**JOHN LALLY** | *John is the animation technical director at Insomniac Games. In addition to* RATCHET & CLANK, *John has animated characters for* SPYRO *2 and 3 and for several Squaresoft Games. He is currently working on Insomniac's newest Playstation 2 project and can be contacted at jpl@insomniacgames.com.*

# Giving Life to Ratchet & Clank

## Enabling Complex Character Animations by Streamlining Processes

At first, we were thrilled. As character animators, we couldn't have asked for a better project. There were two heroes, dozens of enemies, scores of NPCs, and more than 100 character-driven cutscenes. Enthusiasm and artistic latitude made it all ours for the taking.

But staying true to our shared vision of Ratchet & Clank meant that our digital actors needed to become more than mere cycling automatons. We regarded each character as an intermediary through which we could reach out to players and draw them deeper into our universe. This meant our characters needed to blend physically into their environments, emotionally into their situations, and expressively into our narrative. It was on these principles that we based both our objectives and our standard of success.

Our team acknowledged that a rift existed between the level of complexity we desired and the time we had scheduled to implement it. In order to surmount this obstacle, we developed several methods for using Maya, our artistic skills, and our time more effectively.

This article will discuss these methods both in terms of their functionality and their implementation. To this end, it will provide technical details on our testing practices, our MEL shortcuts, and our real-time animation procedures. Furthermore, it will explain how each of these methods saved us valuable production time, enabling us to achieve our artistic goals.

### Testing with Prototypes: Why and How

Part of achieving our goal of tying our characters closely to their environments and gameplay meant prototyping low-resolution versions of our characters and their respective animations. Like coalmine canaries, we sent proto-models into our new levels to nose out potential animation, programming, and design problems. We relied on prototyping throughout the course of our production as a means of refining a character's move set. This process of refinement was key to winnowing down unworkable ideas before animating a character's high-resolution incarnation.

As a rule, our prototypes emphasized function over style. And although we set the aesthetic threshold low, these previsualization models still needed to be built and animated accurately enough to function as valid test cases. For the animators, this meant that prototype characters needed to jump to their correct heights, attack to their design specifications, and run at their proper speeds.

Generally, we created prototypes using a character design sketch as a guide. These proto-characters were constructed with primitive objects and only roughly resembled their future incarnations, as you can see in Figure 1 (on page 32). Since previsualization models were so simple to construct, every animator could assist in building them, regardless of their modeling experience. Accuracy was required only in the representation of the character's height, proportions, and posture.

For the most part, our prototypes had extremely simple skeletons: all geometric components were assigned to a single bone with no special deformation. Though such simplicity made for blocky-looking models, in practice our animators had all the flexibility they needed to test out a move set.

Animating our proto-characters was similar to sketching a traditional pencil test. Although animators were given a designer-approved move set, it was understood that animations needed only to be rendered into their roughest forms. One pass was often sufficient, as polish and overlap were unnecessary.

The areas where precision did count were timing, measurement, and interaction with other characters. As they have the greatest direct impact on gameplay, these attributes were considered critical to testing a new character's behavior accurately.

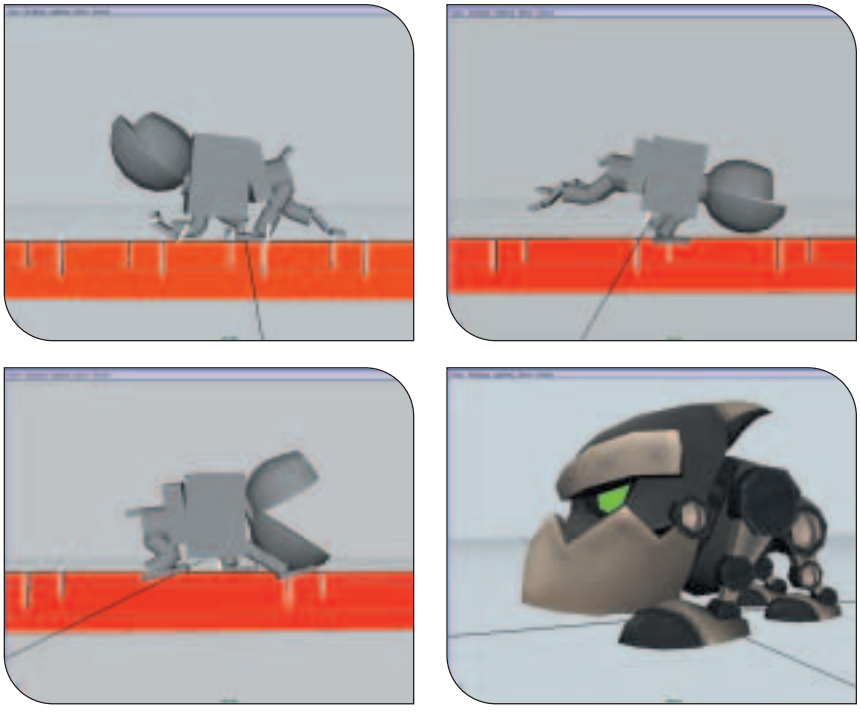
Timing has a major effect on both the readability of an animation and on gameplay. From a distance, a poorly timed idle can look muddy. An attack animation can be too slow to make an enemy a worthy opponent, or too fast to be registered. Emphasis or a lack thereof on just a few frames can make or break any animation, especially within the short cycles of the real-time universe we were creating. We discovered that by testing and fine-tuning our timings in the prototype stage, we could often avoid reworking polished animations on final characters.

Making sure that proto-characters adhered to design measurements was also important. For example, if the design document called for an enemy to attack at a range of 4 meters, animators would ensure that the prototype did exactly this. Designers could then get an accurate idea of whether an enemy traveled at the correct speed, was tuned to the appropriate difficulty, and was scaled appropriately in relation to the main characters.

Prototyping also gave us a means of pretesting character behaviors and interactions. Whether it was with Ratchet or Clank, with the environment, or with another character, proto-models provided invaluable early glances at interactive behavior. For artists, programmers, and designers, previsualization served to telegraph character behaviors both in terms of their technical feasibility and their gameplay value.

Ultimately we found that our previsualization process was beneficial not just to animators but to our design and programming staff as well. It gave our programmers a head start on coding gameplay, while designers could test, tune, and ask for changes at a very early stage, allowing room for refinements.

Prototyping saved animators time and energy that otherwise would have been spent painstakingly modifying or redoing final multi-pass animations. It provided a relatively simple means for evaluating character behaviors with respect to their timing, specifications, and interactivity. Moreover, it provided our animators with a practice run, complete with feedback, before moving on to a high-resolution character (Figure 2).



FIGURES 1A–C. The Dog Charger prototype was used to pretest the final character's animations, including its walk, run, and attack. FIGURE 2 (bottom left). The final Dog Charger model.

## MEL Shortcuts: Automating Our Setups

Maya Embedded Language (MEL) scripts were essential for bridging the gap between the level of complexity we desired and the time we had scheduled to implement it. Through MEL scripts, we were able to streamline setup operations, customize animation processes, and level our technological playing field.

Two such scripts (examined later in this article) allowed our team to take advantage of driven key functionality that otherwise would have been too cumbersome to animate or too tedious to rig by hand. Another tool enabled our artists, regardless of technical experience, to fit characters with IK systems automatically.

Most of our bipedal characters had leg setups like the one pictured in Figure 3. As seen in the hierarchy (Figure 4) our legs had standard hip, knee, and ankle joints, a heel joint, and two to three bones in the feet. (For clarity purposes, please note that we referred to our foot bones as "toes.")

Our IK-rig consisted of three to four RP (Rotate Plane) IK-handles. These connected hip-to-ankle, ankle-to-toe, toe-to-toe and/or toe-to-null. All were configured into a hierarchy (Figure 5) that specified relationships between the IK-handles, a set of locators, and several NURBS constraint objects.

Though relatively simple, setting this IK-system up by hand for every NPC, enemy, and prototype would have taken more time than we had. Moreover, we knew that this time would be better spent bringing our characters to life.

An actual tools programmer might scoff at the artist-authored MEL script we developed to make our leg chains. In the end, however, our "IK Setup Tool" reduced an hourlong technical chore to a simple task that took seconds. Furthermore, the script did not require setup expertise, and our relatively simple code could be customized and refined entirely from within the art department.

Using the IK Setup Tool (Figure 6) was a three-step process. First, an artist checked their characters' leg joint names against the tool's presets, making any necessary changes. Next, a scale factor for the constraint objects was entered,

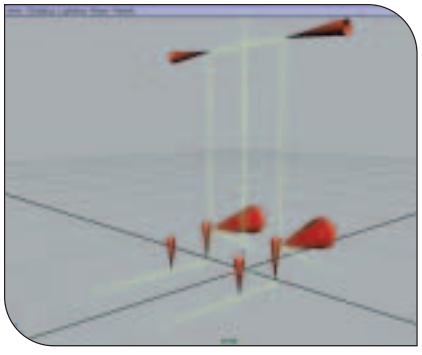


FIGURE 3. This leg setup was used for most bipedal characters, saving tedious hand-setups for IK systems for individual characters.

FIGURE 4. Standard hierarchy for a character's leg, as shown in the Hypergraph.


FIGURE 6. The IK Setup Tool streamlined repetitive, error-prone setup procedures and kept customization within the artists' hands.

based loosely on a character's size. The artist then hit nine buttons in sequence. These buttons would auto-attach the IK handles and instantly build the constraint hierarchy.

## Dissecting the IK Setup Tool

MEL is a quirky and often inconsistent language. A good portion of the time we spent developing our IK Setup Tool was used to track down the proper commands for the tasks we needed to execute. Still, we managed to uncover the MEL commands we needed to actuate the core tasks of each of our nine tool buttons.

The first button's purpose was to place IK handles on a character's legs. It read the names of the bones from the top text fields by using the `textFieldGrp` command in its query (`-q`) mode. These string variables were then passed to the `ikHandle` command, which in turn created the IK handles.

The second button placed NURBS cones on a character's hip, ankle, and toe joints. These cones, created using MEL's `cone` command, were the primary constraint objects an animator would use to manipulate the legs. The `xform` command was used to query (`-q`) the positions of the leg bones and store them as variables. The `move` command then read these variables and moved the cones into place. Finally, MEL's `pointConstraint` locked the hip cones to the character's hips.

FIGURE 5. The leg constraint hierarchy viewed in the Hypergraph, showing connections between the IK handles, locator set, and NURBS constraint objects.

Pressing the third button called `CreateLocator` to place a pair of locators in the scene. Next, the `group` command grouped the locators to themselves. Then `xform` (`-q`) queried the positions of the character's knees, and `move` translated the two new parent objects to the knee joints and the locators to positions in front of the knees.

Button number four configured the cones, locators, IK handles, parent groups, and constraints into a standardized hierarchy via the `parent` command. Again, the new groups were translated into place using `move`. New constraint relationships were created between the knee locators and main leg IK handles, and the new constraint hierarchy and the skeleton. These were implemented using the `poleVectorConstraint` and `scaleConstraint` commands, respectively.

Button five added several expressions to the scene, saving us data-entry drudgery. We added expression code for specifying both constraint and skeletal behavior using the `expression` command, allowing us to automate both the creation and the specifications of our setup expressions.

Number six altered the rotate order of the heel and toe NURBS cones from XYZ to YXZ using `setAttr`. We had previously determined that this rotate order produced the most reliable rotations in our quaint Z-up environment.

Buttons seven through nine performed some final housekeeping tasks. Button seven grouped custom rotation guides to

FIGURES 7A–C. Joint scaling and translation offered animators direct manipulation of poses and gave characters' moves extra verve for the modest cost of a low-tech solution.

a character's spine using the `polyCube` and `parent` commands. Button eight used `setAttr` to ensure Maya's segment scale compensate was switched off for all of a character's joints. Finally, button nine keyed a reference frame at –10 on the character's skeleton and constraint hierarchies using `setKeyframe`. Listing 1 (page 34) shows some of the MEL procedures we found most useful.

Automating this process with MEL both saved us time and eliminated the steps most prone to human error. Furthermore, by enabling any artist, regardless of their setup experience, to fit a prototype and/or character with a functioning IK system quickly, we alleviated bottlenecks. This conservation of both time and human resources saved energy that could then be devoted to artwork.

LISTING 1. Some helpful MEL procedures.

```
// A method for querying a bone's position in world space:
   xform -query -worldSpace -translation my_joint_name;

// A method for querying the contents of a text field:
   textFieldGrp -query -text my_text_field_name;

// A method for setting a keyframe at frame -10 on a hierarchy:
   setKeyframe -time -10 -hierarchy below my_hierarchy_name;

// Basic transformation methods: translation, rotation, and scaling:

      // Moves an object to (0,0,5):
      move -absolute 0 0 5 my_object_name;

      // Rotates an object by 90 degrees on Z,
      // relative to its current Rotation:
      rotate -relative 0 0 90 my_object_name;

      // Scales an object to 3 times its current size:
      scale -relative 3 3 3 my_object_name;

// (Note: All flags are listed in their long forms.)
```

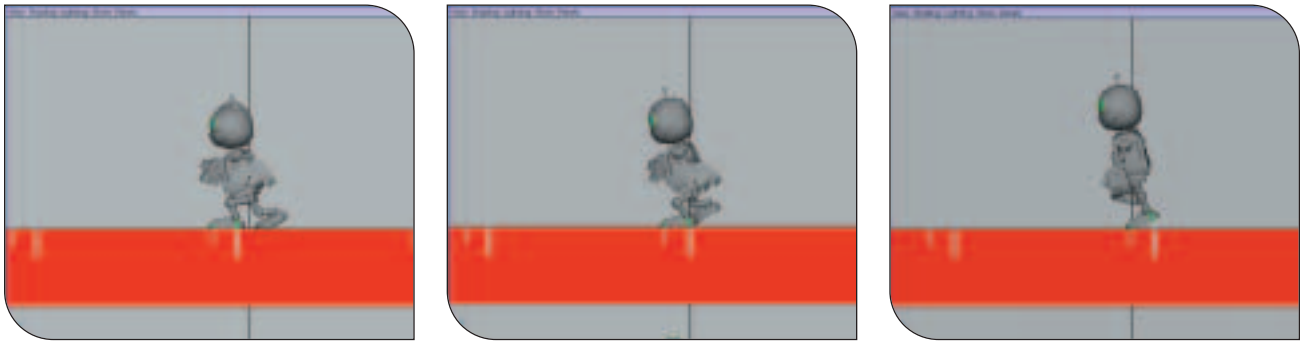
## Low-Tech Animation Solutions

The shortcuts and prototypes I've described so far shared a common purpose: to help us create better animation more efficiently. Both of these methods accomplished this by either by telegraphing problems or by saving time. Often, however, we would spurn a high-tech solution due to its specificity, inefficiency, and/or complexity. And still at other times, we embraced traditional CG taboos.

We consistently and repeatedly translated and scaled our characters' bones. True, most of us learned on our grandmothers' knees never to do that to a CG character. "Use your constraints," she would say. "Rotate your bones if you must. But avoid scaling them, and don't ever, *ever* let me catch you in a translation!" We all love our grandmothers, but we found that the tenets of traditional animation called for — nay, demanded — that we defy her.

The reason behind our disobedience was squash and stretch. We found that by scaling our joints, and especially by translating them, we could instill our animations with extra gravity and snap. Major translations often lasted only a couple of frames and, borrowing an idea from Disney, were "more felt than seen."

Since we had no IK setups to speak of on the spines and arms of our characters, translating the bones in these body parts was quite simple. If needed, we could key the leg IK solvers "off" in order to



FIGURES 8A–C. The Walk Guide helped line up characters' feet on the ground properly every frame to minimize unattractive foot sliding.

manipulate these joints. Translation and scaling were effective across the board and worked wonders on anything from walks to attacks to facial animation (Figures 7a–c).

Requiring no additional setup, these low-tech solutions saved us time. Within limits, this method of animation provided animators with a direct, tactile, and expedient method of sculpting their characters' poses. Although unglamorous, this technique was as effective as any in terms of preserving our resources and improving our animations.

## Walks and the Walk Guide

Another device we used to aid our animation was called the Walk Guide. We used this tool help our characters' feet stick to the ground during walk and run animations. Although foot slippage is commonly forgiven in the world of games, we hoped that by eliminating it we could add an extra dimension of believability to our characters' locomotion.

The Walk Guide was an elongated cube with many smaller cuboids attached to it. The smaller cuboids were identical to the polygonal markers on our characters' ankles and toes, which were grouped to their feet during setup.

By scaling a special parent node, the Guide's small cuboids could be adjusted to match a character's foot size. Scaling the large cuboid allowed an animator to accommodate for the character's stride length. A set of constraints and locators ensured that as the stride length changed, the preset foot size remained constant.

Since our walk cycles were animated in place, we needed a way in which to simulate forward movement while keeping track of the positions of a character's feet. The solution was to animate the Walk Guide to the speed specified by the designer (2 meters per second, for example). Once the Walk Guide was moving at the proper speed and the small cuboids correctly scaled, an animator could begin working on the character's walk cycle.

The trick to using the Walk Guide to eliminate foot sliding was to keep the character's foot markers lined up with the small cuboids on the Guide. This applied for every frame in which the foot made contact with the ground (Figures 8a–c).

Upon a cycle's completion, a character could be put into a level and moved at its preset speed with little or no foot slippage. Additionally, programmers could scale the playback speed of the cycle relative to the character's velocity and still have the feet stay grounded.

There were several gameplay situations that were not as clean as the test case I just described; however, the Walk Guide did serve to plant our character's feet properly in most of our worlds. Once accustomed to the Guide, we animators found that using it benefited both our schedule and our artwork, as it kept track of the more technical aspects of locomotion for us.

## Making Faces: Artistic Reasons and Technical Details

We knew from the start of developing RATCHET & CLANK that facial expression would be an important component not just to our cinematics but to our gameplay animations as well. Once again, we were faced with the dueling goals of animation depth and scheduling efficiency. We settled on two methods for making faces: one simple one for our enemies and one more complex for our heroes. Expressions exaggerated the idles, intensified the attacks, and sealed the deaths our of enemies and heroes alike.

When animating our enemies, we drew on a traditional animation dictum: A viewer of animation is usually drawn to a character's face, particularly to the eyes. Attention paid to a character's eyes and mouth was very important to making convincing actions, especially during our quick gameplay cycles.

Most enemy characters had fairly simple face skeletons. However, these skeletons allowed for a high degree of manipulation of the eyes and mouth. Each eye had between two and four bones controlling its brow and lids. Mouths were generally simpler, using only one or two bones. In most cases, this setup gave us all the flexibility we needed to exaggerate the enemy's features and thus heighten the emotion of its actions (Figures 9a and 9b).

Our heroes' faces had a more sophisti-

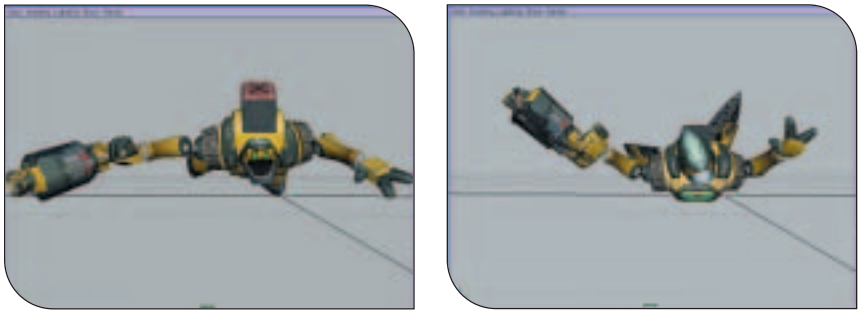
cated setup, which they shared with the NPCs. Though NPC faces were manipulated mostly in our cinematics, RATCHET & CLANK made heavy use of expression during gameplay, as well.

Like the enemy setups, hero and NPC faces were manipulated via their face joints. Unlike the enemies', these joints were animated though a driven key system instead of being transformed directly. Since they clocked more screen time, hero and NPC faces tended to have a far greater amount of bones — and hence expressive range — than their enemy counterparts.

Figures 10a and 10b show some of the range of expression Ratchet and Clank exhibit during gameplay. He smiles when excited, grimaces when he's hit, grits his teeth during combat, chatters them when he's cold, and drops his jaw when he dies. Clank's expressions change both while he's strapped to Ratchet's back and when he's played independently.

As I mentioned earlier, hero and NPC expressions were animated by combining preset driven key attributes via a MEL script slider interface. These presets allowed the animator to combine and create a wide array of facial expression without having to build them from scratch. Like color primaries, these attributes could be blended together to form new combinations.

About half of a character's 40 or so facial attributes were dedicated to producing a basic expression, either on all or on parts of the face. These basic expressions included anger, disgust, fear, happiness, sadness, and surprise, all of which would be easily recognizable to a player. More subtle attributes were dedicated to animating phonemes and controlling individual facial features. Unique and varied emotional ranges could then be achieved by combining expression, phoneme, and feature attributes together.



FIGURES 9A & 9B. With enemy face skeletons, less was more. Bone detail was reserved for the eyes and mouth to enable simple, exaggerated expressions. Here, during an in-game animation, the Robot Paratrooper's face reacts to being knocked down.

FIGURES 10A–B. Ratchet and Clank's gameplay facial animation system (also used for NPCs) needed more sophisticated setups than enemies' for the broader range of expression they were required to show during gameplay.

## Scripting Facial Presets

Assigning facial presets to our characters cost us some setup time. However, we were able optimize some of the processes with another MEL script. Like our other MEL tools, this script automated some of the tedious steps, allowing a setup artist to spend more time on the art of sculpting facial poses.

Facial presets were created in a separate animation file, where each expression, phoneme, or feature pose was stored as a separate keyframe. Upon completing this file, a character artist would use our MEL Driven Key Generator (Figure 11) to set the driven keys automatically for each pose.

The Driven Key Generator worked by comparing the transformations of the keyframed pose to those of a default. When the script registered that a channel had changed from the default, it would set a driven key on that channel based on its changed value. The script relied on MEL's arithmetic functions to identify value changes, and its `setAttr` and `setDrivenKeyframe` commands to activate the drivers. Listing 2 shows some of the Driven Key Generator's sample code.

The drivers for our facial animations were stored on a model called the Control Box, shown in Figure 12. This hierarchy of cubes served as a visual outline of facial attributes, and could also double as a second interface. For efficiency's sake, Ratchet, Clank and all of our NPC characters had identical Control Boxes, though Ratchet's had many more active drivers.

We found our automated setup method to be advantageous for three reasons. First, it saved a setup artist from having to manually identify and key bones, channels, and drivers. Second, it assigned driven keys to changed channels only, leaving any non-affected channels free for animators to keyframe. Finally, it circumvented Maya's built-in driven key interface, which we found to be cumbersome and even unreliable when simultaneously assigning multiple bones and channels to a driver.

Regardless of method, facial animation played a vital role in breathing life into our gameplay characters. Again, MEL was instrumental both in granting our artists access to an advanced Maya feature, and in optimizing our workflow. Whether a hero or an enemy, virtually every character personality in our game was strengthened through facial expressions. In turn, this enhanced interactions both with players as well as between the characters themselves.

## End of Cycle

Like all character-driven projects, RATCHET & CLANK presented our animation team with a unique set of artistic and technical challenges. Our artistic philosophy was built on the understanding that our characters were the instruments though which a player would experience our universe. We knew that in meeting these challenges, our puppets would transcend mere game space and become the entities that our players would identify with, vilify, and even personify.

However, this philosophy needed to be coupled with practical methodology if it was to see our project to its conclusion. From this necessity grew our testing practices, MEL shortcuts, and real-time animation procedures. Throughout production, these methods removed many of the barriers that would otherwise have obstructed the artistic efforts of our animators.

As the Insomniac team cycles into our next project, we continue to refine and expand upon the systems and procedures we developed during RATCHET & CLANK. Though our procedures continue to evolve, our underlying goals remain unchanged. For in the end, we can only prove a technology's worth by an audience's response to our characters.

**LISTING 2. Sample code from the Driven Key Generator.**

```
// The "if" gate checks for changed X-Translation values
// between the Default and Posed frames.

if ($txa != $txb)
{

   // Sets the Driver Attribute and the Current Joint´s
   // X-Translation to their Default Values;
   // Sets a Driven Key Frame for the Default Values.

   setAttr $atnm $dr0;
   setAttr ($current + ".tx") $txa;
   setDrivenKeyframe -currentDriver $atnm -attribute
      translateX $current;

   // Sets the Driver Attribute and the Current Joint´s
   // X-Translation to their Posed Values;
   // Sets a Driven Key Frame for the Posed Values.

   setAttr $atnm $dr1;
   setAttr ($current + ".tx") $txb;
   setDrivenKeyframe - currentDriver $atnm -attribute
      translateX $current;

   // Prints command summary to the Script Editor for
   // easy reference.

   print ($current + ": TX has been keyed for slider
      value 0: " + $txa + " and slider value 10: " +
      $txb);  print " \n";
}

// In this loop segment, $current is the current joint,
// and $atmn is the attribute name. $dr0 and $dr1 represent
// Default and Posed Driver values. $txa & $txb are the
// Default and Posed X-translation values, respectively.

// (Note: All flags are listed in their long forms.)
```

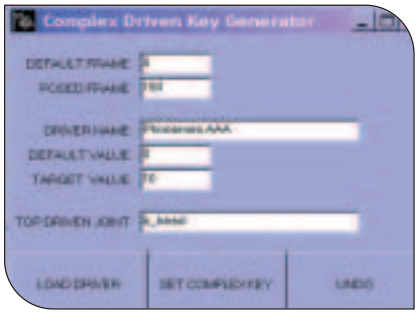



FIGURE 11. The Driven Key Generator analyzed a preset facial pose, compared it to a neutral pose, and assigned driven keys to the affected channels.

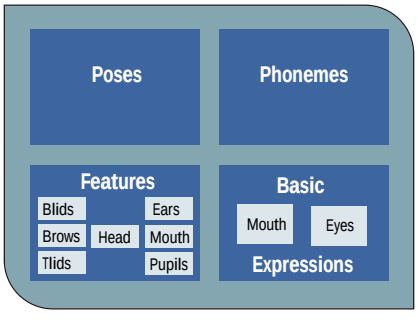



FIGURE 12. Each NPC and hero had its own Control Box on which its facial drivers were stored. Facial drivers were actually attributes of the Control Box's cubes.

### FOR MORE INFORMATION


BOOKS

Gary Fagin. *The Artist's Complete Guide to Facial Expression.* New York: Watson-Guptill Publications, 1990.

Frank Thomas and Ollie Johnson. *The Illusion of Life.* New York: Hyperion, 1981.


RECOMMENDED MAYA TRAINING

MEL Fundamentals (formerly MEL for Artists): Information available at www.aliaswavefront.com, click on Maya Training under "Education"