Alex (Yu) Li
CS284: Professor Séquin
December 11, 2009

Face Morphing

Introduction

Face morphing, a specific case of geometry morphing, is a powerful tool for animation and graphics. It consists of the fluid transformation of one 3D face mesh to another. Its application spans a vast range of fields, anywhere from character animation for films and advertising to facial surgery planning. Although the subject matter has become popular in the past few years due to an increase in the relevance of CGI, there still remain many problems to be solved.

Some of main problems that deal with face morphing are feature specification, dense point-to-point correlation, and transition control. Feature specification is the method in which a person chooses the correspondence between pairs of feature primitives. In many morphing algorithms, primitives such as line segments or points are used to determine feature positions in the images[1]. Then the feature correspondence can be used to compute mapping functions, or dense point-to-point alignments, that are used to interpolate the positions of the vertices across the morph sequence. Transition control is what determines the rate of the morph across a transformation sequence[1].

For my project I will focus on the problem of point-to-point correlation. I propose creating a program that allows the smooth morphing between two meshes.

Related Work

Since the field of face and image morphing has become popular in the past few years, many researchers have attempted to improve upon its problems. Zanella and Fuentes in the paper *An Approach to Automatic Morphing of Face Images in Frontal View* decided to tackle the problem of automated feature specification. They created a model of 73 points based on a simple parameterized face model, and used the information about the geometrical relationship among the elements of the face to perform automatic face morphing in 2D images.

Furthermore, Hu, Zhou, and Wu in *A Dense Point-to-Point Alignment Method for Realistic 3D Face Morphing and Animation* decided to create a new point matching method that uses the thin plate spline transformation to model the warping of different faces. The basic idea of their implementation is to deform the initial starting mesh with a thin plate spine algorithm, then perform a search for each point on the deformed initial surface to find its closest neighbor in the destination surface. If there are collision points, or multiple points that map to a single point in the destination surface, then those are detected and stored. For each collision point, the nearest neighbor is then found reversely, from the destination surface to the initial given surface, and the point with the minimum distance is stored as a corresponding pair[2]. In this way the algorithm can guarantee a one to one correspondence without collision. I adopted most of the techniques from Hu, Zhou, and Wu's implementation for my project. However I did not implement the thin plate spline transformation of the initial source surface.

Proposal

I first proposed to solve the face morphing problem through displacement mapping. Building upon the Genus4 code, I would implement a displacement map. By overlaying a face map over a fine mesh plane and perturbing the local surface normals based on the map, I could render a human face. Furthermore by inputting another face map into the program, I could then interpolate between the two face maps and create a morphing of meshes. However as I began implementing the displacement map, I ran into many problems.

Difficulties

One of the major problems with displacement maps is that there is a lack of suitable human face maps online. All the maps I found were detailed, but did not have the basic face geometry. Most face maps assume that the user has already modeled the underlying geometry of the face and the map is used only to fill in precise detail. As a result, when I inputted them into a displacement map simulator I found online, they looked terrible and very flat, not at all how a human face should look. Because of the difficulty of finding good input files and the fact that mapping the files into a fine mesh plane would produce poor results I had to rethink my first proposed approach.

My second idea was instead of using displacement maps to interpolate meshes; I could directly move the vertices of the meshes by calculating the nearest points. However, this implementation also has its problems. For example, how can I ensure that the vertices from one mesh have a one-to-one correspondence to the vertices in the second mesh. At first I simply implemented the interactive closest point (ICP) approach where I associated points by the nearest neighbor criteria, then estimated transformation parameters using a mean square cost function, and transformed the points using the estimated parameters[3]. However this approach did not result in a very smooth mesh interpolation.

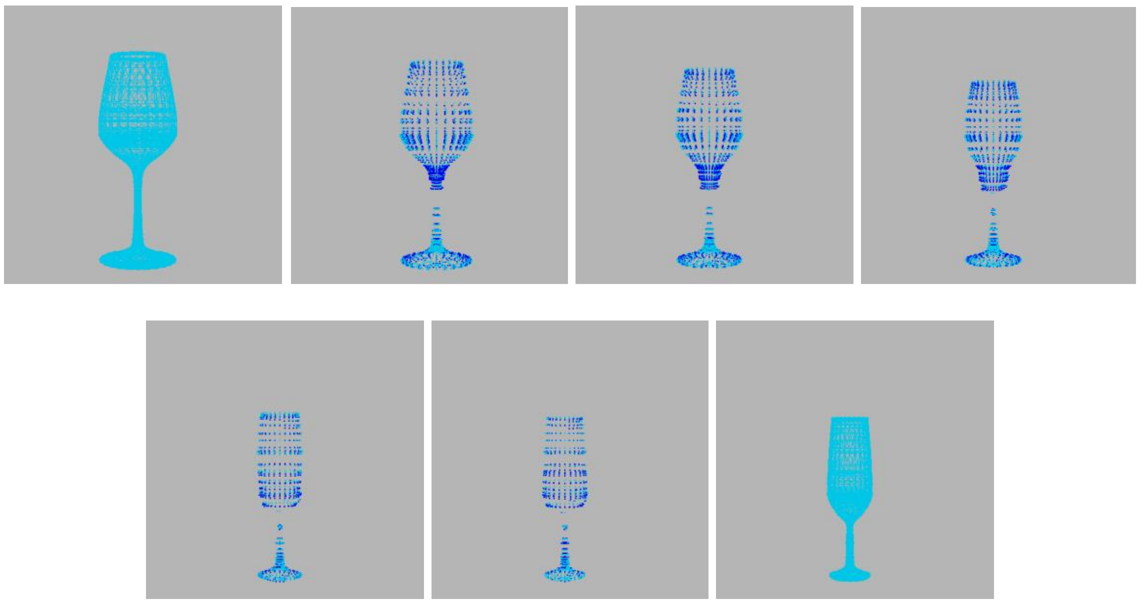
Solution

The implementation of my face morpher program resembles the methods used in Hu, Zhou, and Wu's paper. First I wrote a .obj parser that could read the input object files. However, since .obj files can contain a variety of geometric shapes, I converted all geometry into triangles. By doing this I could guarantee that as the program drew the interpolation geometry to openGL it would look smooth and there would be no strange conversions from quads to triangles (since it has already been pre-computed). Then I wrote an interactive closest point algorithm, as described above, which iteratively loops though and finds the nearest vertex from the starting mesh to the destination mesh (calculated by the mean square cost function). In order to solve the collision points problem, I was inspired by Hu, Zhou, and Wu's paper. To eliminate these collision points, I revised the point matching algorithm so that a reverse distance list for every collision point is constructed, and only the vertex point with the minimum distance is considered the matching one to one correspondence point. The final step to my implementation is the interpolation of the starting vertex point to the now calculated corresponding end point. I basically did this by subtracting the starting vertex from the ending vertex and dividing by the step size in order to find the distance the vertex needs to move at every frame. Now all that is left is to draw the results into openGL.

Overview of Implementation:

- Write .obj parser that converts all geometry into triangles
- Write a ICP algorithm that will:
  Loop though all starting vertices to find the closest destination vertex
  Detect collision points and store them into a 2D vector
- Create a reverse distance list for every collision point detected and find the corresponding minimum distance vertex
- Interpolate between the start and end vertices though a given step size
- Draw to openGL

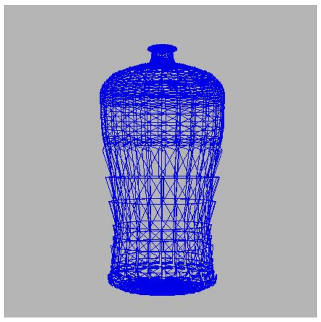
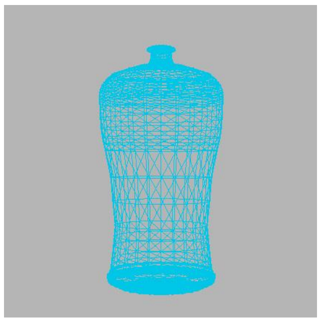
Results



Goblet Mesh Interpolation (goblet1 and goblet3)

The above are the results of my face morphing program. The first panel is the beginning mesh, and the last panel is the destination mesh, while the panels in between display a smooth interpolation of the two given meshes. My program works well for meshes that have the same vertex count, because this will guarantee a one-to-one correspondence of vertex points.

However my algorithm for the morpher program can still be improved. As seen in the pictures below, on the left is the resulting vase mesh after morphing and what it should look like is on the right. The picture on the left has some vertices sticking out and does not perfectly resemble the objective mesh yet.

Left: Resulting Vase Mesh After Morph   Right: Destination Vase Mesh

Manual

The FaceMorpher program is written in C++. The best way to use the program is to open *facemorpher.sln* in Microsoft Visual Studio version 2008. Much of how to run the program is done with keyboard strokes.

- In order to input two .obj files into the command line, the user must type *file* then the source mesh followed by a space and then the destination mesh:

```
file data/vase1.obj data/vase2.obj
```

- To start the program, simply press *f5* or go to Debug/Start Debugging.
- If the program is working, a black screen will appear saying *calculating*…
- After the program has finished calculating the point-to-point correlation, it will output a openGL window with the starting mesh.
- In order to move/rotate the mesh, see the key strokes section.
- If the user wants to start the morph, press the *m* key.

<u>Key strokes:</u>

• '+'	Zoom in
• '- '	Zoom out
• 'm'	Begin morph
• Left arrow	Rotate left
• Right arrow	Rotate right
• Up arrow	Rotate up
• Down arrow	Rotate down
• Shift + left arrow	Translate left
• Shift + right arrow	Translate right
• Shift + up arrow	Translate up
• Shift + down arrow	Translate down
• 'w'	Wireframe
• 's'	Smooth
• Space bar	Quit

References

1. http://www.cs.utep.edu/ofuentes/zanella.pdf
2. http://www.hindawi.com/journals/ijcgt/2009/609350.html
3. http://en.wikipedia.org/wiki/Iterative_Closest_Point