To Deceive is To Enchant: Programmable Animation

When we create a game, we create an illusionary world for the player. Hopefully this world is so rich and alive that the player may actually begin to believe in the existence of this game world. As game makers, we are in the business of creating life.

Games are getting pretty good at creating believable and immersive 3D game worlds. Graphics hardware has enabled multi-texture lighting techniques and higher polygon counts, greatly improving the environments where we play. It's not uncommon for a person passing by a computer monitor or television screen to mistake a game environment for a video broadcast or a still photograph of some real place.

There are few people, however, who would mistake a 3D animated character for a real person. In the early days of 3D games, it was sometimes even hard to tell what a character was supposed to be. Now that computing power has enabled us to create more realistic looking characters, we need to make these creations come alive. It is not enough that the character looks as good as a still rendering. Characters need to have what Frank Thomas and Ollie Johnston of Disney called "the illusion of life."

Creating the Illusion

One of the great challenges that an animator faces is giving a model the appearance of life. Creating a walk cycle for a character where the feet move correctly and do not slide will not make it come to life. A good animation will give the character a sense of weight, purpose, and emotion, attributes which must be individually crafted for each character. This is one of the pitfalls many encounter when using motion capture to generate animation. It would seem that when capturing the performance of a live person, you would get all the subtle nuances that gives the person life. To some extent that's true. However, what you are capturing is the performance of an individual of a particular size and type operating under certain physical limitations. These limitations affect the range of reality the actor can impart on a character.

An interesting example of this was a visual effects company that needed to create a superhero. They hired a very talented stuntman and attached him to various harnesses and had him leap, roll, pose, strut, and fight all over a stage while the performance-capture cameras were rolling. Unfortunately, when the motion was applied to the hero model, instead of looking like a superhero, it looked like a guy hooked up to a harness jumping around. This is because the equipment accurately captured the performance of this person. He was an actor, not a superhero. He didn't move the way we expect a superhero to move, and the company ended up hiring a group of animators to bring this superhero to life. Sometimes even captured reality is not enough to make a work of art come to life.

Winkin', Blinkin', and Nod

One of the things that can kill the illusion of life very quickly is to have a dialogue with a character staring at you continuously without ever blinking. This is very distracting, as we are very obviously accustomed to talking with people who blink occasionally. When this doesn't happen, it's immediately apparent that something is wrong. So, an easy step toward making your characters more realistic is to allow them to blink.

In a low-polygon model, there may

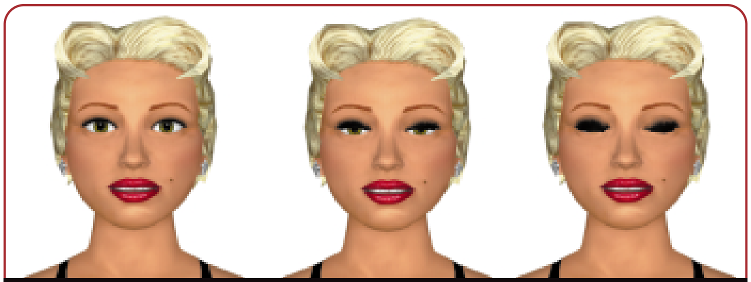

FIGURE 1. *Programming a blinking animation can add realism to a character.*

Jeff is becoming more and more concerned about what procedural functions are controlling his thoughts. If you are the one actually in control, drop him a line and let him know what is really going on at jeffl@darwin3d.com.

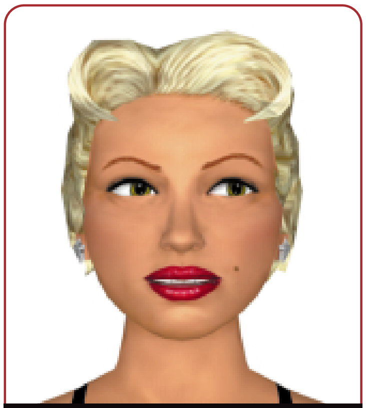

FIGURE 2. *People tend to look around their surroundings when idle.*

not be polygons allocated to the eyes that would make a blink possible. In this case, it's possible to use texture animation to make a blink happen. This has been done in a few 3D games (GRIM FANDANGO, for example). However, if I really want to create expressive characters that can show a range of emotions, I need a model with enough facial polygons to convey the expressions I want. I could hand-manipulate the eyes throughout all of my animations, including an idle cycle. But this would mean the blinks would always happen at the same time in the cycle and it may look canned. Why not animate the eyes automatically through the game engine?

If you read my column on real-time facial animation ("Flex Your Facial Animation Muscles," Graphic Content, July 1999), you may remember that I advocated creating a series of facial morph targets that simulated the actions of the facial muscles. In this system, there is one muscle that controls the closing of each eye. By activating these two muscles, I can make my character blink whenever I want. It may seem overkill to have each eye individually controlled instead of having one action that closes both eyes. However, if you want the character to have the ability to wink, you will need this flexibility. I also don't really care for synchronized blinks. There are arguments over this among animators, but I happen to prefer it if the eyes do not close at the exact same time. I like the eyes to close just off a half-frame or so. Creating my models with separate muscle targets gives me this flexibility.

Anyway, assume I have a model with individual muscle targets for closing the left and right eye. The muscle settings go from 0 percent (open) to 100 percent (closed). I know that I generally want the blinks to happen every four to ten seconds and each blink should take only two to three frames to complete. This gives me the outline for a procedural blinking model:

1. Pick a random number from 90 to 300 that signifies the number of frames until the blink begins (three to ten seconds at 30FPS).
2. Every frame, count that number down until it hits 0.
3. Pick a random number from 30 to 50 for each eye.
4. Add that number to each eye muscle, limiting to 100 percent.
5. Subtract that number from each eye to get back to 0.
6. Start over at step 1.

You can see a sample blink sequence in Figure 1.

This procedural sequence can run as part of my animation loop without any other controller. As the actual morphing system is part of the standard facial animation system, the blinking generates very little overhead.

This idea of a procedural function for generating motion can go beyond simple blinking. When people are standing idle, their attention drifts. They look around to get a sense of the surrounding environment. Likewise, a character in a game should not stare rigidly forward waiting for input from the player. If your facial controller allows you to orient the eyes, you can apply the same techniques as the blink function to make the character look around a little. Obviously, if the character is actually looking at an object or a person, this look-at constraint would override the random eye-wandering behavior. Similar random facial effects can easily be crafted to move the eyebrows and mouth slightly, giving the character even more life.

The game AI can also drive the character animation system automatically. For example, the AI system may track many of the character's various attributes such as fatigue level, mood, and physical pain. Facial expressions can be crafted to exhibit these various traits automatically as the AI system changes them. This not only adds realism to the game, it provides a secondary feedback mechanism to the player.

Now that computing power has enabled us to create more realistic looking characters, we need to make these creations come alive.

Looking Beyond the Face

While simple facial adjustments can do a great deal to break a character's wooden stare, something needs to be done with the static poses. An idle animation or two can help a great deal. However, these animations are usually cyclical and the pattern is easy for players to detect. Adding some procedural noise to these standard animations can help a lot. If your animation system is composed of a character that is deformed by the use of a skeletal system, these kinds of on-the-fly modifications are easy to implement. For one thing, you could add a little random rotation to the head joint to match the wandering gaze of the procedural eye controller. If there is a bone controlling the finger rotation, the character's grip can be relaxed and tightened, which people often do when they are waiting around for something to happen.

I have often found it useful to create special bones in the character skeleton specifically for these sorts of custom procedural effects. One example is literally breathing life into a game character. In my column on skeletal deformation techniques ("Over My Dead, Polygonal Body," Graphic Content, October 1999), I discussed the use of a full transformation matrix to deform a 3D character. A side benefit of this technique is the ability to use transformation components beyond simple rotation. A bone can also be translated and scaled to deform the character. Though it may not be obvious when these techniques are useful, this is one of those cases. Suppose I made a child bone of the chest and called it "breastbone." I could then attach the vertices at the front of the chest to this bone. By cyclically scaling this bone up and down, I can give the character the appearance of breathing. Say my character normally breathes 12 times a minute and this goes up to 20 times a minute when the character is very excited or fatigued. I can create a simple procedural formula that will automatically create a breath cycle in the game without any animation. Something like breastbone.scale = (sin(DEG2RAD(frame * 1.2)) / 4.0) + 1.25 will make the breastbone scale up from 1 to 1.5 once every 150 frames, or 12 times a minute at 30 frames per second. Increase that 1.2 to 2 and the character would breathe 20 times a minute. You can see how formulas can easily be crafted that would enable all kinds of automated behavior.

Ken Perlin has experimented more than most with using procedural functions and controlled noise to generate animation. The Improv animation system, developed at New York University's Media Research Lab and since licensed to Improv Technologies, uses controlled noise and a variety of animation blending functions to create unique and believable animation. The animation is controlled from a high level using a scripting language. You can see a Java interface to an Improv-driven facial animation performance in Figure 4. Improv Technologies is licensing its animation system for game development applications.

Muscles Flexing

A procedural noise or cyclical animation effect can be very interesting. However, sometimes an effect needs to be the direct result of an action. For example, a strong character may have biceps that bulge as the character bends its elbows **{Edit OK?}**. Just as I did with the breathing example above, I can create a child bone of the upper arm and place

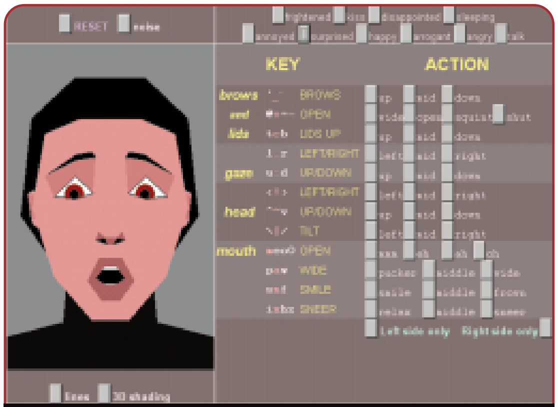

FIGURE 4. *Improv Technologies' system uses animation blending functions and controlled noise for unique effects.*

FIGURE 3. *Trust me, this chest is heaving. Adding a breathing sequence can enhance a character's emotional response.*

it in the middle of where I desire the biceps "muscle" to appear. This logical bone for the biceps is then associated with the vertices along the top of the upper arm, exactly where the biceps would appear. In order to make the muscle bulge, I just need to scale the biceps bone up a bit. In order to detect when I want this to happen, I could watch the rotation of the forearm and adjust the biceps as the forearm rotates. However, I really do not want to create all this code to monitor the bones involved in these effects. I just want it to happen automatically.

In many 3D animation packages, you can create a structure called an "expression." An expression creates a mathematical link between two objects in the animation system. I want to use this same idea to simplify my muscle-bulge situation. When the lower arm rotates around 120 degrees about its local X-axis, I want the biceps to be scaled up to 1.5 of their original size. An expression that performs this task may look like bicep.scale = (forearm.rotX / 240.0) + 1.0.

As you can see, if the forearm is not rotated, the scale would be one. When the bone rotates, the scale will increase. Extra care needs to be taken to make sure that the scale doesn't go below one or get too great. However, this is easy to accomplish through the use of degree-of-freedom restrictions.

Effects similar to this can be achieved simply by animating the biceps directly along with the forearm. Unfortunately, this would not work if the player can dynamically pose the character through inverse or forward kinematics. The use of expressions also saves animation space by eliminating the need to store the data directly. And while this is just a simple example, but you can see that using expressions to generate real-time animation data can be a very powerful tool.

Looking Around the Room

Many algorithmic techniques for animation cannot be simply triggered by a mathematical expression or stochastic procedure. They require some input from the user. A simple example is the look-at constraint. When the player (or

AI system) directs a character to look at a location, this direction takes the form of a request for an animation solution that solves a geometry problem. Given a character who has a head that can look at a limited subspace, we need to find the orientation for that head that points in the direction of a target location **{Edit OK?}**.

Recall that a 3×3 orientation matrix is composed of three orthogonal unit vectors that define the local coordinate axes of the rigid body. These vectors are often known as the right (R), up (U), and forward (F) vectors which define the local X, Y, and Z axes in the right body. If I can determine the direction for these three vectors, I can combine them to form the orientation matrix for the head of my character.

$$Orientation = \begin{bmatrix} R_x & U_x & F_x \\ R_y & U_y & F_y \\ R_z & U_z & F_z \end{bmatrix}$$

When the character looks at a location, the head is aligning one of these three axes with the vector between the root of the head and the look-at target. In my case, I have constructed the head such that it normally looks down the positive Z axis. So to make the head look at something **{Edit OK?}**, I create a vector between the root and the target and normalize it so it becomes a unit vector. This defines the forward vector in the above matrix giving me one piece of the puzzle. I know that generally I still want the head to be aligned upright along the original Y axis. So for now, I am going to set the U vector to be (0,1,0). This may not be correct as the look-at may cause the head to tilt a bit, but for now it will be fine.

To determine the R vector, I am going to use the fact that the cross product of two vectors is perpendicular to them.

$$R = U \times F$$

But I still need to fix up U since my guess may not have been correct. This is easily accomplished by crossing the F vector with the new R to determine the actual U vector.

$$U = F \times R$$

That gives me all the pieces to the orientation matrix and I can make the head look at any point in my 3D world. However, this could lead to problems if the point is behind the character. The head would spin around like Linda Blair's in *The Exorcist*. In most cases, you will want to have limits on look-at constraints so the characters will only do what is physically possible. For the head, I probably want to restrict the character so it can only turn its head 60 degrees or so in each direction. In order to make this happen using the above method, I would need to take the orientation matrix I calculated and convert it to Euler angles and make sure none of the angles were outside the range of plus or minus 60 degrees. That is kind of a pain and mathematically intensive so let's take a look at the problem in another way.

From a geometric perspective, the problem of looking at an object can be thought of as a two-degrees-of-freedom problem. I want to find the angle around the local Y axis (or yaw) and the angle around the local X axis (or pitch) that the head needs to rotate in order to look at the target. I can solve the problem by projecting the target point onto the local XZ plane to determine the yaw, then projecting the target on the local YZ plane to determine the pitch. Each of these steps then becomes very easy. Figure 5 shows the target position projected onto the XZ plane.

I know from trigonometry that the value for tanθ is equal to T_x/T_z. So I can determine what yaw the head needs to turn by taking atan(T_x/T_z). I can do the same for the pitch using atan(T_y/T_z). If those values are within the range of motion for the head, the character can animate to that position.

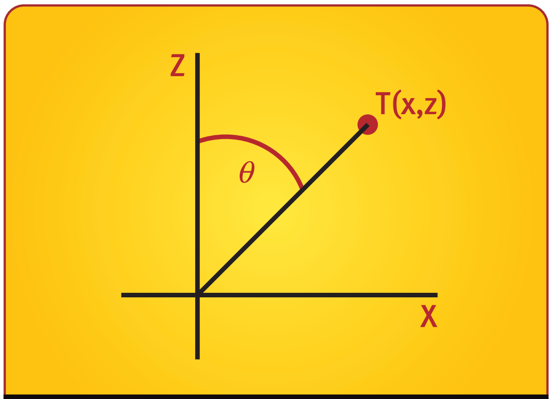

FIGURE 5. *Projecting the target position.*

The Rise of the Programmer/Animator

I think we are quite a ways from being able to programmatically create complete, believable, and interesting animations for interactive 3D characters. But an interactive character is not like an actor in a video clip or a 3D cutscene movie. In order for this character to create the illusion of life while the player interacts with it, we need to accentuate the 3D model with programmable animation techniques. Through creative use of methods such as procedural animation and programmable expressions, we can supplement the basic animation with interactive elements that react with the game environment. In this respect, game developers are clearly treading across new ground. These challenges and opportunities are unique to interactive animation. However, this is certainly not the domain of game programmers alone. Creation of tools and production procedures that can get artists, with their creative vision for the game characters, involved in the process will be critical if we are to deceive and enchant our audience successfully. ■

FOR FURTHER INFO

Disney Animation

Thomas, Frank, and Ollie Johnston. *The Illusion of Life: Disney Animation*. New York: Hyperion Press, 1995.

Improv Technologies

For information on Improv see Ken Perlin's web site as well as Improv Technologies'.

http://www.kenperlin.com

http://www.improv-tech.com

Acknowledgements

Thanks to Lisa Washburn at Vector Graphics (http://www.vectorg.com) for the model of Vivian used in the article.