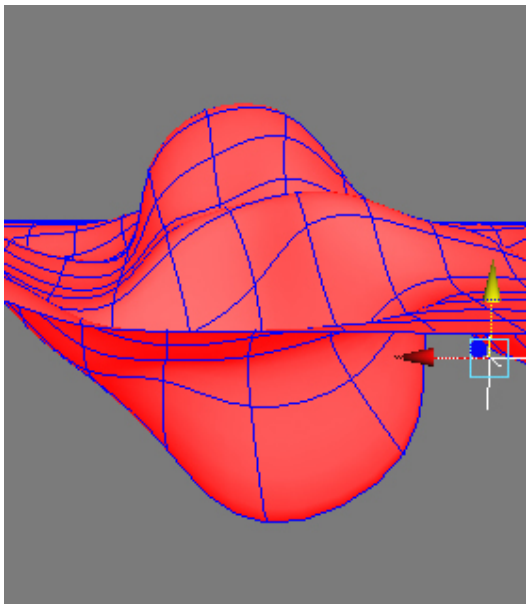
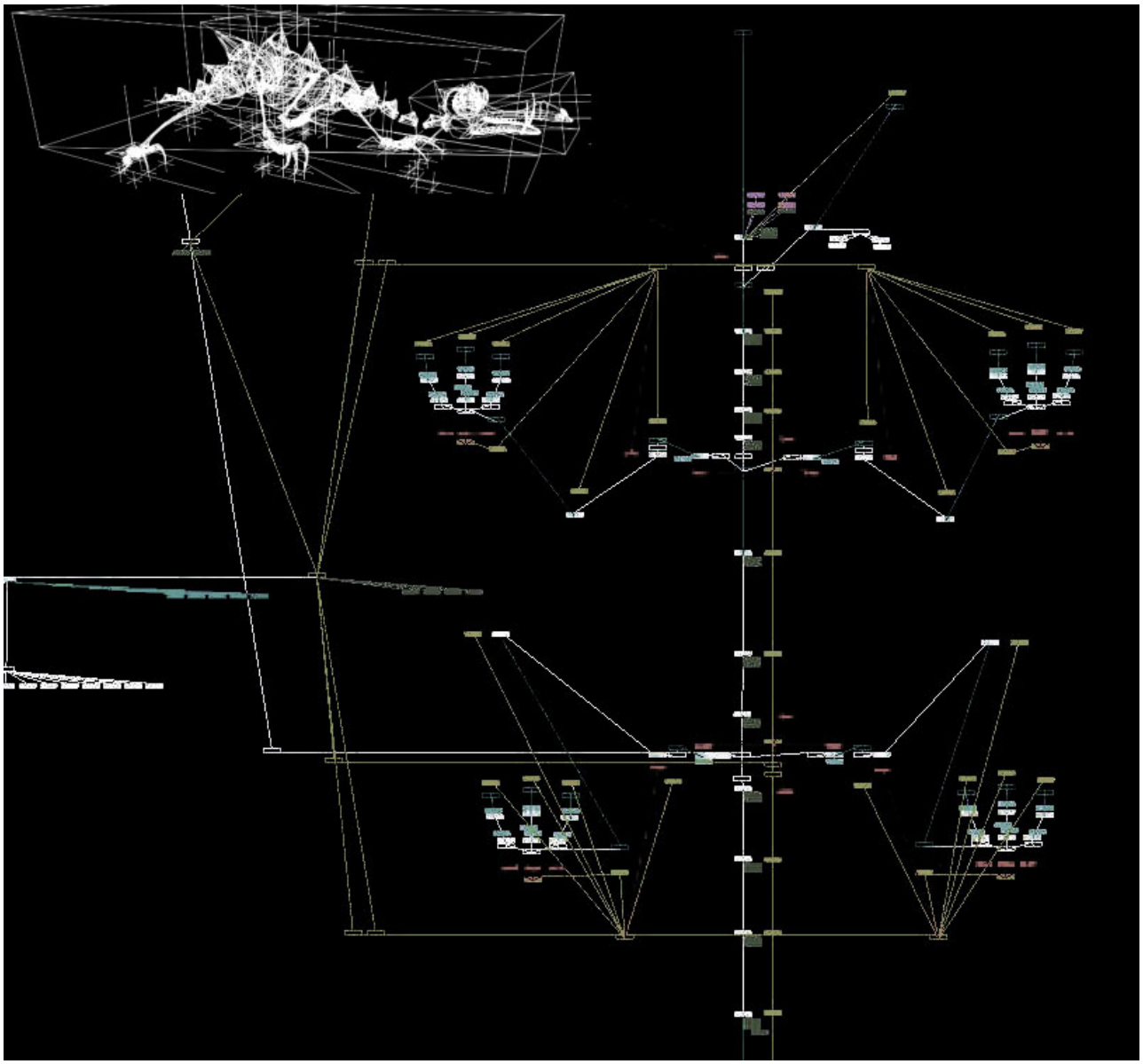


Figure 16 – Locator Inside Surface

Here are some of the places where you can use a technique like this to warn animators that they're going to break something:

- Foot/Ground interaction.
- Arm passing through body (good to ensure solid cloth solves).
- Shoulder limits (create a nurbs cone of "acceptable" behaviour. If the shoulder moves outside that cone, the shoulder turns red).
- Thigh intersection.



Building a better puppet.

by Raffaele Scaduto Mendola

Syllabus

- Introduction *(~2min)*

- The working process and character development. *(~5min)*

  Character Pipeline Strategies

- Building a framework. *(~20min)*

  Example: Rocko, the pet dragon.
  Example: A biped rig.

  Example Animations.

- Hybrid Set Up. *(~25min)*
  Pose to Pose vs. Layered Motion controls.

  Example Animations.

- New Animation tools. *(~25min)*
  SoftImage Animation curves.
  Maya Render Utility nodes.
  SoftImage Dope sheet.

  Example Animations.

- Design a character's Engine. *(~10min)*
  Putting it all together.

  Example Animations.

- Conclusion. *(~3min)*

- Q & A. *(~15min)*

Introduction

In this paper I will discuss some of the main issues associated with setting up (or rigging) a 3D characters for use in television broadcast events and film work.

With both the advent of faster CPUs and more sophisticated commercially available 3D applications, the demand for generating more complex character based projects is increasing every year.

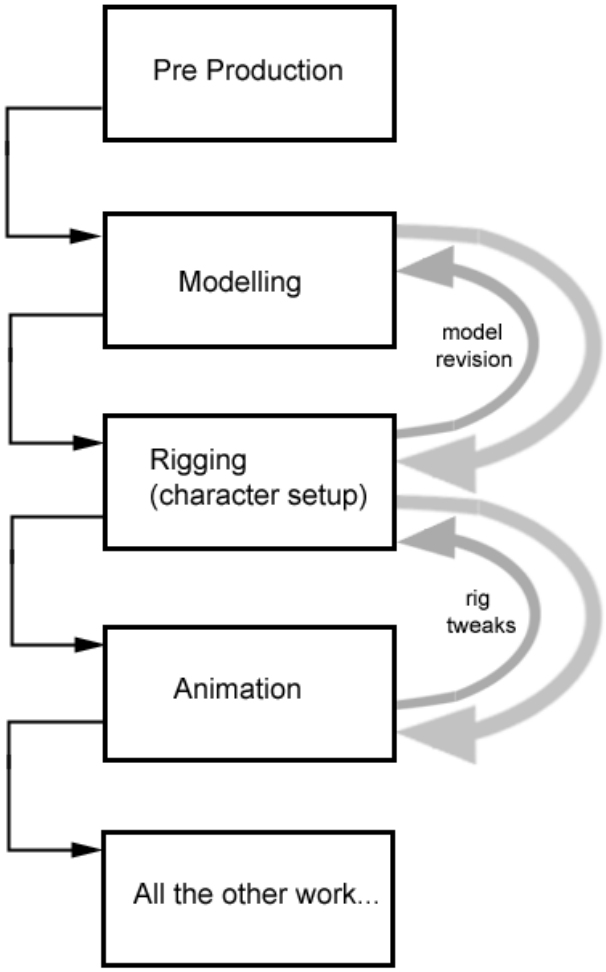
The job of a character setup artist both demands a good technical understanding of physics based simulation system, mathematic as well as usually a programming background, and on the creative side an understanding of “expressive” motion mechanics for animation work. A possible alternative description of this type of work would be as someone creating the complex mechanical framework (or engine) that drives the different pieces of geometry that make it possible to bring an inanimate 3D geometry to life.

Since it is extremely hard to quantify all the different steps involved in this process, and many different character setup artist have many different methodology, I am focusing in this lecture not so much on any specific rigging techniques but more on describing the framework concepts that help me solve some of the problems in character rigging.

Hopefully in a broader context, this course can also serve to bridge the gap between animators, character setup artists and software developers. With the advent of more sophisticated 3D commercial applications, writing more tools geared towards given animators more intuitive animation tools (without sacrificing control).

Working in parallel

In any basic “character driven” production pipeline character setup holds a very critical place. After the initial pre-production design work is done, in production the characters need to be modeled, rigged and animated.

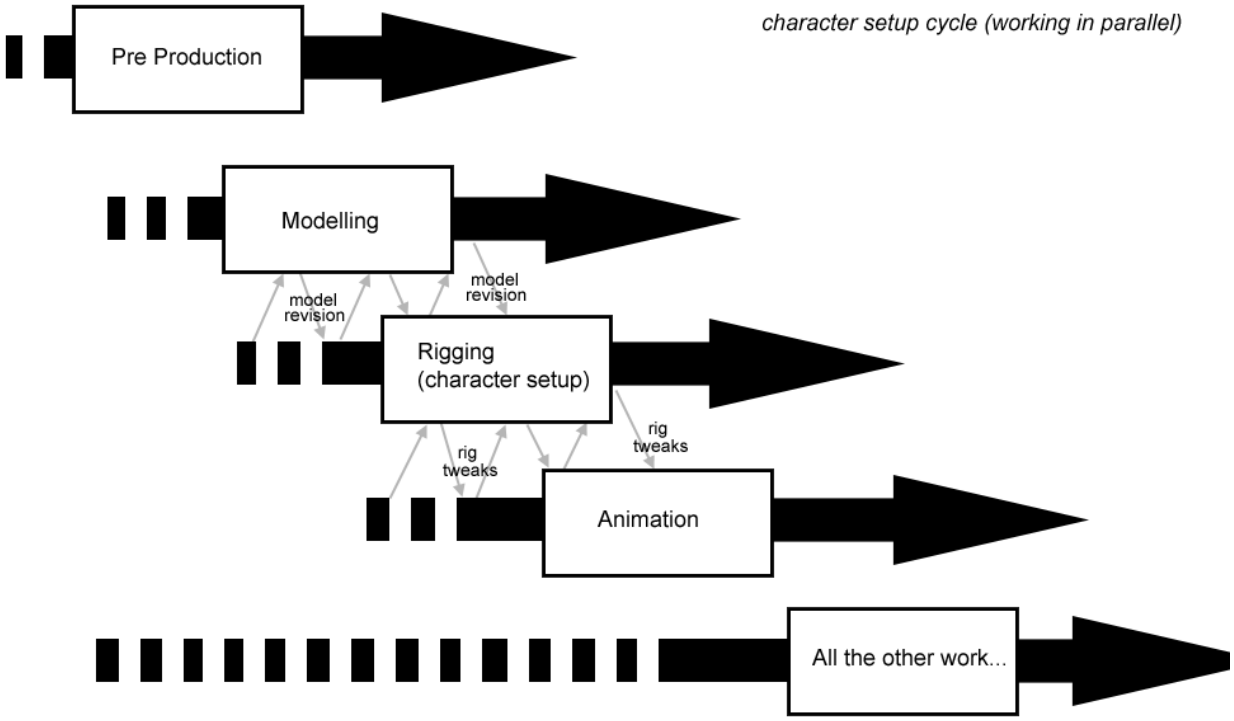


character setup cycle

Although here this simple example process seems to be sequential it never really works out to be the case. With the exceptions of simpler projects or very well planned out staged projects, you always end up having to re-model some parts of characters after you’ve already started rigging the characters (for tweaks, deformation and other unforeseen problems). As well, you always end up re-working your rigs after animators have already started animation, tweaking the behavior of these controls.

And there are many other technical production pipelines issues falling outside the scope of this course, which will make this sequential (or linear) process difficult and costly to work with.

In order to alleviate some of these inherit technical difficulties of production you end up trying to parallelize the tasks to work as much in possible in tandem.



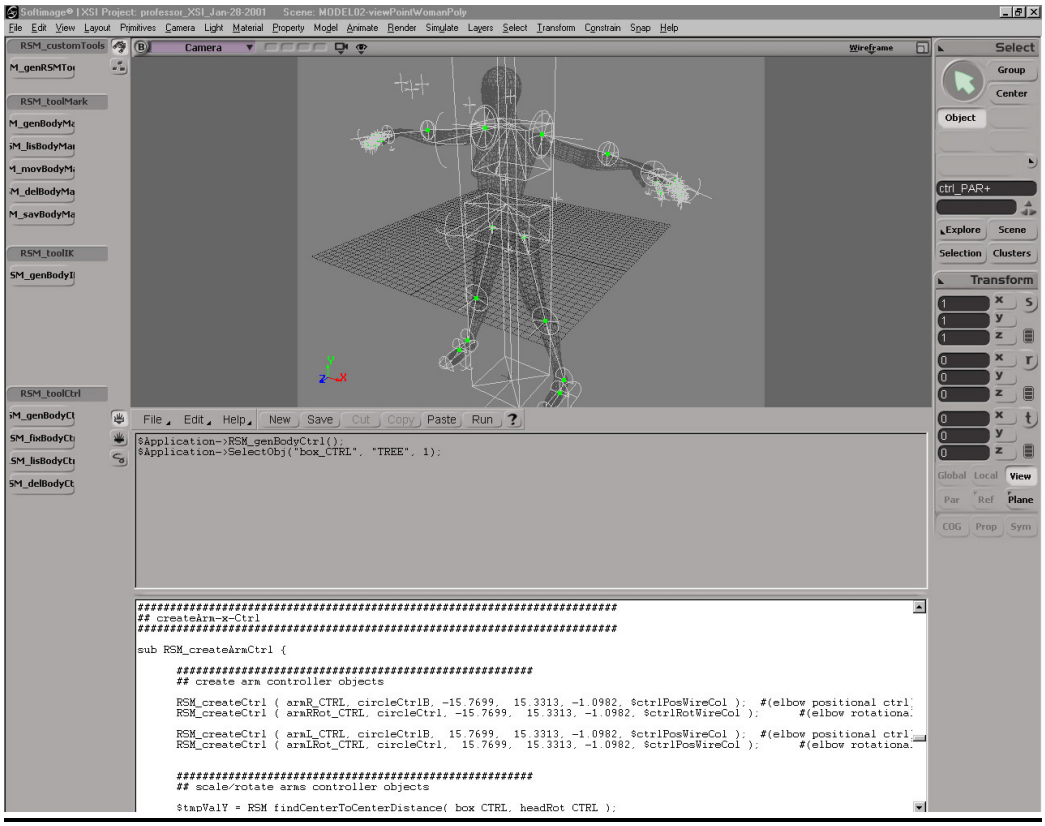
Two examples of parallelizing your work methodology are:

A traditional oil painter
First you pencil in the main shapes that make you your composition.
Then you start fixing the main colors.
Then you paint in the details.
Because oil painting can take a while to dry you can still re-visit different parts of your canvas and make changes.
So you can work in layers, making multiple passes at same areas of your canvas.

A super computer simulation.
As different simulation systems usually require many calculations
Some of which are dependant on previous calculations, and others that are independent.
With some clever programming some of these calculations steps can be distributed to several processors to be computed at the same time, cutting down on the amount of time needed.

With today's available 3D commercial applications, it is now much easier to build repetitive procedure based on assembly of characters, working on characters before the modeling is finished (giving you a chance to tweak the geometry). And animation can be started before committing final rigging (making animation/behavior changes possible).

The assembly procedure can be automated by writing multiple scripts that can be run to rebuild either parts of, an entire character, or multiple version of similar character (biped humans being the most common forms).



Example rig assembly environment

The basic procedure usually involves:

1. Importing and cleaning (by correctly parenting) the geometry

2. Setting up joints and associated controls for primary and secondary animations.

3. Skinning and/or apply skin deformation tools, to correctly deform the geometry.

4. Applying additional procedure, like facial blend shapes, or clothe simulation algorithms.

5. Prepping the character framework to conform to naming any conventions, and/or associating animation reference motion clips.

From this basic step-by-step breakdown, more complex custom character assembly pipelines can be designed.

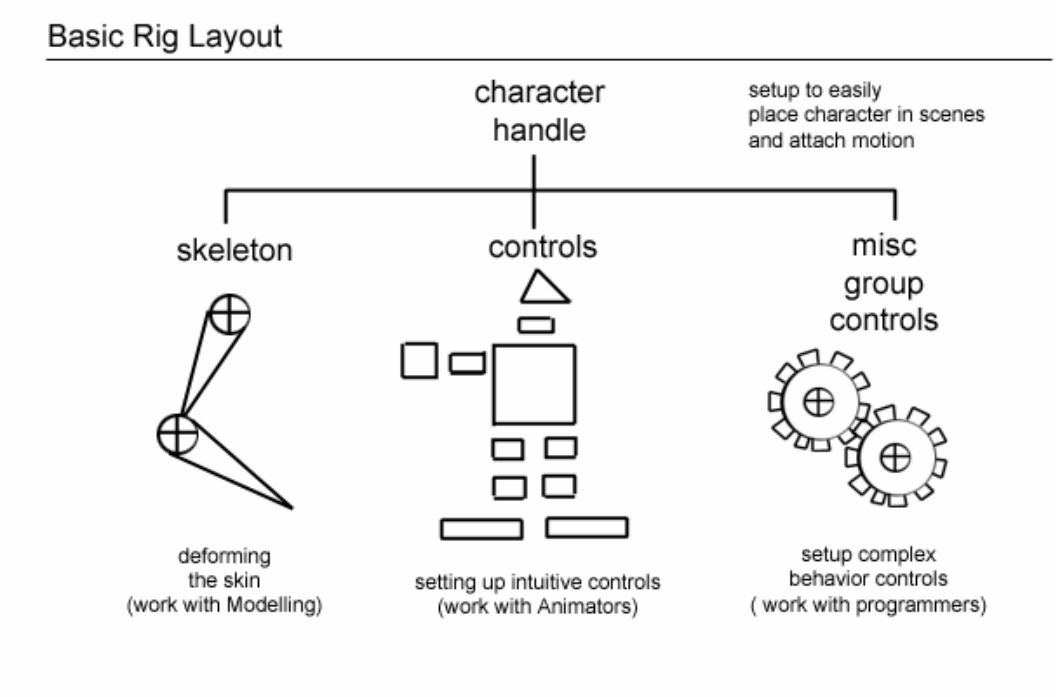
You don't necessarily know what will work.

Another big problem in rigging characters (especially non-human, non-biped) is that you don't necessarily know for sure which procedures will work for specific projects. Since complex characters usually involve rigging several different types of control systems, having a good frame work (or organized scheme) where you can combine these different procedures and eventually re-combine different parts without breaking other parts, or re-purposing multiple parts is important to your workflow.

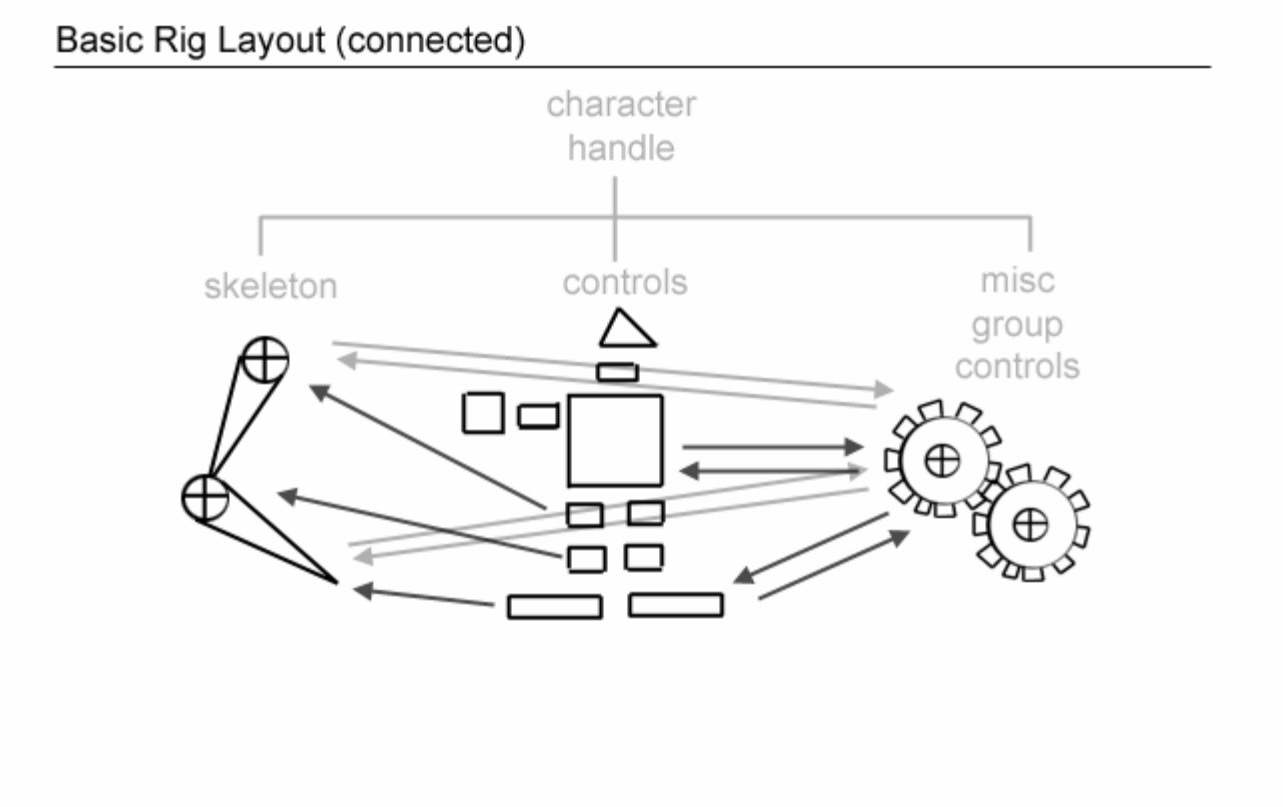
My first inspiration of this concept came from using 2D compositing software. By building everything in layers, you can breakdown multiple complex compositing operation into more manageable smaller sub-composite operations, which can then be tweak on an individual basis.

In the realm of 3D character rigging the same idea can be used with slightly different goal. By breaking up the rigs in "layers" interconnected between each other via mathematical expressions, various constraints and other tools, you gain flexibility in setting up your character. All the while keeping it possible to manage and change individual sub-parts of the rigs without having to rework the entire assembly.

So developing a good framework is an essential part to building and maintaining many 3D digital puppets.



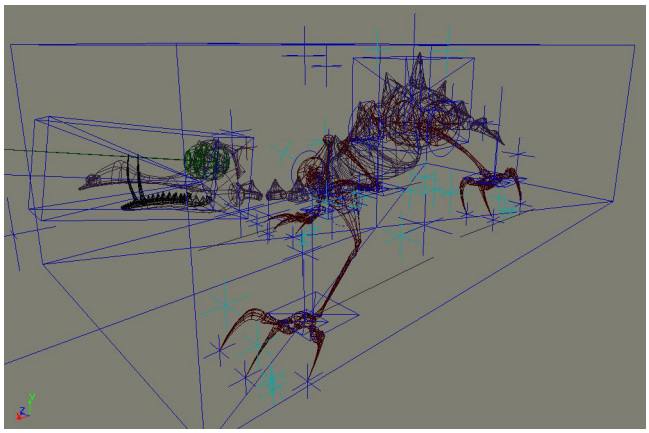
By systematically organizing your character rigs in organized sub-tree you can more easily breakup different sections of the rig and develop the glue tools (or interconnection) that enable all of the pieces to work together.



Another way to think of this is by using programmable visual aids (like Alias Maya's Hypergraph and Dependency graph or SoftImage's
Schematic view port).

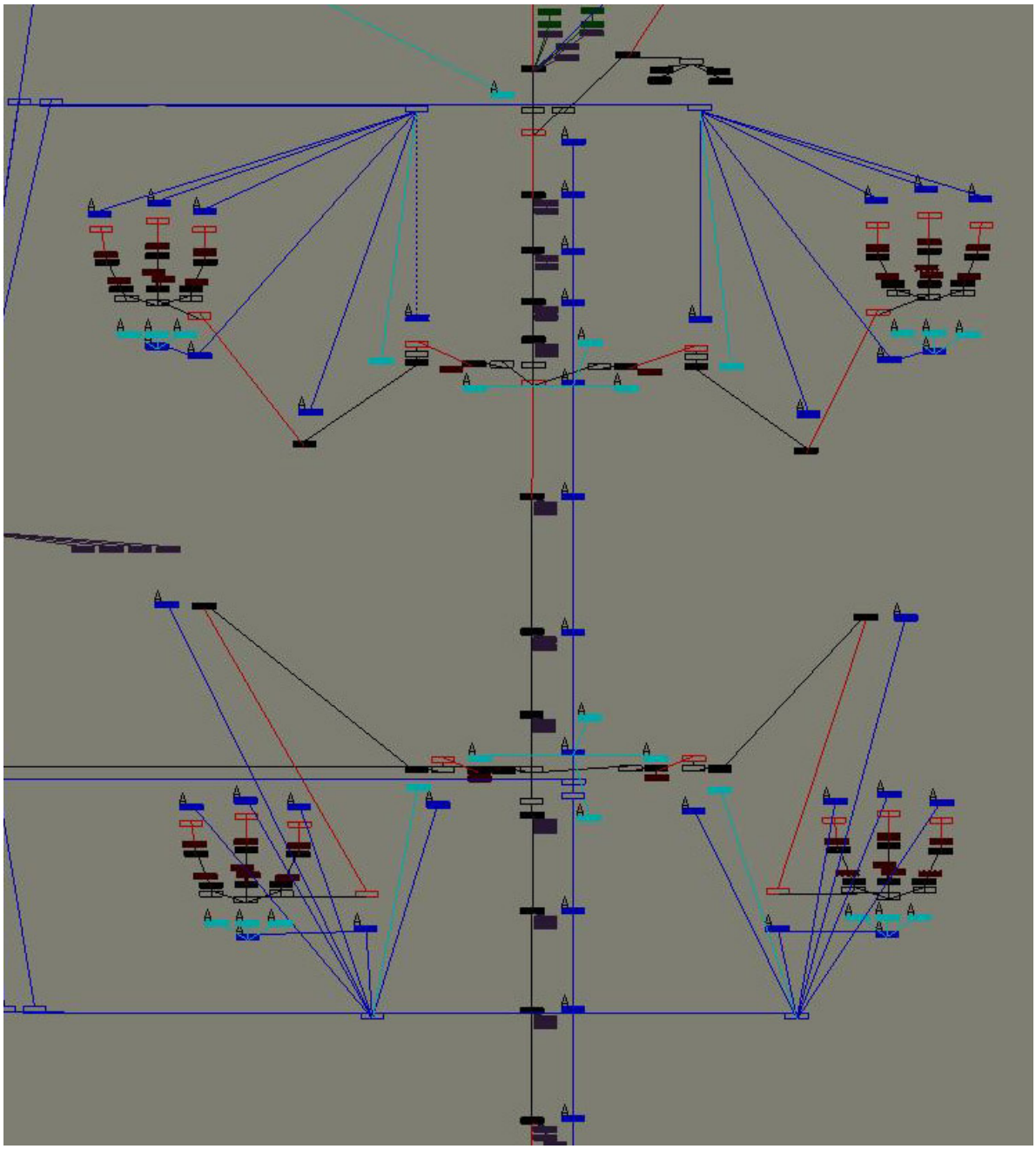
Unlike more traditional "hierarchal trees" views, these tools enable you to visual organize more complex rigs (with hierarchal and other relationships, constrains, expressions, set driven keys…) into more logical visual patterns.

This approach was originally used for animating two complex characters in a 3D CG short.

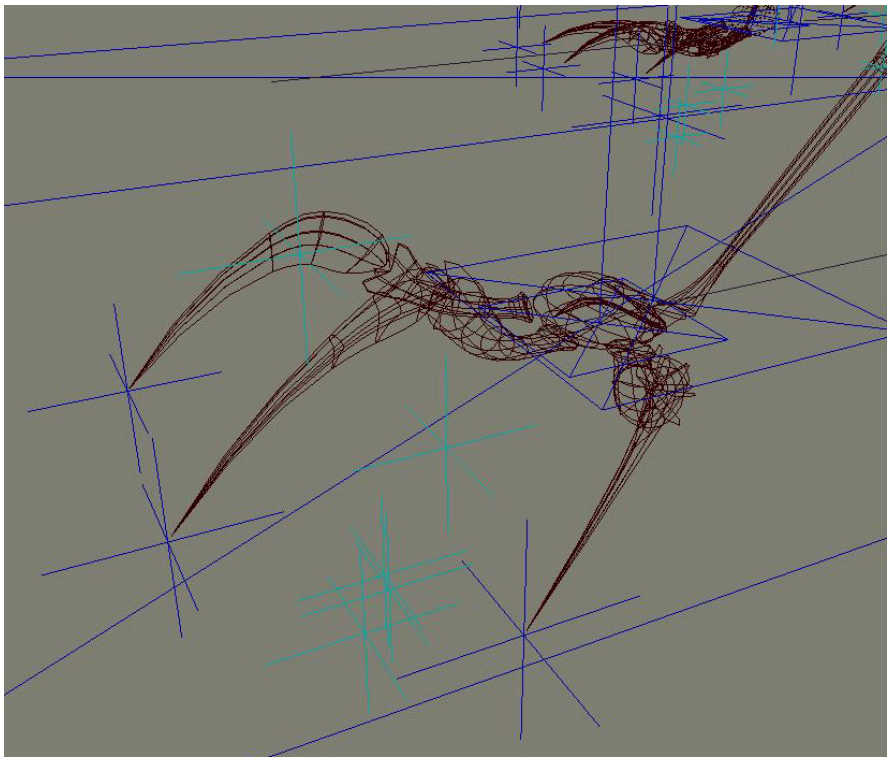


Here a four-legged pet dragon with tail was rigged to provide for the maximum range of movements. The control system setup was design for pose-to-pose motion with a maximum of the controls design to be translated.

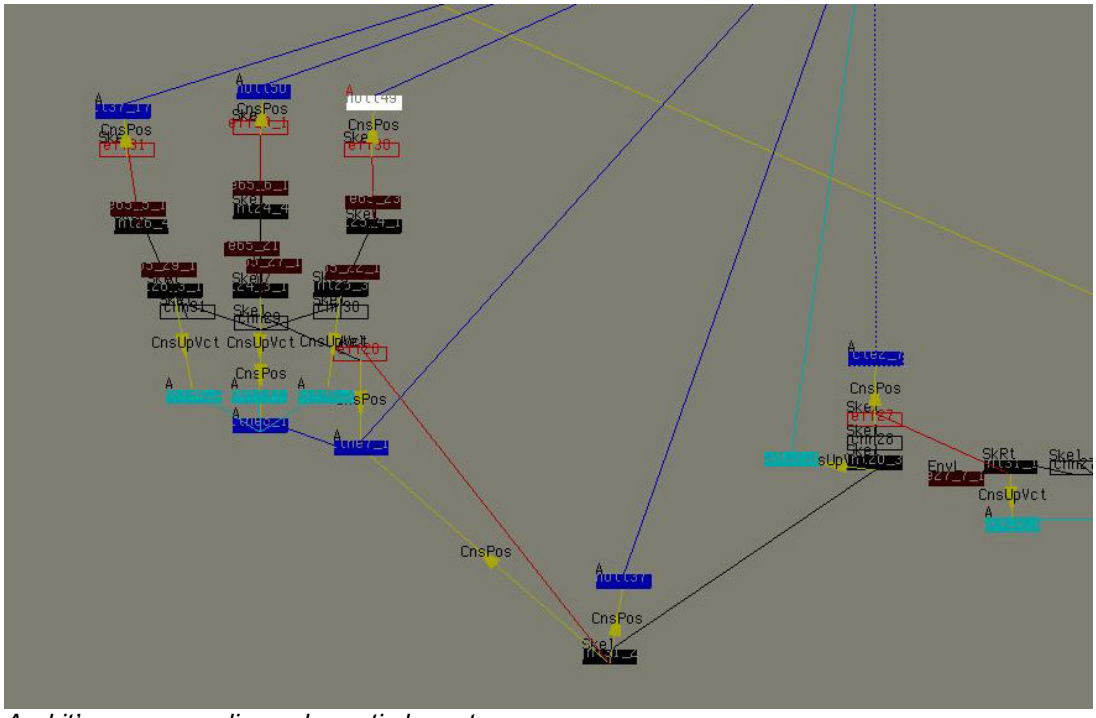
To help visualize and tweak the mechanics, the rig was laid out in the schematic to loosely follow the shape of the character.



By visually setting up the rig to match the various parts of the character makes it much more easy to debug and maintain specific components of the rig.



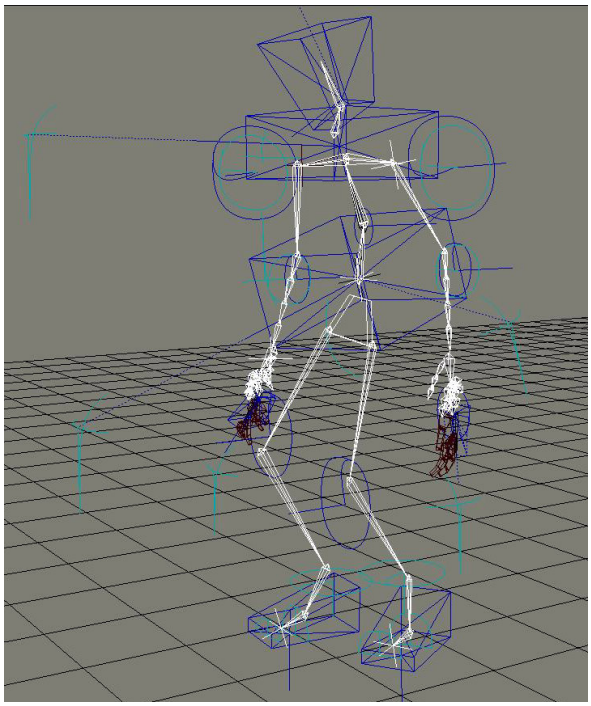
Example hand



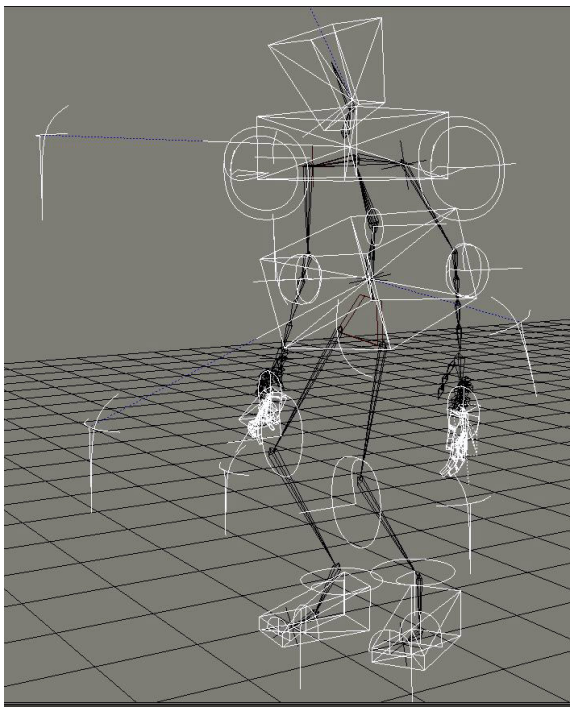
And it's corresponding schematic layout.

The yellow connections show the various constraints and expression connections.

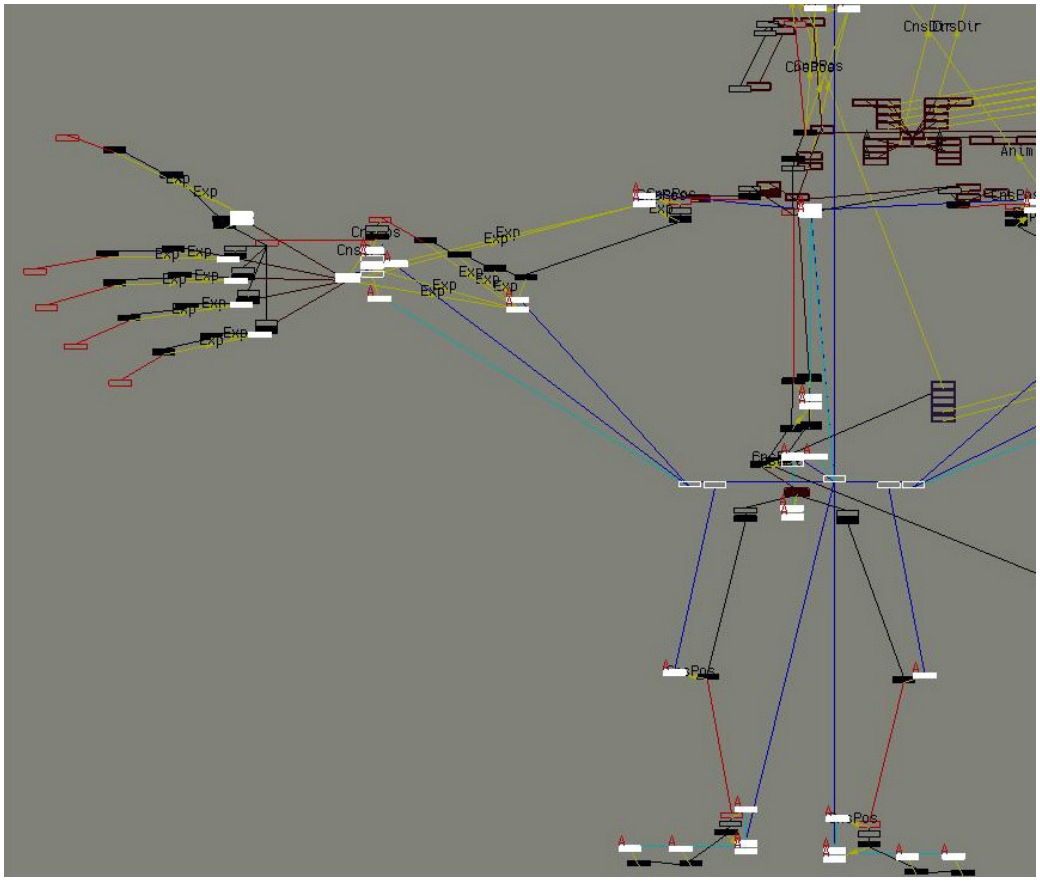
A similar approach also works on biped/human forms.



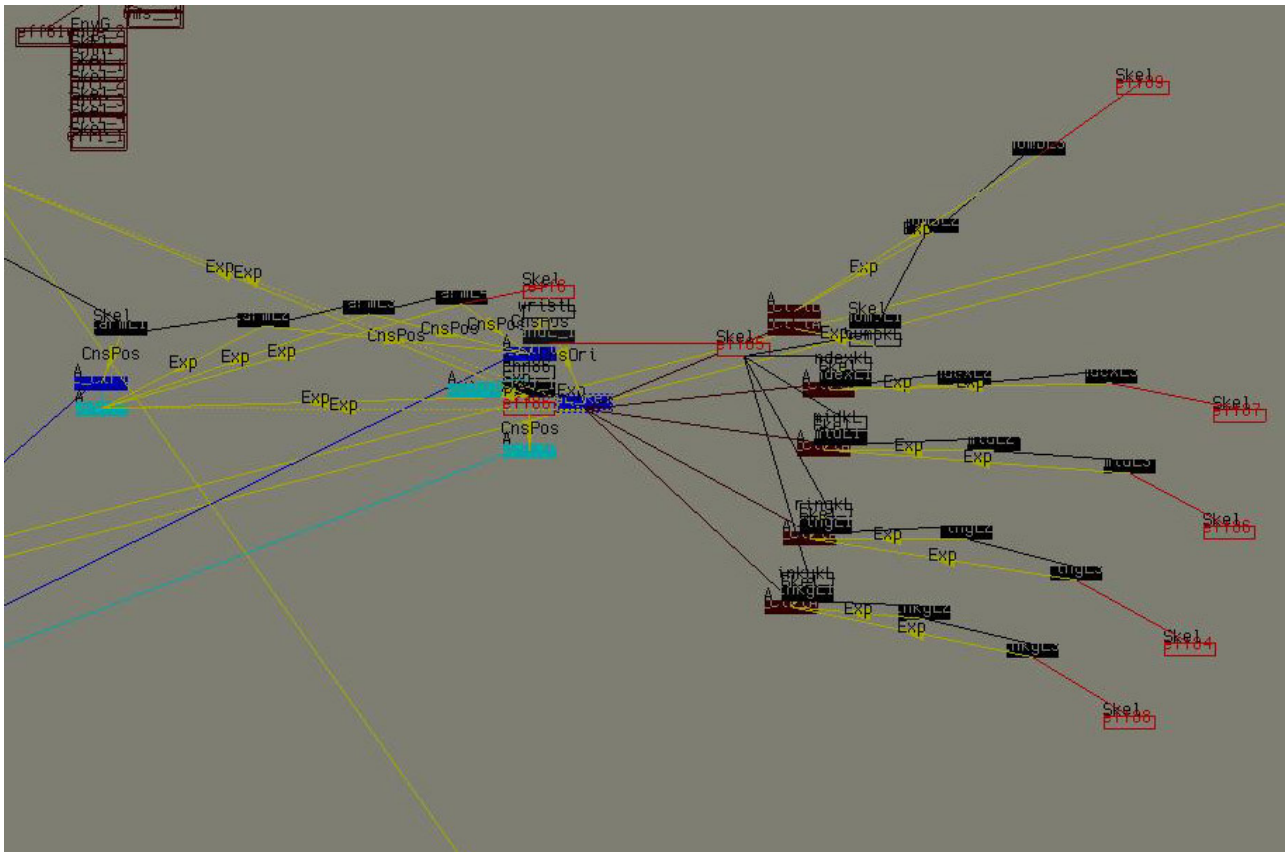
Skeleton group

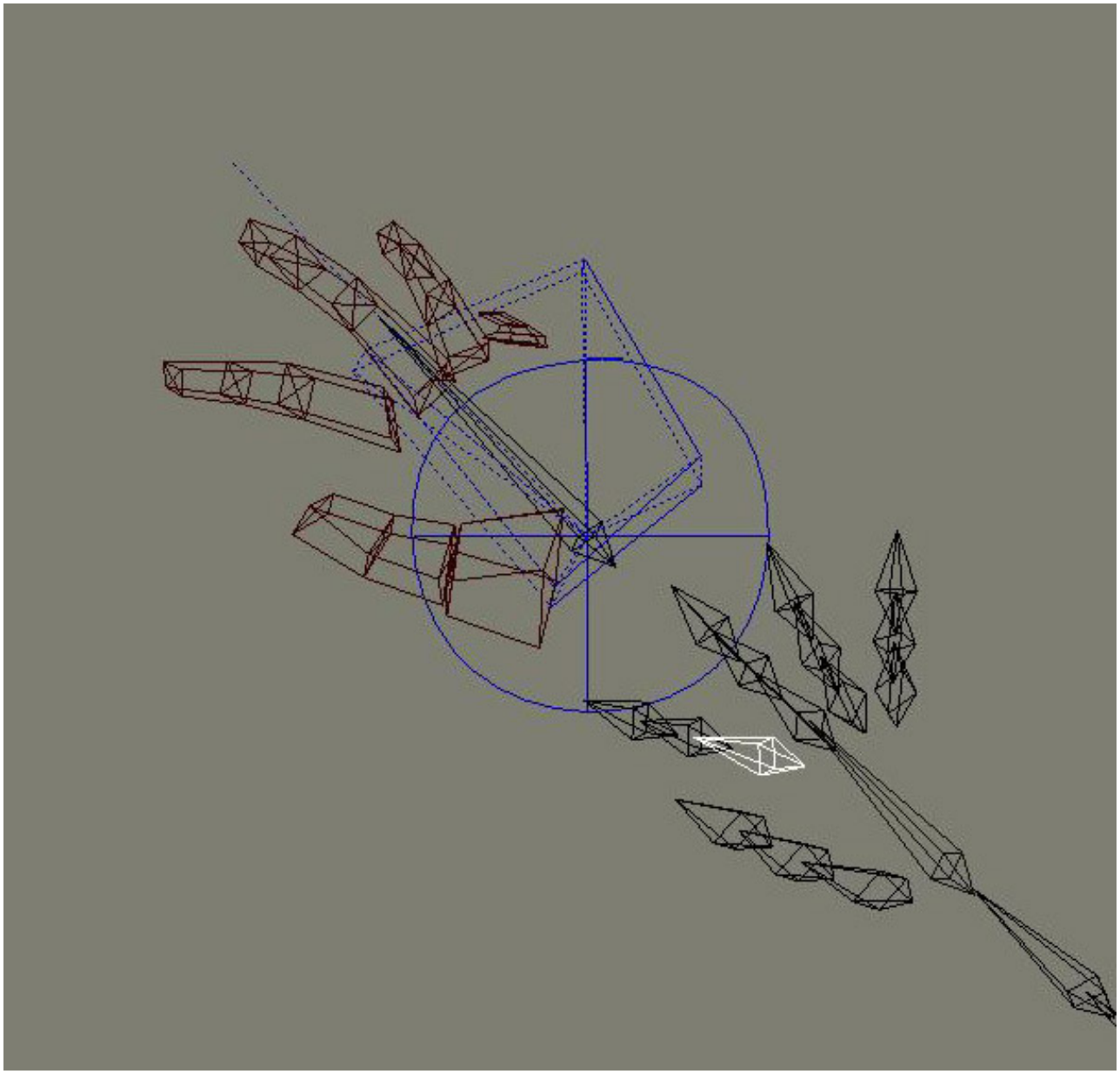


Control group



And again by laying out the control system and skeletal system in a logical visual pattern makes it easier for us to develop connections (this behavior) for specific parts of the rig.





Here the wire control hand acts as a remote hand for the skeleton hand.

Setting up for Pose to Pose versus Layered Animation.

Alternatively, this could also be considered as the Inverse Kinematics vs. Forward Kinematics debate.

In a traditional Forward Kinematics style rigs, controls are parented to each other simulating a more anatomically correct mechanic logic.
In this scheme the child controls follows the translation, rotation and scale of the parent control.

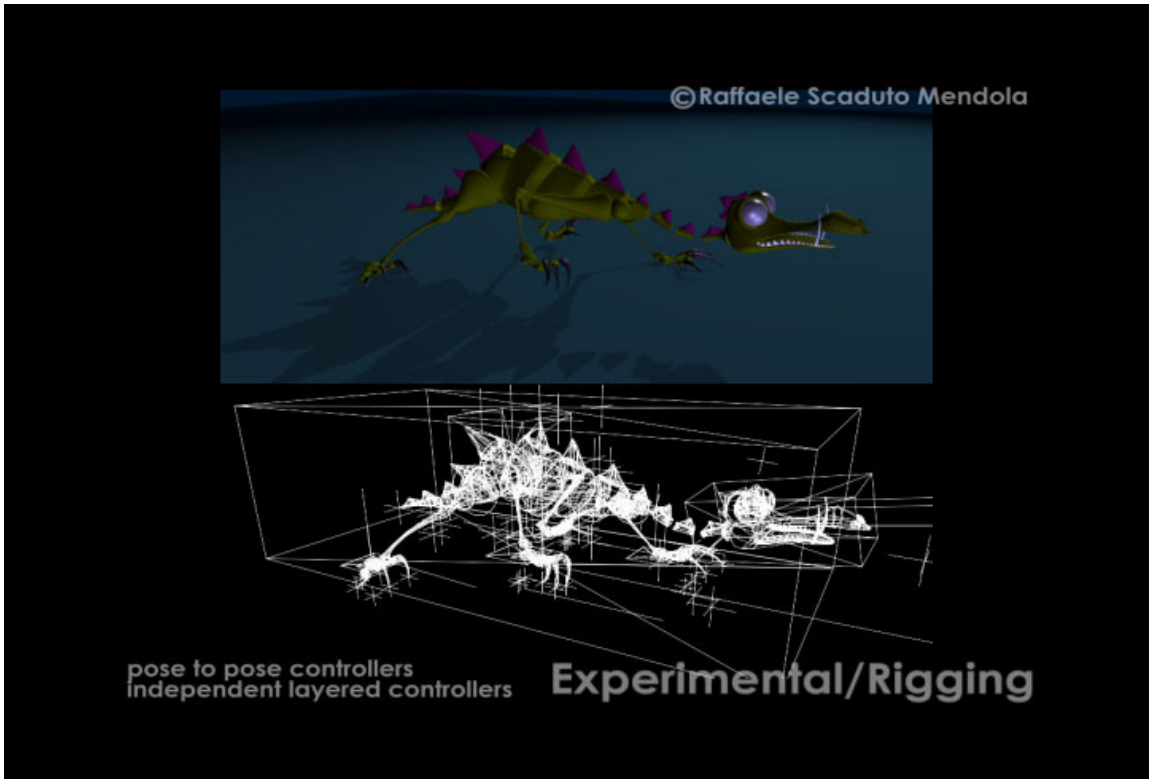
Alternatively, controls can also be setup to be independent of each other's translation, rotation and scale. Each control has to be separately "posed" in order to get character properly positioned.

Each method has both its advantages and disadvantages.

Layered Controls (Forward Kinematics)	**Pose to Pose Controls (Inverse Kinematics)**
Advantages: Fairly straightforward controls. You first animate the parent, and work your way down. You get exact control over the different parts of the rigs (one to one correlation). When you rotate one control the associated skeleton part rotates by the same amount. You can animate "arcs" type of motion more easily.	Advantages: Even easier to animate. Controls are usually setup to be translation only. Translation based controls don't have guimble lock problems. You don't have to think so technically about the rig's bio-mechanics when animating; Can be considered more intuitive. More of a push and pull work. If the children don't follow the parent, you can tweak individual controls without having to re-animate every descending child.
Disadvantages: Rotation type animation can lead to guimble lock problems. More current version of 3D application now provides Quaternion Mathematic solvers to help (defining and extra fourth axis). You usually have to have a much closer	Disadvantages: Usually involves more elaborate rigs, without necessarily one to one correlation between controls and joints. With limited experience, motion can be more floating. As translation based systems usually require more in between key frames

technical bio-mechanic understanding of the rig and associated controls. If you need to re-animate the parent, you will most likely have to re-animate all descending children controls. Makes revisions more time consuming.	to arc type motion.

With some clever rigging procedures coupled with a flexible layered rig frame work, can actually help to implement hybrid rigs using controls that can act either as traditional layered (parented) behavior or pose to pose behavior.



These hybrid systems usually involve either building multiple control system which you can then either blend in between, or clever tracking of both relative (to the parent) coordinate space and world coordinate space, building the capability to switch at any time. Although this second technique is still not widely used, it could be very promising as it proves to be easier to implement as a simpler modular component of a character rig.

New animation tools

Hard to define and too long to enumerate, high-end commercial 3D application contain many different interactive animations tools.
In recent years, more sophisticated tools designed to be used for animation and alternatively, as part of automated complex rig behaviors have become available.

One interesting development is SoftImage XSI's mixer tools.
Initially developed to as a means to manage, assemble and edit multiple clips of animation together, this tool can also be used to pre-program complex behaviors in character rigs.

This kind of capability already partially possible by using set-driven key, is augmented with a mixer like tool, as now entire sets (or groups) of animation curves can no be treated as one "modular" element to be implemented in rigs. Making complex behaviors much more manageable.

Although, too broad of a topic to cover, this type of tool is helping design and simulated complex crowd animations.

Even more common tools can also be used to non-traditional forms, as part of character rigs. Tools designed originally to support directly animation work, can themselves be used in part of mathematic based algorithms, helping calculate values much faster than traditional expression (which have to be parsed, then evaluated).

Examples of these binary tools include,

SoftImage Animation curves which can be used as projected custom sine waves.

Maya Render Utility nodes that let connect values to be multiplied added or blended together.

SoftImage Dope sheet, which can alternatively be used as an activation switch to calculate or disable calculations.

All of these tools were originally designed to serve a specific purpose, but having 3D commercial application with a flexible modular architecture, these tools can be used in other ways.
In the context of building procedure these 3D applications themselves can be considered as 3D development engines where any of the available tools (or widgets) can be combined to simulate complex mechanical behaviors.

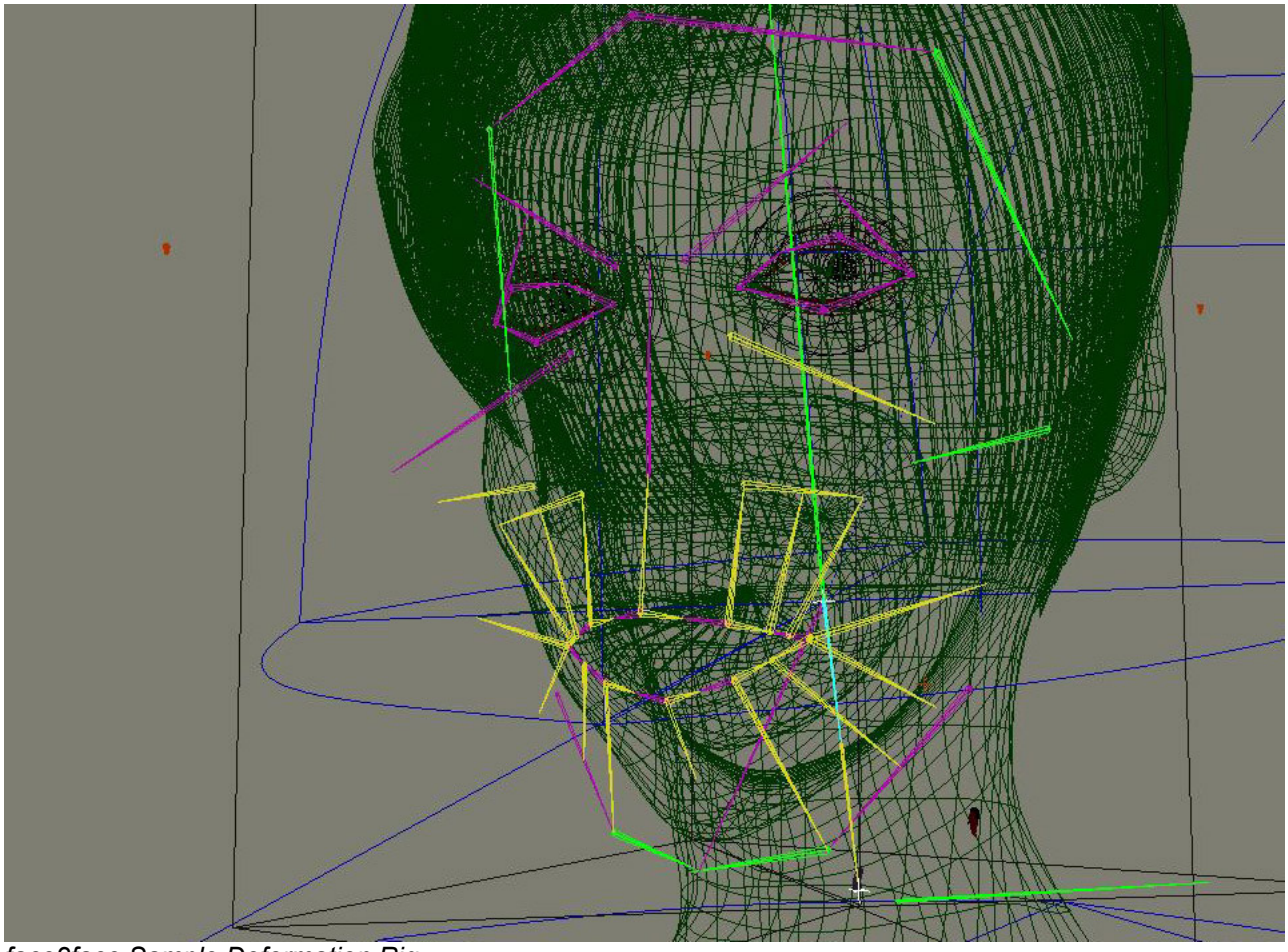
3D Commercial Applications and Game Engines

As this topic is fairly broad, it is my intent to cover only the basic practical concepts of using 3D real-time processing engine in high-end character setup.

With CPU speed now above 2Ghz, and dual processor PCs (with large amounts of RAM) commonly available, it is now possible to setup complex character rigs to work close to real-time.

Following a similar metaphor that 3D game engines developers use, it is possible to design character rigs to act as its own encapsulated custom 3D “character engine”.

A simple example of the use of this concept came from some development work on facial muscle systems.



face2face Sample Deformation Rig

Originally designed for rapidly applying large amounts of lip sync facial motion in a batch process mode on low to mid-rez characters. I quickly found that on faster CPUs, the rig itself could be animated in near “real-time” mode. As this rig was originally designed with

commercial only tools, I also found that by optimizing some of the sub-systems I could dramatically increase the speed of real-time calculation processing.

One particular example of this is with using expression versus a combination of “binary” 3D tools to automatically scale joints.
The formula used is

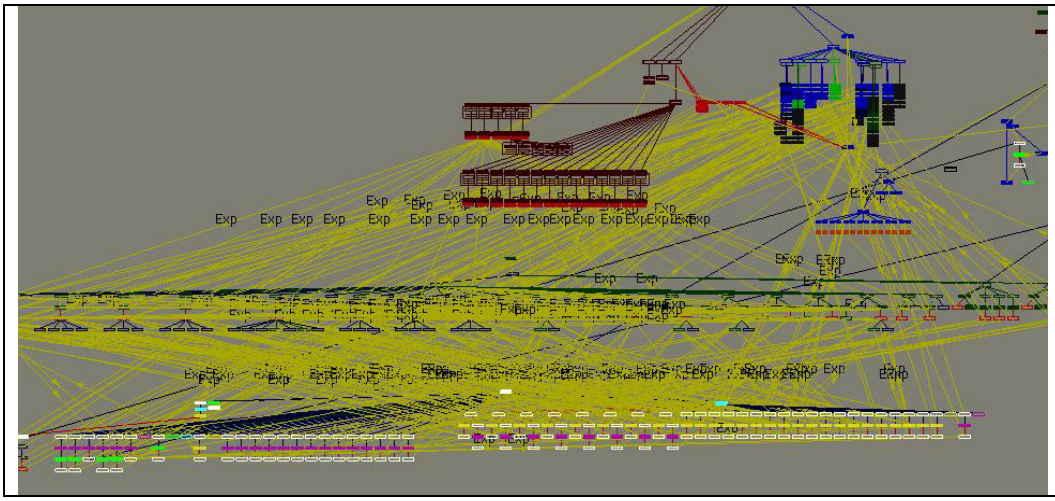
```
Joint.scaleX = current joint length / initial joint length
```

Where:

The current joint length is the absolute (in world Space units, i.e. not dependant on hierarchy structure) length of the joint being calculated at all times.

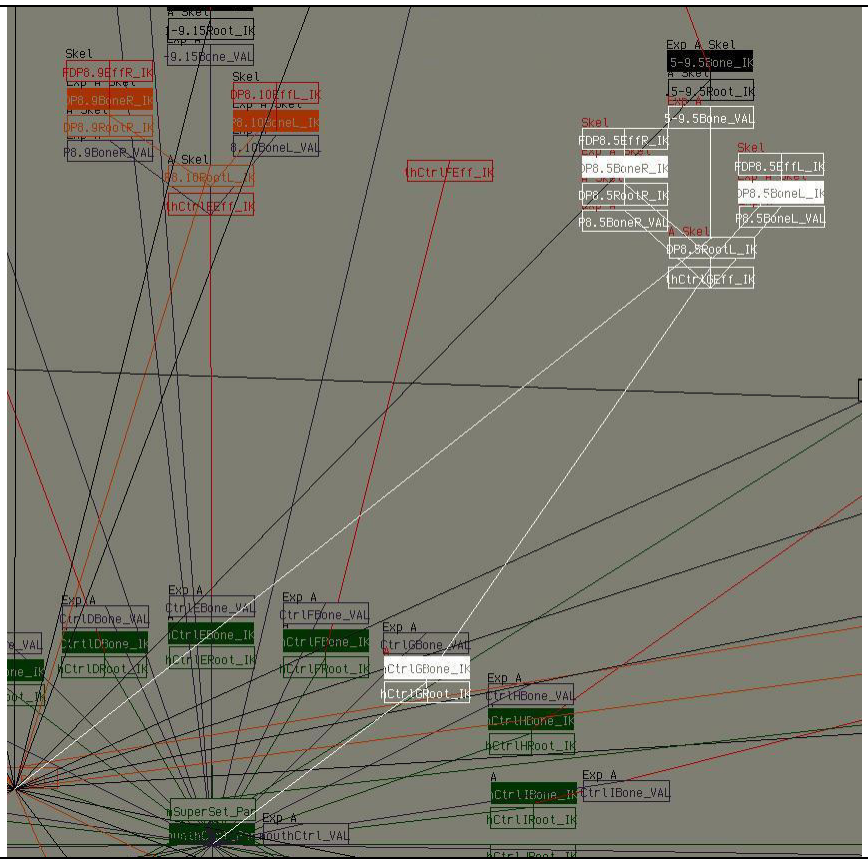
The initial joint length is the absolute length of the joint only calculated when the joint is created.

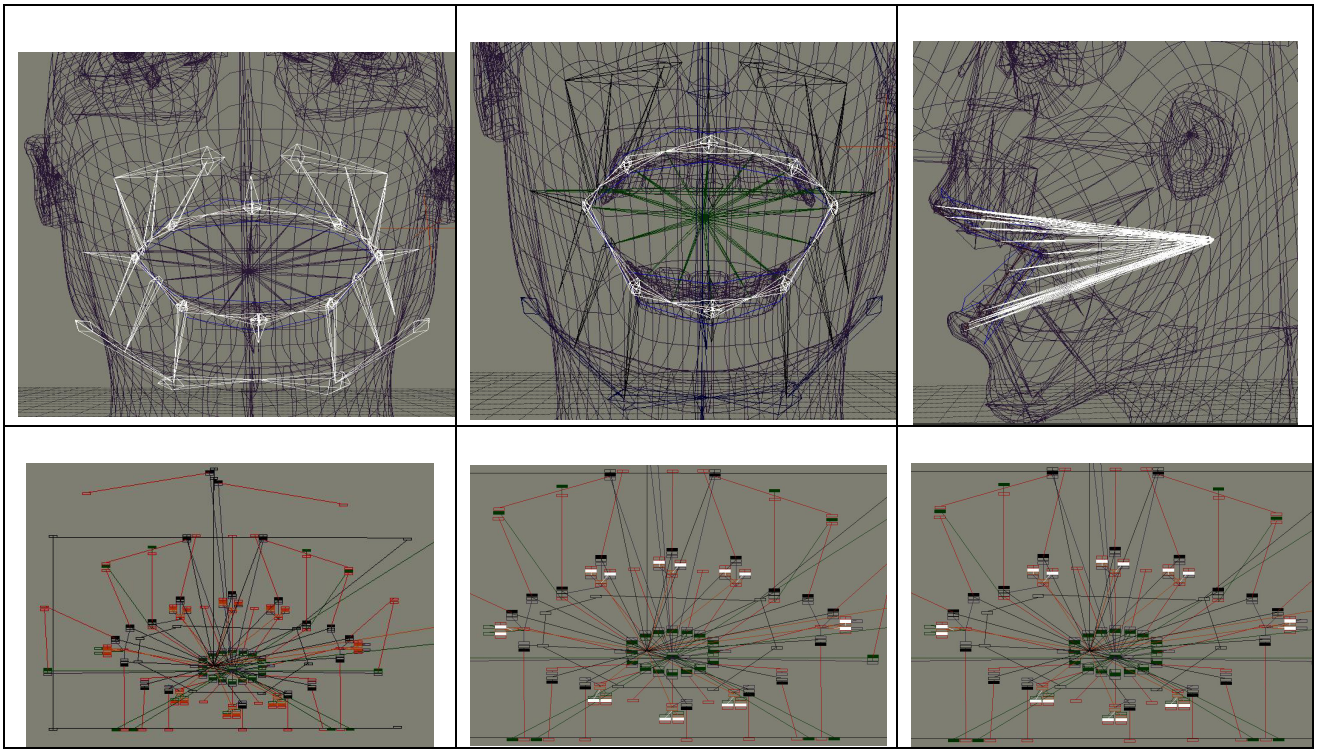
When using an expression, the 3D software has to first evaluate (or parse) that expression, and then calculate the result (i.e.: two processing steps). By using a binary operator (e.g.: a Maya render utility multiply/divide node) you can directly calculate the resulting scale without having to parse an expression. So you save one processing step. If your rig is very complex with many scalable joints, those processing steps add up. Other similar math like operation can be evaluated by using “binary” tools to calculate the algorithms.



One draw back is the amount of inter-object connection can increase very quickly.

But by logically laying these objects in clusters according to their interconnected relations, and building modular assembly script/functions the rig can stay manageable.





Node based system vs. Objects based systems.
With the new crop of commercial software, and the introduction of procedure-based techniques, two separate types of 3D applications have evolved.

Node Based	**Object Based**
Alias|Wavefront's Maya system is node based system. Where you can build up complex procedure by interconnecting various simple nodes.	SoftImage's XSI is an Object based system. Where individual objects include more custom information. (eg: relative_coordinate_system, world_coordinate_system, constraint_information,…).
Advantages: This system is very modular, given you the capability to interconnect nodes in many different ways.	Advantages: Each Object encapsulates more accessible information (by default), and lets you build more complex rigs with fewer elements, making it easier for your 3D engine to manage complexity.
Disadvantage: With more complex rigs, you end up having to setup more complex node networks, having the 3D application keep track of many more nodes.	Disadvantages: Since each object carries more information than might be needed, you may be evaluating more data then necessary for simple rigs.

One thing still partially lacking in many 3D commercial applications is the simple ability to use dual processors for real-time evaluation of rigs. Many renders are now capable of threading complex rendering tasks to two or more processors, yet "3D real-time engines" do not inheritably offer this capability.

Although there are practical and logistical considerations which make this type of tasks difficult to implement, having the capability to run multiple characters assigning there real-time calculation to be processed by a specific processor (tagging a processor id number) or assigning priority levels to different component groups of a rig could dramatically streamline the real-time performance of handling complex rigs.

Conclusion

With renewed interest in 3D character work in film and television, the challenges of rigging complex character can be overwhelming. By developing a good framework on which you can assemble and attach individual mechanical sub-systems that don't break the rigs, you can make any complex rig more manageable.
This approach makes it also possible to upgrade individual sub-systems and re-purpose procedures, saving time and money.

As this is still very much a growing field with no set methodology to rig characters, there are still great needs for new types or real-time tools to be implemented.
Hopefully this lecture short one and half lecture can help both animators and producer understand the underlying engineered work that goes into building these 3D character engines, as well as providing software developers a better understanding of the unique process involved in character building and the tools that are still needed.

For more information and expanded examples, please consult my website **www.turbolinea.com**.

References.

[1] Richard Williams, The Animator's Survival Kit, Faber and Faber, 2001.

[2] Frank Thomas and Ollie Johnston, The Illusion of Life, Hyperion, 1981.

[3] Fritz Schider, An Atlas of Anatomy for Artists, Dover Publication, 1947.

[4] Dr. Paul Richer - Robert Beverly Hale, Artistic Anatomy, Watson-Guptill Publications, 1971.

[5] W. Ellenberger – H. Dittrich – H. Baum, An Atlas of Animal Anatomy for Artists, Dover Publications, 1949.

[6] SoftImage|3D, SoftImage XSI, SoftImage Inc.

[7] Maya Complete - Maya Unlimited, Alias|Wavefront.

Improving Skin Deformations – An Artist's Perspective.

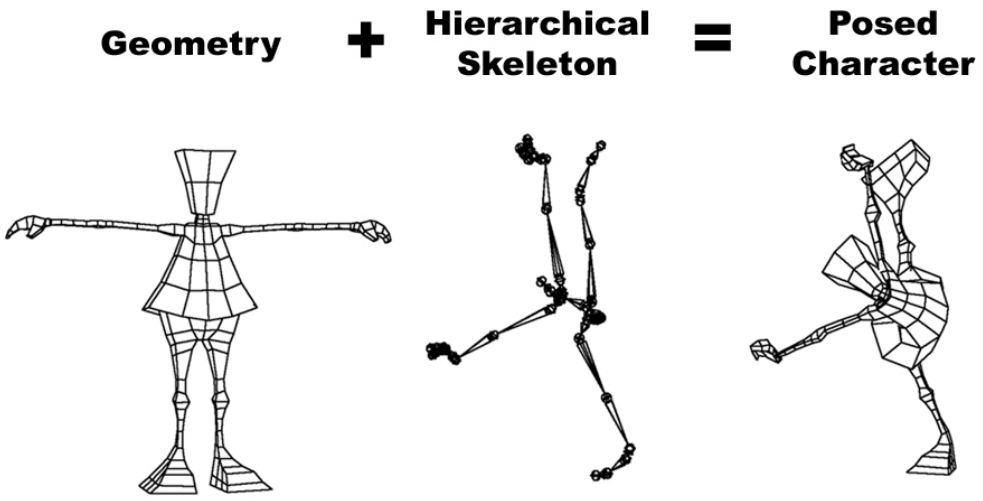
Yaron Canetti

yaron@summerbreezethefilm.com

Table of Content

I. Introduction.

The simplest form of a deformed character consists of two layers, skin geometry and hierarchical joint structure. To deform the skin we use the joints. In most cases, regardless of the methods used to control them, the joints rotate.



The connection between the joints and the skin is done by binding skin points to specific joints, usually based on the distance between points and joints. A significant improvement is the ability to assign more then one joint influence to a single skin point. In this case the points translate will be the results of blending between the rotations of the different joints. For additional control the user can set manually the weights of each point.

While the structure described above is sufficient for the purpose of posing a character, in many cases it results in poor deformations. Figure 1_1 shows the typical problem of loss of volume in creases areas, in this case at the elbow.

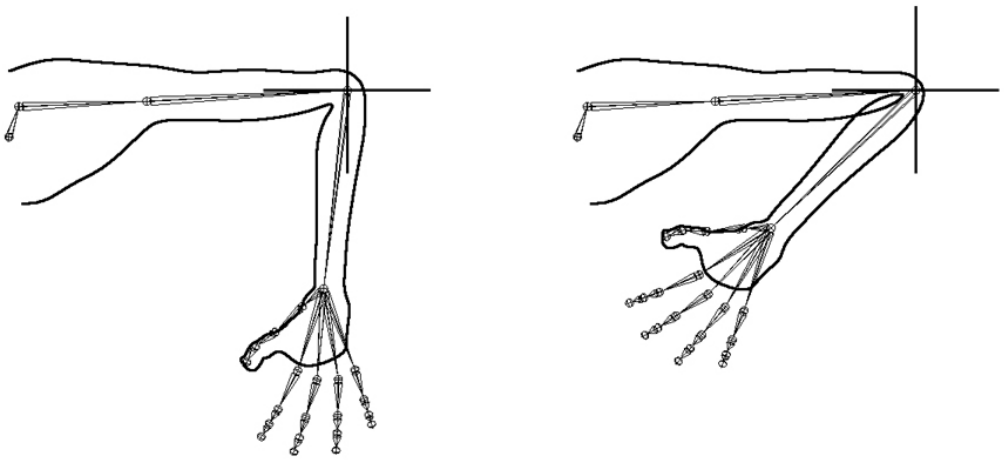


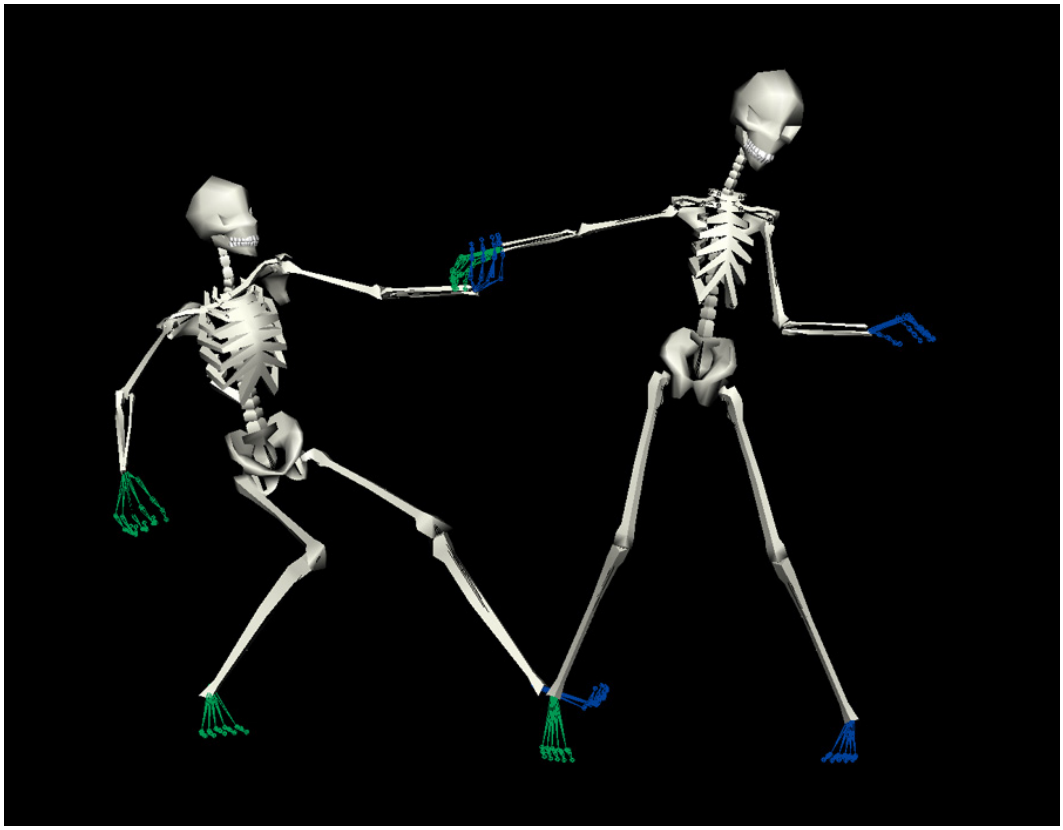
Figure 1_1 Loss of volume at the elbow area.

This paper suggests some methods and pipelines to improve skin deformations. All the methods described are "artists' methods". As such no there is no need for programming skills other than basic scripting.
It is important to keep two things in mind when reading this paper:

1. The techniques described are meant to create "beautiful pictures" rather than simulate the behavior of the real body.

2. The ultimate goal here is to use these techniques in real, small scale productions with limited resources. As such one have to bare in mind that skin deformations, as well as the entire character, is just one aspect of a more complicated picture and thus it have to share resources with other production needs.

II. Using Bones Geometry – improving skin weights.



Since we can use geometry as skin deformers (influence objects in Maya terminology) we can build a model of a skeleton and following these simple steps use it instead of the joints to deform the skin.

1. Build joints skeleton.
2. Create skin geometry.
3. Create bones geometry.
4. Bind only the minimum necessary joints.
5. Add the bones geometry as influence objects.

1. Why use bones geometry?

Since the bones do not deform and are simply parented to the joints there seems to be no advantage that will justify using them.
There are two reasons why I like to use bones geometry:
1. An anatomical visual reference
2. A weighting aid.

Anatomical visual reference.

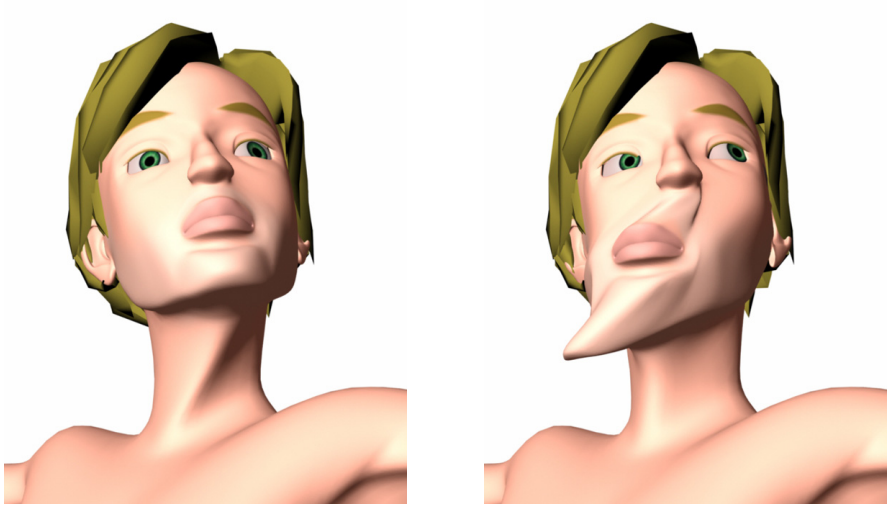
Having a visual reference of the skeleton can greatly help avoiding mistakes in positioning joints such as the spine or the hips. Bones geometry can also help positioning muscles.

Weighting aid.

Binding skin to bones geometry can improve the weighting pipeline.

Figure 2_1_1

Both samples don't use hand weighting. The right sample uses the joints. On the left the head is bind to skull geometry.



Both sides of Figure 2_1_1 present the same geometry bind with no hand weighting. The sample on the right uses only joints. While most noticeable, the stretching of the face as a result of the ribs influence is not the only problem. Notice that while the character turns his head there is no visible turning in the neck. This is because the entire region is being dominated by the ribs. The left sample uses geometry both for the head and neck. While not perfect this is a much superior starting point.

The bones geometry does not have to be highly detailed. In Figure 2_1_2 is the skull geometry that was used to deform the character from 2_1_1. There is a good reason to prefer a simple geometry over a detailed one. To achieve better weighting by using bones we have to use the geometry. Rather than simply take the position of the bone the software needs to sample the geometry for the weighting. The more complex the shape the longer this process will take. It is important to note that due to the technique I will explain here the complexity of the bones geometry is not a major problem and more complicated mesh can be used at little cost.

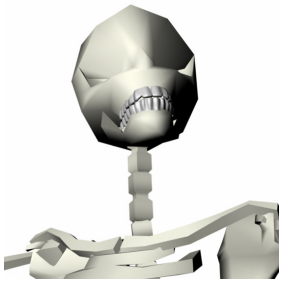
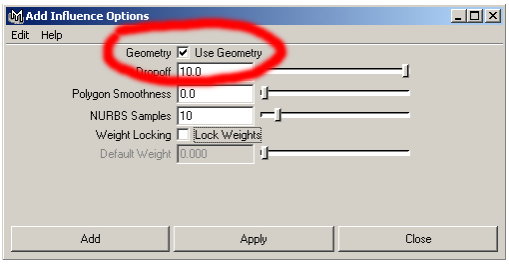


Figure 2_1_2

The skull geometry used in

Figure 2_1_3

When adding geometry as influence to a skin check the `Use Geometry` checkbox.



When adding influence objects in Maya you can choose if you want to use geometry (Figure 2_1_3). If you choose not to use geometry there is no need for one, thus you can use a simple locator. In this case the weighting will be done based on the objects position in world space*. Using geometry set skin weights based on the geometry of the influence.

In Figures 2_1_4 and 2_1_5 we see the same skin geometry bind to the same bones geometry. In 2_1_4 I didn't use the geometry of the bone. Since the real position of the bone is at its base end, next to the shoulder, the weights (in white) concentrate around that area. While, by changing the dropoff, I can widen the area of influence it will extend around the shoulder. Not the preferred results. In 2_1_5 the geometry was used to set the weights (by checking the use geometry check box). Obviously this is much closer to the desired results. In this example I used NURBS skin and poly bones. Poly skin will work the same way.

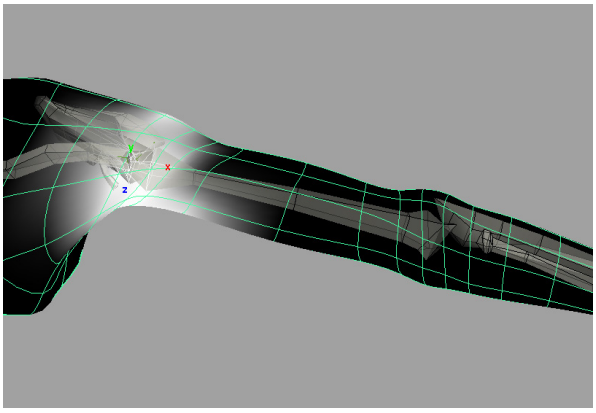
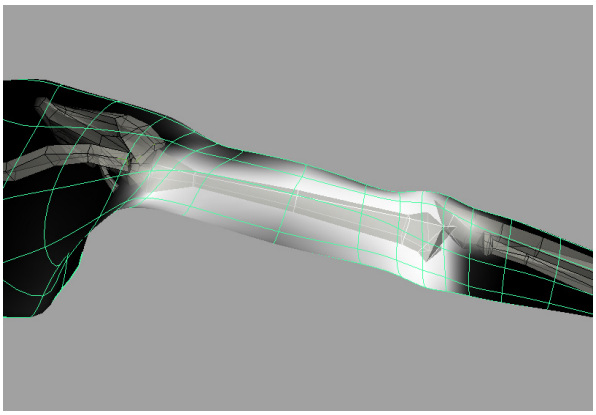


Figure 2_1_4 Using objects transforms



* For users convenience Maya shows objects translate manipulators based on local rotation pivot. Once freezed, the translate manipulator do not represent the objects real transforms. Using the object as an influence without geometry will set weights based on the real translates rather than the manipulators positions. In some cases the result will be no weightings assigned to the object. If you are using geometry the objects translates are irrelevant.

Figure 2_1_5 Using objects geometry

2. Optimizing bones geometry weights.

The only reason to use the geometry of the bones in the process of weighting is to get good weights. Once we got the weights we don't need the shape anymore. To get everything a little more elegant we can now get read of the geometry.

1. Bind the skin to as little joints as possible.
2. Add the influence bones using their geometry.
3. Acquire the skin weights and write them to a text file. (You will need to either write a mel script or download one)
4. Disconnect the skin.
5. Rebind the skin to the same joints as before.
6. Add the same influence bones, this time without using geometry.
7. Read the skin weights from the text file and assign them to the skin. (You will need to either write a mel script or download one)

The method described above will work fine but we can clean things farther more. Since the bones do not deform and we use them now in the same way we use joints we might as well get rid of them. The idea here is to acquire the weights of each bone and assign it to a joint. We can even combine weights of several bones and assign it to one joint. Unlike the previous pipeline that deals with the entire hierarchy and all the influences in this case we are dealing with influences and joints locally.

To simplify the pipeline all the stages are scripted to one procedure. The user need to select one joint (which will get the weights), one bound skin and as many bones as needed. The bones should not be influence objects and will not be at the end of the process. They are only to be used for automating weighting and can be deleted after running the script.

Assuming the skin is already bound to all the joints we want eventually to control the skin, here is a breakdown of the process.

Assumptions for this pseudo mel code;

a) skin geometry is poly
b) skin clusters name is 'skinCluster'
c) skin geometry name is 'skin'
d) bone geometry is 'bone'
e) the number of vertexes in the skins is stored in $numOfVTX

1. Add bone geometry as an influence object.

2. Get all the skin points that are influenced by a given influence;

   We will run a loop checking for each of the skin points what joints and bones influence it. We will then compare the results (a string array) with the name of the bone (influence object) we are working on. If (for the point currently checked in the loop) we have a match we will store the current points' index number in the $pointIndex variable.

```
//
int $pointIndex[];
int $i = 0;
for ($c=0;$c<$numOfVTX;$c++){

      // skinPercent is the command to get the information.
      // -q is to put it in query mode.
      // -t is to get the influence object.
      // the next argument (skincluster) is the skin cluster.
      // and last, the point we are checking (say skin.vtx[554]).

      string $infs[] = `skinPercent -q -t skinCluster
      ("skin.vtx["+$c+"]")`;

      // this will result in an array that might look like this:
      // {"joint1","joint2","bone"}
      // we now check to see if there is a match to the influence
      // we are looking for, in this case `bone`.

      for ($n=0;$n<size($infs);$n++){
            if ($infs[$n]=="bone"){

                  // if we do fine a match we store the number
                  // in the $pointIndex var

                  $pointIndex[$i++] = $c;
            }
      }

}
// Result: 124 125 470 949 950 951 952 958 960 961 962…
//
```

3. Get the weights of the influence for each of the points.

   We will run a loop for each of the points we found in the previous stage and get the weights assigned at each of the points to the bone (influence object) that we are checking. We will store the values in a float array. The values must not exceed 1.0.

```
//
float $pointVal[];

// we will run this look for every skin point that was stored
// in the previous loop.

for ($n=0;$n<size($pointIndex);$n++){

      // if we want to add the weights of several
      // influences we need to run another loop for each
      // of the influences. If this is the case we will need
      // to run the previous loop (stage 2) for every bone
      // and clean redundancy in the results (skin points
      // that are influenced by more than one bone in our
      // list. (here instead of using `bone` as I did in
      // stage 2 I will use $infs[], a string array.)

      for ($c=0;$c<size($infs);$c++){

            // once again, the skinPercent command.
            // -t for the influence.
            // -q for query.
            // the skin cluster.
            // the skin point.
            // notice that these are the same flags as before.
            // the difference are the order and that this time
            // we define for the -t flag the influence.

            float $tempPointVal = `skinPercent -t $infs[$c] -q
            $skinCluster ($skin+".vtx["+$pointIndex[$n]+"]")`;

            // if we have more than one influence we can add the
            // the values. It is better to use the min command to
            // avoid errors that might occur due to inaccuracy
            // that will results in values greater than 1

            $pointVal[$n] = `min 1.0
            ($pointVal[$n]+$tempPointVal)`;
      }
}
//
```

4. Detach the influence(s) from the skin.

   While this stage can easily done in the interface, assuming that the entire process will be scripted the command is as follow:

```
//
skinCluster -e -ri bone skin;
//
```

5. Assign the weights to a joint

```
//
// run the loop for every point in the array we got at stage 2.

for ($n=0;$n<size($pointIndex);$n++){

      // the skinPercent command
      // -tv assign weight value to a specified influence
      // the first argument is the influence (string)
      // the second argument is the weight.
      // the skin cluster
      // the skin point to assgine the weights to.

      skinPercent -tv joint $pointVal[$n] skinCluster
      ("skin.vtx["+$pointIndex[$n]+"]");
}
//
```

3. Summary.

We can use geometry to get better weights for our joints. We don't have to pay the cost of additional geometry and heavier skin clusters (the more influence in a skin cluster, the slower it is) because we can translate the weights of the geometry to the joints. The bones therefore, are used merely to generate better weights and can be deleted after. In many cases of minor characters who do not perform demanding poses this can be sufficient to get a good looking character.

The process of transforming the weights can take about 30 seconds for a 3000 points skin. Having to do so for series of joints and bones can get tedious. To automate the process we need to establish a connection between bones and joints. This can be done by generating a message attribute that connect between joint and corresponding bone(s). Creating a character null and connect all the characters` joints using the same message attribute method will enable as to easily write a script that will scan and find all the joints of a character, will than find all bones that are associated with every joint and than using the method described above set the weights in "one button" style.

III. Muscles – deformed deformers.

In most cases the challenge of getting better skin deformation is more complex than just improving weights assigned to joints. To understand this lets first take an arm as a test case and examine what is happening when it bent*.

Figure 3_1

Direct skinning of skin geometry to joints. The crease collapse inside

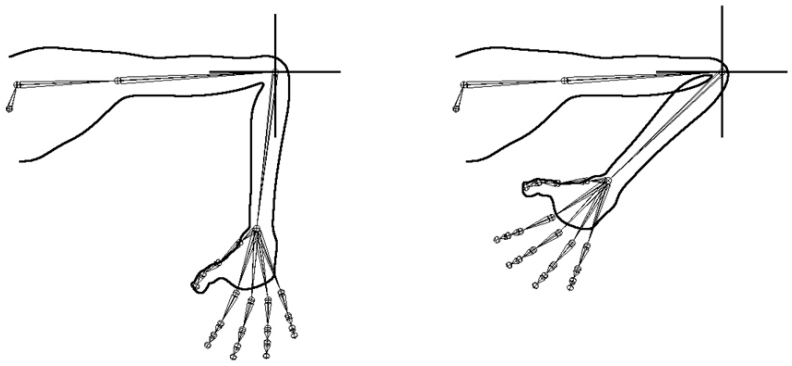
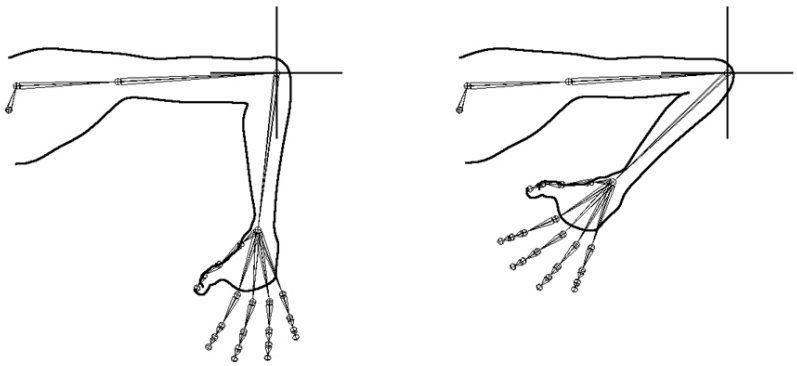


Figure 3_2

Skinning using intermediate deformers layer helps maintaining the volume of the skin



Both Figures present the same geometry and joint structure. The geometry in Figure 3_1 fails to keeps its volume as it bents. The sample on 3_2 does a better job. The red line in 3_3 represents the results from 3_2.
We can see that both the arm and the forearm maintain their volume while the crease area is being "pushed" away from the elbow. To get a better understanding of what is going on let's take a closer look and try to follow the skin surface points.

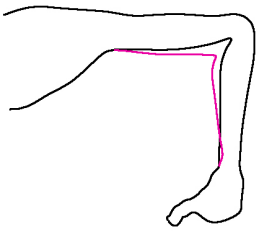


Figure 3_3
Comparing the two samples

* All the line samples in this section are processed (for better visualization) Maya screen shots of actual bound geometry (the curve representing the skin line is generated from a bound NURBS surface). It therefore represents real 3D deformations.

Figure 3_4

Path of skin points at the crease area. Direct bind of skin to joints. Darker points (in red) correspond to the darker skin curve.

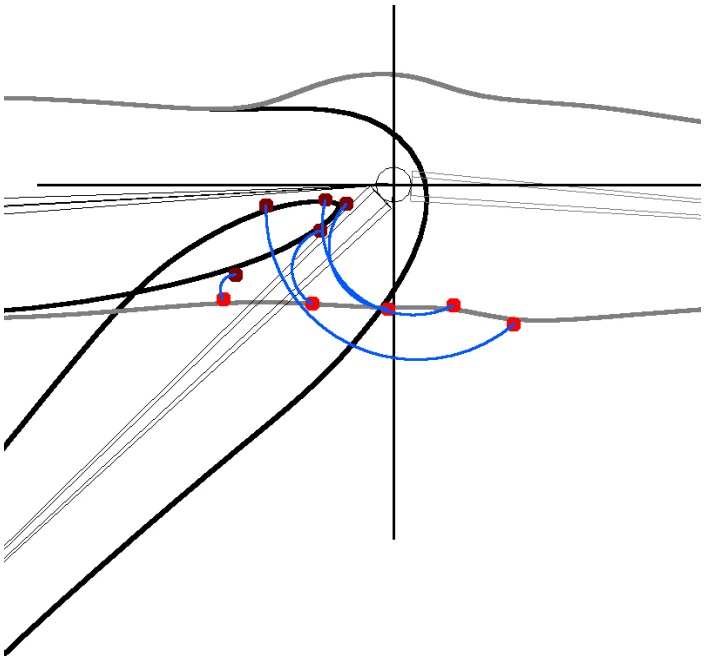
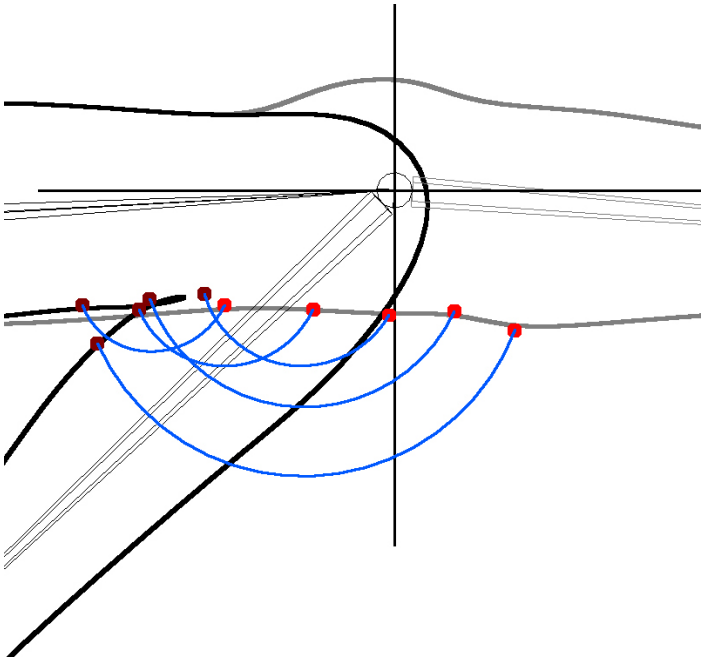


Figure 3_5

Same geometry but with additional deformers. Points pushed away from the elbow. Darker points (in red) correspond to the darker skin curve.



Careful inspection of the paths the skin points in Figure 3_4 gives some hints to the nature of our problem. Since all the motion of the points stems from the rotation of the joints the motion paths create arcs with their center around the pivot of the controlling joint. At its simplest form skin cluster is based on groups of points that are assigned to specific joints. The center of the arc of the extreme right point is almost entirely at the forearm joint pivot point. The more the left the

pivot seems to move. This is because the forearm gradually shares the weights of each point with the arm joint. The ability to share weights of individual skin points between joints is important improvement in modern 3D software. It is however insufficient as long as it is being applied solely to the joints. Of the five marked points in 3_4 and 3_5 the middle (third from both sides) is the most interesting. It is the point at the exact center of the crease, thus critical for the deformation. Its final position in 3_4 is too close to the joints pivot. We need to push it "out" but no simple weighting will place it at its position in 3_5. In this example the skin is bound with additional deformer that offset the points.

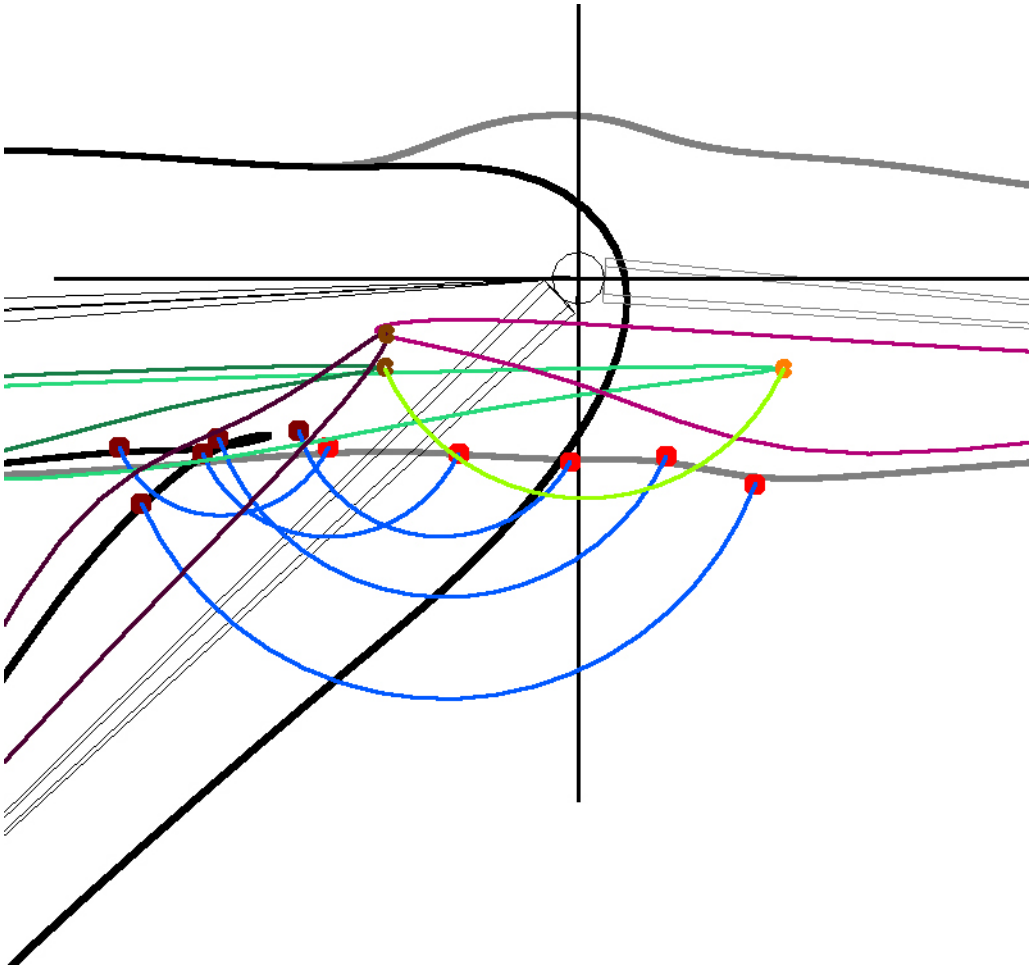


Figure 3_6 Same geometry with "muscle" deformers.

Figure 3_6 reveals the additional deformers. These are NURBS surfaces, connected at their poles to the joints. 3_6 depict two stages, spread and bent arm. All the components are lighter in the spread mode. One deformer is red (for the forearm) while the other is green (the arm deformer). The path of the end pole of the arm deformer is in light green and its pole point is in orange brown. This end pole is parented to the forearm joint. Both deformers actually stretch through the motion. The arm (green) deformer shifts both last arm and first forearm points up the arm.

While there are other ways to achieve similar results setting up deformers that will behave in a way that resembles muscles is my method of choice. I see it as a systematic solution and an answer for several character setup issues as I will try to demonstrate here.

1. Anatomy of a simple muscle.

First we need to define the characteristics of a muscle deformer.

1. It should be flexible.
2. Can have multiple control points, typically two or three, that can be connected to the skeleton.
3. Any change to one control point should influence the entire muscle.
4. It should maintain its volume.

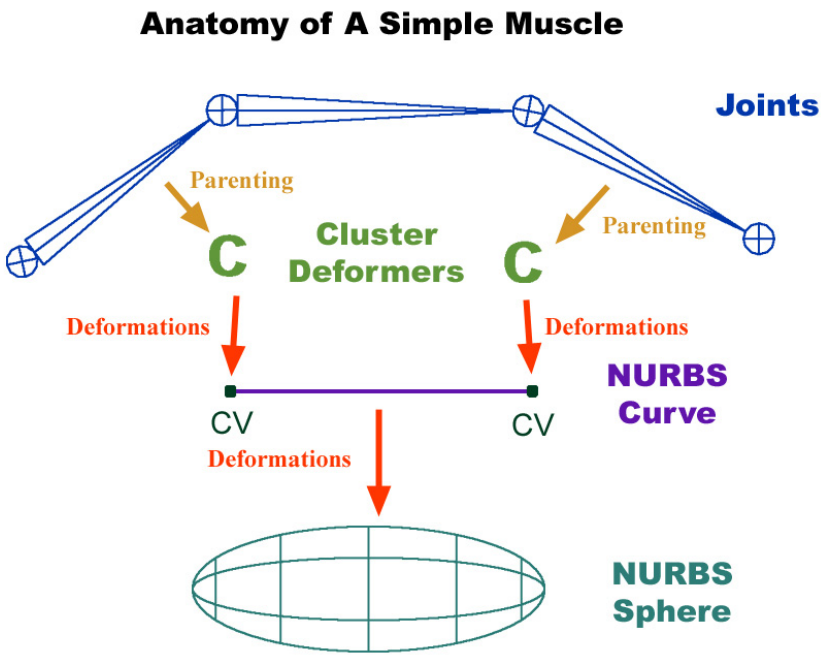


Figure 3_1_1 Anatomy of a simple muscle

In Figure 3_1_1 we can see the simplest form of hierarchy and deformers that can create a behavior that will resemble a muscle. A NURBS sphere in being manipulated using a NURBS curve functioning as a wire deformer. Each of the curve's CVs is being controlled by a cluster deformer. The clusters are parented to the joints. The number of CVs in the curve determines how many points we have to hold the muscle.

2. Building a simple "muscle".

8. Creating NURBS sphere geometry

   Duos to their flexible nature NURBS surfaces are ideal for muscles geometry. Sphere is a good shape to start with.

9. Controlling the muscle surface

   Ideally when manipulating one point the entire object should respond. A NURBS curve used as a deformer (wire deformer in Maya terminology) can give much of the needed control. The curve should have a single span (two edit points) so every point influence the entire curve.

   - Click select the muscle (NURBS sphere)
   - **Display → NURBS Components → CVs**
   - Using the **v** key (snap to point) create a NURBS curve between the two poles of the NURBS sphere.

   For two control points use a degree 1 (linear) curve (two CVs).
   For additional control use a degree 2 curve. This will give three CVs.

   - Select **Deform → Wire Tool**
   - Click the muscle geometry (NURBS sphere)
   - Press **Enter**
   - Click the NURBS curve
   - Press **Enter**

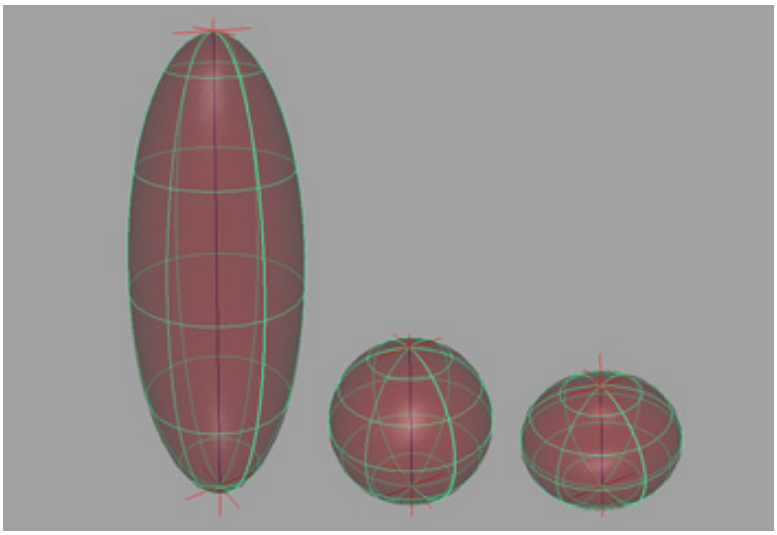
10. Controlling the curve

    We will use cluster deformers to “hold” the curves from each side.

- Click select the muscle (NURBS sphere)
- Hide the muscle. **Display → Hide → Hide Selection**.
- Click select the NURBS curve.
- **Display → NURBS Components → CVs**
- From the **Status Line** choose **Select by component type, points**.
- Click select one of the curve’s CVs.
- **Deform → Create Cluster**.
- Repeat for each of the curve’s CVs.

Figure 3_2_1

This is what we have now. We can stretch and squash the sphere by dragging the clusters. The sphere will not maintain its volume.



11. Maintain the muscles volume.

    To maintain the muscles volume we need to scale the two axes that are perpendicular to the sphere's length axis using this simple formula:

    Scale = Original Length / Current Length

    There are several ways to achieve this. One way is to write it as an expression. A faster way though is to use Maya's binary nodes. I will show both ways. Both ways do exactly the same[*] but because the expression method force Maya to parse the commands the results are slower.

    The expression will be shown here because even though it uses Maya's expressions syntax I believe it will make sense to non Maya users.

    The binary nodes method is more of a Maya way of doing things but is faster and involves less coding.

[*] The expression described bellow is working fine but will not update only when time change, i.e. it will render and work fine for animation but will not update to work space as you manipulate joints.

Using expression:

Write all of the texts in the **Script Editor** window.

- First get the original length:

```
Float $pos0[3] = `xform -q -ws -t curve.cv[0]`;
Float $pos1[3] = `xform -q -ws -t curve.cv[curve's
degree]`;
```

  This will create two float arrays representing the world space coordinates (the –ws flag) of the first (0) and last control vertexes of the curve. The number of the last CV of a one span NURBS curve is the same as the curves degree, i.e. 1 for a degree one curve, 2 for a degree two and so.
- Now, based on the positions find the length

```
Vector $origV =
<<$pos1[0]-$pos0[0],$pos1[1]-$pos0[1],$pos[2]-
$pos0[2]>>;
float $origLength = mag($origV);
```

  By creating a vector from the difference between the two sets of coordinates and using the `mag` command to measure the vectors length we can get the original length of the muscle.
  All this process should generate a float number.
- Now we need to write the same lines as an expression. To easily scale the sphere-muscle we can use the scale attribute of the wire deformer.

```
Wire.scale[0] = X /  $currentLength;
```

  With X being the result of the previous stage and $currentLangth is the equivalent to $origLength.

Once the text is done we can make it an expression.

- **Window → Animation Editors → Expression Editor…**
- Copy the text from the **Script Editor** to the **Expression Editor**.
- Press **Create**.

Using Binary nodes:

- First find the curves CVs position in world space

```
//
// The curveInfo node generate information about
// given curves, including world space coordinates
// of the curves CVs

createNode curveInfo;
// Result: curveInfo1 //

// using the connectAttr command connect the
// worldSpace attribute of the curve used to
// deform the muscle surface to the inputCurve
// attribute of the curveInfo node

connectAttr -f curve.worldSpace[0] curveInfo1.inputCurve;
//
```

- Now to find the distance between the CVs

```
//
// The distanceBetween node accept two points
// and output their distance.

createNode distanceBetween;
// Result: distanceBetween1 //

// Now connect the first point (0) to the
// first input of the distanceBetween node.

connectAttr curveInfo1.controlPoints[0]
distanceBetween1.point1;
// Result: Connected curveInfo1.controlPoints to
// distanceBetween1.point1 //

// connect the last point (1 if a degree 1 curve, 2
// if a degree 2 etc.) to the second input point
// in the distanceBetween node.

connectAttr curveInfo1.controlPoints[1]
distanceBetween1.point2;
// Result: Connected curveInfo1.controlPoints to
// distanceBetween1.point2 //
```

After acquiring the necessary information we need to calculate the scale.

To do this we can use the **multiplyDivide** node.
Unlike the previous two nodes there is GUI for creating this one.

- **Right Mouse Button** in the **Hypergraph**
- **Rendering → Create Render Node**
- Click on the **Multiple Divide** button to create a multipleDivide node.

Figure 3_2_2 The MultiplyDivide node icon

- Open the **Attribute Editor** window.
- Change the operation to **Divide** as shown in 3_2_3

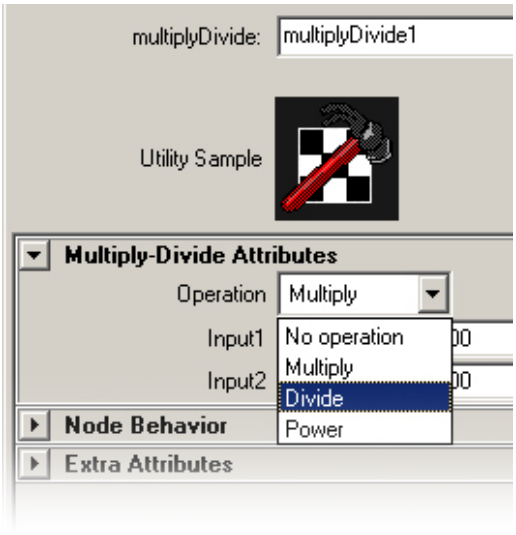


Figure 3_2_3 Set operation to divide

The **multplyDivide** node is a render node, thus have three channels for each input. We need only one so we'll use the X channel.
We need to input the output of the **distanceBetween** node into both input1 and input2 but in a different way. While for input1 (the original length of the muscle) we want a constant, for input2 (current length) we need a variable. This will dictate the way we set the two inputs.

For the dynamic connection we can simply use the connection editor.

- Open the **Connection Editor** by dragging with the **middle mouse button** the **distanceBetween** node to the **multiplyDivide** node.
- Connect the distance attribute of the **distanceBetween** to the input2X attribute of the **multiplyDivide** node (3_2_4).

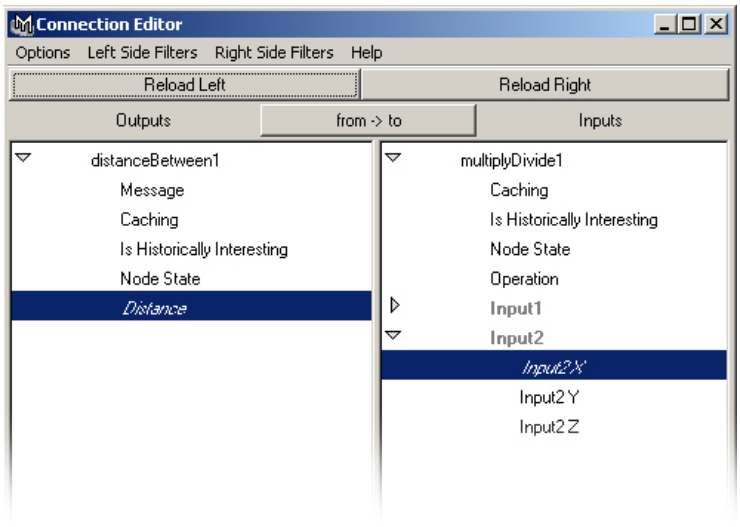


Figure 3_2_4 Connecting the distance attribute to input2X

To set the constant we can simply copy paste the value from the input to the first one. A more accurate way is to do this from command line.

```
//
setAttr multiplyDivide1.input1X
`getAttr distanceBetween1.distance`;
//
```

In both cases the **Attribute Editor** window of the **multiplyDivide** node should look something like 3_2_5.

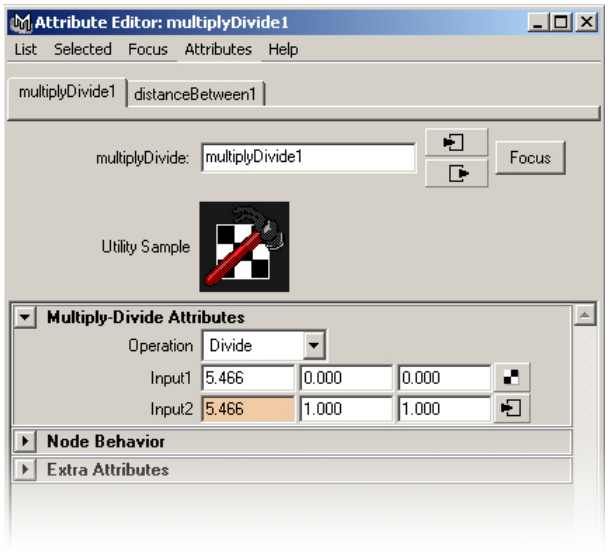


Figure 3_2_5 multiplyDivide attribute editor window

The last stage is to connect the output of the **multiplyDivide** node to the scale attribute of the wire deformer.

- **Middle mouse drag** the **multiplyDivide** node to the wire node.
- Connect the outputX attribute of the **multipleDivide** node to the scale attribute of the wire node.

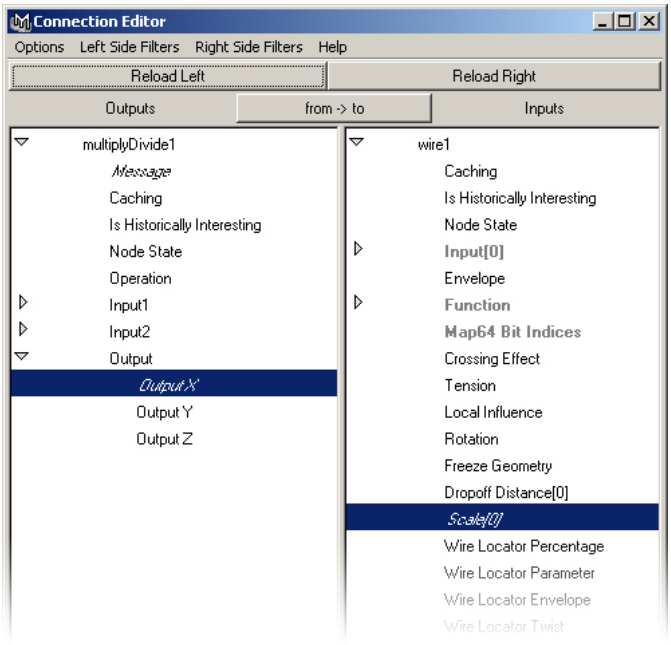


Figure 3_2_6 Connecting outputX to scale

Here is a hypergraph representation of the structure we've created. (This one was nicely organized by hand. While building this structure it will not look as neat.)

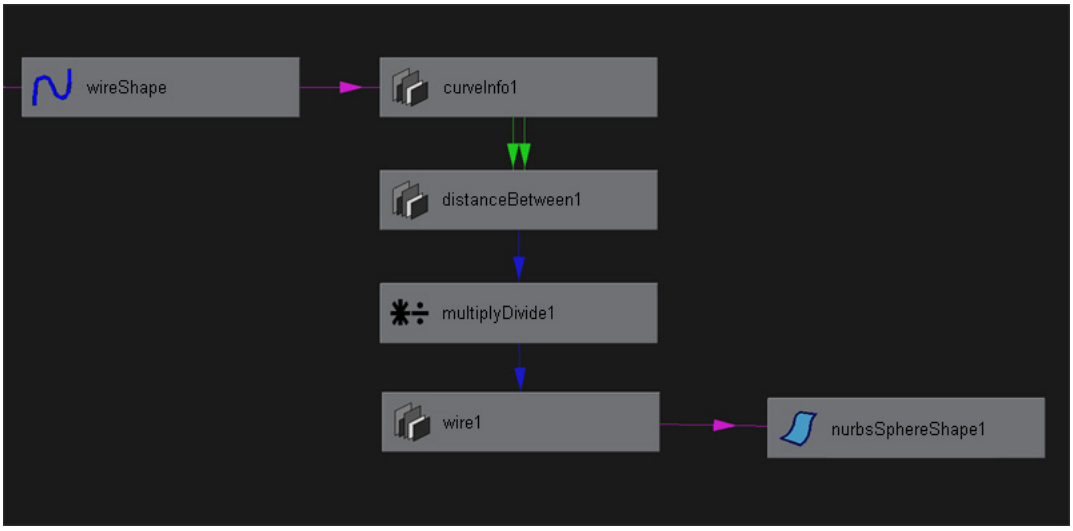
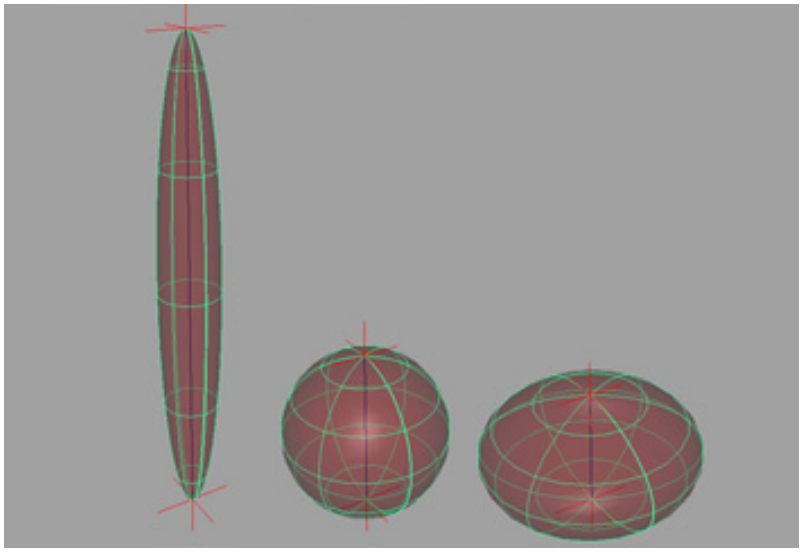


Figure 3_2_7 Hypergraph tree representing the structure

And finally here is what the "muscle" look like when manipulated.

Figure 3_2_8

This is the same muscle but the scale is controlled by the expression described above.



12. Connecting the muscle to the skeleton.

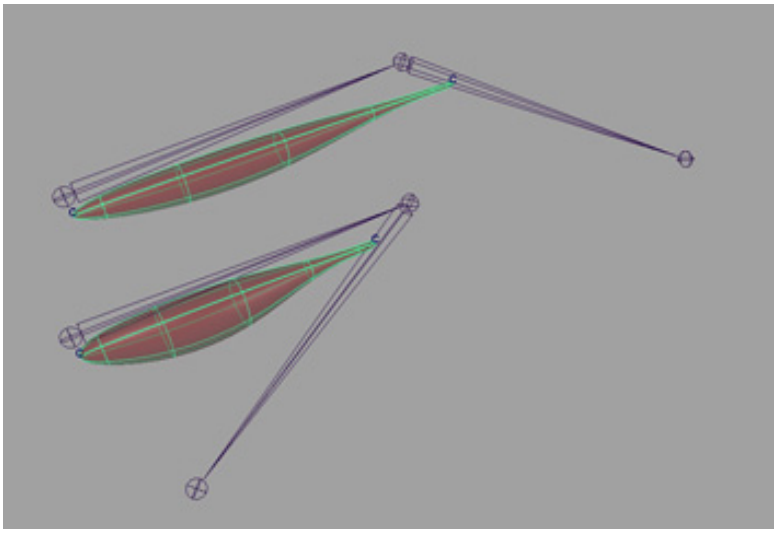
    This is doe by simply parenting the clusters to the joints. The only thing here is to understand what cluster to parent to what joint. Usually it is fairly easy to relate a muscle to a specific joint. Let's call this joint the muscle joint. The cluster that controls the base (start) of the muscle should be connected to the parent joint of the muscle joint. The cluster that controls the end of the muscle should be connected to the joint that is the direct child of the muscle joint.

    For example let's assume that we have a three joints chain; arm joint, forearm joint and hand joint at this hierarchy. If our muscle is part of the forearm than the start cluster should be parented to the arm joint and the end cluster should be parented to the hand joint.

    - Click select the joint you want to be the clusters parent.
    - **Edit → Parent** (or just press **p**).

Figure 3_2_9

Here we can see how the muscle deform and maintain its volume once connected to the joints.



13. Getting the "muscle" to influence the skin.

- Select the skin geometry.
- Shift select the muscle geometry.
- **Skin → Edit Smooth Bind → Add Influence → option box**

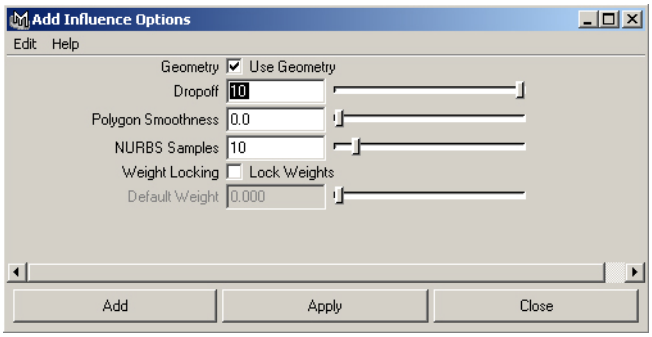
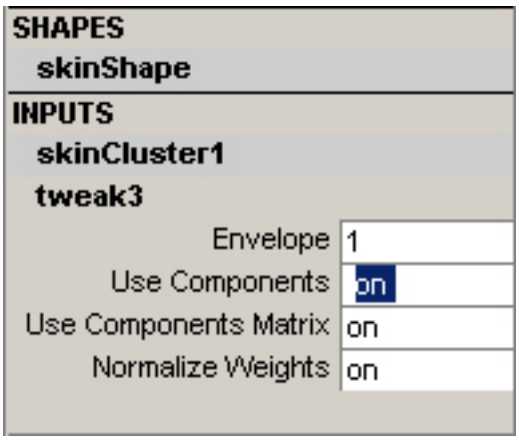


Figure 3_2_10 Add influence option box

- Check on the **Use Geometry** check box (see 3_2_10).
- On the work space window click on the skin geometry.
- On the channel box click on the skin cluster
- Check **Use Components** on (3_2_11).

Figure 3_2_11

Skin cluster options. Set Use Components to on.



The fine prints:

- When using geometry in influence and components in skin cluster it is important to freeze scales to 1 1 1 for the influence object.
- Do not freeze the translates of the influence object (this rule applies to every object that influence a skin cluster, whether it's a joint or a geometry, using or not using the actual geometry).
- Once you've clicked the Use Components on the skin deformations will become much slower. This requires adjustments in workflow, such as proxy light skin or separate rigs for animation and deformations.

14. Setting the hierarchy.

    The last stage is to set the hierarchy so everything will work properly. Figure 3_2_12 shows a simple sample that includes all the components in the right order.

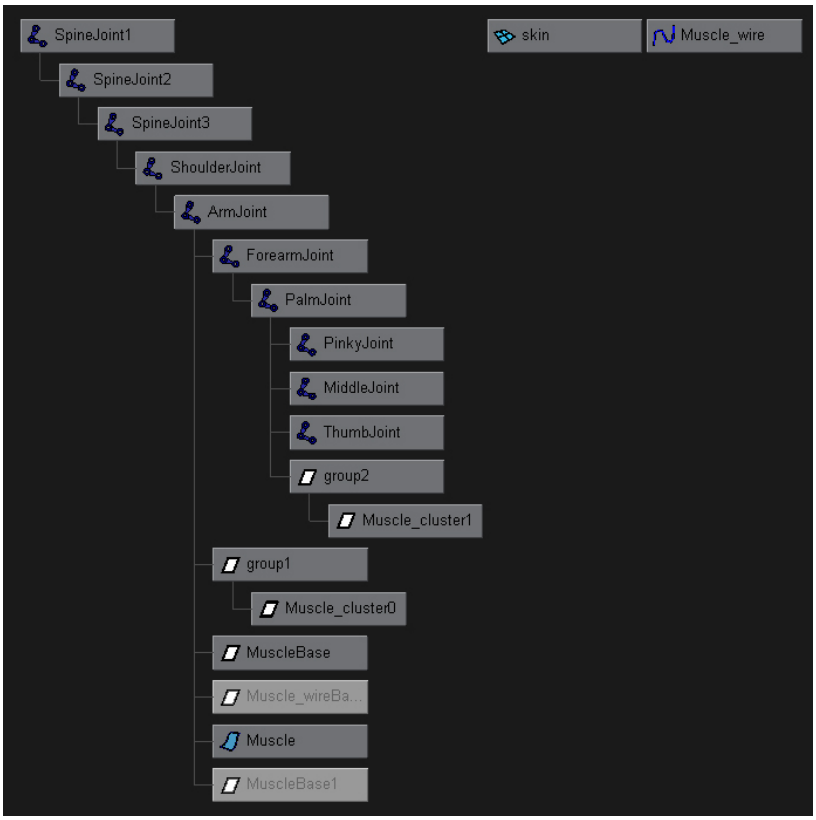


Figure 3_2_12 Muscle hierarchy

All the components of the muscle but the NURBS curve that is used as the wire deformer should be parented under the joint that is the direct parent of the joint the muscle correspond to.

Since this is confusing here's an example:

- Say we have a muscle between the arm and the hand.
- We can consider this muscle as corresponding to the forearm.
- Therefore all the components should be parented under the arm joint.
- These include the NURBS surface that is used as the muscle, the two NURBS surface base nodes (one for the wire deformation, the other for the skin cluster deformation) and the wire base node (created for the cluster deformers).
- Do not parent the wire curve and do not change the clusters position in the hierarchy.

3. Making muscles twist.

As long as our muscle doesn't need to twist everything should work fine. Once a twist is involved, like in the forearm, we get the problem depicted in 3_3_2. This happens because the wire deformer can not twist, therefore the CVs from one side simply move to the other side.

Figure 3_3_1

The circle around the second joint shows the twist motion of the joint. Notice that although the muscle follows the joint it does not twist. Such a muscle is useless for joints like that twist like the forearm.

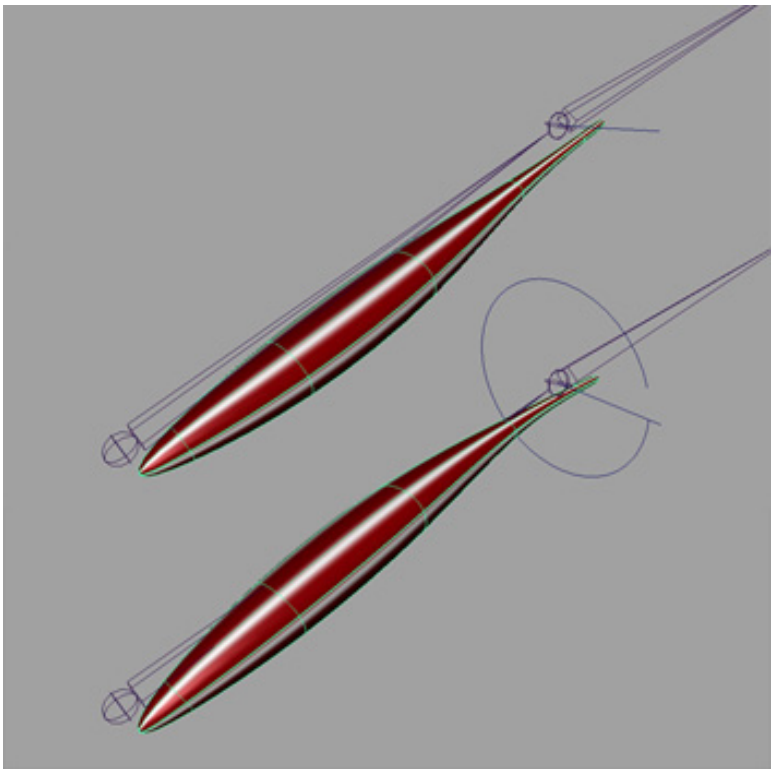
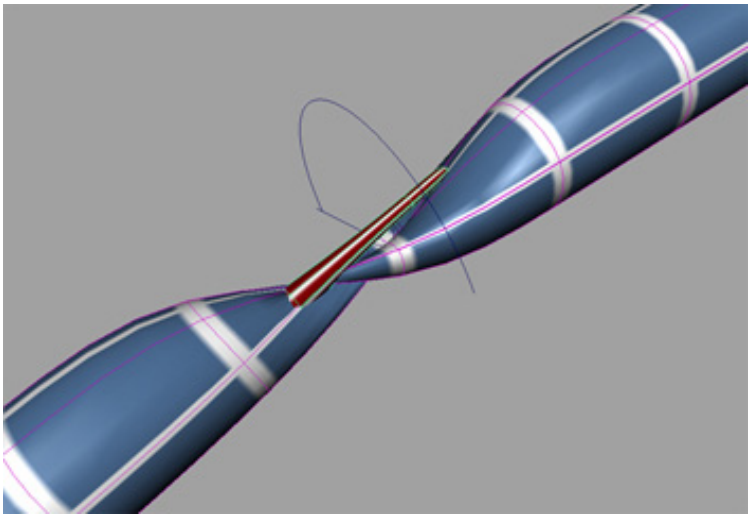


Figure 3_3_2

And this is what we'll get if we will try to attach such muscle to a skin. Instead of a nice gradual twist the skin points just switch positions.



Once we need to deal with twist problems we have to break rotation axes. The reason is that we will need to tie certain operations to the twist axis. This requires certain considerations:

a) We must decide what will be the twist axis. I always use Y.

b) We must get the twist axis as consistent as possible. While there is no absolute solution for this generally it is a good idea to let the software calculate the twist axis either first or last. In the case of the Y axis this means changing rotation order to something like YXZ or XZY as oppose to the default XYZ*.

c) We must have all the joints properly and consistently oriented. There is no redundancy in the previous sentence. It is not enough to have the joints nicely oriented for easy forward kinematics animation. We will need to set several nodes and all will have to rotate consistently and maintain their relationship. It is important that if, for example, joints are oriented pointing towards positive Y with positive Z as their up axis these conventions will be kept all through the rig.

Adding non linear twist deformer.

One very simple way to get a twist is to add a non linear twist deformer directly to the muscle geometry (the NURBS surface).

- If we add the twist deformation after setting the wire deformer we will have to reorder the deformations so the twist will work before the wire. The line should be something like this (depend upon objects names):

```
reorderDeformers "wire1" "nurbsSphere1_twist"
"nurbsSphere1";
```

There are no flags for this command. It will set deformer two before deformer one so the twist deformer will work before the wire deformer.
Now all we need to do is connect the `.endAngle` attribute of the twist to the twist axis of the joint.

* For additional information regarding rotation orders see part one, rotation order, of Jason Schleifer's section of this course.

Figure 3_3_3

Exactly the same objects at the same angle and with the same weights. Much better results.

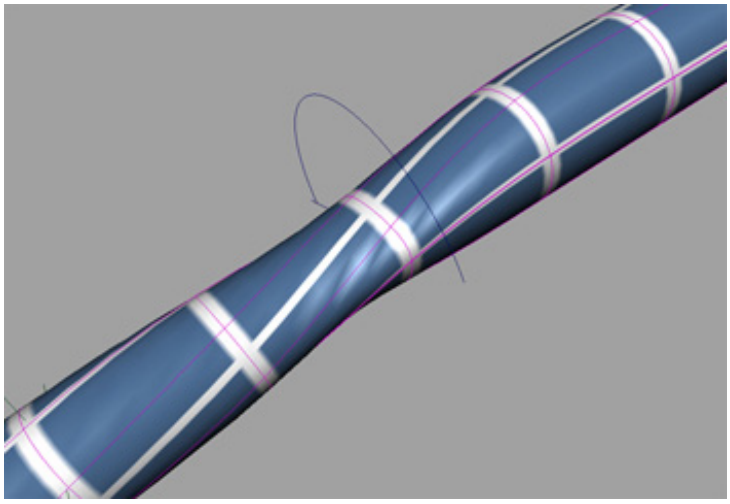
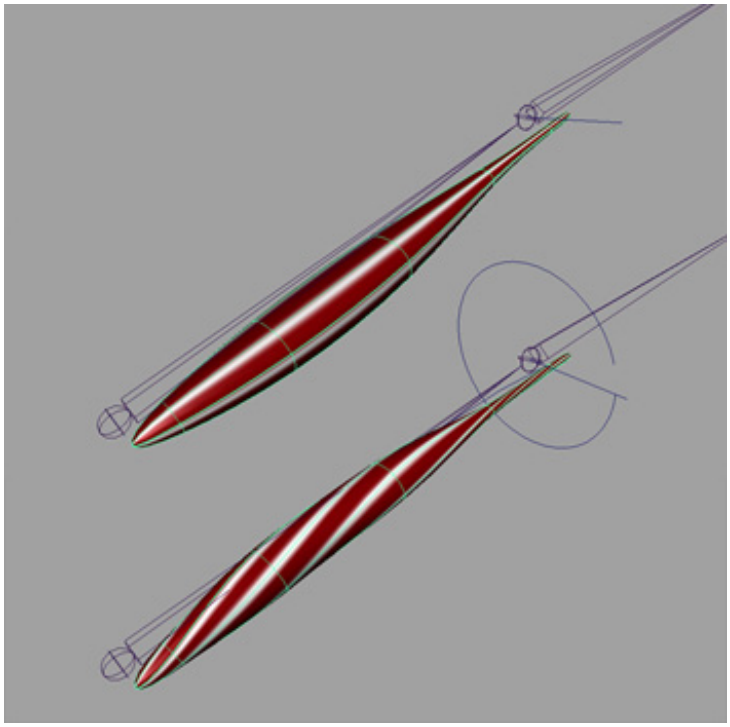


Figure 3_3_4

Here is what the muscle looks like after setting the twist.



The problem with this setup is that while twisting the muscle fails to keep its end pole in the right direction. While sufficient for simple, not too demanding setups, for more ambitious rigs we need a more accurate method.

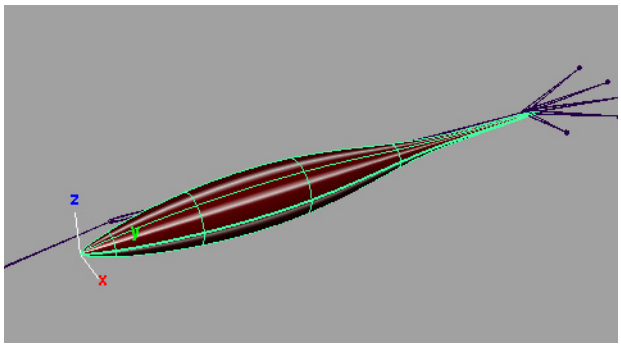
The multi wire Muscle.

This is actually a flavor of the previous setup. Instead of setting one wire deformer for the entire muscle we can set multiple wire deformers to get more control over the muscle surface. In this case we will set a wire deformer for each of the V isoparms of the sphere. However using multiple curves is not the only principle difference here. To achieve the twist we will need to grab in each side more than one CV per curve, thus the rules are going to be somewhat different. The best way is, as usual, to demonstrate a step by step.

1. Create muscle geometry based on NURBS sphere.

Figure 3_3_5

Creating a NURBS sphere at the right place and size.

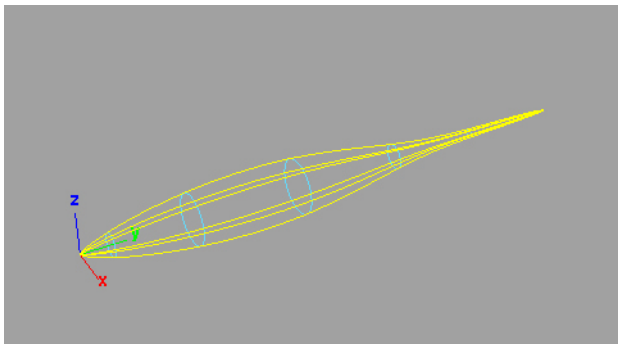


2. Select all the V isoparms of the sphere.

- From the **Status Line** choose **Select by component type, lines**.

Figure 3_3_6

Selecting all the V isoparms of the sphere.

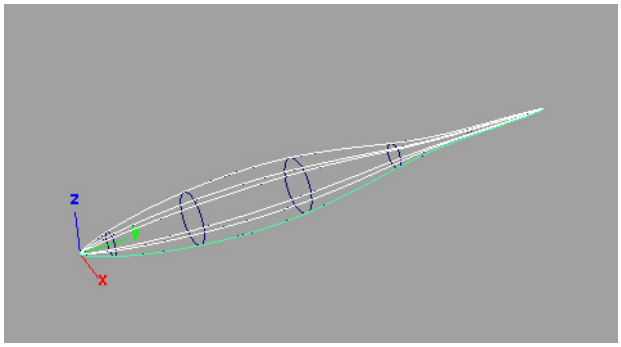


3. Duplicate the isoparms.

- **Edit Curves → Duplicate Surface Curves**.

Figure 3_3_7

Duplicating the isoparms to NURBS curves.



4. Rebuild the curves to a degree 3, two spans curves.

- **Edit Curves → Rebuild Curve → option box**.

Figure 3_3_8

Rebuild curve option box set to a degree 3 curve with two spans.

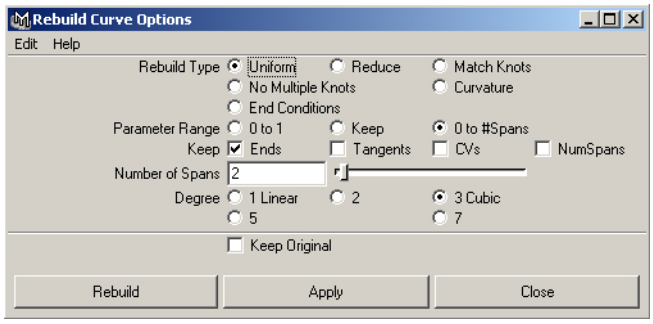
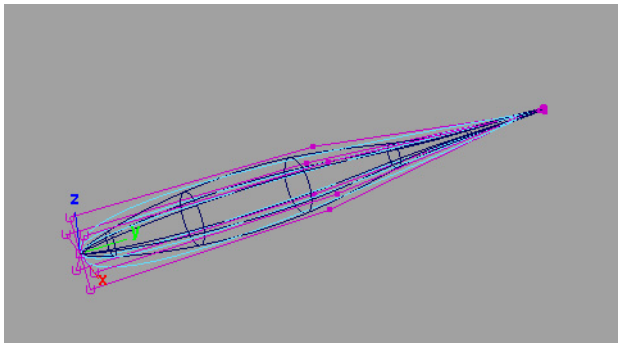


Figure 3_3_9

This is what the new curves look like after rebuilding. Each has five CVs now. Though it is hard to see in the picture the second CVs from the far pole actually flipped places and generally the curves do not follow the spheres surface.

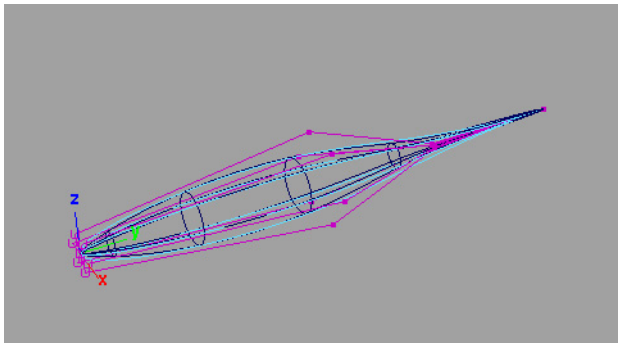


This will give us curves with 5 CVs each. We need two CVs at each pole to get the twist effect. This is because the only operation possible for CVs is to move them. CVs can not rotate or be scaled (which is the case for every real point). Rotate or scale points is actually changing their relative positions, thus we need at least two points to see any effect. This will require four points. The additional, fifth, CV, at the center of the curve, is to get a nice twist effect. Beyond certain degrees we must have at least three holding points. If we grab the geometry only from its poles the center will break and collapse into itself.

5. Manipulate the CVs to better follow the lines of the geometry.

Figure 3_3_10

Manipulating the CVs of the curves to better correspond to the spheres surface.

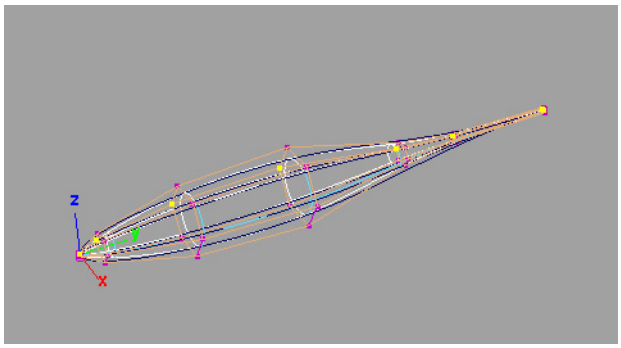


6. Set every curve as a wire deformer to the CVs that correspond to the isoparm from which the curve was generated. This is a rather tedious and confusing stage and is best handled by an automating script[*].

- Set mode to **object mode**.
- **Deform → Wire Tool**.
- Click on the sphere to highlight it.
- **Right mouse button** click on the highlighted sphere and select **Hull** from the **marking menu**.
- Click on the hull to highlight a raw of CVs.

Figure 3_3_11

Selecting one row of CVs by clicking the hull.

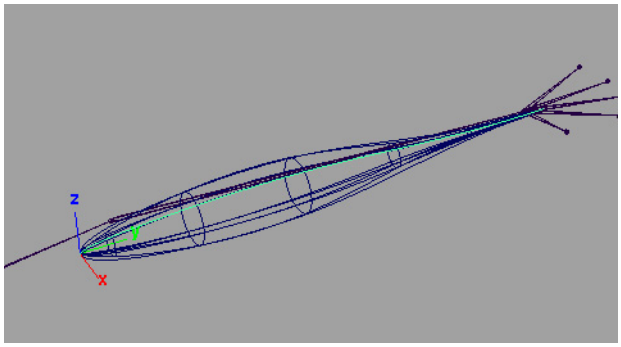


- Press **enter**.
- From the **panel menu** select **Show → NURBS Surfaces** to hide the sphere.
- Click on the curve that is closest to the CVs row you've just selected.

[*] While not in the scope of this paper one tip for writing such a script should be given here. Each isoparm in a NURBS sphere have a number. The same is true for every CV though the CVs have two numbers, for U and V coordinates. The CVs numbers can also be seen as row and columns numbers. If we have six isoparms (0:5) we should have six rows of CVs (0:5). This indeed is the case but the CVs row corresponding to isoparm 0 is row 1. Isoparm 1 have row 2 etc. the last isoparm (in this example 5) correspond to row 0.

Figure 3_3_12

Selecting the corresponding curve.



- Press **enter**.
- From the **panel menu** select **Show → NURBS Surfaces** to show the sphere.
- Repeat for all the curves.

7. Connect the muscle to the joints.

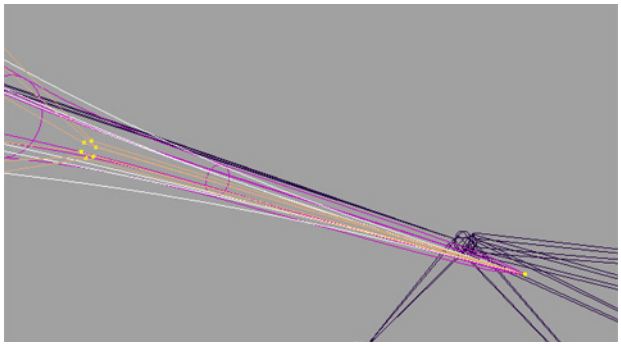
This is similar to the simple muscle from the previous example. We will use clusters to "hold" the wire deformers. We will than parent the cluster in the same way we did in the previous example.

In this case we need to create three clusters, one for each pole and one for the center. Each pole cluster holds the two extreme CVs of each curve. For a six isoparms sphere each of the pole clusters should hold 12 CVs while the center cluster should hold six CVs.

- From the **Status Line** choose **Select by component type, points**.
- **Drag click** on the curves to highlight all of them (but not the sphere)
- **Drag click** to select all the CVs that will be connected to the cluster.

Figure 3_3_13

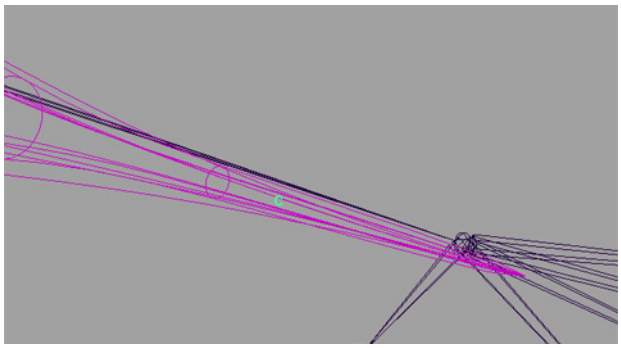
Selecting CVs to attach to a cluster deformer. The extreme two CVs of each curve should be selected. Since all the last CVs of the curves share the same position they look as one point. The second points which shows more clearly. Each group has six CVs.



- **Deform → Create Cluster**.

Figure 3_3_14

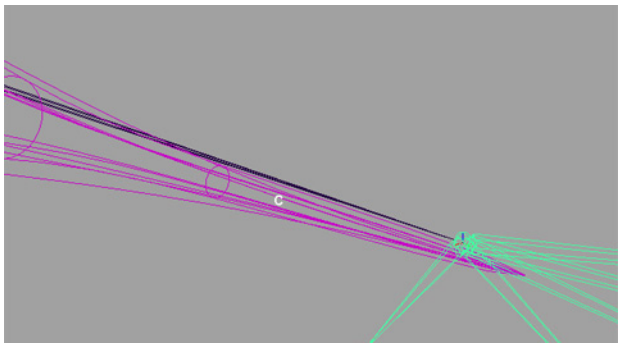
Creating the cluster.



- From the **Status Line** choose **Select by object type**
- Click select the joint you want to be the clusters parent.

Figure 3_3_15

Parenting the cluster to the hand joint.



- **Edit → Parent** (or just press **p**).

We should end up with three clusters. One cluster fro each pole and one cluster in the middle of the muscle.

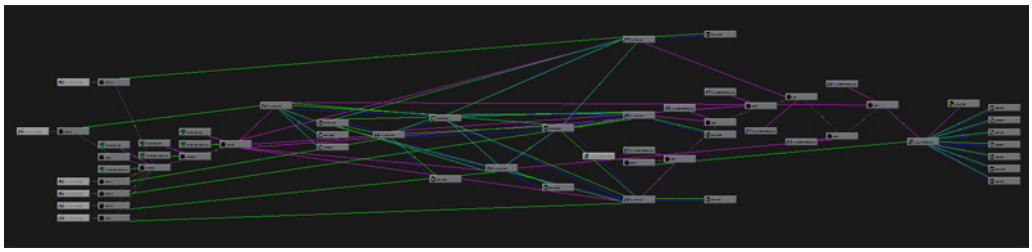


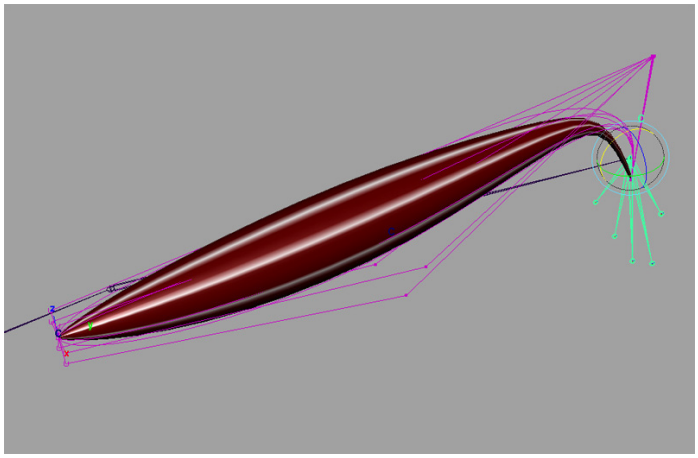
Figure 3_3_16 This is a dependency graph representation of the "muscle we just created.

8. Try rotating the joints.

In Figure 3_3_16 we can see the dependency graph of the structure we've just created. Now is the time to check how the Muscle deforms. Figure 3_3_17 shows what happens when rotating the hand joint. Actually the wire deformers are doing exactly what we've asked them to do. They rotate with the joints. This however is certainly not the effect we want!

Figure 3_3_17

Bad deformations. The second curve's CVs from the end rotate around the hand joint rather than point towards the start of the muscle.



Setting the muscle to behave properly.

What we need is for both poles to point each other with the end pole (the one on the hand side) turning only on the twist axis. Once again we will use the Y axis for twisting. As usual it is important to get consistent rotations on the twist axis but not on the other two axes.

We are going to aim constraint two nodes, each for each pole, at each other. I will refer to these nodes as 'manipulators'. To secure consistent up axis we will use object to point where the up is. We will first take care the two manipulators are pointing at each other on all three axes. Only later we will set the Y axes. I got better result when handling three axes and than compensating for one then when trying to pipe two axes to one node and third to another. The setup we'll have here is somewhat more elegant.

For clarity we'll assume that we set a forearm muscle. We have a chain that starts with an arm joint, a forearm joint and a hand joint.

1. Create four manipulators.

We are going to use these manipulators to make the two poles to point at each other. Two manipulators will control the clusters directly while the other two manipulators will be used to keep consistent up axis and get the twist to work.

The manipulators can be simple nulls (empty groups) but it is more comfortable to use locators so you'll have visual feedback*. For this demonstration I've ionized the manipulators for more clarity.

2. Position two of the manipulators at the poles of the muscle.

- Click the muscle to select it.
- Show the muscle CVs. **Display → NURBS → Components CVs**.
- With the **v** key pressed snap one manipulator to each pole.

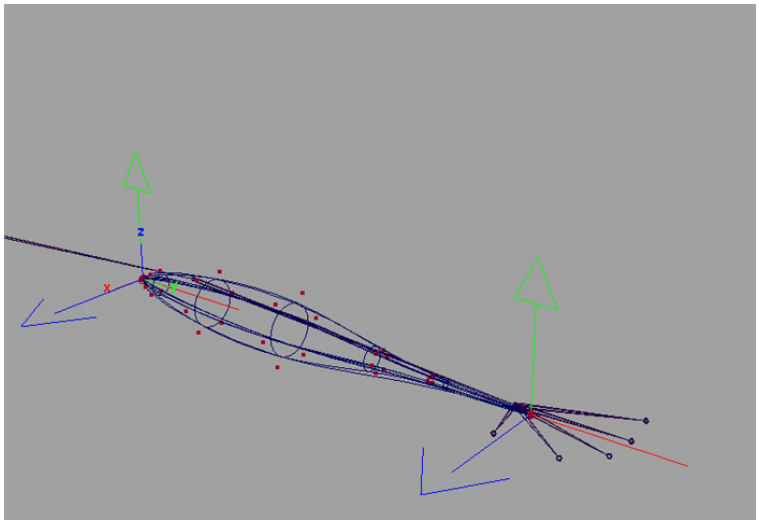


Figure 3_3_18 Position manipulators at muscle poles.

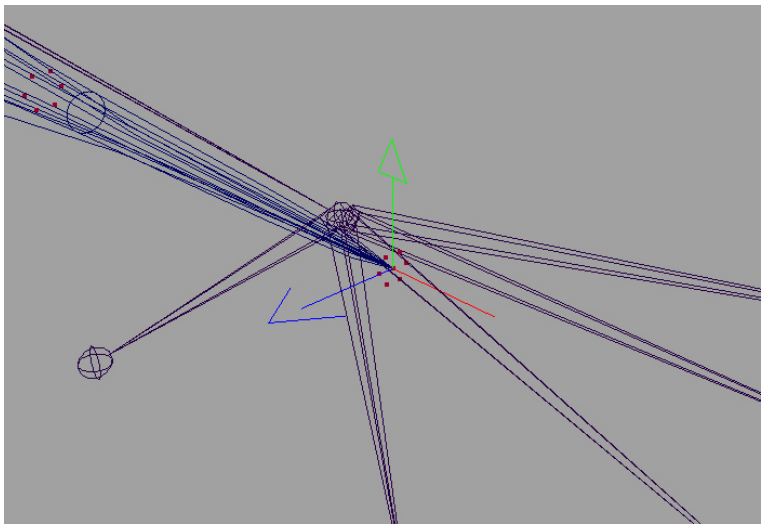


Figure 3_3_19 Use muscles CVs to position the manipulators.

* Tip, you can use locator and after finishing the setup delete the shape node so you'll end up with a neat setup.

3. Parent the two manipulators to joints.

   Parent the manipulators you've just positioned in the same way that we've parented the clusters in previous examples.

   - Click select the base manipulator.
   - Shift select the arm joint.
   - **Edit → Parent** (or just press **p**).
   - Click select the end manipulator.
   - Shift select the hand joint.
   - **Edit → Parent** (or just press **p**).

4. Position the other two manipulators above the first two manipulators, one for each.

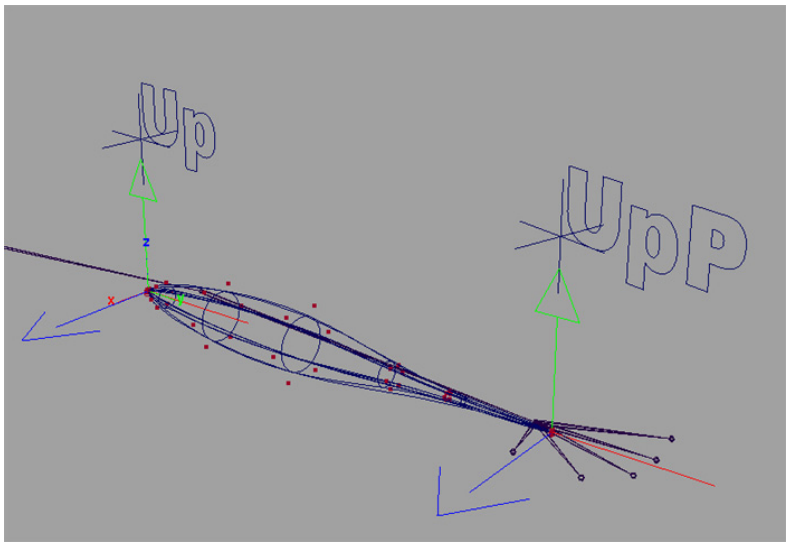


Figure 3_3_20 Position up manipulators above the manipulators.

5. Parent the other two manipulators to joints.

   These manipulators, let's call them up manipulators, will be used us up axis. While, for now, we will parent the end up manipulator to the hand joint, the same way we've parented the end manipulator, the base up manipulator should be parented to the forearm joint.

   - Click select the base up manipulator.
   - Shift select the forearm joint.
   - **Edit → Parent** (or just press **p**).
   - Click select the end up manipulator.
   - Shift select the hand joint.
   - **Edit → Parent** (or just press **p**).

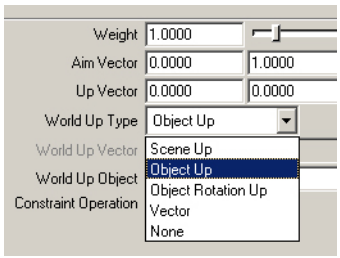
6. Constraining the manipulators.

We will now constrain the two manipulators to each other. Since the arm will rotate in the animation if we will try to use the world up axis as the manipulators up axis at certain angles we will have flipping problems. To avoid this we will use the other two objects as up axes that will move with the arm.

- From the menu bar **Constrain → Aim → Option Box**.
- Set **Aim Vector** to 0 1 0. This will set the constrain to point at the positive Y axis.
- Set **Up Vector** to 0 0 1. The up vector will be positive Z.
- Set **World Up Type** to **Object Up**.

Figure 3_3_21

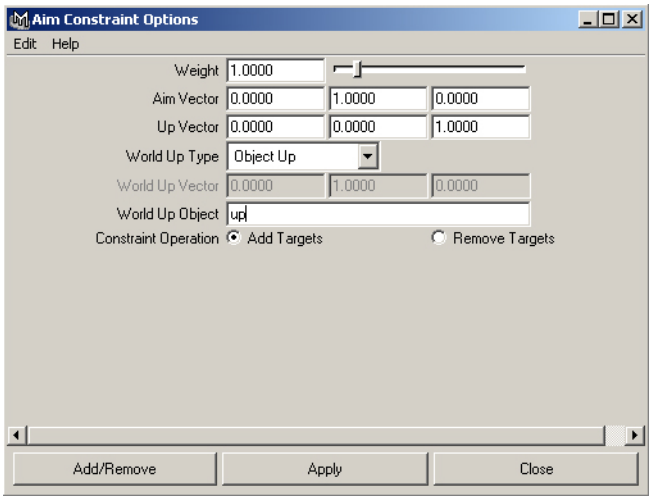
Aim constraint option window.
Set up vector to object up.



- Type the name of the start up manipulator in the **World Up Object** field.
- Click select the end manipulator.
- Shift select the start manipulator.
- Press **Apply** in the aim constrain option window.

Figure 3_3_22

Aim constraint option window.



- Type the name of the end up manipulator in the **World Up Object** field.
- Click select the start manipulator.
- Shift select the end manipulator.
- Press **Add/Remove** in the aim constrain option window.

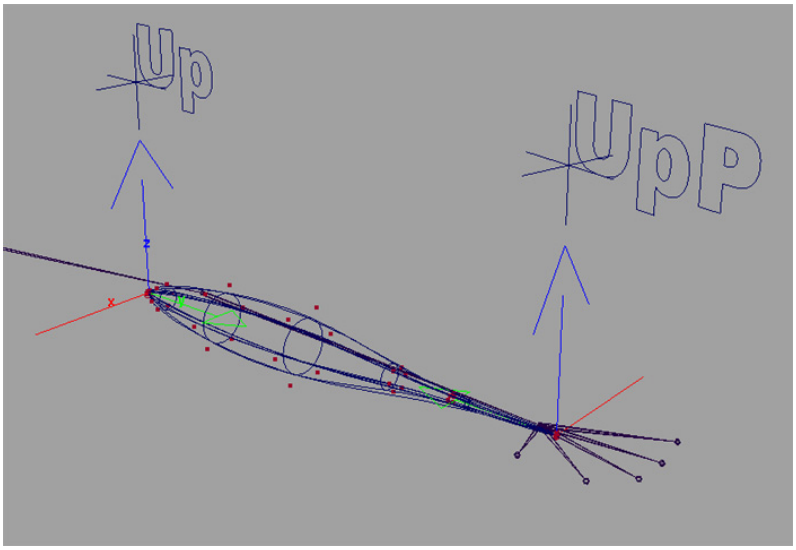


Figure 3_3_23 Oriented manipulators.

7. Add control object for the middle cluster.

   Since the middle cluster needs to rotate as well, we need to add control object for this cluster as well.

   - From the menu bar **Create → Locator**.
   - Shift select the forearm joint.
   - **Edit → Parent** (or just press **p**).
   - With the locator still selected set all translates and rotates to 0 in the **Channel Box**.
   - Double click on the **Move Tool** icon and set **Move Settings** to object.
   - Drag the locator until it'll be approximately in the middle of the joints, close to the center cluster.

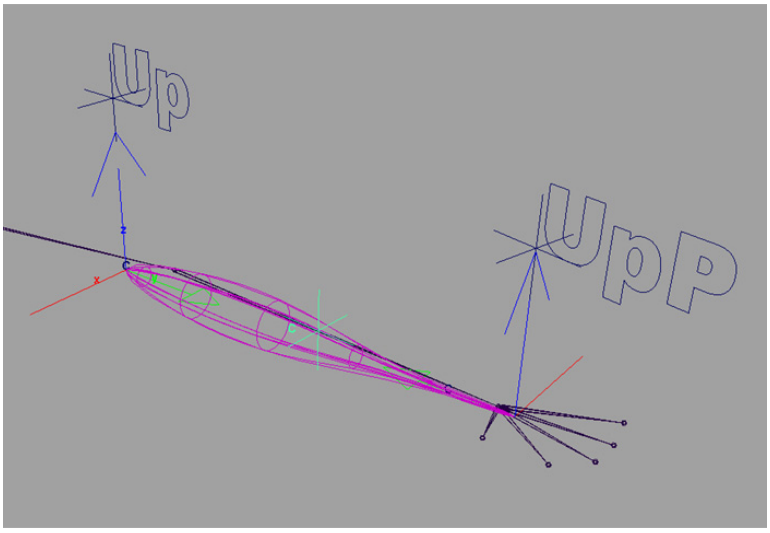


Figure 3_3_24 Middle locater.

8. Connect the clusters to the manipulators.

Parent each of the clusters to its corresponding manipulator.

- Select the base cluster.
- Shift select the base manipulator.
- **Edit → Parent** (or just press **p**).
- Select the middle cluster.
- Shift select the locator at the middle of the forearm joint.
- **Edit → Parent** (or just press **p**).
- Select the end cluster.
- Shift select the end manipulator.
- **Edit → Parent** (or just press **p**).

Let's try and see what we have now. First try rotate the arm and forearm joints. Indeed the start manipulator points towards the end one. Now let's try to twist the hand. We have twist on the muscle, sort of, but it only starts in the middle of the muscle. This is because the middle cluster does not rotate. When we try to rotate the hand on the Z axis the end manipulator flips and the muscle breaks. This is because we've parented the end up manipulator to the hand. While we do desire the end up manipulator to rotate with the hand on the Y axis this is not the case for the other two axes.

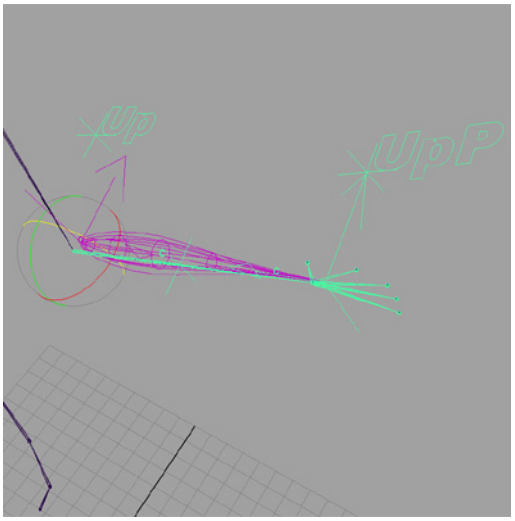


Figure 3_3_24 (left)

The start manipulator responds as expected.

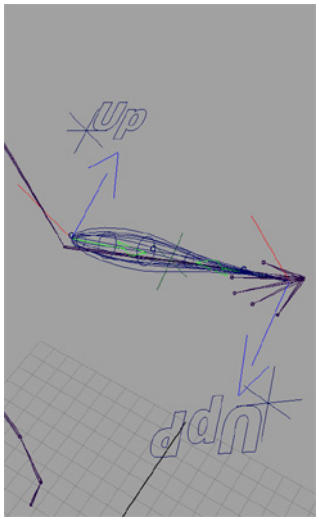


Figure 3_3_25 (right)

As we rotate the hand on the Z axis the end manipulator flips.

We will now set the end up manipulator to rotate with the hand only on the Y axis. We will also set the middle locator that holds the middle cluster to rotate half way with the hand.

9. Set the end up manipulator to turn only on the Y axis.

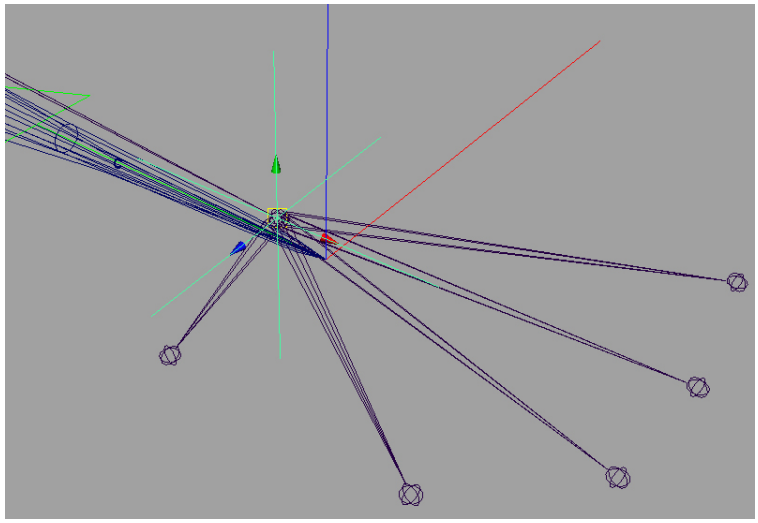
Theoretically what we want is to cancel the X and Z rotations for the end up manipulator. However, since the moment you break them it is hard to anticipate rotations and get consistent single axis rotations I believe it is better to deal with one axis.

We will create a locator at the position of the hand joint and set it to rotate only on the Y axis with the hand.

- From the Menu Bar **Create → Locator**.
- With the **v** key pressed snap the locater to the hand joint.

Figure 3_3_26

Placing the locator at the hand joint's position.



We can now parent the locator to the forearm and set its rotations to 0 0 0 but this wont give us the desired results. Joints in Maya are different then other transforms because they have the joint orient attribute. To properly orient a transform after a joint we need to parent the transform to a null[*].

- From the Menu Bar **Edit → Group**.
- Press the **Insert** key.
- With the **v** key pressed snap the null's rotation pivot to the hand joint.
- Shift select the forearm joint.
- **Edit → Parent** (or just press **p**).
- With the null still selected set all rotates to 0 in the **Channel Box**.

[*] Another way to get this is to set, instead of the locator, an additional joint. It is just a matter of personal preference.

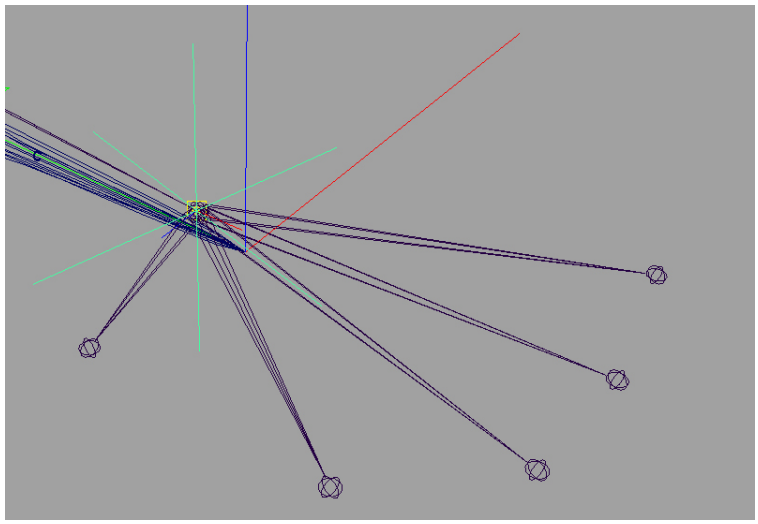
Now we need to set the rotations of the null to the joint orients of the hand joint. While we can copy paste the values it will be more accurate to do that from command line.

```
// assuming that the hand joint is named handJoint
// and the null is named group1
//
float $rot[3] = `getAttr handJoint.jo`;
setAttr group1.r $rot[0] $rot[1] $rot[2];
//
```

The locator is now properly oriented.

Figure 3_3_27

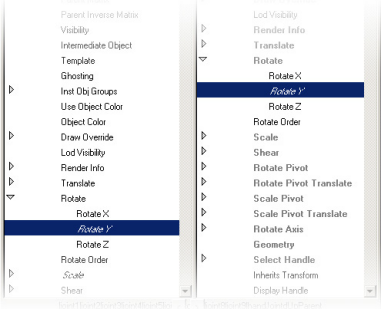
Orienting the null
You can notice that the locator now is properly oriented.



- Click select the locator.
- Shift select the hand joint
- **Window → Hypergraph.**
- In the **Hypergraph** window press the **Up and Downstream Connections** button
- **Middle mouse drag** the hand joint to the locator.
- In the **Connection Editor** connect the RotateY attribute of the hand joint to the RotateY attribute of the locator.

Figure 3_3_28

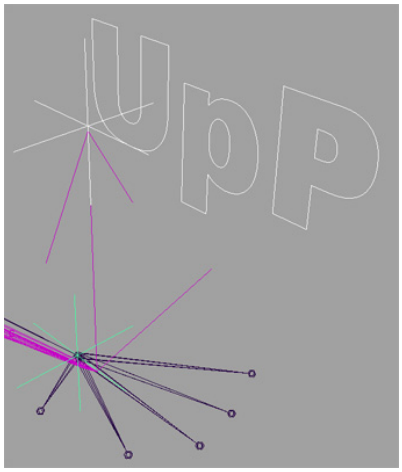
Connecting only Y rotations of the locator to Y rotations of the hand joint.



- Buck in the workspace click select the end up manipulator.
- Shift click the locator.
- **Edit → Parent** (or just press **p**).

Figure 3_3_29

Parent the end up manipulator to the locator.



Try and rotate the hand joint now. Only when you rotate the Y axis the end up manipulator rotate.

Figure 3_3_30

Rotating the hand on the Y axis. The end manipulator twist.

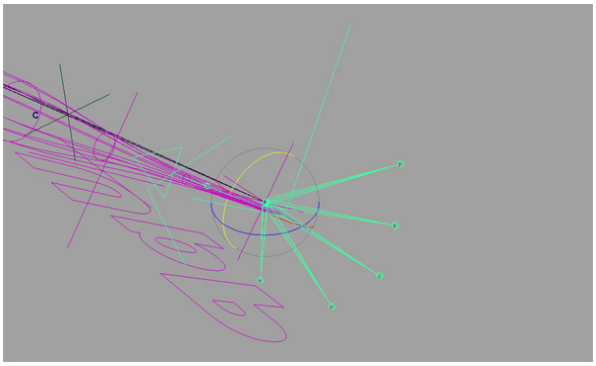
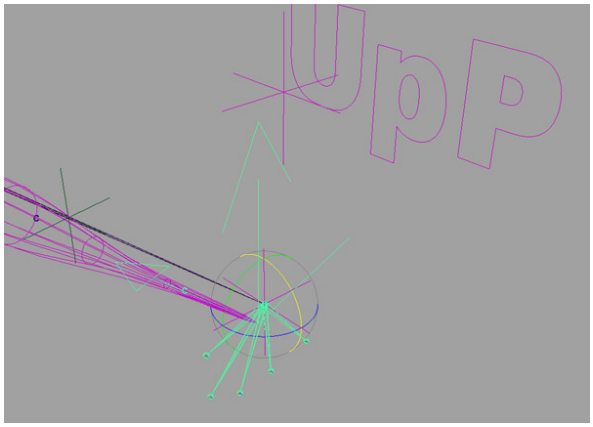


Figure 3_3_31

Rotating the hand on the Z axis. The end manipulator move properly and do not twist.



10. Set the middle locator to rotate half way.

    We will divide the Y rotation of the hand joint in half and apply the result to the middle locator.

    The easiest way to do this is to write it as an expression.

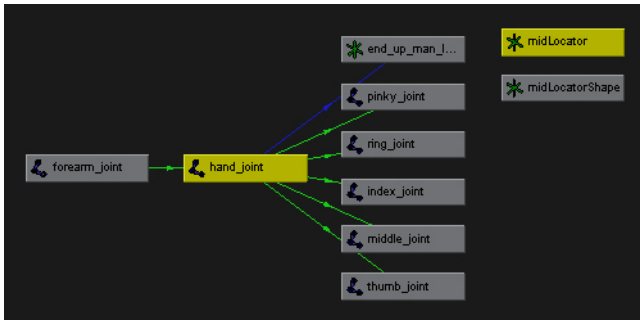
```
//
midLocator.rotateY = handJoint.rotateY / 2
//
```

    However, once again, using binary nodes will get the setup to work faster.

- Click select the hand joint.
- Shift select the mid locator.
- **Window → Hypergraph.**
- In the **Hypergraph** window press the **Up and Downstream Connections** button

Figure 3_3_32

This is how the Hypergraph should look after you've pressed the Up and Downstream Connection button.

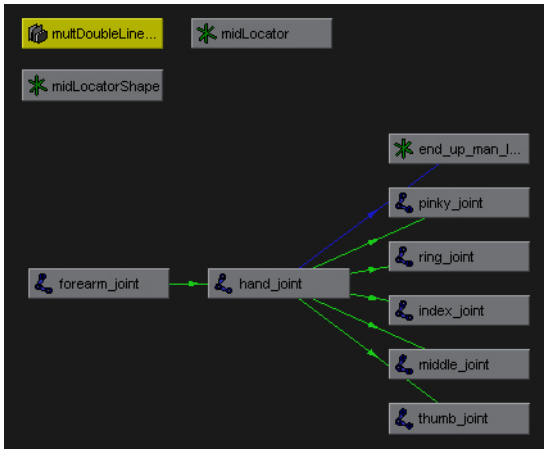


- In the **Script Editor** type:

```
//
createNode multDoubleLinear;
// Result: multDoubleLinear1 //
```

Figure 3_3_33

Same window as 3_3_32 after adding the multDoubleLinear.

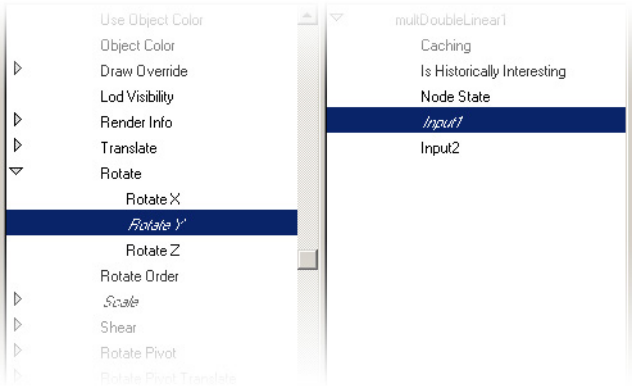


This node multiply two inputs.

- **Middle mouse drag** the hand joint to multDoubleLinear node.
- In the **Connection Editor** connect the RotateY attribute of the hand joint to the Input1 attribute of the multDoubleLinear node.

Figure 3_3_34

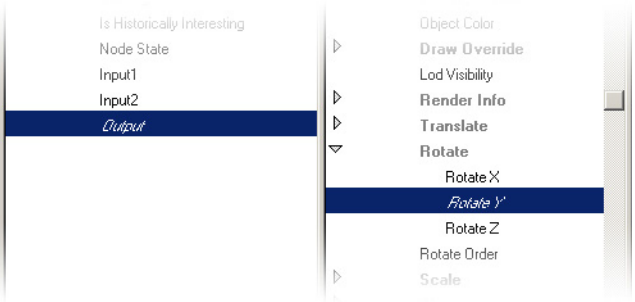
Connect the Y rotation of the hand joint to input 1 of the multDoublelinear node.



- **Middle mouse drag** the multDoubleLinear to mid locator node.
- In the **Connection Editor** connect the Output attribute of the multDoubleLinear to the RotateY attribute of the mid locator.

Figure 3_3_35

Connect the output of the multDoubleLinear node to the Y rotation of the mid locator.



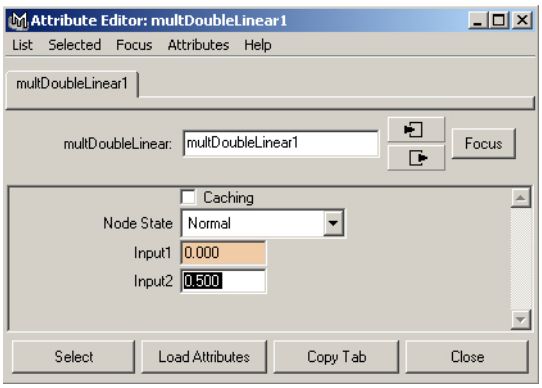
- In the **Hypergraph** window click select the multDoubleLinear.
- Open the **Attribute Editor**.

Now we will set the multDoubleLinear node to divide the rotation of the hand joint in half.

- Type 0.5 in the Input2 field.

Figure 3_3_36

The attribute window of the multDoubleLinear node.



- In the **Hypergraph** window click select the multDoubleLinear.
- Press the **Up and Downstream Connections** button

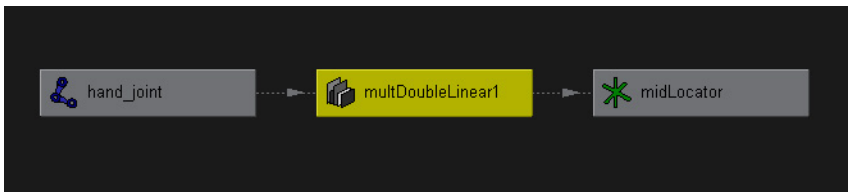


Figure 3_3_37 After finishing the connections this is how the Hypergraph should look.

11. Keeping the muscle's volume.

    Unfortunately when we set multiple wire deformers to influence specified CVs we can not use the scale attribute of the wire node. There are several solutions, none of them as elegant. The easiest way I've found is to scale the middle cluster deformer. We still need to subdivide the original muscle length with the length at any given frame.

    Though we can generate this information in the same way we did before, using a curveInfo node attached to any one of the wire deformer curves (or using a surfaceInfo node directly from the muscle surface) we will not be able to scale the cluster successfully using this information. Since both the curves and the muscle are being controlled by the clusters, trying to use any of them to control these clusters will result in a loop. Instead we will generate the information directly from the start and end manipulators.

    - In the Script Editor type:

```
//
createNode distanceBetween;
// Result: distanceBetween1 //
```

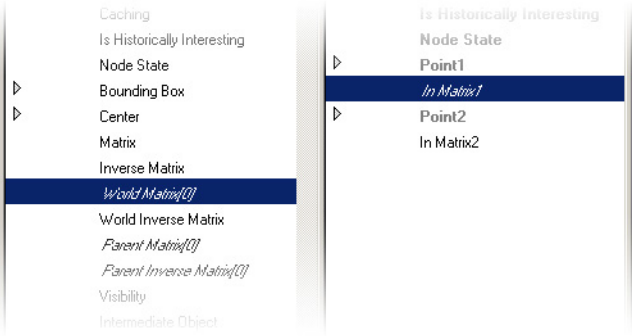
- In the workspace shift click the start manipulator.
- Shift click the end manipulator.

You should have now three objects selected, the two manipulators and the distanceBetween node.

- In the **Hypergraph** window press the **Up and Downstream Connections** button.
- **Middle mouse drag** the start manipulator to the distanceBetween node.
- Connect the worldMatrix attribute of the start manipulator to the inMatrix1 attribute of the distanceBetween node.

Figure 3_3_38

Rather than connecting translates we connect the world matrix of the transform to the un matrix of the distanceBetween node. This will get as world space coordinates.



This is an easier way to get the world space coordinates of the start manipulator. Since it is parented to the joints we can not take it's translate attributes because they do not represent the manipulators real coordinates and do not change as we rotate the joints.

In an expression we will use a line that should look something like this:

```
//
float $pos[3] = `xForm -q -ws -t startMan`;
//
```

- Repeat for the end manipulator, connecting it to the inMatrix2 attribute of the distanceBetween node.
- **Right Mouse Button** in the **Hypergraph**
- **Rendering → Create Render Node**
- Click on the **Multiple Divide** button to create a multipleDivide node.
- Open the **Attribute Editor** window.
- Change the operation to **Divide**.
- Back in the Hypergraph **middle mouse** drag the **distanceBetween** node to the **multiplyDivide** node.
- Connect the distance attribute of the **distanceBetween** to the input2X attribute of the **multiplyDivide** node.

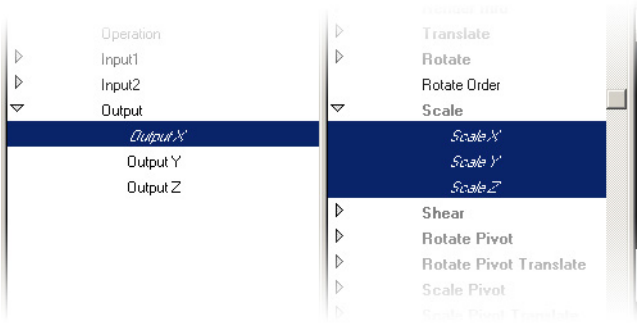
- In the **Script Editor** type:

```
//
setAttr multiplyDivide1.input1X
`getAttr distanceBetween1.distance`;
//
```

- In the Hypergraph **middle mouse** drag the **multiplyDivide** node to the middle cluster.
- Connect the outputX attribute of the **multipledivide** node to all three scale attributes of the cluster.

Figure 3_3_39

We use the outputX of the multipleDivide node to control the scales of the middle cluster.



You should have now something that looks like Figure 3_3_40

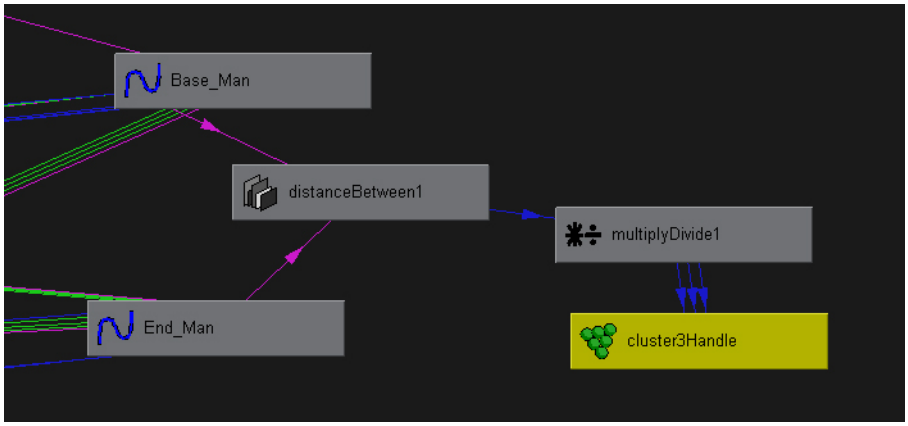


Figure 3_3_40 The Hypergraph tree structure we should get at the end of the process.

Variation – A lattice muscle.

A more elegant way to set a similar muscle is to use lattice deformer instead of the multiple wire deformers. This setup is much easier and faster to set, in twists rotations however the results might be less accurate.

For this setup set the manipulators just as we did in the previous example.

- In the Workspace click select the muscle.
- From the menu bar **Deform → Create Lattice → Option Box**.
- In the lattice option window set **Divisions** to 2 3 2*.

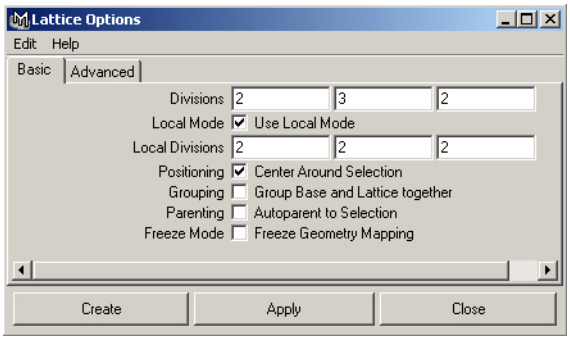
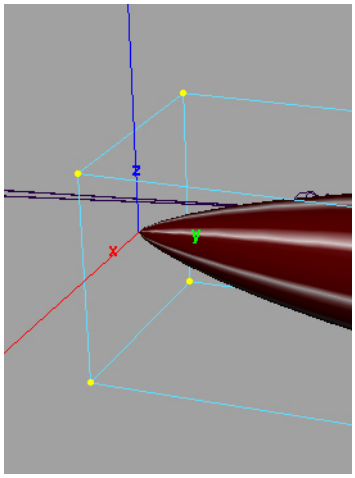


Figure 3_3_41 Lattice creation option window

- Drag select the four lattice CVs that are closest to the start manipulator.

Figure 3_3_42

Select four lattice CVs

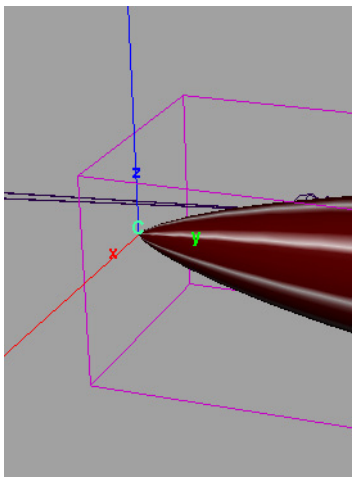


* The axis that have the 3 division might change based on the orientation of the NURBS sphere. If Y is used as the twist axis the Y division should be the 3. If you are scripting this operation it is important to know in advance what will be the axis that will take the 3 division, hence another good example why it is important to properly and consistently orient everything and not only the joints.

- From the menu bar **Deform → Create Cluster**.

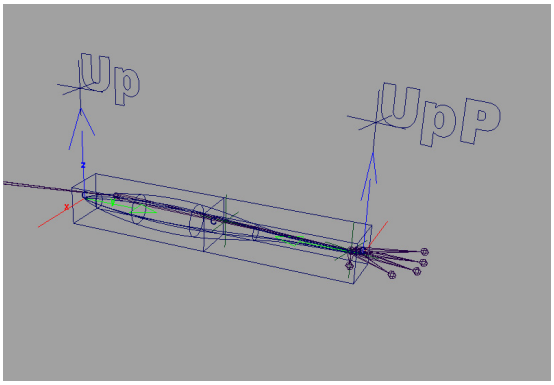
Figure 3_3_43

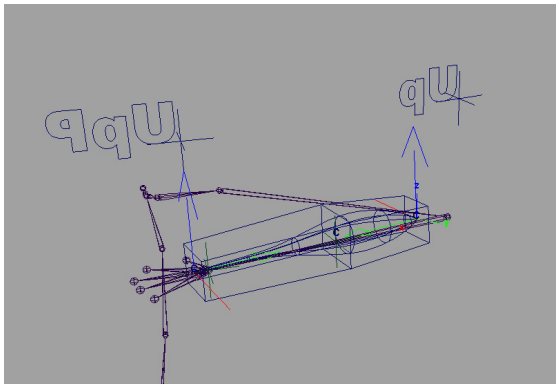
Create cluster to control the lattice CVs.



- **Parent** the cluster to the start manipulator.
- Repeat for the middle locator and end manipulator, setting clusters for four lattice CVs each.
- Connect the outputX of the **multiplyDivide** node to the scales of the middle cluster.

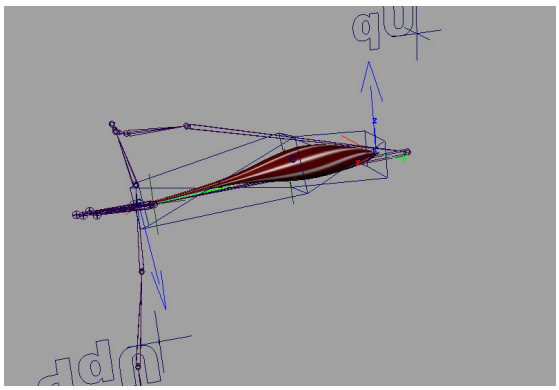
Figures 3_3_44-46 shows the lattice muscle in different states. The bulging effect is more noticeable on the lattice deformer in 45 and 46.





Figures 3_3_44-46

The lattice muscle in different states.



4. Non spherical muscles.

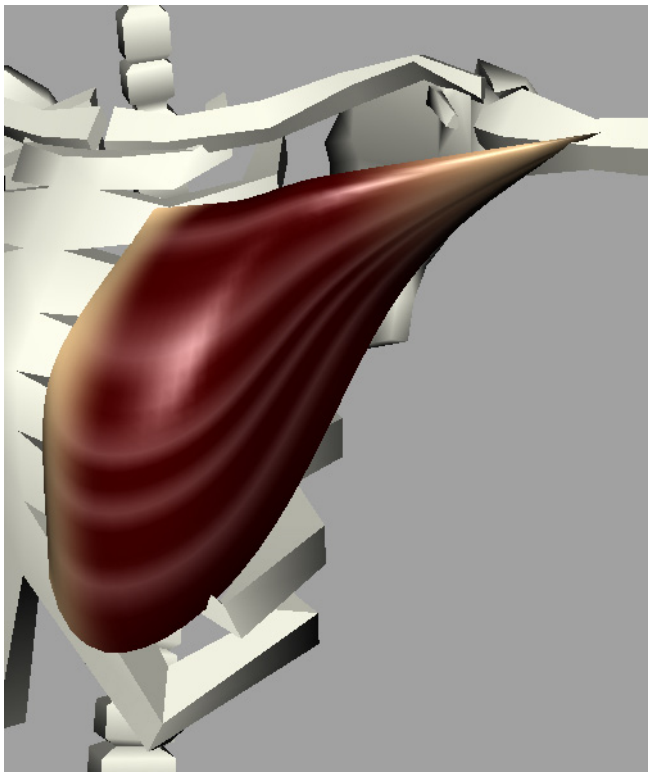


Figure 3_4_1

Not all muscles shapes derive from spheres. In some cases muscles takes different shapes. Another case is when we might want to merge several muscles into one muscle. There is a good reason to do that. Once start to attach muscles to a skin you'll discover that your rig is becoming increasingly heavy and slow to manipulate. I believe that even on a very fast computer it is impractical to animate even a relatively simple muscle rig, and a multi rig approach is preferable, so it's not so much a matter of fast respond to animation. However setting the rig on itself is a time consuming process and I always believe it trying to keep things as simple as possible. Merging several muscles into one will usually end up in a simpler, faster setting, rig.

Building a chest muscle

The chest area is a good example. While we can get away with a group of muscles of the type described before this setup is unnecessarily slow and cumbersome. Instead we can sculpt a simple NURBS patch to fit in neatly.

1. Creating the muscle's surface.

   When curving a muscle it is extremely important to have good topology (this rule applies, of course, for every deformable surface, such as skin geometry). The topology has to follow the lines that the muscle will stretch. In 3_4_2 we can see the topology of the muscle.

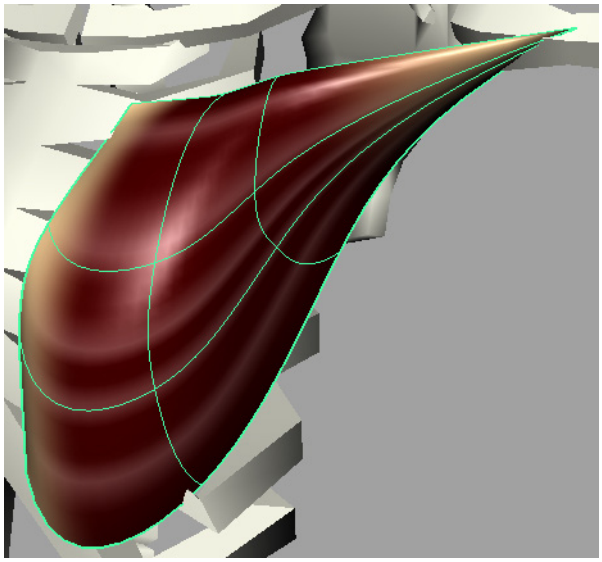


Figure 3_4_2 The topology of the muscle has to follow the lines it will deform.

   In this case the muscle stretches in two directions from the arm pit towards the center of the chest and the belly. In this case the V isoparms follow exactly these directions.

2. Building the control curves.

   To control the muscle we will use again several wire deformers.

   To generate the curves for the wire deformer you can use the same technique from the multi wire muscle. Select the V isoparms of the muscle (3_4_3), duplicate them using the duplicate surface curve command (3_4_4) and rebuild them (3_4_5). Since this type of muscle is more customized based on specific cases the process is less formulated so it is harder to answer to what degree should the curve built or how many spans should it have. Still the 'entire muscle has to respond to every manipulation' rule applies. It is therefore make sense to build the curve with one (if you plane to hold the edges with one CV) or two (in the twist case with two extreme CVs) spans. Unlike the sphere muscles linear curves are less useful in this case. Usually a degree 2 or 3 should work fine.

Figure 3_4_3

Select the V isoparms of the muscle's surface.

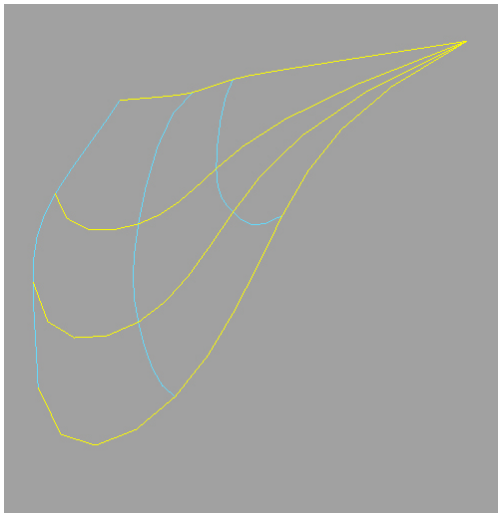


Figure 3_4_4

Using the duplicate surface curves command duplicate the curves.

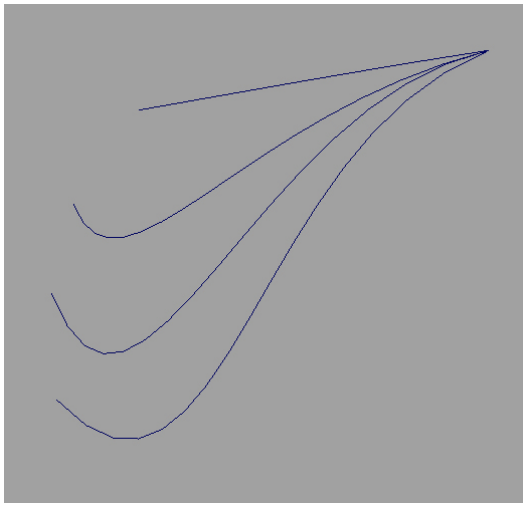
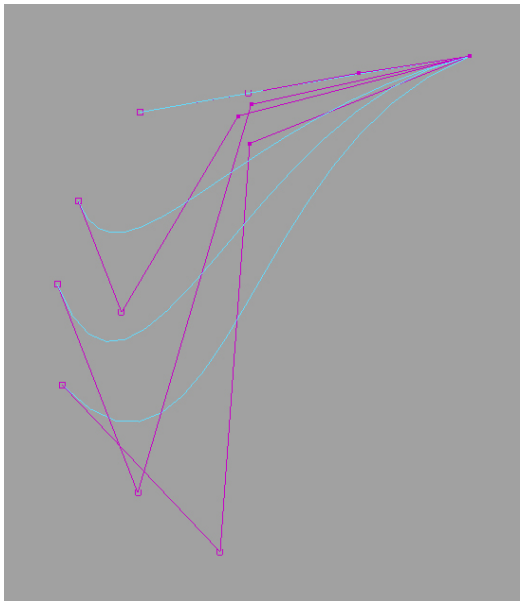


Figure 3_4_5

Rebuild the curves to a simpler shape.



3. Connecting the curves to the surface.

Just as in the case of the spherical muscles we will use the curves as wire deformers.

Generally there are two approaches for using multiple wire deformers:

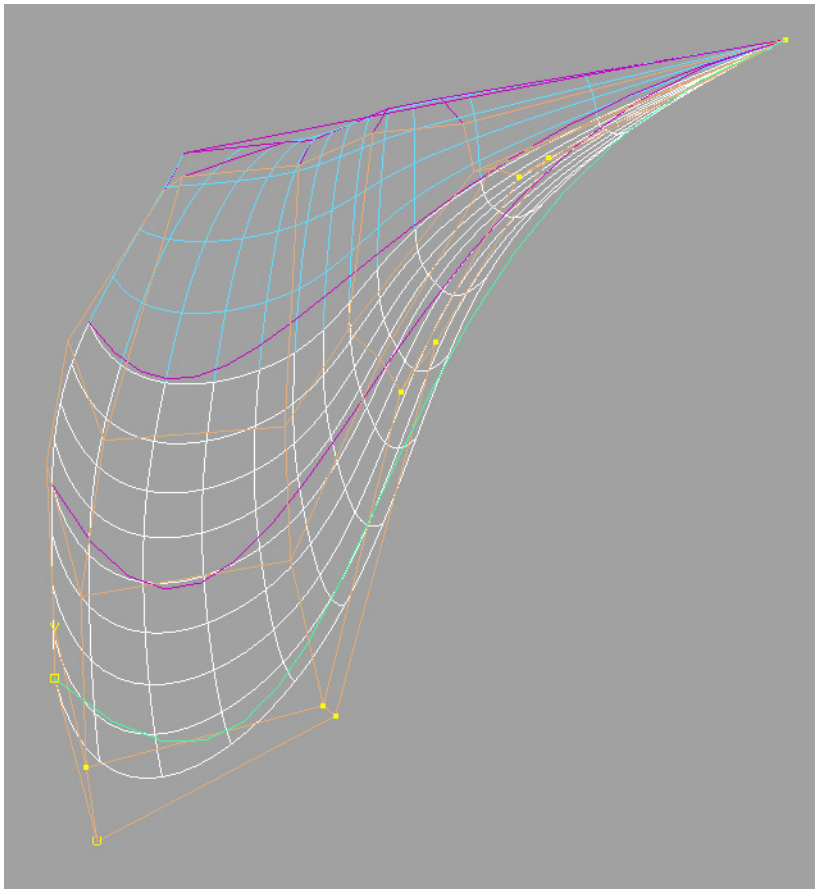
a) Apply all the wire deformers to the entire surface and then balance them by weighting.
b) Apply wire deformers per components with every component influenced by one deformer.

I usually prefer the second approach. I feel it gives me more of a one on one results, thus behave more predictably.

Choosing which CVs connect to which curve is based on the distance between entire CV rows and curves. Since the muscle's surface is open we will have two additional CV rows compare to the number of isoparms. In the case of this example I've attached two rows of CVs to each of the two extreme curves (Figure 3_4_6)

Figure 3_4_6

Connecting CVs to curves is done based on distance. In the pictures the first two rows of CVs (in yellow) will be connected to the first curve (green).

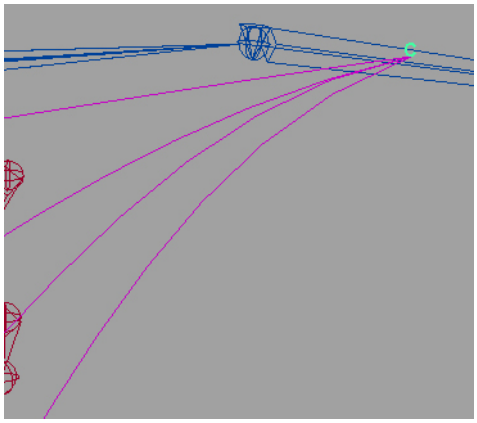


4. Connecting the muscle to the skeleton.

Again done with clusters. Once again there are no rules and the process is basically trial and error per case. It is important to note however that there is important relationship between the position and number of CVs in the control curves (the wire deformers) and the position of the clusters. Don't expect to lock the position of the CVs and than simply connect the clusters. In most cases after connecting the clusters and checking you will have to go back to the curves and readjust them.

Figure 3_4_7

Clusters connect the curves to the joints.



5. Rotate the joint to test the muscle.

As is clearly seen in 3_4_9 the muscle penetrates the rib bones and fails to maintain the body's volume.

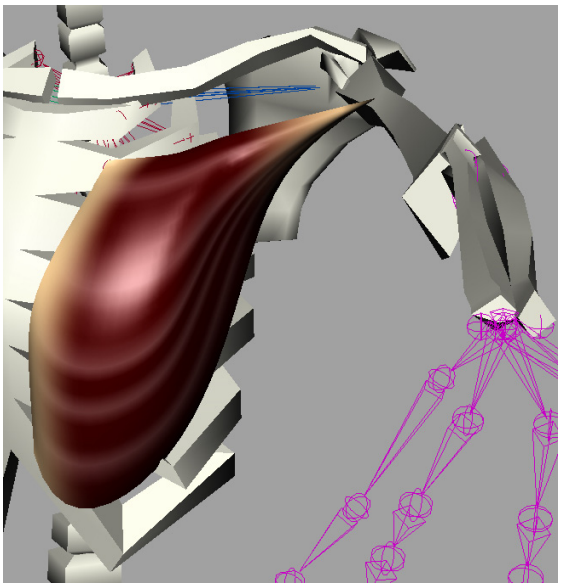


Figure 3_4_8 Testing the muscle

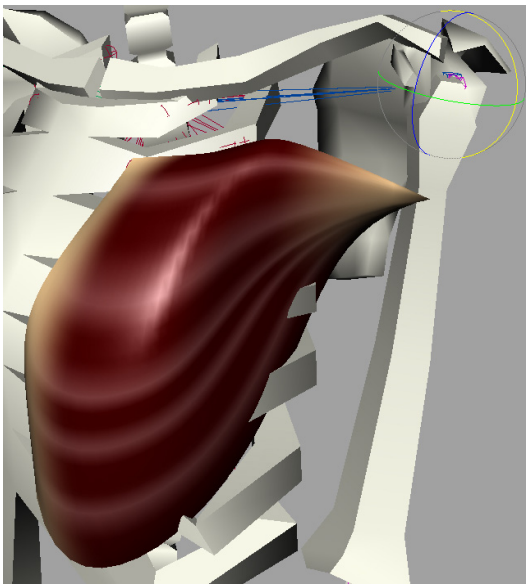


Figure 3_4_9 Testing the muscle

5. Getting muscles to collide with bones.

Note: the technique described here will work fine for the following example but still have many limitations. Generally speaking this technique works for relatively flat surfaces. The more rounded the collision surface the less this process succeeds.

What we want is to make the muscle collide with the ribs. Colliding means that as long as the muscle does not touch the ribs it moves freely. Once the muscle touches the ribs it should slide on them.

We will incorporate two techniques. One is the technique described in Jason Schleifer's notes for this course, under the Limit Notification chapter. Essentially this is a technique to determine on what side of a NURBS surface a transform is. Jason has an excellent explanation of this technique in his notes and therefore I will not repeat it here.

The other technique is a bit similar to the way deformers works in Maya. Deformed shapes have base shapes with the final result stems from these base shapes.

To get the curve to collide with the surface we duplicate it. The original curve will not collide with the surface but will generate the information needed. The second curve will be the one that will collide with the surface and it motion will be our final output.

So how it works?
The first curve moves freely in the world. For every point in this curve we check at what side of the surface it is. This information is piped into a condition node. In addition we check for every CV what its closest point on the collision surface is. The condition node has two options. Either move a CV in the second (collision) curve to the position of its corresponding CV in the first curve or send it to the closest point to the on the collision surface that belongs to the corresponding CV in the first curve.

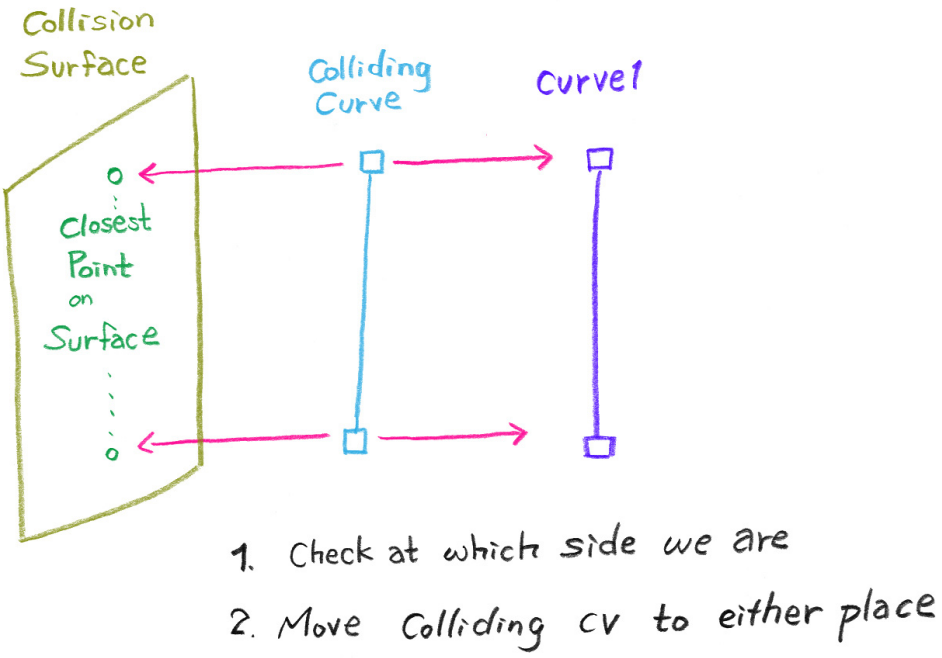
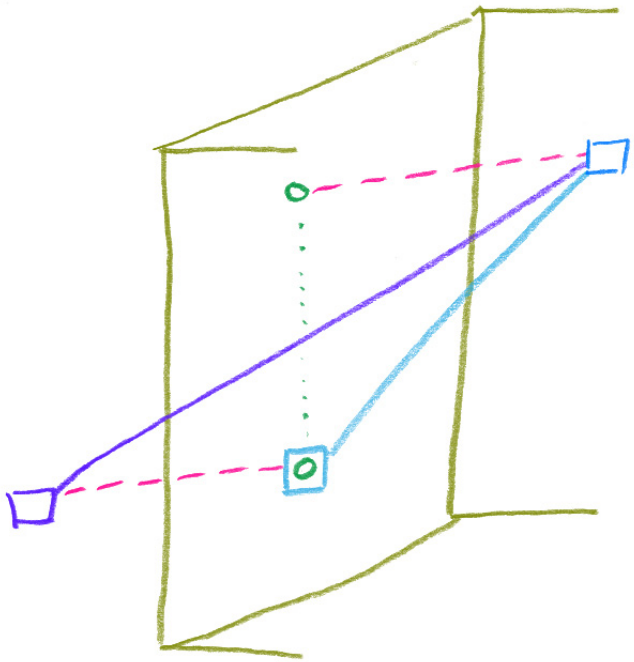


Figure 3_5_1

Here are the relationships between the objects. The dark blue curve is the original curve. The collision surface is in light green. For every CV in the original curve we find its closest point on surface (dark green). Each CV in the colliding curve follows either its corresponding CV in the original curve or its closest point on surface.

Figure 3_5_2



And here's how it works. The dark blue curve is the original curve. The collision surface is in light green (the four lines at the edges are the normals). The rounded dark green points are the closest points on surface, connected with red dotted lines to their corresponding CVs. If the CVs are on the outside of the surface the CVs of the light blue collision curve stick with the original curve. If the CVs of the original curve are on the inside of the surface the collision curve CVs stick with the closest point on surface.

I will split this stage to two phases, determining where a CV is relative to a surface and how to control the colliding CV.

1. Where a CV is?

   This is just a brief reminder. For a thorough description see the 'Limit Notification' chapter in Jason Schleifer's section of this course.

   - Connect **curveShape.worldSpace[0]** to a **curveInfo.inputCurve**
   - Connect **curveInfo.controlPoint[*]** to **closestPointOnSurface.inPosition** and **nurbsPlanShape.worldSpace[0]** to **closestPointOnSurface.inSurface**.
   - Connect **closestPointOnSurface.parameterU** and **.parameterV** to **pointOnSurfaceInfo.parameterU** and **parameterV** and **nurbsPlanShape.worldSpace[0]** to **pointOnSurfaceInfo. inSurface**.
   - **Connect curveInfo.controlPoint[*]** to **plusMinusAvarage.input3D[0]** and **pointOnSurfaceInfo.position** to **plusMinusAvarage.input3D[1]**.
   - Set the **plusMinusAvarage** node's operation to subtract.
   - Connect **plusMinusAvarage.output3D** to **vectorProduct.input1** and **pointOnSurfaceInfo.normal** to **vectorProduct.input2**.
   - Set the **vectorProduct** node's operation to dot Product and set it to normalize output.

   The Hypergraph structure should be something like figure 3_5_3.

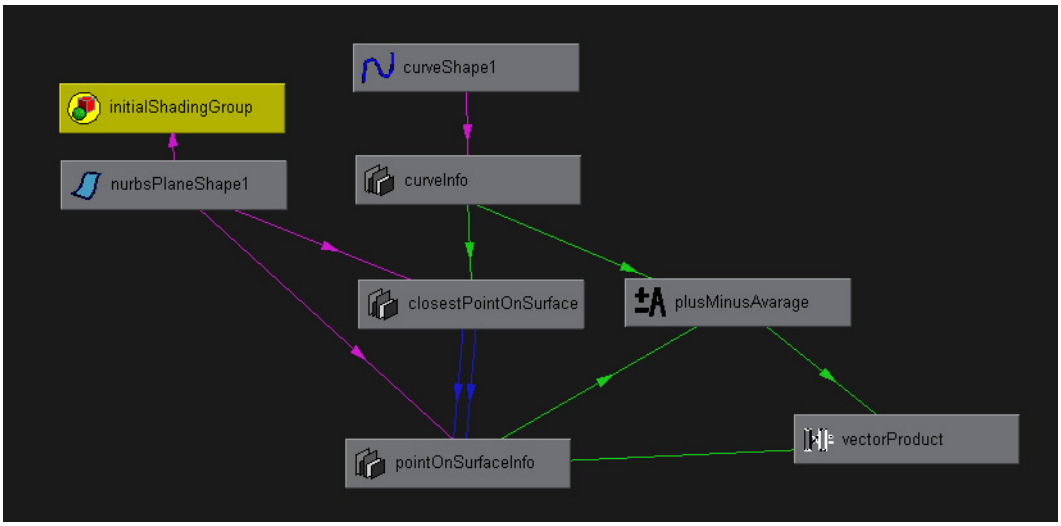


Figure 3_5_3 Hypergraph tree to determine on which side of a surface a CV is.

This phase results in a vectorProduct output that is either 1 or -1. Now to control the colliding curve:

2. Controlling colliding CVs.

We will now set a condition to control CVs of a new curve.

- In the workspace click select the NURBS curve.
- Form the menu bar **Edit → Duplicate** (leave the duplicate defaults).
- In the hypergraph **Right Mouse Button**.
- **Rendering → Create Render Node**.
- Under the **Utilities** tab in **General Utilities** click the **condition** button. This will create a condition node.
- Middle mouse drag the **vectorProduct** node on the **condition** node and connect **vectorProduct.output** to **condition.firstTerm**.
- Connect **curveInfo.controlPoint[*]** to **condition.color1**. This is the position of the original CV.
- Connect **pointOnSurfaceInfo.position** to **condition.color2**. This is the position of the CV as it collide with the surface.
- Connect **condition.outColor** to **curveShape2.controlPoint[*]** (the stars represent corresponding CV index numbers).
- Set the **condition** node's operation to **Greater Than** and leave second term to 0.000.

Figure 3_5_4

Settings for the condition node. We input world space coordinates for the colors. CVs positions of the first curve in color 1 and closest point on surface of each of those CVs in color 2. If the output of the vector product (which is in our case either 1 or -1) is greater than 0 the condition node will output color 1. if it is smaller than 0 the output will be color 2, thus either point on the original curve or on the surface.

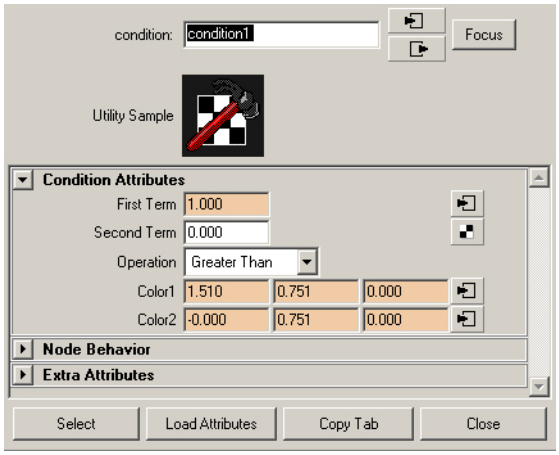


Figure 3_5_5 shows the hypergraph with the condition node.

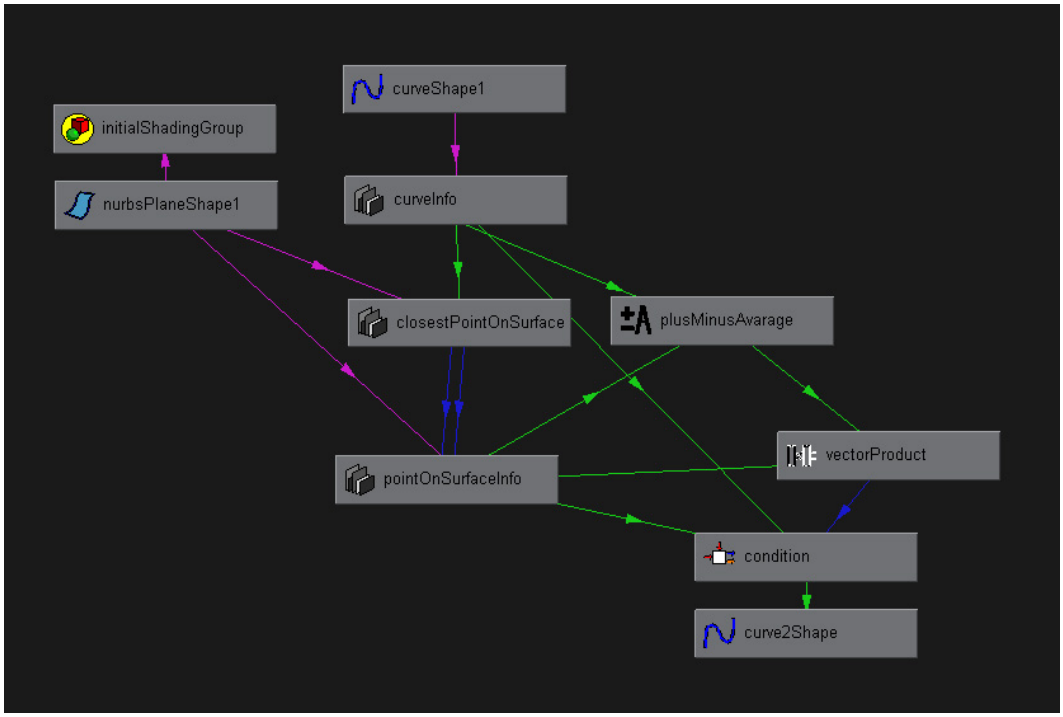


Figure 3_5_5 Hypergraph tree of controlling structure for the CVs.

The above description was referring to a single CV. We need to build the same structure for every CV in the curve. Obviously it is highly recommended to script the process. Figures 3_5_6 and 3_5_7 show the behavior of the curves after the setup is complete. The pink curve in 3_5_7 is the colliding curve which we will use as a wire deformer for the muscle geometry.

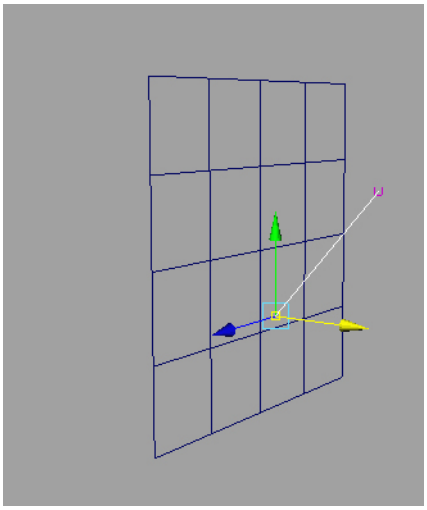


Figure 3_5_6 Curve before its lower CV collides with the surface.

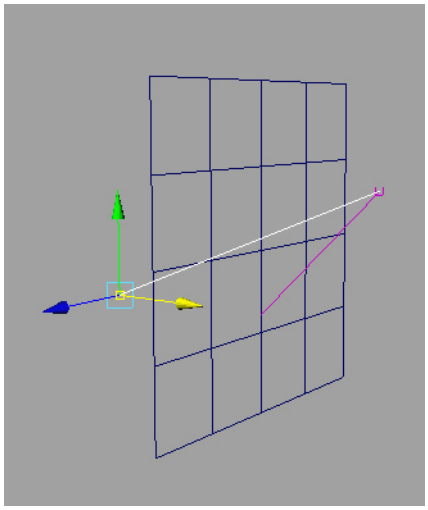


Figure 3_5_7 Curve after its lower CV collides with the surface.

Smoothing collision curves.

For a successful collision we need more than two points. However, the principle of using as few "holding" of control points as possible still applies. To overcome this conflict we need a structure such as the one presented at figure 3_5_8, in fact an improved version of 3_5_1 with an added layer. Instead of directly connect the original curve (curve 1) to the sliding curve we will add an intermediate curve, similar in shape to curve 1 but with more CVs. This new "info curve" will be used in the system in the same way that curve 1 was used, i.e. with the curveInfo, closestPointOnSurface etc. To connect curve 1 to the info curve we will use a wire deformer.

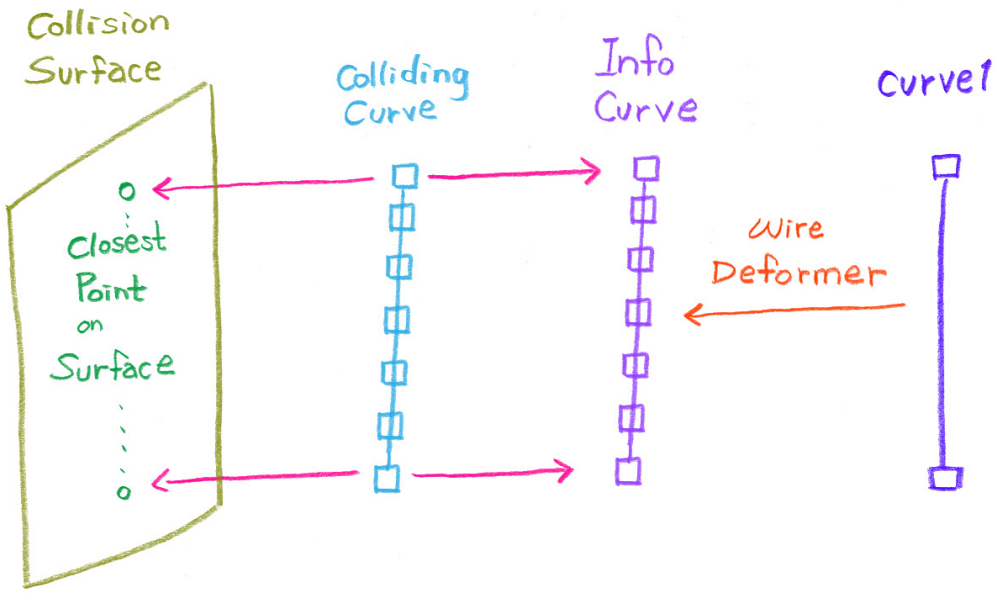


Figure 3_5_8 In this version The original curve (curve 1) does not control the colliding curve directly but rather is used as a wire deformer to control a similar curve but with more CVs. This "info curve" is the one that generate all the information that was originally taken from curve one. This information is used to control the colliding curve, which should have the same number of CVs as the info curve. The colliding curve should be used as a wire deformer for the muscle surface.

- Create a one span degree 1 NURBS curve and a NURBS surface.

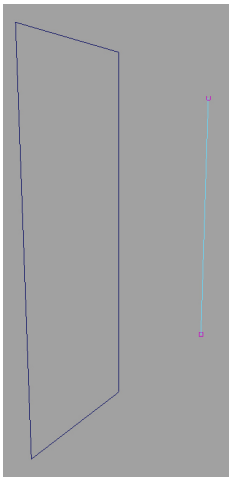


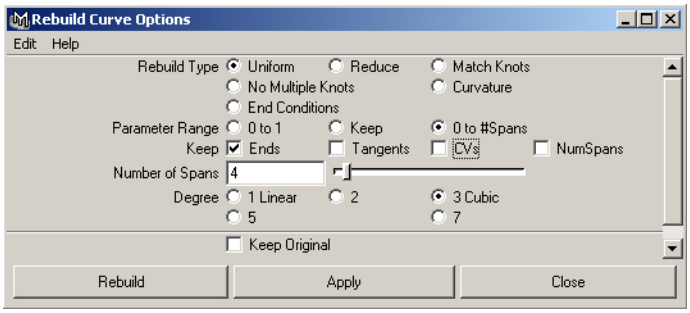
Figure 3_5_9

A degree 1 one span curve.

- Duplicate the NURBS curve.

Figure 3_5_10

Rebuild curve options set to a degree 3 with 4 spans. Try to avoid creating too many spans.



- Rebuild the curve to a degree 3 curve with as many spans as needed. How many spans will you need? This depends more on the complexity of the curve and the accuracy of the collision you need then the complexity of the surface. This technique is meant for fairly flat surfaces.

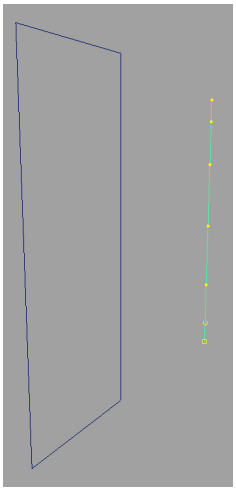


Figure 3_5_11

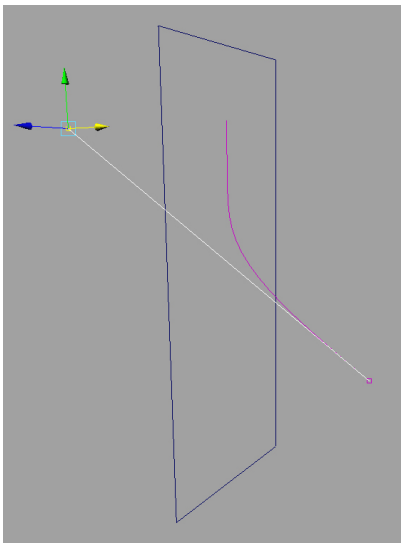
The rebuilt curve have more CVs.

- Build the entire system as described before, but based on the new curve.
- Connect the first curve to the new curve as a wire deformer.

When selecting the CVs of the original curve and dragging them towards the surface you should see something like figure 3_5_12.

Figure 3_5_12

The white curve is the original curve. The info curve cannot be seen since it is with the original curve. The pink curve is the colliding one.



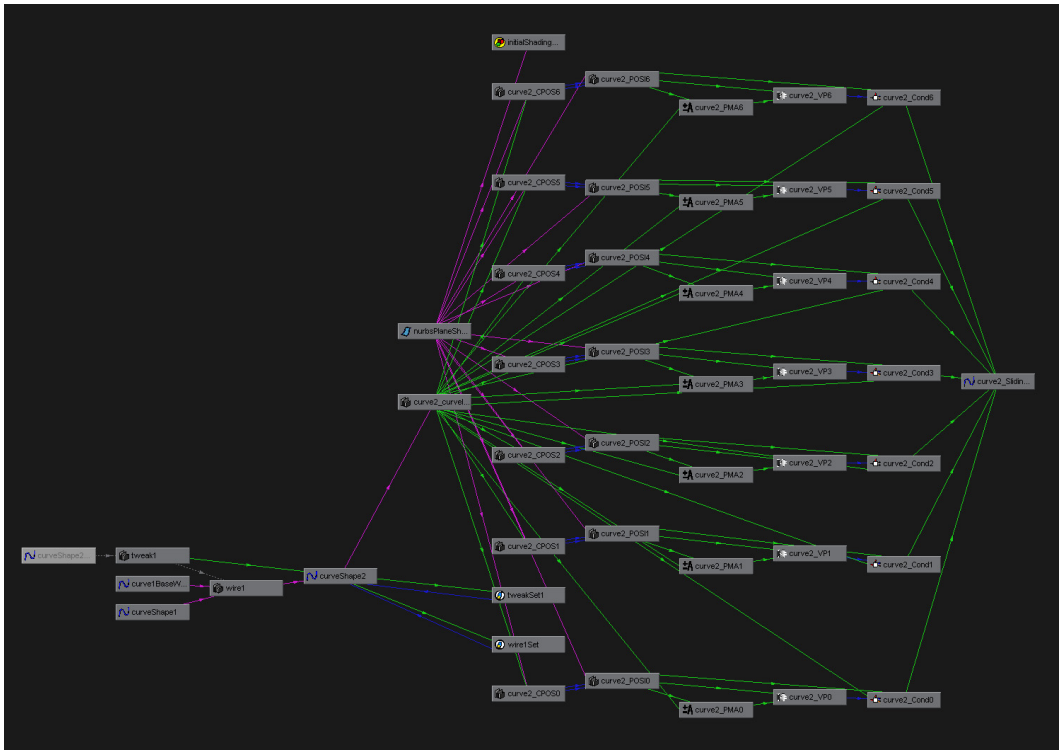
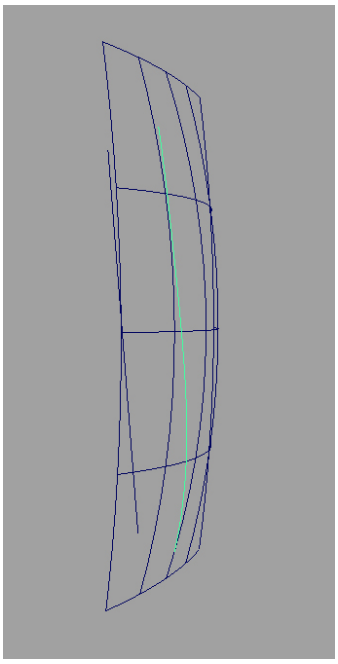


Figure 3_5_13 Unfortunately this is what the Hypergraph looks like after connecting one curve. This image underlines the need to script the entire process.

Figure 3_5_14

Although the system described is not ideal for non planar surfaces nevertheless it can handle them to a certain degree. In this screenshot we can see a curve sticks to a non planar surface.



Implementing with muscles.

Let's look again at figure 3_4_9 and compare it to 3_5_15. The only difference between the two is the use of intermediate curves as described above. This time the muscle is not penetrating the bones, thus the chest will keeps its volume and still will respond to the rotation of the arm.

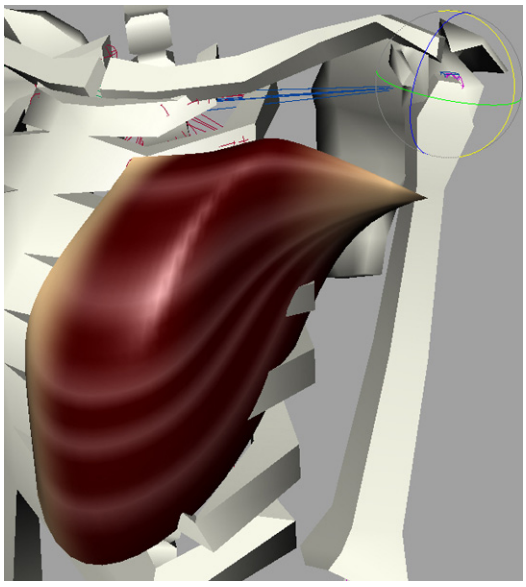


Figure 3_4_9 This is the muscle simply connected to the joints.

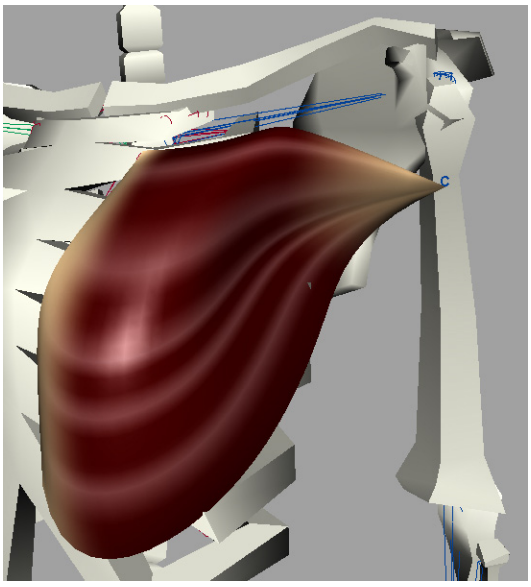
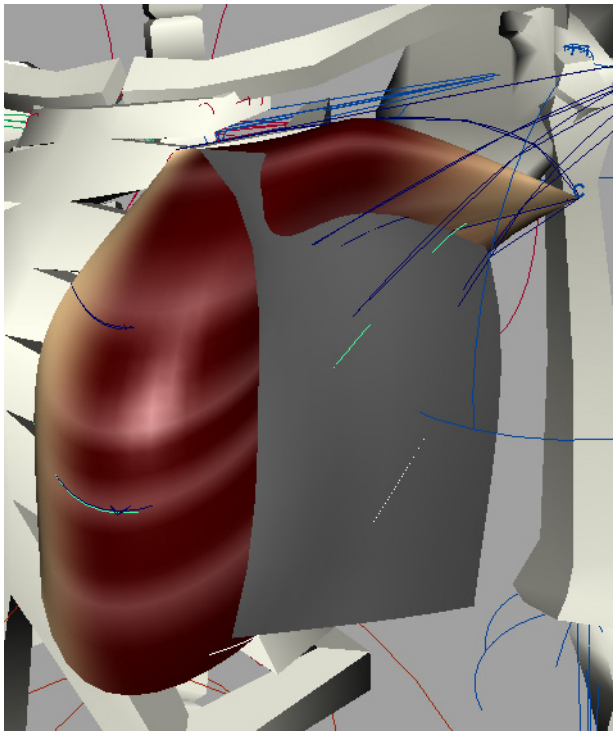


Figure 3_5_15 The same muscle connected to the same bones with the same curves, only this time with the intermediate curves.

The actual setup however is not as neat and elegant. The bones are poly mesh while this system works only with NURBS surfaces. Though we could build the bones from NURBS the resulting geometry will not be suitable for collision. But even if we had a perfect system it would not make sense to connect it to an unnecessarily complicated geometry. Instead we can build a simple NURBS wall and set the system to collide with it. Figure 3_5_16 shows this wall. Notice that due to the nature of the wire deformer we actually have to move the wall out to get the desired results. Don't move the wall too much so the muscle will not deform in its natural pose.

Figure 3_5_16

Here we can see the NURBS surface, the "wall", I use for the collisions. Notice that while the curves themselves can be seen on the surface (two of them are highlighted in green and white) the muscle itself is behind. This is partly because of the nature of the wire deformer. The solution is to simply drag the surface out to compensate for the wire deformer's offset. Dragging the surface outside should be done carefully as to avoid distorting the muscle in its natural position.



Notice that the muscle is three dimensional in nature and obviously some of the CVs of the curves are on the wall's other side. You shouldn't connect all of the CVs to the surface/curve condition. Some of the colliding curve's CVs should be connected directly to the info curve (by connecting them directly to the **controlPoint[*]** attribute of the **curveInfo**).

6. Integrating muscle rig into the pipeline.

When I first tried to set a muscle rig I made an extra effort to optimize the setup so it'll work as fast as possible and I will be able to animate it efficiently.

It didn't work.

After some major compromises I've managed to get a respond time of two to three seconds for a pose. While that might sound like a small price to pay for decent deformations it still held the animation back. I believe that the rig's respond time is critical for the quality of the animation. It is important for the animator to be able to scrub the Time Slider freely.

Not too originally, my conclusion was to animate a proxy rig, and than get the final deformations and render the entire thing. There are several ways to approach this.

Layered rig versus multiple files.

The first question I had to ask is whether I wanted to work in one file or multiple files.

At first glance the simplest way would be to layer a rig that includes two skins, one fast, simply bind to the joints, while the other includes all the deformations. By hiding the complicated skin layer as well as deformation layers we can get the simple skin to work fast.

However I choose to work with multiple files.

There are few advantages to working with multiple files:

A. It allows keeping the animation files simple and small.

- The file will open faster (animation files get a lot of hand work, thus can be opened many times).
- If there are problems the file is easier to debug. A full muscle rig can become quite monstrous.

B. Once you can transfer animation between files, you can do that between characters.

C. Development pipeline is flexible. You can actually start animating and still tweak the mesh to improve character design, or adjust it for unexpected needs or problems.

To understand the pipeline involving multiple files lets first take a look at a simple muscle rig diagram is figure 3_6_1. Note that a real rig is much more complicated.

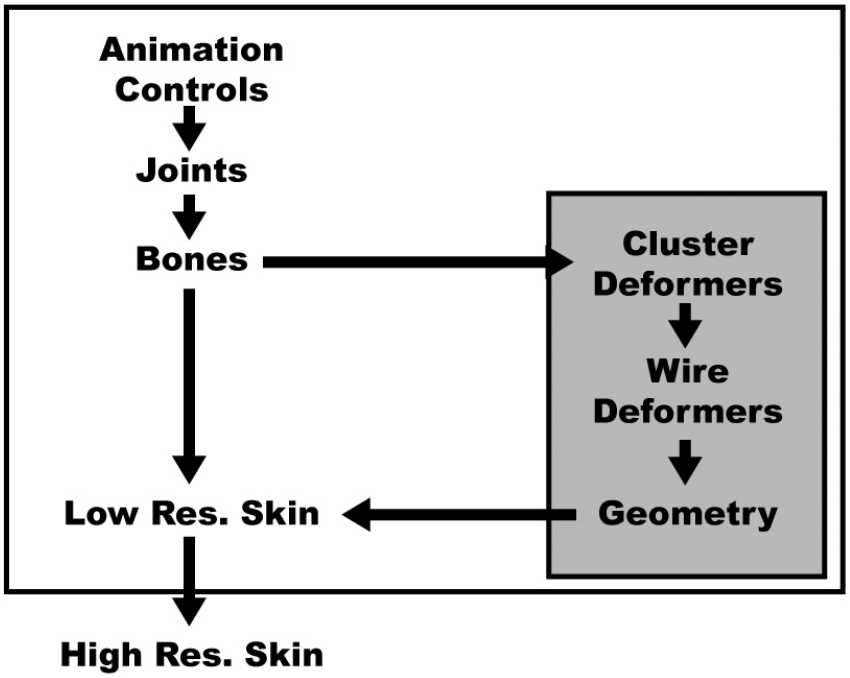


Figure 3_6_1 A simple muscle rig

All the animation information should be associated only with the animation controls. These are the handles that the animator uses to control the character. The animation controls control the joints and bones of the character. The muscles can be treated as a “black box” (grey box in the picture) that is connected on one side to the bones, or joints and on the other side to a low resolution skin. The skin can also be influenced directly by the bones. From the low resolution skin we generate higher resolution geometry suitable for rendering. It is important to remember that this only an example. There can be many flavors to this arrangement.

Figure 3_6_2 shows a two files pipeline. Here we have an animation rig and a deformation one.

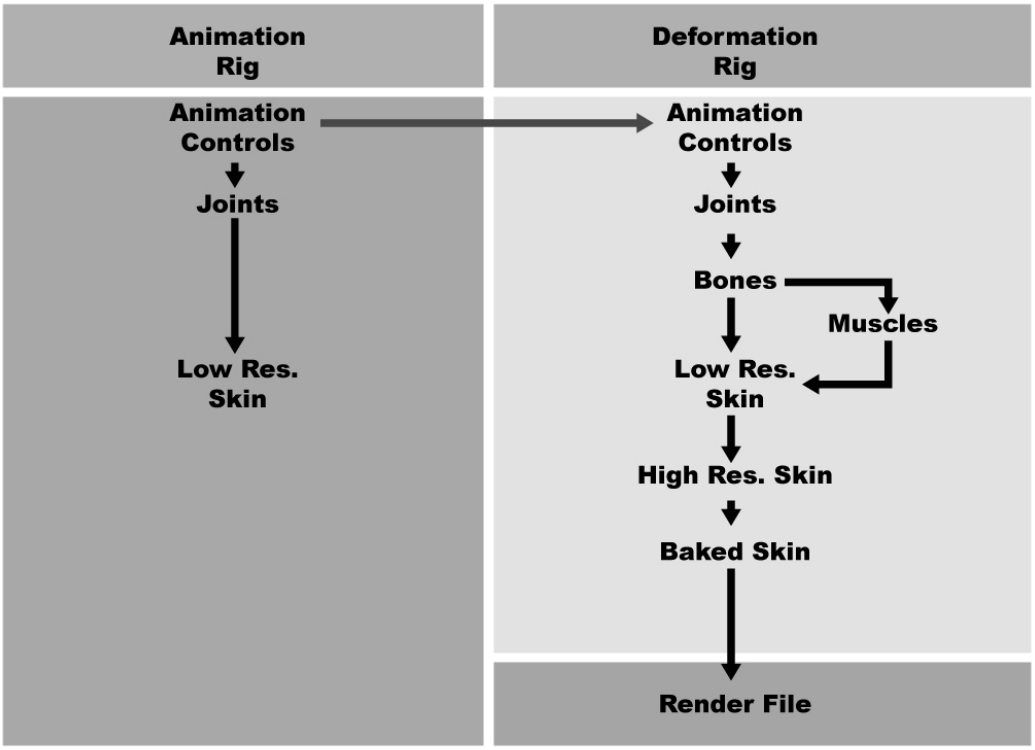


Figure 3_6_2 An animation rig and a deformation rig.

The deformation rig is basically the same rig from 3_6_1.
The animation rig should be as simple as possible. There is usually no need for anything other than the animation controls, the joints layer and a low resolution skin. You shouldn't worry about the quality of the deformations; the skin is there only as a rough visual feed back for the animator.

The most important thing is the relationship between the two rigs. The animation controls need to be identical in both. While the joints don't have to have identical they must respond in the same way. For example you might want to have a Radius-Ulna setup for the forearm in the deformation rig. You don't need to have one for the animation rig but both forearms should be oriented in the same way.

You will need to transfer the animation information between the animation control objects. After I have the high resolution skin deformation (and many times even at the low resolution stage) I bake the skin to simplify the render process and avoid unexpected (and always unpleasant) surprises.

Essentially the deformation rig is a non interactive file. The full muscle rig is in the file. You input animation information from one side and output a deformed skin from the other side.

Things can get even uglier as we can see in 3_6_3.

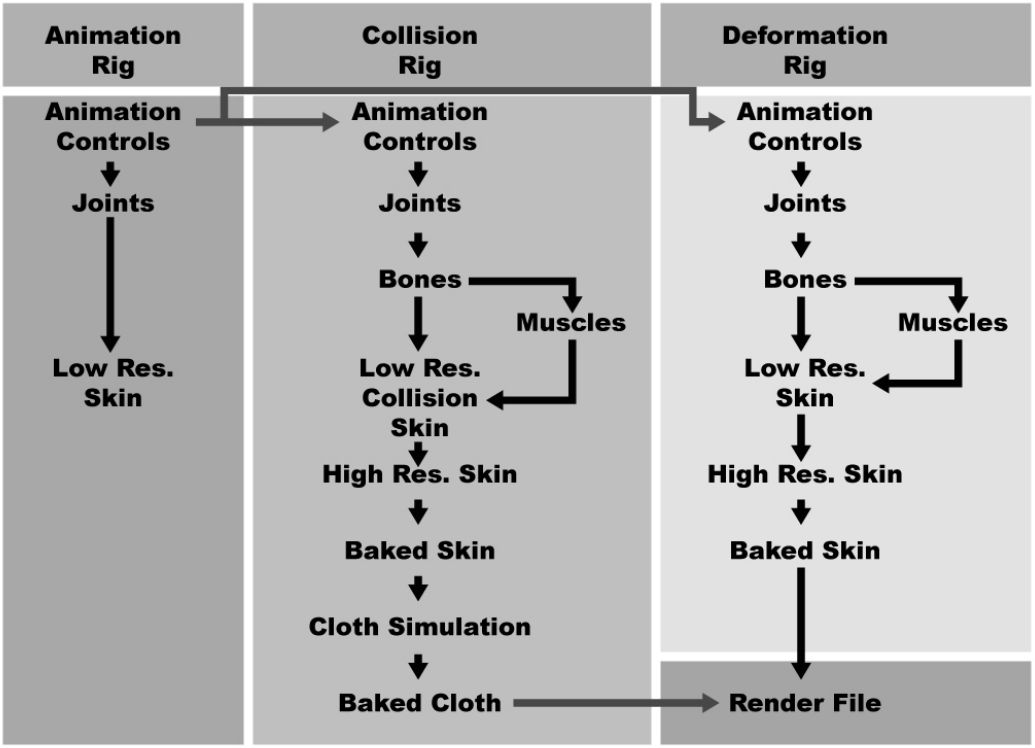


Figure 3_6_3 An animation rig, a collision rig and a deformation rig.

Sometimes, if we are to do cloth simulation fro example, we might need a third rig. This will be a collision rig. Its under-skin layers are identical or very similar to the deformation rig but the skin is different. One reason to have a collision rig is to get rid of unnecessary details such as facial features or fingers. In other cases we might need to change proportions in some parts of the character (if we have layers of cloths for example).

In a three rigs setup we will transfer the animation information from the animation rig to both the collision rig and the deformation rig. WE will also need to transfer skin weights from the deformation rig to the collision rig. Some of the weights have to be identical to the once in the deformation rig, others should be approximately the same. You will have to transfer weights between two geometries that are not identical and do not share the same number of control points. I do that by comparing world space coordinates of the two skins, looking for closest points.

Transferring the animation means transferring the animation curves. You can right the data from the curve into a text file or directly transfer the animation curve nodes. The two challenges are finding all the curves when acquiring the animation information and connecting the right curve to the right object in the transfer stage.

There are several ways to do both and in most cases it will be one of the easiest scripts to write. However it is not hard to get things even better by acquiring some basic abilities to transfer animation between characters. Since this part is not within the scope of this course I will settle for a brief description of the concept.

I use my own character node which connects to all the animation controls (amongst other things)[*]. Every control also has an attribute that indicates what it is (i.e. *Left_Hand_IK* or *Pelvis*). All I need to do is to select the character I want to extract the animation from. The script will first find, through the character node, which are all the animation controls. It will than find the animation curves and before exporting them will add an attribute that will describe to what animation control and what attribute in that control should this curve be connected.
Two points to note:

- I prefer to export the animation curve nodes themselves into a Maya binary or ASCII files rather than write them to a text file. It just saved me some coding and I didn't see any problem in this way.

- I do not write to what character the animation curve belong too, just to what animation control's attribute. It is of course important to know from which character the animation came from but this can be done in the file's name.

When assigning the animation I choose the character I want to assign the animation too and choose the animation file. The script import the animation curves, check for each to what control's attribute it belongs and connect the two.

By not specifying the character in the curve nodes I leave the door open for transferring animations between characters.

It is of course important to mark the controls of different characters and different rigs exactly in the same way. For most cases I'm automating the process of creating rigs and the marking of the controls is done via the scripts. I find it to be the best way to easily and safely keep consistency between rigs.

[*] The use of custom character nodes and attributes to connect them is not my own original idea but a technique I've learned from others.

7. Summary.

This chapter should be considered work in progress. As such it does not offered some sort of "über rig" that will solve all skin deformation problems, personally I don't think there is currently such rig, but rather a range of ideas and localized solutions wrapped into one evolving concept. This system does not require any major programming skills and is based entirely on off the shelf software thus suitable for small studios and individual artists who wants to get better looking characters. A successful implementation of these solutions can significantly improve the skin deformations.

Two Facial Rigs

By Mark Piretti

Sid

Korso

Introduction

On productions such as Ice Age, where the schedules are short and the rigging department small, it is necessary to develop an effective and reliable system for rigging faces that is relatively easy to implement and flexible enough to handle any species. A year before work began on Ice Age, there was talk of Titan AE being made as a 3D animated film at Blue Sky Studios. Within in a few months, I modeled the face of the character Korso from the movie and rigged him as the first step in developing such a system for future feature film projects. This Korso test rig eventually evolved into the template for the facial rigs on the movie Ice Age. From him we derived a standard jaw rig and library of blend targets. Regardless if the character was a rhino or a sabre cat they all shared the same basic joint set-ups and rigging philosophy. As the movie progressed, facial riggers, who were often people borrowed from the animation department, added to and improved the original design sometimes modifying it to fit the unusual facial topology of many of the Ice Age characters. After the movie was complete, I returned to Korso, applying what I learned from myself and from others in order to update the template for the next feature film. It was also an opportunity to test the new subdivided surface technology available now in many software packages.

To illustrate this system I will describe the creation of the facial rigs for the animated character Sid from the movie Ice Age and next generation Korso. The stylistic differences between the two characters gave them as divergent facial rig requirements as is possible. Korso needed realistic but limited skin deformations and extensive wrinkling of the skin. Sid's wide mouth and lack of anatomical features meant that control and flexibility were more important then realism. However, they still share a very similar library of blend targets and joint rigs which makes it easier for animator to jump from animating Sid the sloth to Korso the human.

Geometry

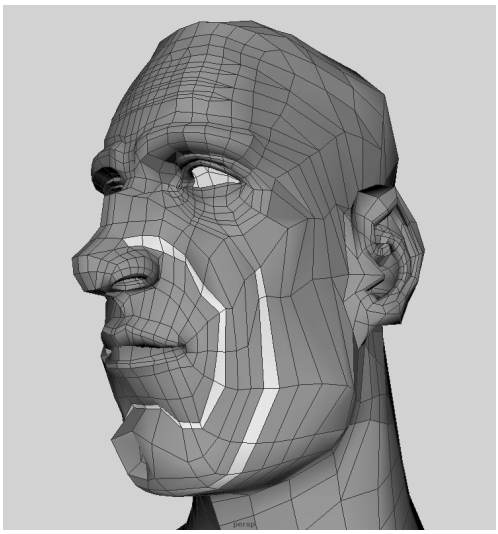
First, a brief discussion on the geometry of the two characters is necessary. In the humble opinion of the author, hierarchal subdivided surfaces, as implemented in Maya, is the ultimate geometry for the facial rigger. Both subdivided surfaces and polygons offer the same great flexibility by allowing the modeler to limit detail to those areas that need it. The hierarchal nature of Maya's subdivided surfaces give the rigger an even deeper level of control. Rigging the refinement levels is at the moment slow and quirky but I was able to make use of them to better detail Korso's facial wrinkles. Sid's head was built before subdivided surfaces became part of the Blue Sky Studio's pipeline and uses our old method of doing things. He is as a collection of NURBS patches that are connected using a Blue Sky proprietary plugin called Suction Cup.

Modeling

A facial rig is doomed to rigor mortis from the start if the geometry is not built correctly. Riggers and modelers must strike a balance between putting in enough detail to get the control you want without adding so much geometry that creating the facial deformers becomes cumbersome and slow. When time permitted on Ice Age a rigger, would test the geometry by modeling a few blend targets, then make some comments and suggestions to the modelers on how to make the model more rig friendly.

I have always felt that the radial method for modeling the face is the best method and it is how every face on Ice Age was constructed. I suggest thinking of the lips as a squashed torus primitive. The lines of the geometry should be as evenly spaced as possible and the bottom lip geometry should match the top lip geometry. I normally model the lips in the closed position. This makes it difficult to know how the interior of the lips will look when the jaw is open. A modeler should check and make certain that the lips are round and tubular as they should be before passing the model to the riggers. For Korso I had to model a special fix blend target after I cranked the jaw open and realized that the lips where not as round and tubular as I like. Wrinkles, not only those which are apparent when the face is at rest, but those that only appear when a muscle activates must be included into the modeling process from the very beginning. One of the most common wrinkles is the nasolabial fold. An oft seen mistake is to model the fold from beneath the nostril or on the same level as the nostril. The fold actually rides above the nostril. It then circles the mouth to define the top of the chin. The nasolabial fold represents the boundary between the fatty cheek regions and the leaner skin above the musclular

obicularis oris. That is the muscle which forms the lips and makes the surrounding skin very taut. Even if hidden or subdued in a neutral pose, the fold will often appear perpendicular to the muscle action. The fold requires a minimum of one isoparm or line of polygonal edges. When I want to get a sharp, defined, crease I flank the initial row on each side with two extra rows. There are other wrinkles that develop with age which form near the mouth but don't a fit the radial topology but with a subdivided surface or polygonal head it should not be that difficult to accommodate them.



The highlighted regions represent the wrinkle zones.

On Korso in order to cut down on the geometry that needed rigging I modeled the fold with two rows instead three. To make up for the missing geometry I refined the faces of the subdivided surface which I later rigged using blend targets to model nice creases. This is such a slow process so that in some of the obvious places one might employ subdivided surface refinement such as the brow wrinkles I actually decided not to use them and modeled the wrinkles as part of the polygonal control surface.

Setting Up For Rigging

I like to separate the neck rigging from the facial rigging. On Ice Age we duplicated the head, and rigged the duplicate later piping it back into the actual head as a blend target. This way most of the facial deformers do not interfere with the rigging of the neck. It also makes it easier for two people to work on the same character, one on the body, one on the face, at the same time.

Because asymmetry is an important part of a facial expression I always model left and right versions of the blend targets. Modifying them both at the same time can be made easy by this simple and obvious trick. Duplicate the target and then scale the duplicate – 1. Now both the right and left blend targets are sitting on top of each other and if you drag select you can adjust their points at the same time. For this to work of course you have to have a symmetrical facial design. If the director insists on a very asymmetric design this can add more time to the rigging of the face.

Setting Up Subdivided Surfaces

Subdivision surfaces update very slowly and those with rigged hierarchal resolutions are even slower. In the Korso test I had three levels of geometry that I could view. I first took the polygonal control mesh and converted it to a sub-d head with "keep original" and "proxy" turned on. I then duplicated the subdivided head, returned it to standard

mode from polygonal mode, and refined the level 1 areas of the duplicate that I wanted to rig with. This is the final rendering geometry. I place the shape nodes of the polygon and both subdivided surfaces into different layers. By clicking on and off layer visibilities I can now view the different resolutions. I do most of my work viewing the polygon and occasionally turning on the subdivided. All of the blend targets and other rigging is done on polygonal geometry which is then piped into the polygonal control mesh of the unrefined subdivided surface.

.**Jaw Rigging**

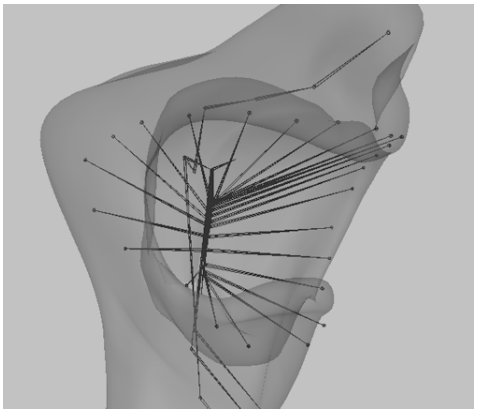
We rig jaws with a series of expression driven joints. Jaw rigs are much more then just an open command. Jaws benefit from controls that allow the lips to press together and close even if the mandible is lowered. The animators can then make an "mm" phoneme regardless of how far the jaw is open. The animators can even cushion the phoneme by animating the jaw while the lips are in the closed position. Another important benefit of joints over modeling a jaw with blend targets is that when you apply the blend targets for the facial expressions the deformations will happen in joint space not world space.

Sid had an extremely wide mouth which made me concerned about the transitions between a wide phoneme such as an "ee" and an "oo" shape. That concern rise to a progressive lip close command that closes the lips starting at the corners and then moving toward the middle of the mouth. This was later implemented in Korso when it was realized how much this can help "o" and "u" phonemes since at a low setting the close command will keep the lip corners tight together.

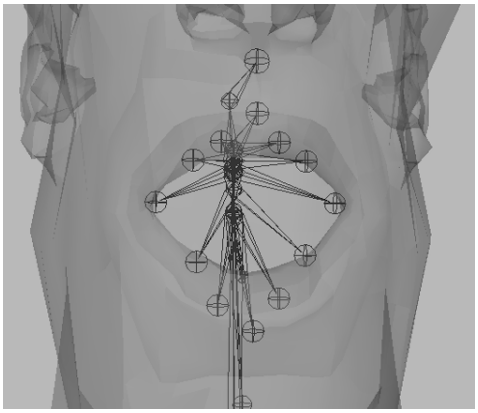
Another issue that arose with Sid was how to maintain lip corner to eye relationships while the mouth was opened wide. In a typical jaw rig the lip corners should travel about 50% with the jaw. Without being able to modify that when the rigger attempts a smile he would have to find a way to translate the lip corners back up towards the eyes. Since the jaws are rigged with a series of joints we could change the amount the lip corners move with the mandible. The number of joints necessary is dependent upon the width of the mouth. Sid for instance has twenty four jaw joints.

All of these needs gave rise to a standardized jaw control system for both Sid and Korso. Each character had these same controls:

> Jaw Open: This is the amount that the mandible moves.
>
> Lip Close: By turning this control to 1 the lips close together regardless of the value of the Jaw Open. For Korso this only represents the jaw action. Sid's Close attribute closes not just the jaw action but any extra lip seperation.
>
> Jaw Height: By keeping a node height value close to 1 the animator basically locks the lip corners in an upper position. By default, the lip corners move 50% with the jaw. If the node height is at –1 then the lip corners move 100% of the jaw motion.



Sid's Many Jaw Joints



Korso's Jaw Joints

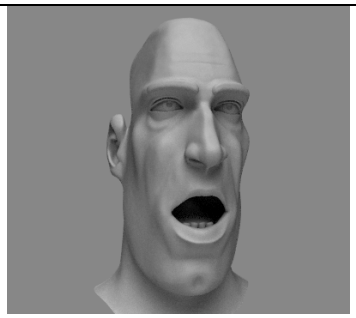
The amount each joint moves with the mandible is handled by the Jaw Control. The value in each of the channels is animated by a set driven key tied to the "Height" attribute. The LeUp1 value of ".247" means that this joint will follow 24% of the motion of the mandible. This value of ".247" is further modified by the "Close" command". When the Close command approaches one the ".247" will begin to match the LeLipCorner value which at a Height of 0 is ".45". The speed at which LeUp1 will match the position of the LeLipCorner is controlled by the proximity, ease, and threshold variables. All of this information is passed into a simple procedure called the maxMaker which calculates what percentage of the mandible motion should the joint move.

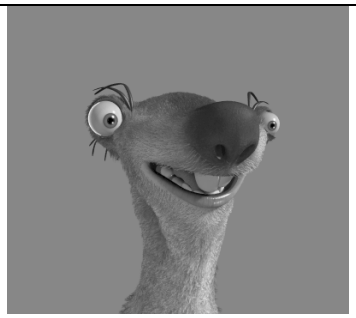
JawControl	
Le Up1	0.247
Le Lip Corner	0.45
Le Lo1	0.65
Lo Center Lip	0.85
Rt Lo1	0.65
Rt Lip Corner	0.45
Rt Up1	0.244
Up Center Lip	0.1
Le Lo2	0.85
Rt Lo2	0.85
Le Up2	0.15
Rt Up2	0.15

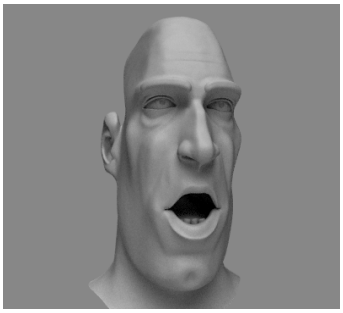
JawControl	
Le Up1	0.1
Le Lip Corner	0.1
Le Lo1	0.3
Lo Center Lip	0.85
Rt Lo1	0.3
Rt Lip Corner	0.1
Rt Up1	0.1
Up Center Lip	0.1
Le Lo2	0.8
Rt Lo2	0.8
Le Up2	0.1
Rt Up2	0.1

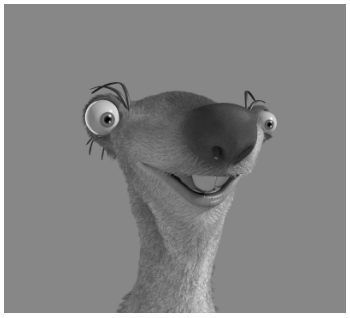
JawControl	
Le Up1	0.588
Le Lip Corner	0.85
Le Lo1	0.85
Lo Center Lip	0.85
Rt Lo1	0.85
Rt Lip Corner	0.85
Rt Up1	0.588
Up Center Lip	0.2
Le Lo2	0.895
Rt Lo2	0.895
Le Up2	0.299
Rt Up2	0.299

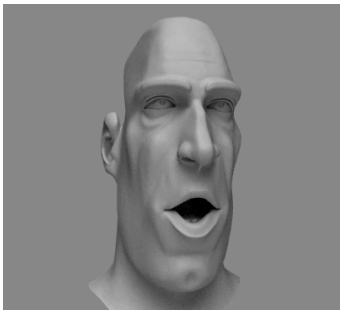
These three snapshots of the Jaw Control show how the motion of the joints change when the Height animates. The first snapshot shows the affect of the Height at 0, the second at 1, and the third at –1.

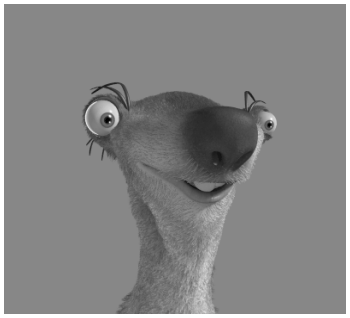

Close=0


Close = 0

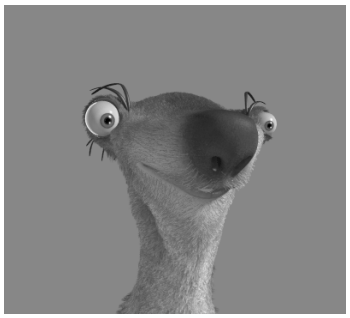

Close = .25

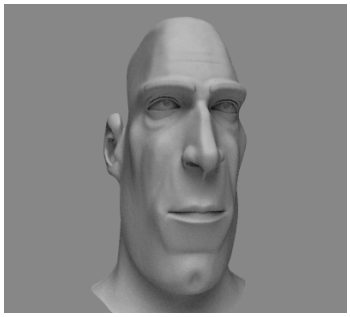

Close = .25

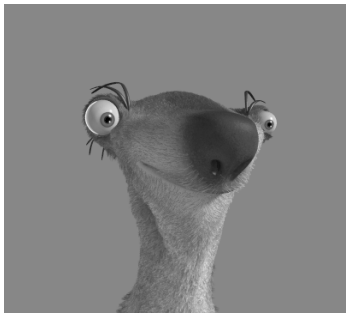

Close = .50

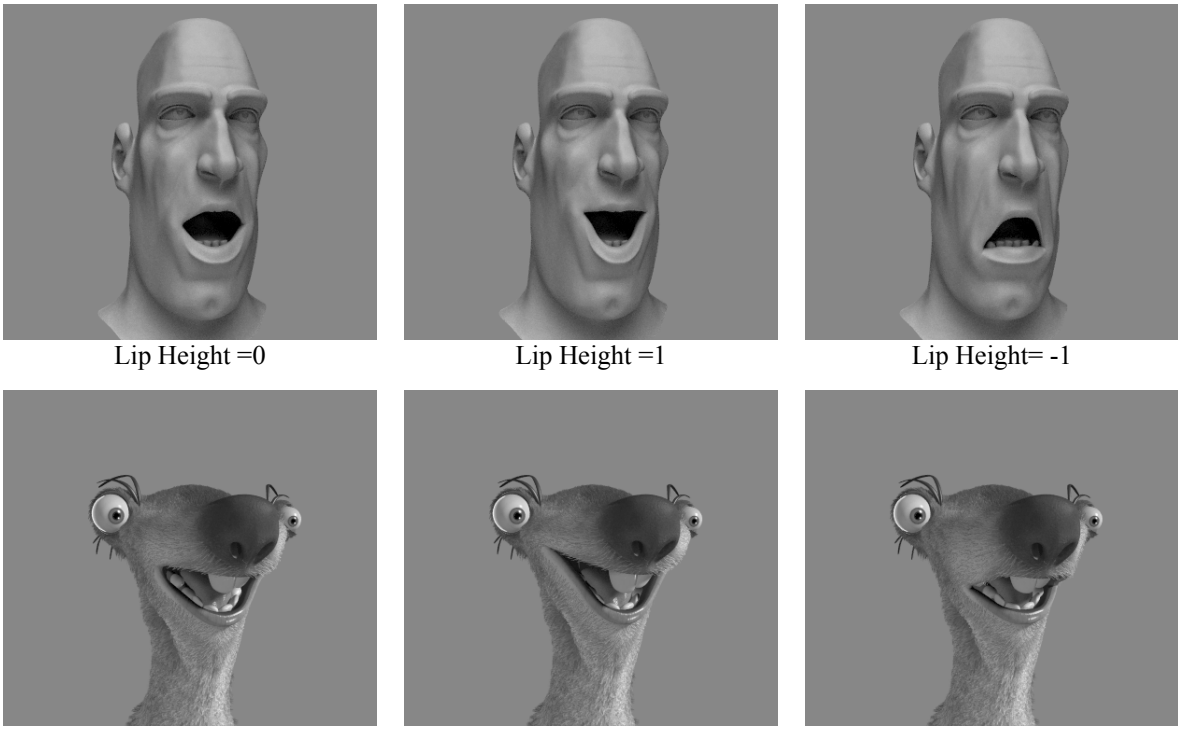

Close = .50


Close = .75


Close = .75


Close = 1


Close = 1



Example:

```
global proc float maxMaker(float $input,float $close,float
$thresh,float $prox,float $ease, float $value)
{

/*

input= value of the joint
close= value of close attribute
thresh= the value of the close command at which point the close begins
to take effect.
prox = proximity to the lip corner. The closer to 1 the closer the
joint is to the lip corner and the faster it reacts to the close
command.
value = value of the lip corner
ease= this value controls the easing in.
*/

float $input;float $close;float $thresh;float $prox;float $value;float
$ease;
```

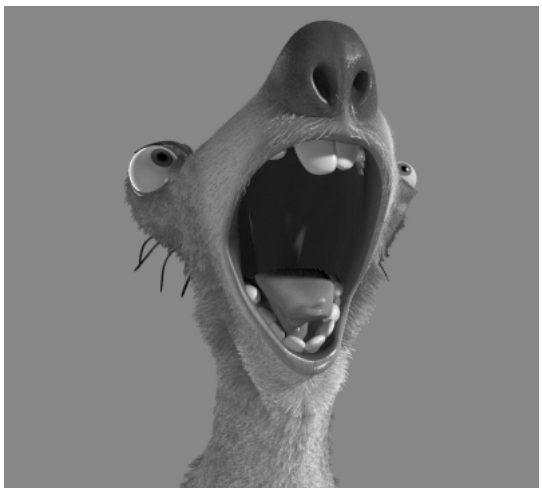
```
float $aa = `linstep $thresh $prox $close`;
vector $zz = hermite (<<0,0,0>>,<<1,1,0>>, <<$ease,0,0>>,
<<0,$ease,0>>,$aa);
float $output = $input - (($input - $value)*($zz.y*$aa));
return $output;
}
```

There are also several other attributes I added to the jaw control. The "add" attributes will move the joint regardless of the value of the jaw open. The "mod" attributes move the joint relative to the jaw open. Only when the jaw is open 100% will the entire value be added to the channel.

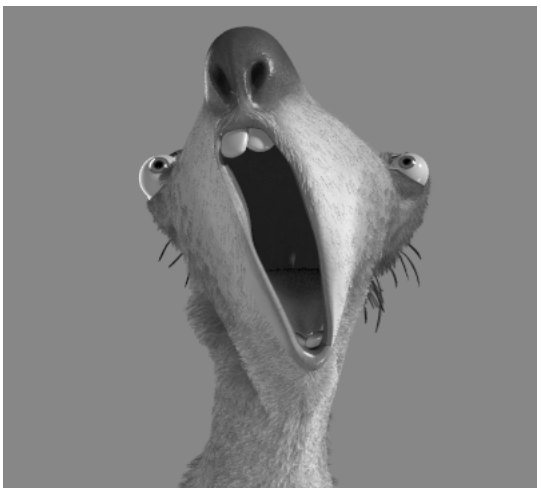
Even with all of these jaw controls it was still difficult to get Sid's mouth to open as far as I wanted to. Simply rotating the neck back was still difficult to keep the neck points from crossing. So I mad use of the jaw joints in place and added parents to them. The parent's rotation is tied into the Max attribute so that jaw joints will continue to operate and close the lips regardless of the neck rotation. This also affects how the jaw rotates along the y axis. The animator can modify the behavior so that the neck will perform normally.

jntJawRtUp2	
Translate X	0
Translate Y	-0.006
Translate Z	-0.002
Rotate X	1.2
Rotate Y	0
Rotate Z	0
Scale X	1
Scale Y	1
Scale Z	1
Visibility	on
Proximity	0.6
Threshold	0.1
Value	0.1
Max	0.06
Addtranslate X	0
Modtranslate X	0
Addtranslate Y	0
Modtranslate Y	0
Addtranslate Z	0
Modtranslate Z	0
Addrotate X	0
Modrotate X	0
Addrotate Y	0
Modrotate Y	0
Addrotate Z	0
Modrotate Z	0
Addscale X	1
Modscale X	0
Addscale Y	1
Modscale Y	0
Addscale Z	1
Modscale Z	0
Ease	1.1

Here is an example of what a jaw joint looks like.



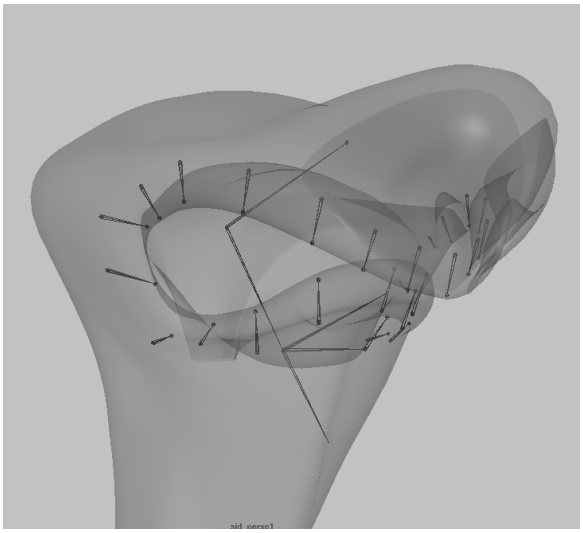
Neck control rotated in Z.



Neck control rotated in Y.

Lips Control

To control the shape, volume, and curvature of the lips Korso makes heavy use of blend targets. Sid uses blend targets only to change the line of the lips as in a smile or lip corner down and utilizes joints for the actually lip separation. These lips are controlled in a similar fashion to the jaw joints. However the animator has another "height" control that modifies how the lip close operates. A –1 Lip Height value will cause the upper lip to close to lower lip. A 0 lip height will close the lips as an average of the values of the upper and lower lip joints. A 1 value will form the lower lip to fit the upper lip.



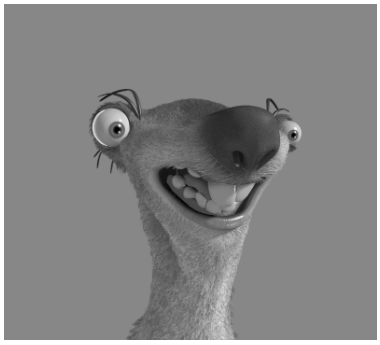
Sid Jaw Joints

Calculating the value of a lip control joint is done by adding this simple expression which takes the initial value of the control, and the value of the opposing control on opposite lip.

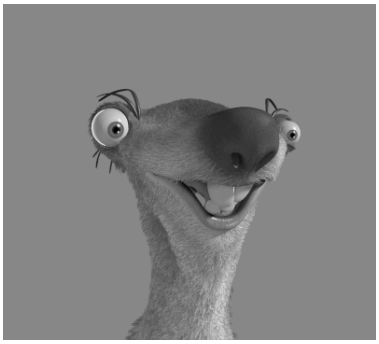
```
global proc float valMaker(float $lipH,  float $iVal, float $oVal)
{

/*
lipH= value of the lip height
iVal= initial value
oVal= other value
*/
```

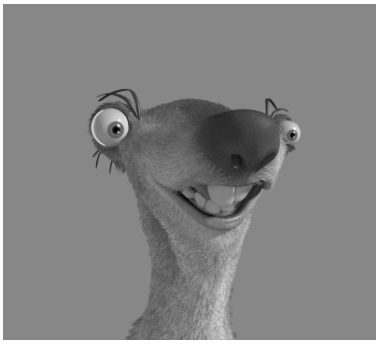
```
float $lipH; float $iVal; float $oVal; float $iVal;
float $avg= (($iVal + $oVal)/2);
float $output=($avg*(1-`abs($lipH)`)) + ($oVal*max(0,$lipH))+
($iVal*`abs(min(0,$lipH))`);
return $output;
};
```



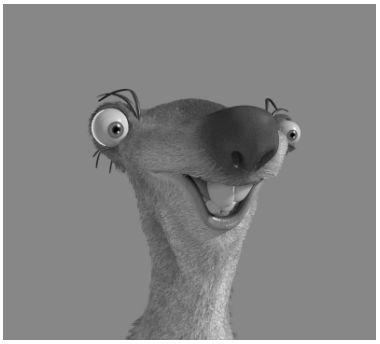
Close = 0.



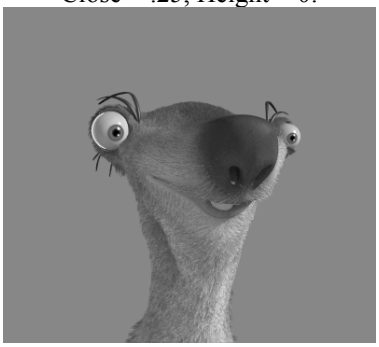
Close = .25, Height = 0.



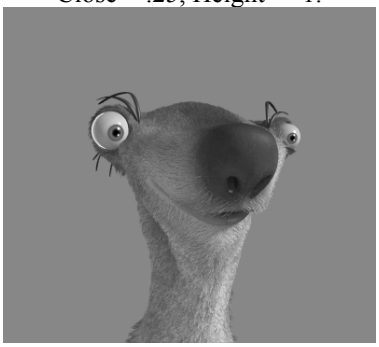
Close = .25, Height = -1.



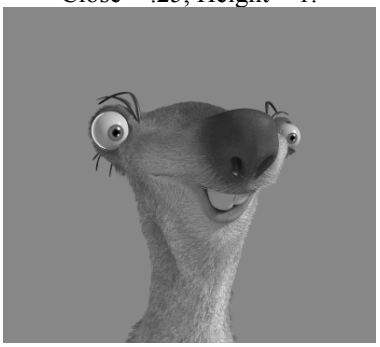
Close = .25, Height = 1.



Close = .75, Height= 0



Close = .75, Height = -1.

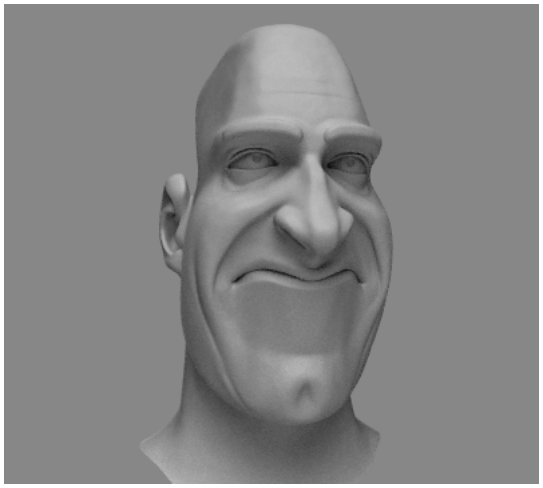


Close = .75, Height = 1

In order to get some of the same flexibility that Sid has but without spending so much time on joint weighting we began adding wire deformers to the lips of the characters to give the animator the ability to move any part of the lip however they want within the

limits. As an extra bonus the wire deformers automatically maintain lip volume and are very quick to add.

Korso's list of blend targets and their creation is pretty standard. One trick I did employ was to model all of the sneers in a closed mouth position. I pass the sneer through two blendshapes, one which has the points of the lower lip dropped out of the deformer set and one that has the points of the upper lip dropped out of the deformer set. The Close attribute is then multiplied against the value of the lower lip blend shape. Thus the expressions below rely on only a right and left sneer target.



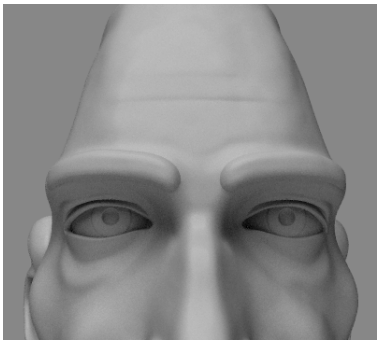
Sneer= .5, Smile= .5, Close=1.

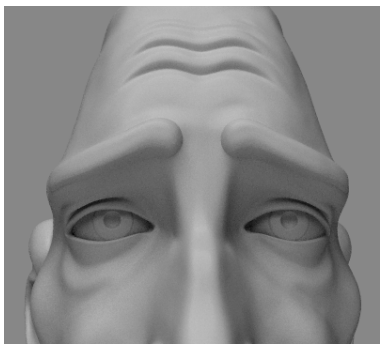
Sneer= .5, Smile= .5, Close = 0.

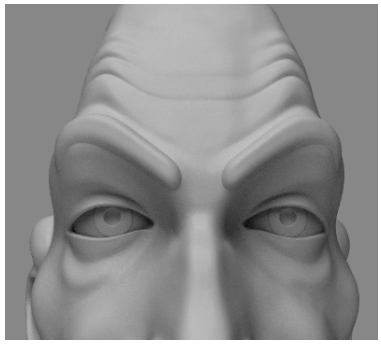
One trick to when creating blend targets is to model them on the geometry itself and use the tweak node so you can duplicate them out later. Make certain that the history goes from the skin cluster to the tweak node and then to the blend target.

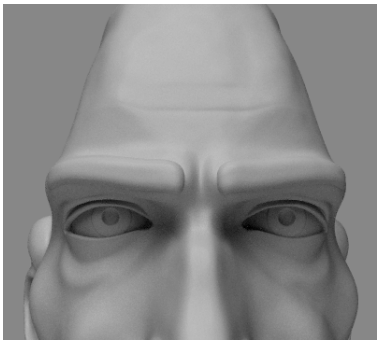
Brow Control

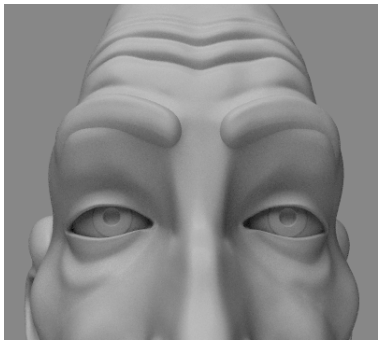
Brows are done through blend targets. There are a few basic brow positions that are added to depending upon the shape of the brows. The six examples shown below show how the inner and outer brow raisers and the brow cruncher can be combined to create a variety of brow poses.

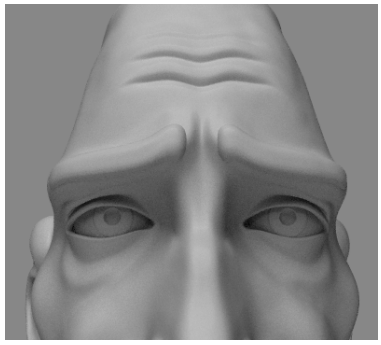

Brow at rest.


Inner Brow Raiser


Outer Brow Raiser


Brow Cruncher

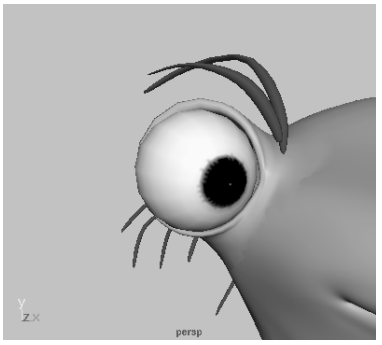

Inner and outer brow raisers combined.


Inner brow and brow crunchers combined.

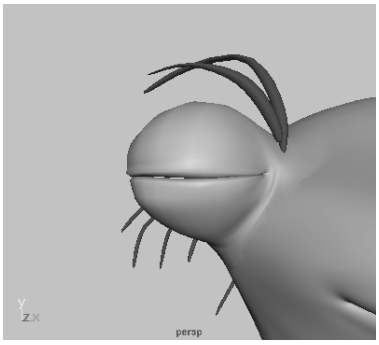
Eyelids

Eyelids were something I used to not pay much attention too. There is however a great opportunity there not only to make the animation of them a little easier but to improve the lifelike quality of the face. I instituted I similar set of controls to what I employed on the mouth. Using the Lid Close attribute the animators can blink the eyes by animating one control instead of heaving to pose the upper and lower eyelids. The Lid Height control allows them to adjust whether the blink is weighted to the upper lid or the lower lid. Otherwise the eyelids have a Lower Lid and an Upper Lid control.

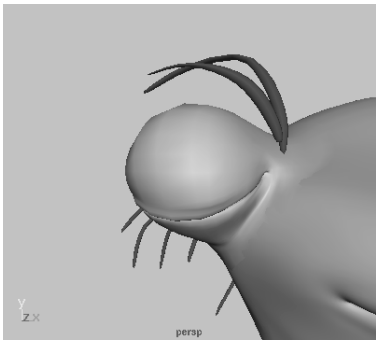
If there are going to be close ups of the character's face it can greatly help make the character more believable if blend targets or deformers exist that change the shape of the eyelid when the eye ball moves. As a test on Korso I tracked the motion of the eye and used that data to drive blend targets. To track the eyeball I created five arrows. One represents the orientation of the eye and is called the Eye Vector. The other four represent directions the eye might look and are called the Direction Vectors.



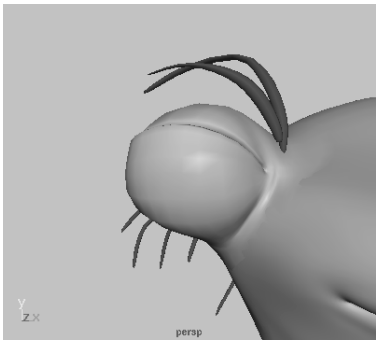
Close = 0.



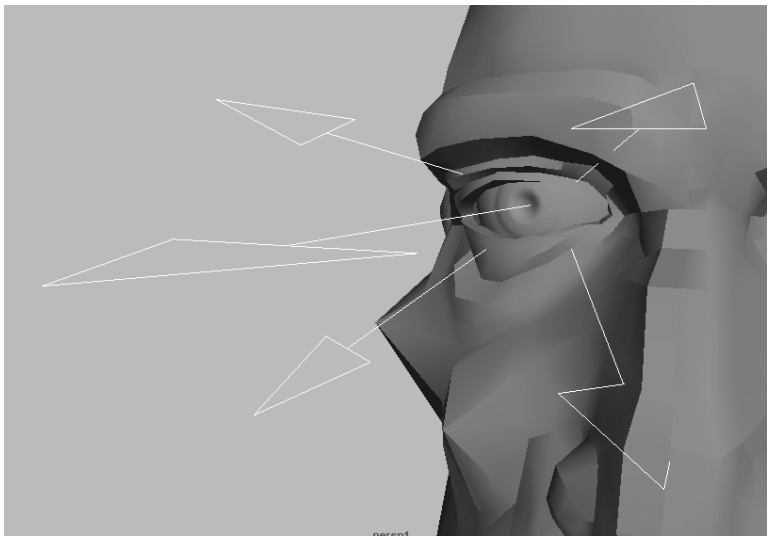
Close = 1.



Close = 1, Height = .75.



Close = 1, Height = -.75.



Eye vectors.

By setting up a simple series of utility nodes in Maya I can calculate the angle between the Eye Vector and the four Direction Vectors. I connect the world Matrix of a Direction Vector to the input of a vector product node whose operation is set to "Vector Matrix Product". I then pipe the output from that into an angle between node. Into the other input of the angle between node I pipe in the output of a vector product node whose input is the

Eye Vector. Now I take the output of the angle between node which will be in degrees to drive the four eyelid blend targets by set driven key (or you can use an expression) .

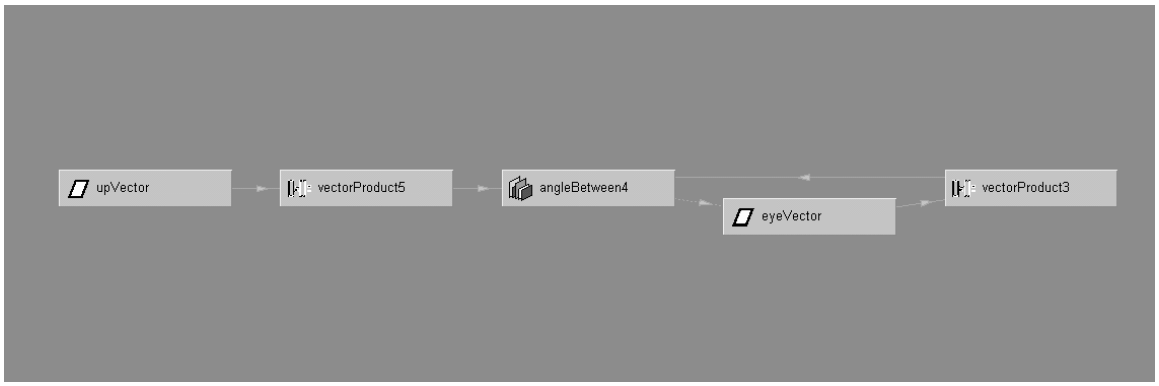
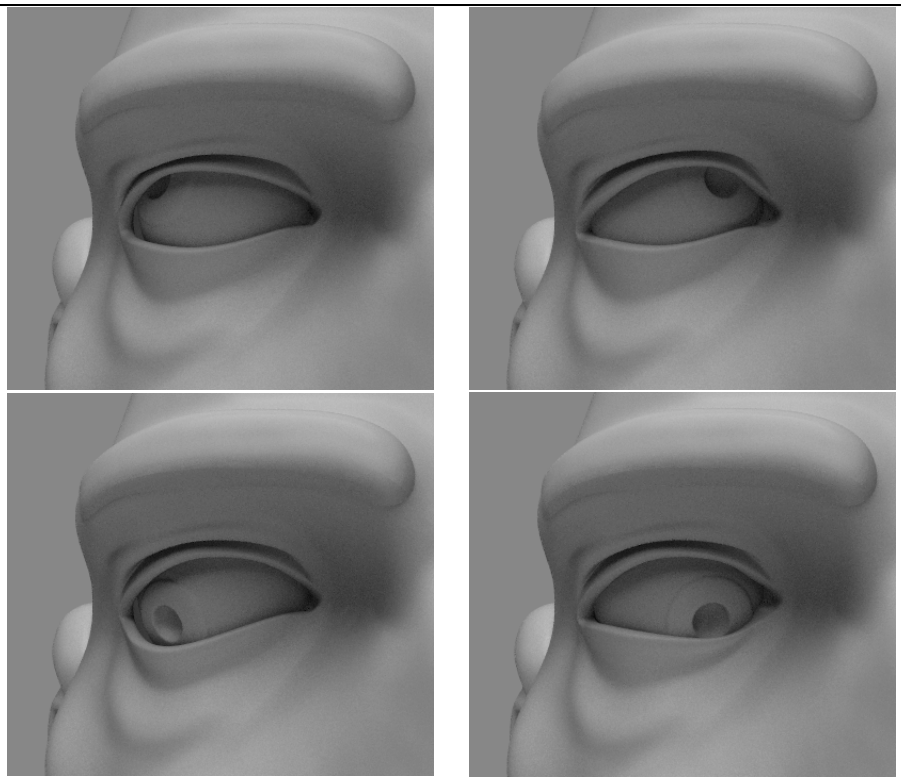


Figure : Angle calculation nodes.



Four different eyelid shapes driven by the direction of the eyeball.

Sculpt deformers in Maya are also helpful in rigging eyelids. They have to be used in twos, one to deform the surface of the eyelid, the other to deform the inside so that the volume of the eyelid stays the same. By moving the sculpt deformers with the eyeball you can create some subtle eyelid motion or not so subtle eyeball bulging.

Conclusion

I will end with a few words that might assuage the pain or perhaps answer a few last questions about facial rigging. First, if your facial rig doesn't look as expressive as you would like it is possible that it was a bad or uninteresting character design to begin with. Part of a facial rigger's responsibility is to at least attempt to steer the design in a direction that will make the character more expressive. Remember when designing a character that the eye and brow relationship is where every expression starts and should be carefully scrutinized. A big mouth can create expressions when there is not much going on with the eyes and brows, but it will never give you the same expressive mileage. Remember that something simple which works is better then any fancy mechanics. Animators should be able to understand the mechanics of a facial rig so that they can take advantage of its abilities in those shots where they want to go beyond the predefined poses.

References:

Goldfinder, Eliot. *Human Anatomy for Artists*, Oxford University Press, 1991.

Acknowledgements

I would like to give special thanks to Donald Greenberg from the Program of Computer Graphics at Cornell University for his generous support. Thanks also to everyone at Blue Sky Studios especially Michael Reed for getting Korso into Studio, Carl Ludwig and Eugene Troubetzky for making Korso look so good once he got there, Maurice van Swaaij, Trevor Thomson, Richard Hadsell, Scotty Sharp for helping me get the math right, Chris Wedge, and Carlos Saldanha.