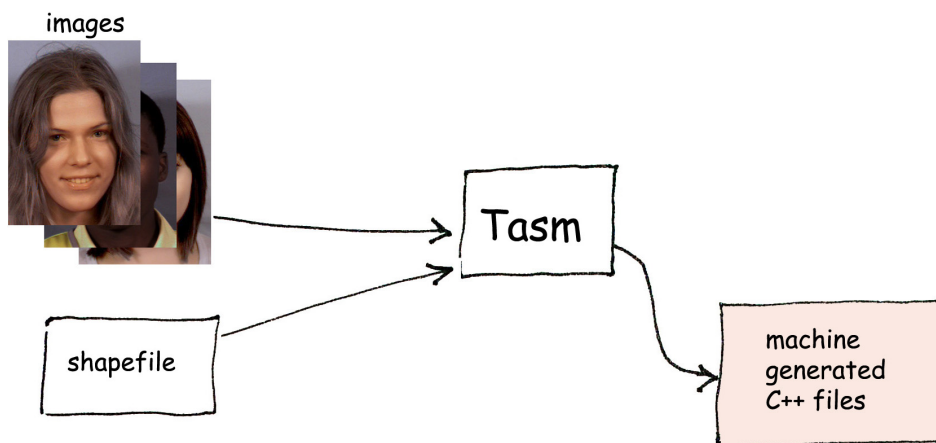


---

# Building Stasm 4 Models

---



Stephen Milborrow

January 7, 2014

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Prerequisites . . . . .	4
1.2	Non-face objects . . . . .	4
1.3	A note on programming philosophy . . . . .	5
<b>2</b>	<b>Overview of model building</b>	<b>6</b>
2.1	Output of Tasm . . . . .	6
2.2	The model name yaw00 . . . . .	7
2.3	Inputs to Tasm . . . . .	8
2.4	Training descriptors and .desc files . . . . .	8
<b>3</b>	<b>Building a model: an example</b>	<b>9</b>
3.1	Step 1: Build Stasm and Tasm . . . . .	9
3.2	Step 2: Prepare the images . . . . .	10
3.3	Step 3: Prepare the shapefile . . . . .	10
3.4	Step 4: Prepare the landmark table . . . . .	10
3.5	Step 5: Prepare the face detector . . . . .	11
3.6	Step 6: Primary edits to the source code . . . . .	11
3.7	Step 7: Possible additional edits to the source code . . . . .	11
3.7.1	facedet.cpp . . . . .	11
3.7.2	initasm.cpp . . . . .	12
3.7.3	tasmconf.h . . . . .	12
3.7.4	tasmconf.cpp . . . . .	12
3.7.5	startshape.cpp . . . . .	12
3.7.6	shape17.cpp . . . . .	13
3.7.7	convshape.cpp . . . . .	14
3.7.8	Scattered constants . . . . .	15
3.8	Step 8: Rebuild Stasm and Tasm . . . . .	16
3.9	Step 9: Do a trial run of Tasm with a small set of images . . . . .	16
3.9.1	Tasm prints . . . . .	17
3.9.2	Face detector records in the shapefile . . . . .	17
3.10	Step 10: Rebuild Stasm with new model files . . . . .	18
3.11	Step 11: Test the new version of Stasm . . . . .	19
3.12	Step 12: Run Tasm with the complete set of shapes . . . . .	19
3.13	Step 13: Tune the model . . . . .	20

---

<b>4</b>	<b>Non-faces: the hand example</b>	<b>22</b>
4.1	Step 1: Build Stasm and Tasm . . . . .	22
4.2	Step 2: Prepare the images . . . . .	23
4.3	Step 3: Prepare the shapefile . . . . .	23
4.4	Step 4: Prepare the landmark table . . . . .	23
4.5	Step 5: Prepare the “face” detector . . . . .	23
4.6	Step 6: Primary edits to the source code . . . . .	24
4.7	Step 7: Additional edits to the source code . . . . .	24
4.7.1	facedet.cpp . . . . .	24
4.7.2	initasm.cpp . . . . .	25
4.7.3	tasmconf.h . . . . .	25
4.8	Step 8: Rebuild Stasm and Tasm . . . . .	26
4.9	Step 9: Do a trial run of Tasm . . . . .	26
4.10	Step 10: Rebuild Stasm with new model files . . . . .	27
4.11	Step 11: Test the new version of Stasm . . . . .	27
4.12	Step 12: Run Tasm with the complete set of shapes . . . . .	28
4.13	Step 13: Tune the model . . . . .	28
<b>5</b>	<b>Shapefiles</b>	<b>29</b>
5.1	The <code>Directories</code> string . . . . .	29
5.2	Face detector records in the shapefile . . . . .	29
5.2.1	How to add facedets to a shapefile . . . . .	31
5.2.2	What triggers generation of <code>facedet.shape</code> ? . . . . .	32
5.2.3	Eyes and mouths in <code>facedet.shape</code> . . . . .	32
<b>6</b>	<b>The landmark table</b>	<b>33</b>
<b>7</b>	<b>Images generated by Tasm</b>	<b>34</b>
7.1	The landmark images . . . . .	34
7.2	The <code>meanshape</code> image . . . . .	35
7.3	The <code>shapeN</code> images . . . . .	35
<b>8</b>	<b>Marki</b>	<b>36</b>
8.1	Using Marki . . . . .	36
8.2	Selecting the current landmark . . . . .	37
8.3	The mouse . . . . .	38
8.4	Zooming . . . . .	38
8.5	Backups . . . . .	38
8.6	Preparing an initial shapefile for Marki . . . . .	39
<b>9</b>	<b>Missing points and three-quarter faces</b>	<b>40</b>
9.1	Example three-quarter model . . . . .	40
9.2	The <code>imputed.shape</code> file . . . . .	41
9.3	Issues with the above example . . . . .	43
9.4	The imputation algorithm . . . . .	43
<b>10</b>	<b>Mirroring shapes</b>	<b>45</b>
10.1	The <code>shapemirror</code> utility . . . . .	45

---

<b>11 FAQs and error messages</b>	<b>47</b>
11.1 NELEMS(LANDMARK_INFO_TAB) 77 does not match the number of points . . . . .	47
11.2 The application was unable to start correctly (0xc000007b) . . . . .	47
11.3 Warning: Only plain strings (not regexs) are supported . . . . .	47
11.4 How to rebuild the models for the released version of Stasm? . . . . .	47
Bibliography . . . . .	48

# Chapter 1

## Introduction

This manual tells you how to build Stasm models.

Note that this manual is devoted to Stasm **Version 4**. For Stasm Version 3, model building is significantly different (see the Stasm version 3 documentation).

### 1.1 Prerequisites

Some prerequisites are assumed.

You should be comfortable working from the command line and have the basic Unix utilities like `cp` on your system (even if working under Windows).

You should be at least somewhat familiar with writing and building C++ programs in general, including building the Stasm executables from source and doing a stack trace after a crash.

You should have read the Stasm Version 4 user manual.

You should also be somewhat familiar with Active Shape Models (ASM). Chapters 1 and 2 of [2] are a good basic explanation. For more detail see the Technical Report by Cootes and Taylor [1]. Stasm Version 4 differs from the classical ASM described in those references in that it uses HAT descriptors (as well as classical 1D profiles). See [5] for a description of HATs.

### 1.2 Non-face objects

Obviously Stasm's emphasis is on faces. Stasm models can however be created for objects other than faces, such as medical images. Chapter 4 works through an example which trains a model for landmarking images of human hands.

For ease of explanation this manual usually uses the term “face”, so please mentally

substitute the name of your object where necessary. Even if your ultimate goal is to train a model for other types of object, it's probably best to first work through the sections on training face models.

## 1.3 A note on programming philosophy

The Stasm code is structured to make it easy for users of the landmark search library to understand the code. It is not really structured for ease of building and tuning new models. Thus model constants are defined near where they are used (which is convenient when reading the code) rather than in a single central file (which would be convenient for building new models).

For example, the constant `BINS_PER_HIST` (which defines the number of bins in a HAT histogram) is `static` to `hat.cpp`. Another example is

```
static const int stasm_NLANDMARKS = 77; // number of landmarks
```

in `stasm_lib.h`. When building a new model, we may have to modify the definition of `stasm_NLANDMARKS` by modifying `stasm_lib.h`.

Having said that, where possible Tasm specific definitions are grouped together in `tasmconf.h`, and most model-specific definitions are in the `MOD_1` directory. In practice there are usually only a few places where source code has to be modified when building new models. These are carefully enumerated in Sections 3.6 and 3.7.

# Chapter 2

## Overview of model building

New Stasm models are built by running Tasm (for “train ASM”). Some modifications to the source code are also necessary.

### 2.1 Output of Tasm

Figure 2.1 is an overview of the model-building process, and shows the files output by Tasm.

The primary outputs are the machine generated C++ include files. These go into the `mh` subdirectory and have a `.mh` suffix (for “machine generated .h”). See the `stasm/MOD_1/mh` directory for an example. Tasm generates the following `mh` files:

1. **The shape model** (`yaw00_shapemodel.mh`). This defines the mean shape, the eigenvalues, and the eigenvectors.
2. **The descriptor models**, one for each landmark at each pyramid level. Examples are `yaw00_lev0_p00_classic.mh` (which defines the descriptor model for pyramid level 0 point 0, which uses a classical 1D profile) and `yaw00_lev2_p57_hat.mh` (which defines the descriptor model for pyramid level 2 point 57, which uses a HAT descriptor).
3. **The list of descriptor models** (`yaw00.mh`). This is a list of references to the model files just mentioned.

Files in the `log` subdirectory are primarily for checking the model.

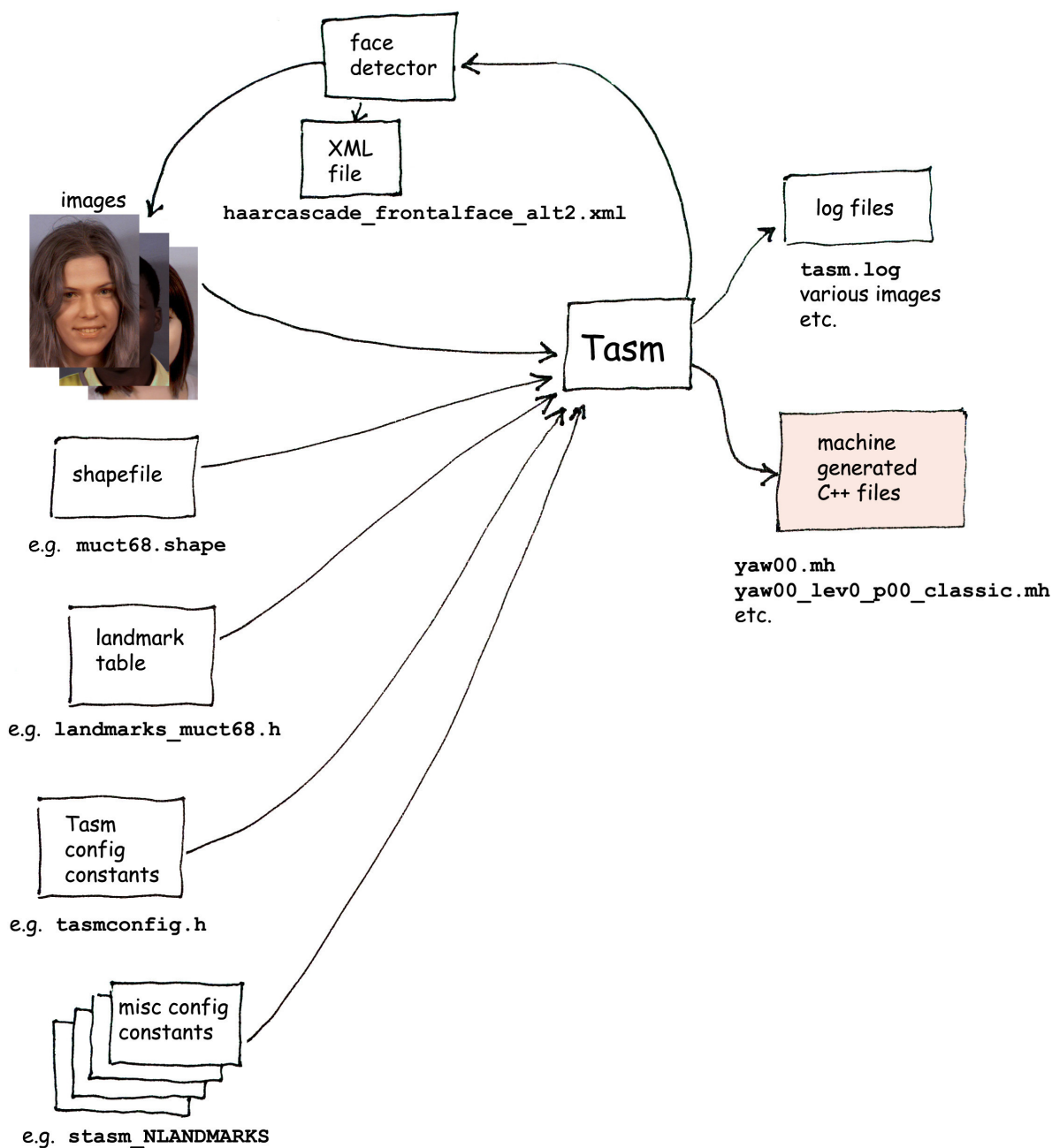


Figure 2.1: Overview of the model-building process.

## 2.2 The model name yaw00

All of the files mentioned above are prefixed by the model name yaw00. This name can be changed by changing `TASM_MODNAME` before running Tasm.

However, this name is not really important. Historically it refers to faces with a yaw of 0 degrees, i.e. frontal faces. Different names are in fact only necessary for multi-view models. The source code for that is not ready for release but our approach to multi-view models is described in [3].



## 2.3 Inputs to Tasm

Figure 2.1 shows the inputs to Tasm.

The **shapefile** is a text file listing the name of each image and its landmark positions (Chapter 5). Associated with each face in the shapefile are optional attribute bits (**atface.h**). For example, we can tag that the face is wearing glasses or has facial hair.

The **landmark table** is a C++ **.h** file listing each landmark and its characteristics (Chapter 6). For example, it specifies for each landmark whether a landmark uses a classical 1D profile or a HAT descriptor. We can also specify for example that a landmark is an eye landmark, so when building the descriptor model Tasm should ignore the landmark if the face is wearing glasses. (This is a nicety that usually has only a small effect on the generated model.) See the function `PointUsableForTraining` for details.

The **face detector** is necessary during training because Tasm needs to figure out how to place the mean shape relative to the face detector frame. This is because prior to an ASM search, we form the start shape by aligning the mean shape to the detected face rectangle. (Actually in the default model we align to the rectangle only if the eyes cannot be detected.)

The `tasmconfig.h` file has Tasm specific definitions, such as the model name:

```
static const char* const TASM_MODNAME = "yaw00";
```

**Other miscellaneous definitions** necessary for Tasm and Stasm are scattered through the source code. For example `stasm_NLANDMARKS` is defined in `stasm_lib.h`.

## 2.4 Training descriptors and `.desc` files

A bit of background on the training descriptors. (We use the term “descriptor” in a general sense to mean both classical 1D profiles and HAT descriptors.) For each landmark at each pyramid level in each training image, Tasm generates a *positive* training descriptor at the correct position of the landmark. For classical 1D profiles, the positive data is all we need (because from this data Tasm can generate the mean profile and the covariance matrix). However, for HAT descriptors, Tasm also generates one or more *negative* training descriptors displaced randomly from the correct position of the landmark (since Tasm needs to build a regression model which estimates if an image patch is situated on or off the correct position). See `GenAndWriteDescMods` for details.

If Tasm’s `-w` command line flag is used, Tasm writes the training descriptors to `.desc` files. You can then do experimental analysis of this data outside of Tasm.

# Chapter 3

## Building a model: an example

This section describes in detail how to build a new model for faces. For a non-face example, please see Chapter 4 (although we recommend that you work through this chapter anyway).

For concreteness, we will build a 68-point frontal face model using the MUCT face data [4]. These 68 point definitions are very close to the XM2VTS definitions.

**Note 1:** The released version of Stasm was trained on MUCT data landmarked with 77 points, not the 68 point shapefile used in this chapter’s example. See the FAQ Chapter 11.

**Note 2** (for users of the Multi PIE faces): We emphasize that the MUCT/XM2VTS 68 points are not the same as the 68 points of the Multi PIE data. If you are preparing a model using the 68 point Multi PIE data, then you will need to modify the Stasm source code — grep for the string 68. But before you do that, first work through this example using the MUCT 68 points.

### 3.1 Step 1: Build Stasm and Tasm

Copy the Stasm source code to a working directory and check that you can rebuild Stasm from scratch.

The file paths in this document assume that we are working in the same directory as the make files (or `makefile_include` files). That is we are working in `vc10`, `vc10x64`, `mingw`, or similar, which could also be a directory of your own name at the same level. To verify that our working directory is correct, the following should correctly list `stasm.h`:

```
ls ../stasm/stasm.h
```

## 3.2 Step 2: Prepare the images

The first step after ensuring that we can build Stasm is to prepare the images. For this example, simply download the images from the MUCT website <http://www.milbo.org/muct> and put them into a local directory. It's probably most convenient to put all the MUCT images in one directory, but you don't have to.

## 3.3 Step 3: Prepare the shapefile

The shapefile must now be prepared (Chapter 5). For this example, we will use a shapefile `muct68_nomirror.shape` provided with the Stasm source code.

Edit the `Directories` string (Chapter 5.1) in the shapefile for the location of the images we downloaded above.

For safety, we recommend that you keep shapefiles read-only unless you are actively editing them:

```
chmod 444 ../tasm/shapes/*.shape
```

Use Marki to verify the landmarks in the shapefile (Chapter 8):

```
marki -V ../tasm/shapes/muct68_nomirror.shape
```

### Notes on this shapefile

The `nomirror` in the filename indicates that mirrored shapes are not included in the shapefile. We use a non-mirrored shapefile in this example simply so we don't have to generate the mirrored images before we run the example (Chapter 10).

(The file `muct68.shape` includes mirrored shapes. It and several other shape files are included in the `more` zip file on the Stasm web page.)

In this shapefile, the coordinate system has 0,0 at the top left, as opposed to the coordinate system used in the older versions of Stasm and also on the MUCT webpage, where 0,0 is the center of the image.

This shapefile includes facedets (Section 3.9.2).

## 3.4 Step 4: Prepare the landmark table

The landmark table describes the attributes of each point (Chapter 6). For example, it specifies if we generate a classical 1D profile or a HAT descriptor at the point.

The landmark table should be created before running Tasm. In this example we will use the provided MUCT 68 point landmark table `landtab_muct68.h`. In Section 3.6 we will modify `landmarks.h` to invoke this landmark table.

## 3.5 Step 5: Prepare the face detector

We will use the OpenCV frontal face detector file currently used by Stasm, `haarcascade_frontalface_alt2.xml`.

More generally, for non-frontal faces or for objects other than faces, we will need to generate a new XML file by training a new detector, typically with the OpenCV tools. We would be interested in hearing from you if you plan to do so. You can do a basic check of a new detector by examining the images generated by Stasm with `TRACE_IMAGES` defined.

## 3.6 Step 6: Primary edits to the source code

Make changes to the source files as follows.

In `stasm_lib.h`, change

```
static const int stasm_NLANDMARKS = 77;
```

to

```
static const int stasm_NLANDMARKS = 68;
```

In `landmarks.h`, change

```
#include "landtab_muct77.h"
```

to

```
#include "../tasm/landtab/landtab_muct68.h"
```

## 3.7 Step 7: Possible additional edits to the source code

We will often also need to change the code as described in this section (Section 3.7). However, for the current 68-point example *none of the changes* listed in this section are necessary.

### 3.7.1 facedet.cpp

If necessary, change the face detector from `haarcascade_frontalface_alt2.xml` to your new face detector file. You may also want to change the definitions of `SCALE_FACTOR`, `MIN_NEIGHBORS`, and `DETECTOR_FLAGS`.

### 3.7.2 `initasm.cpp`

We may need to change the arguments to the constructor for `mod_yaw00`. These changes affect the landmark search, but don't affect Tasm.

`eyaw` is used to (i) guide the selection of the start shape in `startshape.cpp` and (ii) to choose the appropriate submodel in multi-model versions of Stasm (not released). If we are working with non-frontal faces (Chapter 9), we should change `eyaw` and make the appropriate downstream changes (Section 9.3).

`estart` determines if we should use the eyes and mouth to help position the start shape in `startshape.cpp`.

Sections 3.7.5 and 9.3 describe `eyaw` and `estart` in more detail. The `e` in `eyaw` and `estart` is a naming convention indicating that these are `enums`.

`neigs` and `bmax` control the shape model. The only reliable way to adjust these values (as for most model constants) is to test a range of values by building and tuning a variety of models (Section 3.13).

`hackbits` is ignored unless the shapes have 77 points. It is used by the code that corrects absurd point positions (e.g. the chin inside the mouth). This correction is done during the landmark search at coarse pyramid levels after the points have been conformed to the shape model.

### 3.7.3 `tasmconf.h`

The file `tasmconf.h` has a number of definitions for Tasm. Please see the file for details.

### 3.7.4 `tasmconf.cpp`

The `tasmconf.cpp` file has code that may have to change if we are using a different face detector or if our shape format cannot be converted to a `shape17` (Section 3.7.6).

The function `FaceDetFalsePos` in `tasmconf.cpp` is used by Tasm to check if the face detector box is positioned correctly (using the the shape from the `shapefile` as a reference). Badly positioned detects (i.e. false positives) are ignored by Tasm. It has some code customized for frontal faces which requires conversion to `shape17s`.

### 3.7.5 `startshape.cpp`

The start-shape code is driven by the setting of `estart` in `initasm.cpp`. The start-shape code is invoked when doing a landmark search but not when running Tasm.

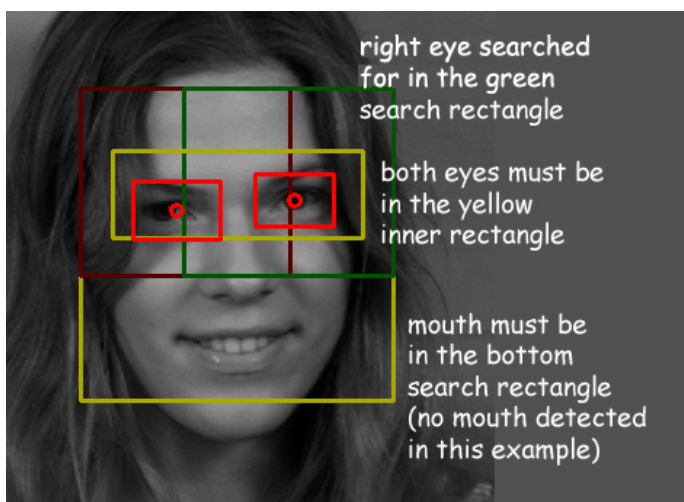


Figure 3.1:  
*Eye and mouth search rectangles*

*This image was generated by building Stasm with `TRACE_IMAGES` defined.*

```
enum ESTART // do we use the detected eyes or mouth to help position the startshape?
{
    ESTART_RECT_ONLY,    // use just the face det rect to align the start shape
    ESTART_EYES,        // use eyes if available (as well as face rect)
    ESTART_EYE_AND_MOUTH // uses eye(s) and mouth if both available
};
```

With a new model, for easily working code (but probably non-optimal fits), set `estart` to `ESTART_RECT_ONLY`. This tells the start shape code to use just the detected face rectangle to position the start shape. For non-faces, use `ESTART_RECT_ONLY`.

For frontal faces, we get better results with `ESTART_EYES`, which tells the start shape code to first detect the eyes and use their positions if available to position the start shape. The eye positions are also used to rotate the face upright, of special benefit for HAT descriptors which have limited ability to handle object rotation. See `StartShapeAndRoi`.

Use `ESTART_EYE_AND_MOUTH` to also use the mouth to position the start shape. In our experience this gives better results for three-quarter faces but not for frontals. Your mileage may vary.

If eye or mouth detection is necessary, the OpenCV eye and mouth detectors will be initialized and invoked before generating the start shape. These detectors are dependent on the face detector used, because they depend on the way the face detector rectangle frames the face, because the eye and mouth search areas are defined as a subset of this rectangle (Figure 3.1). Build Stasm with `TRACE_IMAGES` to generate images that show these areas. The existing start shape code is optimized for the current OpenCV frontal face detector. For other face detectors you will probably have to adjust the hard-coded constants in `startshape.cpp`.

### 3.7.6 `shape17.cpp`

Most face landmark schemes can be converted to what we call a *shape17*. A *shape17* has a standard set of 17 points (Figure 3.2). These are the same as Cristinacce's `me17`

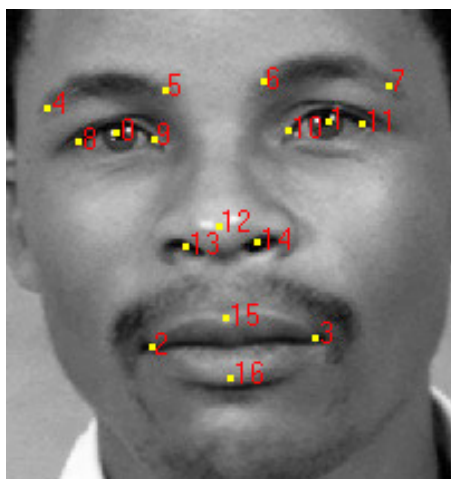


Figure 3.2: *The shape17 points.*

*We can convert any set of face landmarks to this lowest-common-denominator set of points.*

points.

Converting to a shape17 is convenient when we need the positions of the eyes and mouth regardless of whether the original shape format has 77, 68, 22, etc., points. For example, we may need the eye-mouth distance. Shape17s are also used to calculate the me17 fitness measure.

You will have to extend the code in `shape17.cpp` if it does not already support conversion of your shape format to a shape17. (It currently supports AR, BioID, XM2VTS, and a few other formats.) You can check your changes by running Tasm and looking at `shape17.bmp` in the Tasm log directory (Figure 3.3). Tasm does some automatic sanity checks but cannot check everything (see `SanityCheckShape17`).

Some shape formats do not define the eye pupils. These can be approximated in the shape17 from surrounding landmarks. Adjustments can also be made for minor differences in point definitions — for example, the nostril points may be at the top of the nostrils instead of in the center.

Note for Multi PIE 68 users: The 68 point shapes currently supported by `shape17.cpp` are the XM2VTS/MUCT points. They are not the Multi PIE 68 points. See Note 2 in the introduction to Chapter 3 (page 9).

### 3.7.7 convshape.cpp

The file `convshape.cpp` has code for converting a shape format to another format (e.g. from 77 points to 68 points). In the special case where the output shape has 17 points, it calls the shape17 code (Section 3.7.6).

We may need to convert our shapes to a different number of points for testing. For example, if we use the Swas utility to measure fits with our model on the BioID faces (Section 3.13), we will need to convert the shapes to the 20 point BioID shapes.

Check the changes to `convshape.cpp` by running Tasm and looking at the `shapeN.bmp`

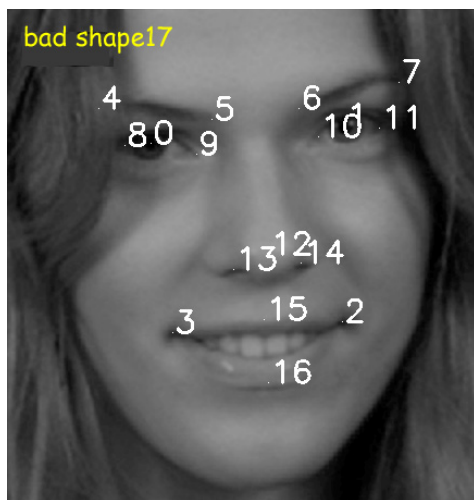


Figure 3.3: An example of incorrect conversion to `shape17` (the table in `shape17.cpp` is wrong).

The points seem correct at first glance, but actually the mouth corner points are misnumbered. Subtle problems may occur when running `Tasm` or `Stasm`.

`Tasm` creates log images for catching this kind of mistake.

images in the log directory (where N is the converted number of points).

### 3.7.8 Scattered constants

There are other constants in the code which affect the generated model. The default settings of these are probably near optimal for 68 point faces (because that is similar to the 77 point model that `Stasm` was tuned for). Some of these constants are listed below. See the source code for details. If in doubt leave them as they are.

```
int EYEMOUTH_DIST = 100; // scale image to this before ASM search starts

double PYR_RATIO = 2; // scale image by 2 at each pyramid level
int N_PYR_LEVS = 4; // number of levs in image pyramid
int SHAPEMODEL_ITERS = 4; // shape model iterations per pyr level

int EYE_MIN_NEIGHBORS = 3; // for the OpenCV eye detector
int MOUTH_MIN_NEIGHBORS = 3; // for the OpenCV mouth detector
double MIN_MOUTH_WIDTH = .27; // as frac of face width

int TASM_1D_PROFLEN = 9; // length of classic 1D profiles

// constants for the HAT descriptors

int GRIDHEIGHT = 4; // 4 x 5 grid of histograms in descriptor
int GRIDWIDTH = 5;
int BINS_PER_HIST = 8; // 8 gives a 45 degree range per bin
double WINDOW_SIGMA = .5; // gaussian window as frac of patch width
double FINAL_SCALE = 10; // arb but 10 is good for %g printing of descriptors

int HAT_MAX_OFFSET = 4; // search grid +-4 pixs from current posn
int HAT_SEARCH_RESOL = 2; // search resolution, search every 2nd pixel
int HAT_PATCH_WIDTH = 9*2+1;
int HAT_PATCH_WIDTH_ADJ = -6;
int HAT_START_LEV = 2; // HAT descriptors are for pyr levs 0..2
```

We make special mention of `HAT_START_LEV`. `Tasm` will generate classical 1D profiles



for all points at pyramid levels higher than `HAT_START_LEV`, regardless of the `AT_Hat` bits in the landmark table. If you want HAT descriptors for all pyramid levels, change `HAT_START_LEV` to a value bigger than any pyramid level (say 9).

In the default model, HAT descriptors are not used at low pyramid resolutions. More precisely, the default model uses classical 1D gradient descriptors at the coarsest pyramid level and on the jaw points at all pyramid levels, and HAT descriptors otherwise. That is what gave the best results during model tuning.

## 3.8 Step 8: Rebuild Stasm and Tasm

After making the above changes to the source code, rebuild the Stasm executables. Make sure Tasm gets rebuilt.

## 3.9 Step 9: Do a trial run of Tasm with a small set of images

Run Tasm on just 100 images as follows:

```
tasm ../tasm/shapes/muct68_nomirror.shape 100 0 [ade]
```

This call to Tasm builds a model from the first 100 faces in the shapefile. This low number of shapes allows us to quickly build a model to iron out obvious errors. For even more speed, decrease the 100 on the above command line to 3, although this will cause a few extra messages. (These are harmless for our trial purposes here. There will not be enough profiles to build the profile model covariance matrices, so Tasm will generate identity matrices for the covariance matrices.)

The string `[ade]` is a regular expression specific to this example telling Tasm to use only the frontal faces in the MUCT set. It tells Tasm to use only faces with `a`, `d`, or `e` in their names, that is, only approximately frontal faces. See the MUCT documentation for the MUCT file naming conventions.

(The `b` shapes could also be included in this example, since they are also more-or-less frontal. But they have missing points, and so for maximum simplicity have been omitted.)

Note that no points are missing in this subset of shapes. Chapter 9 discusses how to build a model with shapes with missing points. By convention in Stasm missing points have coordinates `0,0`. (Our in-house approach has been to manually landmark points as missing if they are obscured by a hand or the side of the face or nose. We do not mark points obscured by glasses or thinnish hair as missing, but instead estimate their position during manually landmarking. There is a certain amount of arbitrariness about such decisions.)

Tasm results go to the `tasmout` directory. You should do a manual sanity check on the images in `tasmout/log`. See Section 7.

Use `tasm -?` for further details on the Tasm command line.

### Note for UNIX-like systems

The Stasm code currently supports regular expressions only with the Microsoft compiler. (It is difficult for us to provide a universal solution because regular expression libraries are currently compiler dependent, and we are reluctant to require say the Boost libraries to build Stasm.) But this should be easily fixable for your particular compiler. Search the Stasm sources for `HAVE_REGEX` to see what has to change. A simpler work-around is to run the example without the regular expression:

```
tasm ../tasm/shapes/muct68_nomirror.shape 100
```

Tasm will use all shapes, not just the ones matching `[ade]`, but it doesn't really matter for this example. Another work-around is use a new shapefile created by extracting only the shapes that you are interested in from the existing shapefile. (We use `awk` or Emacs keyboard macros for this kind of thing.)

## 3.9.1 Tasm prints

Figure 3.4 show the prints from Tasm for this example. There may be minor differences on your system, depending on the version of Tasm, directory names, etc.

Messages ending in `...` indicate that Tasm has printed just the first of such similar messages. Further similar messages go into the log file `tasm.log`, but to reduce clutter are not printed on the screen. Actually only the first 100 such messages go to the log file. That is usually enough to debug problems. See the function `PrintOnce`.

## 3.9.2 Face detector records in the shapefile

Tasm invokes the face detector to figure out how to align the mean face to the mean face detector frame.

In the shapefile used in the current example, the facedets are pre-saved in the shapefile. For speed, Tasm uses these pre-saved facedets instead of invoking the face detector for each image.

More generally, if your shapefile does not have facedets, you can add them to the shapefile as described in Section 5.2. The only advantage is that Tasm runs faster. This matters during model tuning where we may want to build dozens of models.

```

>tasm ../tasm/shapes/muct68_nomirror.shape 100 0 [ade]
rm -f tasmout/log/* # tasm removes stale files
rm -f tasmout/mh/*
Generating tasmout/log/tasm.log # tasm logs to tasm.log
Reading ../tasm/shapes/muct68_nomirror.shape: 3755 shapes
Using the first 100 shapes matching [ade] and mask 0x700 0x0 (i000qa-fn ... i011qa-mn)
(Mask0 0x700 is BadImg Cropped Obscured) # shapes with these attrs skipped, TASM_DISALLOWED_ATTR_BITS
--- Generating the shape model ---
Reference shape i000qa-fn is at index 0 and has 68 landmarks # ref shape used for aligning shapes
Mean shape outermost points (0,13) angle -5.89 degrees, eye angle 2.67 degrees
88% percent variance is explained by the first 10 shape eigs: 37 14 11 6 5 5 3 2 2 1%
Generating mean shape aligned to the facedets
    100.00% of shapes with all landmarks have a facedet in the shapefile
    will not run the facedet on the actual images (will use just the facedets saved in the shapefile)
Done generating mean shape aligned to the facedets # Tasm prints a summary of the shapes used
    100 faces detected (100.00%), 100 facedets in the shapefile, 0 facedets in the images
    0 facedet false positives, 0 shapes with missing landmarks
    100 faces actually used (valid facedet and all points) (100.00% of the shapes in the shapefile)
    0 missing both eyes, 0 missing just left eye, 0 missing just right eye, 0 missing mouth
Generating tasmout/mh/yaw00_shapemodel.mh
[0.3 secs to generate the shape model]
--- Generating images for manual checking of the 68 point landmark table ---
    tasmout/log/landmark00.bmp... # an image for each point 0...67
    tasmout/log/meanshape.bmp
    tasmout/log/shape17.bmp # the shape after conversion to a shape17
    tasmout/log/shape_all168.bmp # all 68 points
--- Generating yaw00.mh from the landmark table ---
Generating tasmout/mh/yaw00.mh
--- Generating the descriptor models from 100 shapes ---
Pyramid level 0 reading 100 training images 0_1_2_3_
    ignoring eye in i003se-f3 (tag 0x800)... 4_5_6_7_8_9_0 [4.1 secs]
Generating pyramid level 0 models
Generating tasmout/mh/yaw00_lev0_p00_classic.mh...
0_1_2_
Generating tasmout/mh/yaw00_lev0_p15_hat.mh...
3_4_5_6_7_8_9_0 [1.5 secs]
Pyramid level 1 reading 100 training images 0_1_2_3_4_5_6_7_8_9_0 [1.6 secs]
Generating pyramid level 1 models 0_1_2_3_4_5_6_7_8_9_0 [1.1 secs]
Pyramid level 2 reading 100 training images 0_1_2_3_4_5_6_7_8_9_0 [1.0 secs]
Generating pyramid level 2 models 0_1_2_3_4_5_6_7_8_9_0 [1.0 secs]
Pyramid level 3 reading 100 training images 0_1_2_3_4_5_6_7_8_9_0
    in point 27, only 97 descriptors used from 100 shapes because some points were skipped...
Generating pyramid level 3 models 0_1_2_3_4_5_6_7_8_9_0 [0.6 secs]
[Total time 14.6 secs, 47% physical mem all processes, 34MB peak this process]

```

Figure 3.4: *Tasm prints for the MUCT 68 point example (trial run with 100 shapes).*

## 3.10 Step 10: Rebuild Stasm with new model files

Once we get a clean run of Tasm, copy the machine generated C++ files to the MOD\_1 directory:

```

rm ../stasm/MOD_1/mh/* # not essential but prevents mixing old and new files
cp tasmout/mh/* ../stasm/MOD_1/mh

```

Rebuild the Stasm executables.

Note that we can avoid the above copying step by running Tasm as follows:

```
tasm -d ../stasm/MOD1 ../tasm/shapes/muct68_nomirror.shape 100 0 [ade]
```

which puts the output subdirectories directly into the `../stasm/MOD1` directory. This is often convenient, but the danger is that if we have a faulty or partial build of the `mh` files, we will be unable to re-build the Stasm executables. Recover from this by restoring the `mh` directory from the original Stasm sources.

### 3.11 Step 11: Test the new version of Stasm

After rebuilding Stasm above, do a basic test of our new version of Stasm with

```
stasm ../data/testface.jpg
```

There should be no error or warning messages. Manually examine the output image `testface_stasm.bmp` (Figure 3.5). The shape is roughly positioned correctly on the face but the eye and mouth corners are quite off. That's not suprising because we used only 100 images to build the model.

### 3.12 Step 12: Run Tasm with the complete set of shapes

Repeat Steps 10 and 11, but run Tasm with the full set of faces (instead of just 100):

```
tasm -d ../stasm/MOD1 ../tasm/shapes/muct68_nomirror.shape 0 0 [ade]
```

and rebuild and test the model.

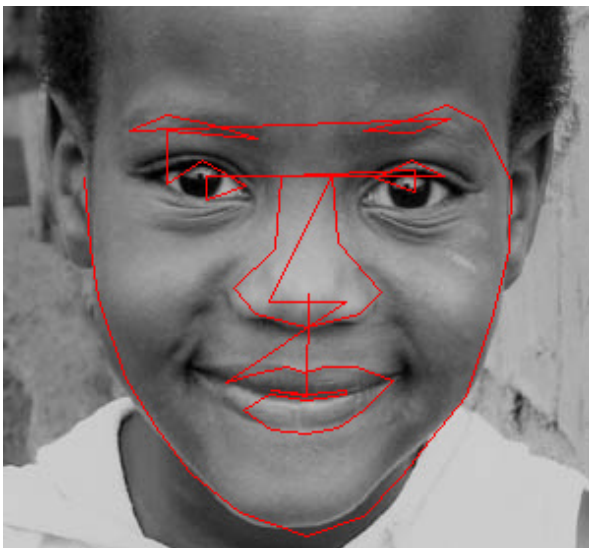


Figure 3.5: *An image landmarked by the 100-shape model.*

More generally, you may get somewhat better results if you enlarge the training set by including mirrored faces. Do this as described in Chapter 10. Or mirror the images yourself (using tools such as Photoshop or GraphicsMagick) and use the shapefile which includes the mirrored shapes `muct68.shape` (included in the `more` zip file on Stasm web page).

### 3.13 Step 13: Tune the model

To measure our model's performance, run the Swas utility on a tuning set. For example:

```
swas -i ../tasm/shapes/muct68_nomirror.shape 300 2013 a
```

This runs Swas on 300 frontal faces randomly chosen with seed 2013 from the MUCT set. The string `a` matches only strictly frontal faces.

The `-i` flag tells Swas not to generate images, for speed. Use `swas -?` and see the Swas source code for details.

Note: The above example uses the *training* set to *tune* the model. Preferably but not essentially, the tuning set should be disjoint from the training set to minimize overfitting. The danger is that we will create a model that works very well on the training set but doesn't generalize well to new data. More importantly, the *training* and final *test* set should be disjoint. See for example Section 4.2 of [2] or indeed any book on modern statistical modeling.

When Swas generates images (no `-i` flag), it prefixes the image name with the fitness (e.g. `0.064.imagename.me17.bmp` where `0.064` is the `me17` fitness). Thus for manual eyeballing the images are conveniently ordered in the directory with the worst fits last.

The Swas fit results go to a "fit" file in the `fit` directory. The fit file name depends on the parameters we pass to Swas. Use MATLAB or similar to analyze and plot the fit file. Our personal weapon of choice is the R language with the provided file `swas.R`. Edit the top of that file to reference your fit file(s).

You may of course prefer to create your own utility instead of Swas, or modify Swas for your purposes.

#### Tuning the model

To tune the model, change constants in the source code, then rebuild and re-test the model. You could start by changing `neigs` and `bmax` in `initasm.cpp` (Section 3.7.2). Try something like `neigs=30` and `bmax=3`, but really for the 68 point model the existing values are probably already close to optimal. Typically for this kind of thing we use a shell script or Python script which builds and tests a range of values.

Some changes require rebuilding the model; others just require a re-compile:

Example 1: Changes to `TASM_1D_PROFLEN` require rebuilding the model, to build `mh` files with 1D profiles with the new length.

Example 2: Changes to `neigs` or `bmax` in `initasm.cpp` require just a re-compile because these constants are used only during searching.

### Extending the code that converts the shape format

For testing against arbitrary shapefiles, we may need to extend `shape17.cpp` and `convshape.cpp`. This will be necessary if the shapefile has a different number of points from the shapefile. See Sections 3.7.6 and 3.7.7.

For example, let's say we are using Swas to test our 68 point model on the 20 point BioID faces:

```
swas -i ../data/bioid.shape
```

For the 68 point shapes, the current code supports conversion to shape17s but not to BioID shapes. Swas will thus generate a fit file with `me17s`, but not with fits measured against all 20 BioID points (the `meanfit` column in the fit file will have the special value 0.9999).

To remedy this, extend the `case` statement in `ConvertShape` by creating and calling the function `Shape68As20`. You can check the changes to `ConvertShape` by running `Tasm` and looking at `shape20.bmp` in the `Tasm` log directory.

# Chapter 4

## Non-faces: the hand example

This section describes how to build a new model for objects that are not faces. We will build a toy model using Mikkel Stegmann’s hand data generated at the Technical University of Denmark (Figure 4.1).

The word “toy” is used because there are only 40 training images. Also, the hand detector we will use was created for this chapter and was trained on just these 40 images. It is a poor hand detector, but suffices for this example.

### 4.1 Step 1: Build Stasm and Tasm

Copy the Stasm source code to a working directory and build Stasm from scratch.

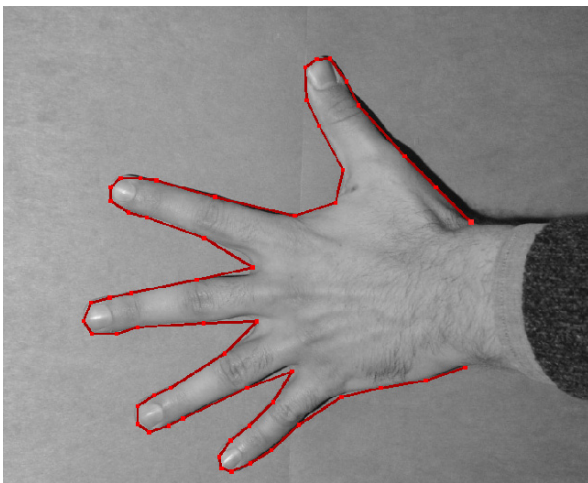


Figure 4.1:  
*An example from the Stegmann DTU hand dataset with 56 manually landmarked points.*

*This data is downloadable from <http://www2.imm.dtu.dk/~aam>*

## 4.2 Step 2: Prepare the images

Download the hand images from DTU website <http://www2.imm.dtu.dk/~aam> and put them into a local directory.

## 4.3 Step 3: Prepare the shapefile

We will use `hands.shape` which is provided with the Stasm source code. (This shapefile was created by reformatting the landmark data downloaded from the DTU website.)

Edit the `Directories` string (Chapter 5.1) in `hands.shape` for the local directory of the images we downloaded above.

Use Marki to verify the landmarks in the shapefile (Chapter 8):

```
marki -V ../tasm/shapes/hands.shape
```

## 4.4 Step 4: Prepare the landmark table

We will use the provided 56-point hands table `landtab_hands.h` and incorporate it into the code in Section 4.6.

## 4.5 Step 5: Prepare the “face” detector

We will use the hand detector `hands_toy.xml` provided with Stasm, and incorporate it into Stasm in Section 4.7.1.

As mentioned above, this is a toy hand detector. It suffices for this example, but it won't reliably detect arbitrary hands and returns a lot of false positives (it thinks there are hands everywhere). We will work around the false-positive issue in Section 4.7.1 where we change `minwidth` to 50% (so the detector only finds hands that are larger than 50% of the image width).

More generally, if you do not yet have a detector for your object, but your objects are always positioned at roughly the same place in the image, you may want to build a provisional dummy detector that always returns the same rectangle. Do this by modifying the `DetectFaces` function to something like the code below (the constants below are approximately correct for the current hand data but have not been optimized):



```

void DetectFaces(          // approximate position of hand returned in detpars
    vec_DetPar& detpars, // out
    const Image& img,     // in
    int minwidth) // in: ignored (func signature compatibility)
{
    if (img.cols != 800 || img.rows != 600) // sanity check
        Err("Image must be 800x600 (your image is %dx%d)", img.cols, img.rows);
    DetPar detpar;
    detpar.x = 350;          // approximate center of hands
    detpar.y = 350;
    detpar.width = 600;     // approximate size of hands
    detpar.height = 400;
    detpar.yaw = 0;
    detpar.eyaw = EYAW00;
    detpars.resize(1);
    detpars[0] = detpar;
}

```

## 4.6 Step 6: Primary edits to the source code

Make changes to the source files as follows.

In `stasm_lib.h`, change

```
static const int stasm_NLANDMARKS = 77;
```

to

```
static const int stasm_NLANDMARKS = 56;
```

In `landmarks.h`, change

```
#include "landtab_muct77.h"
```

to

```
#include "../tasm/landtab/landtab_hands.h"
```

## 4.7 Step 7: Additional edits to the source code

Depending on the object you are landmarking, you may need to make the code changes described in Section 3.7. For the current hands example we need to change only `initasm.cpp`, `facedet.cpp`, and `tasmconf.h` as described in the next three subsections.

### 4.7.1 `facedet.cpp`

In `facedet.cpp` make the following changes:

1. Change

```
OpenDetector(facedet_g, "haarcascade_frontalface_alt2.xml", datadir);
```

to

```
OpenDetector(facedet_g, "../tasm/landtab/hands_toy.xml", datadir);
```

2. Add the following line of code to `DetectFaces` before the initialization of `minpix`:

```
minwidth = 50; // force minwidth to reduce false positives with toy detector
```

This says that the minimum width of a hand must be 50% of the image width, as a work-around for the propensity of our toy hand detector to generate false positives. (The command line help for some executables will now be wrong, where the help says that you can change `minwidth` from the command line. For our current purposes we don't really care that the help is wrong.)

## 4.7.2 `initasm.cpp`

In the constructor for `mod_yaw00` in `initasm.cpp`:

1. Change `estart` to `ESTART_RECT_ONLY`. This tells the start shape code to use the position of the detected hand rectangle to position the start shape, and not to search for eyes and mouths.
2. Change `neigs` and `bmax` to 10 and 3. These values are probably not optimal but allow us to get started.
3. Change `hackbits` to 0. Not essential (because the shape hacks code ignores shapes without 77 points) but makes it clearer that the `hackbits` code is not activated.

The resulting code should look like this:

```
static const Mod mod_yaw00( // constructor, see asm.h
    EYAW00,                // eyaw
    ESTART_RECT_ONLY,      // estart
    datadir,
    yaw00_meanshape,
    yaw00_eigvals,
    yaw00_eigvecs,
    10,                    // neigs
    3,                    // bmax
    0,                    // hackbits
    YAW00_DESCMODS,        // defined in yaw00.mh
    NELEMS(YAW00_DESCMODS));
```

## 4.7.3 `tasmconf.h`

In `tasmconf.h` make the following changes:

1. Change

```
static const bool TASM_DET_EYES_AND_MOUTH = true;
```

to

```
static const bool TASM_DET_EYES_AND_MOUTH = false;
```

This tells Tasm not to call the eye and mouth detectors when creating `facedet.shape`. This is not essential, but prevents irrelevant messages from Tasm about failing to find eyes and mouths in images of hands.

## 2. Change

```
static const int TASM_FACEDET_MINWIDTH = 10;
to
static const int TASM_FACEDET_MINWIDTH = 50;
```

This is not essential, because we have already forced `minwidth` to 50 in Section 4.7.1.

Note that we don't need to change the model name `yaw00`, but as a nicety we could change that to say `hands` (Section 2.2).

## 4.8 Step 8: Rebuild Stasm and Tasm

After making the above changes to the source code, rebuild the Stasm executables. Make sure Tasm gets rebuilt.

## 4.9 Step 9: Do a trial run of Tasm

Since in this toy example all we have is a small set of images, we can actually use all available images for the trial run:

```
tasm -d ../stasm/MOD1 ../tasm/shapes/hands.shape
```

Figure 4.2 show the prints from Tasm for this example. There may be minor differences on your system, depending on the version of Tasm, directory names, etc.

Note that Tasm issues the message

```
Do not know how to convert a 56 point shape to a 17 point face...
```

The `shape17` code (Section 3.7.6) does not know how to convert a 56 point hand to a 17 point face (which is correct, because hands cannot be converted to faces). The `shape17` code is invoked when normalizing the shape before the ASM search begins:

```
stasm_search_auto_ext -> ModSearch_ -> GetPrescale -> EyeMouthDist -> Shape17OrEmpty
```

We can simply ignore the message, or easily suppress it by commenting out the code that issues the message. Or better, write a custom `GetPrescale` function for our shape format. In the current implementation (`EyeMouthDist`), if the eye-mouth distance is not available Stasm falls back to using half the shape's horizontal extent. That approach is vulnerable to a single outlying point. A better approach may be to scale the hand so the mean distance of points from the hand centroid is say 100 pixels.

```

>tasm -d ../stasm/MOD_1 ../tasm/shapes/hands.shape
rm -f ../stasm/MOD_1/log/*
rm -f ../stasm/MOD_1/mh/*
Generating ../stasm/MOD_1/log/tasm.log
Reading ../tasm/shapes/hands.shape: 40 shapes
Using all 40 shapes matching mask 0x700 0x0 (0000 ... 0069)
(Mask0 0x700 is BadImg Cropped Obscured)
--- Generating the shape model ---
Reference shape 0000 is at index 0 and has 56 landmarks
Mean shape outermost points (27,55) angle -6.97 degrees
Do not know how to convert a 56 point shape to a 17 point face...
99% percent variance is explained by the first 10 shape eigs: 63 17 8 4 2 2 1 1 0 0%
Generating mean shape aligned to the facedets
  no facedets in the shapefile
  opening face detector (TASM_FACEDET_MINWIDTH is 50)
  0000: found face in the image...
  will not search for eyes or mouth (TASM_DET_EYES_AND_MOUTH is false)
Generating ../stasm/MOD_1/log/facedet.shape
Done generating mean shape aligned to the facedets
  40 faces detected (100.00%), 0 facedets in the shapefile, 40 facedets in the images
  0 facedet false positives, 0 shapes with missing landmarks
  40 faces actually used (valid facedet and all points) (100.00% of the shapes in the shapefile)
Generating ../stasm/MOD_1/mh/yaw00_shapemodel.mh
[3.3 secs to generate the shape model]
--- Generating images for manual checking of the 56 point landmark table ---
  ../stasm/MOD_1/log/landmark00.bmp...
  ../stasm/MOD_1/log/meanshape.bmp
  ../stasm/MOD_1/log/shape_all156.bmp
No points have partners in the 56 point landmark table
--- Generating yaw00.mh from the landmark table ---
Generating ../stasm/MOD_1/mh/yaw00.mh
--- Generating the descriptor models from 40 shapes ---
Pyramid level 0 reading 40 training images [2.9 secs]
Generating pyramid level 0 models
Generating ../stasm/MOD_1/mh/yaw00_lev0_p00_classic.mh...
0_1_2_3_4_5_6_7_8_9_0 [0.4 secs]
Pyramid level 1 reading 40 training images [2.2 secs]
Generating pyramid level 1 models 0_1_2_3_4_5_6_7_8_9_0 [0.3 secs]
Pyramid level 2 reading 40 training images [1.8 secs]
Generating pyramid level 2 models 0_1_2_3_4_5_6_7_8_9_0 [0.3 secs]
Pyramid level 3 reading 40 training images [1.8 secs]
Generating pyramid level 3 models 0_1_2_3_4_5_6_7_8_9_0 [0.4 secs]
[Total time 19.7 secs, 54% physical mem all processes, 16MB peak this process]

```

Figure 4.2: *Tasm prints for the hand example.*

## 4.10 Step 10: Rebuild Stasm with new model files

Rebuild the Stasm executables.

## 4.11 Step 11: Test the new version of Stasm

Do a basic test of our new version of Stasm with something like

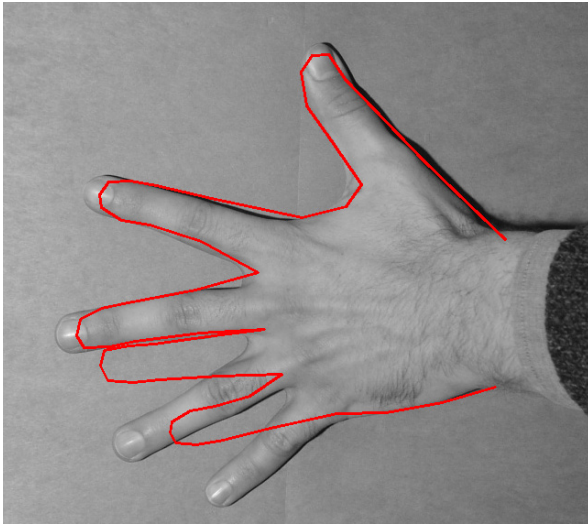


Figure 4.3:  
*An example fit from our toy hand model*

*Stasm has confused the boundaries between some of the fingers.*

```
stasm /faces/hands/0000.jpg
```

Manually examine the output image `0000_stasm.bmp`. The actual fit will most likely be unimpressive because we used only 40 images to build the model (Figure 4.3).

## 4.12 Step 12: Run Tasm with the complete set of shapes

This step is not necessary since we already used all available images in Section 4.9.

## 4.13 Step 13: Tune the model

To measure our model's performance, run the Swas utility: For example:

```
swas ../tasm/shapes/hands.shape
```

Plot the resulting fit file using MATLAB or similar.

There is not much point in tuning this toy model. Perhaps we could minimize the finger confusion in Figure 4.3 by playing with `EYEMOUTH_DIST`, `neigs`, and `bmax`, or by using HATs instead of 1D profiles (although hand rotation may be an issue with HATs).

# Chapter 5

## Shapefiles

Shapefiles are central to training and testing Stasm models. A shapefile is a text file which lists the basename of each image and its landmarks (Figure 5.1).

Also associated with each shape is an optional set of face attributes (also called *tag bits* in the code). For example, we can specify that the face is wearing glasses or has facial hair. See `atface.h` for the bit definitions. To see exactly how Tasm uses the tag bits, grep `AT_` in `../tasm/src/*`, and especially look at the function `PointUsableForTraining`.

Use the tool Marki for checking and creating shapefiles (Chapter 8).

### 5.1 The Directories string

Near the top of the shapefile is the `Directories` string.

This is a list of one or more directories (separated by semicolons) specifying where the images are stored.

You should edit this string for your environment.

### 5.2 Face detector records in the shapefile

Face detector results can be stored in the shapefile. For speed, Tasm will use these facedets if available instead of invoking the face detector for each image (Section 3.9.2).

Facedets are stored in the shapefile with the tag `81000000`. See the lower part of Figure 5.1 and `muct68_nomirror.shape` for examples.

Use the `CtrlQ` key in Marki to check the facedet record for an image. Marki will draw the facedet rectangle, and show the eye and mouth positions if available (Figure 5.2).

```

shape # first word must be "shape", comments are preceded by "#"

# "Directories" is the image search path with each directory separated by a semicolon
Directories /faces/muct/jpg;/faces/muct/jpg-mirror

0000 image1      # attributes (in hex) and image name (without JPG suffix)
{ 68 2          # a matrix with 68 rows, each row is the x y coordinates of a point
201 347
201 380
0 0             # special coords 0,0 means that the point is missing
202 407
...
}
0004 image2      # attribute bit 4 means the face is wearing glasses, see atface.h
{ 68 2
162 356
157 386
...
}

...

# frontal face detector records (optional) have a tag 81000000
81000000 image1      # frontal face detector record
{ 1 10          # a vector with 10 elements
291 367 218 218     # x, y, width, height
249 343 330 336     # xlefteye, ylefteye, xrighteye, yrighteye (optional)
290 436            # xmouthx, ymouth (optional)
                  # rot, yaw (optional, not included here)
}
81000000 image2
{ 1 10
244 274 221 221
199 244 292 255
99999 99999       # special value 99999 indicates mouth not found
}

...

```

Figure 5.1: *An example shapefile.*

Facedets are stored as vectors in the file, ordered as per the `DetPar` format defined in `misc.h`:

```

struct DetPar      // face and feature detector parameters
{
    double x, y;    // center of detector shape
    double width, height; // width and height of detector shape
    double lex, ley; // center of left eye (left and right are wrt the viewer)
    double rex, rey; // center of right eye
    double mouthx, mouthy; // center of mouth
    ...           // extra fields
}

```

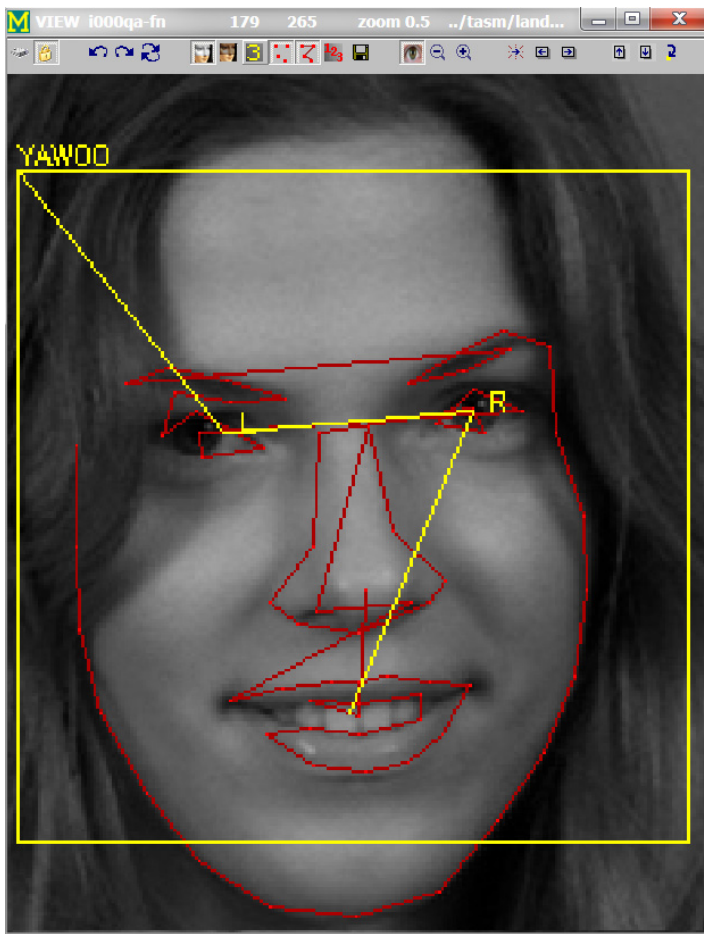


Figure 5.2: *Marki showing the detected face, eye, and mouth in the shapefile*

*Use `CtrlQ` within Marki to toggle display of these detector results.*

### 5.2.1 How to add facedets to a shapefile

This section tells you how to add facedets to a shapefile.

First if necessary incorporate your new face, mouth and eye detector XML files into the Stasm source code (Section 3.7.1).

Then run Tasm on the shapefile without facedets. Tasm will create a file `facedet.shape` in the `log` subdirectory (Section 5.2.2). Manually paste the face detector records from `facedet.shape` to the end of the shapefile (as was done for example in `muct68_nomirror.shape`).

Facedets that are deemed invalid by Tasm (Section 3.7.4) will not appear in `facedet.shape`.

Note that if we later modify the face detector then we will also of course have to update the facedets in the shapefile. This makes the shapefile consistent with the face detector used during the landmark search. If we don't update the shapefile, Tasm will generate a mean shape aligned incorrectly. This will cause a (perhaps subtle) degradation in fit quality.

Facedets for non-frontal detectors can also be stored in the shapefile. These are stored using tag 82000000 and similar, see `atface.h`. This is currently not used in the released version of Stasm. Pose data can also be stored, although once again this is currently not used in the released version of Stasm.



### 5.2.2 What triggers generation of `facedet.shape`?

In the current implementation, Tasm creates `facedet.shape` if less than 60 percent of the shapes in the shapefile have an associated `facedet` record. Otherwise, to save time Tasm does not generate `facedet.shape`.

The assumption is that if most of the shapes have a `facedet` record, then a `facedet` record is missing only because the face detector failed to find the face when the shapefile was first created, so don't re-search for it now. See the function `MustSearchImgIfFaceDetNotInShapefile`.

To force generation of `facedets` if necessary, manually remove the existing `facedets` from the shapefile before running Tasm.

### 5.2.3 Eyes and mouths in `facedet.shape`

When creating `facedet.shape`, Tasm will also invoke the eye and mouth detectors and include the eyes and mouths in the `facedet` records. But this only happens if `TASM_FACEDET_INCLUDES_EYES_AND_MOUTH` in `tasmconf.h` is set. Typically we would set this false before running Tasm if we are not working with faces (Section 4.7.3).

Tasm itself does not use the eye and mouth positions, but includes them in the shapefile for possible other uses.

A value of 99999 in an eye and mouth field (see the definition of `INVALID` in `misc.h`) means that the eye or mouth was not found.

# Chapter 6

## The landmark table

The landmark table is a C++ include file listing each landmark and its characteristics. See `landtab_muct68.h` for an example.

For each landmark we have the following information:

```
struct LANDMARK_INFO // landmark information
{
    int    partner;    // symmetrical partner point, -1 means no partner

    int    prev, next; // previous and next point
                // special val -1 means prev=current-1 and next=current+1

    double weight;    // weight of landmark relative to others (for shape mod)

    unsigned bits;    // used only during training (AT_Glasses, etc.)
};
```

The `partner` field is used to renumber landmarks when mirroring images (Chapter 10).

The `prev` and `next` fields set directions for classical 1D profiles. (See [2] Section 5.4.8 "Whisker Directions"). See these fields to -1 if you are not sure.

The `weight` field is used for discounting points in the shape model. (For example, in the standard Stasm 77-point model, the shape model doesn't use the forehead points suggested by the descriptor models because the forehead suggestions are unreliable.) See this field to 1 if you are not sure.

The `bits` fields use the definitions in `atface.h`. See the function `PointUsableForTraining` to see how Tasm uses these bits during training. See these bits to 0 if you are not sure.

# Chapter 7

## Images generated by Tasm

This section describes the images which Tasm generates for sanity checking. These images go into the `log` directory.

### 7.1 The landmark images

An image is produced for each landmark in the landmark table (Figure 7.1). Use these images to check the landmark table.

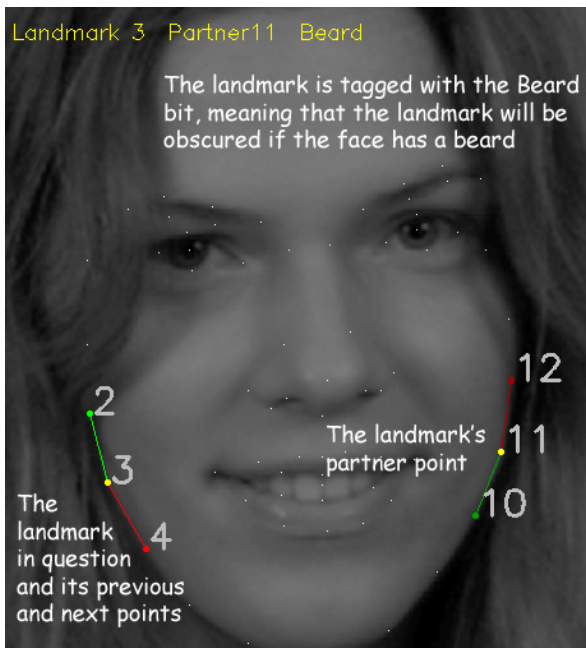


Figure 7.1: A landmark image generated by Tasm.

*This example is for landmark 3.*

*The previous, next, and partner point annotations are generated from point numbers in the landmark table.*

*If the landmark table has an incorrect entry it will usually be obvious in the image.*

## 7.2 The meanshape image

This shows the mean shape superimposed on the first face in the shapefile.

## 7.3 The shapeN images

These images are for checking the conversion of the shape to 17 points (in `shape17.cpp`) and to other numbers of points (in `convshape.cpp`). Each image shows the first face with the converted shape superimposed. Figure 3.3 is an example.

# Chapter 8

## Marki

Marki is a tool for manual landmarking. It is used to create and check shapefiles. Its zooming and other features facilitate accurate and fast landmarking. We have used it in house to manually landmark tens of thousands of images.

The Windows executable for Marki is supplied with the Stasm release. This is a 64-bit executable built with `OpenCV 2.4.0`, so make sure you have the `OpenCV 2.4.0` 64-bit DLLs on your path

Marki has not yet been ported to the Stasm 4 sources, and so the full source code for Marki is not yet supplied, although `marki.cpp` is supplied as a form of documentation.

Mouse clicks within Marki will move landmarks, so to prevent mishaps where someone inadvertently modifies a shapefile, change the shapefiles to read-only (after modifying the `Directories` strings for your environment):

```
chmod 444 ../tasm/shapes/*.shape
```

To look at the contents of a shapefile use the `-V` flag. The `-V` flag (for “view”) prevents anyone from modifying the shapefile by mistake:

```
marki -V ../tasm/shapes/muct68.shape
```

You can also use the provided Windows executable `cshapes.exe` for summarizing shapefiles:

```
cshapes ../tasm/shapes/muct68.shape
```

To automatically check that all images are accessible, use the `-C` flag:

```
marki -C ../tasm/shapes/muct68.shape
```

Use `marki -?` for a list of the other flags.

### 8.1 Using Marki

From within Marki click on the help button  to see what keystrokes are available. For example, `PageDn` or `Space` moves to the next image, `Shift-PageDn` skips 10 images

ahead, and **Home** takes us to the first image.

Cycle forwards and backwards through images in the shapefile with buttons, keystrokes, or the mouse wheel. You can also choose an image or landmark using the dialog window. (Marki searches for the next image name which contains the string we enter in the dialog window; we can enter a partial name.) The tag bits (**atface.h**) of the current shape are shown in the dialog window.

Change the position of a landmark with a left mouse click (Section 8.3). Most people use Marki with their left hand on the keyboard controlling zooming and their right hand on the mouse.

Save the modified landmarks to the shapefile using the **Save** button. Note by default that this will overwrite the original shapefile. If you don't want that, change the output name with the **-o** command line flag. When saving the landmarks, Marki makes a backup of the previous shapefile (Section 8.5).

On exit Marki will ask if you want to save the landmarks if you have not already done so.



The current settings are remembered so the next time Marki is used it starts from where it left off. Override the automatic return to the same landmark number with the **-l** command line flag. Or revert to all the defaults with the **-F** flag (for "fresh").


Marki uses the Windows registry entry **HKEY\_CURRENT\_USER/Software/Marki/Config**.

A note on the **-p** and **-P** flags. Be careful. Only the shapes that were read in are saved. If the **-p** or **-P** command line flags is used, this will be a subset of the shapes in the original shapefile.

## 8.2 Selecting the current landmark

Change the current landmark number with the dialog window or the **CtrlN** and **CtrlP** keys (N and P for next and previous).

You can also change the landmark number with buttons  . But to prevent mishaps, these buttons are displayed only if the **-B** command line flag is used (B for buttons).

Or click on the **AutoJumpToLandmark** button  to dynamically select the landmark nearest the mouse pointer. This is intended for easy touch-ups or corrections but is not efficient for mass landmarking. Right click twice to turn auto-jumping off. The **AutoJumpToLandmark** button is displayed only if the **-B**, **-H**, or **-V** command line flags are used.


You can also set the current landmark with the **-l** command line flag.

## 8.3 The mouse

**Left click** changes the position of the current landmark.

The current landmark is shown in red and the others in dull red. After the click, the new position is shown in cyan and the old in orange. Undo the click (and just about anything else) with the **Undo** button or **CtrlZ**.

Marki displays unused landmarks (coordinates 0,0) as green Xs, interpolating their positions from adjacent landmarks.

If the **AutoNextImage** button  is set, Marki will automatically move to the next image after the click. This is useful when we want to move “laterally”, setting just one landmark in each image. We have found that is the most efficient and consistent approach for mass landmarking.

**Right click** toggles **DarkImage**, and disables **AutoJumpToLandmark** if in effect.

**The wheel** moves backwards and forwards through the images.

## 8.4 Zooming

The centering  and zoom buttons  control zooming.

If centering is not enabled (the centering button is not pressed) the zoom setting is ignored, and we see the whole image.

If centering is enabled, the current landmark is shown in the center of the image with the current zoom setting. Zooming works off multiples of the eye-mouth distance. (The idea is that at a given zoom level the size of facial features remains roughly constant across all images.)

Marki has to estimate the eye-mouth distance if eye or mouthlandmarks are not available. Marki uses nearby landmarks to do this. If the necessary other landmarks are not available, Marki falls back to using one quarter the image size as the eye-mouth distance.

## 8.5 Backups

Things can easily go wrong when manually landmarking images. We suggest you make manual backups in addition to the automatic backups made by Marki described below.

When you save the shapefile, Marki also creates a backup shapefile. If the original file is `example.shape` the backup file will be `example_L99xDATE_TIME.bshape`, where

- 99 is the landmark number and `_` if no coordinates were changed by the user
- `x` is `_` if no other landmarks were changed and `x` if coordinates for other landmarks also changed.

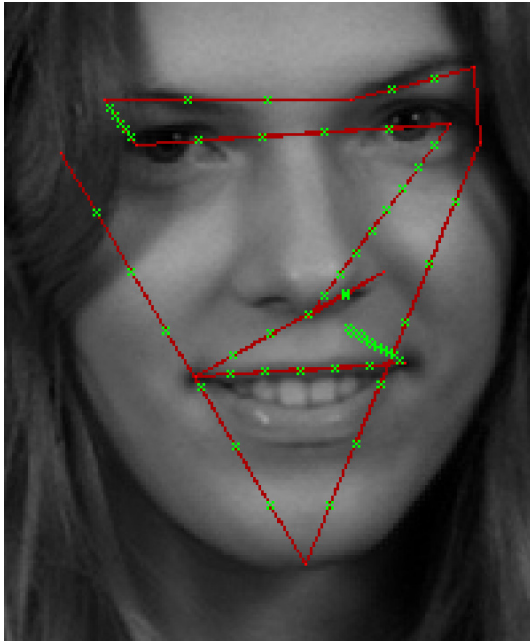


Figure 8.1: *An image with just a few points manually landmarked.*

*Marki shows the missing points as green Xs.*

The suffix `.bshape` is intended to make it easy to find or delete backup files. You will tend to collect a large number of them.

On exit, Marki copies `marki.log` to a backup log file using the same naming scheme as above but with the suffix `.blog`.

The `.bshape` and `.blog` backup files will be saved in the subdirectory `backup`, if that directory exists. If not, they will be saved in the working directory.

## 8.6 Preparing an initial shapefile for Marki

Marki requires a shapefile (Chapter 5). Typically we create a initial shapefile with dummy landmarks and then use Marki to correct the dummy landmarks.

If the rough positions of the landmarks are available then manual landmarking is easier if we use those positions as dummy landmarks. For faces, you may be able to generate the approximate positions of the landmarks with Stasm and manually paste the records from the Stasm log file into the initial shapefile.

One approach is to first (manually or otherwise) landmark just a few strategic landmarks, with the remaining landmarks marked as missing (coordinates 0,0). Marki will string the missing landmarks between these strategic landmarks (Figure 8.1). You can then get to work marking the missing landmarks, with the order of marking chosen to make marking of subsequent points easier.

When choosing the strategic landmarks, remember that it's best if the green X is near the correct position of the landmark. This allows zooming with both the X and the correct position simultaneously visible.



# Chapter 9

## Missing points and three-quarter faces

Tasm can build a model even if some shapes have missing points. Tasm imputes the missing points, and then proceeds as usual, building the models with the imputed points. (Points are labeled as missing during manual landmarking typically because they are obscured. For example, in Figure 9.1 the outer right eyebrow is around the side of the face. A missing point is given the special coordinates  $0,0$ .)

Missing points are typically common in three-quarter faces, where landmarks are obscured by the side of the face or nose. We could avoid this problem by using a different set of landmarks for three-quarter views, but typically we want to use the same set of landmarks for frontal and three-quarter faces.

To build a decent model, sufficient numbers of each landmark should be present. There should not be too many missing points. It is difficult to quantize “too many”. You will have to experiment with your data. The current implementation arbitrarily requires that each point must be valid in at least 33% of the shapes (`ShowPercentagePointsMissing`).

### 9.1 Example three-quarter model

This section presents an example three-quarter model built with the MUCT faces.

First modify the `stasm_lib.h` and `landmarks.h` for 68 point shapes as described in Section 3.6. Then invoke Tasm as

```
tasm muct68_nomirror.shape 0 0 [bc]
```

This builds a model with the MUCT `b` and `c` shapes (Figure 9.1). In the regular expression `[bc]`, the `b` selects the “partial-three-quarter” views and the `c` selects the three-quarter views.

Alternatively, use the full MUCT shapefile with mirrored shapes, but exclude the mirrored shapes when invoking Tasm:

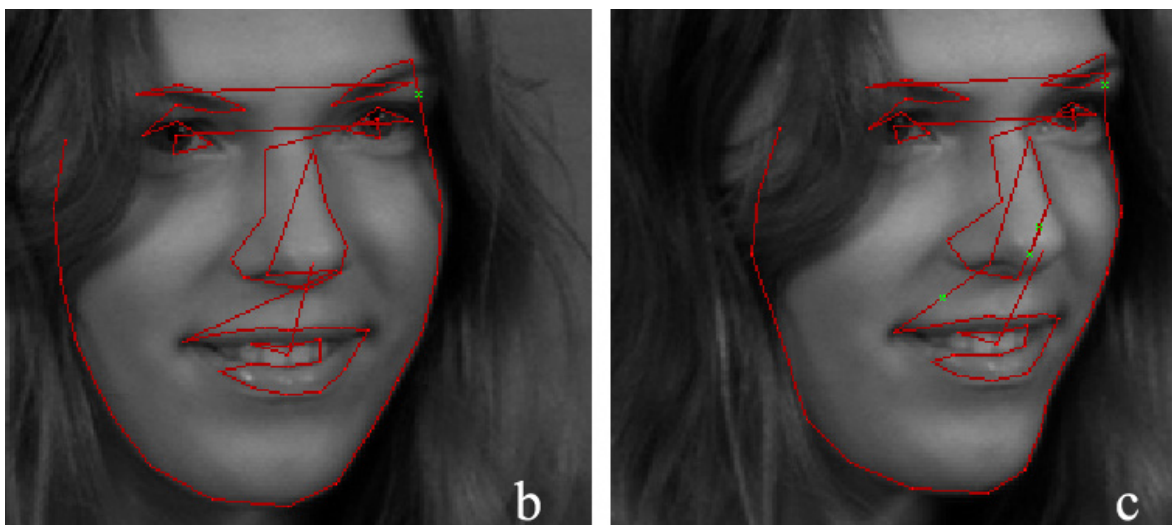


Figure 9.1: *Example b and c images from the MUCT set. Note the missing points. These are displayed by Marki as green crosses and positioned by Marki with simple linear interpolation between non-missing points.*

```
tasm muct68.shape 0 0 "i[~r].*[bc]"
```

The first part of the regular expression `i[~r]` excludes shapes beginning with `ir` (i.e. it excludes mirrored shapes). We certainly don't want to include the left facing three-quarter faces. (We could build a separate model for those later.)

The quotes surrounding the regular expression prevent the command shell from expanding the `*`. We need quotes when the regular expression includes characters that have special meaning to the shell, such as `*` or `|`. The shell strips the quotes so Tasm sees the regular expression without the quotes.

We include the partial-three-quarter faces in the above calls to Tasm because if we use just the three-quarter faces there would be too many missing points. If we use just the `c` shapes Tasm quits after issuing the message

```
Point 15 is unused in 74% of the shapes
```

After running Tasm we can check that the mean shape is plausible by examining `log/meanshape.bmp`.

## 9.2 The `imputed.shape` file

Tasm generates `imputed.shape` with the imputed points. This file is for manual sanity checking of the imputed points. For example:

```
marki -V tasmout/log/imputed.shape
```



Figure 9.2: *Same as the previous figure, but with points imputed by Tasm.*

Some imputed shapes are shown in Figure 9.2. The outermost right hand eyebrow in the c subfigure in Figure 9.2 has been imputed too far to the right. This is understandable, because Tasm builds the imputation model predominantly from the b shapes (because this point is present in nearly all b shapes and missing in nearly all the c shapes). The imputation model is a 2D model, and does not understand 3D concepts like rotation. The shifted nearby points in the c shape in the figure cause this imputation model to position the eyebrow point too far to the right. (In practice this seems to have at most just a very small harmful effect on the overall fit.)

Tasm also displays a table showing the percentage of missing points at each landmark position:

661 (44%) of 1493 shapes have all 68 points										
Percentage of points unused over all shapes:										
	0	1	2	3	4	5	6	7	8	9
0	.	.	.	.	.	.	.	.	.	.
10	.	.	.	.	.	45.81	0.47	.	.	.
20	0.27	.	.	.	.	.	.	.	.	.
30	.	.	.	.	.	.	.	.	.	0.07
40	.	.	19.69	28.40	29.07	25.32	.	19.29	.	.
50	.	.	.	.	.	.	.	.	.	.
60	.	.	.	.	.	.	.	.	.	.

A dot indicates that no points are missing for the landmark in question. In this table we see that points are missing on the outer right eyebrow and on the far side of the nose. For example, 45.81% of the shapes are missing point 15 (the outer right eyebrow). Conversely, this landmark is valid in about 54% of the shapes, which is easily sufficient to build a shape model.

## 9.3 Issues with the above example

There are a few issues with the model we built in Section 9.1 above.

1. Tasm used the OpenCV frontal face detector and issued the message

```
1191 faces detected (79.77%)
```

That's a low percentage, indicating that the face detector is not up to the task of locating the three-quarter faces in the training data. We should really use a more capable face detector that reliably detects three-quarter views, or use a face detector optimized for three-quarter views.

2. The default settings in `eyaw` and `estart` settings in `initasm.cpp` (Section 3.7.2) are good for frontal models but should be changed for this three-quarter model. The start shape will be better positioned and thus the overall search results will be better if we initialize the three-quarter model with `eyaw=EYAW22` and `estart=ETART_EYE_AND_MOUTH`.

- (a) The `EYAW22` setting says that Stasm should search for the eyes and mouth in search areas appropriate for right three-quarter faces. Figure 3.1 showed the search rectangles for frontal faces. To see a similar image for your model, run Stasm after building with `SHOW_IMAGES` set.

You should tweak the hard-wired constants which define the search regions in `startshape.cpp` and `eyedet.cpp`. The current values are for an in-house three-quarter face detector which probably frames the face slightly differently from your detector.

The `22` indicates that the faces have a nominal yaw of 22 degrees. For more strongly yawed faces, use `YAW45`. For *left* three-quarter faces, use `EYAW_22`, note the underscore.

- (b) The `ETART_EYE_AND_MOUTH` setting says that Stasm should also search for the mouth and use it to help position the start shape. In our experience with three-quarter views, this tends to give better results (Figure 9.3).
3. We need to tune constants such as `neigs` and `bmax`. To do so, we should set aside some images for tuning and final testing.

## 9.4 The imputation algorithm

As mentioned above, Tasm imputes the missing points and then builds the models with the imputed shapes. We don't claim that the imputation algorithm and implementation are optimal, but have been used successfully to build multi-view models (e.g. [3]).

Tasm imputes points by first building a shape model using points that are available and ignoring points that are missing. It then imputes the missing points using this shape model. It does this by conforming the shape to the shape model: it pins the positions

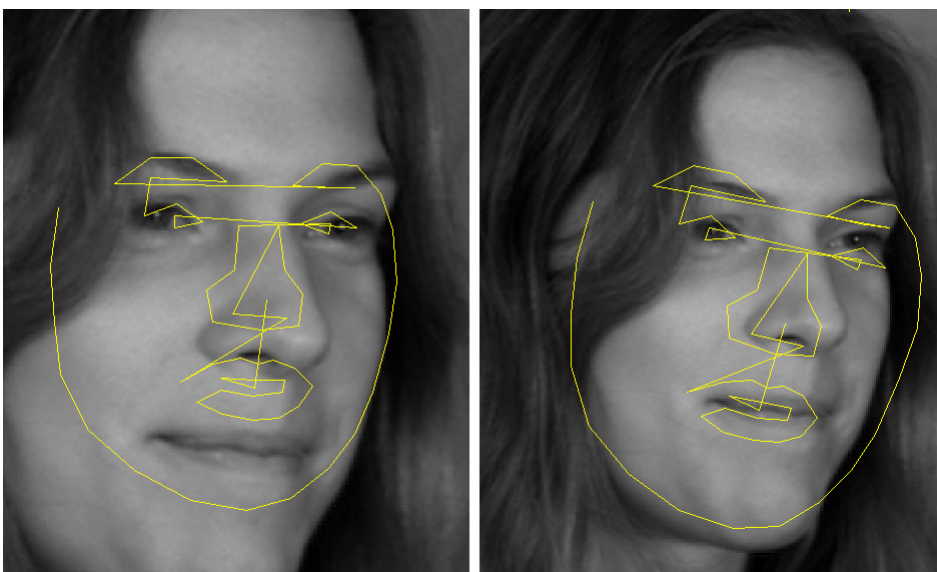


Figure 9.3: *Start shapes with a three-quarter model.*

**Left** Start shape initialized from just the eye positions (`ESTART_EYES`).

**Right** Better start shape from both the eyes and mouth (`ESTART_EYE_AND_MOUTH`).

*This is a fairly extreme example. With other images the discrepancy will not be as big.*

of the non-imputed points and uses the model (with a limited number of eigenvectors) to predict the positions of the missing points. It iterates until the points are stable. See `tasmimpute.cpp`.

Once this is done there is optional further processing which is controlled by configuration constants in `tasmconf.h`. With the default settings of these constants, this further processing does *not* occur. It is not clear at this time how these constants should best be defined. We suggest you leave them disabled initially and then experiment with different settings. The optional further processing is as follows:

(i) If `TASM_REBUILD_SUBSET` is specified, after imputing the shapes Tasm rebuilds the shape model using the specified subset of imputed shapes. For example, in the above example after using the `b` and `c` shapes to build the initial shape model, we can specify that just the `c` shapes should be used to build the final shape model. To do this, set `TASM_REBUILD_SUBSET = "c"`. See the comments in `tasmconf.h`.

(ii) If `TASM_SUBSET_DESC_MODS` is set `true`, Tasm uses only the above subset to build the descriptor models.

# Chapter 10

## Mirroring shapes

If our objects are symmetrical (e.g. frontal faces), we can effectively double the number of training images by mirroring the existing images. We will need to add the mirrored shapes to the shapefile. The `shapemirror` utility described below can be used to generate the mirrored images and shapes.

By convention in Tasm the second character of the filename of mirrored images is `r` for “reversed”. Thus `b0000.jpg` becomes `br0000.jpg`. This convention allows easy inclusion or exclusion of mirrored shapes using regular expressions on the Tasm command line.

### 10.1 The `shapemirror` utility

The `shapemirror` utility mirrors the shapes in a shapefile and also writes the mirrored images to the `mirrored` subdirectory. Use it like this

```
shapemirror ../tasm/shapes/muct68_nomirror.shape
```

A message like

```
NELEMS(LANDMARK_INFO_TAB) 77 != number of points 68 in the shapefile
```

means we need to update the `landmarks.h` file for the landmark table appropriate for the shapefile (Section 3.6). So in the current example, in `landmarks.h` change

```
#include "landtab_muct77.h"
```

to

```
#include "../tasm/landtab/landtab_muct68.h"
```

After rebuilding the executables with the new landmark table, we can check the table by running Tasm and examining the images in the `log` directory (Section 7.1):

```
tasm ../tasm/shapes/muct68_nomirror.shape 3
```

The 3 in the above command line is optional. It limits Tasm to just the first three shapes, for speed, because all we need here are the landmark images in the log directory, and we don't need to waste time building a full model. In the landmark images in the Tasm log directory, our only concern for this exercise is to check the partner point for each landmark. The partner point is the point's number after it has been mirrored.

Now run `shapemirror`:

```
shapemirror ../tasm/shapes/muct68_nomirror.shape
```

which will print the following

```
>shapemirror ../tasm/shapes/muct68_nomirror.shape
mkdir mirrored
Reading ../tasm/shapes/muct68_nomirror.shape: 3755 shapes
Generating muct68_nomirror_r.shape and mirrored/*r.jpg
0.0% 0 /faces/muct/jpg/i000qa-fn.jpg
0.0% 1 /faces/muct/jpg/i000qb-fn.jpg
0.1% 2 /faces/muct/jpg/i000qc-fn.jpg
0.1% 3 /faces/muct/jpg/i000qd-fn.jpg
...
100.0% 3754 /faces/muct/jpg/i624ze-fn.jpg
Ignoring facedet records (run Tasm to generate the facedets)
Processing pose data: 0 records
```

This creates a new shapefile with `_r` appended to the original name (we can use Marki to verify the file's contents). The mirrored images go into the `mirrored` subdirectory. Typically we would now move the images from the `mirrored` subdirectory to their final destination directory.

Note that `shapemirror` does not mirror the facedet records (Section 3.9.2). This is because face detectors typically return not-exactly-mirrored results on mirrored images. To get the facedet records, run Tasm on the mirrored shapefile to generate `facedet.shape` (Section 5.2.1):

```
tasm muct68_nomirror_r.shape
```

Create a combined shapefile by manually pasting (i) the original shapefile, (ii) the shapes from the mirrored shapefile, (iii) the facedets from the original shapefile, and (iv) the facedets from `facedet.shape` for the mirrored shapefile. Finally, manually edit the `Directories` string in this combined shapefile to include the mirrored image directory.

Check the results with Marki:

```
marki -V combined.shape
```

Within Marki, use `CtrlQ` to display the face detector records. Enter just `r` in the dialog window to search for the first mirrored shape (i.e. the first image with `r` in its name).

# Chapter 11

## FAQs and error messages

### **11.1** `NELEMS(LANDMARK_INFO_TAB) 77` does not match the number of points 68 in `file.shape`

You built Tasm with a landmark table with 77 points, but invoked Tasm with a shapefile with 68 points. Probably you need to change `landmark.h` to use a 68 point landmark table (Section 3.6).

### **11.2** The application was unable to start correctly (0xc000007b)

You are probably on Windows system and your 32-bit application is trying to use a 64-bit DLL, or vice versa. Check that the correct DLLs are on your DLL path, or copy the correct DLLs into the current directory. See `copydlls.bat` (you will have to tweak the OpenCV directory specified in that batch file).

### **11.3** Warning: Only plain strings (not regexs) are supported

The executable you are using does not support regular expressions. See the note on page 17. Stasm will do a plain substring match, not a regex match.

### **11.4** How to rebuild the models for the released version of Stasm?

You can't, because the released version of Stasm was built with a 77-point shapefile `muct_77.shape` which for historical reasons was created on cropped versions of the MUCT images. The cropped images are not yet prepared for release.

Note that Stasm 4.1.0 gives slightly different results from Stasm 4.0.0, which used identical training data but slightly different versions of Tasm and the start shape code.



# Bibliography

- [1] T. F. Cootes and C. J. Taylor. *Technical Report: Statistical Models of Appearance for Computer Vision*. The University of Manchester School of Medicine, 2004. [http://www.isbe.man.ac.uk/~bim/Models/app\\_models.pdf](http://www.isbe.man.ac.uk/~bim/Models/app_models.pdf).
- [2] S. Milborrow. *Locating Facial Features with Active Shape Models*. Master's Thesis. University of Cape Town (Department of Image Processing), 2007. <http://www.milbo.users.sonic.net/stasm>.
- [3] S. Milborrow, Tom E. Bishop, and F. Nicolls. *Multiview Active Shape Models with SIFT Descriptors for the 300-W Face Landmark Challenge*. ICCV, 2013.
- [4] S. Milborrow, J. Morkel, and F. Nicolls. *The MUCT Landmarked Face Database*. Pattern Recognition Association of South Africa, 2010.
- [5] S. Milborrow and F. Nicolls. *Active Shape Models with SIFT Descriptors and MARS*. VISAPP, 2014.