Making Faces

Brian Guenter[†] Cindy Grimm[†] Daniel Wood[‡]
Henrique Malvar[†] Fredrick Pighin[‡]
[†]Microsoft Corporation [‡]University of Washington

ABSTRACT

We have created a system for capturing both the three-dimensional geometry and color and shading information for human facial expressions. We use this data to reconstruct photorealistic, 3D animations of the captured expressions. The system uses a large set of sampling points on the face to accurately track the three dimensional deformations of the face. Simultaneously with the tracking of the geometric data, we capture multiple high resolution, registered video images of the face. These images are used to create a texture map sequence for a three dimensional polygonal face model which can then be rendered on standard 3D graphics hardware. The resulting facial animation is surprisingly life-like and looks very much like the original live performance. Separating the capture of the geometry from the texture images eliminates much of the variance in the image data due to motion, which increases compression ratios. Although the primary emphasis of our work is not compression we have investigated the use of a novel method to compress the geometric data based on principal components analysis. The texture sequence is compressed using an MPEG4 video codec. Animations reconstructed from 512x512 pixel textures look good at data rates as low as 240 Kbits per second.

CR Categories: I.3.7 [Computer Graphics]: Three Dimensional Graphics and Realism: Animation; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1 Introduction

One of the most elusive goals in computer animation has been the realistic animation of the human face. Possessed of many degrees of freedom and capable of deforming in many ways the face has been difficult to simulate accurately enough to convince the average person that a piece of computer animation is actually an image of a real person.

We have created a system for capturing human facial expression and replaying it as a highly realistic 3D "talking head" consisting of a deformable 3D polygonal face model with a changing texture map. The process begins with video of a live actor's face, recorded from multiple camera positions simultaneously. Fluorescent colored 1/8" circular paper fiducials are glued on the actor's face and their 3D position reconstructed over time as the actor talks and emotes. The 3D fiducial positions are used to distort a 3D polygonal face model in mimicry of the distortions of the real face. The fiducials are removed using image processing techniques and the video streams from the multiple cameras are merged into a single texture map. When the resulting fiducial-free texture map is applied to the 3D reconstructed face mesh the result is a remarkably life-like 3D animation of facial expression. Both the time varying texture created from the video streams and the accurate reproduction of the 3D face structure contribute to the believability of the resulting animation.

Our system differs from much previous work in facial animation, such as that of Lee [10], Waters [14], and Cassel [3], in that we are not synthesizing animations using a physical or procedural model of the face. Instead, we capture facial movements in three dimensions and then replay them. The systems of [10], [14] are designed to make it relatively easy to animate facial expression manually. The system of [3] is designed to automatically create a dialog rather than faithfully reconstruct a particular person's facial expression. The work of Williams [15] is most similar to ours except that he used a single static texture image of a real person's face and tracked points only in 2D. The work of Bregler et al [2] is somewhat less related. They use speech recognition to locate visemes[1] in a video of a person talking and then synthesize new video, based on the original video sequence, for the mouth and jaw region of the face to correspond with synthetic utterances. They do not create a three dimensional face model nor do they vary the expression on the remainder of the face. Since we are only concerned with capturing and reconstructing facial performances out work is unlike that of [5] which attempts to recognize expressions or that of [4] which can track only a limited set of facial expressions.

An obvious application of this new method is the creation of believable virtual characters for movies and television. Another application is the construction of a flexible type of video compression. Facial expression can be captured in a studio, delivered via CDROM or the internet to a user, and then reconstructed in real time on a user's computer in a virtual 3D environment. The user can select any

[1] Visemes are the visual analog of phonemes.

Figure 1: The six camera views of our actress' face.

arbitrary position for the face, any virtual camera viewpoint, and render the result at any size.

One might think the second application would be difficult to achieve because of the huge amount of video data required for the time varying texture map. However, since our system generates accurate 3D deformation information, the texture image data is precisely registered from frame to frame. This reduces most of the variation in image intensity due to geometric motion, leaving primarily shading and self shadowing effects. These effects tend to be of low spatial frequency and can be compressed very efficiently. The compressed animation looks good at data rates of 240 kbits per second for texture image sizes of 512x512 pixels, updating at 30 frames per second.

The main contributions of the paper are a method for robustly capturing both a 3D deformation model and a registered texture image sequence from video data. The resulting geometric and texture data can be compressed, with little loss of fidelity, so that storage requirements are reasonable for many applications.

Section 2 of the paper explains the data capture stage of the process. Section 3 describes the fiducial correspondence algorithm. In Section 4 we discuss capturing and moving the mesh. Sections 5 and 6 describe the process for making the texture maps. Section 7 of the paper describes the algorithm for compressing the geometric data.

2 Data Capture

We used six studio quality video cameras arranged in the pattern shown in Plate 1 to capture the video data. The cameras were synchronized and the data saved digitally. Each of the six cameras was individually calibrated to determine its intrinsic and extrinsic parameters and to correct for lens distortion. The details of the calibration process are not germane to this paper but the interested reader can find a good overview of the topic in [6] as well as an extensive bibliography.

We glued 182 dots of six different colors onto the actress' face. The dots were arranged so that dots of the same color were as far apart as possible from each other and followed the contours of the face. This made the task of determining frame to frame dot correspondence (described in Section 3.3) much easier. The dot pattern was chosen to follow the contours of the face (i.e., outlining the eyes, lips, and nasio-labial furrows), although the manual application of the dots made it difficult to follow the pattern exactly.

The actress' head was kept relatively immobile using a padded foam box; this reduced rigid body motions and ensured that the actress' face stayed centered in the video images. Note that rigid body motions can be captured later using a 3D motion tracker, if desired.

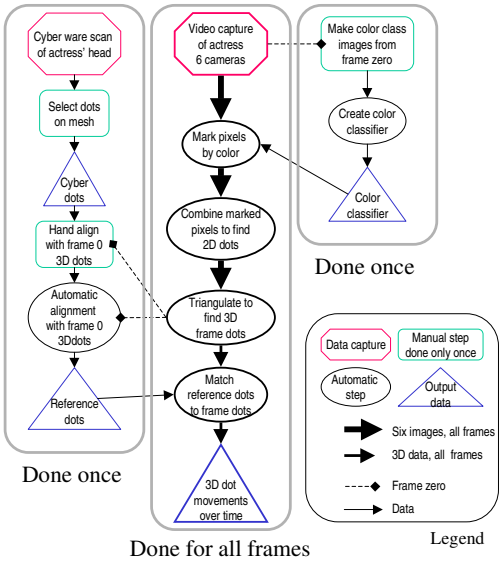

Figure 2: The sequence of operations needed to produce the labeled 3D dot movements over time.

The actress was illuminated with a combination of visible and near UV light. Because the dots were painted with fluorescent pigments the UV illumination increased the brightness of the dots significantly and moved them further away in color space from the colors of the face than they would ordinarily be. This made them easier to track reliably. Before the video shoot the actress' face was digitized using a cyberware scanner. This scan was used to create the base 3D face mesh which was then distorted using the positions of the tracked dots.

3 Dot Labeling

The fiducials are used to generate a set of 3D points which act as control points to warp the cyberware scan mesh of the actress' head. They are also used to establish a stable mapping for the textures generated from each of the six camera views. This requires that each dot have a unique and consistent label over time so that it is associated with a consistent set of mesh vertices.

The dot labeling begins by first locating (for each camera view) connected components of pixels which correspond to the fiducials. The 2D location for each dot is computed by finding the two dimensional centroid of each connected component. Correspondence between 2D dots in different camera views is established and potential 3D locations of dots reconstructed by triangulation. We construct a reference set of dots and pair up this reference set with the 3D locations in each frame. This gives a unique labeling for the dots that is maintained throughout the video sequence.

A flowchart of the dot labeling process is shown in Figure 2. The left side of the flowchart is described in Section 3.3.1, the middle in Sections 3.1, 3.2, and 3.3.2, and the right side in Section 3.1.1.

3.1 Two-dimensional dot location

For each camera view the 2D coordinates of the centroid of each colored fiducial must be computed. There are three

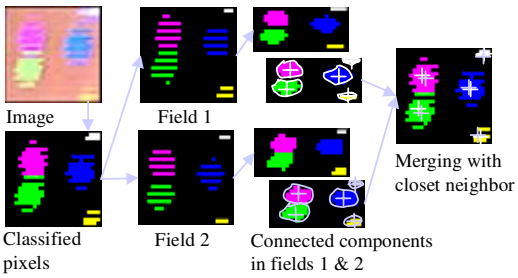

Figure 3: Finding the 2D dots in the images.

steps to this process: color classification, connected color component generation, and centroid computation.

First, each pixel is classified as belonging to one of the six dot colors or to the background. Then depth first search is used to locate connected blobs of similarly colored pixels. Each connected colored blob is grown by one pixel to create a mask used to mark those pixels to be included in the centroid computation. This process is illustrated in Figure 3.

The classifier requires the manual marking of the fiducials for one frame for each of the six cameras. From this data a robust color classifier is created (exact details are discussed in Section 3.1.1). Although the training set was created using a single frame of a 3330 frame sequence, the fiducial colors are reliably labeled throughout the sequence. False positives are quite rare, with one major exception, and are almost always isolated pixels or two pixel clusters. The majority of exceptions arise because the highlights on the teeth and mouth match the color of the white fiducial training set. Fortunately, the incorrect white fiducial labelings occur at consistent 3D locations and are easily eliminated in the 3D dot processing stage.

The classifier generalizes well so that even fairly dramatic changes in fiducial color over time do not result in incorrect classification. For example, Figure 5(b) shows the same green fiducial in two different frames. This fiducial is correctly classified as green in both frames.

The next step, finding connected color components, is complicated by the fact that the video is interlaced. There is significant field to field movement, especially around the lips and jaw, sometimes great enough so that there is no spatial overlap at all between the pixels of a fiducial in one field and the pixels of the same fiducial in the next field. If the two fields are treated as a single frame then a single fiducial can be fragmented, sometimes into many pieces.

One could just find connected color components in each field and use these to compute the 2D dot locations. Unfortunately, this does not work well because the fiducials often deform and are sometimes partially occluded. Therefore, the threshold for the number of pixels needed to classify a group of pixels as a fiducial has to be set very low. In our implementation any connected component which has more than three pixels is classified as a fiducial rather than noise. If just the connected pixels in a single field are counted then the threshold would have to be reduced to one pixel. This would cause many false fiducial classifications because there are typically a few 1 pixel false color classifications per frame and 2 or 3 pixel false clusters occur occasionally. Instead, we find connected components and generate lists of potential 2D dots in each field. Each potential 2D dot in field one is then paired with the closest 2D potential dot in field two. Because fiducials of the same color are spaced far apart, and because the field to field movement is not very large, the closest potential 2D dot is virtually guaranteed to be the correct match. If the sum of the pixels in the two potential 2D dots is greater than three pixels then the connected components of the two 2D potential dots are merged, and the resulting connected component is marked as a 2D dot.

The next step is to find the centroid of the connected components marked as 2D dots in the previous step. A two dimensional gradient magnitude image is computed by passing a one dimensional first derivative of Gaussian along the x and y directions and then taking the magnitude of these two values at each pixel. The centroid of the colored blob is computed by taking a weighted sum of positions of the pixel (x, y) coordinates which lie inside the gradient mask, where the weights are equal to the gradient magnitude.

3.1.1 Training the color classifier

We create one color classifier for each of the camera views, since the lighting can vary greatly between cameras. In the following discussion we build the classifier for a single camera.

The data for the color classifier is created by manually marking the pixels of frame zero that belong to a particular fiducial color. This is repeated for each of the six colors. The marked data is stored as 6 *color class images*, each of which is created from the original camera image by setting all of the pixels *not* marked as the given color to black (we use black as an out-of-class label because pure black never occurred in any of our images). A typical color class image for the yellow dots is shown in Figure 4. We generated the color class images using the "magic wand" tool available in many image editing programs.

A seventh color class image is automatically created for the background color (e.g., skin and hair) by labeling as out-of-class any pixel in the image which was previously marked as a fiducial in any of the fiducial color class images. This produces an image of the face with black holes where the fiducials were.

The color classifier is a discrete approximation to a nearest neighbor classifier [12]. In a nearest neighbor classifier the item to be classified is given the label of the closest item in the training set, which in our case is the color data contained in the color class images. Because we have 3 dimensional data we can approximate the nearest neighbor classifier by subdividing the RGB cube uniformly into voxels, and assigning class labels to each RGB voxel. To classify a new color you quantize its RGB values and then index into the cube to extract the label.

To create the color classifier we use the color class images to assign color classes to each voxel. Assume that the color class image for color class C_i has n distinct colors, $c_1 ... c_n$. Each of the voxels corresponding to the color c_j is labeled with the color class C_i. Once the voxels for all of the known colors are labeled, the remaining unlabeled voxels are assigned labels by searching through all of the colors in each color class C_i and finding the color closest to p in RGB space. The color p is given the label of the color class containing the nearest color. Nearness in our case is the Euclidean distance between the two points in RGB space.

If colors from different color classes map to the same sub-cube, we label that sub-cube with the background label since it is more important to avoid incorrect dot labeling than it is to try to label every dot pixel. For the results shown in this paper we quantized the RGB color cube into a 32x32x32 lattice.

Figure 4: An image of the actress's face. A typical training set for the yellow dots, selected from the image on the left.

3.2 Camera to camera dot correspondence and 3D reconstruction

In order to capture good images of both the front and the sides of the face the cameras were spaced far apart. Because there are such extreme changes in perspective between the different camera views, the projected images of the colored fiducials are very different. Figure 5 shows some examples of the changes in fiducial shape and color between camera views. Establishing fiducial correspondence between camera views by using image matching techniques such as optical flow or template matching would be difficult and likely to generate incorrect matches. In addition, most of the camera views will only see a fraction of the fiducials so the correspondence has to be robust enough to cope with occlusion of fiducials in some of the camera views. With the large number of fiducials we have placed on the face false matches are also quite likely and these must be detected and removed. We used ray tracing in combination with a RANSAC [7] like algorithm to establish fiducial correspondence and to compute accurate 3D dot positions. This algorithm is robust to occlusion and to false matches as well.

First, all potential point correspondences between cameras are generated. If there are k cameras, and n 2D dots in each camera view then $\binom{k}{2} n^2$ point correspondences will be tested. Each correspondence gives rise to a 3D candidate point defined as the closest point of intersection of rays cast from the 2D dots in the two camera views. The 3D candidate point is projected into each of the two camera views used to generate it. If the projection is further than a user-defined epsilon, in our case two pixels, from the centroid of either 2D point then the point is discarded as a potential 3D point candidate. All the 3D candidate points which remain are added to the 3D point list.

Each of the points in the 3D point list is projected into a reference camera view which is the camera with the best view of all the fiducials on the face. If the projected point lies within two pixels of the centroid of a 2D dot visible in the reference camera view then it is added to the list of potential 3D candidate positions for that 2D dot. This is the list of potential 3D matches for a given 2D dot.

For each 3D point in the potential 3D match list, $\binom{n}{3}$ possible combinations of three points in the 3D point list are computed and the combination with the smallest variance is chosen as the true 3D position. Then all 3D points which lie within a user defined distance, in our case the sphere subtended by a cone two pixels in radius at the distance of the 3D point, are averaged to generate the final 3D dot position. This 3D dot position is assigned to the corresponding 2D dot in the reference camera view.

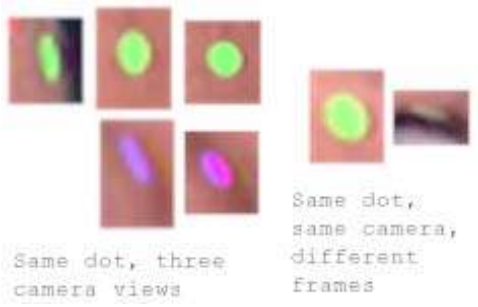

Figure 5: Dot variation. Left: Two dots seen from three different cameras (the purple dot is occluded in one camera's view). Right: A single dot seen from a single camera but in two different frames.

This algorithm could clearly be made more efficient because many more 3D candidate points are generated then necessary. One could search for potential camera to camera correspondences only along the epipolar lines and use a variety of space subdivision techniques to find 3D candidate points to test for a given 2D point. However, because the number of fiducials in each color set is small (never more than 40) both steps of this simple and robust algorithm are reasonably fast, taking less than a second to generate the 2D dot correspondences and 3D dot positions for six camera views. The 2D dot correspondence calculation is dominated by the time taken to read in the images of the six camera views and to locate the 2D dots in each view. Consequently, the extra complexity of more efficient stereo matching algorithms does not appear to be justified.

3.3 Frame to frame dot correspondence and labeling

We now have a set of unlabeled 3D dot locations for each frame. We need to assign, across the entire sequence, consistent labels to the 3D dot locations. We do this by defining a reference set of dots D and matching this set to the 3D dot locations given for each frame. We can then describe how the reference dots move over time as follows: Let $d_j \in D$ be the neutral location for the reference dot j. We define the position of d_j at frame i by an offset, i.e.,

$$d_j^i = d_j + \vec{v}_j^i \tag{1}$$

Because there are thousands of frames and 182 dots in our data set we would like the correspondence computation to be automatic and quite efficient. To simplify the matching we used a fiducial pattern that separates fiducials of a given color as much as possible so that only a small subset of the unlabeled 3D dots need be checked for a best match. Unfortunately, simple nearest neighbor matching fails for several reasons: some fiducials occasionally disappear, some 3D dots may move more than the average distance between 3D dots of the same color, and occasionally extraneous 3D dots appear, caused by highlights in the eyes or teeth. Fortunately, neighboring fiducials move similarly and we can exploit this fact, modifying the nearest neighbor matching algorithm so that it is still efficient but also robust.

For each frame i we first move the reference dots to the locations found in the previous frame. Next, we find a (possibly incomplete) match between the reference dots and the

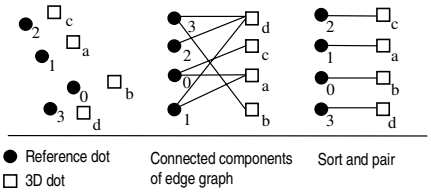

Figure 6: Matching dots.

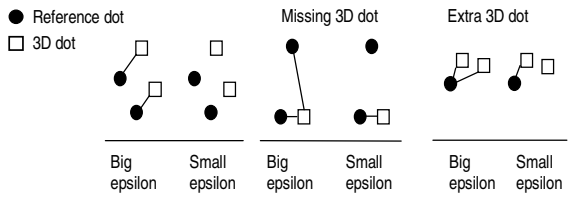

Figure 7: Examples of extra and missing dots and the effect of different values for ϵ.

3D dot locations for frame i. We then move each matched reference dot to the location of its corresponding 3D dot. If a reference dot does not have a match we "guess" a new location for it by moving it in the same direction as its neighbors. We then perform a final matching step.

3.3.1 Acquiring the reference set of dots

The cyberware scan was taken with the dots glued onto the face. Since the dots are visible in both the geometric and color information of the scan, we can place the reference dots on the cyberware model by manually clicking on the model. We next need to align the reference dots and the model with the 3D dot locations found in frame zero. The coordinate system for the cyberware scan differs from the one used for the 3D dot locations, but only by a rigid body motion plus a uniform scale. We find this transform as follows: we first hand-align the 3D dots from frame zero with the reference dots acquired from the scan, then call the matching routine described in Section 3.3.2 below to find the correspondence between the 3D dot locations, f_i, and the reference dots, d_i. We use the method described in [9] to find the exact transform, T, between the two sets of dots. Finally, we replace the temporary locations of the reference dots with $d_i = f_i$.
and use T^{-1} to transform the cyberware model into the coordinate system of the video 3D dot locations.

3.3.2 The matching routine

The matching routine is run twice per frame. We first perform a conservative match, move the reference dots (as described below in Section 3.3.3), then perform a second, less conservative, match. By moving the reference dots between matches we reduce the problem of large 3D dot position displacements.

The matching routine can be thought of as a graph problem where an edge between a reference dot and a frame dot indicates that the dots are potentially paired (see Figure 6). The matching routine proceeds in several steps; first, for each reference dot we add an edge for every 3D dot of the same color that is within a given distance ϵ. We then search for connected components in the graph that have an equal number of 3D and reference dots (most connected components will have exactly two dots, one of each type). We sort the dots in the vertical dimension of the plane of the face and use the resulting ordering to pair up the reference dots with the 3D dot locations (see Figure 6).

In the video sequences we captured, the difference in the 3D dot positions from frame to frame varied from zero to about 1.5 times the average distance separating closest dots. To adjust for this, we run the matching routine with several values of ϵ and pick the run that generates the most matches. Different choices of ϵ produce different results (see Figure 7): if ϵ is too small we may not find matches for 3D dots that have moved a lot. If ϵ is too large then the connected components in the graph will expand to include too many 3D dots. We try approximately five distances ranging from 0.5 to 1.5 of the average distance between closest reference dots.

If we are doing the second match for the frame we add an additional step to locate matches where a dot may be missing (or extra). We take those dots which have not been matched and run the matching routine on them with smaller and smaller ϵ values. This resolves situations such as the one shown on the right of Figure 7.

3.3.3 Moving the dots

We move all of the matched reference dots to their new locations then interpolate the locations for the remaining, unmatched reference dots by using their nearest, matched neighbors. For each reference dot we define a valid set of neighbors using the routine in Section 4.2.1, ignoring the blending values returned by the routine.

To move an unmatched dot d_k we use a combination of the offsets of all of its valid neighbors (refer to Equation 1). Let $n_k \subset D$ be the set of neighbor dots for dot d_k. Let $\hat{n}_k$ be the set of neighbors that have a match for the current frame i. Provided $\hat{n}_k \neq \emptyset$, the offset vector for dot d_k^i is calculated as follows: let $\vec{v}_j^i = d_j^i - d_j$ be the offset of dot j (recall that d_j is the initial position for the reference dot j).

$$\vec{v}_k^i = \frac{1}{||\hat{n}_k||} \sum_{d_j^i \in \hat{n}_k} \vec{v}_j^i$$

If the dot has no matched neighbors we repeat as necessary, treating the moved, unmatched reference dots as matched dots. Eventually, the movements will propagate through all of the reference dots.

4 Mesh construction and deformation

4.1 Constructing the mesh

To construct a mesh we begin with a cyberware scan of the head. Because we later need to align the scan with the 3D video dot data, we scanned the head with the fiducials glued on. The resulting scan suffers from four problems:

- The fluorescent fiducials caused "bumps" on the mesh.
- Several parts of the mesh were not adequately scanned, namely, the ears, one side of the nose, the eyes, and under the chin. These were manually corrected.
- The mesh does not have an opening for the mouth.
- The scan has too many polygons.

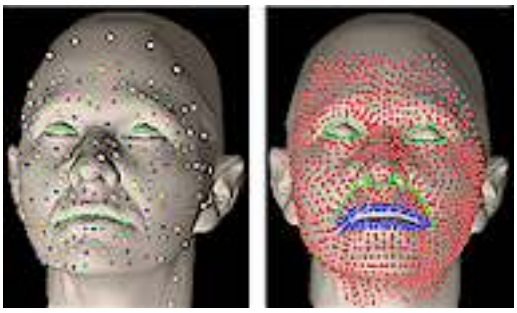

Figure 8: Left: The original dots plus the extra dots (in white). The labeling curves are shown in light green. Right: The grid of dots. Outline dots are green or blue.

The bumps caused by the fluorescent fiducials were removed by selecting the vertices which were out of place (approximately 10-30 surrounding each dot) and automatically finding new locations for them by blending between four correct neighbors. Since the scan produces a rectangular grid of vertices we can pick the neighbors to blend between in (u, v) space, i.e., the nearest valid neighbors in the positive and negative u and v direction.

The polygons at the mouth were split and then filled with six rows of polygons located slightly behind the lips. We map the teeth and tongue onto these polygons when the mouth is open.

We reduced the number of polygons in the mesh from approximately $460,000$ to 4800 using Hoppe's simplification method [8].

4.2 Moving the mesh

The vertices are moved by a linear combination of the offsets of the nearest dots (refer to Equation 1). The linear combination for each vertex v_j is expressed as a set of blend coefficients, α_k^j, one for each dot, such that $\sum_{d_k \in D} \alpha_k^j = 1$ (most of the α_k^js will be zero). The new location p_j^i of the vertex v_j at frame i is then

$$p_j^i = p_j + \sum_k \alpha_k^j ||d_k^i - d_k||$$

where p_j is the initial location of the vertex v_j.

For most of the vertices the α_k^js are a weighted average of the closest dots. The vertices in the eyes, mouth, behind the mouth, and outside of the facial area are treated slightly differently since, for example, we do not want the dots on the lower lip influencing vertices on the upper part of the lip. Also, although we tried to keep the head as still as possible, there is still some residual rigid body motion. We need to compensate for this for those vertices that are not directly influenced by a dot (e.g., the back of the head).

We use a two-step process to assign the blend coefficients to the vertices. We first find blend coefficients for a grid of points evenly distributed across the face, then use this grid of points to assign blend coefficients to the vertices. This two-step process is helpful because both the fluorescent fiducials and the mesh vertices are unevenly distributed across the face, making it difficult to get smoothly changing blend coefficients.

The grid consists of roughly 1400 points, evenly distributed and placed by hand to follow the contours of the face (see Figure 8). The points along the nasolabial furrows, nostrils, eyes, and lips are treated slightly differently than the other points to avoid blending across features such as the lips.

Because we want the mesh movement to go to zero outside of the face, we add another set of unmoving dots to the reference set. These new dots form a ring around the face (see Figure 8) enclosing all of the reference dots. For each frame we determine the rigid body motion of the head (if any) using a subset of those reference dots which are relatively stable. This rigid body transformation is then applied to the new dots.

We label the dots, grid points, and vertices as being *above*, *below*, or *neither* with respect to each of the eyes and the mouth. Dots which are *above* a given feature can not be combined with dots which are *below* that same feature (or vice-versa). Labeling is accomplished using three curves, one for each of the eyes and one for the mouth. Dots directly above (or below) a curve are labeled as *above* (or *below*) that curve. Otherwise, they are labeled *neither*.

4.2.1 Assigning blends to the grid points

The algorithm for assigning blends to the grid points first finds the closest dots, assigns blends, then filters to more evenly distribute the blends.

Finding the ideal set of reference dots to influence a grid point is complicated because the reference dots are not evenly distributed across the face. The algorithm attempts to find two or more dots distributed in a rough circle around the given grid point. To do this we both compensate for the dot density, by setting the search distance using the two closest dots, and by checking for dots which will both "pull" in the same direction.

To find the closest dots to the grid point p we first find δ_1 and δ_2, the distance to the closest and second closest dot, respectively. Let $D_n \subset D$ be the set of dots within $1.8\frac{\delta_1+\delta_2}{2}$ distance of p whose labels do not conflict with p's label. Next, we check for pairs of dots that are more or less in the same direction from p and remove the furthest one. More precisely, let $\hat{v}_i$ be the normalized vector from p to the dot $d_i \in D_n$ and let $\hat{v}_j$ be the normalized vector from p to the dot $d_j \in D_n$. If $\hat{v}_1 \cdot \hat{v}_2 > 0.8$ then remove the furthest of d_i and d_j from the set D_n.

We assign blend values based on the distance of the dots from p. If the dot is not in D_n then its corresponding α value is 0. For the dots in D_n let $l_i = \frac{1.0}{||d_i - p||}$. Then the corresponding α's are

$$\alpha_i = \frac{l_i}{(\sum_{d_i \in D_n} l_i)}$$

We next filter the blend coefficients for the grid points. For each grid point we find the closest grid points – since the grid points are distributed in a rough grid there will usually be 4 neighboring points – using the above routine (replacing the dots with the grid points). We special case the outlining grid points; they are only blended with other outlining grid points. The new blend coefficients are found by taking 0.75 of the grid point's blend coefficients and 0.25 of the average of the neighboring grid point's coefficients. More formally, let $g_i = [\alpha_0, \ldots, \alpha_n]$ be the vector of blend coefficients for the grid point i. Then the new vector g_i' is found as follows, where N_i is the set of neighboring grid points for the grid point i:

$$g_i' = 0.75 g_i + \frac{0.25}{||N_i||} \sum_{j \in N_i} g_j$$

Figure 9: Masks surrounding important facial features. The gradient of a blurred version of this mask is used to orient the low-pass filters used in the dot removal process.

Figure 10: Standard cylindrical texture map. Warped texture map that focuses on the face, and particularly on the eyes and mouth. The warp is defined by the line pairs shown in white.

We apply this filter twice to simulate a wide low pass filter.

To find the blend coefficients for the vertices of the mesh we find the closest grid point with the same label as the vertex and copy the blend coefficients. The only exception to this is the vertices for the polygons inside of the mouth. For these vertices we take β of the closest grid point on the top lip and $1.0 - \beta$ of the closest grid point on the bottom lip. The β values are 0.8, 0.6, 0.4, 0.25, and 0.1 from top to bottom of the mouth polygons.

5 Dot removal

Before we create the textures, the dots and their associated illumination effects have to be removed from the camera images. Interreflection effects are surprisingly noticeable because some parts of the face fold dramatically, bringing the reflective surface of some dots into close proximity with the skin. This is a big problem along the naso-labial furrow where diffuse interreflection from the colored dots onto the face significantly alters the skin color.

First, the dot colors are removed from each of the six camera image sequences by substituting skin texture for pixels which are covered by colored dots. Next, diffuse interreflection effects and any remaining color casts from stray pixels that have not been properly substituted are removed.

The skin texture substitution begins by finding the pixels which correspond to colored dots. The nearest neighbor color classifier described in Section 3.1.1 is used to mark all pixels which have any of the dot colors. A special training set is used since in this case false positives are much less detrimental than they are for the dot tracking case. Also, there is no need to distinguish between dot colors, only between dot colors and the background colors. The training set is created to capture as much of the dot color and the boundary region between dots and the background colors as possible.

A dot mask is generated by applying the classifier to each pixel in the image. The mask is grown by a few pixels to account for any remaining pixels which might be contaminated by the dot color. The dot mask marks all pixels which must have skin texture substituted.

The skin texture is broken into low spatial frequency and high frequency components. The low frequency components of the skin texture are interpolated by using a directional low pass filter oriented parallel to features that might introduce intensity discontinuities. This prevents bleeding of colors across sharp intensity boundaries such as the boundary between the lips and the lighter colored regions around the mouth. The directionality of the filter is controlled by a two dimensional mask which is the projection into the image plane of a three dimensional polygon mask lying on the 3D face model. Because the polygon mask is fixed on the 3D mesh, the 2D projection of the polygon mask stays in registration with the texture map as the face deforms.

All of the important intensity gradients have their own polygon mask: the eyes, the eyebrows, the lips, and the naso-labial furrows (see 9). The 2D polygon masks are filled with white and the region of the image outside the masks is filled with black to create an image. This image is low-pass filtered. The intensity of the resulting image is used to control how directional the filter is. The filter is circularly symmetric where the image is black, i.e., far from intensity discontinuities, and it is very directional where the image is white. The directional filter is oriented so that its long axis is orthogonal to the gradient of this image.

The high frequency skin texture is created from a rectangular sample of skin texture taken from a part of the face that is free of dots. The skin sample is highpass filtered to eliminate low frequency components. At each dot mask pixel location the highpass filtered skin texture is first registered to the center of the 2D bounding box of the connected dot region and then added to the low frequency interpolated skin texture.

The remaining diffuse interreflection effects are removed by clamping the hue of the skin color to a narrow range determined from the actual skin colors. First the pixel values are converted from RGB to HSV space and then any hue outside the legal range is clamped to the extremes of the range. Pixels in the eyes and mouth, found using the eye and lip masks shown in Figure 9, are left unchanged.

Some temporal variation remains in the substituted skin texture due to imperfect registration of the high frequency texture from frame to frame. A low pass temporal filter is applied to the dot mask regions in the texture images, because in the texture map space the dots are relatively motionless. This temporal filter effectively eliminates the temporal texture substitution artifacts.

6 Creating the texture maps

Figure 11 is a flowchart of the texture creation process. We create texture maps for every frame of our animation in a four-step process. The first two steps are performed only once per mesh. First we define a parameterization of the mesh. Second, using this parameterization, we create a

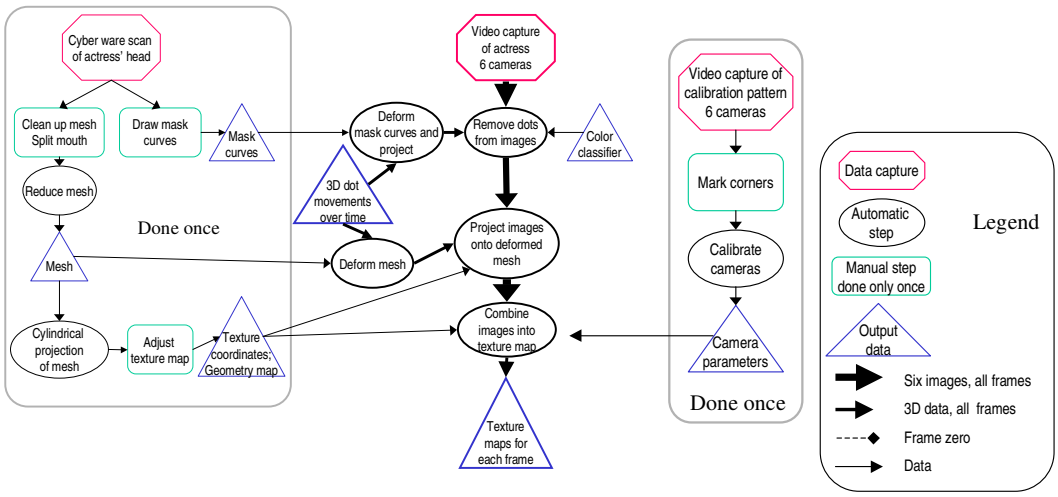

Figure 11: Creating the texture maps.

geometry map containing a location on the mesh for each texel. Third, for every frame, we create six preliminary texture maps, one from each camera image, along with weight maps. The weight maps indicate the relative quality of the data from the different cameras. Fourth, we take a weighted average of these texture maps to make our final texture map.

We create an initial set of texture coordinates for the head by tilting the mesh back 10 degrees to expose the nostrils and projecting the mesh vertices onto a cylinder. A texture map generated using this parametrization is shown on the left of Figure 10. We specify a set of line pairs and warp the texture coordinates using the technique described by Beier and Neely[1]. This parametrization results in the texture map shown on the right of Figure 10. Only the front of the head is textured with data from the six video streams.

Next we create the geometry map containing a mesh location for each texel. A mesh location is a triple (k, β_1, β_2) specifying a triangle k and barycentric coordinates in the triangle $(\beta_1,\ \beta_2,\ 1-\beta_1-\beta_2)$. To find the triangle identifier k for texel (u, v) we exhaustively search through the mesh's triangles to find the one that contains the texture coordinates (u, v). We then set the β_is to be the barycentric coordinates of the point (u, v) in the texture coordinates of the triangle k. When finding the mesh location for a pixel we already know in which triangles its neighbors above and to the left lie. Therefore, we speed our search by first searching through these triangles and their neighbors. However, the time required for this task is not critical as the geometry map need only be created once.

Next we create preliminary texture maps for frame f one for each camera. This is a modified version of the technique described in [11]. To create the texture map for camera c, we begin by deforming the mesh into its frame f position. Then, for each texel, we get its mesh location, (k, β_1, β_2), from the geometry map. With the 3D coordinates of triangle k's vertices and the barycentric coordinates β_i, we compute the texel's 3D location t. We transform t by camera c's projection matrix to obtain a location, (x, y), on camera c's image plane. We then color the texel with the color from camera c's image at (x, y). We set the texel's weight to the dot product of the mesh normal at t, $\hat{n}$, with the direction back to the camera, $\hat{d}$ (see Figure 12). Negative values are clamped to zero. Hence, weights are low where the camera's view is glancing. However, this weight map is not smooth at triangle boundaries, so we smooth it by convolving it with a Gaussian kernel.

Last, we merge the six preliminary texture maps. As they do not align perfectly, averaging them blurs the texture and loses detail. Therefore, we use only the texture map of our bottom, center camera for the center 46 % of the final texture map. We smoothly transition (over 23 pixels) to using a weighted average of each preliminary texture map at the sides.

We texture the parts of the head not covered by the aforementioned texture maps with the captured reflectance data from our Cyberware scan, modified in two ways. First, because we replaced the mesh's ears with ears from a stock mesh (Section 4.1), we moved the ears in the texture to achieve better registration. Second, we set the alpha channel to zero (with a soft edge) in the region of the texture for the front of the head. Then we render in two passes to create an image of the head with both texture maps applied.

7 Compression

7.1 Principal Components Analysis

The geometric and texture map data have different statistical characteristics and are best compressed in different ways. There is significant long-term temporal correlation in the geometric data since similar facial expressions occur throughout the sequence. The short term correlation of the texture data is significantly increased over that of the raw video footage because in the texture image space the fiducials are essentially motionless. This eliminates most of the intensity changes associated with movement and leaves primarily shading changes. Shading changes tend to have low spatial frequencies and are highly compressible. Compression schemes such as MPEG, which can take advantage of short term temporal correlation, can exploit this increase in short term correlation.

For the geometric data, one way to exploit the long term correlation is to use principal component analysis. If we represent our data set as a matrix A, where frame i of the data maps column i of A, then the first principal component of A is

$$\max_u (A^T u)^T (A^T u) \qquad (2)$$

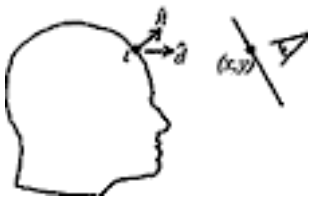

Figure 12: Creating the preliminary texture map.

The u which maximizes Equation 2 is the eigenvector associated with the largest eigenvalue of AA^T, which is also the value of the maximum. Succeeding principal components are defined similarly, except that they are required to be orthogonal to all preceding principal components, i.e., $u_i^T u_j = 0$ for $j \neq i$. The principal components form an orthonormal basis set represented by the matrix U where the columns of U are the principal components of A ordered by eigenvalue size with the most significant principal component in the first column of U.

The data in the A matrix can be projected onto the principal component basis as follows:

$$W = U^T A$$

Row i of W is the projection of column A_i onto the basis vector u_i. More precisely, the jth element in row i of W corresponds to the projection of frame j of the original data onto the ith basis vector. We will call the elements of the W matrix projection *coefficients*.

Similarly, A can be reconstructed exactly from W by multiplication by the basis set, i.e., $A = UW$.
The most important property of the principal components for our purposes is that they are the best linear basis set for reconstruction in the l_2 norm sense. For any given matrix U_k, where k is the number of columns of the matrix and $k < rank(A)$, the reconstruction error

$$e = ||A - U_k U_k^T A||_F^2 \tag{3}$$

where $||B||_F^2$ is the Frobenius norm defined to be

$$||B||_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} b_{ij}^2 \tag{4}$$

will be minimized if U_k is the matrix containing the k most significant principal components of A.

We can compress a data set A by quantizing the elements of its corresponding W and U matrices and entropy coding them. Since the compressed data cannot be reconstructed without the principal component basis vectors both the W and U matrices have to be compressed. The basis vectors add overhead that is not present with basis sets that can be computed independent of the original data set, such as the DCT basis.

For data sequences that have no particular structure the extra overhead of the basis vectors would probably outweigh any gain in compression efficiency. However, for data sets with regular frame to frame structure the residual error for reconstruction with the principal component basis vectors can be much smaller than for other bases. This reduction in residual error can be great enough to compensate for the overhead bits of the basis vectors.

The principal components can be computed using the singular value decomposition (SVD) [13]. Efficient implementations of this algorithm are widely available. The SVD of a matrix A is

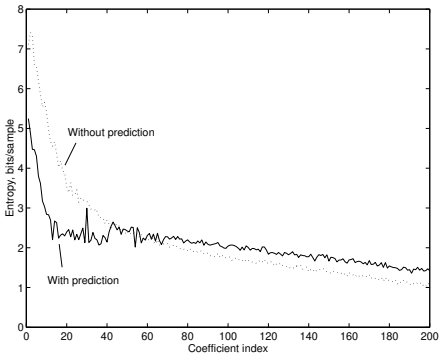

Figure 13: Reduction in entropy after temporal prediction.

$$A = U\Sigma V^T \tag{5}$$

where the columns of U are the eigenvectors of AA^T, the singular values, σ_i, along the diagonal matrix Σ are the square roots of the eigenvalues of AA^T, and the columns of V are the eigenvectors of $A^T A$. The ith column of U is the ith principal component of A. Computing the first k left singular vectors of A is equivalent to computing the first k principal components.

7.2 Geometric Data

The geometric data has the long term temporal coherence properties mentioned above since the motion of the face is highly structured. The overhead of the basis vectors for the geometric data is fixed because there are only 182 fiducials on the face. The maximum number of basis vectors is $182 * 3$ since there are three numbers, x, y, and z, associated with each fiducial. Consequently, the basis vector overhead steadily diminishes as the length of the animation sequence increases.

The geometric data is mapped to matrix form by taking the 3D offset data for the ith frame and mapping it the ith column of the data matrix A_g. The first k principal components, U_g, of A_g are computed and A_g is projected into the U_g basis to give the projection coefficients W_g.

There is significant correlation between the columns of projection coefficients because the motion of the dots is relatively smooth over time. We can reduce the entropy of the quantized projection coefficients by temporally predicting the projection coefficients in column i from column $i-1$, i.e., $c_i = c_{i-1} + \Delta_i$ where we encode Δ_i.

For our data set, only the projection coefficients associated with the first 45 principal components, corresponding to the first 45 rows of W_g, have significant temporal correlation so only the first 45 rows are temporally predicted. The remaining rows are entropy coded directly. After the temporal prediction the entropy is reduced by about 20 percent (Figure 13).

The basis vectors are compressed by choosing a peak error rate and then varying the number of quantization levels allocated to each vector based on the standard deviation of the projection coefficients for each vector.

We visually examined animation sequences with W_g and U_g compressed at a variety of peak error rates and chose a level which resulted in undetectable geometric jitter in reconstructed animation. The entropy of W_g for this error level is 26 Kbits per second and the entropy of U_g is 13

kbits per second for a total of 40 kbits per second for all the geometric data. These values were computed for our 3330 frame animation sequence.

8 Results

Figure 16 shows some typical frames from a reconstructed sequence of 3D facial expressions. These frames are taken from a 3330 frame animation in which the actress makes random expressions while reading from a script[2].

The facial expressions look remarkably life-like. The animation sequence is similarly striking. Virtually all evidence of the colored fiducials and diffuse interreflection artifacts is gone, which is surprising considering that in some regions of the face, especially around the lips, there is very little of the actress' skin visible – most of the area is covered by colored fiducials.

Both the accurate 3D geometry and the accurate face texture contribute to the believability of the reconstructed expressions. Occlusion contours look correct and the subtle details of face geometry that are very difficult to capture as geometric data show up well in the texture images. Important examples of this occur at the nasolabial furrow which runs from just above the nares down to slightly below the lips, eyebrows, and eyes. Forehead furrows and wrinkles also are captured. To recreate these features using geometric data rather than texture data would require an extremely detailed 3D capture of the face geometry and a resulting high polygon count in the 3D model. In addition, shading these details properly if they were represented as geometry would be difficult since it would require computing shadows and possibly even diffuse interreflection effects in order to look correct. Subtle shading changes on the smooth parts of the skin, most prominent at the cheekbones, are also captured well in the texture images.

There are still visible artifacts in the animation, some of which are polygonization or shading artifacts, others of which arise because of limitations in our current implementation.

Some polygonization of the face surface is visible, especially along the chin contour, because the front surface of the head contains only 4500 polygons. This is not a limitation of the algorithm – we chose this number of polygons because we wanted to verify that believable facial animation could be done at polygon resolutions low enough to potentially be displayed in real time on inexpensive ($200) 3D graphics cards[3]. For film or television work, where real time rendering is not an issue, the polygon count can be made much higher and the polygonization artifacts will disappear. As graphics hardware becomes faster the differential in quality between offline and online rendered face images will diminish.

Several artifacts are simply the result of our current implementation. For example, occasionally the edge of the face, the tips of the nares, and the eyebrows appear to jitter. This usually occurs when dots are lost, either by falling below the minimum size threshold or by not being visible to three or more cameras. When a dot is lost the algorithm synthesizes dot position data which is usually incorrect enough that it is visible as jitter. More cameras, or better placement of the cameras, would eliminate this problem. However, overall the image is extremely stable.

In retrospect, a mesh constructed by hand with the correct geometry and then fit to the cyberware data [10] would be much simpler and possibly reduce some of the polygonization artifacts.

Another implementation artifact that becomes most visible when the head is viewed near profile is that the teeth and tongue appear slightly distorted. This is because we do not use correct 3D models to represent them. Instead, the texture map of the teeth and tongue is projected onto a sheet of polygons stretching between the lips. It is possible that the teeth and tongue could be tracked using more sophisticated computer vision techniques and then more correct geometric models could be used.

Shading artifacts represent an intrinsic limitation of the algorithm. The highlights on the eyes and skin remain in fixed positions regardless of point of view, and shadowing is fixed at the time the video is captured. However, for many applications this should not be a limitation because these artifacts are surprisingly subtle. Most people do not notice that the shading is incorrect until it is pointed out to them, and even then frequently do not find it particularly objectionable. The highlights on the eyes can probably be corrected by building a 3D eye model and creating synthetic highlights appropriate for the viewing situation. Correcting the skin shading and self shadowing artifacts is more difficult. The former will require very realistic and efficient skin reflectance models while the latter will require significant improvements in rendering performance, especially if the shadowing effect of area light sources is to be adequately modeled. When both these problems are solved then it will no longer be necessary to capture the live video sequence – only the 3D geometric data and skin reflectance properties will be needed.

The compression numbers are quite good. Figure 14 shows a single frame from the original sequence, the same frame compressed by the MPEG4 codec at 460 Kbps and at 260 KBps. All of the images look quite good. The animated sequences also look good, with the 260 KBps sequence just beginning to show noticeable compression artifacts. The 260 KBps video is well within the bandwidth of single speed CDROM drives. This data rate is probably low enough that decompression could be performed in real time in software on the fastest personal computers so there is the potential for real time display of the resulting animations. We intend to investigate this possibility in future work.

There is still room for significant improvement in our compression. A better mesh parameterization would significantly reduce the number of bits needed to encode the eyes, which distort significantly over time in the texture map space. Also the teeth, inner edges of the lips, and the tongue could potentially be tracked over time and at least partially stabilized, resulting in a significant reduction in bit rate for the mouth region. Since these two regions account for the majority of the bit budget, the potential for further reduction in bit rate is large.

9 Conclusion

The system produces remarkably lifelike reconstructions of facial expressions recorded from live actors' performances. The accurate 3D tracking of a large number of points on the face results in an accurate 3D model of facial expression. The texture map sequence captured simultaneously with the 3D deformation data captures details of expres-

[2] The rubber cap on the actress' head was used to keep her hair out of her face.

[3] In this paper we have not addressed the issue of real time texture decompression and rendering of the face model, but we plan to do so in future work

Figure 14: Left to Right: Mesh with uncompressed textures, compressed to 400 kbits/sec, and compressed to 200 kbits/sec

sion that would be difficult to capture any other way. By using the 3D deformation information to register the texture maps from frame to frame the variance of the texture map sequence is significantly reduced which increases its compressibility. Image quality of 30 frame per second animations, reconstructed at approximately 300 by 400 pixels, is still good at data rates as low as 240 Kbits per second, and there is significant potential for lowering this bit rate even further. Because the bit overhead for the geometric data is low in comparison to the texture data one can get a 3D talking head, with all the attendant flexibility, for little more than the cost of a conventional video sequence. With the true 3D model of facial expression, the animation can be viewed from any angle and placed in a 3D virtual environment, making it much more flexible than conventional video.

REFERENCES

[1] Beier, T., and Neely, S. Feature-based image metamorphosis. In *Computer Graphics (SIGGRAPH '92 Proceedings)* (July 1992), E. E. Catmull, Ed., vol. 26, pp. 35–42.

[2] Bregler, C., Covell, M., and Slaney, M. Video rewrite: Driving visual speech with audio. *Computer Graphics 31*, 2 (Aug. 1997), 353–361.

[3] Cassell, J., Pelachaud, C., Badler, N., Steedman, M., Achorn, B., Becket, T., Douville, B., Prevost, S., and Stone, M. Animated conversation: Rule-based generation of facial expression, gesture and spoken intonation for multiple conversational agents. *Computer Graphics 28*, 2 (Aug. 1994), 413–420.

[4] DeCarlo, D., and Metaxas, D. The integration of optical flow and deformable models with applications to human face shape and motion estimation. *Proceedings CVPR* (1996), 231–238.

[5] Essa, I., and Pentland, A. Coding, analysis, interpretation and recognition of facial expressions. *IEEE Transactions on Pattern Analysis and Machine Intelligence 19*, 7 (1997), 757–763.

[6] Faugeras, O. *Three-dimensional computer vision.* MIT Press, Cambridge, MA, 1993.

[7] Fischler, M. A., and Booles, R. C. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM 24*, 6 (Aug. 1981), 381–395.

[8] Hoppe, H. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 99–108. held in New Orleans, Louisiana, 04-09 August 1996.

[9] Horn, B. K. P. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America 4*, 4 (Apr. 1987).

[10] Lee, Y., Terzopoulos, D., and Waters, K. Realistic modeling for facial animation. *Computer Graphics 29*, 2 (July 1995), 55–62.

[11] Pighin, F., Auslander, J., Lishinski, D., Szeliski, R., and Salesin, D. Realistic facial animation using image based 3d morphing. Tech. Report TR-97-01-03, Department of Computer Science and Engineering, University of Washington, Seattle, Wa, 1997.

[12] Schürmann, J. *Pattern Classification: A Unified View of Statistical and Neural Approaches.* John Wiley and Sons, Inc., New York, 1996.

[13] Strang. *Linear Algebra and its Application.* HBJ, 1988.

[14] Waters, K. A muscle model for animating three-dimensional facial expression. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), M. C. Stone, Ed., vol. 21, pp. 17–24.

[15] Williams, L. Performance-driven facial animation. *Computer Graphics 24*, 2 (Aug. 1990), 235–242.

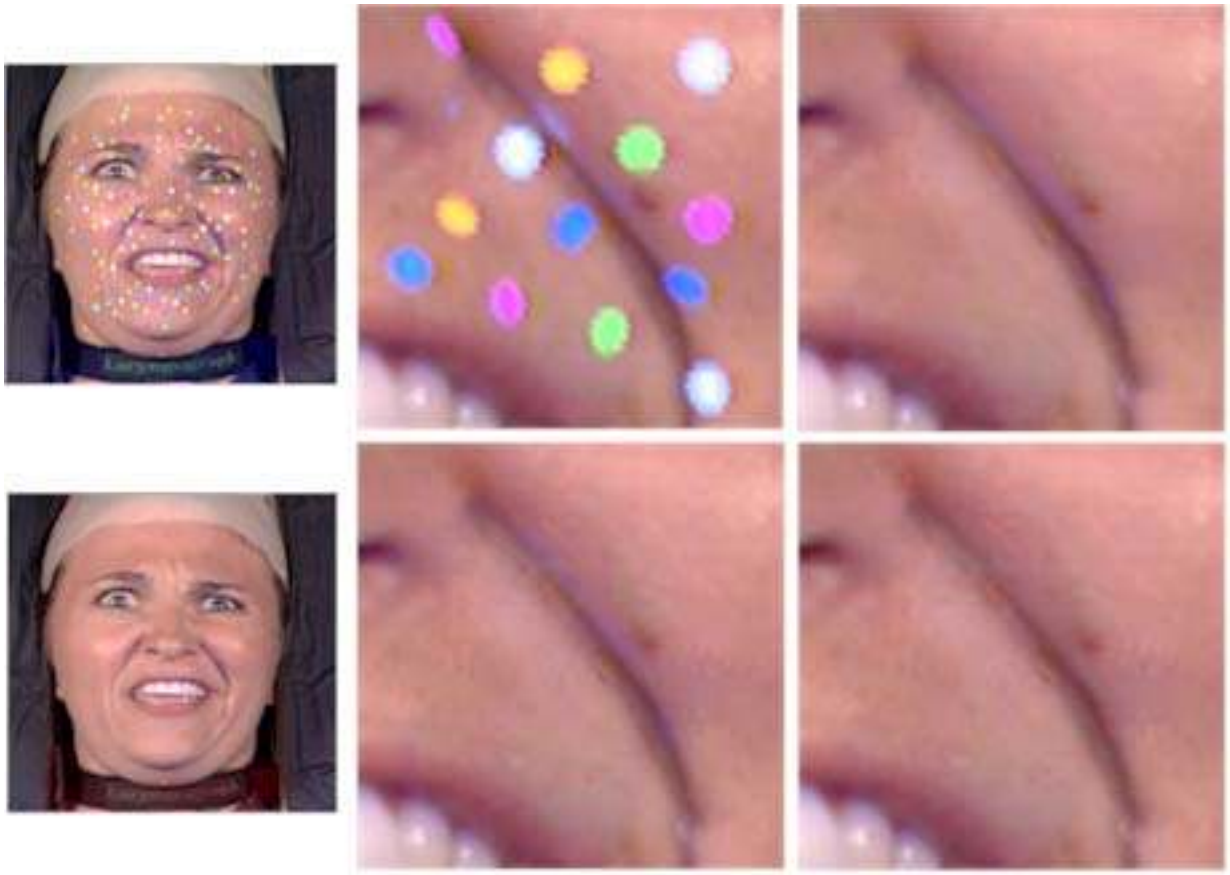

Figure 15: Face before and after dot removal, with details showing the steps in the dot removal process. From left to right, top to bottom: Face with dots, dots replaced with low frequency skin texture, high frequency skin texture added, hue clamped.

Figure 16: Sequence of rendered images of textured mesh.