AN ABSTRACT OF THE PROJECT REPORT OF

David Burri for the degree of Master of Science in Computer Science presented on August 14, 2012.

Title: Shape Modeling and GPU Based Image Warping

Abstract approved: ______________________________

Prof. Mike Bailey

This project addresses the problems of manually placing facial landmarks on a portrait and finding a fast way to warp the annotated image of a face. While there are many approaches to automatically find facial landmarks, most of them provide insufficient results in uncontrolled environments. Thus I introduce a method to manually adjust a non-rigid shape on a portrait. This method utilizes a statistical shape model based on point distribution models. With these manually placed landmarks the image of a face can be warped into another shape. To warp the image I use a piecewise affine transformation. This way of transforming, however, tends to be computationally intense and therefore slow. Thus in the second part of the project I introduce a way to perform a piecewise affine transformation with enhanced performance using shaders in OpenGL. This project is made in collaboration with the Pedagogical University of Berne, Switzerland and will be part of a system for diversity research named chic-o-mat. Eventually the system will run on an iPhone as an application available to the public. Therefore, the provided solutions are based on iPhone programming using the multi-touch screen for the shape adjustment and the GPU of the latest iPhone 4S. A test application demonstrates up to 20X speedup performing piecewise warping using the GPU.

Shape Modeling and GPU Based Image Warping

by

David Burri

A PROJECT REPORT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented August 14, 2012
Commencement June 2013

TABLE OF CONTENTS

Page

1 Introduction 1

2 Related Work 3

2.1 chic-o-mat^{v2} . . . 3

2.2 Face Exchange . . . 4

2.3 Active Models . . . 5

3 iPhone Programming 6

3.1 Objective-C and iOS SDK . . . 6

3.2 Hardware . . . 7

3.3 NEON SIMD . . . 7

3.4 GPU . . . 8

3.5 SIMD Performance . . . 8

3.5.1 Test 1 . . . 9

3.5.2 Test 2 . . . 10

4 Shape Adjustment 12

4.1 Point Distribution Model . . . 13

4.1.1 Align a set of shapes . . . 13

4.1.2 Build a model . . . 14

4.1.3 Building new shapes . . . 15

4.2 Database . . . 15

4.3 Shape Modeling . . . 17

4.3.1 Match Shape to New Points . . . 17

4.3.2 Non-rigid Shape . . . 18

4.4 Implementation . . . 19

4.4.1 Model Training and Model Data . . . 20

4.4.2 Touch Shape Modeling . . . 20

4.4.3 Application . . . 22

5 Piecewise Affine Warp 23

5.1 Background . . . 23

5.2 Implementation . . . 26

5.2.1 CPU Algorithm . . . 27

TABLE OF CONTENTS (Continued)

Page

5.2.2 GPU Algorithm 29

6 Results 31

6.1 Shape Modeling 31

6.1.1 Conclusion 32

6.2 Piecewise Affine Warp 33

6.2.1 Performance Against Image Size 35

6.2.2 Performance Against Image Coverage 37

6.2.3 Conclusion 39

7 Conclusion 42

7.1 Shape Modeling 42

7.2 Piecewise Affine Warp 43

Bibliography 43

Appendices 48

A Symbolic Solution to Affine Transformation Matrix 49

LIST OF FIGURES

Figure Page

1.1 Examples (leftmost image = source, top row = target, bottom row = mix) [Images extracted in the research part of chic-o-mat^{v2}, see section 2.1, Portraits by Kate Burgener (C)] 2

3.1 Performance Test Matrix Multiplication (higher is better) 9

3.2 Performance Test Retrieving the Transformation Matrix (KOPS = Kilo Operations Per Second, higher is better) 11

4.1 Varying the first five modes between -4 and +4 standard deviation 16

4.2 Shape Modeling 19

4.3 Class Diagram 21

4.4 Shape Modeling Interface, Portrait by NeilsPhotography (CC) Some rights reserved. Source: `http://www.flickr.com/photos/neilspicys/` 22

5.1 Piecewise Image Warping [Portrait by Kate Burgener (C)] 23

5.2 Piecewise Image Warping of Triangles [Portrait by Kate Burgener (C)] . . 24

5.3 Mapping using Barycentric Coordinates 26

5.4 Triangle Rasterization 28

5.5 Determine Pixel Color 28

5.6 Pairs of points and interpolation of texture coordinates 30

6.1 Aligned Examples [Left: by kaybee07, `http://www.flickr.com/people/kurtbudiarto/`; Middle: by hobvias sudoneighm, `http://www.flickr.com/people/striatic/`; Right: by M Yashna, `http://www.flickr.com/people/yashna13/`; (CC) Some rights reserved.] 31

6.2 PAW Testprogram Screenshot 33

6.3 Image points transformed per second (MTPTS = Mega Pixels Transformations Per Second) 35

LIST OF FIGURES (Continued)

Figure Page

6.4 Speedup using the GPU 36

6.5 Frames per second 37

6.6 Performance with only subparts of the image transformed (top row: 640x480, bottom row: 1600x1200) 38

6.7 Performance Against Image Size - Data 40

6.8 Performance Against Image Coverage 41

LIST OF TABLES

Table Page

3.1 iPhone 4S CPU Specifications by Macotakara 7

3.2 Test 1 - Data Subset 10

4.1 Different Databases in Comparison [18] - Remark: Put Face Database provides a subset of 2193 images annotated with 194 landmarks 17

Chapter 1: Introduction

In this project I address two critical image processing parts. One common problem in image processing and computer vision is retrieving information regarding the meaning of different segments in an image. In my specific task I consider the problem of finding facial landmarks in a portrait. With this project I introduce a method to manually position and adjust a shape on an underlying image using the iPhone's touch functionalities. To simplify the procedure of adjusting this shape I use a statistical model called point distribution model to constrain deformation.

The second part of this project is about warping an annotated image into another shape. To warp the image I use a piecewise transformation. As graphic cards are becoming more popular on mobile devices, I introduce a piecewise affine warp algorithm based on OpenGL ES, exploiting the hardware abilities of the latest iPhone 4S. The introduced shader exhibits a big improvement in speedup and scalability compared to a native algorithm.

This project is made in collaboration with the Pedagogical University of Berne, Switzerland whereby Kate Burgener is the initiator and main correspondence. The work done in this project eventually will be useful for the so-called chic-o-mat system in its second revision. The chic-o-mat is a playful approach to modify a players identity based on the portrait of the player and that of another person. The chic-o-mat will finally be available to the public as an iPhone App which will be the basis of diversity research done by Kate Burgener et al. Essentially this research is about image science in the context of visual culture theory and investigates identity by virtually modifying the appearance of a user based on portraits.

The main goal of the modification is to fit a face given by an image (source image) into another person's portrait image (target image) by applying some transformation to the source face and thus generating a mixed image. The underlying transformation needs to take care of face geometry as well as skin color and structure so that the source image will be clearly recognizable in the mixed image but adjusted to the traits of the target's face. Figure 1.1 depicts some examples. The leftmost image is the source image, the top row shows three target images and the bottom row the resulting mix images (refer to [6] for more specifications).

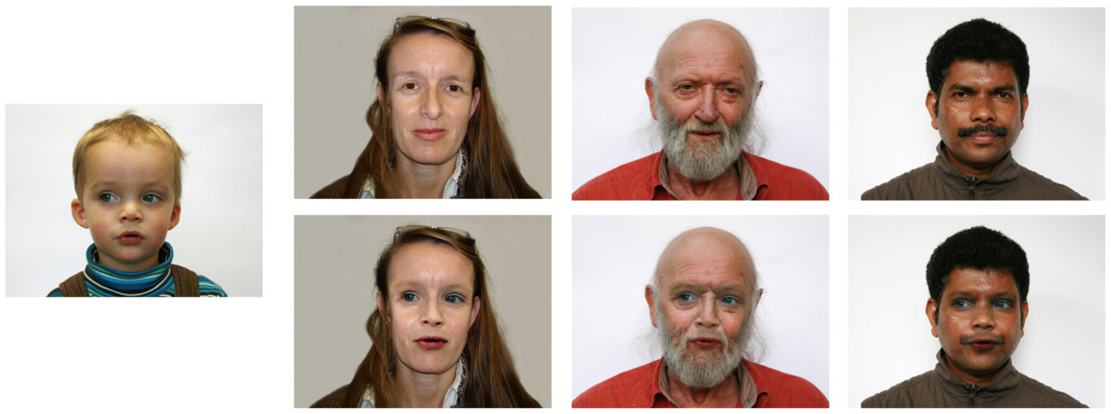


Figure 1.1: Examples (leftmost image = source, top row = target, bottom row = mix) [Images extracted in the research part of chic-o-mat^{v2}, see section 2.1, Portraits by Kate Burgener (C)]

Chapter 2: Related Work

2.1 chic-o-mat^{v2}

Before starting this project I had been working on chic-o-mat^{v2} for two months with the purpose of investigating technical capabilities to realize such a system with an emphasis on the adjustment of one face to another face given facial landmarks. In two months of research I found methods to locate a face in a portrait image, extract some characteristic points (facial landmarks) and then fit this face into another face by using the facial landmarks of the source and target images [7]. The following points present a list of steps which seemed promising in achieving good results:

- Do for both images:
  - Find the most significant face (using AdaBoost algorithm by Viola and Jones)
  - Extract facial landmarks and create point cloud (using Active Appearance Model - AAM)
  - Generate mask and cut face out of the image
- Apply the face swapping algorithm:
  - Morph the source shape (point cloud) into a mean shape gained from the two shapes and do it with the target form as well
  - Adjust color (histogram) and structure of the skin (frequency)
  - Copy the resulting face over the target face

The main approach proved to work well and will remain the same. However, there are some parts that need improvement, especially because I consider applying this to an iPhone. The main problem is extracting facial landmarks using Active Appearance Models since they do not depict a satisfying accuracy fitting landmarks into images that differ from the training set. Another problem is the performance of the exchange algorithm as a whole, but especially during the warping stage.

2.2 Face Exchange

Volker Blanz and Thomas Vetter [5] proposed in the paper 'A Morphable Model for the Synthesis of 3D Faces' a new technique to model textured 3D faces. Based on a set of 3D face models which are in full correspondence, a Morphable Face Model is derived by transforming the shape as well as the texture into a vector space representation. By varying linear combinations of the shape and the texture, new faces can be generated. To avoid unnatural looking faces, the statistical model limits the parameters. The paper then presents a method to estimate the 3D shape, orientation and illumination of a face given a single photograph. Based on a manually given hint about the size, orientation and illumination, the proposed algorithm is able to optimize the parameters to find the best match.

As an application, Blanz et al [5] show how to match a model into an image of a face, extract all the parameters such as 3D shape, position and illumination, modify the face with different facial expressions and then render the face back.

Based on this 3D Morphable Model (3DMM) Blanz et al [4] introduced an algorithm to exchange faces in images with respect to different viewpoints and illumination in the paper 'Exchanging Faces in Images'. From a single image the algorithm estimates 3D shape and texture with scene parameters like pose and lighting whereby the algorithm needs some manual interaction. They provide a web browser based application which applies the proposed algorithm. A user can upload images or take a new image from the webcam, manually label points in the image and then choose images to substitute the face in. Although this approach provides good results, it will not be applicable on an iPhone due to its cumbersome way of annotating an image.

Other related work done by Kyle McDonald and Arturo Castro [21] is Dynamic Face Substitution which is based on Jason Saragih's Face Tracker (see chapter 2.3). Although it provides good results, the problem is that the adaption of the face is much simpler than chic-o-mat^{v2} requires it. In addition the Face Tracker, which probably would provide good detection results (see chapter 2.3), is not publicly available.

2.3 Active Models

While investigating ways to detect facial landmarks I came across different approaches. First of all Cootes et al [12] introduced Active Shape Models (ASM) which is an algorithm to fit a non-rigid model into new data based on a region around each point. Subsequently Cootes et al [14] introduced the Active Appearance Model (AAM) which unlike the ASM considers the whole texture to fit the model. Since the Active Appearance Model Library which I used before and the model in general have big issues fitting the model into images that are different from the training set, at this point I introduce some papers explaining possible improvements. There is a paper published by M. B. Stegmann [26] explaining extensions and cases of AAMs. Furthermore, there are papers on Constrained AAM [10], 3D AAM [8] and other improvements for occlusion [17, 16]. Recently Liu Xiaoming [19] introduced a Boosted Appearance Model which makes use of a boosting based classifier. Another recent paper by Jason M. Saragih et al [25] introduces the Deformable Model Fitting by Regularized Landmark Mean-Shift. Both of these techniques seem to be promising improvements to an Active Shape or Active Appearance Model which could be used later on. For this project it is not yet necessary to use such an automated algorithm. Therefore, I decided to wait for improved techniques. Facial landmark extraction is a highly active research topic, promising more elaborate techniques and libraries will become available.

Nevertheless, all of the models presented above have one thing in common: they are all based on a point distribution model (PDM) building the non-rigid model to fit into new data. Since there are some promising approaches to detect facial landmarks based on this model I am also going to use the PDM for the non-rigid model introduced by Cootes et al [12] in the paper 'Active Shape Models - Smart Snakes'. Using this approach I maintain the possibility to add such an automated algorithm in a later development step. In this project I only consider manual methods.

Chapter 3: iPhone Programming

Since the final application will run on an iPhone, a substantial part of this project is to implement the proposed methods using the iOS development environment. This chapter gives insight into the iOS development, the underlying hardware as well as some techniques to exploit the performance of an iPhone 4S before moving to the shape modeling part.

3.1 Objective-C and iOS SDK

The goal of this section is to make the reader familiar with the main programming language used by Apple and the application programming interface of the iOS operating system. Apple does not provide a big choice when it comes to software development for devices such as the iPhone. Therefore, I used the Xcode Integrated Development Environment (IDE) provided by Apple including the iOS Software Development Kit (SDK) using its native programming language Objective-C.

Objective-C is the programming language used by Apple for the iOS and OS X operating systems. The language is object-oriented and is a thin layer on top of C so that it is possible to use C code inside of Objective-C classes. Objective-C is a strict superset of ANSI C whereby the object syntax is derived from the language Smalltalk to add a messaging system to the C syntax. Unlike the C++ extension to C, Objective-C does not provide any standard library. In most places, however, where Objective-C is used, a library is provided whereas Apple provides the Cocoa and Cocoa Touch libraries. [33]

Cocoa is the object-oriented Application Programming Interface (API) used to program Apple's Mac operating system OS X. For applications on Apple's mobile devices Apple provides the Cocoa Touch API which includes a user interface library adequate for mobile devices and a gesture recognition extension with support for the multi-touch screen. Cocoa Touch features a memory management system based on reference counting and makes heavy use of the model-view-controller (MVC) design pattern. [35]

The Cocoa Touch framework is part of the iOS SDK. The iOS SDK is a software development kit developed by Apple which can be used to create applications for the iPhone. This development kit is the basis of the applications I wrote for this project.

3.2 Hardware

Table 3.1 shows the CPU specifications of an iPhone 4S.

CPU Model	ARM Cortex-A9 Apple A5	L1 Cache Size	32KB
GPU Model	PowerVR SGX543MP2	L1D Cache Size	32KB
Num Cores	2	L2 Cache Size	1024KB
CPU Freq	800MHz	Byteorder	1234
BUS Freq	200MHz	Cacheline	32
TB Freq	24MHz		

Table 3.1: iPhone 4S CPU Specifications by Macotakara

The iPhone 4S uses the so called A5 System-on-Chip (SoC) which is designed by Apple and is based upon the dual-core ARM Cortex-A9 NP Core CPU and a dual core PowerVR SGX543MP2 GPU. One additional customization Apple did to the chip is the image signal processor unit (ISP) which can be used for image post-processing such as face detection, white balance and automatic image stabilization. The A5 chip includes 512MB low-power DDR2 RAM which is clocked at 533MHz. It supports some features that are used in this project. First of all the A5 has two cores allowing multi-core processing. In addition the NEON SIMD instruction set extensions of the Cortex-A9 core can perform up to 16 operations per instruction. Lastly the A5 has a GPU which is capable of a programmable pipeline with OpenGL ES. [1, 32, 30]

3.3 NEON SIMD

The advanced single instruction multiple data (SIMD) extension provided by the A5 is called NEON. NEON is a 128-bit SIMD instruction set which is similar to MMX/SSE for x86 providing speedup for multimedia and signal processing tasks. [31, 2]

There are different possibilities to use NEON in your own code. The most easy and probably the most effective one is to use either the library OpenMAX DL (Development Layer) developed by the Khronos Group or the Accelerate Framework which is part of the Apple development environment. OpenMAX DL API provides a comprehensive set of audio, video and signal processing primitives whereby the key target is to provide algorithms for different codecs. The Accelerate Framework, however, includes a wider range of algorithms exploiting vector processing. Other ways to use NEON is to use C intrinsics, Assembler code at the lowest level or rely on the vectorizing compiler. [2, 9, 15]

For my project I decided to use the Accelerate Framework already included in Xcode (Apple's development environment). It provides a nice layer of abstraction so that there is no need for the use of low-level instructions. Furthermore, it supports all necessary matrix handling functions needed to handle the point distribution model and affine warping. The Accelerate Framework includes the vImage image processing framework, the vDSP digital processing framework and for Linear Algebra LAPACK and BLAS libraries.

3.4 GPU

The A5 chip includes a dual core GPU called PowerVR SGX543MP2 from PowerVR graphics IP which is a division of the company Imagination Technologies. The PowerVR SGXMP Series 5XT family incorporates multi-core and the Universal Scalable Shader Engine (USSE). [36, 24]

GPU programming can be done with different APIs. The most common and probably the most meaningful way at the moment is OpenGL ES (Embedded System). Although the GPU has support for OpenCL, Apple does not provide a way to use it at the moment and therefore, I pursued the way of using OpenGL ES. In general, OpenGL ES is more or less a subset of OpenGL and is managed by the Khronos Group. The chip supports the OpenGL ES 2.0 whereby the most important change to predecessors is that version 2.0 requires a programmable graphics pipeline.

3.5 SIMD Performance

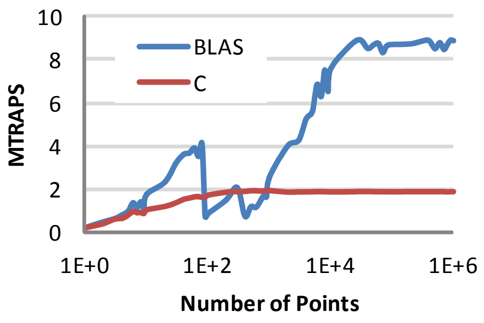
This section is dedicated to SIMD performance testing on the iPhone 4S. All tests are done on the same iPhone with the serial number C39HC0YDDT9Y and use single precision. The following tests show the abilities of the Accelerate Framework that makes use of SIMD. There are two different performance tests that will show in which cases it makes sense to rely on SIMD and in which cases it does not. Each test value is an average of 10 test samples to remove anomalies.

3.5.1 Test 1

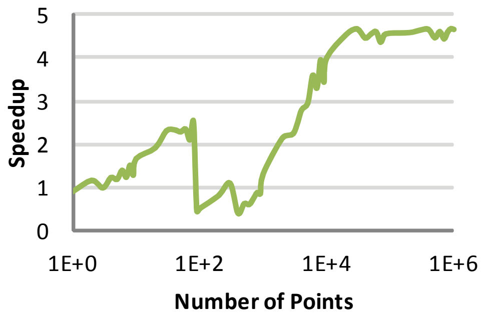
The first performance test applies an affine transformation to a certain set of 2D points. The affine transformation using homogeneous coordinates can be described as follows:

$$\begin{bmatrix} x'_1 & y'_1 & 1 \\ x'_2 & y'_2 & 1 \\ \vdots & \vdots & \vdots \\ x'_N & y'_N & 1 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_N & y_N & 1 \end{bmatrix} \begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix}$$

Obviously this transformation can be done using matrix multiplication with BLAS (the function `cblas_cgemm` in the Accelerate Framework). Figure 3.1(a) shows a performance comparison between an implementation using BLAS and one using just plain C. On the x-axis is the number of points to transform and on the y-axis the number of affine transformations applied per second in millions. Note that the x-axis uses a logarithmic scale. Figure 3.1(b) shows the resulting speedup using SIMD with BLAS against plain C without any threading.



(a) Performance using BLAS vs. plain C



(b) Speedup using BLAS

Figure 3.1: Performance Test Matrix Multiplication (higher is better)

As it is obvious from those two graphs it makes sense to use matrix multiplication with BLAS. As expected the speedup reaches a little bit more than four with bigger array sizes since the vector unit is of size 128-bit. The drop of performance using BLAS between 100 and 1000 points, however, is a strange behavior that might be due to some internal memory handling issues.

	BLAS		C		
N	**Time [s]**	**MTRAPS**	**Time [s]**	**MTRAPS**	**Speedup**
1	5.13E-06	0.1949	4.51E-06	0.2217	0.8792
5	5.19E-06	0.9642	6.04E-06	0.8274	1.1654
10	5.69E-06	1.7571	9.37E-06	1.0673	1.6463
50	1.36E-05	3.6687	3.09E-05	1.6190	2.2661
100	1.17E-04	0.8580	5.70E-05	1.7546	0.4890
500	4.25E-04	1.1754	2.56E-04	1.9547	0.6013
1000	3.94E-04	2.5398	5.09E-04	1.9658	1.2920
5000	8.97E-04	5.5739	2.62E-03	1.9055	2.9252
10000	1.32E-03	7.5829	5.26E-03	1.9024	3.9859
50000	5.75E-03	8.6895	2.61E-02	1.9141	4.5397
100000	1.15E-02	8.7024	5.24E-02	1.9098	4.5568
500000	5.88E-02	8.5070	2.62E-01	1.9120	4.4492
1000000	1.13E-01	8.8793	5.23E-01	1.9126	4.6425

Table 3.2: Test 1 - Data Subset

3.5.2 Test 2

The second performance test is also based on affine transformations, in fact it is about retrieving the transformation matrix given two triangles with corresponding points. Like the previous test, this one only handles the two dimensional case. Basically the problem is to determine the matrix **M** whereby matrix **X** and **X′** are both 3x3 matrices with the points of triangle 1 and triangle 2 respectively:

$$\mathbf{X}' = \mathbf{X}\mathbf{M} \tag{3.1}$$

$$\mathbf{X}^{-1}\mathbf{X}' = \mathbf{M} \tag{3.2}$$

This performance test compares two approaches. The first more general approach solves the problem using BLAS by actually inverting matrix $\mathbf{X}^{-1}$ and then multiply with $\mathbf{X}'$ while the second approach makes use of the symbolic solution (see Appendix A). For more information on both approaches refer to chapter 5.

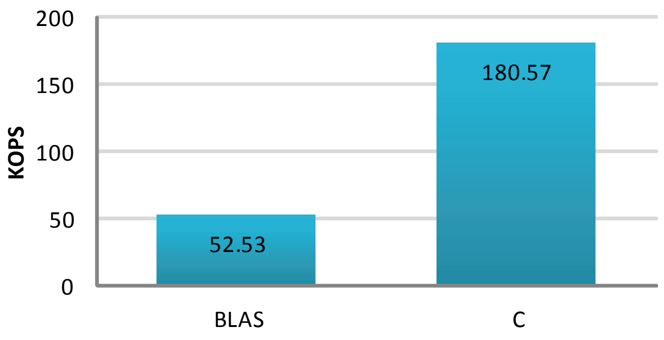


Figure 3.2: Performance Test Retrieving the Transformation Matrix (KOPS = Kilo Operations Per Second, higher is better)

Figure 3.2 depicts the performance of both methods. The more general approach represented by the implementation with BLAS shows a reduced performance compared to the symbolic solved method implemented in C. I believe this is because the overhead of using SIMD for matrix multiplications of 3x3 matrices is too big. Whereas the implementation in C has much less workload and overhead since it uses the already given symbolic solution. Therefore, it makes sense to use BLAS only in connection with large matrices.

Chapter 4: Shape Adjustment

This chapter is about manual segmentation of an image. In my specific case I consider the segmentation of faces in portraits. That is, a face in an image should be described by a certain amount of landmarks so that we can divide a face into regions of interest such as eyes, nose, mouth and so on. In addition we want to have a one-to-one correspondence of points between two shapes to be able to relate them. As already discussed in section 2.3, I only pursue manual ways of image annotating. There are several ways of how we can find those points manually in an image.

The most simple way is to just manually annotate every single shape point on the image. To maintain one-to-one correspondence of the points the user has to mark each point in a given order. This can be done for example using an interface with a mouse clicking on the image or using the touch screen to place markers with a finger. Although this method would provide excellent results assuming that it is used by a thorough person it is very cumbersome since there can easily be more than 40 points to mark in a certain order.

A second possibility is to provide static segments representing eyes, mouth and so on which can be placed on the image by drag and drop. In addition to just placing them, the interface could provide a way to scale and rotate those segments to achieve higher accuracy. This method is less cumbersome than annotating by hand, but lacks accuracy.

The way I pursued was to provide a face shape which again can be modeled using drag and drop. That is, the shape is initially placed on the image and a user should be able to adjust the shape so that the shapes vertices match the face. In this process a user, however, should be guided to maintain simplicity. This method provides the most intuitive and least cumbersome way but needs to be based on some kind of guiding model which leads us to point distribution models.

4.1 Point Distribution Model

Cootes and Taylor [13] introduced the point distribution model as a way to statistically study shapes with applications in computer vision. It is a model that is built from a set of shapes and represents the mean geometry as well as the variation of the geometry gained from this set of shapes. Based on this model, new shapes can be created which are similar to the shapes in the training set.

A shape can be represented by a set of points in arbitrarily many dimensions. Since I restrict the application to only two dimensions, I use a vector $\mathbf{x}$ of length $2n$ whereby n denotes the number of vertices (landmarks):

$$\mathbf{x} = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)^T \tag{4.1}$$

4.1.1 Align a set of shapes

In order to retrieve some statistical information about the shape of a face we need a whole set of shapes. Let s be the number of samples and $\mathbf{x}_i$ be the i'th shape. Before the variation can be modeled the set needs to be aligned. This is done by transforming all the shapes into a common coordinate frame using translation, uniform scale and rotation such that $|\mathbf{x}_i - \overline{\mathbf{x}}|^2$ is minimized for every shape $\mathbf{x}_i$ whereby $\overline{\mathbf{x}}$ denotes the mean shape.

Suppose we have two shapes $\mathbf{x}$ and $\mathbf{x}'$ and we want to find the optimal similarity transformation T which applied to shape $\mathbf{x}$ minimizes $|T(\mathbf{x}) - \mathbf{x}'|^2$. This optimal similarity transformation includes translation, uniform scaling and rotation:

$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{4.2}$$

If the shapes do not have their center of gravity at the origin then the translation's only goal is to match their center of gravity whereby we get the values a and b as follows:

$$a = \left(\mathbf{x} \bullet \mathbf{x}'\right) / |\mathbf{x}|^2 \tag{4.3}$$

$$b = \left(\sum_{i=1}^{n} (x_i y_i' - y_i x_i')\right) / |\mathbf{x}|^2 \tag{4.4}$$

For more information refer to [13].

To align a whole set of shapes, we can use a simple iterative process: Translate each shape so that its center of gravity is at the origin, then make an estimate of the mean shape and scale the estimate so that $|\mathbf{x} = 1|$ then align all shapes using scale and rotation to the mean shape and finally build a new estimate of the mean shape using all aligned shapes. For further details refer to [11, 13, 27, 28].

4.1.2 Build a model

Given a set of aligned shapes we can think of the data forming a distribution in the $2n$ dimensional space. This distribution can be used to create a statistical model which allows us to create new shapes similar to the ones in our set by adjusting a set of parameters. To reduce the dimensionality we apply a PCA (Principal Component Analysis) computing the main axis of the point cloud in $2n$ dimensions which in return will allow us to approximate training shapes using less than $2n$ parameters.

In a first step we compute the mean shape in our aligned frame:

$$\overline{\mathbf{x}} = \frac{1}{s}\sum_{i=1}^{s}\mathbf{x}_i \tag{4.5}$$

Then we compute the covariance matrix of the shape data:

$$\mathbf{\Sigma} = \frac{1}{s}\sum_{i=1}^{s-1}(\mathbf{x}_i - \overline{\mathbf{x}})(\mathbf{x}_i - \overline{\mathbf{x}})^T \tag{4.6}$$

On the covariance matrix we apply eigenanalysis and sort the eigenvectors in descending order of the eigenvalues. In order to reduce the number of parameters we choose the t first eigenvectors (corresponding to the t biggest eigenvalues) to approximate the training set to a certain accuracy A. To achieve that 98% of the training data can be modeled, we choose A to be 0.98. The value t is determined as follows whereby λ_j is the j'th biggest eigenvalue:

$$\sum_{j=1}^{t}\lambda_j \geq A\sum_{j=1}^{2n}\lambda_j \tag{4.7}$$

4.1.3 Building new shapes

The t eigenvectors are being stacked into the matrix $\mathbf{P} = [\mathbf{p_1} \dots \mathbf{p_t}]$. Every shape in the training set can be approximated as a linear combination by varying the t parameters in vector $\mathbf{b}$ as follows:

$$\mathbf{x}' \approx \overline{\mathbf{x}} + \mathbf{Pb} \tag{4.8}$$

To ensure that only similar shapes can be generated, we set limits for each parameter b_i to be between a certain multiple of the corresponding eigenvalue λ_i. A recommended range would be $-3\sqrt{\lambda_i} \leq b_i \leq 3\sqrt{b_i}$ since this covers 98% of the training set because $\lambda_i = \sigma^2$ which is the square of the standard deviation of the i'th component. However, as we will see later, for manual adjustment it does make sense to use looser bounds.

Restricting the parameters to a hypercube allows the shape to adapt to forms that are looking unnaturally, for example by letting all parameters be $+3\sqrt{\lambda_i}$ at the same time. Another way of setting bounds would be by restricting the parameters to a hyperellipsoid using the Mahalanobis distance[1].

Figure 4.1 shows the shape of a face and its variations on the first 5 modes. The model is trained with the MUCT database described in section 4.2 whereby the shape contains 76 landmarks and the database comprises of 3,755 different shapes. The first mode describes the horizontal head movement and the second mode the vertical head movement. The third mode describes the width of the face and the fourth and fifth modes describe the form and position of eye and mouth.

4.2 Database

Apparently there is the need for a big set of shapes to train such a model. There are several databases available which provide manually annotated portrait images. Most of them provide the image and a file describing the shape of the face with a certain amount of landmarks. This section provides a short overview of some publicly available

[1]Gives the probability that the point $\mathbf{b}$ in t dimensions forms a plausible shape. The Mahalanobis distance from a point $\mathbf{x} = [x_1, \dots, x_N]^T$ to a group with mean $\mu = [\mu_1, \dots, \mu_N]^T$ and covariance matrix $\mathbf{S}$, is defined as follows: $d(\mathbf{x}, \mu) = \sqrt{(\mathbf{x} - \mu)^T \mathbf{S}^{-1} (\mathbf{x} - \mu)}$. In our case the covariance matrix is diagonal (variance of each parameter) and the mean μ is the origin, therefore, it reduces to $d(\mathbf{x}) = \sqrt{\sum_{i=1}^{N} x_i^2 / s_i^2}$ where s_i is the standard deviation. [34]

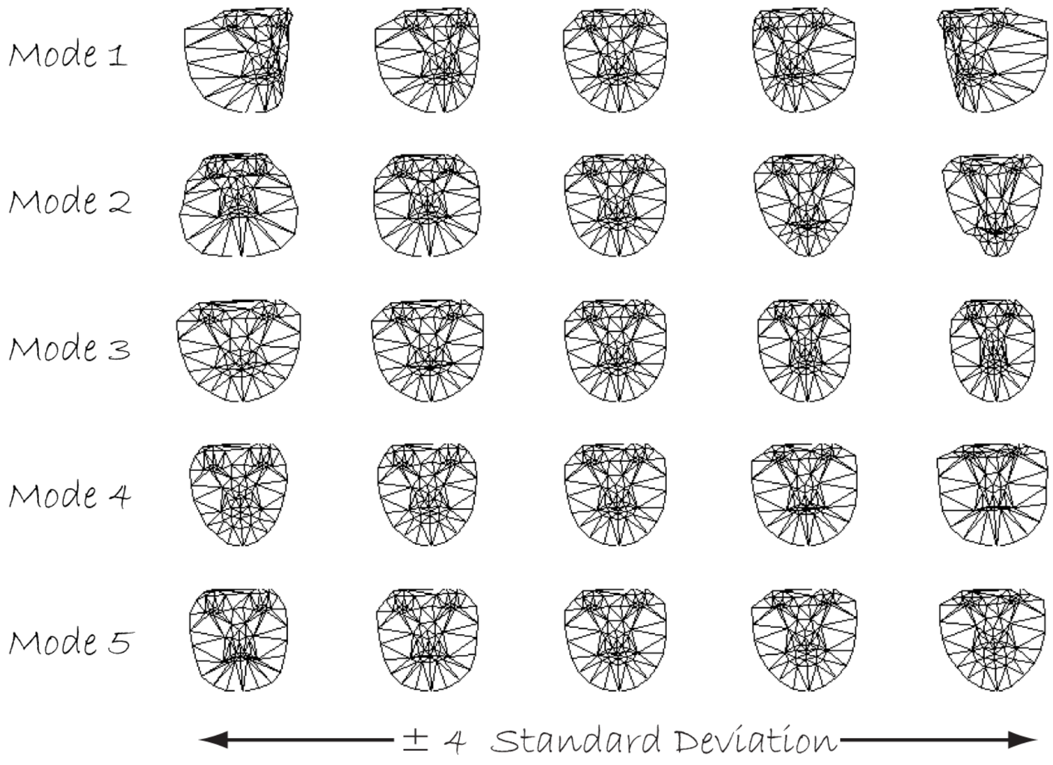


Figure 4.1: Varying the first five modes between -4 and +4 standard deviation

databases and explains the one I chose. Table 4.1 shows a quick summary of databases I considered.

Since I am only interested in the actual shape and not the image, I do not care about the image resolution. Since the goal for this project is to provide the ability to model a large variety of faces the dataset should be as big as possible and should provide a high amount of facial landmarks to ensure an accurate face modeling. Therefore, I chose the MUCT dataset which has the highest amount of markers with the biggest set of faces. Furthermore, it provides a nice csv-sheet including all shapes.

Database	**Images**	**Marker**	**Subjects**	**Image Res.**	**License**	
IMM Face Database	240	58	40	640x480	research	[23]
Put Face Database	9971	30	100	2048x1536	research	[3]
XM2VTS Markup	2360	68	295		free	[29]
MUCT	3755	76	276	640x480	research	[22]
AFLW	25,993	21	-	different	research	[18]

Table 4.1: Different Databases in Comparison [18] - Remark: Put Face Database provides a subset of 2193 images annotated with 194 landmarks

4.3 Shape Modeling

Based on this point distribution model, we want to find a way to model the mean shape such that the shape points match the underlying image. In our case, we want to adjust the mean shape of the face (column in the middle in figure 4.1) to match an underlying portrait.

The easiest way to model a shape would be to use movements such as pushing and dragging parts of the shape into the right position since it is similar to modeling clay, for example. As a user moves his finger on the touch screen, the vertices of the shape inside a certain radius should follow the fingers movements. To not shift the points uncontrolled, we want to constrain the possible movements of each point individually with respect to the shape as a whole. This is the point where the trained point distribution model comes into play. To relate the movement of vertices within image coordinates to the shape parameters we need a way to find the set of parameters for the point distribution model which best matches a shape within image coordinates.

4.3.1 Match Shape to New Points

A shape of a face in an image can be fully described by the shape parameters **b** and a transformation from the model coordinates (aligned frame) into the image coordinates using translation, scale and rotation. Let T be the transformation from the model coordinates into the image coordinates. By applying translation, scaling and rotation then we can describe a shape within image coordinates based on the model as follows:

$$\mathbf{X} = T(\overline{\mathbf{x}} + \mathbf{Pb}) \tag{4.9}$$

Our goal is to find the parameters T and $\mathbf{b}$ which best match the model points $\mathbf{X}$ to a new set of points $\mathbf{Y}$. This can be viewed as minimizing the distance between the points $\mathbf{X}$ and $\mathbf{Y}$ in image coordinates, whereby $\mathbf{x}$ and $\mathbf{y}$ are points in model coordinates:

$$\min(|\mathbf{Y} - \mathbf{X}|^2) = \min(|\mathbf{Y} - T(\overline{\mathbf{x}} + \mathbf{Pb})|^2) \tag{4.10}$$

Cootes et al [13] introduced the following iterative approach to find a solution for equation 4.10:

1. Set all parameters $\mathbf{b}$ to zero
2. Create the model $\mathbf{x} = \overline{\mathbf{x}} + \mathbf{Pb}$
3. Search for the best alignment transformation M to align $\mathbf{x}$ to $\mathbf{Y}$
4. Apply the inverse transformation M^{-1} to the points $\mathbf{Y}$ to transform them into the model coordinate frame: $\mathbf{y} = M^{-1}(\mathbf{Y})$
5. Transform $\mathbf{y}$ into the tangent space to remove non-linearities: $\mathbf{y}' = \mathbf{y}/(\mathbf{y} \bullet \overline{\mathbf{x}})$
6. Determine parameters $\mathbf{b}$ which match best $\mathbf{y}$: $\mathbf{b} = \mathbf{P}^T(\mathbf{y}' - \overline{\mathbf{x}})$
7. If the parameters of the transformation or the shape have changed since the last iteration go back to step 2

4.3.2 Non-rigid Shape

Now that we are able to relate some vertex modifications done by a user in image coordinates back to our model, we can come up with a way to apply the model. Starting with the mean shape a user can push and drag vertices onto the image. After moving one or more vertices of the shape we can search for the model parameters which describe the closest possible shape as described in section 4.3.1. This matched shape, however, is still unconstrained and thus can adopt the shape formed by the user. To constrain the shape modification we apply bounds to the shape parameters $\mathbf{b}$ according to section

4.1.3. This does not only cause the touched vertices to change but rather the whole shape to adjust itself since we generate a new shape based on new parameters which approximate the shape with the moved vertices. By doing this each time one or more vertices are moved, we get a non-rigid shape.

Figure 4.2 depicts this behavior. The shape is modified by shifting, in this example one vertex. By matching the model into this new modified shape we can find the model parameters **b** which best describe this new shape. Applying constraints to the parameters, however, forces the shape into a form that is more similar to the shapes in the training set and that is exactly what we want. Vertices can be moved by a certain degree as long as they meet the constraints of what a form should look like based on the training set. The resulting shape always provides a plausible face shape.

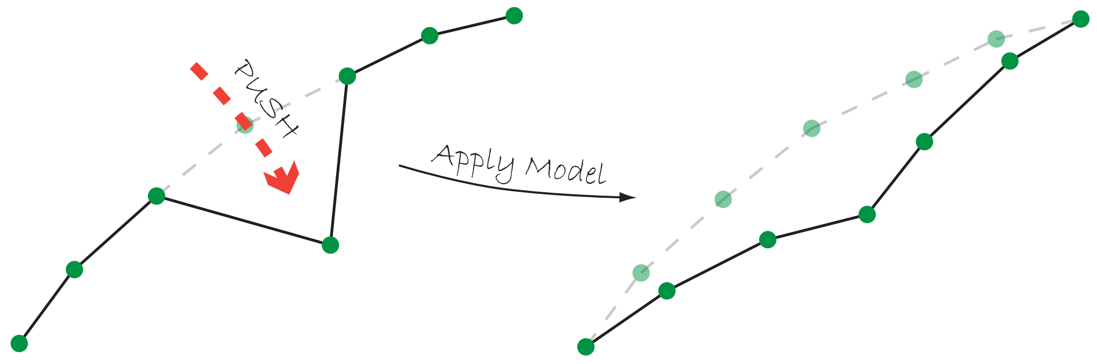


Figure 4.2: Shape Modeling

4.4 Implementation

Now that the basics of point distribution models are clear this section provides some details about my implementation which applies such a model on the iPhone 4S. This section provides a brief explanation of how the model is trained, which data is created from the training and how the shape modeling part is realized.

4.4.1 Model Training and Model Data

Before the model can be used in the adjustment process, we need to train it. To do this I decided to use Matlab and save the model to files so that the adjustment application on the iPhone can load the trained model. This makes sense because the model needs to be trained only once.

The training in Matlab loads a set of shapes, aligns the shapes and retrieves the mean shape and the eigenvalues and eigenvectors of the variation with which 98% of the training set can be described. In addition the training already triangulates the mean shape so that the model is complete. Finally the training creates four different csv-files which describe the full model.

The first file holds the mean shape $\overline{x}$. The second and third file contain the most significant t eigenvectors P and eigenvalues λ respectively. As the last file, the training outputs the triangle indices TRI. Once the model is trained the iPhone application can load all files to make use of the model.

$$\overline{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \quad P = \begin{bmatrix} a_{(1,1)} & \cdots & a_{(1,t)} \\ \vdots & \ddots & \vdots \\ a_{(n,1)} & \cdots & a_{(n,t)} \\ a_{(n+1,1)} & \cdots & a_{(n+1,t)} \\ \vdots & \ddots & \vdots \\ a_{(2n,1)} & \cdots & a_{(2n,t)} \end{bmatrix} \quad \lambda = \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_t \end{bmatrix} \quad TRI = \begin{bmatrix} i_{11} & i_{12} & i_{13} \\ \vdots & \vdots & \vdots \\ i_{m1} & i_{m2} & i_{m3} \end{bmatrix}$$

4.4.2 Touch Shape Modeling

As a test application I created a program using the iOS development environment which demonstrates the shape modeling process on a real device. As already mentioned, the provided frameworks for iPhone application development rely heavily on the model-view-control design pattern. The central idea is to assign each class one of the types and define a communication pattern between classes. View objects provide what the user can see, they are responsible to draw on the screen and receive feedback from the user. The model object's task is to hold application data. In between the view and the model is the controller decoupling the model from the view. The controller mainly controls the model and view that are associated to it.

With respect to this design pattern I divided the shape adjustment application and the point distribution implementation into different classes of the specific type: model, view and controller. Figure 4.3 shows the class diagram I came up with. At the top of the diagram is the blue colored model layer. The model layer consists of two main classes: the PDMShape and the PDMShapeModel. The PDMShape serves as a class to hold a shape and manage the shape data. The class provides methods to modify and access the shape data. The PDMShapeModel class manages a trained point distribution model. The class holds a mean shape, a triangulation of it, eigenvectors and eigenvalues. The model layer is connected to the green colored control layer by the PDMModelingController class.

The PDMModelingController controls the process of modeling a shape to match an image. The life cycle of this class begins with a new adjustment task of one shape and ends when the adjustment is done. Furthermore, on the control layer is one or more view controller. This is a special controller which is connected to a view to control it. I omit the view layer in the class diagram since there can be different views with different purposes of visualizing the model.

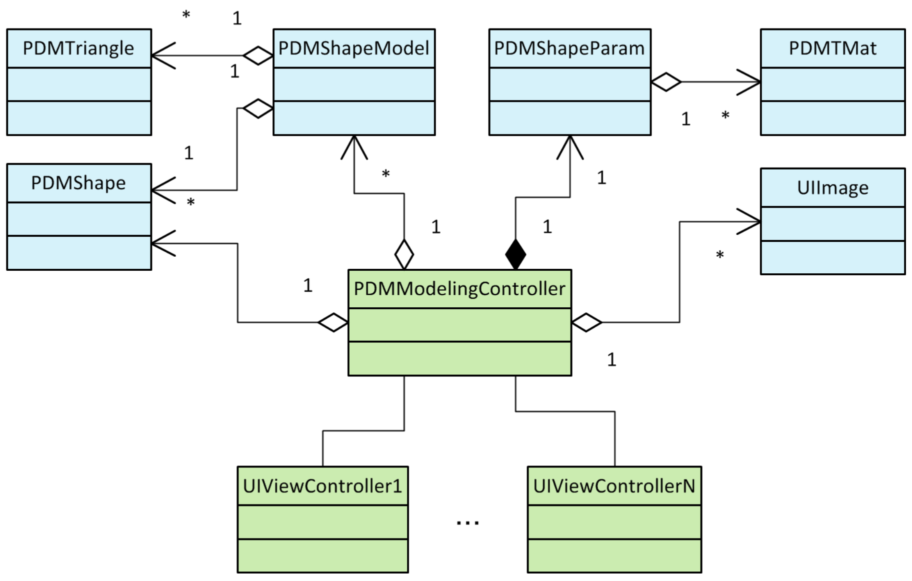


Figure 4.3: Class Diagram

4.4.3 Application

Based on this, I developed an application which demonstrates the proposed shape modeling process. The test application I came up with provides three different interfaces to modify the shape and is dedicated to test the adjustment process. The first interface can be used solely to modify the transformation from model coordinates to image coordinates. That is a user can translate scale and rotate the shape. The second interface allows us to modify the first 8 shape modes using a slider. This demonstrates what the different modes represent. The third and main interface is the actual shape modeling interface shown if figure 4.4.

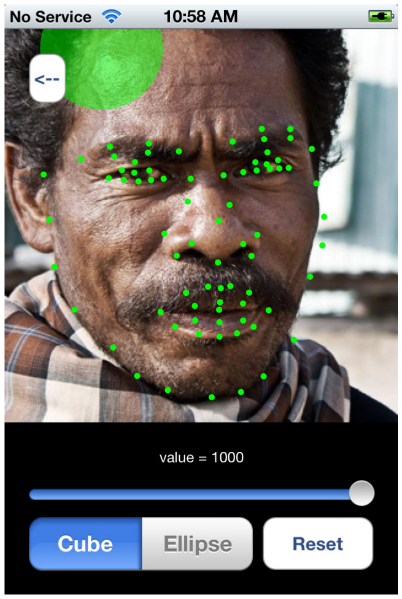


Figure 4.4: Shape Modeling Interface, Portrait by NeilsPhotography (CC) Some rights reserved. Source: `http://www.flickr.com/photos/neilspicys/`

The shape modeling interface allows the user to modify the shape according to the method described in section 4.3 by using the multi-touch screen of the iPhone. The application will recognize up to 5 different fingers to model the shape. For test purposes there is the possibility to switch between parameter restriction by hypercube or by hyperellipsoid and its actual bounding value.

Chapter 5: Piecewise Affine Warp

The second part of the project addresses the problem of warping an annotated image into another shape. Figure 5.1 depicts this warping process. On the left side there is a portrait annotated with landmarks (green dots) and on the right side are the same landmarks but transformed with random displacement of each landmark. By applying a warping function to the image, every landmark in the left image has to match its corresponding landmark in the right image.

However, it is not possible to achieve this matching using a global transformation. With a fixed number of coefficients we can only approximate the target shape to a certain degree. Using global transformation would result in unnaturally looking distortions. To avoid these problems I use a piecewise transformation which basically transforms different parts of the image independently.

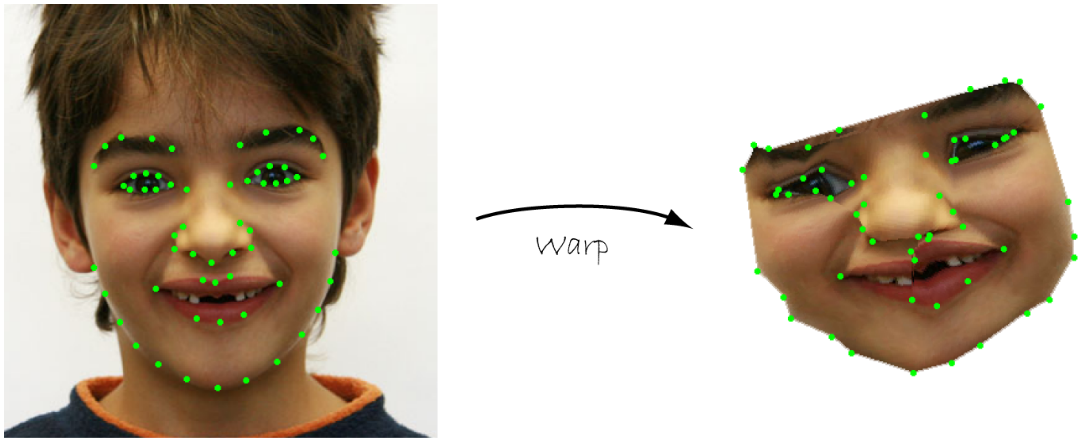


Figure 5.1: Piecewise Image Warping [Portrait by Kate Burgener (C)]

5.1 Background

For my application it is sufficient to assume that the image can be mapped locally linear. This is a reasonable assumption since there are enough landmarks available to define patches to map. The main approach is to use the given landmarks to divide the image into regions and then determine the transformation from one region to its corresponding region. Figure 5.2 depicts this process. On the left hand side the landmarks are used to divide the image into regions using Delaunay triangulation. On the right hand side the landmarks are again non-linearly distorted whereby the triangulation is the same as on the left side. The face image on the right hand side, however, is the result of a piecewise affine transformation of each triangle into its corresponding triangle.

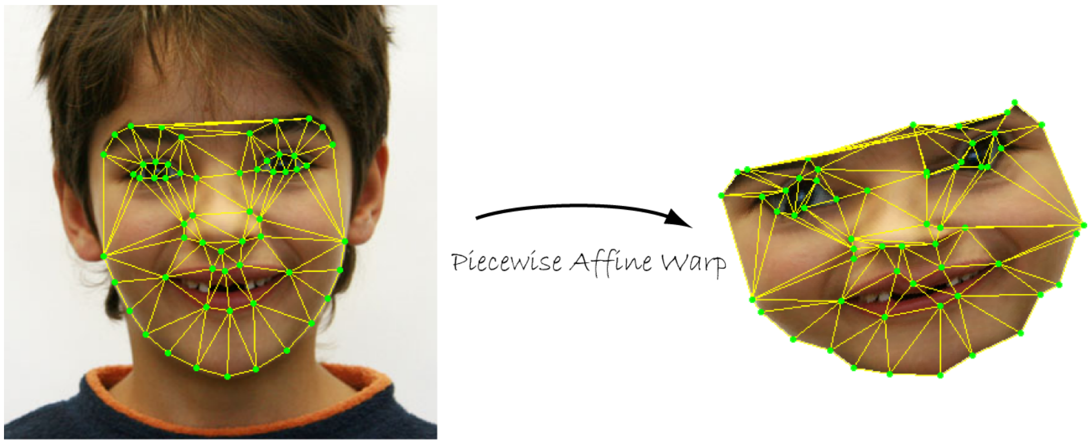


Figure 5.2: Piecewise Image Warping of Triangles [Portrait by Kate Burgener (C)]

By definition affine transformations allow one to apply translation, scale, rotation and shearing whereby the transformation preserves parallelism. Let **M** denote the affine transformation matrix in the planar case using homogeneous coordinates. We can write a transformation of a point using homogeneous coordinates as follows:

$$\underbrace{\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}}_{\mathbf{x'}} = \underbrace{\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix}}_{\mathbf{M}} \underbrace{\begin{bmatrix} x \\ y \\ w \end{bmatrix}}_{\mathbf{x}} \tag{5.1}$$

Note that if we transform a point, that has been bumped up into homogeneous coordinates (adding a third dimension), using the transformation matrix **M** we will receive a vector in three dimensions. Thus the resulting point can be found on a line in three dimensions. This can be written using an unknown multiplier c whereby $\mathbf{x}$ is the input point and $\mathbf{x}'$ the transformed point:

$$\mathbf{M}\mathbf{x} = c\mathbf{x}' \tag{5.2}$$

If we, however, constrain the homogeneous input points to $w = 1$, that is the z-coordinate is always 1, we can simplify equation 5.1 by setting the values $a_7 = 0$, $a_8 = 0$ and $a_9 = 1$. This simplifies matrix $\mathbf{M}$ to only six unknown values. Thus it is enough to have three pairs of different points to fully determine the transformation matrix $\mathbf{M}$ resulting in the following equation whereby a_1 to a_6 are unknown:

$$\underbrace{\begin{bmatrix} x'_1 & y'_1 & 1 \\ x'_2 & y'_2 & 1 \\ x'_3 & y'_3 & 1 \end{bmatrix}}_{\mathbf{X}'} = \underbrace{\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} a_1 & a_4 & 0 \\ a_2 & a_5 & 0 \\ a_3 & a_6 & 1 \end{bmatrix}}_{\mathbf{M}} \tag{5.3}$$

The general solution to this problem is a left matrix multiplication in the case of a unique solution (three pairs of points). In the overdetermined case when there are more than three points, we can use a least squares method to find the best fit. I do not consider this case since I am going to work solely with triangles. Using a left multiplication, we get for $\mathbf{M}$:

$$\mathbf{M} = \mathbf{X}^{-1}\mathbf{X}' \tag{5.4}$$

Using this method, however, results in a big computational effort since there is the need to compute the inverse of a matrix which is computationally intense. Additionally we only have to consider affine transformation matrices which are a special case of matrices yielding simpler symbolic solutions to the problem. Therefore, it does make more sense to solve the system of equations and translate it to code. See Appendix A for the symbolic solution to the linear system of equations.

Another way of finding the transformation matrix is the use of barycentric coordinates. Figure 5.3 shows the purpose of using barycentric coordinates. Corresponding points $\mathbf{x}$ and $\mathbf{x}'$ inside the triangles can be expressed using the parameters p, q and r whereby $p + q + r = 1$:

$$\mathbf{x} = \mathbf{x_1} + p(\mathbf{x_2} - \mathbf{x_1}) + q(\mathbf{x_3} - \mathbf{x_1}) \tag{5.5}$$

$$= r\mathbf{x_1} + p\mathbf{x_2} + q\mathbf{x_3} \tag{5.6}$$

$$\mathbf{y} = \mathbf{x'_1} + p(\mathbf{x'_2} - \mathbf{x'_1}) + q(\mathbf{x'_3} - \mathbf{x'_1}) \tag{5.7}$$

$$= r\mathbf{x'_1} + p\mathbf{x'_2} + q\mathbf{x'_3} \tag{5.8}$$

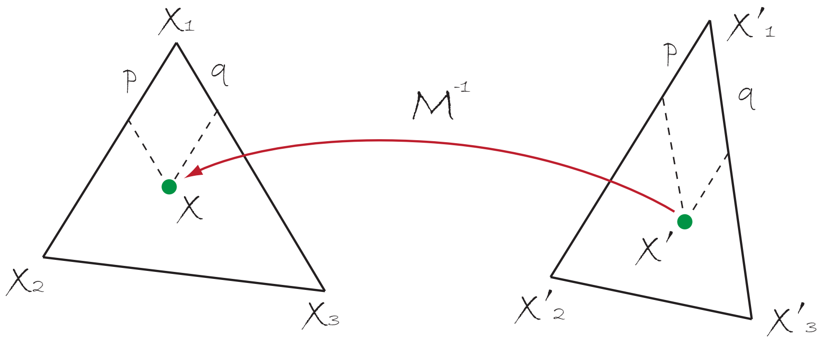


Figure 5.3: Mapping using Barycentric Coordinates

Let $\mathbf{X}$ and $\mathbf{X'}$ again be matrices holding the points of triangle 1 and triangle 2 respectively and let $\mathbf{u} = [r, p, q]$. Thus we can write one equation for both triangles: $\mathbf{uX} = \mathbf{x}$ and $\mathbf{uX'} = \mathbf{x'}$ whereby the third value (z-coordinate) of $\mathbf{x}$ and $\mathbf{x'}$ is 1. Substituting the first equation into the second yields $\mathbf{x'} = \mathbf{xX}^{-1}\mathbf{X'} = \mathbf{xM}$ which is the same result as equation 5.4.

5.2 Implementation

In this section I provide some details about two different implementations which I realized and tested in this project. One implementation is fully based on calculations on the CPU, the second one uses the GPU to accelerate the processing. I have built both of these versions from scratch using the Xcode environment for the latest iOS 5 based on Objective-C and C++. In section 6.2 I compare both of these methods concerning performance.

Both implementations provide exactly the same interface. Each one consists of one class with one method that can be called to warp an image. This method `warpImage` takes as arguments one image, two shapes and a triangulation of the points and returns the warped image.

Listing 5.1 explains one way of performing a piecewise affine warp given an image, some points in the image, the corresponding set of points to which the image should be warped and a triangulation:

Listing 5.1: Piecewise Affine Warp Algorithm

```
Go through all pairs of triangles
    Determine the inverse of the transformation matrix
    Find all pixels in the destination triangle
    Go through all found pixels
        Determine the color using the inverse transformation
```

5.2.1 CPU Algorithm

This first method relies fully on the CPU and handles image data as big arrays. It strongly follows the list presented above, thus I will briefly explain each point of this list.

To determine the inverse transformation, we can compute the transformation from the destination triangle to the source triangle to avoid computing the inverse. The implementation makes use of the symbolic solution for the transformation matrix since this provides a much better performance (see section 3.5).

In a second step, we need to find all pixels in the target image which should be assigned to a color. This can be done by either checking all pixels in the whole image if they lie inside of the triangle or by doing rasterization. The latter one obviously will provide a much better performance because not every pixel has to be checked individually. Figure 5.4 depicts the triangle rasterization process I use. First, the three point connecting lines are rasterized using Bresenham's line algorithm (refer to [20] for more detail) and stored in an array. Then the algorithm determines the pixels in the triangle row by row beginning at the top of the triangle and stores all pixel indices in an array.

Finally we want to assign each pixel a color. Given the inverse of the transformation matrix, every point inside the triangle can be mapped to a location in the source image where we get the color from. Figure 5.5 shows how this mapping works. For every point

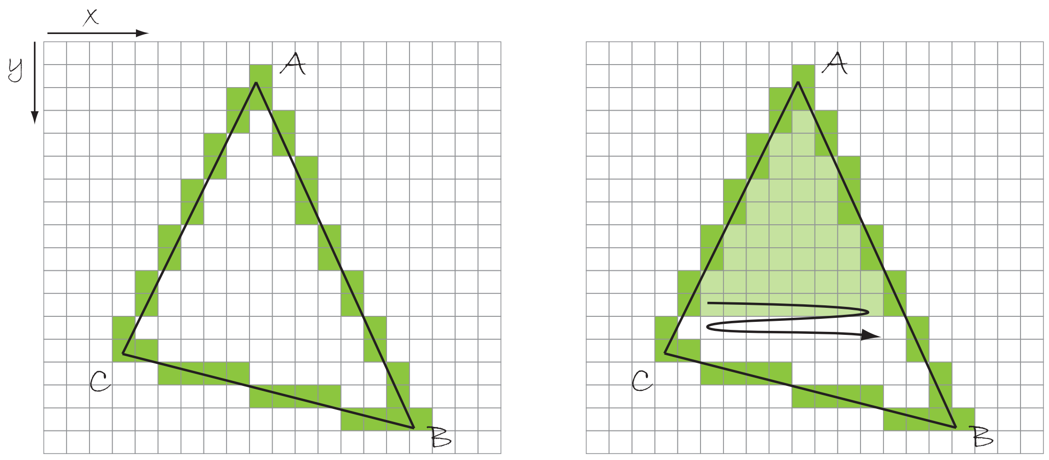


Figure 5.4: Triangle Rasterization

$\mathbf{x}'$ inside the triangle on the right side we can determine the location $\mathbf{x}$ to get the color from in the image on the left side: $\mathbf{x} = \mathbf{M}^{-1}\mathbf{x}'$. In my implementation, I use the nearest neighbor method to determine the pixel's color since this is the fastest method. To get smoother results, one can also determine the pixel's value by considering a patch of pixels around the source location and interpolate them.

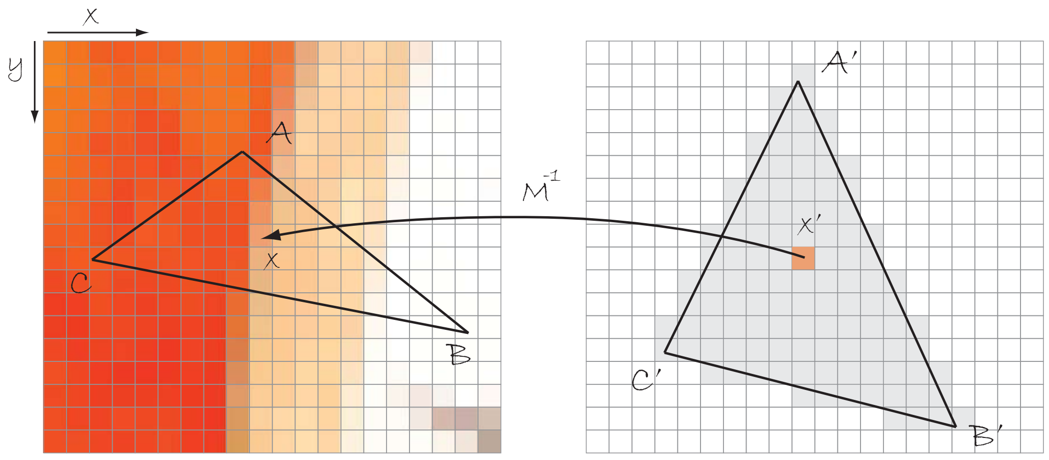


Figure 5.5: Determine Pixel Color

5.2.2 GPU Algorithm

This second method makes use of the GPU to accelerate and simplify the algorithm explained in the CPU implementation. With respect to the steps explained above, the use of the GPU apparently makes sense. The GPU is a master in matrix calculations, rasterization and interpolating values and exactly these strengths should be utilized in this implementation. But first of all here is a quick review of what the graphics card does and how OpenGL works in the context of this implementation.

The GPU is a graphics processing hardware dedicated to building images. Its main purpose is to process vertex data and create a 2D pixel image to show on a screen. Vertex data is created and sent down to the graphics card with attributes such as color information, texture coordinates and so on where they enter the OpenGL ES graphics pipeline.

The first stage where OpenGL ES 2.0 lets us modify data on the graphics card is the vertex shader. Each vertex is processed individually by some shader code and passed further to the clipping stage where primitives are clipped and culled. After that the rasterizer creates potential pixels from the primitives, interpolates parameters and passes them to the fragment shader. In the fragment shader, each fragment can be individually modified by user defined code and then the pixels are passed to some further stages and finally written into the framebuffer which can be read back to the program or used to display the image.

With the graphics pipeline in mind, we can come up with a way to do a piecewise affine transformation given an image, two shapes and a triangulation. First of all we let the source image be a texture which is passed to the graphics card. The warped image then will be the image finally written to the framebuffer. Our goal is to determine the corresponding triangle and its texture coordinates for every triangle in the destination shape similar to the approach in figure 5.5.

Since the graphics card works with vertex data, we can use the vertices of the destination shape to define the triangles. All vertices are passed to the vertex shader where they are scaled into normalized coordinates between -1 and 1. The next step is to assign a color to each pixel inside a triangle in the fragment shader. That is, each fragment needs to have its own texture coordinates to look up the color value.

To do this we need to provide some information about the corresponding triangles.

Therefore, every vertex from the source shape gets attached to its corresponding destination vertex and sent to the vertex shader. The vertex shader in return translates those source shape vertices into texture coordinates between 0 and 1. Since the rasterizer can interpolate values, we want to utilize this, and hence the vertex shader passes those coordinates varying to the fragment shader. The fragment shader finally receives the right texture coordinate for every fragment and determines the fragment's color with a single texture query.

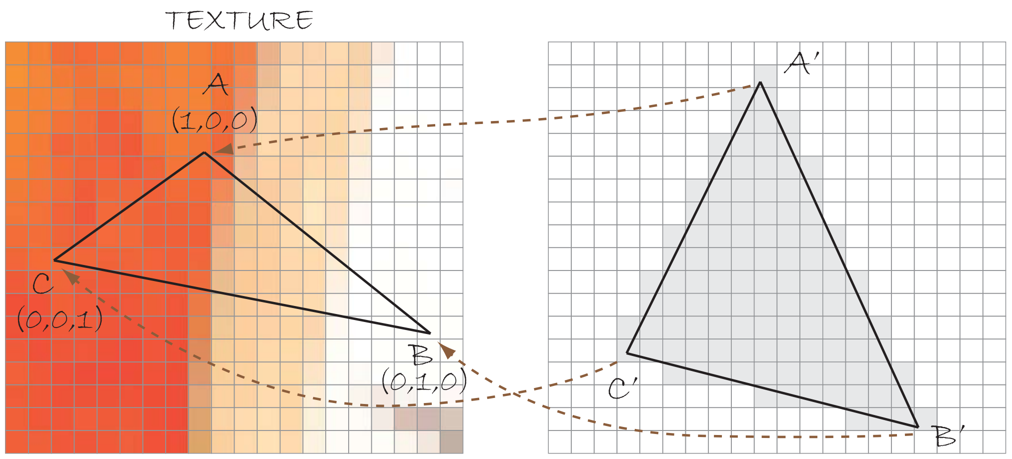


Figure 5.6: Pairs of points and interpolation of texture coordinates

Figure 5.6 depicts how those point pairs are built and how the texture coordinates are interpolated. On the right side is the warped image and on the left side the texture which is the source image. Every shape vertex is paired with the corresponding vertex illustrated with the arrows. Both vertices are translated into their coordinate system whereby the rasterizer interpolates the texture coordinates using barycentric coordinates depicted in figure 5.6 with the coordinate values below the vertices in the texture.

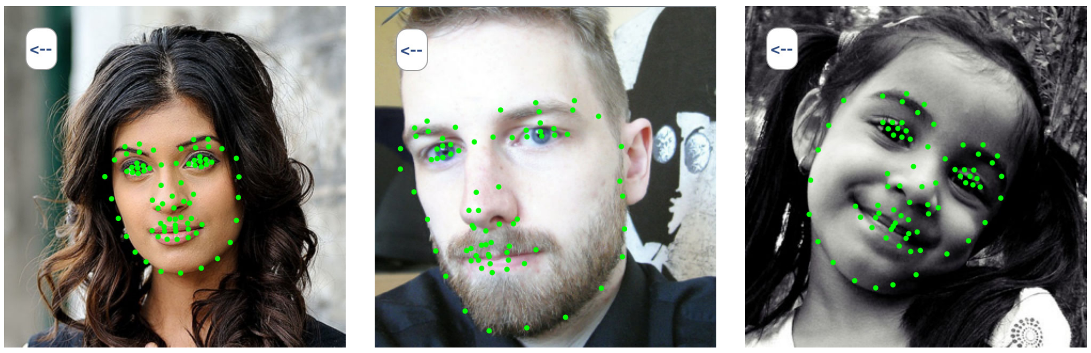
After writing the result into the framebuffer the program can read the image back for further processing.

Chapter 6: Results

This chapter provides a review of the achievements of this project. First I review the shape modeling process, discuss the user-friendliness and provide some recommendation for future work. Afterwards I discuss the warping implementation and its performance.

6.1 Shape Modeling

Based on the shape modeling interface, I tested the modeling process. The goal of this process was to adjust a mean shape to match the underlying face. The mean shape was initialized at the center of the image and scaled relatively to the image size. The user then had to modify the shape using touch gestures on the screen. The conclusions are based on my personal experience working with images from Flickr under a Creative Commons license. In order to make more detailed conclusions about this whole modeling process a user study could be necessary.



(a) Frontal face example. (b) Example of suboptimal face (c) Rotated face example

Figure 6.1: Aligned Examples [Left: by kaybee07, `http://www.flickr.com/people/kurtbudiarto/`; Middle: by hobvias sudoneighm, `http://www.flickr.com/people/striatic/`; Right: by M Yashna, `http://www.flickr.com/people/yashna13/`; (CC) Some rights reserved.]

Figure 6.1 shows three test examples. The images represent faces with different head poses and facial expressions. On the left side 6.1(a) is an example of a successful shape adjustment on an underlying image with a frontal face. Image 6.1(b) in contrast shows a case where the adjustment failed due to some distortions of the image and not fully frontal representation of the face. The image 6.1(c) on the right side shows another successful alignment on an inclined face.

6.1.1 Conclusion

In comparison to other manual ways of retrieving landmarks, I experienced the proposed method as user-friendly and fast, provided that the image is suitable. Due to its similarity to forming something, I think a user should be ultimately familiar with the way of how to adjust the shape. However, it seems that the method works best for frontal faces with neutral expression. In other cases it can become a challenging and frustrating experience since the shape sometimes behaves stubbornly from a user's view. From my own experience, frontal faces can be modeled in reasonable time with sufficient accuracy. Faces that are not frontal require more time and patience and sometimes make a satisfying result impossible.

While testing, I recognized two main points that need to be reconsidered in the future. First, the application seems to be too detailed for general use. The many points and the small size of the iPhone's screen make a fast adjustment difficult. Reducing the number of landmarks and refining the model with an automated algorithm would help this issue. In practice this could mean only having one point for the center of each eye, one for the tip of the nose and only a few points describing mouth and chin. Increasing the screen size by using an iPad instead of an iPhone or adjusting the sizes of the shifting touch are other ideas. Besides that, there is the problem that when using a touch screen, the finger covers the part of the shape that is currently being modified. Many other iPhone applications avoid this problem by providing a detail view showing the image underneath the touch position. Therefore one possible solution to this would be to provide a detail view right above the touch position.

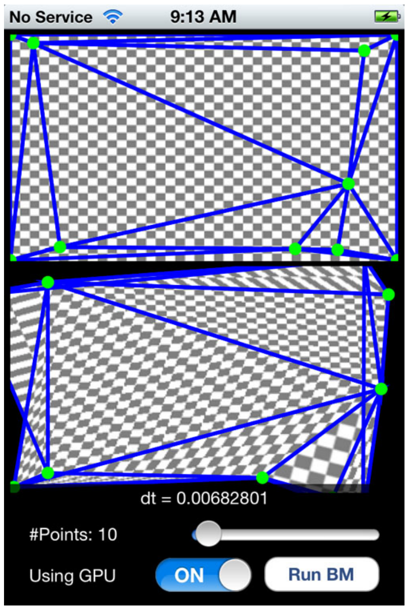
The second problem is the strong restriction of the model. While the restriction seems to be helpful for the coarse adjustment it can be disruptive during fine adjustments. A possible improvement could be to stack various point distribution models whereby each model describes a group of points (eyes, mouth, etc.) that have a stronger relationship. This solution, however, requires a lot of work and investigation.

In summary the method seems to work well for its purpose. It provides a good basis for additional improvements.

6.2 Piecewise Affine Warp

The second goal of this project was to provide a way to perform piecewise affine transformation with enhanced performance and scalability using mobile computation power. In this section, I provide a performance comparison between my piecewise affine warp implementation with enhanced performance using the GPU and the one which solely relies on the CPU. But first here is a short discussion about how to come up with a fair performance test.

Both implementations have their strengths and weaknesses. The CPU implementation obviously has to consider every triangle individually whereby for every triangle pair the algorithm determines the transformation matrix and the triangles pixels. Thus the computing time depends linearly on the number of pixels the shape covers as well as the number of triangles. On the other hand the GPU implementation is more concerned about the texture size and the number of vertex pairs since both of them need to be passed to the graphics card. In addition to that, the iPhone does not support non-power of two textures and thus the image needs to be padded which results in more work for the GPU and also increases the used GPU memory.



dt = 0.00682801

#Points: 10

Using GPU ON

Figure 6.2: PAW Testprogram Screenshot

With respect to those characteristics, I came up with two different performance tests. The first test measured the performance against the image resolution, namely the number of pixels that were transformed. The goal of this test was to verify that the CPU requires linearly more computation time with increasing number of pixels and that the GPU will perform better overall. The goal of the second test was to compare how well the CPU performs against the GPU if only subparts of the image need to be transformed. This gave some idea of how big the overhead is using the GPU.

All tests were done on the same iPhone 4S with the serial number C39HC0YDDT9Y. Every test was performed 10 times independently and averaged. Listing 6.1 shows how one test was made. The interface to each warping implementation was one function call with the image, source and destination shape and a specific triangulation. As a source image I used an automatically created checkerboard with a certain resolution. Then the test randomly distributed a certain amount of points on the image additionally to a point in each image corner, triangulated them using Delaunay triangulation and then displaced them randomly to create the destination shape. Based on this data both algorithms were called and the execution time stored. After 10 of those tests the average running time as well as the average number of triangles and covered area was stored. Figure 6.2 shows the visualization of one single warping run based on artificial data as an application on the iPhone.

Listing 6.1: Independent Test

```
Repeat 10 times
    Create image of desired size
    Randomly place points on the image which serve as the shape1
    Based on the shape1 create a shape2 by randomly displace vertices
    Triangulate shape1

    warpedImage = warpImageCPU:image :shape1 :shape2 :triangles
    warpedImage = warpImageGPU:image :shape1 :shape2 :triangles

    Store #triangles , covered area , exec. time CPU and GPU

Save #points , avg. #triangles , avg. covered area and avg. exec. times
```

Note that all the following GPU tests were made with an already initialized OpenGL context which provided a texture and framebuffer that was big enough for the image to warp. This can be done without loss of generality since the OpenGL context, as well as, the buffers can be initialized precautionary at the start of the program with buffers that are big enough for image sizes up to the camera resolution for example.

6.2.1 Performance Against Image Size

This first test measured the performance of the warping algorithm at different resolutions and number of shape vertices. The number of transformed pixels was around 90% of the whole image. As already mentioned each data point is the average of 10 independent tests whereby one independent test was executed with artificially created data. Both implementations were called in one test using the same data.

Figure 6.3 shows a comparison of the performance of both implementations in Mega Pixel Transformations Per Second (MPTPS). On the x-axis is the number of pixels that were actually transformed, using a logarithmic scale, up to the camera resolution of the iPhone which is 8MP. At each resolution there are three different values. For each implementation there is one test with 10, 50 and 100 vertices used to define the shape.

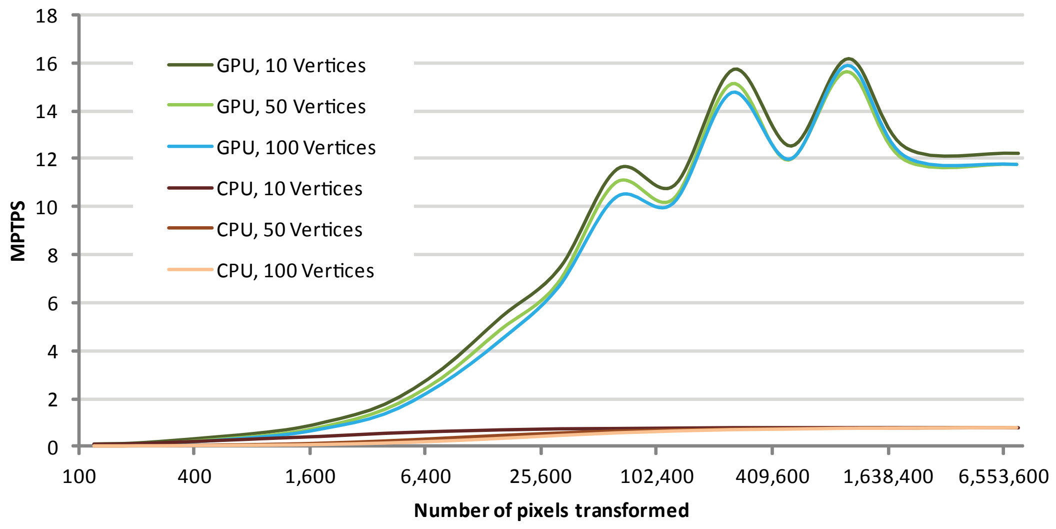


Figure 6.3: Image points transformed per second (MTPTS = Mega Pixels Transformations Per Second)

As figure 6.3 depicts the GPU performed much better compared to the CPU with increasing number of pixels to transform. The graph shows that the CPU required approximately linear time with the number of pixels to transform which is what I was expecting. The GPU was able to compensate its overhead already at smallest resolutions. The number of vertices in this test case had only little to almost no influence on the performance of both implementations. The ripples in the GPU implementation might be due to the fact that images are padded to a size of power of two and thus when using adverse resolutions the performance dropped a little bit. For example the second peak at 262,144 pixels, corresponding to a resolution of 512x512, required no padding. The following valley at 524,176 pixels on the other side, corresponds to a resolution of 724x724, needed to be padded to 1024x1024.

Figure 6.4 depicts the speedup gained by using the GPU. Again on the x-axis is the number of transformed pixels and on the y-axis is the speedup. As this graph depicts using the GPU provided a big speedup gain which went up to 20 times faster. Apparently the GPU implementation was also able to perform much better when there were more vertices.

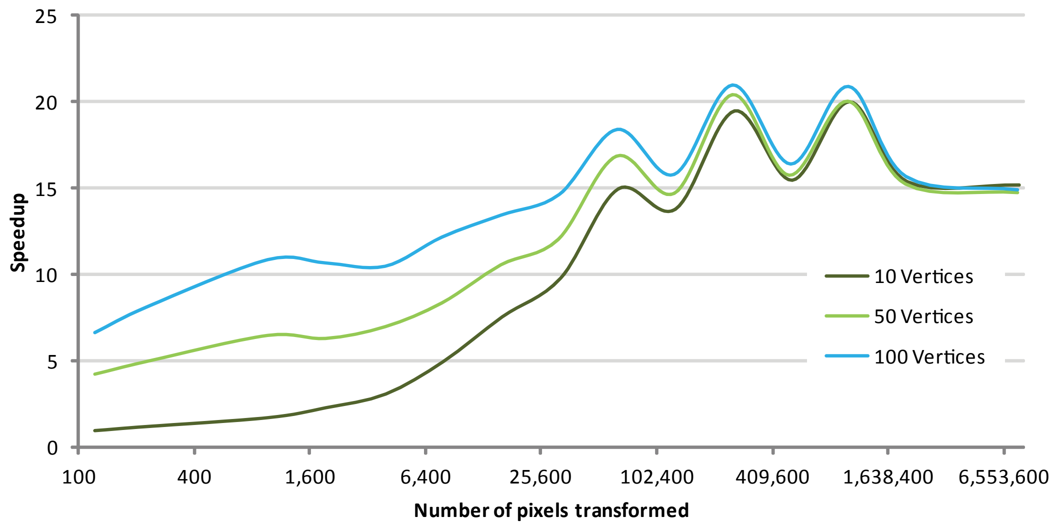


Figure 6.4: Speedup using the GPU

To get a feeling of how real time capable those implementations are, figure 6.5 depicts the performance in frames per second. To still have at least 25 frames per second we can use images of size around 724x724 which equates to almost the iPhone's 4S screen resolution at 960x640 pixels using the GPU implementation. On the other hand we can use images of size 181x181 with 10 vertices to get a frame rate of 23 frames per second using the CPU.

To maintain a fair competition, the frame rate comparison was based on the same data like the two graphs before which means that a frame calculation includes always a new image, new shapes and a new triangulation. The GPU performance could possibly be improved by just updating the vertices and omit sending down the texture data another time.

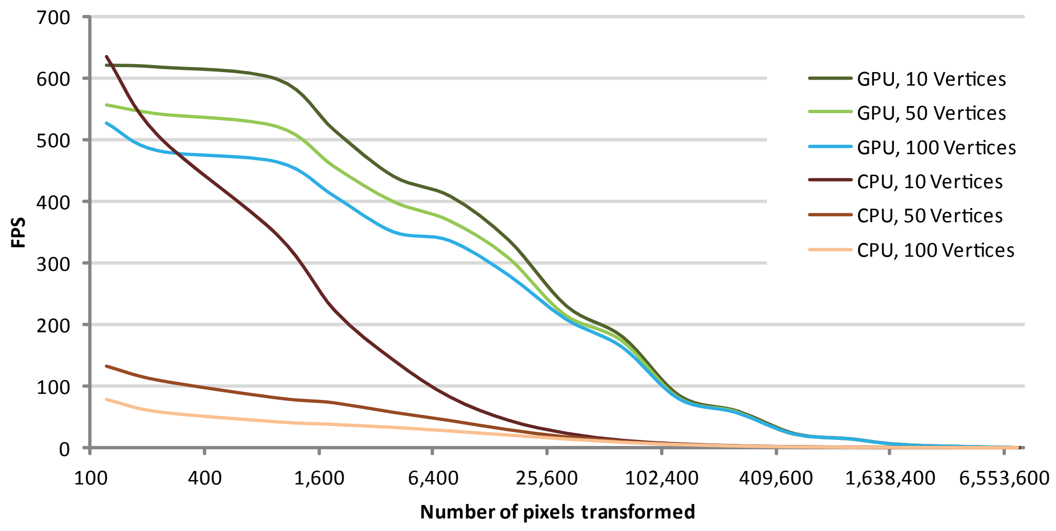
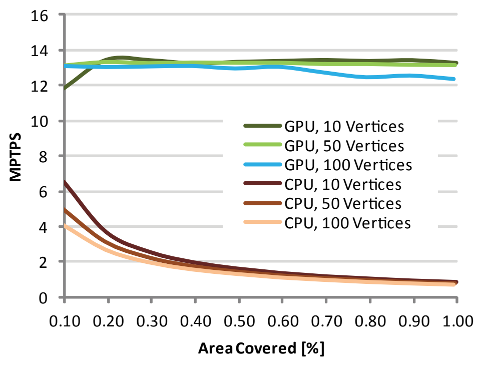


Figure 6.5: Frames per second

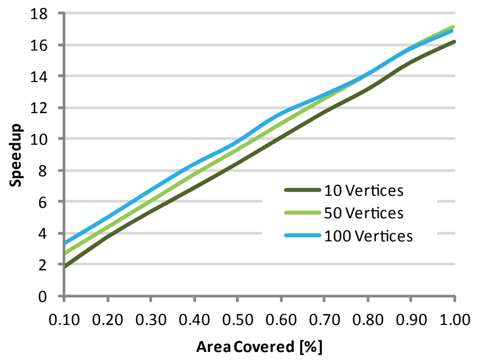
6.2.2 Performance Against Image Coverage

This second test provided some details about the overhead of using the GPU by only considering smaller subparts of the image to transform. The basic idea was that the GPU implementation always used the whole input image while the CPU implementation only considered the destination triangles. There were two different tests with different image resolutions whereby in each test the shape coverage went from 10% to 100%.

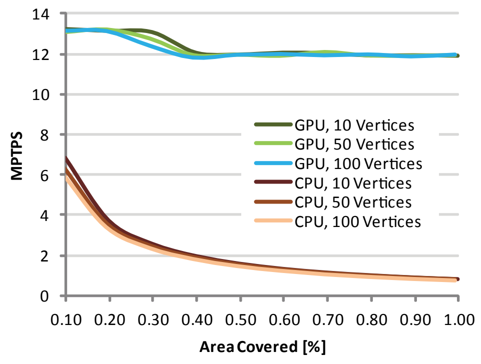
Figure 6.6 depicts the two different tests. The top row (fig. 6.6(a) and 6.6(b)) shows the performance in Mega Pixel Transformations Per Second and the speedup of the GPU



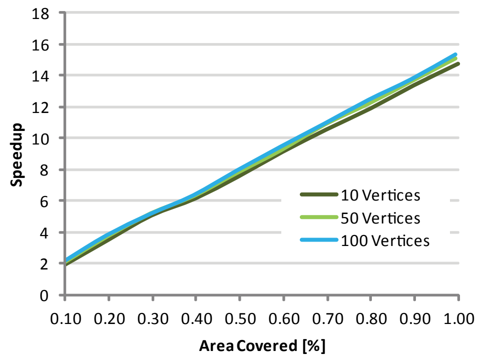
(a) Mega Pixel Transformations Per Second



(b) Speedup using GPU



(c) Mega Pixel Transformations Per Second



(d) Speedup using GPU

Figure 6.6: Performance with only subparts of the image transformed (top row: 640x480, bottom row: 1600x1200)

implementation based on an image with 640x480 pixels. The bottom row (fig. 6.6(c) and 6.6(d)) shows the same information but with an image resolution of 1600x1200.

From figure 6.6, it can be observed that the behavior was almost the same with different image resolutions. The performance graphs on the left side show that the GPU implementation required almost always the same computation time regardless of what area was being transformed. The CPU implementation, however, was faster the smaller the area that needed to be transformed. The speedup graphs on the right side support my expectation by showing that the smaller the area that needed to be transformed the bigger the overhead of the GPU implementation compared to the CPU.

6.2.3 Conclusion

The presented tests compare both implementations in a fair manner. The GPU implementation stands out as a clear winner in all cases. Although there is an overhead in using OpenGL, it is worth using it for really small images of around 150 pixels in total. However, the implementation on the CPU is eligible as well for cases where there are only few and small triangles. If we compare the complexity and work to write both implementations, the GPU version is a lot more straightforward making it the more preferable method overall.

For future work, I recommend cleaning out the code and including better support OpenGL extensions. That is, for example, if the GPU supports non-power of two texture sizes, the code should make use of this advantage. Furthermore, it would be nice to provide a library for public use.

		Avg. #Triangles			Avg. Area Coverage			Avg. Pixels Transformed			Speedup GPU vs. CPU		
Resolution	Pixel	10	50	100	10	50	100	10	50	100	10	50	100
12x12	144	13.1	70.5	122.0	0.84	0.84	0.84	121	121	121	0.98	4.22	6.60
16x16	256	13.2	80.5	150.7	0.88	0.88	0.88	225	225	225	1.22	4.96	8.09
32x32	1024	13.3	88.8	179.6	0.94	0.94	0.94	961	961	961	1.73	6.47	10.84
45x45	2025	13.8	88.9	184.8	0.95	0.95	0.94	1,932	1,919	1,907	2.31	6.29	10.66
64x64	4096	13.7	91.1	187.7	0.97	0.96	0.96	3,963	3,930	3,916	3.10	6.98	10.45
90x90	8100	13.6	89.6	186.1	0.98	0.96	0.95	7,899	7,753	7,727	4.98	8.36	12.15
128x128	16384	13.1	90.1	188.9	0.98	0.96	0.96	15,976	15,698	15,770	7.56	10.57	13.43
181x181	32761	13.6	90.3	187.8	0.98	0.95	0.96	32,266	31,208	31,591	9.81	12.04	14.62
256x256	65536	13.0	89.7	188.0	0.98	0.96	0.97	64,147	62,842	63,282	14.96	16.88	18.40
362x362	131044	13.0	88.6	187.6	0.97	0.96	0.96	127,755	125,527	125,396	13.80	14.72	15.79
512x512	262144	13.3	90.0	186.5	0.99	0.96	0.96	259,077	252,287	252,366	19.48	20.46	20.98
724x724	524176	13.3	88.9	187.4	0.99	0.95	0.97	516,942	498,072	507,455	15.47	15.77	16.39
1024x1024	1048576	13.1	88.3	186.4	0.98	0.96	0.97	1,028,129	1,003,592	1,012,190	20.01	20.08	20.90
1448x1448	2096704	13.1	89.2	186.1	0.98	0.96	0.96	2,051,835	2,003,401	2,017,658	15.39	15.27	15.67
3264 x 2448	7990272	13.0	88.8	186.3	0.99	0.96	0.96	7,870,418	7,691,436	7,691,436	15.19	14.76	14.89

		FPS GPU			MPTPS GPU			FPS CPU			MPTPS CPU		
Resolution	Pixel	GPU 10	GPU 50	GPU 100	10	50	100	CPU 10	CPU 50	CPU 100	10	50	100
12x12	144	620.27	556.70	526.40	0.08	0.07	0.06	635.61	131.96	79.70	0.08	0.02	0.01
16x16	256	617.40	542.33	482.00	0.14	0.12	0.11	508.00	109.43	59.56	0.11	0.02	0.01
32x32	1024	597.80	521.57	463.80	0.57	0.50	0.45	345.99	80.59	42.78	0.33	0.08	0.04
45x45	2025	514.59	456.58	409.02	0.99	0.88	0.78	223.10	72.59	38.38	0.43	0.14	0.07
64x64	4096	440.32	400.13	350.93	1.74	1.57	1.37	141.96	57.32	33.57	0.56	0.23	0.13
90x90	8100	408.18	369.52	336.86	3.22	2.86	2.60	81.93	44.18	27.73	0.65	0.34	0.21
128x128	16384	337.40	310.53	281.63	5.39	4.87	4.44	44.61	29.38	20.97	0.71	0.46	0.33
181x181	32761	230.65	217.82	210.42	7.44	6.80	6.65	23.51	18.08	14.39	0.76	0.56	0.45
256x256	65536	180.34	175.04	164.27	11.57	11.00	10.40	12.05	10.37	8.93	0.77	0.65	0.56
362x362	131044	85.28	82.21	80.86	10.90	10.32	10.14	6.18	5.59	5.12	0.79	0.70	0.64
512x512	262144	60.71	59.88	58.39	15.73	15.11	14.74	3.12	2.93	2.78	0.81	0.74	0.70
724x724	524176	24.23	23.97	23.59	12.52	11.94	11.97	1.57	1.52	1.44	0.81	0.76	0.73
1024x1024	1048576	15.74	15.56	15.67	16.18	15.61	15.87	0.79	0.77	0.75	0.81	0.78	0.76
1448x1448	2096704	6.06	5.96	5.99	12.44	11.94	12.08	0.39	0.39	0.38	0.81	0.78	0.77
3264 x 2448	7990272	1.55	1.53	1.53	12.22	11.76	11.74	0.10	0.10	0.10	0.80	0.80	0.79

Figure 6.7: Performance Against Image Size - Data

	Avg. Area Cov.			Time GPU [s]			MPTPS GPU			Time CPU [s]			MPTPS GPU			Speedup		
	10	50	100	10	50	100	10	50	100	10	50	100	10	50	100	10	50	100
640x480	0.10	0.10	0.10	0.026	0.023	0.023	11.80	13.08	13.12	0.047	0.062	0.075	6.52	4.99	4.10	1.81	2.62	3.20
	0.20	0.19	0.19	0.023	0.023	0.024	13.46	13.31	13.06	0.084	0.098	0.114	3.66	3.13	2.69	3.68	4.25	4.85
	0.30	0.29	0.29	0.023	0.023	0.023	13.43	13.24	13.09	0.122	0.137	0.154	2.52	2.24	2.00	5.33	5.92	6.56
	0.40	0.39	0.39	0.023	0.023	0.023	13.24	13.28	13.12	0.159	0.176	0.193	1.94	1.75	1.59	6.84	7.61	8.26
	0.49	0.49	0.49	0.023	0.023	0.024	13.33	13.26	12.97	0.193	0.213	0.229	1.59	1.44	1.34	8.36	9.20	9.67
	0.60	0.59	0.59	0.023	0.023	0.023	13.38	13.27	13.07	0.230	0.251	0.270	1.34	1.23	1.14	10.00	10.82	11.48
	0.70	0.69	0.69	0.023	0.023	0.024	13.44	13.19	12.74	0.266	0.290	0.306	1.15	1.06	1.00	11.64	12.44	12.70
	0.80	0.79	0.79	0.023	0.023	0.025	13.38	13.18	12.44	0.301	0.326	0.346	1.02	0.94	0.89	13.12	13.98	14.01
	0.90	0.89	0.89	0.023	0.023	0.024	13.43	13.14	12.54	0.339	0.367	0.384	0.91	0.84	0.80	14.83	15.69	15.66
	1.00	0.99	0.99	0.023	0.023	0.025	13.27	13.12	12.33	0.374	0.400	0.421	0.82	0.77	0.73	16.15	17.08	16.88

	Avg. Area Cov.			Time GPU [s]			MPTPS GPU			Time CPU [s]			MPTPS GPU			Speedup		
	10	50	100	10	50	100	10	50	100	10	50	100	10	50	100	10	50	100
1600x1200	0.10	0.10	0.10	0.145	0.147	0.146	13.20	13.04	13.14	0.281	0.302	0.321	6.83	6.35	5.99	1.93	2.05	2.19
	0.20	0.19	0.19	0.146	0.146	0.146	13.12	13.17	13.14	0.515	0.541	0.561	3.73	3.55	3.42	3.52	3.72	3.84
	0.30	0.29	0.29	0.147	0.151	0.155	13.06	12.73	12.39	0.748	0.781	0.806	2.57	2.46	2.38	5.09	5.18	5.20
	0.40	0.39	0.39	0.159	0.161	0.163	12.07	11.92	11.79	0.981	1.013	1.042	1.96	1.90	1.84	6.17	6.29	6.40
	0.50	0.49	0.49	0.160	0.161	0.161	11.98	11.94	11.93	1.211	1.252	1.284	1.59	1.53	1.50	7.56	7.78	7.98
	0.60	0.59	0.59	0.159	0.162	0.160	12.07	11.88	11.97	1.450	1.486	1.523	1.32	1.29	1.26	9.12	9.19	9.50
	0.70	0.69	0.69	0.159	0.159	0.161	12.05	12.07	11.91	1.678	1.728	1.769	1.14	1.11	1.09	10.53	10.86	10.97
	0.80	0.79	0.79	0.161	0.161	0.161	11.93	11.90	11.95	1.912	1.964	2.004	1.00	0.98	0.96	11.88	12.17	12.47
	0.90	0.89	0.89	0.161	0.162	0.162	11.94	11.89	11.85	2.145	2.199	2.237	0.90	0.87	0.86	13.34	13.61	13.80
	1.00	0.99	0.99	0.161	0.161	0.161	11.92	11.92	11.94	2.369	2.426	2.471	0.81	0.79	0.78	14.71	15.06	15.37

Figure 6.8: Performance Against Image Coverage

Chapter 7: Conclusion

To summarize, I reached the goals set for both parts of this project. I did this by providing a way to manually adjust a non-rigid shape to an underlying image and introducing a fast method to warp an image piecewise which is based on GPU computing. The next two sections discuss the achievements for both parts of this project in more detail.

7.1 Shape Modeling

The introduced shape modeling algorithm allows manual alignment of a shape to an underlying image. In order to make conclusions about the user-friendliness a user study is necessary. Although this method can be applied to arbitrary shapes, in this project I successfully applied it to faces. The underlying point distribution model helps to get a smooth adjustment process using the fingers on the touch-screen of the iPhone which is similar to modeling clay.

However, one drawback of it is that the model restricts the modeling process to shapes that are similar to the ones in the training set which might be unwanted by a potential user. This leads to the next problem namely that there is the need for a big database with training examples that present a big variety of face shapes. Another critical part is the adjustment of the shape parameter bounds since they have a big influence on how hard the shape can be modeled.

In summary, the shape modeling part provides an appropriate way to manually retrieve information about a face in an image. One advantage of this method is, that it could be useful to create training data for automated algorithms by applying loose bounds to the parameters. Although I used it in this project for faces, it could as well be used for other objects with well defined shapes that do not have a large variety.

7.2 Piecewise Affine Warp

As the performance test showed, the introduced method to warp an image using the GPU is a big gain compared to an implementation using the CPU. Considering the simplicity of the required code and how well the hardware can be exploited, the proposed method is useful in almost every case of piecewise affine transformation. As a side effect of my implementation, which is packed into a class that does everything, a user does not even have to know about the graphics pipeline. One drawback of this implementation is that every time the image size changes, the OpenGL framebuffer needs to be recreated in a time consuming step. To avoid this, the buffers could be initialized to a big size at the program startup which causes wasted memory.

To summarize, the use of the GPU brings a striking speedup gain facilitating an application in real time using the 960x640 screen resolution of the iPhone 4S.

Bibliography

[1] 9to5mac, "iphone 4s image stabilization makes all the difference, clip shows." [Online]. Available: http://9to5mac.com/2011/10/14/iphone-4s-image-stabilization-makes-all-the-difference-clip-shows/

[2] ARM, "Neon." [Online]. Available: http://www.arm.com/products/processors/technologies/neon.php

[3] C. Biometrics, "Put face database description." [Online]. Available: https://biometrics.cie.put.poznan.pl/index.php?option=com_content&view=article&id=4&Itemid=2&lang=en

[4] V. Blanz, K. Scherbaum, T. Vetter, and H.-P. Seidel, "Exchanging Faces in Images," in *Proceedings of EG 2004*, vol. 23, no. 3, Sep. 2004, pp. 669–676.

[5] V. Blanz and T. Vetter, "A morphable model for the synthesis of 3d faces," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 187–194. [Online]. Available: http://dx.doi.org/10.1145/311535.311556

[6] K. Burgener, "Dossier programmierer – chic-o-mat an interactive system for diversity research, version 2," 2009.

[7] D. Burri and W. Jenni, "Forschungsteil chic-o-mat v2," 2011.

[8] C.-W. Chen and C.-C. Wang, "3d active appearance model for aligning faces in 2d images," *2008 IEEERSJ International Conference on Intelligent Robots and Systems*, pp. 3133–3139, 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4650788

[9] W. Coder, "Introduction to neon on iphone." [Online]. Available: http://wanderingcoder.net/2010/06/02/intro-neon/

[10] T. F. Cootes and C. J. Taylor, "Constrained active appearance models," in *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Springer, 2001, pp. 484–498.

[11] T. F. Cootes, C. J. Taylor, D. H. Cooper, and J. Graham, "Active shape models - their training and application," *Comput. Vis. Image Underst.*, vol. 61, no. 1, pp. 38–59, Jan. 1995. [Online]. Available: http://dx.doi.org/10.1006/cviu.1995.1004

[12] T. Cootes and C.J.Taylor, "Active shape models - smart snakes," in *In British Machine Vision Conference.* Springer-Verlag, 1992, pp. 266–275.

[13] T. Cootes and C. Taylor, "Statistical models of appearance for computer vision," 2000.

[14] T. F. Cootes, G. J. Edwards, and C. J. Taylor, "Active appearance models," in *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Springer, 1998, pp. 484–498.

[15] A. Developer, "Taking advantage of the accelerate framework." [Online]. Available: https://developer.apple.com/performance/accelerateframework.html

[16] R. Gross, I. Matthews, and S. Baker, "Constructing and fitting active appearance models with occlusion," *2004 Conference on Computer Vision and Pattern Recognition Workshop*, vol. 00, no. C, pp. 72–72, 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1384865

[17] ——, "Active appearance models with occlusion," *Image and Vision Computing*, vol. 24, no. 6, pp. 593–604, 2006.

[18] M. Koestinger, P. Wohlhart, P. M. Roth, and H. Bischof, "Annotated facial landmarks in the wild: A large-scale, real-world database for facial landmark localization," in *First IEEE International Workshop on Benchmarking Facial Image Analysis Technologies*, 2011.

[19] X. Liu, "Discriminative face alignment," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 11, pp. 1941–1954, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2008.238

[20] K. McDonald and A. Castro, "Bresenham's line algorithm." [Online]. Available: http://en.wikipedia.org/wiki/Bresenham\%27s_line_algorithm

[21] ——, "Dynamic face substitution." [Online]. Available: http://flowingdata.com/2012/01/04/face-substitution/

[22] S. Milborrow, J. Morkel, and F. Nicolls, "The MUCT Landmarked Face Database," *Pattern Recognition Association of South Africa*, 2010, http://www.milbo.org/muct.

[23] M. M. Nordstrøm, M. Larsen, J. Sierakowski, and M. B. Stegmann, "The IMM face database - an annotated dataset of 240 face images," Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Tech. Rep., may 2004. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?3160

[24] I. T. PowerVR, "Sgx series5 graphics ip core family." [Online]. Available: http://en.wikipedia.org/wiki/PowerVR

[25] J. M. Saragih, S. Lucey, and J. F. Cohn, "Deformable model fitting by regularized landmark mean-shift," *Int. J. Comput. Vision*, vol. 91, no. 2, pp. 200–215, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1007/s11263-010-0380-4

[26] M. B. Stegmann, "Active appearance models: Theory, extensions and cases," Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, p. 262, aug 2000. [Online]. Available: http://www.imm.dtu.dk/~aam/main/

[27] M. B. Stegmann, R. Fisker, and B. K. Ersbøll, "On properties of active shape models," Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Tech. Rep., 2000. [Online]. Available: http://www.imm.dtu.dk/~aam/downloads/asmprops/index.html

[28] M. B. Stegmann and D. D. Gomez, "A brief introduction to statistical shape analysis," Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, p. 15, mar 2002, images, annotations and data reports are placed in the enclosed zip-file. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?403

[29] UFACE and FGNET, "Xm2vts 68pt markup." [Online]. Available: http://www-prima.inrialpes.fr/FGnet/data/07-XM2VTS/xm2vts_markup.html

[30] Wikipedia, "Apple a5." [Online]. Available: http://en.wikipedia.org/wiki/Apple_A5

[31] ——, "Arm architecture." [Online]. Available: http://en.wikipedia.org/wiki/ARM_NEON#Advanced_SIMD_.28NEON.29

[32] ——, "Arm cortex-a9 mpcore." [Online]. Available: http://en.wikipedia.org/wiki/ARM_Cortex-A9_MPCore

[33] ——, "Cocoa (api)." [Online]. Available: http://en.wikipedia.org/wiki/Cocoa_%28API%29

[34] ——, "Mahalanobis distance." [Online]. Available: http://en.wikipedia.org/wiki/Mahalanobis_distance

[35] ——, “Objective-c.” [Online]. Available: http://en.wikipedia.org/wiki/Objective-C

[36] ——, “Powervr.” [Online]. Available: http://en.wikipedia.org/wiki/PowerVR

APPENDICES

Appendix A: Symbolic Solution to Affine Transformation Matrix

We can write the affine transformation for three corresponding points $\mathbf{X}$ and $\mathbf{X'}$ as follows:

$$\underbrace{\begin{bmatrix} y_{11} & y_{12} & 1 \\ y_{21} & y_{22} & 1 \\ y_{31} & y_{33} & 1 \end{bmatrix}}_{\mathbf{X'}} = \underbrace{\begin{bmatrix} x_{11} & x_{12} & 1 \\ x_{21} & x_{22} & 1 \\ x_{31} & x_{32} & 1 \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} a_1 & a_4 & 0 \\ a_2 & a_5 & 0 \\ a_3 & a_6 & 1 \end{bmatrix}}_{\mathbf{M}} \tag{A.1}$$

Rewriting equation A.1 to bring into the form $\mathbf{Ax} = \mathbf{b}$ yields the following linear equation system:

$$\begin{bmatrix} x_{11} & x_{12} & 1 & 0 & 0 & 0 \\ x_{21} & x_{22} & 1 & 0 & 0 & 0 \\ x_{31} & x_{32} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_{11} & x_{12} & 1 \\ 0 & 0 & 0 & x_{21} & x_{22} & 1 \\ 0 & 0 & 0 & x_{31} & x_{32} & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} = \begin{bmatrix} y_{11} \\ y_{21} \\ y_{31} \\ y_{12} \\ y_{22} \\ y_{32} \end{bmatrix} \tag{A.2}$$

This system in return can be solved using e.g. Matlab's symbolic toolbox which gives us the following solution:

$$\begin{aligned} a_1 =& -k(x_{12} * y_{21} - x_{22} * y_{11} - x_{12} * y_{31} + x_{32} * y_{11} + x_{22} * y_{31} - x_{32} * y_{21}) \\ a_2 =& k(x_{11} * y_{21} - x_{21} * y_{11} - x_{11} * y_{31} + x_{31} * y_{11} + x_{21} * y_{31} - x_{31} * y_{21}) \\ a_3 =& k(x_{11} * x_{22} * y_{31} - x_{11} * x_{32} * y_{21} - x_{12} * x_{21} * y_{31} + \\ & x_{12} * x_{31} * y_{21} + x_{21} * x_{32} * y_{11} - x_{22} * x_{31} * y_{11}) \end{aligned}$$

$$\begin{aligned}
a_4 =& - k(x_{12} * y_{22} - x_{22} * y_{12} - x_{12} * y_{32} + x_{32} * y_{12} + x_{22} * y_{32} - x_{32} * y_{22}) \\
a_5 =& k(x_{11} * y_{22} - x_{21} * y_{12} - x_{11} * y_{32} + x_{31} * y_{12} + x_{21} * y_{32} - x_{31} * y_{22}) \\
a_6 =& k(x_{11} * x_{22} * y_{32} - x_{11} * x_{32} * y_{22} - x_{12} * x_{21} * y_{32} + \\
& x_{12} * x_{31} * y_{22} + x_{21} * x_{32} * y_{12} - x_{22} * x_{31} * y_{12})
\end{aligned}$$

and

$$k = 1/(x_{11} * x_{22} - x_{12} * x_{21} - x_{11} * x_{32} + x_{12} * x_{31} + x_{21} * x_{32} - x_{22} * x_{31})$$

Listing A.1: Matlab code to solve linear equations

```
syms x_11 x_12 x_21 x_22 x_31 x_32;
syms y_11 y_12 y_21 y_22 y_31 y_32;

syms a_1 a_2 a_3 a_4 a_5 a_6;

eq1 = 'a_1*x_11 + a_2*x_12 + a_3 = y_11';
eq2 = 'a_1*x_21 + a_2*x_22 + a_3 = y_21';
eq3 = 'a_1*x_31 + a_2*x_32 + a_3 = y_31';
eq4 = 'a_4*x_11 + a_5*x_12 + a_6 = y_12';
eq5 = 'a_4*x_21 + a_5*x_22 + a_6 = y_22';
eq6 = 'a_4*x_31 + a_5*x_32 + a_6 = y_32';

S = solve(eq1,eq2,eq3,eq4,eq5,eq6, 'a_1','a_2','a_3','a_4','a_5','a_6');
```