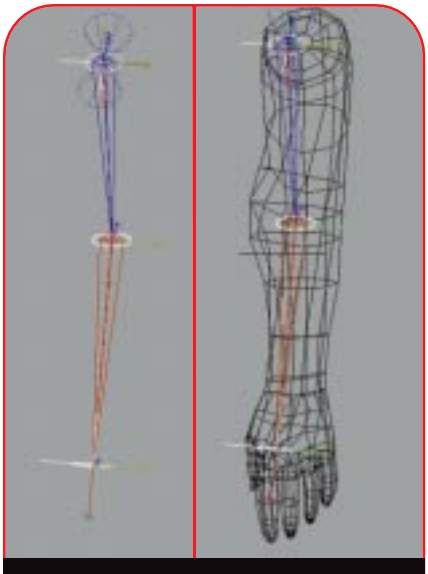
Skin Them Bones: Game Programming for the Web Generation

Well, it's true. The Internet has really changed things, and I don't mean in the way the news has been hyping it. Although it hasn't quite lived up to all of the media hoopla, the Internet has changed the way people communicate.

This is particularly true of accessing information. When I was initially learning 3D graphics for display on my lowly Apple II (and then on my Amiga), I really had to dig. I was fortunate to live near several major universities, and when SIGGRAPH was in Anaheim, it was right in my backyard. I would also comb through magazines and journals trying to figure out what the heck was going on. The books were never up-to-date on the latest techniques, and the people who were working on the coolest things were scattered all over the world. I could never afford to attend the seminars and symposiums where the professors met and compared notes, so I waited for the printed word to get back to me.

Today, all of this information is at your fingertips. Most journals and papers are now available directly online. All researchers post their papers on their own web pages before they're published in print. Even better, the people creating this work post their e-mail addresses on these pages. Now I can read up on the latest and greatest. If I have any questions, I just ask the author. Most impressive of all, the majority of these authors get right back to you and are flattered that you find their work interesting. Imagine being a kid in California and hearing about an Englishman named Newton. This guy has just come up with some interesting ideas about how things react when they bump into each other. So, because you're trying to make a pinball game, you fire off an e-mail about the problem. Newton fires back a quick explanation of his third law of motion, complete with animated .GIFs of things bouncing into each other. I can't wait to see what this generation of kids will come up with.



FIGURES 1 and 2. *First the bare arm skeleton, then with skin applied.*

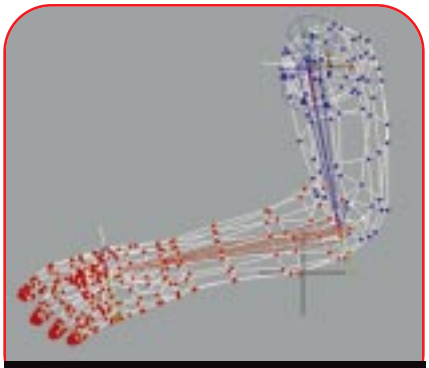


FIGURE 3. *The arm skeleton once it has been weighted and deformed.*

Bend without Breaking

So this brings me to my current problem. In my last column, I wanted to deform a skin mesh to a set of bones in a hierarchy ("Better 3D: The Writing Is on the Wall," April 1998). A real-time 3D character created from a single deformed mesh looks much better than one made up of separate objects. Every major 3D graphics animation package has a method of deforming a single mesh object. Most of them work by embedding some form of bone system inside the character, then using these bones to influence the shape of the mesh. This is the approach that I wanted to take for my character animation.

Not wanting to build a bridge where there was already a tunnel, I hit the books. Just by looking through the SIGGRAPH proceedings and hitting the

When not bending the bones of some strange alien creature, Jeff can be found hanging out at his studio at the beach. See if you can smack some sense into him by writing to jeffl@darwin3d.com.

Web, I came up with a whole bunch of stuff on character animation. These sources provided a good start, but weren't quite right for my purposes. I followed up on some references in those papers and still wasn't satisfied that I had found anything that directly applied. So, I fired off some e-mail to the different authors and some colleagues, asking if they knew any good sources for information. Amazingly, I received over an 80 percent return on those e-mails — including responses from some of the biggest names in computer graphics over the past decade. I don't know why I was ever intimidated by asking questions of the people best suited to answer them. Every one of them helped and encouraged me. Within a week, I was plowing through a pile of information and suggestions.I encourage everyone to ask questions, but keep in mind that you should be willing to reciprocate.

My basic approach was pretty sound. I really like the way Softimage handles skeletal deformation. It allows you to individually weight a vertex in a mesh to any bone in a skeleton. These weights represent the degree of influence each bone has on the final position of that vertex. This allows me a much greater degree of control than if I were working with a system that only had a sphere of influence with a falloff. My research convinced me that if I were to build a real-time system for displaying these weighted meshes, I could create quite compelling 3D characters. As a bonus, I could use the weighting interface from Softimage and preview how the animated character would look in the game.

For my sample mesh, I created a two-bone hierarchy to represent my arm (Figure 1). The blue bone represents the upper arm, and the red bone represents the lower arm. I then attached the mesh for the arm to this two-bone hierarchy (Figure 2). Applying the weights to each vertex and rotating the lower arm produced a deformed mesh (Figure 3). I took special care in weighting the vertices near the joint between the two bones. If I allow one bone to completely influence (weight 100 percent) the vertex position, then it's possible that in certain orientations, the mesh will fold in on itself. You'll achieve better results when each bone contributes to the final vertex position. Figure 4 shows the Softimage interface for editing vertex weights. This is an example of an individual vertex being weighted between two bones.

Once I'd the weighted mesh, I needed a way to perform the deformation. I created the prototype for the deformation engine in OpenGL. OpenGL can get this type of tool up and running very quickly. From there, you can easily port the routines over to the API or platform of your choice. As an added benefit, the resulting image in the tool is identical to the preview window in Softimage because Softimage uses OpenGL for its real-time display. When you're trying to develop and debug pathways, this eliminates one source of image problems.

DisplayLists has proved itself very effective for drawing static geometry. For my application, I wanted to display the bones in the user interface in the same way that Softimage draws them (as a diamond shape). The code to create the bone geometry DisplayList appears in Listing 1. The routine descends the hierarchy and creates the diamond-shaped display if a bone has a child. I used the y translation element of the child to determine how long the bone should be. To make it easy to access the DisplayList later, I used the ID for the bone as the list number. You can see the results of a two-boned arm hierarchy in the OpenGL application in Figure 5.

Matrix Fun

My process for deforming the mesh was simple. I calculated the position of each affected vertex as if it were completely under the influence of each bone. I then used the weighting values to interpolate between these positions. Let's look at that in different terms.

For each vertex
finalPosition =
(position[1] * weight[1]) +
(position[2] * weight[2]) + …

where each position[N] is the initial position of that vertex multiplied by the transformation matrix of bone N.

However, to efficiently calculate the position of each vertex as it would be transformed by each bone, I needed to know each transformation matrix in the hierarchy. I obviously didn't want to recalculate the matrix for each bone at every vertex. So during an initial pass, I stored the transformation matrix as it accumulated down the hierarchy. The OpenGL method of handling a hierarchy of transformations via a matrix stack is very efficient — as you may remember from my article on motion capture ("Working with Motion Capture File Formats," January 1998).

The call to get the current matrix is

```
glGetFloatV(GL_MODELVIEW_MATRIX, float
*matrix);
```

This returns the sixteen values that make up the current matrix. The values are laid out as follows:

$$M(v) = \begin{bmatrix} 00 & 04 & 08 & 12 \\ 01 & 05 & 09 & 13 \\ 02 & 06 & 10 & 14 \\ 03 & 07 & 11 & 15 \end{bmatrix}$$

Note that this representation (called column-major) is different from many matrix routines you may see (usually row-major). Because of this difference,

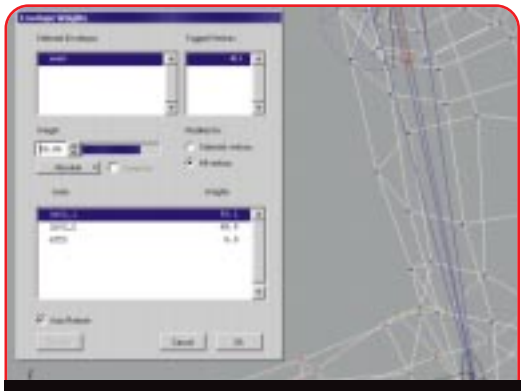


FIGURE 4. *Softimage interface for editing vertex weights.*

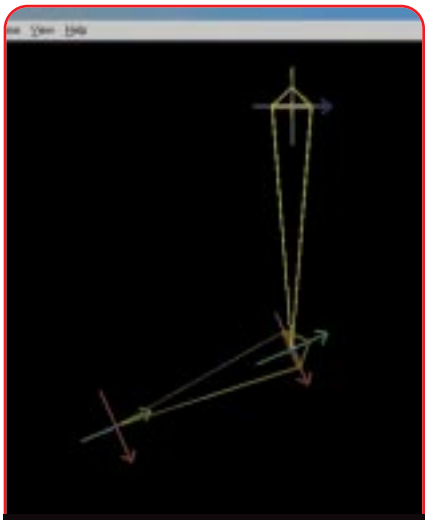


FIGURE 5. *Displaying the skeleton.*

if you declare the matrix structure to be **float matrix[4][4]** you'll get the wrong result when trying to access the data in C. It's much better to use **float matrix[16]** for your matrix storage. This is how OpenGL handles this procedure.

The matrices are stored in the bone structure for use when I actually calculate the positions. I have found this routine to be notoriously slow in OpenGL implementations. However, since the call to get matrix is only done once per frame for each bone in the system, it's not a big problem. For speed-critical applications, it may be wise to create your own matrix stack and matrix routines to speed up this process.

The code for saving the matrices is in Listing 2. It's a recursive call that will descend the hierarchy. At each node, it will draw an axis at the root of that bone. If that bone has a child, it will draw the bone geometry that I created earlier and highlight any selected bone. Note that the transformation operations are called in reverse order. OpenGL handles matrix operations this way. You may need to change this if you use a different API. This OpenGL feature causes a great deal of confusion for many people starting to work with the API.

At this point, I had to calculate the positions for each vertex. I could have called a **glLoadMatrix** for each bone. However, the call to **glLoadMatrix** is particularly slow (it would be called for each bone for each vertex). I wanted to avoid this one, because it would be called for each bone for each vertex. I could've avoided the issue by multiplying out all the vertex positions by each bone and storing all the intermediate results. However, that approach can be a huge memory issue for a mesh of significant size, so I decided against it. I chose instead to implement my own **MultVectorByMatrix** routine to calculate the intermediate positions. The drawback to this method is that you loose any benefit from 3D transformation hardware that you may have. This isn't an issue for consumer 3D hardware cards because they don't have hardware transformation acceleration. This may change in the future or in specific applications, so you'll have to evaluate the costs and benefits for yourself.

I multiplied each vertex by the matrix for every bone that vertex influences. I needed to subtract the root position of each bone from the vertex first in order to be sure that it was rotated about the bone's base. I calculated this distance back in during the matrix multiplication. I combined the results of all these calculations using the weight values to arrive at the final position for each vertex. I now have a mesh that is deformed in world space according to the settings of the bones controlling it. This mesh can be drawn as any other 3D mesh object. You can see the results in a standalone OpenGL application in Figure 6.

FIGURE 6. *OpenGL deformed mesh.*

Is it Worth it?

I now have a mesh object that can be deformed realistically in real time. This realism adds a lot of flexibility to your application. You can use it to create very compelling characters that react to their environment. But all this flexibility comes at a cost. Each vertex that is affected by more than one bone requires more calculations. The interpolation code and all the extra matrix handling add to the burden. I wouldn't even use these techniques on an enemy character whose entire motion sequence is finite and scripted. However, for a key character who can make unique moves that react to the people and environment around him, it's well worth it. The image quality generated by a weighted, deformed, single mesh is significantly

LISTING 1. *Displaylists for skeleton.*

```
////////////////////////////////////////////////////////////////////////////
// Function:        CreateBoneDLists
// Purpose:         Creates the Displaylists for the Bones in a Skeleton
// Arguments:       Pointer to a bone hierarchy
////////////////////////////////////////////////////////////////////////////
void COGLView::CreateBoneDLists(t_Bone *bone)
{
   // ONLY MAKE A BONE IF THERE IS A CHILD
   if (bone->childCnt > 0)
   {
      // CREATE THE DISPLAY LIST FOR A BONE
      glNewList(bone->id,GL_COMPILE);
         glBegin(GL_LINE_STRIP);
                  glVertex3f( 0.0f,  0.4f, 0.0f);                    // 0
                  glVertex3f(-0.4f,  0.0f,-0.4f);                    // 1
                  glVertex3f( 0.4f,  0.0f,-0.4f);                    // 2
                  glVertex3f( 0.0f,  bone->children->trans.y, 0.0f);          // Base
                  glVertex3f(-0.4f,  0.0f,-0.4f);                    // 1
                  glVertex3f(-0.4f,  0.0f, 0.4f);                    // 4
                  glVertex3f( 0.0f,  0.4f, 0.0f);                    // 0
                  glVertex3f( 0.4f,  0.0f,-0.4f);                    // 2
                  glVertex3f( 0.4f,  0.0f, 0.4f);                    // 3
                  glVertex3f( 0.0f,  0.4f, 0.0f);                    // 0
                  glVertex3f(-0.4f,  0.0f, 0.4f);                    // 4
                  glVertex3f( 0.0f,  bone->children->trans.y, 0.0f);          // Base
                  glVertex3f( 0.4f,  0.0f, 0.4f);                    // 3
                  glVertex3f(-0.4f,  0.0f, 0.4f);                    // 4
         glEnd();
      glEndList();
      // CHECK IF THIS BONE HAS CHILDREN, IF SO RECURSIVE CALL
      if (bone->childCnt > 0)
         CreateBoneDLists(bone->children);
   }
}
```

better than a character composed of separate objects, or whose joints are simply skinned over. Also, since you only need to store the orientations of the base skeleton, you can save a lot of memory on animation over straight, predeformed, single mesh characters.

The Application

The sample application that accompanies the article allows you to play with a deformable mesh. You can control the orientation of the bones as well as adjust the weighting on individual vertices. This will allow those who don't have access to Softimage to adjust the weighting on a deformable mesh and see the results. The arm itself is composed of an interleaved array of triangles. Each triangle is vertex colored to create a realistic shaded look.

I also used OpenGL's feedback mechanism to allow you to select vertices. I don't have space to cover that now.

Next issue I will address feedback as well as some other user interface issues.

Grab the source and the executable at the *Game Developer* web site at www.gdmag.com. ■

RESOURCES

I never found any one source that applied to the techniques I was using, but many resources were very helpful. If you're interested in learning more, check out these publications:

Badler, Norman, et al. *Simulating Humans: Computer Graphics Animation and Control*. Oxford: Oxford University Press, 1993.

Badler, Norman and M. A. Morris. "Modelling Flexible Articulated Objects." Computer Graphics, Proceedings of the Online (1982): pp. 305-314.

Magnenat-Thalmann, N. and D. Thalmann. *Interactive Computer*. Upper Saddle River, N.J.: Prentice Hall, 1996.

Parke, Frederic and Keith Waters. *Computer Facial Animation*. Wellesley, Mass.: A. K. Peters, 1996.

Terzopoulos, Demetri, et al. "Elastically Deformable Models." Computer Graphics, Vol. 21, No. 4 (SIGGRAPH 1987): pp. 205-14.

Acknowledgements

Thanks to the many people who have contributed to my knowledge of these techniques and methods over the past six months. Here are a few of them: Paul Atkinson, Norman Badler, Paul Douglas, Chris Hecker, Hexapod, Frederic Parke, Demetri Terzopoulos, and Nadia and Daniel Thalmann.

LISTING 2. *Grabbing the* `ModelViewMatrix`.

```
////////////////////////////////////////////////////////////////////////////
// Function:        drawSkeleton
// Purpose:         Actually draws the Skeleton it is recursive
// Arguments:       None
////////////////////////////////////////////////////////////////////////////
GLvoid COGLView::drawSkeleton(t_Bone *rootBone)
{
/// Local Variables ///////////////////////////////////////////////////////
   int loop;
   t_Bone *curBone;
////////////////////////////////////////////////////////////////////////////
   curBone = rootBone->children;
   for (loop = 0; loop < rootBone->childCnt; loop++)
   {
      glPushMatrix();

      // Set base orientation and position
      glTranslatef(curBone->trans.x, curBone->trans.y, curBone->trans.z);

      glRotatef(curBone->rot.z, 0.0f, 0.0f, 1.0f);
      glRotatef(curBone->rot.y, 0.0f, 1.0f, 0.0f);
      glRotatef(curBone->rot.x, 1.0f, 0.0f, 0.0f);

      // THE SCALE IS LOCAL SO I PUSH AND POP
      glPushMatrix();
      glScalef(curBone->scale.x, curBone->scale.y, curBone->scale.z);

      // DRAW THE AXIS OGL OBJECT
      glCallList(OGL_AXIS_DLIST);
      // DRAW THE ACTUAL BONE STRUCTURE
      // ONLY MAKE A BONE IF THERE IS A CHILD
      if (curBone->childCnt > 0)
      {
         if (curBone == m_SelectedBone)
                  glColor3f(1.0f, 1.0f, 0.0f);   // Selected bone is bright Yellow
         else
                  glColor3f(0.4f, 0.4f, 0.0f);   // Selected bone is dull Yellow
         // DRAW THE BONE STRUCTURE
         glCallList(curBone->id);
      }

      // GRAB THE MATRIX AT THIS POINT SO I CAN USE IT FOR THE DEFORMATION
      glGetFloatv(GL_MODELVIEW_MATRIX,curBone->matrix);

      glPopMatrix();          // THIS POP IS JUST FOR THE SCALE

      // CHECK IF THIS BONE HAS CHILDREN, IF SO RECURSIVE CALL
      if (curBone->childCnt > 0)
         drawSkeleton(curBone);

      glPopMatrix();          // THIS POPS THE WHOLE MATRIX

      curBone++;
   }
}
//// drawSkeleton ////////////////////////////////////////////////////////
```