Facial Tracking and Animation
1st Progress Report

Todd Belote
Brad Busse
David Brown
Bryan Harris

Monday, February 23, 2004
**Introduction**
So far we have experimented with various improvements for the project: using QuickTime for portability, creating an initialization algorithm, and finding a way to restore perspective distortions from a skewed 2D image. Also, we have learned how to calculate FAPUs from pixel movements of feature points.

The current libraries for video capture are used to interface with the current video capture card. QuickTime is a very complicated set of libraries, and therefore we will not be using it.

Our feature point initialization algorithm will be very similar to the feature point tracking algorithm previously used. Instead of defining a small area of the image, however, the initialization algorithm will search for feature points across the entire screen.

The proposed algorithm for restoring an image which is skewed or rotated uses several feature points on a circle and another coming out away from the person’s head on a pin. Several calculations are performed to find the rotation angles, and a matrix is created and applied to the rest of the feature points to account for the rotation.

**Video Capture Emulation Program**

To insure parallel development, we have identified the need to be able to validate and test our tracking algorithms on multiple machines. In order to do so, we must be able to execute our program on systems without the video capture card, camera and microphone. For this we are developing a Video Capture Emulation Program. This program will utilize pre-recorded movies as the data source for algorithm testing.

Our initial proposed design called for the use of Apple’s QuickTime Format for emulation due to its portability and extensive libraries. Upon thorough investigation, it has been determined that using the QuickTime Libraries is not the best means to our solution. Using the QuickTime format will complicate the design of the Video Capture Emulation Program, and a much easier solution can be implemented using Microsoft’s Audio Video Interleave format.

Microsoft Visual Studio by default has a simplistic API dedicated to the controlling of AVI files and streams. The AVIFile API has direct function calls to handle the frame extraction from AVI files and streams (AVIStreamGetFrame, AVIFileReadData). The AVIFile API also has a function call which parses and returns pertinent AVI file information. Also, since we are developing our system to run on Windows machines, the AVI file format will be portable enough for our needs, since it was developed for Windows applications.

The current application has the built-in capability to generate AVI files during execution. This will be useful because when an error occurs during run-time, the input data can be reapplied to the system, instead of having to recreate the event which caused the error.
The Video Capture Emulation Program will be implemented in a cyclical manner. First we will develop a standalone program that opens an AVI file and passes the video data at 30 frames per second to a standalone version of the processing algorithm. This system will not interface with the hardware. The second version of the system will be integrated into the system that interfaces with the hardware. A level of abstraction will be implemented to handle the selecting of hardware data acquisition, or software simulated data acquisition. The third version of the Video Capture Emulation System, if proven useful, will emulate audio data in synchrony with the video data.

**Point Initialization**

One of the major steps in the current application is point initialization. This involves requesting a still image from the audio video card, manually selecting the points, and then returning to the main part of the application. The points are identified by the order in which they are selected. We are developing an automatic point initialization process, so that the user doesn’t have to manually select or identify the points.

The new system will also request a still from the card, but then will find the points automatically. Currently we have developed an algorithm to do this, and implemented the algorithm to find the first point. The algorithm works by defining a minimum density or connection level, and minimum size to define a point, and then searches for a section of the bitmap which meets these requirements.

As for identifying this is accomplished by noticing geometric relationships between the 22 facial points. First the limits of all the points are found. Then the used portion of the bitmap is divided into the two eyebrow sections, the cheek sections, and the mouth section. The eyebrows are then just two apiece, left and right, one on each cheek, and then the mouth. The mouth can be derived by starting with the left most point and just walking across.

**Affine transformations**

One of the primary requirements for this system is the ability to robustly handle losing data points. When the data point is no longer observed, the system must accept this without crashing. Should a point (re)appear, the system must also determine which of the original data points it corresponds with.

To accomplish this, we intend to use a series of markers arranged so that we can determine the parameters necessary to transform the facial image we have received into one which is centered and directly facing the camera. This image can then be compared with our initial data point capture to map the observed data to initial data, reattaching or correcting any data points that may have been lost or corrupted in the interim. Our proposed orientation pattern is a series of markers arranged in a circle, with a single marker in the center, and another centered in the circle but extended some amount into the Z-axis. This is illustrated below.
In order to transform the observed arrangement of markers into one that can be compared with the original, we use our orientation markers to determine an affine transformation matrix, which we then use on each observed data marker to find its original position. The transformation matrix can be broken down into a series of component matrices: an XY translation matrix, a scaling matrix (for Z translations), and X, Y, and Z rotational matrices.

The first step is to remove the translation differences between the two sets of data points. For X and Y translations, the answer is simple: translate the observed points so that the central orientation marker overlaps in both sets. Determining the amount of Z translation is more complicated, since at this point we have only X and Y coordinates for each data point. To correct for Z displacement, we use the fact that if a circle is rotated in three dimensions, the major axis of the resulting ellipse is always the same length as the diameter of the original circle. Using the circular set of orientation markers, if we take the longest observed distance between opposite points as the major axis, we can get a reasonable approximation of what the diameter of the circle would be if the observed points were viewed head-on. With 12 or greater circular points, our approximation should be accurate enough for most tasks we wish to accomplish (see figure below). We can use the ratio between the original and derived diameter as a scaling factor to eliminate any translations in the Z dimension.
This results in a data set composed of a number of original data points and the output of a rotational transformation performed on them. We can perform a series of straightforward calculations to find a rotational matrix that will transform our observed coordinate system into our initial coordinates. This process is graphically illustrated in the appendix, but can be briefly summarized as follows: Since we know the relationships between our centered and normalized orientation markers, we can easily find the markers’ Z values, giving us an \( \{X,Y,Z\} \) coordinate for each marker. Then, we choose an axis to align with the original. In this case, we began with the Y axis. This corrects any rotation about the Z-axis, and lines up another set of axes for the second step. We then correct the Y axis for rotation. This cancels out any rotation about the X-axis, and lines up the X axes for the final step. We repeat the process for the X axis, removing any Y-axis rotation.

The result of all this is a matrix capable of transforming any data point relative to the observed position of the head into its corresponding location relative to the default head position. Once this matrix is applied to each observed marker, a simple search can be conducted to determine which data points are missing or inaccurate. Any errant data points can then be corrected before positional information is passed to the feature point identification algorithm.

**Feature Point Identification**

The current priority of the Feature Point Identification algorithm is complete understanding of the solution that is being used currently. Once the algorithm is completed, alterations and improvements can be decided on and implemented. The current implementation of feature points involves four steps. The first step is finding the points initially by clicking the top-left corner with the mouse. The second step is tracking the feature points as the person moves during speech. The third step converts the measurements into FAP units (FAPUs) and stores them in an array. The fourth step is
writing the FAP information to the FAP file in the correct format. Each of these three topics will only be discussed to the extent that it has in common with calculating FAPUs.

To start measuring facial point movements the user must first initialize the points. This is done by clicking on a still image, which is also considered in the source code to be the first frame of the video. The user should click the top-left portion of each point; the software will add five pixels in x and y to compensate such that the final position marked is closer to the center of the point. These reference positions are later used in the tracking algorithm to find the points on the next frame.

The tracking algorithm then takes over the process to keep track of each of the feature points; it stores them into temporary variables so that they can be later converted to FAPUs --- which are then stored in an array. At this step in the tracking process each point is stored in a positional array, with units of pixels.

The conversion process into FAPUs takes place before the points are stored in the array. The points are normalized with an equation for each of the 22 points. Normalization occurs by first converting each pixel into centimeters by multiplying by a conversion constant --- one pixel = .02645 centimeters. Then the point is normalized by dividing by one of the predefined facial measurements --- ES, ENS, MNS, or MW depending on the feature point. The points are now normalized integers from 0 to 1024 and are saved into the FAP file.

The format of the FAP file includes a header and then the FAP information itself in text format. The header specifies extra information such as which model to use. The next lines are the feature point information; each measurement is contained in a set of two lines of the file. The first line in each set contains only ones, zeros, and spaces; it is a bit mask which informs the FAE which points we will be using and which to ignore. The second line is exactly the same except the ones are replaced by the values measured in step three above. The file is closed after this information is written and it is then ready to be read in by the FAE.

**Conclusion**

According to the Gantt chart provided with our project proposal, we are making progress steadily and on time. The first version of the Video Capture Emulation Program will be complete prior to Critical Design Review (CDR). By Wednesday of this week the rest of the point discovery process will be coded. The week after that will be used developing the point identification process. An algorithm has been developed to reorient data points for tracking, and will be translated into C++ within the next few weeks. Finally, the original Feature Point Identification algorithm has been examined, and we are currently investigating areas which need improvement.
Appendix: Facial Orientation Correction

The above graph charts the process of orientation correction:
- Original orientation: Black
- Observed orientation: Red
- After Z axis correction: Yellow
- After X axis correction: Green
- After Y axis correction: Blue