FAST IMAGE WARPING FOR ACTIVE MODELS

Jörgen Ahlberg

Report No. LiTH-ISY-R-2355
May 14, 2001

Image Coding Group, Dept. of Electrical Engineering,
Linköping University, SE-581 83 Linköping, SWEDEN
ahlberg@isy.liu.se

Abstract

In this paper our system for face and facial feature tracking using an Active Model is described. The focus of the paper is image warping methods for achieving real-time performance, in order to show that it is possible to implement a real-time model based coding system on consumer hardware.

1. Introduction

Our goal is to find and track a human face and its features in a video sequence. We want to do this by adapting, in each frame, a wireframe model to the face in the image. This should be done accurately enough to allow a realistic-looking facial animation to be created from the extracted face model parameters, and it should also have real-time performance.

This paper describes how we use an Active Model to adapt the model to a face in the image, and then track the face through an image sequence. The main focus is on how to lower the computational complexity and exploit the hardware of the experiment platform to achieve real-time performance.

The Active Model finds locally optimal face model parameters. Thus, a quite accurate à priori estimate of the face model parameters (at least the size and position of the face) must be available. In a video sequence, we can use the parameters extracted from the previous frame as the à priori estimate. For the first frame of the sequence, we need an additional algorithm, for example a colour or motion based face candidate finder. Such algorithms are not treated in this paper.

Section 2 treats the parameterization of the face model and how to find the optimum parameters. Section 3 describes our implementation and finds the bottlenecks. Section 4 describes how we speed up the necessary image warping step so that we can achieve real-time performance. In Section 5, some additional optimizations are treated, followed by our conclusions in Section 6.

2. The Active Model

The concept of Active Appearance Models (AAMs) was introduced a few years ago [1], and has been the subject of several reports and investigations, especially by the original inventors [2]. Together with the AAMs came a directed search algorithm for adapting the model to an image, here referred to as the Active Appearance Algorithm (AAA). The AAA can be used on a complete AAM, or, as here, on a simpler model being parameterized in geometry and texture separately.

The face model is a wireframe model with a texture mapped on its surfaces. The texture is represented as a standard-shaped image, being a linear combination of a set of *texture modes* or *geometrically normalized eigenfaces* [3]. We formulate this as

$$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{X}\tau \qquad (1)$$

where $\bar{\mathbf{x}}$ is the mean texture, the columns of $\mathbf{X}$ are the texture modes and τ is the vector of texture parameters. The synthesized texture $\mathbf{x}$ is mapped on the wireframe model.

The geometry of the wireframe model is parameterized according to

$$\mathbf{g} = \bar{\mathbf{g}} + \mathbf{S}\sigma + \mathbf{A}\alpha \qquad (2)$$

where the resulting vector $\mathbf{g}$ contains the (x, y, z) coordinates of the vertices of the model. $\bar{\mathbf{g}}$ is the standard shape of the model,

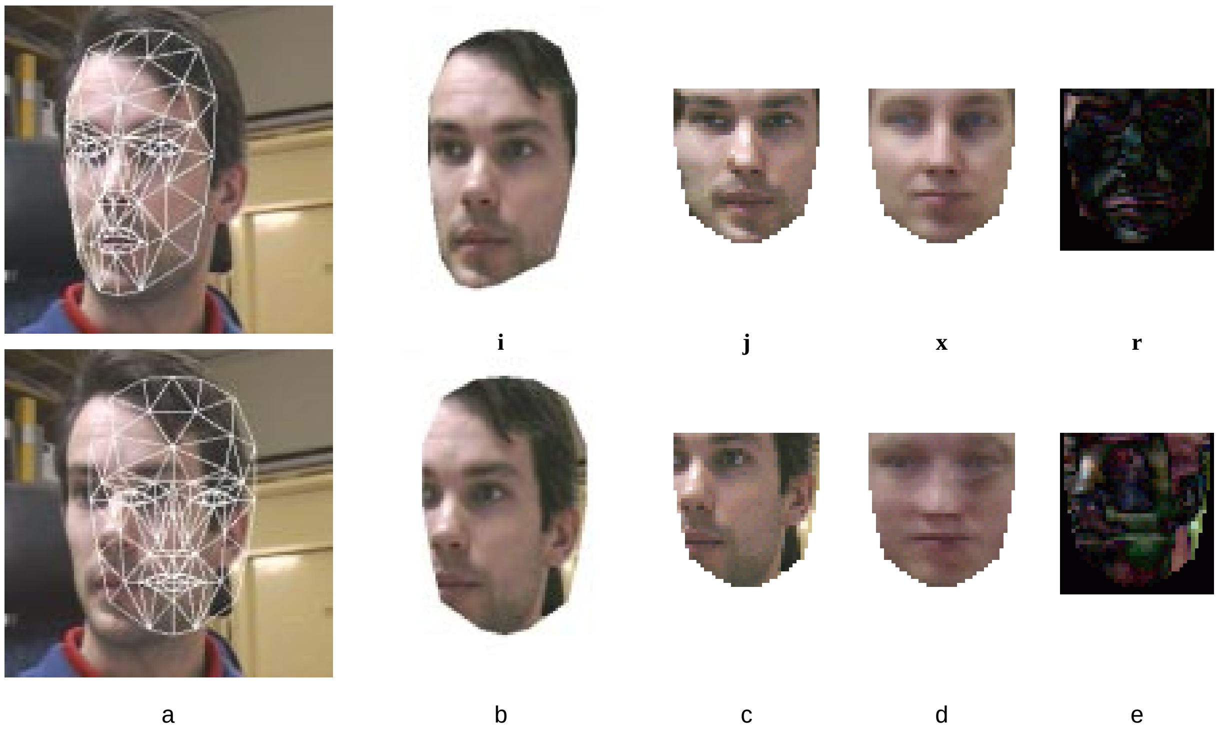

Figure 1. The model matching and texture analysis-synthesis process. a) A good and a bad (top and bottom row respectively) model adaptation is shown. b) The image mapped onto the model. c) The model is reshaped to the standard shape, producing the image **j**. d) The normalized texture is approximated by the texture modes, producing the image **x**. e) The residual image **r** is computed. The images **j** and **x** are more similar the better the model adaptation is.

and the columns of **S** and **A** are the Shape and Animation Units respectively, and thus σ and α contain the shape and animation parameters.

Since we also want to perform global motion, we need six more parameters for rotation, scaling, and translation. Thus, we replace (2) with

$$\mathbf{g} = s\mathbf{R}(\bar{\mathbf{g}} + \mathbf{S}\sigma + \mathbf{A}\alpha) + \mathbf{t} \qquad (3)$$

where $\mathbf{R} = R(r_x, r_y, r_z)$ is a rotation matrix, s is the scale, and $\mathbf{t} = t(t_x, t_y)$ is the 2D translation vector.

The geometry of our model is thus parameterized by the parameter vector

$$\mathbf{p} = [r_x, r_y, r_z, s, t_x, t_y, \sigma, \alpha]^{\mathrm{T}}. \qquad (4)$$

Note that this differs from the original formulation of the AAMs, where out-of-plane rotation is rather built-in in the Shape Units than a separate parameter. Another difference is that in the original formulation, there is no distinction between Shape Units and Animation Units.

When adapting a model to a video sequence, the shape parameters σ should only be changed in the first frame(s) – the head shape does not vary during a conversation – while the animation parameters α and the global parameters naturally change each frame.

The shape parameters can be converted to MPEG-4 Facial Definition Parameters (FDPs) and the animation parameters to MPEG-4 Facial Animation Parameters (FAPs).

2.1. Matching the model and the image

Our goal is to find the optimal adaptation of the model to the input image, that is to find the **p** that minimizes the distance between the model and the image. As distance measure, we choose the summed squared error (SSE) between the re-mapped input image and the synthesized texture. We compute this by, for a given **p**, reshaping the model according to $\mathbf{g}(\mathbf{p})$, and warping the input image **i** onto the model. We then reshape the model to the standard shape, $\bar{\mathbf{g}}$, and get the resulting image as a vector

$$\mathbf{j} = \mathbf{j}(\mathbf{i}, \mathbf{g}(\mathbf{p})). \qquad (5)$$

This image can be approximated by the texture modes by computing texture parameters according to

$$\tau(\mathbf{p}) = \mathbf{X}^{\mathrm{T}}(\mathbf{j}(\mathbf{p}) - \bar{\mathbf{x}}) . \qquad (6)$$

Inserting this in (1), we get

$$\mathbf{x}(\mathbf{p}) = \bar{\mathbf{x}} + \mathbf{X}\mathbf{X}^{\mathrm{T}}(\mathbf{j}(\mathbf{p}) - \bar{\mathbf{x}}) \qquad (7)$$

and we compute the residual image as

$$\mathbf{r}(\mathbf{p}) = \mathbf{j}(\mathbf{p}) - \mathbf{x}(\mathbf{p}) , \qquad (8)$$

and the SSE as

$$e(\mathbf{p}) = \|\mathbf{r}(\mathbf{p})\|^2 \qquad (9)$$

The entire process is illustrated in Figure 1.

2.2. Finding the optimal parameters

With this formulation, our goal is to find the parameter vector **p** that for a given input image **i** minimizes $e(\mathbf{p})$. We do that by using the Active Appearance Algorithm (AAA) in the following way. For a starting value of **p**, supposed to be close to the optimum, we compute $\mathbf{r}(\mathbf{p})$ and $e(\mathbf{p})$, and find the update vector $\Delta\mathbf{p}$ by multiplying the residual image with an update matrix:

$$\Delta\mathbf{p} = \mathbf{U}\mathbf{r}(\mathbf{p}) . \qquad (10)$$

The vector $\Delta\mathbf{p}$ gives us a probable direction in the search space, and we compute a new parameter vector and an new SSE:

$$\mathbf{p}' = \mathbf{p} + \Delta\mathbf{p} , \qquad (11)$$

$$e' = e(\mathbf{p}') . \qquad (12)$$

If $e' < e$, we update **p** accordingly ($\mathbf{p}' \rightarrow \mathbf{p}$) and iterate until convergence. If $e' > e$, we try smaller update steps (0.5 and 0.25). If neither of these improves the SSE, we declare convergence. The update matrix **U**, that we create in advance by training from example images with models correctly adapted, as explained below.

2.3. Creating the update matrix

Taylor-expanding **r** around $\mathbf{p} + \Delta\mathbf{p}$, we can write

$$\mathbf{r}(\mathbf{p} + \Delta\mathbf{p}) = \mathbf{r}(\mathbf{p}) + \mathbf{R}\Delta\mathbf{p} + O(\Delta\mathbf{p}^2) \qquad (13)$$

where

$$\mathbf{R} = \frac{\partial}{\partial\mathbf{p}}\mathbf{r}(\mathbf{p}) . \qquad (14)$$

Given a **p** (and thus an $\mathbf{r}(\mathbf{p})$), we want to find the $\Delta\mathbf{p}$ that minimizes

$$e(\mathbf{p} + \Delta\mathbf{p}) = \|\mathbf{r}(\mathbf{p}) + \mathbf{R}\Delta\mathbf{p} + O(\Delta\mathbf{p}^2)\|^2 \qquad (15)$$

which we approximate as

$$e(\mathbf{p} + \Delta\mathbf{p}) \approx \|\mathbf{r}(\mathbf{p}) + \mathbf{R}\Delta\mathbf{p}\|^2 . \qquad (16)$$

Minimizing (16) is a least squares problem with the solution

$$\Delta\mathbf{p} = -(\mathbf{R}^{\mathrm{T}}\mathbf{R})^{-1}\mathbf{R}\mathbf{r}(\mathbf{p}) , \qquad (17)$$

which gives us the update matrix **U** as the negative pseudo-inverse of the gradient matrix **R**:

$$\mathbf{U} = -\mathbf{R}^{\dagger} = -(\mathbf{R}^{\mathrm{T}}\mathbf{R})^{-1}\mathbf{R} . \qquad (18)$$

To be able to use the AAA, we should consequently estimate the gradient matrix **R**. We do that by perturbating **p** from a set of (manually) adapted models, parameter by parameter, step by step. The *j*:th row in **R** can thus be estimated as

$$\mathbf{R}_j = \sum_k (\mathbf{r}(\mathbf{p} + \Delta\mathbf{p}_{jk}) - \mathbf{r}(\mathbf{p})) , \qquad (19)$$

where $\Delta\mathbf{p}_{jk}$ is a vector that perturbs **p** in the *j*:th component to the amount of $k \cdot c$ for some suitable constant *c*.

2.4. Training the model

To try out this scheme, the CANDIDE-3 model [4] has been (manually in the beginning, then semi-automatically) adapted to 257 images of five different persons from different angles and with different facial expressions. As Animation Units, the following Action Units from CANDIDE-3 have been chosen:

1. Jaw drop
2. Lip stretcher
3. Lip corner depressor
4. Upper lip raiser
5. Eyebrow lowerer
6. Outer eyebrow raiser

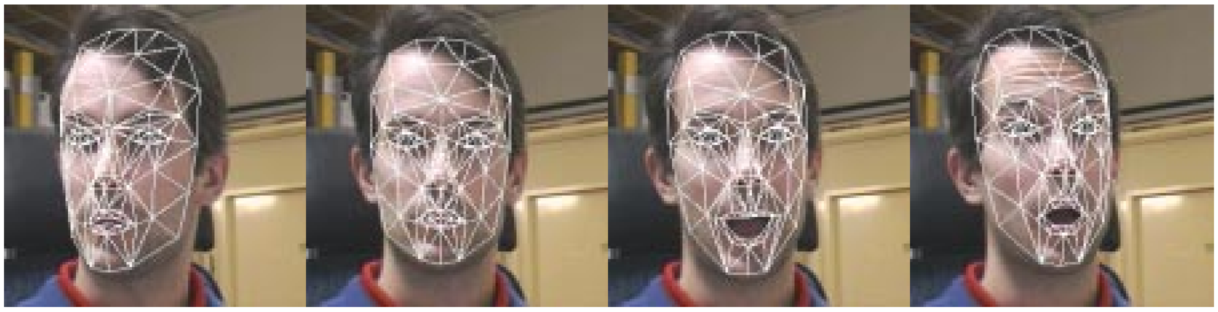

Figure 2. The model adapted to four frames of a video sequence.

The adapted models have, for each image, been normalized to a standard shape with the size 40x42 pixels, as Figure 1 c), top row, and the resulting training textures collected in a matrix. A PCA has been performed on this matrix, to compute the texture modes. The mean texture $\bar{\mathbf{x}}$ and the DC-level have been subtracted from the training textures prior to the PCA.

When the texture modes are available, all the parameters have been perturbed, one by one and for each image, in steps of 0.01 in the range [-0.1, 0.1], and the matrix **R** estimated. Finally, from **R**, the update matrix **U** has been computed.

3. Implementation

The algorithm has been implemented in Visual C++. The images have been captured using a Sony EVI-D31 camera and digitized with an Asus V3800 Ultra graphics card with video input. The tests has been performed on a PC with a 500 MHz Pentium III processor. Except for the high-quality camera, this is quite typical consumer hardware that can be found in many homes.

Resulting face model adaptations in a video sequence are show in Figure 2. As can be seen, the global adaptation (rotation, scale, translation) is good and the mouth and eyebrow parameters behave well. All parameters suffer from the algorithm being greedy and easily getting stuck in a local optimum. Typically, this results in the model not being able to follow fast moves, like when the mouth is closed too fast or when the head is moved away quickly. The higher the frame rate, the smaller this problem becomes, and already at a few Hertz normal head motion is handled.

When running the algorithm on live video, input directly from a camera, the computation time is very critical. If the time for each iteration could be reduced somewhat, the frame rate would, of course, be higher. Noticing that the algorithm needs fewer iterations if there is small motion between each frame, it is clear that higher frame rate implies that fewer iterations would be needed each frame, which would improve the frame rate even more.

Studying the algorithm, there are three time consuming parts; the image warping, the texture analysis-synthesis, and the update vector computation.

Timing the computations needed in each iteration gives the measurements below:

Task	Time (ms)
Image warping	~2000
Analysis-synthesis	3.3
Update vector computation	1.1
Total time per iteration	~2000

Apparently, the image warping step (5), is the one that needs to be optimized, as discussed below.

The second step, corresponding to (6) - (9), is the projection of the normalized image onto the eigentextures, reconstruction, and SSE computation.

The vector update computation (10) needs 0.09 ms per parameter being optimized; the time in the table is for 12 parameters.

4. Image Warping

In the first step above, we want to warp an image from one shape to another using a mesh of triangles (in two different shapes). In our case, the destination can be regarded as a two dimensional mesh, which simplifies things.

We thus have:

- One destination mesh **M** containing triangles $\mathbf{M}_0$, ... $\mathbf{M}_N$. Each triangle is a triplet of (2D) vertex coordinates, that is,

$$\mathbf{M}_n = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix} \tag{20}$$

- One source mesh $\mathbf{M}'$.
- One source image $f(x, y)$.
- One destination image $\tilde{f}(x, y)$.

The warping process is as follows:

1. For each pixel coordinate (x, y) in the destination image, compute the barycentric coordinates (a, b, c) with respect to the first triangle in the destination mesh. That is, find a, b, c so that

$$(x, y) = (a, b, c)\mathbf{M}_0. \tag{21}$$

   The barycentric coordinates are valid if (and only if)

$$\begin{cases} 0 \le a \le 1 \\ 0 \le b \le 1 \\ 0 \le c \le 1 \end{cases}. \tag{22}$$

   If the barycentric coordinates are not valid, that is, the pixel is not inside the triangle, try the next triangle until the correct triangle n is found.

2. Compute the source coordinates (x', y') from the barycentric coordinates applied to the corresponding triangle in the source mesh. That is, $(x', y') = (a, b, c)\mathbf{M}'_n$.

3. Interpolate the source image f' in (x', y') and set $f(x, y) = f'(x', y')$.

Since the pixel at (x, y) often corresponds to the same triangle as $(x - 1, y)$ we can by trying the previous triangle first speed up step one significantly. Our new measurement is 170 ms for the image warping – much better than 2 seconds, but it will still not do if we want a real-time system. We will therefore study two alternative ways of speeding up the image warping.

4.1. *Method 1: Hardware acceleration*

Almost all PCs of today have a graphics card with specialized hardware for 3D graphics, including texture mapping. Since the computation of the normalized texture **j**(**i**, **g**) (see (5)) is essentially a texture mapping, it could be suitable to use the graphics card hardware for this computation.

We assume in the following that the image **i** is already present in the graphics card texture memory. This is a reasonable assumption since the image should be displayed anyway. To compute **j**(**i**, **g**) we then let the graphics card render a wireframe model with geometry $\bar{\mathbf{g}}$. The wireframe should use **i** as texture, and use the texture coordinates given by **g**. The result will be a normalized image **j** as shown in Figure 1c).

The normalized image is rendered to the graphics card memory, and it needs thus to be transferred to the main memory to be further processed by the CPU. This transfer might actually be more time-consuming than the rendering.

Measuring the times on the experiment platform gives the following results:

Task	Time (ms)
Render	0.7
Transfer	4.7
Total time (image warping)	5.4

This is a huge improvement, and it enables real-time performance in our system.

4.2. *Method 2: Barycentric coordinate pre-computation*

The hardware in the graphics card is specialized for animating 3D mesh objects with texture mapped onto them and/or with different shadings and light sources. In such situations the texture coordinates are usually fixed, and the object coordinates are variable.

In our situation, we have the opposite situation; the object coordinates (the normalized/destination shape) are fixed, and the texture coordinates are variable. This has an important implication: The barycentric coordinates are always the same for each pixel in the destination image! This means that we can compute them once, and store *a*, *b*, *c*, and *n* for each pixel. Thus, we only perform steps 2 and 3 in the image warping process in real-time. Fortunately, almost all the time needed for the image warping is spent on step 1 – the remaining two steps are very cheap:

Task	Time (ms)
Step 2: Source coordinates	0.5
Step 3: Interpolation	0.4
Total time (image warping)	0.9

The time for interpolation is when using linear interpolation. Using the nearest neighbour, the time is halved; using cubic interpolation, the time is doubled.

Thus, since step 1 can be omitted, we can reduce the time per iteration from our original 2 seconds to less than one millisecond.

4.3. Discussion

Using either the specialized hardware of the graphics card (method 1) or pre-computing the barycentric coordinates and triangle indices (method 2), the image warping can be done fast enough to allow a real-time implementation. In our tests, method 2 was six times as fast as method 1, but this might be different on another platform. For example, on a platform without floating point SIMD instructions, but with a fast graphics card and graphics bus, it might well be that it is better to use the graphics hardware than the processor.

Since the image warping can be executed in only a millisecond, we now have now reason to take a look at the analysis-synthesis part as well.

5. Analysis-Synthesis and Update Vector Computation

The analysis-synthesis of the normalized image consists mainly of a set of vector operations (see equations (6) - (9)) that can be very fast using floating point SIMD instructions in modern processors. The same goes for the update vector computation (10).

5.1. Grayscale computations

One way to decrease the computations time even more, is to use grayscale instead of colour images. This requires re-training of the system, creating grayscale eigentextures and update images, and could also influence the accuracy. Practical tests show no noticeable difference in accuracy, though.

Using grayscale, the computation time for the analysis-synthesis step drops from 3.3 ms to 0.9 ms and the update vector computation from 1.1 ms to 0.6 ms, a gain of 2.9 ms. However, this also requires the input data to be converted to grayscale, which takes some time:

- Using the hardware acceleration method (image warping method 1), it is easy to specify that the data should be read back as grayscale values. The graphics card will then perform the colour-to-grayscale conversion. The time for the data transfer (including the conversion) increases from 4.7 to 5.0 ms. The total gain is thus 2.6 ms.
- Using method 2, the conversion takes about 0.6 ms. The total gain is thus 2.3 ms.

The resulting total computing times are as follows:

Task	CPU	Gfx card
Image warping	0.9	5.7
Analysis-Synthesis	0.9	0.9
Update vector	0.6	0.6
Total time per iteration	2.4	7.2

6. Conclusion

By using features of the hardware (the graphics card, the SIMD capability of the CPU) in an efficient way and/or algorithm specific solutions, we can perform an Active Model iteration in a few milliseconds on consumer hardware. Typically, about 10 iterations per frame are needed, which means that the facial feature tracking could be run in about 40 Hz on the experiment platform. In practice, the video capture and transfer requires some time as well, allowing the system to run at half that speed.

The main goal, to show that it is possible to implement a real-time model based coder on consumer hardware, is achieved.

Acknowledgements

This work is sponsored by the Swedish SSF-project VISIT and the European IST-project InterFace. The author is thankful to Dr. Tim Cootes at the University of Manchester for advice on the AAA training.

References

[1] T. F. Cootes, G. J. Edwards, and C. J. Taylor, “Active Appearance Models,” *Proc. 5th European Conference on Computer Vision*, pp. 484 - 498, 1998.

[2] T. F. Cootes and C. J. Taylor, *Statistical Models of Appearance for Computer Vision*, Draft report, Wolfson Image Analysis Unit, University of Manchester, December 2000. http://www.wiau.man.ac.uk

[3] J. Ström et al., “Very Low Bit Rate Facial Texture Coding,” *Proc. Int. Workshop Synthetic/Natural Hybrid Coding and 3D Imaging*, Rhodes, Greece, pp. 237 - 240, September 1997.

[4] J. Ahlberg, *CANDIDE-3 - an updated parameterized face*, Report No. LiTH-ISY-R-2326, Dept. of EE, Linköping University, Sweden, January 2001. http://www.icg.isy.liu.se/candide