DMorph

Andrew Glassner

From Ovid to Kafka, the idea of metamorphosis has been a powerful literary metaphor. More recently it has become a common but strong visual device.

Image metamorphosis became a sensation in 1992 when Pacific Data Images produced Michael Jackson's video *Black or White*, which showed a variety of very different people seamlessly changing from one into another. In almost no time at all, morphing was everywhere from movies to television.

Image morphing is an artist-driven process. A creative human being looks at the images (or sequences) to be bridged and determines a way to make the changes that will be pleasing to the viewer and harmonious with the piece. To help these artists, I've developed a technique for automatically and smoothly turning one convex 3D shape into another. Like any automatic morph method, this technique should be viewed as a tool and not a self-guided process.

The critical reason for this is that a morph doesn't simply transform two images or shapes. Much more significantly, it says something about those shapes. If I turn a human face into another human face, I'm saying that these faces are essentially similar and that I only need to make small changes to turn one into another. That was the whole message of *Black or White*, and it's why the technique was so perfect for that subject.

Suppose that we want to turn a picture of a giraffe into a picture of a rhinoceros. We'd probably turn the giraffe's body and feet into the rhino's body and feet, which implicitly tells the viewer that these are both quadrupeds. So we're saying something by choosing what goes to what.

Now suppose we're working on a movie where the evil villain keeps a rattlesnake as a pet, and at one point we want to morph from the snake into its owner, a guy holding a gun. We could turn the snake's head into the man's head and grow arms and feet from the rest of the snake's body. But we might want to emphasize the bad guy's nature by turning the snake's rattle into his head. Or we might choose to turn the snake's rattle into the gun and grow the other body parts from the snake's body. These are all pretty cheesy transformations from a storytelling point of view, but each one says something different about the correspondence between the snake and the man and his weapon. There's no way a computer can choose among these for us; it's an artistic decision motivated by what we want the transformation to say.

Figuring change

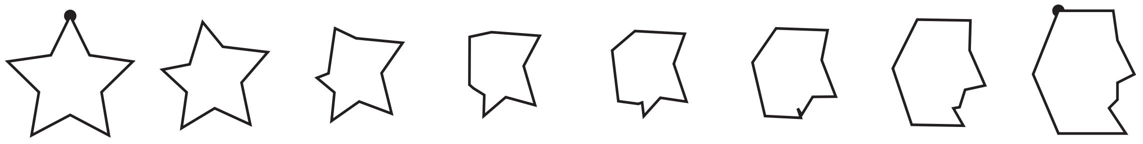
The problems of artistic choice in a transformation are no less important for shapes than they are for images. The essential problem involves *feature matching*—determining which elements of one shape should correspond to elements of the other shape.

We can use automatic feature-matching tools to get a good starting guess, which can later be adjusted by an artist. Alternatively, an artist can use a program to manually correspond features (for example, by telling the computer that the front left foot of a giraffe should turn into the front left foot of a rhino). Then the computer can do its best to make the rest of the shape move in a reasonable while respecting these features.

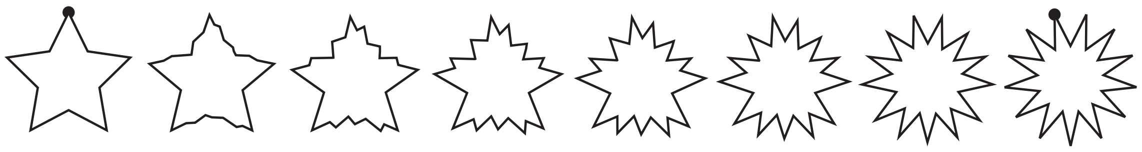
Working in 2D

Let's look at how people have done things in 2D. Perhaps the simplest way to transform two polygonal shapes S_0 and S_1 is to require that they each have the same number of vertices, numbered clockwise, and identify the first vertex v_0 in each shape. Then to morph the shapes, we move v_0 from its position in S_0 to its position in S_1, and repeat this process with all the other vertices. The result is often acceptable as long as the two shapes are pretty simple, as in Figure 1.

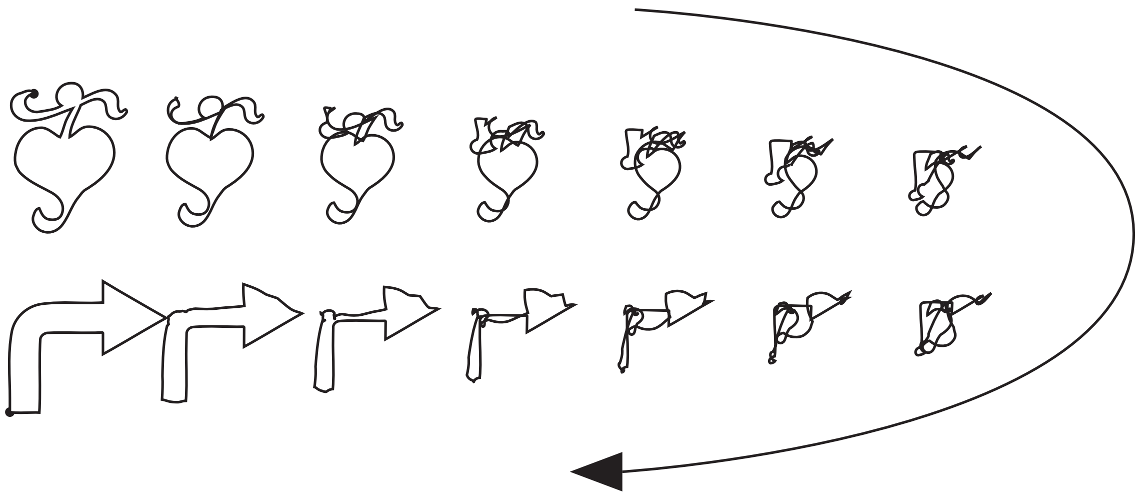
However, requiring both shapes to have the same number of vertices is a tough constraint, often meaning we have to manually add or remove vertices to get the



1 A simple 2D morph between two shapes with the same number of vertices.



2 **A morph where the system inserted new vertices into both shapes so they both had the same number of points.**



3 **Automatic morphs often get tangled up in the middle as vertices pass through each other en route to their destinations. Read the figure clockwise, as indicated by the arrow.**

right number on each, which can be a lot of work for big shapes. An alternative is to compute the least common multiple of the number of vertices in each shape. For example, if S_0 has five vertices and S_1 has 13, then their least common multiple is 65. A system can run around each shape and automatically insert vertices between the existing ones to bring both up to 65. Typically it tries to spread them out as evenly as possible. Figure 2 shows a result of this process for two similar figures and it looks pretty good.

Unfortunately, when the figures are dissimilar, the results are much more disappointing, as Figure 3 shows. The tangled knot that appears at the halfway point is a common result of this technique.

We can look at this problem in several ways, both manually and automatically. For example, Tom Sederberg and colleagues took a sophisticated approach that involved blending the edge lengths and vertex angles of the two shapes being morphed (see the "Further Reading" sidebar for more information).

The automatic morphing problem is tough because algorithmic solutions can't substitute for artistic judgment, and manual solutions are very time-consuming and repetitive.

Working in 3D

The 3D morphing problem is just like the 2D morphing problem, only a lot harder.

Many 3D systems let us easily transform one shape into another as long as they both have the same number of vertices, arranged into the same number of faces. Such packages let animators create "morph targets" that they can then blend together. For example, we might carefully construct a 3D model of someone's face. We could then painstakingly move the vertices of that face around so that the person is smiling happily and save that as the "smile" target. Simliarly, we could make targets for "frown," "grin," "cry," and so on. Then if someone walks into a birthday party and cries with pleasure, we might tell the system to take the original, neutral face and add in 50 percent of the displacement to get to the "happy" face and add another 50 percent of the displacement to get to the crying face. For a real performance the technique is much more subtle, but the basic idea stays the same.

This morph-target approach is useful for lip-synching animated characters. The animator makes targets for the standard mouth positions (such as pursed lips for "o," and tongue between the the teeth for "th"), and then dials in the appropriate target to match the words in the soundtrack.

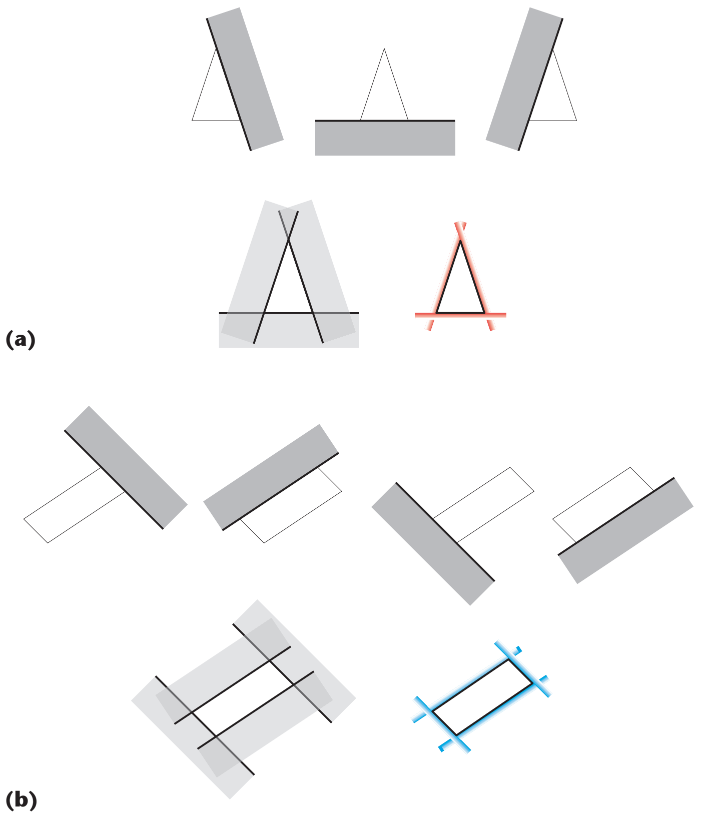
Further Reading

You can read about Tom Sederberg's blending algorithm in "2D Shape Blending: An Intrinsic Solution to the Vertex Path Problem," by T.W. Sederberg et al., *Proc. Siggraph 93*, ACM Press, 1993, pp. 15-18.

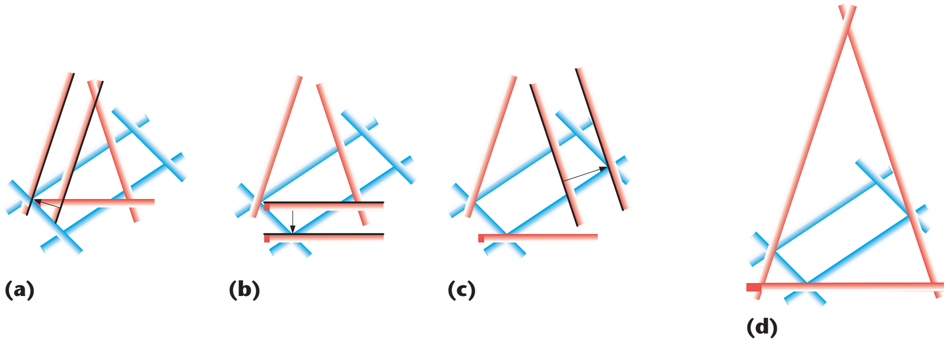
A great VRML library of polyhedra is at the kaleido Web site: http://www.math.technion.ac.il/~rl/kaleido/.

The Geometry Center, as always, provides a great resource at http://www.geom.uiuc.edu/software/weboogl/zoo/polyhedra.wrl.html.

The sculptor George Hart provides a huge number of polyhedra—including many obscure ones—at http://www.georgehart.com/virtual-polyhedra.



4 Defining convex objects as the intersection of planes. (a) A triangle made by three planes. (b) A rectangle made by four planes.



5 (a–c) Moving the triangle's three planes along their normals until they just enclose the rectangle. (d) Final position of the planes.

Morph targets don't work well when the models don't have the same topology (that is, the same numbers of vertices connected the same way into faces). Many systems don't support morphing at all when the topologies are different. Part of the problem is that there's just no clear way of making the shapes match: How would you morph a beach ball into a doughnut? In 2D, we could run around the shape's perimeter and add new vertices as needed. But in 3D, choosing the locations of new vertices isn't well specified.

Even when the models have the same topologies, if the deformation from one morph target to the next is too extreme, the shape can crumple into itself just like the 2D example of Figure 3.

I can think of several ways to fix this problem, and it's fun to dream up heuristics that work in one kind of special case or another. As yet, though, nobody's found an automatic method that's both general and efficient enough to be widely adopted.

DMorph

I wanted to find an automatic technique for morphing polygonal objects that wouldn't demand that the shapes have the same numbers of vertices and faces, nor require any kind of feature matching. Being automatic, it would be appropriate either for things that happen in the background, or as a starting point for manual tweaking.

The method, which I call DMorph, is robust and fast. It has one important limitation, though: it only works for *convex* objects (objects that don't have holes or indentations). One way to test whether an object is convex is to imagine moving around inside it with a piece of string. Pick any two points inside the object and pull the string taut between them. If the string never goes outside the shape for any pair of points, then it's convex. (Otherwise, we say it's nonconvex, or *concave*.) Convex objects can have flat sides, like a cube, or they can be curved, like a sphere. Convex objects include soccer balls, loaves of bread, and dice. Objects like bagels, puppies, and chairs are concave.

Another helpful way to think about convexity is to start with a convex blob (say a big sphere) and slice off a piece of it with a straight edge. As long as that's all you do, your object will still be convex after every slice.

Let's get a feeling for DMorph with a 2D analogy—we'll turn a triangle into a rectangle. Note that these two shapes have different numbers of points and edges. Rather than represent the triangle as three vertices and the three edges that join them, I'll instead represent it as the intersection of three planes. Figure 4a shows the idea. We build three planes perpendicular to the paper's plane. Each one cuts the paper into an *inside* in front of the plane and an *outside* behind it. The triangle is the paper region that's inside all three planes. Figure 4b shows the same thing for our rectangle, which is defined by its own set of four planes. We'll create our morphs by moving these planes around and finding the area inside them.

We begin by taking the triangle's four planes and moving each one until it hugs the rectangle as closely as possible, as in Figure 5. Although in the figure I've drawn the planes with line segments, in theory they go on forever in both directions. So all we need to do is move each plane forward (or backward) along its normal until it's got the rectangle sitting right on its positive side.

The result is in Figure 5. These are the ending positions for the three planes that define the triangle. Their starting positions are where they sit to define the triangle—that is, where they appear in Figure 5a.

Now we do the same thing for the rectangle's planes. This time their positions in Figure 6a are their ending positions. We move each plane forward or backward until it abuts the triangle on its positive side, which marks its starting position. Figure 6e shows the result. Note that this isn't just a scaled version of the rectangle. Because each plane moves individually, the shape formed by the planes in these positions is only reminiscent of the original rectangle.

That's the entire setup. To create the morph, we simply move all seven planes in unison from their starting positions to their ending positions. The shape formed by their intersection is the morph.

Figure 7 shows this in action. The red planes came from the triangle, and the blue ones came from the rectangle. I've marked the starting position of each plane with a pair of dots. You can see that at each step we just move each plane along its normal, and the shape that's formed inside them is the morph at that step.

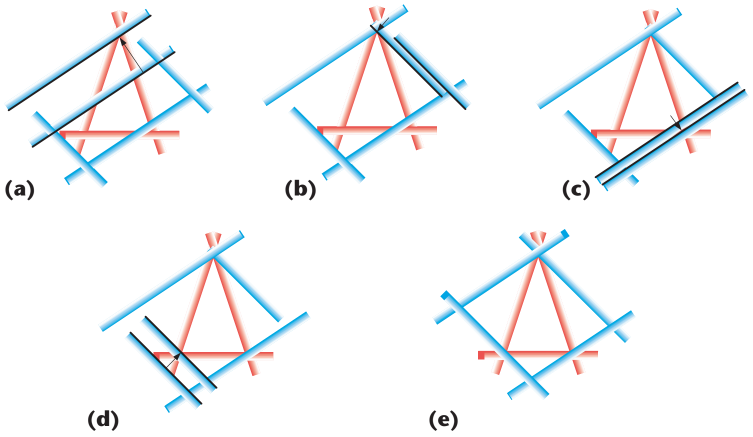
Figure 8 shows the shapes from this process stacked so you can see how they change over time.

That's it! In 3D, it's precisely the same algorithm, except we use 3D planes and 3D vertices. Just treat each face of the starting polyhedron as a plane and move those planes from their starting locations to the spots where they just barely enclose the ending polyhedron. At the same time, move the planes defining the ending polyhedron from their location just outside the starting shape to their resting spots.

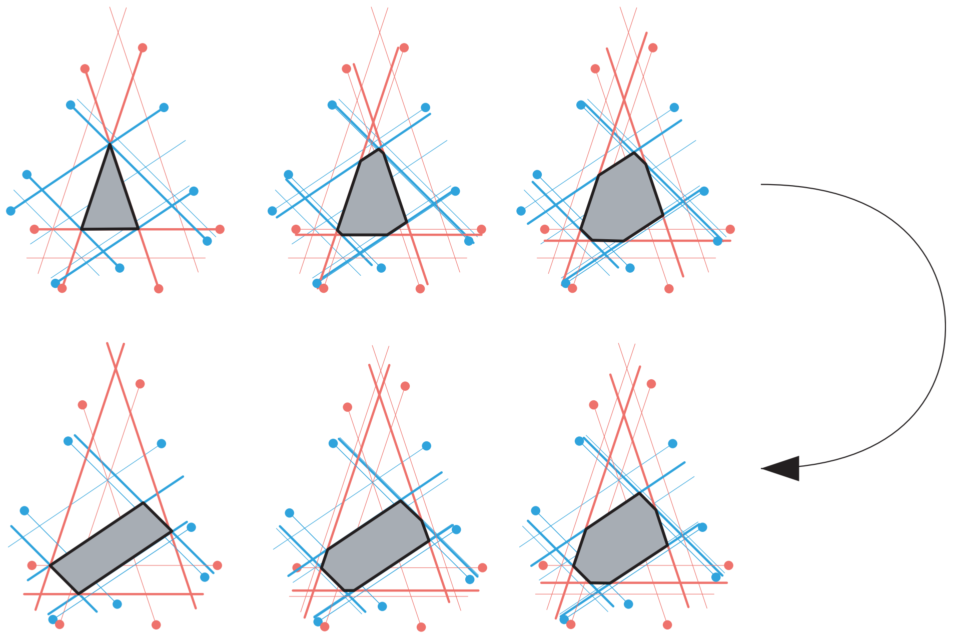
Figures 9 through 14 (next page) show a variety of 3D morphs. To make a chain of more than two transformations, just string together a series of two-object morphs. You can download an animation of these objects morphing together from the *IEEE Computer Graphics and Applications* Web site (see http://csdl.computer.org/comp/mags/cg/2003/05/g5toc.htm).

Note that the objects don't have to be in any specific position relative to each other. If you're morphing a lemon at one end of a table into an orange at the other end, the intermediate shapes will be nice blends of the two shapes and will appear on the table moving from one location to the other.

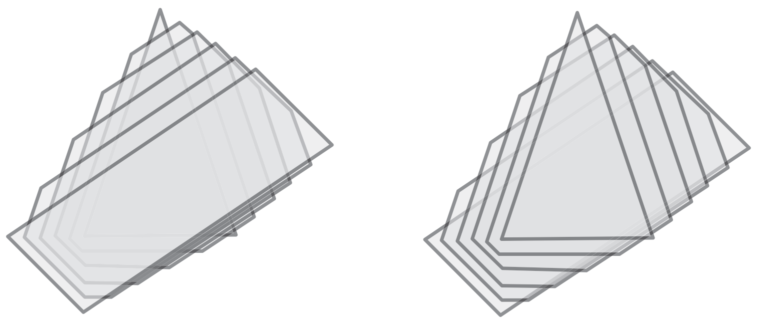
Of course, you could extend this algorithm easily to handle three or more shapes, which you can blend



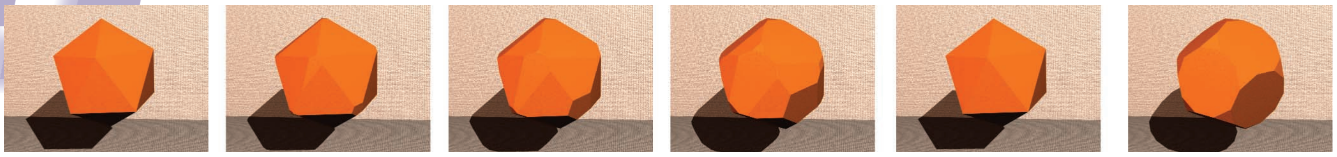
6 (a–c) Moving the four planes of the rectangle along their normals until they just enclose the triangle. (e) Final position of the planes.



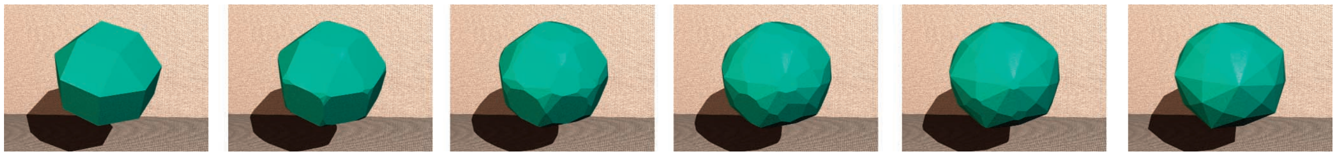
7 During the transformation we move each plane from its starting location to its ending location. The morph is that region of space on the positive side of all the planes.



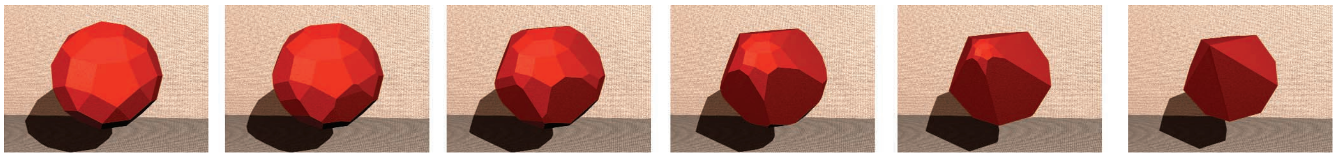
8 Two views of the morphs of Figure 7 stacked on top of each other.



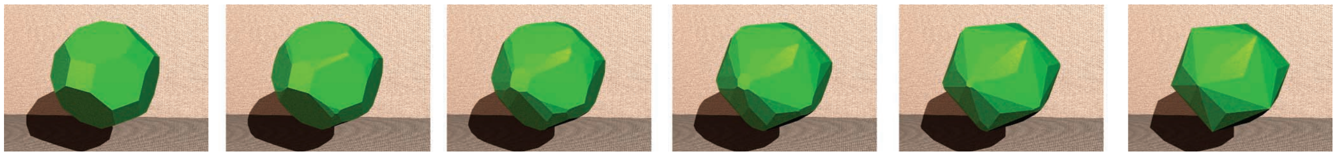
9 Six steps in a 3D morph from an icsoahedron to a truncated dodecahedron.



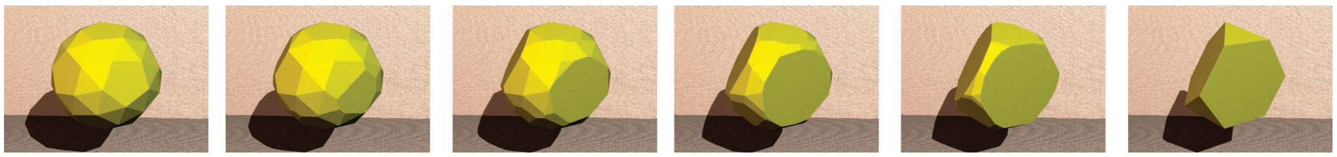
10 Six steps in a 3D morph from a small rhombicuboctahedron to the dual of a great rhombicosidodecahedron.



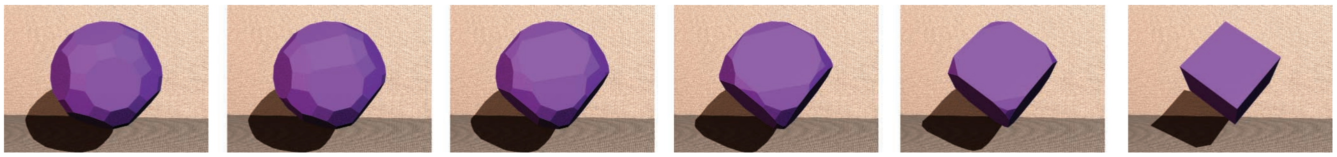
11 Six steps in a 3D morph from a small rhombicosidodecahedron to the dual of a twisted pentagonal prism.



12 Six steps in a 3D morph from a great rhombicuboctahedron to the dual of a truncated dodecahedron.



13 Six steps in a 3D morph from a snub dodecahedron to a truncated tetrahedron.



14 Six steps in a 3D morph from a great rhombicosidodecahedron to a cube.

together to any degree you wish.

The algorithm is nice because it doesn't require anything except the two objects, and they don't need to have any specific properties except that they're both convex.

Programming

I implemented an early version of this idea many years ago in the Cedar programming environment at Xerox PARC. I had to write my own code to create the models, find the planes, clip polygons, fill holes, find volumes of intersection, and more. Some of these steps are tricky, because special cases can wreak havoc on the integrity of your models. This algorithm invites some of those special cases.

For example, suppose you're in the process of constructing a morph. You have a 3D polyhedron and a plane and you want to cut away those parts of the polyhedron that are in the back of the plane. Easy enough, except when one or more vertices is just touching the plane. Then numerical precision becomes important. Just as important is numerical consistency. If the vertex is considered to be just barely in front of the plane for one polygon, it should be evaluated the same way for the next polygon. This becomes an even more critical problem when a polygon lies just about in the plane itself. You must be careful to classify all the vertices the same way and do it consistently.

You can get all of these details right, but it's hard work.

Life is much better when you can get someone else to do the hard work. To that end, the code I wrote for this column is actually in three pieces, each in its own language.

VRML to text via Perl

I wanted to test the algorithm with a bunch of complex but interesting models. Many cool convex polyhedra are available for free on the Web in Virtual Reality Modeling Language (VRML) format. I decided they would serve as my test objects. I chose to make my life easy by using text-only VRML files that contain a single object in point-polygon format.

To process a model, I read in the VRML file, and then search it for the pieces I need using a little script written in Perl. To get the vertices, I look for a list of numbers in square brackets preceded by the keyword *Coordinate*. To find the polygons, I similarly look for a list of numbers in square brackets preceded by the keyword *CoordIndex*. In both cases I simply pull out the numbers in brackets, apply a little reformatting to them, and then print them out to a new text file.

Text to MAXScript via C#

The next stage of the system takes that text file and creates a new text file in the MAXScript language, used by Discreet's 3ds max 5. This program is written in C#. I read in the text files for the two objects I want to morph, and for each one I compute the planes that make up the object.

To find the plane for each polygon, I just take the first three points of the polygon. (If they're colinear, I keep moving the three-point window forward until I find three points that aren't colinear.) Let's call these points v_0, v_1, and v_2. I find the vectors $\mathbf{A} = v_1 - v_0$ and $\mathbf{B} = v_1 - v_2$, and find their cross product $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. I normalize $\mathbf{C}$ by scaling it to a length of 1, giving me the plane normal $\mathbf{N}$. Any point $\mathbf{P}$ on the plane satisfies the plane equation $\mathbf{N} \cdot \mathbf{P} + d = 0$, so I plug in v_0 for $\mathbf{P}$ and solve for d. Together, $\mathbf{N}$ and d tell me how this plane is oriented in space and how far it is from the origin.

Once I have all of the planes for both objects, it's time to find their starting and ending positions. Since the planes don't rotate, all I need to find are the values of d that specify these two extremes; I call them d_0 and d_1. Each plane moves during a morph from d_0 at the start to d_1 at the end. This is why I call the algorithm DMorph.

Let's start with the first object, S_0. Because its planes begin where they were computed, I set $d_0 = d$ for each plane. To start the process of finding d_1 for this plane, I grab the first vertex of S_1 and move the plane so it includes that vertex. Then I test every remaining vertex S_1 one by one, and if it's on the negative side of the plane, I move the plane by computing a new value of d so it includes the vertex. When I'm done, I save the final value of d for that plane as d_1. I repeat this process for all the planes in S_0.

The planes for S_1 are handled the same way, but in reverse. I move each plane when needed so all the vertices of S_0 are on its positive side, and I save its original value in d_1 and the computed value in d_0.

Next I merge the two lists of planes together. From now on, the fact that they originally belonged to two different objects is lost. All I need now is this one combined list of planes.

Now I open up a new text file and start writing MAXScript into it. Let's suppose that I've told the program to create an animation that takes n steps to go from the start shape to the ending shape. For each frame f from $[0, n)$ the process is the same.

I first create a large box, centered at the origin (I call this box *the marble*, like the slab of marble a sculptor starts with before he starts cutting away at it). Then I compute $\alpha = f/(n - 1)$, which gives me a floating-point number from 0 to 1 telling me where I am in the animation. For each plane in the combined list, I compute the value of d at this time as $d_f = d_0 + \alpha\,(d_1 - d_0)$. To cut the marble using this plane, I use a technique provided by 3ds max called a *slice modifier*. This is a plane that you can set up to cut the model and remove everything behind it.

I create a new slice modifier for each plane. Using the plane's normal and this value of d_f, I build up matrices to rotate and position the slicer so it's sitting where the plane is located. I then tell 3ds max to cap any holes produced by the slicing (using a *cap_holes* modifier), and then I collapse the modifier stack so that I'm back to just a single, simple object. Then I cut it again with the next plane, and so on again and again until I've processed every plane. The last step is to tell the program that this cut-up box is visible at frame f, but invisible at all frames before and after.

Now I deselect the box, increment the frame counter f, and start over again with a new piece of marble.

MAXScript to image

This is the easy part. I just open up 3ds max and tell MAXScript to evaluate the text file produced by the previous step. After a little chugging away, the result is an animation of the first shape turning into the other, ready for rendering.

Because I can rely on 3ds max to handle all the delicate work of numerical accuracy and stability in the clipping phase, I save myself a ton of hard work. This is a really satisfying way to do geometry!

Wrapping up

Like any algorithm, DMorph has pros and cons. On the pro side, it's satisfyingly simple and robust and easy to program. It doesn't place any constraints on the two objects being interpolated except that they're convex. One object can be a 200-sided cone where 200 polygons all share a single vertex, and the other object a baseball, and the algorithm doesn't care. There's no feature matching, automatic or otherwise. The transformations are smooth, and the blending looks nice.

On the con side, the program is limited to convex objects. If you look around, you'll probably see few convex objects in the environment around you. Umbrellas, trees, and paper clips are all concave. So are pianos, people, and giraffes. DMorph doesn't help us with these objects, much less any really wacky transformations that we might want to do, such as turning a paper clip into the Golden Gate Bridge.

That's okay, though. It just means we have another interesting problem to think about. ■

Readers may contact Andrew Glassner at andrew@glassner.com.