In This Corner... The Crusher!

Everyone has their favorite memory of watching a classic Tex Avery or Chuck Jones cartoon. For many, these moments usually involve one of the characters being drastically deformed by a large and massive object. Simple 2D drawings made the physically impossible seem natural, believable, and fun for the kids.

Last month, I discussed how to use the power of new graphics hardware to give characters more life. Using matrix deformation techniques along with interpolated morphing of meshes has gone a long ways toward improving real-time character animation. However, creating the illusion of life needed for truly realistic characters requires more sophisticated techniques. According to John Lasseter (see References, p. 22), one of the chief advantages of computer animation is the ability to combine techniques in layers to achieve more complex and realistic results.

This idea of layering can be applied to real-time character animation to make characters more realistic. In the past in this column, I have talked about how a skeletal animation system is composed of a hierarchical structure of matrices (also referred to as bones) to provide the base animation layer for the character. The matrices are attributed {**Edit OK?**} to a mesh "skin" by vertex weight assignments that relate each vertex to matrices in the system. These matrices are then kinematically animated to provide the motion.

However, fine details in a matrix deformation system are difficult to achieve. In order to create detailed articulation, such as for fingers or facial expressions, a great many matrices must be used. This increases both the production time required for creating these characters and the processor time needed to render the character, thus reducing run-time performance.

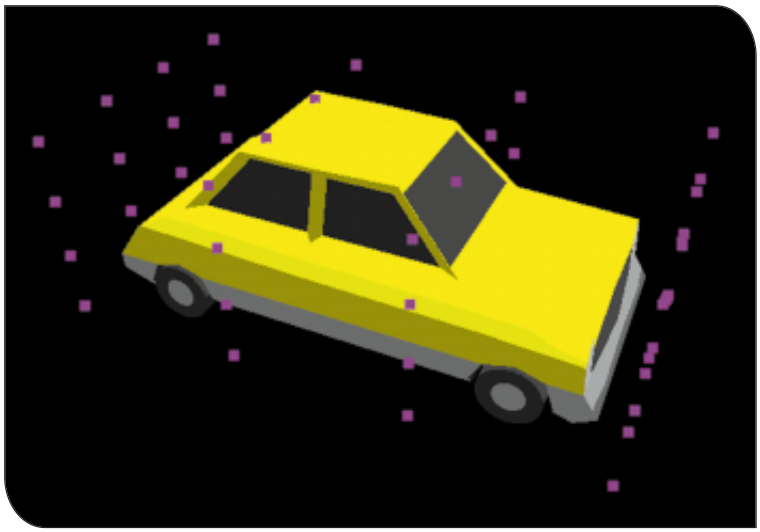

FIGURE 1. A free-form deformation lattice around a body.

AUTHOR'S BIO | *Cartoon inspirations come to Jeff after an afternoon at the brewery. If you have any visions you want to share, please contact him at jeffl@darwin3d.com.*

Vertex morphing techniques are a very useful animation tool for efficiently achieving fine detailed animation. Vertex morph animations for facial expressions and hand poses are easy to create and require minimal processing at run time. Typically, a single vertex morph target involves moving a very small subset of the vertices in a base mesh. The vertex morphing layer can provide the input to the skeletal animation layer, thereby providing a very flexible animation system.

Dropping a Virtual Anvil on My Characters

After a long afternoon of Cartoon Network research, I decided that it was time to combine my cartoon renderer (see "Shades of Disney: Opaquing a 3D World," Graphic Content, March 2000) with some animation techniques so that I can start smashing things up. In their compelling work *The Illusion of Life* (see References), Frank Thomas and Ollie Johnston outlined the use of squash and stretch, exaggeration, follow-through, and overlapping action as key components for character animation. The combination of a skeletal animation system with vertex morphing described above allows for a lot of character control. However, these animation controls are not well suited to creating characters that can dynamically squash and stretch. As I described last month ("To Deceive Is to Enchant: Programmable Animation"), the matrices in a skeletal animation system are full transformation matrices that can be translated, rotated, and scaled. These structures provide a great deal of local control over the vertices that the matrix influences. It certainly seems that matrix manipulation is a possibility for achieving some nice squishy effects. However, manipulating the control matrices individually can be tedious. Animators need a more intuitive parameterization of these properties in order to achieve fluid results.

Sederberg and Parry (see References) introduced the use of free-form deformations (FFDs) as an efficient method for animating soft bodies via a structural hyperpatch. By abstracting the control surface from the surface of the animated body, the deformation controls can be manipulated without regard to the model itself. This technique has been used successfully to model semi-elastic surfaces.

An FFD works by positioning a 4×4 lattice of control vertices (CVs) around the model you wish to deform, as you can see in Figure 1. This lattice is aligned along the global X-, Y-, and Z-axes for clarity. It also makes sense to align it with the model's principal axes. These CVs are the controls the animator (or simulation, but I don't want to get ahead of myself) will manipulate to animate the object.

In order for the control vertices to change the model, I need to establish a relationship between the control lattice and the model. For this I'm going to use a cubic Bézier volume. This structure is composed of a 3D lattice of Bézier curves of degree 3 (cubic) which share control vertices. This gives me a total of 64 control vertices in a 4×4×4 grid. (For more information on the mathematics of Bézier curves and patches, see Brian Sharp's article on curved surfaces as well as Alex Ferrier's introduction to free-form deformations on Gamasutra.com, both in the References box.)

LISTING 1. Converting the FFD function to vertex weights.

```
////////////////////////////////////////////////////////////////////
// Function:        SetFFDWeights
// Purpose:         Approximate an FFD by setting up control
weights
// Arguments:       Pointer to base mesh visual structure
////////////////////////////////////////////////////////////////////
void SetFFDWeights(t_ToonVisual *visual)
{
/// Local Variables ///////////////////////////////////////////////
   tVector *vertex;
   int loop,cvLoop;
   float  XBasis[4], YBasis[4], ZBasis[4];
   float  u, v, w;
   float  *vertexWeight;
   int    px,py,pz;
////////////////////////////////////////////////////////////////////
   // Allocate the space for all the weights
   visual->weightData = (float *)malloc(visual->vertexCnt *
      FFD_NODE_COUNT * sizeof(float));
      vertex = visual->vertex;
      // Go through all the vertices
      for (loop = 0; loop < visual->vertexCnt; loop++, vertex++)
      {
      // Find where each vertex is within the FFD grid
      // Effectively scales each vertex to 0-1
      u = (vertex->x - g_FFDmin.x)/(g_FFDmax.x - g_FFDmin.x);
      v = (vertex->y - g_FFDmin.y)/(g_FFDmax.y - g_FFDmin.y);
      w = (vertex->z - g_FFDmin.z)/(g_FFDmax.z - g_FFDmin.z);
```

continued on page 20

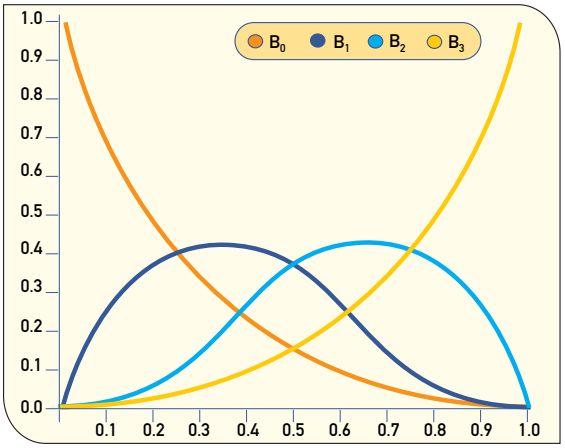

FIGURE 2. Bézier Basis Functions

To evaluate a Bézier curve, I need a function that takes a point along the curve I wish to evaluate and returns the position. There is a function called the Bernstein basis function that serves this purpose. For a cubic curve, it takes the form

$$B^3(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u)p_3 + u^3 p_4$$

where p_n represents the control vertices. To extend this function to a Bézier volume, it becomes a function of three variable (u,v,w) which represents the 3D position within the Bézier volume. The full formula for the Bézier volume basis function is:

$$B^3(u,v,w) = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3} p_{ijk} B_i^3(u) B_j^3(v) B_k^3(w)$$

It seemed to me that layering the FFD technique into my animation system would allow me to get the results I wanted. However, in order to deform a mesh using this FFD lattice, I need to pass every vertex through this function to evaluate its deformed position. That's 27 evaluations of the Bernstein basis function at every vertex. This amounts to quite a few calculations that would fall under the CPU's responsibility.

It occurred to me that the control vertices look an awful lot like matrices in my standard animation pipeline. Even more interesting is the fact that matrix deformation techniques can be accelerated by some graphics hardware. If I can frame the FFD problem in terms of a matrix deformation system, I can use this hardware to relieve the CPU and also streamline my animation pipeline.

The basis functions serve the role of relating the control vertices to the vertices in the base mesh. For matrix deformation, the vertex weights relate the control matrix in the same role as the vertex weights in a skeletal animation system {**Edit OK? I think a word got left out**}. If I treat each control vertex in the FFD lattice as a matrix, I need to create the weights to relate that matrix to the base mesh. This brings me back to the basis functions. Each control vertex in the control curve influences a certain portion along

LISTING 1 (continued). Converting the FFD function to vertex weights.

continued from page 18

```
      // X Bezier Basis Functions
      XBasis[0] = (1.0f - u) * (1.0f - u) * (1.0f - u);
      XBasis[1] = 3.0f * u * (1.0f - u) * (1.0f - u);
      XBasis[2] = 3.0f * u * u * (1.0f - u);;
      XBasis[3] = u * u * u;

      // Y Bezier Basis Functions
      YBasis[0] = (1.0f - v) * (1.0f - v) * (1.0f - v);
      YBasis[1] = 3.0f * v * (1.0f - v) * (1.0f - v);
      YBasis[2] = 3.0f * v * v * (1.0f - v);;
      YBasis[3] = v * v * v;

      // Z Bezier Basis Functions
      ZBasis[0] = (1.0f - w) * (1.0f - w) * (1.0f - w);
      ZBasis[1] = 3.0f * w * (1.0f - w) * (1.0f - w);
      ZBasis[2] = 3.0f * w * w * (1.0f - w);;
      ZBasis[3] = w * w * w;

      // Pointer to Place to store weight data
      vertexWeight = &visual->weightData[loop * 64];
      // Go through the control vertices
      for (cvLoop = 0; cvLoop < FFD_NODE_COUNT;
cvLoop++,vertexWeight++)
      {
         // Some quick math to find the component indices
         px = FFD_WIDTH - (cvLoop % FFD_WIDTH) - 1;
         py = FFD_HEIGHT - (cvLoop / (FFD_WIDTH * FFD_HEIGHT)) - 1;
         pz = FFD_DEPTH - ((cvLoop % (FFD_WIDTH * FFD_HEIGHT)) /
            FFD_WIDTH) - 1;

         // set the vertex weight for this CV
         *vertexWeight = (XBasis[px] * YBasis[py] * ZBasis[pz]);
      }
   }
}
/// SetFFDWeights ////////////////////////////////////////////////
```

the length of the curve. If I examine the influence each CV has on the curve I get the formulas:

$$B^3(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u)p_3 + u^3 p_4$$

$$B^3(u) = B_0^3 p_0 + B_1^3 p_1 + B_2^3 p_3 + B_3^3 p_4$$

$$B_0^3 = (1-u)^3$$

$$B_1^3 = 3u(1-u)^2$$

$$B_2^3 = 3u^2(1-u)$$

$$B_3^3 = u^3$$

These formulas show the influence each control vertex has on the curve, which you can see graphically in Figure 2. Each control vertex has an influence over a region of the volume. These functions will provide the vertex weighting data I need.

To attach a FFD lattice to an object, I take the base mesh and place it within the FFD lattice. The weights are calculated for each vertex by scaling the vertex position to a value between 0 and 1. This represents the relative position of the vertex within the lattice. The scaled value is plugged into the three basis functions for each control vertex and out pops a vertex weight that relates the mesh vertex to that CV. Listing 1 contains the code that calculates these weights. You can also see the influence of a single CV on the mesh in Figure 3.

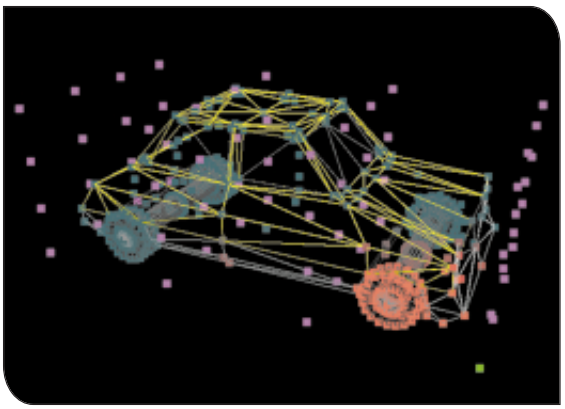

FIGURE 3. Weight values for a control vertex.

One restriction required in order for matrix deformation to work is that the sum of the weights on any one vertex must equal one. Fortunately for us, the wonders of mathematics are working in our favor. Due to the very nature of the Bézier basis functions, the sum of the influences at any point along the curve is always equal to one. You gotta love how well that works out.

Once these weight values are calculated for each vertex, I can run this object through my matrix deformation system and start moving control vertices around. As each CV moves, it deforms the base mesh through the weight values. In fact, I get a bonus over the traditional FFD system. I can apply other transformations on these control points. I can rotate and scale them, giving me even more control over the mesh. However, I still need to move each CV individually to make anything happen. I will need to add a control mechanism to make them move together.

Controlling the Squishy Beast

For many applications, manually positioning the FFD control vertices will work fine. Often, though, I will want them all to move together like a single flexible object. Fortunately, I have played around with something like this in the past. You may recall my column last year on the topic of soft-body dynamics ("Collision Response: Bouncy, Trouncy, Fun," Graphic Content, March 1999). In that column, I connected point masses together using dampened springs. I could then toss those objects around and they bounced off the walls and floor in a flexible manner.

For this application, I will make the point masses the control vertices in my FFD lattice. I then connect those points together with a network of springs in the same way as I did in my March

1999 column so the lattice will be somewhat stable when I drop it. When I run the object through my particle dynamics simulator, the control vertices start bouncing around. Since these control vertices are used to deform the base mesh, the mesh bounces along also. Just for fun, I applied the cartoon shader to the objects so I can really get that Saturday morning, "bang him on the head with a skillet" feel.

You can see the application in action in Figure 4. I loaded in my cartoon car and bashed around some of the CVs to flatten the roof.

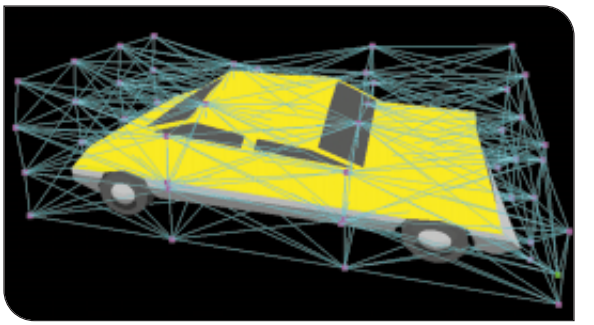

FIGURE 4. A taxicab in a meteor storm.

Putting It All Together

Adding this technique for using FFD lattices in a character animation system opens up some interesting possibilities. The FFD can be positioned in the skeletal hierarchy such that transformations are inherited from parent matrices. That way an FFD lattice can be applied, for example, to an upper arm so that the muscles will bulge. You would need to be careful with how the weights blend across the FFD and skeletal links. However, I have found that scaling and blending weights works very well.

There are a few problems with the use of the mass-and-spring system for FFD animation. My current system does not preserve the volume of the original control mesh. What that means is that the spring system can find valid configurations where it has collapsed inside itself. This may not be totally realistic for certain solid but flexible objects. For an object like my car, though, it works in my favor. I sometimes want the object to collapse inside itself and stay there. If such behavior were not desirable, however, I could add a lot more springs to the control mesh or I could use another method for dynamically connecting the control points that preserves the volume of the object. I may take a look at that issue in another column.

Another problem is a rendering one. When I deform the mesh, I am moving vertices all around. This changes the surface quite a bit and I am not currently adjusting the surface normals to match, which causes some problems with the shading model. Unfortunately, this is a tricky problem. To fix it, I would need to rebuild the vertex normals by creating new face normals and averaging them to get new vertex normals. This is processor-intense, but not terribly hard to code up. I leave that up to industrious readers to add to the sample application.

As a second layer of abstraction, it would be interesting to make the FFD lattice deform a skeleton inside the object instead of individual vertices in the mesh. That would definitely speed up the calculations, as there would be a lot fewer points to process. However, it would also lower the amount of control.

There's also nothing stopping you from using lower- or higher-degree Bézier volumes. I chose cubic because it seemed to provide a good flexibility-to-performance tradeoff. For other objects, more or less control may be needed.

Another interesting extension to this technique would be to make some of the lattice springs active instead of passive, thereby creating virtual "muscles" that animate the object automatically. I will look more into that next month. For now, grab the application and source code at www.gdmag.com and start smashing things around.

CLEARING OUT MY E-MAIL BOX

One of the fun things about writing this column is all the great mail I get from readers. For anything I wonder about or miss, one of you always is quick to let me know about it.

In my last December's column on 2D water effects ("A Clean Start: Washing Away the Millennium"), I mentioned that I didn't know who wrote the original version of this idea. Several people wrote in, the first being Juan Carlos Arevalo Baeza who wrote that the effect was done for Heartquake, a demo entered in a competition {Edit OK?} in Helsinki by Arturo Ramirez-Montesinos of the demo group Iguana in 1994. He based the idea on a crude approximation of the 2D general wave propagation formula. You can get the original demo at ftp://x2ftp.oulu.fi/pub/msdos/programming/iguana/heartq.zip.

I also received a note from Fabio Policarpo who is working on a book on game programming with Alan Watt. He integrated the cartoon rendering technique into his Fly engine that will be the basis of the book. He has a great demo of a cartoon car driving around a terrain. You can download the demo at www.paralelo.com.br/download/car-toon.zip.

REFERENCES

- Lasseter, John. "Principles of Traditional Animation Applied to 3D Computer Animation." *Proceedings of Siggraph '87*. In *Computer Graphics* (Vol. 21, No. 4): July 1987.
- Thomas, Frank, and Ollie Johnston. *Disney Animation: The Illusion of Life*. New York: Abbeville Press, 1984.
- Sederberg, Thomas, and S. R. Parry, "Free Form Deformations of Solid Geometric Models." *Proceedings of Siggraph '86*. In *Computer Graphics* (Vol. 20, No. 4): August 1986.
- Sharp, Brian. "Implementing Curved Surface Geometry" (*Game Developer*, June 1999).
- Ferrier, Alex. "Real-Time Soft-Object Animation Using Free-Form Deformation." www.gamasutra.com/features/19990827/deformation_01.htm.
- Chadwick, John, and others. "Layered Construction for Deformable Animated Characters." *Proceedings of Siggraph '89*. In *Computer Graphics* (Vol. 23, No. 3): August 1989.