# Over My Dead, Polygonal Body

Amazing things have been created with computer graphics over the past several years. Visual effects companies are poised to tackle one of the most difficult challenges in computer-generated (CG) imaging: creating a realistic CG human character. However, in my opinion, it hasn't happened yet no matter how many times I see life-like organic aliens and robotic bipeds trotting all over the screen.

The trouble is, humans are tough to simulate, regardless of how long we take to create the image. It may be obvious, but the difficulty lies in the fact that we are all very familiar with how humans look. We see them all the time. In the morning, I see something resembling a human in the mirror while I shave. Living as I do along the coast, I often see more of the human form than is probably healthy both for my ego and my digestion. In everyday life, we see all varieties of people performing every imaginable action. Each of us is an expert in determining the believability of a CG human form. If the skin looks wrong or the motion looks stiff or the lip-synch is off, each of us screams, "Fake!"

But, these technical and artistic problems are solvable. The brilliant artists and technicians charged with making us believe will make sure that it happens. I have thought a bit about the problems this will present, however. I think about John Wayne. What if, while he was alive, he was asked to do a commercial for beer? Perhaps his answer would have been, "Over my dead body!" That used to mean something, that there was no way that you would ever get me to do something. A bit of stock video footage and some clever video processing tricks and voilà — John Wayne's dead body selling beer.

Now don't get me wrong. I see nothing wrong with a family member licensing the likeness of a relative for commercial or charitable purposes. I think most people would be glad to continue to provide a living for their family even after they have gone from this life. I just think technology has forced us to reexamine that particular ultimatum. Perhaps it is time for something like, "Over my dead body and virtually extinct telepresence." Kind of loses that Wild West charm, doesn't it?

## Bringing Them Back in Real Time

Concerns over these kinds of legal dilemmas are not going to stop me from doing my job, however. As a creator of real-time 3D graphics, I not only face the hurdles which confront my visual effects comrades, but I also need to make these realistic characters move fast. Fortunately, in this task, technology is on my side.

Realistic characters require lots of polygons to make them look realistic. Lots of polygons mean lots of vertices being transformed by an overwhelmed CPU. At Siggraph 1998 I began to see the emergence of transformation acceleration in the consumer graphics hardware space (see "Taking a Break for Siggraph," Graphic Content, October 1998). At that time, I thought it wouldn't be long before we would be able to take advantage of hardware acceleration for transformation and lighting (HW T&L) in our games. Well, Siggraph has come and gone again and we are starting to see this become a reality.

One consumer graphics chip company, Nvidia, has publicly committed to delivery of hardware transformation and lighting with their next generation of graphics chips. There are rumblings that others are likely to deliver on this promise as well. In fact, Microsoft is so certain that hardware T&L will be a reality in the consumer hardware market, they have included support for it in the next version of the DirectX game programming API, DirectX 7. Game developers who are scheduling projects for release in the next production cycle need to consider how their projects will handle hardware T&L.

## How Do We Deal with HW T&L?

Fortunately, the driver writers will do most of the work for us. Games using the transformation and lighting pipeline in OpenGL will take immediate advantage of the hardware if it's available. Now, with the introduction of DirectX 7, games supporting this API will also transparently benefit from the new hardware. By using the built-in transformation pipeline in either API, games will get faster.

This will naturally enable games to increase polygon counts without sacrificing performance. It also means that the load on the CPU will decrease,

*When not being frightened by the idea of virtual versions of himself, Jeff creates 3D graphics for Darwin 3D. Let him know he is just being paranoid by e-mailing him at jeffl@darwin3d.com.*

**FIGURE 1A.** *Modeling organic objects is a challenge for 3D programmers dealing with hardware T&L.*

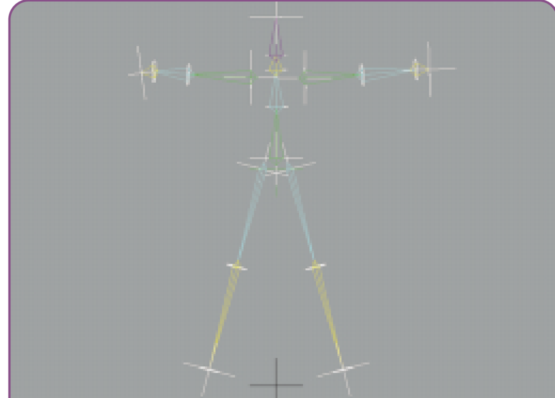

**FIGURE 1B.** *Skeletal deformation systems offer both flexibility and good looks.*



allowing more processor time for game play features such as artificial intelligence and game physics. And, as to be expected, creating content that performs well on a variety of user system configurations will be more important than ever.

**LISTING 1.** *Code for the* `World->RestBone` *matrix.*

```
GLvoid COGLView::GetBaseSkeletonMat(t_Bone *rootBone)
{
/// Local Variables ///////////////////////////////////////////////////////////
    int loop;
    t_Bone *curBone;
    tMatrix tempMatrix;
///////////////////////////////////////////////////////////////////////////////
    curBone = rootBone->children;
    for (loop = 0; loop < rootBone->childCnt; loop++)
    {
        glPushMatrix();

        glTranslatef(curBone->b_trans.x, curBone->b_trans.y, curBone->b_trans.z);

        // Set observer's orientation and position
        glRotatef(curBone->b_rot.z, 0.0f, 0.0f, 1.0f);
        glRotatef(curBone->b_rot.y, 0.0f, 1.0f, 0.0f);
        glRotatef(curBone->b_rot.x, 1.0f, 0.0f, 0.0f);

        // Grab the Matrix that is built up to this point
        glGetFloatv(GL_MODELVIEW_MATRIX,tempMatrix.m);
        // Invert this matrix to get the Base->World matrix
        InvertMatrix(tempMatrix.m,curBone->baseToWorldMat.m);

        // Recursive call if the bone has children
        if (curBone->childCnt > 0)
            GetBaseSkeletonMat(curBone);

        glPopMatrix();

        curBone++;
    }
}
```

Since we the graphics programmers can gain this benefit without doing any real work, hardware T&L doesn't seem to affect us much. The programmer will need to conform to a standard pipeline. For the transformation, this doesn't seem to be a big deal. Most 3D programmers are comfortable with matrix manipulation of vertices. The problem is pretty well solved and the only issue of trust revolves around the quality of the driver implementation. Once the hardware is doing the work, the driver issue largely goes away. Most will gladly accept this pathway to hardware nirvana.

Lighting, however is a much more contentious issue. No one I have talked to is satisfied with the lighting model provided by the OpenGL and DirectX libraries. This issue doesn't bug me that much. I don't know anyone who is ready to give up lighting tricks such as shadow maps and texture-mapped lighting for simplistic Gouraud lighting. While we do need a hardware solution for realistic lighting, this is not it. I advise continued reliance on those lighting tricks. However, as a supplement to pre-computed lighting for dynamic changes or shadow computation, a few hardware point and spot lights couldn't hurt, particularly if they're fast. I can think of a lot things that a few lights could be used for.

No, simple transformation and lighting is not the problem with hardware T&L. We run into the real trouble when we consider what it means for modeling organic objects.

## T&L for Non-Rigid Bodies

Hardware transformation is ideally suited to the display of rigid objects. You first set up the transformation matrix and then submit an object to be drawn. With hardware, it will be costly to obtain the results of the transformation

before the object is drawn. This means everything needs to be set up beforehand.

That's fine for most objects and environments. However, characters do not look their best when composed of rigid objects. As I have explored in a previous article ("Skin Them Bones," Graphic Content, May 1998), creating a character from a single mesh and then deforming it via a skeletal system provides a better solution. By using skeletal deformation, you maintain the good looks of a seamless mesh and the animation flexibility of a hierarchical character system.

Unfortunately, this system requires manipulation of vertex coordinates. Let me review how a skeletal deformation technique works. I have a character in a rest pose in Figure 1a and have created a skeleton for the character in Figure 1b.

In order to deform the mesh with the skeleton, I need to assign each vertex to a bone or set of bones. For example, all of the vertices in the head region should be assigned to the head bone. For now, let me assume that the vertices in the character's head are completely, or 100 percent, assigned to the head bone.

I can determine the position and orientation of the head bone at the rest frame by creating a matrix that represents the transformation needed to move that bone from the origin to its location in the hierarchy. The matrix of each bone is dependent on the matrices of all of its parents, so it is necessary to traverse the entire hierarchy to determine this `World->RestBone` matrix.

Now this matrix will take a vertex and transform it to the location of the bone. However, when I am taking the rest position of character, I will need to know how to take a vertex in the mesh and transform it back to the origin. Fortunately, since we are using a matrix operation here, this is a simple matter of inverting the `World->RestBone` matrix with a standard 4×4 matrix inversion routine. I now have a `RestBone->World` matrix. This only needs to be done once, so I can store this matrix for later use. You can see the code for computing the `RestBone->World` matrix in Listing 1.

I now have the matrix I need to move any vertex from the rest position back to the origin. Now I want to move the vertex to its final pose like the one in Figure 2. I can do this virtually the same way. I go through the hierarchy and create a matrix for each bone that will take a vertex from the origin and move it to the bone position in that animation frame. For this `World->Bone` matrix, I don't need to invert it.

I now have everything I need to take a rest vertex and move it to the final position. The procedure is as follows:

```
worldVertex = baseModelVertex * restBoneToWorldMat
deformedVertex = worldVertex * worldToBoneMat
```

There is an easy optimization step, though. Because two matrices can be multiplied together to create a matrix that is a sum of both transformations, I can create one matrix that will do everything:

```
combinedMatrix = worldToBoneMat * restBoneToWorldMat
```

Then for each vertex, I simply multiply it by that one matrix:

```
deformedVertex = worldVertex * combinedMatrix
```

This is the key to skeletal deformation. If you want to have multiple bones influencing a vertex, this final formula can be scaled by the amount of influence, or weight, of each bone. The sum of all the weights for each vertex should equal 1.

## What's the Problem?

This process requires each vertex to be transformed by a matrix for every bone that influences it. Clearly, this process should benefit greatly by the use of hardware transformations. However, this really breaks the transformation pipeline. It's obviously possible for a single polygon to have vertices that are influenced by multiple bones. Therefore, it is not possible simply to set the transformation matrix and submit a polygon, let alone an entire character.

In order for this to work, I would need to do the matrix operations first, combine the resulting vertices into a single polygon, and submit that to be drawn. However, the pipeline doesn't allow this. Getting the results of a transformation is not a fast process, as it requires values to be returned from the driver through a mechanism such as feedback.

This is precisely why it's crucial that this type of operation be handled by the driver via the graphics API. It's impossible

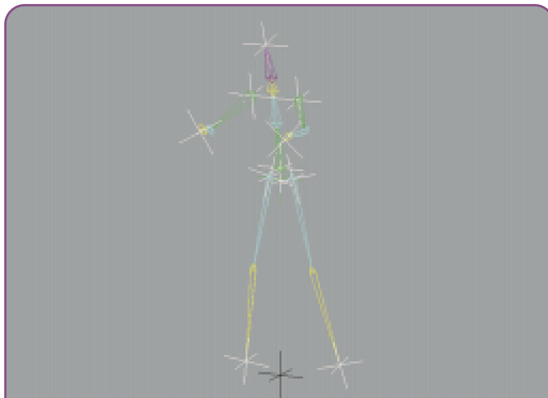

**FIGURE 2A.** *Our matrix enables us to move the skeleton's vertices to our character's final pose.*

**FIGURE 2B.** *Our finished model, the beneficiary of our transformation matrix.*

**LISTING 2.** *Vertex blending solves some problems, but creates others.*

```
typedef struct tVertex
{
  float x, y, z;
  float   weight;
  D3DCOLOR color;
  D3DCOLOR specular;
  float u;
  float v;
} tVertex;

#define FVF_VERTEX ( D3DFVF_XYZ | D3DFVF_XYZB1 | D3DFVF_DIFFUSE | \
                D3DFVF_SPECULAR | D3DFVF_TEX1 )

for (int loop = 0; loop < curBone->visuals[0].faceCnt; loop++)
{
   tFace *face;
   face = &curBone->visuals[0].face[loop];
   // There are two Matrices stored for each vertex
   d3ddev.SetTransform( D3DTRANSFORMSTATE_WORLD, face->matrix1);
   d3ddev.SetTransform( D3DTRANSFORMSTATE_WORLD1, face->matrix2);

   vertex = (tVertex *)curBone->visuals[0].vertex;
   HRESULT hr = d3ddev.DrawPrimitive(D3DPT_TRIANGLELIST, FVF_VERTEX, vertex, 3, 0 );
}
```



to perform this kind of deformation without breaking the transformation pipeline. Now many people are against the idea of adding API features for specific effects such as this, but I see no other way to achieve the goal. If the transformation could be handled as a specific DSP operation that was not tied directly to the display, it would be possible to have simple accelerated matrix operations. However, this is not the direction the hardware development is heading.

## What's Being Done About It?

Microsoft, while creating DirectX 7, realized that this would be an issue. To solve the problem, they created the notion of "vertex blending." In vertex blending, the final position of a vertex can be determined by the weighted transformation of up to four matrices.

You set up the matrices by creating a `transformstate` with the function:

```
SetTransform( D3DTRANSFORMSTATE_WORLD, matrix1);
SetTransform( D3DTRANSFORMSTATE_WORLD1, matrix2);
SetTransform( D3DTRANSFORMSTATE_WORLD2, matrix3);
SetTransform( D3DTRANSFORMSTATE_WORLD3, matrix4);
```

By using the flexible vertex format, you submit weights in each vertex structure. The number of weights is one less then the number of matrices being used. This is so that the API can enforce the constraint that the sum of the weights must equal 1.

For example, if I wanted each vertex to be weighted by two different bones, I would use code that looks something like Listing 2. There are some problems with this approach, however. First, the API supports blending of between one and four matrices. This leads to a content creation issue. If a particular card supports blending of only two matrices and your content was designed for blending four, you will need to clamp and scale the weights to work. This is yet another restriction for content creators.

Secondly, you set the matrices for each primitive instead of each vertex. Each primitive submitted to the rasterizer must be composed of vertices that are blended among the same bones. This can be a problem in certain regions such as the shoulder or waist where it would be quite easy to have each vertex influenced by a different bone.

Finally, to submit primitives efficiently for rendering, the model will need to be sorted on matrix usage. This may mean rearranging your dataset with a sophisticated export utility or custom optimization tool.

While it will be possible to create content to match these restrictions, it won't be easy. Naturally, artists won't enjoy being limited in their methods of weighting, and tools will need to be developed to handle the requirements.

## But I Don't Like Direct3D!

Unfortunately, at this time, there is no way to achieve a similar functionality in OpenGL using transformation hardware. The OpenGL community needs to step up and design an extension that will provide access to this hardware capability.

I would prefer an extension that provides more functionality with fewer restrictions. I imagine a vertex accumulation buffer much like the compiled vertex arrays we have now. In this version, you set a transformation matrix and submit a series of vertices with associated weight values. These are multiplied, scaled, and accumulated. Once all the vertices have been processed, the entire mesh is drawn with this accumulated vertex array.

This system would have no restrictions on the number of matrices in the blend. It would also have the side benefit of enabling many other interesting 3D effects such as morphing. I am not sure if this kind of extension could work with hardware as it exists now but I hope to find out. I'll keep you posted.

## What Are the Goodies?

I have provided a couple of demonstrations on the *Game Developer* web site (http://www.gdmag.com). Both of them allow you to manipulate a hierarchical skeleton to deform a 3D mesh. One was created using DirectX 7 and the vertex blending function, the other was created using OpenGL and implements the method I described above. Let me know how you think the algorithm can be improved. ■