

“Modeling By Example”, SIGGRAPH 2004

Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, David Dobkin (mostly @ Princeton)

Summary by Dan Morris

- This paper addresses the general problem that 3d modeling programs are difficult to use for experts, impossible for novices. They present three clever algorithms that translate into useful user interface features that make modeling easier.

FYI, this paper is awesome.

- They also assume that you’re likely to be doing modeling that goes something like this:
 - Cut a piece away from a model (e.g take the arm away from the Venus de Milo)
 - Build simple primitives that *roughly* represent the piece you want to find in a database (they take this step as given)
 - Search a database of models that might contain your part
 - Splice one of the hits back in
- The first thing a user might want to do is cut a piece away from a model. However, selecting the individual edges you want to cut is a pain in the ass. Trust me. So they provide an algorithm for translating a simple screen-space gesture into the set of edges that the user was ‘probably’ trying to indicate.
- To do this, they let the user make a “stroke” (like painting a line) with a user-specified width. They then seek to find the “best” set of edges to cut at, knowing that the cut *must* pass within [width] pixels of every point on the stroke.
 - To do this, they formulate a ‘cost’ function for edges in the mesh. As you might expect, they’re going to use something like Dijkstra’s algorithm to find the lowest-cost path. Isn’t it amazing when computer science has actual computer science in it?
 - Their cost function looks like this:

$$cost(e) = c_{len}(e) \times c_{ang}(e)^\alpha \times c_{dist}(e)^\beta \times c_{vis}(e)^\gamma \times c_{dot}(e)^\delta$$

- The terms represent (for each edge) :
 - Edge length (find the shortest possible cut)
 - Dihedral angle (users probably want to cut at ridges or valleys in the mesh)
 - Distance from the user-specified stroke
 - Visibility... they specifically bias their algorithm toward selecting edges that weren’t visible when the user made his stroke, because you probably did a much better job hitting exactly the region you meant to cut on the side of the mesh that you could actually see, so the cut would never get “around the back” without a little help.
 - The orientation of the edge’s surface normal relative to the user’s viewpoint... again, this helps bias toward non-visible edges
- And indeed these all seem like clever things to use to help pick your edges...
- I’ll skip the part where they talk about actually finding the minimal-cost set of edges; trust that it resembles Dijkstra’s algorithm and does what it was supposed to do.

- They then provide an interface for making additional “strokes” to refine the original set of edges. If I didn’t like a particular set of edges, I make a new stroke near it. Then it finds the closest vertex on the original cut to each end of my new stroke, and says “I need to find a new path to connect these vertices”, and does the same process, using the new stroke as a constraint.
- Another thing the user might then want to do is search a database of parts for a particular shape. They present a clever algorithm for this too.
- First of all they augment their search by incorporating simple text annotation search. I.e., if I’m looking for arms and some of the objects in my database are labeled ‘arm’, life doesn’t need to be difficult. But the interesting stuff is the mesh-matching...
- Basically any way you look at it I’m probably going to have to loop over my whole database, matching my query mesh (the thing that looks *sort of* like an arm) against each object, and seeing how it matches.
 - The first thing they do is scale each object in the database – and the test object – to fit into a uniform bounding box, to get around the scaling problem (i.e. it’s dumb if I miss an arm in my database because it was ‘smaller’ than my test arm). Not a great solution; they admit this. But a reasonable approach.
 - The second thing they do is rotate each object to align its primary, secondary, and tertiary axes (basically the longest, second-longest, and third-longest axes of an oriented bounding box around the object) with a reference frame. This is a way to solve the rotation problem. It does create some confusion... if I align my object’s longest axis with, say, the z-axis... *which way* does it go on the z-axis? I have the same problem with the other two axes, so effectively I have 8 possible orientations that are all ‘axis-aligned’. They solve this by basically doing every search in 8 different orientations (they call it ‘the efficient axial search method’). Hey, it works.
 - Now for each object they compute two different voxel grids... one is a grid that is 0 everywhere except on the object surface, where it’s 1. The other is the ‘distance transform’ of the object, which is the distance to the nearest mesh point at each voxel.
 - These voxel grids for the scaled, oriented object are the *shape descriptor*, the key feature vector that I’m going to be matching when I do a query. This is what I store for each object in my database, and this is what I build for my test object before I query the database.
 - FYI, they use 64x64x64 grids, which are pretty damn low-res.
 - Also, FYI, they use our favorite numeric technique... SVD... to reduce the effective size of their database. It doesn’t really affect the algorithms much, so that’s all we’ll say about it. Effectively they zip and unzip shape descriptors as they go into and out of the database.
 - Now to do a test for similarity between two shape descriptors, I just take two dot-products... I dot the distance transform of one mesh with the surface rasterization of the other, and vice versa. This basically tells me ‘how far is each point on the surface of one mesh from the surface of the other?’ I add the two dot products, and I’m good to go.
 - Note that their approach to searching only *part* of a shape (e.g. searching for arms in ‘body-like’ meshes) is to apply a weighting term in the dot-product, to zero out all voxels that are far away from the region I care about.
 - This is not a very general way of solving this problem... if I give it a model of an arm, it doesn’t really find ‘arms’, it finds ‘arms that were in exactly the same place in their

parent model as my arm is'. A major limitation. To be addressed in SIGGRAPH 2005, maybe.

- Just to reiterate... given a query mesh, I do this similarity test for all meshes in my database and return the highest n hits to the user.
- Now the last major tool they provide the user with is an algorithm to help you *attach* objects to your mesh. So in their example, maybe I used their tool to cut a cat's head off, searched the database with the removed head to find a new head, and now I want to splice the new head back on.
 - This is really tough, since the edges won't line up right, and it might not be scaled or oriented quite right, etc.
 - The first thing they do it try to line the new head up with the old one (so this part only applies if there *is* a known part that you're replacing, otherwise the user is on his own for this step).
 - To get the optimal translation to line the new mesh up with the old mesh, they simply align their centers of mass.
 - To get the optimal scaling, they simple scale the new mesh so that the variance of the distances from each vertex to the center of mass matches that of the old mesh.
 - To solve for the optimal rotation, they pull out each object's shape descriptor and do a magic algorithm that they cite from someone else's paper but don't explain in much detail. That's enough for me right now.
 - Now I have the new head in just about the right place... how to make the jagged edges merge nicely with the jagged edges left from removing the old head?
 - Basically, they find all the vertices on the 'edges' of the two objects; that is, the cut edges that need re-attaching.
 - Among those vertices, they find the closest pair (one vertex on each object).
 - From here, they walk around the 'cut edge' on each object, performing a greedy algorithm that iteratively walks along one of the edges at each iteration. This basically gives them an approximate correspondence between points on one cut edge and points on the other.
 - Given that, it's easy to triangulate the 'gaps' between the two objects.

This was a fun paper. It has some very clever algorithms without having intense math, and it's definitely all the way at the HCI end of SIGGRAPH. Good times.