Mighty Morphing Mesh Machine

Most game people have played around with those programs that will morph between two graphic images. They allow you to take a picture of your brother and a picture of a freckle-faced baboon, play with the sliders, and enjoy hours of endless amusement.

Now, think about the fun you could have morphing between a 3D model of your brother and the monkey — that could make your week. Most 3D animation packages will allow you to do this, but what you might really be looking for is a real-time 3D demo featuring your brother in some sort of failed lab experiment that you could mercilessly blow away. Or something like that…

To engage in this advanced level of fun, you need a method for morphing between two 3D shapes. This technique is not only amusing, but also quite useful. 3D morphing can help create organic animation that would otherwise be difficult to develop.

Morphing and Real-Time 3D Animation

Another use for 3D morphing is smoothing between keyframes. In games such as QUAKE 2, you may read that the animation in-betweens are interpolated. I have discussed before how in QUAKE (and many 3D action games) each frame of animation is actually represented by an individual mesh. By sequencing through those meshes, the illusion of animation is created. However, the original object frames are created at a set frame rate, say 10 frames per second (FPS). This means that the smoothest those animations can play back is at that original 10 FPS. If, for example, the engine is actually displaying at 60 FPS, each frame of character animation is held for six frames. That's quite a wasted opportunity. Smoother animation could be achieved by morphing from one frame to the next over those six frames. That is exactly what the "feature hypers" (you know, the feature-happy marketing dudes) are talking about when they talk about interpolated in-betweens. But what exactly is being interpolated?

LERP = Morph

This is an important equation in the game programmer's arsenal. Many programmers who have been working in 3D for some time take this for granted as widely known information. But judging by the mail I get and the comments I see in the public forums, there are many individuals who aren't up-to-speed on how morphing works. The secret to object morphing is that the only thing that changes between the two models is the vertex positions in the object (a little white lie — I'll get to the truth later). When you morph between 3D objects, you are doing a "linear interpolation," or LERP, between the vertex positions in the two objects. For this technique to work, the two models you are morphing between must have identical vertex counts, and the vertices must correspond to each other. This means that vertex 1 on the first model should end up in the position of vertex 1 in the second model. The easiest way to make sure that the models are created correctly is to actually create the second model by moving the vertices in the first model into the shape you want. That way, the models will interpolate exactly the way you created them. By now you have the models and want to morph between them. The formula for a LERP between two values is

```
InBetween = Value1 + ((Value2 - Value1) *
   lerpValue);
Where lerpValue is a float between 0 and 1.
```

Now, this formula needs to be applied

FIGURE 1A. *Your model at rest, for the first frame of animation.*

FIGURE 1B. *The same model, frame two. You'll get smoother animation by morphing between frames.*

Morphing between a lounging postion on the beach and a slave position at his desk, Jeff can be found at Darwin 3D working on real-time game technology. Email him at jeffl@darwin3d.com with suggestions for the next keyframe.

to every parameter that will vary during the morph. For a 3D vertex coordinate, those parameters would be the x, y, and z values for that point. This is where I explain the truth behind that little white lie. There may be more parameters than just the coordinates which you wish to interpolate for an individual vertex. If your game engine supports real-time lighting and your model data contains vertex-normal information, for example, you may want to interpolate the normal values as well. Also, if your model contains vertex color information, an interesting effect may be created by interpolating the color.

What about textures? If your 3D model contains texture coordinates, you may want those coordinates to change over time as well. However, there are many possible pitfalls associated with morphing textures. Small changes in UV coordinates can cause a major shift in the appearance of a model that may not be desired. Likewise, it may be more interesting to change the texture completely for the second position. This complicates things a bit. However, if the UV coordinates stay constant throughout the morph and only the texture changes, image processing techniques can help. By using a 2D dissolve to create a blended image between both textures, this will smoothly change along with the model. These blended textures could either be prebuilt as a texture animation or actually created on-the-fly, if possible. Multitexture hardware can provide a hardware-accelerated method for blending between two textures via methods such as the **D3DTOP_BLENDFACTORALPHA** and **D3DTOP_BLENDDIFFUSEALPHA** in DirectX 6 (See "Multitexturing in DirectX 6," *Game Developer*, September 1998). With multitexture hardware

LISTING 1. *The morph code.*

```
#define LERP(a,b,c)  (a + ((b - a) * c))
////////////////////////////////////////////////////////////////////////
// Procedure: morphModel
// Purpose:   Does the Morph for the Model
// Arguments: Pointer to main bone
////////////////////////////////////////////////////////////////////////
GLvoid CDGLView::morphModel(t_Bone *curBone)
{
/// Local Variables ///////////////////////////////////////////////////
   int loop,pointloop;
   float *dest,*src1,*src2,ratio;
////////////////////////////////////////////////////////////////////////
   if (curBone->visualCnt > m_curVisual)
   {
      src1 = curBone->visuals[0].vertexData;       // FRAME 1
      src2 = curBone->visuals[1].vertexData;       // FRAME 2
      dest = curBone->visuals[2].vertexData;       // DESTINATION FOR MORPHED FRAME
      ratio = m_Slider->GetSetting();              // GET MORPH VALUE (0 - 1)
      // LOOP THROUGH THE VERTICES
      for (loop = 0; loop < curBone->visuals[0].triCnt * 3; loop++)
      {
         // GO THROUGH EACH ELEMENT IN THE VERTEX STRUCTURE
         for (pointloop = 0; pointloop < curBone->visuals[0].vSize; pointloop++)
         {
            // THE NEW POSITION IS A LERP BETWEEN THE TWO POINTS
            dest[(loop * curBone->visuals[0].vSize) + pointloop] =
               LERP(src1[(loop * curBone->visuals[0].vSize) + pointloop],
                    src2[(loop * curBone->visuals[0].vSize) + pointloop],ratio);
         }
      }
   }
}
// morphModel
```

TABLE 1. *A comparison of different formats.*

Format	Used By	Exported By	Imported By	Normals	Poly Color	Vertex Color	UV Coords	Animation	Hierarchies	Ascii	Ease of Use (10 = Easy)
.3DS	3DS R4	1,2,8	1,2,8	X	X	X	X	X	X		6 (w/ KTX lib)
.3DS Ascii	3DS R4	1	1	X	X	X	X	X	X	X	6
.DXF	Autocad	1,2,3,4,6,7,8	1,3,5,6,7,8	X	X					X	7
LWOB	Lightwave	5	5	X	X			X			5
.OBJ	Wavefront	2,3,6,7,8	2,3,6,7,8	X	X	X	X	X			10
Game Exchange	Nichimen	7	7	X	X	X	X	X	X	X	9
.HRC	Softimage	8	8	X	X	X	X	X	X		1
MAX	3DS MAX	2	2	X	X	X	X	X	X		0 (Max only)
Maya ASCII	Maya	6	6	X	X	X	X	X	X	X	8
VRML	VRML	2,7,8	2,7,8	X	X	X	X	X	X	X	6
.X	Direct X	2,7,8	2,7,8	X	X		X	X	X	X	7

Program Names:
1 .3DS R4
2 .3DS MAX
3 Alias
4 Hash
5 Lightwave
6 Maya
7 Nichimen
8 Softimage

becoming more common, this could be an area to add value to your hardware-accelerated application.

This algorithm is actually very easy to get up and running. You can see the code for a 3D morph in Listing 1. One easy optimization to make would be to precalculate the deltas between each parameter to remove a subtraction operation. In the case of an animation system, predividing the deltas by the number of desired in-betweens would turn it into a pure addition operation.

That's all there is to 3D morphing. Given how easy a 3D morph actually is to implement, I wonder why we don't see it more in real-time 3D games. There are many uses for this technology. Beyond creating in-betweens for character animation, morphing is an excellent way to change the shape of characters. It can also be used to create facial animation and other special effects. Hopefully, we will start to see more in the next generation of real-time projects.

Getting Your Model Data

So, you're ready to start morphing every object you can get in your hands. That brings up an important question. How do you get your hands on model data? When it comes to game companies with staff artists and tool programmers, many rely on high-end animation packages with SDKs. These toolkits allow the programmers to write plug-ins that allow them to get to the data directly. This is not always possible. Some programmers do not have the money, time, or experience to get models this way. The other option is to use a public 3D file format. The ideal file format contains all the information that the application requires and is easy to use. The ideal format is also public, meaning that information is publicly available describing the format. Code samples are even more desirable.

So, which file format is the best one for you? Table 1 (see page 16) contains a list of several file formats along with some information about them. This is by no means comprehensive, but the list offers a glimpse of what's out there. Ease-of-use is an opinion gathered from my own experience and from discussing it with other programmers.

The key to finding a format to support in a custom tool is to pick a format that contains all the information that is critical to your application. If your game engine uses vertex coloring, make sure the format supports that feature. It's also helpful to pick a format that is supported by your or your artist's favorite art tool. There are commercial 3D file converters available, but they add cost and an additional step to the production process. Also, using an ASCII format makes checking your data and debugging the process much easier.

I have found that one of the easiest to use and most widely supported formats is the Wavefront .OBJ format. It contains support for UV coordinates as well as for vertex normals. The format is so easy that you really don't even need a spec to write a file loader. Listing 2 shows a simple cube in .OBJ format. Lines that begin with the pound sign **#** are comments and can be ignored. The initial **o cube1** defines the name of the object. That is followed by the **mtllib cube.mtl.** This line describes a file that contains material information about the object. But more on that later.

The next block of information actually describes the vertices. Each line starts with a **v** and is followed by the x, y, and z coordinate values. The comment at the end actually tells you the number of vertices. But, that doesn't seem to be a standard feature of this format, so you shouldn't count on it.

The **vn** block gives the x, y, z, values for the normals in the model and the **vt** block describes the texture coordinates. The texture coordinates can be either two coordinates (u and v) or three (u, v, and w). For most real-time applications, however, the **w** value could be ignored. A 3D model may have normals or texture coordinates, or both, or neither, but it obviously

LISTING 2. *An OBJ Cube.*

```
o cube1

mtllib cube.mtl

v -5.000000 -5.000000 -5.000000
v -5.000000 -5.000000 5.000000
v -5.000000 5.000000 -5.000000
v -5.000000 5.000000 5.000000
v 5.000000 -5.000000 -5.000000
v 5.000000 -5.000000 5.000000
v 5.000000 5.000000 -5.000000
v 5.000000 5.000000 5.000000
# 8 vertices

vn  -1.000000 0.000000 0.000000
vn  0.000000 0.000000 1.000000
vn  1.000000 0.000000 0.000000
vn  0.000000 0.000000 -1.000000
vn  0.000000 -1.000000 0.000000
vn  0.000000 1.000000 0.000000
# 6 normals

vt  0.000000 0.000000
vt  0.000000 1.000000
vt  1.000000 0.000000
vt  1.000000 1.000000
# 4 texture vertices

usemtl mat1_FACE
f 1/2/1 2/4/1 4/3/1
f 1/2/1 4/3/1 3/1/1
f 2/2/2 6/4/2 8/3/2
f 2/2/2 8/3/2 4/1/2
f 6/2/3 5/4/3 7/3/3
f 6/2/3 7/3/3 8/1/3
f 5/2/4 1/4/4 3/3/4
f 5/2/4 3/3/4 7/1/4
f 5/2/5 6/4/5 2/3/5
f 5/2/5 2/3/5 1/1/5
f 3/2/6 4/4/6 8/3/6
f 3/2/6 8/3/6 7/1/6
# 12 elements
```

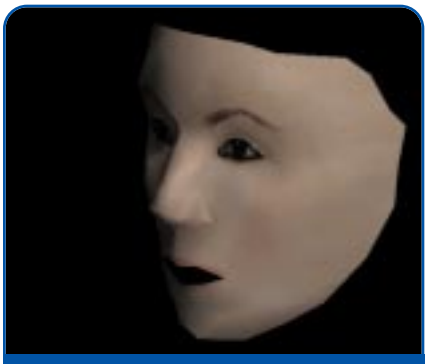

FIGURE 2A. *Your "brother" mapped onto a 3D model.*

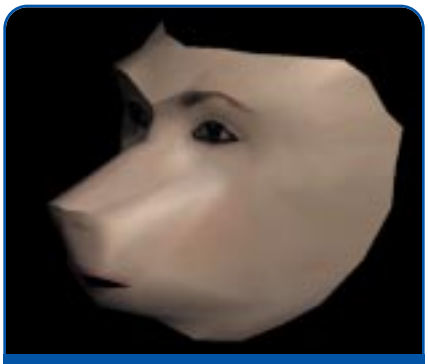

FIGURE 2B. *The original figure morphed into a baboon.*

must have the vertex values.

The final block in the file begins with the **usemtl mat1_FACE**. This says to the loader, from now on all faces defined should use the **mat1_FACE** material. This material is defined in the **cube.mtl** file. All lines that begin with an **f** describe a face in the model. Each face can be composed of multiple vertices. Each face is not required to have the same number of vertices. However, because it is more efficient for 3D hardware if all faces have the same number of vertices, I make sure this is the case. I could tessellate the face to triangles at run time, but this is really easy to do in the modeling program. Therefore, I just make it a requirement that all models are triangulated before exporting. A pop-up box in the loader can warn users that a face is not triangulated.

In the face statement each vertex is defined by three elements separated by forward slashes that describe the vertex, texture coordinate, and normal for that face. The values are indices into the list of elements already defined. It's important to notice that these indices are one-based instead of zero-based. In a file that only has vertex coordinates (and not normals or texture vertices), the vertex index will be followed by two slashes as in **f 1// 2// 3//.** Likewise, if there is a vertex and a normal, the format is **f 1//1 2//2 3//3** and so on. Each vertex in the line is separated by a space.

You can see a material file in Listing 3. The file can describe multiple materials. Each one begins with **newmtl** and the name of the material in this case **mat1_FACE.** The next lines **Ka**, **Kd**, and **Ks** respectively describe the ambient, diffuse, and specular color for the material. The **Ns** term describes the specular highlight. I have never had a need for the**Ni** and **illum** term, though they are there if you want them. Finally, the **map_Kd** describes the diffuse map applied to the object. In other words, this is the texture map that should be applied to the surface. I use this name as the name of the file loaded by the application. I just convert the image to a .TGA or .BMP file to make use of existing file loading code.

See there, I said it was an easy format. Actually there are other blocks that may be useful in a real-time simulation that are not in my sample file. The **g** command allows faces to be grouped together so the file can contain multiple objects even in a hierarchy. This is definitely handy when working with hierarchical characters.

Writing a .OBJ File Loader

The MFC **CString** class makes string manipulation much easier than in basic C. For custom tools, this means loading ASCII file formats is easier than ever. My strategy is to load a line of text from the file and then break it up into a string of words in a **CStringArray** structure. If you haven't used the **CString** class, it will bring back fond memories of BASIC string handling.

I don't want to go through the entire .OBJ loader here. You can just grab it off the web site (http://www.gdmag.com). However, my strategy for loading an .OBJ file is to pass through the file once, determining how many vertices, normals, and texture coordinates for which I need to allocate space. Then, all the actual coordinate values are simple to load in. The only tricky part comes in when I want to load in the face indices. You can see how I approached that in Listing 4.

The point of this loader is not for me to show highly optimized, well formulated code samples for loading these files. My code here certainly is not fine-tuned in any sense of the word. The great thing about creating production tools is that, unlike almost all other

LISTING 3. *MTL file for the Cube.*

```
newmtl mat1_FACE
Ka 0.5000 0.5000 0.5000
Kd 0.7000 0.7000 0.7000
Ks 1.0000 1.0000 1.0000
Ns 50.0000
Ni 1.0000
illum 2
map_Kd FACE.pic
```

LISTING 4. *Handling a face line in an .OBJ.*

```
////////////////////////////////////////////////////////////////////////////
// Procedure: HandleFace
// Purpose:   Handles the Face Line in an OBJ file.  Extracts index info to
//            a face Structure
// Arguments: Array of words from the face line, place to put the data
// Notes:     Not an Official OBJ loader as it doesn't handle more then
//            3 vertex polygons.  This only handles Triangles
////////////////////////////////////////////////////////////////////////////
void HandleFace(CStringArray *words,t_faceIndex *face)
{
/// Local Variables ///////////////////////////////////////////////////////
   int loop;
   CString temp;
   CString vStr,nStr,tStr;     // HOLD POINTERS TO ELEMENT POINTERS
   int nPos,tPos;
////////////////////////////////////////////////////////////////////////////
   // LOOP THROUGH THE 3 WORDS OF THE FACELIST LINE, WORD 0 HAS 'f'
   for (loop = 1; loop < 4; loop++)
   {
      temp = words->GetAt(loop);        // GRAB THE NEXT WORD
      // FACE DATA IS IN THE FORMAT vertex/texture/normal
      tPos = temp.Find('/');            // FIND THE '/' SEPARATING VERTEX AND TEXTURE
      vStr = temp.Left(tPos);           // GET THE VERTEX NUMBER
      temp.SetAt(tPos,' ');             // CHANGE THE '/' TO A SPACE SO I CAN TRY AGAIN
      nPos = temp.Find('/');            // FIND THE '/' SEPARATING TEXTURE AND NORMAL
      tStr = temp.Mid(tPos + 1, nPos - tPos - 1);     // GET THE TEXTURE NUMBER
      nStr = temp.Right(temp.GetLength() - nPos - 1); // GET THE NORMAL NUMBER
      face->v[loop - 1] = atoi(vStr);       // STORE OFF THE INDEX FOR THE VERTEX
      face->t[loop - 1] = atoi(tStr);       // STORE OFF THE INDEX FOR THE TEXTURE
      face->n[loop - 1] = atoi(nStr);       // STORE OFF THE INDEX FOR THE NORMAL
   }
}
///// HandleFace /////////////////////////////////////////////////////////
```

UNITED STATES POSTAL SERVICE

Statement of Ownership, Management, and Circulation (Required by 39 U.S.C. 3685)

1. Publication Title	2. Publication No.	3. Filing Date
Game Developer	1 0 7 5 - - - 9 2 0 0	1-Oct-98
4. Issue Frequency	5. No. of Issues Published Annually	6. Annual Subscription Price
Monthly	12	$34.95

7. Complete Mailing Address of Known Office of Publication (Street, City, County, State, and ZIP+4) (Not Printer)
Miller Freeman, Inc.
600 Harrison Street
San Francisco, CA 94107-1370

8. Complete Mailing Address of Headquarters or General Business Office of Publisher (Not Printer)
Miller Freeman, Inc.
600 Harrison Street
San Francisco, CA 94107-1370

9. Full Names and Complete Mailing Addresses of Publisher, Editor, and Managing Editor (Do Not Leave Blank)
Publisher (Name and Complete Mailing Address)
Cynthia Blair
Miller Freeman Inc.
600 Harrison St
San Francisco, CA 94107-1370
Editor (Name and Complete Mailing Address)
Alex Dunne
Miller Freeman Inc.
600 Harrison St
San Francisco, CA 94107-1370
Managing Editor (Name and Complete Mailing Address)
Tor Berg
Miller Freeman Inc.
600 Harrison St
San Francisco, CA 94107-1370

10. Owner (If owned by a corporation, its name and address must be stated and also immediately thereafter the names and addresses of stockholders owning or holding 1 percent or more of the total amount of stock. If not owned by a corporation, the names and addresses of the individual owners must be given. If owned by a partnership or other unincorporated firm, its name and address as well as that of each individual must be given. If the publication is published by a nonprofit organization, its name and address must be stated.) (Do Not Leave Blank.)

Full name	Complete Mailing Address
Miller Freeman Inc.	Ludgate House
a wholly owned subsidiary	245 Blackfriars Road
of United News & Media, plc	London SE1 9UY England

11. Known Bondholders, Mortgagees, and Other Security Holders Owning or Holding 1 Percent or More of Total Amount of Bonds, Mortgages, or Other Securities. If none, check here ☐ None

Full name	Complete Mailing Address

12. For completion by nonprofit organizations authorized to mail at special rates. The purpose, function, and nonprofit status of this organization and the exempt status for federal income tax purposes: (Check one)
☐ Has Not Changed During Preceding 12 Mos.
☐ Has Changed During Preceding 12 Mos. (If changed, publisher must submit explanation of change with this statement)

PS Form 3526, October 1994 (See Instructions on Reverse)

13. Publication Name	14. Issue Date for Circulation Data Below
Game Developer	October 1998

15. Extent and Nature of Circulation	Avg. No. Copies Each Issue During Preceding 12 Months	Actual No. Copies of Single Issue Published Nearest to Filing Date
a. Total No. Copies (Net Press Run)	38,695	40,894
b. Paid and/or Requested Circulation (1) Sales Through Dealers and Carriers, Street Vendors, and Counter Sales (Not Mailed)	5,451	5,885
(2) Paid or Requested Mail Subscriptions (Include Advertisers' Proof Copies/Exchange Copies)	24,724	26,157
c. Total Paid and/or Requested Circulation (Sum of 15b(1) and 15b(2))	28,186	32,243
d. Free Distribution by Mail (Samples, Complimentary, and Other Free)	341	352
e. Free Distribution Outside the Mail (Carriers or Other Means)	1,300	0
f. Total Free Distribution (Sum of 15d and 15e)	1,641	352
g. Total Distribution (Sum of 15c and 15f)	29,826	32,595
h. Copies Not Distributed (1) Office Use, Leftovers, Spoiled	1,286	1,100
(2) Return from News Agents	7,184	7,200
i. Total (Sum of 15g, 15h(1), and 15h(2))	38,695	40,895
Percent Paid and/or Requested Circulation (15c / 15g)	94.50%	98.92%

16. This Statement of Ownership will be printed in the December issue of this publication. ☐ Check box if not required to publish.

17. Signature and Title of Editor, Publisher, Business Manager, or Owner — Date: 10-9-98

I certify that all information furnished on this form is true and complete. I understand that anyone who furnishes false or misleading information on this form or who omits material or information requested on the form may be subject to criminal sanctions (including fines and imprisonment) and/or civil sanctions (including multiple damages and civil penalties).

Instructions to Publishers

1. Complete and file one copy of this form with your postmaster on or before October 1, annually. Keep a copy of the completed form for your records.
2. In items 10 and 11, in cases where the stockholder or security holder is a trustee, the name of the person or corporation for whom the trustee is acting. Also include the names and addresses of individuals who are stockholders who own or hold 1 percent or more of the total amount of bonds, mortgages, or other securities of the publishing corporation. In item 11, if none, check box. Use blank sheets if more space is required.
3. Be sure to furnish all information called for in item 15, regarding circulation. Free circulation must be shown in items 15d,e, and f.
4. If the publication had second class authorization as a general or requester publication, this Statement of Ownership, Management, and Circulation must be published; it must be printed in any issue in October or the first printed issue after October. If the publication is not published during October.
5. In item 16, indicate date of the issue in which this Statement of Ownership will be printed.
6. Item 17 must be signed.

Failure to file or publish a statement of ownership may lead to suspension of periodicals class authorization.

PS Form 3526, October 1994 (Reverse)

coding in the game industry, the focus is not directly on the speed of the routine. It doesn't have to be very fast. In fact, when working on tools, it is often better to sacrifice speed for clarity. Often, a tool that you create now and will have a long life and pass through many hands after you. This is not usually true of the actual game code (no matter what your producer may want to think) as most core game routines are rewritten for each project. Tools tend to linger.

The point of showing these types of file loading routines is to demonstrate how easily it can be done. I talk to programmers who say they understand the algorithms but have no models with which to work. They are not comfortable writing a 3D Studio MAX plug-in to get models. I hope that this shows that very commonly used 3D file formats can be easily integrated into your own production tools. Once you build up a library of routines like this, you can easily get models to work within your game applications. If you create a loader for a commonly used format, there are models everywhere that you can use. For actual production applications, I tend to create custom binary formats because they are compact and exactly tuned to the application. But by loading a general format, you can easily make a file converter.

This month, I have provided an application that allows you to load two .OBJ files that have identical vertex arrangements. You then can use the slider to morph between the two. The program can handle objects with and without texture mapping. Grab it at the *Game Developer* web site at http://www.gdmag.com. Special Thanks to Bennie Terry for providing the model, and to Eddie Smith for providing the texture map for the LAG14 character from their real-time action title, ARIES PROJECT. ■

FOR FURTHER INFO

You can find a list of 3D file formats and their specs at:
http://www.cica.indiana.edu/graphics/3D.objects.html

Rule, Keith. *3D Graphics File Formats : A Programmer's Reference.* Addison-Wesley, 1996.

Covers the .OBJ, .3DS, and VRML file formats, among others.